Partial Exam
Compilation and Program Analysis (CAP)
November, 8th, 2017
Duration: 2 Hours

Instructions :

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

Exercise #1 ▶ Attribution with visitors (15 min)

We consider ordered lists (`<ol>`) in html files :

`<html> ... <ol> ... <li> ... </li> </ol> ... </html> EOF`

where ... are texts that are *ignored* and list elements are tagged with `li`s (`<li>` denotes the beginning of a list element and `</li>` its end). The grammar is depicted in Figure 1. Nested lists are authorized, and the content of an li element can be a list.

```
grammar Html;

prog: '<html>' listo '</html>' EOF;

listo:
        '<ol>' listi  '</ol>' listo   #listorec
    |                                 #listobase
;

listi : onei                #listibase
    | onei listi            #listirec
    ;

onei : '<li>' listo '</li>'         #listiel
    ;

// [plus some other rules to skip text between tags]
WS  :   (' '|'\t'|'\n')+  -> skip;
```

FIGURE 1 – HTML grammar

Question #1.1
Give a non trivial exemple of a file that belongs to the language.

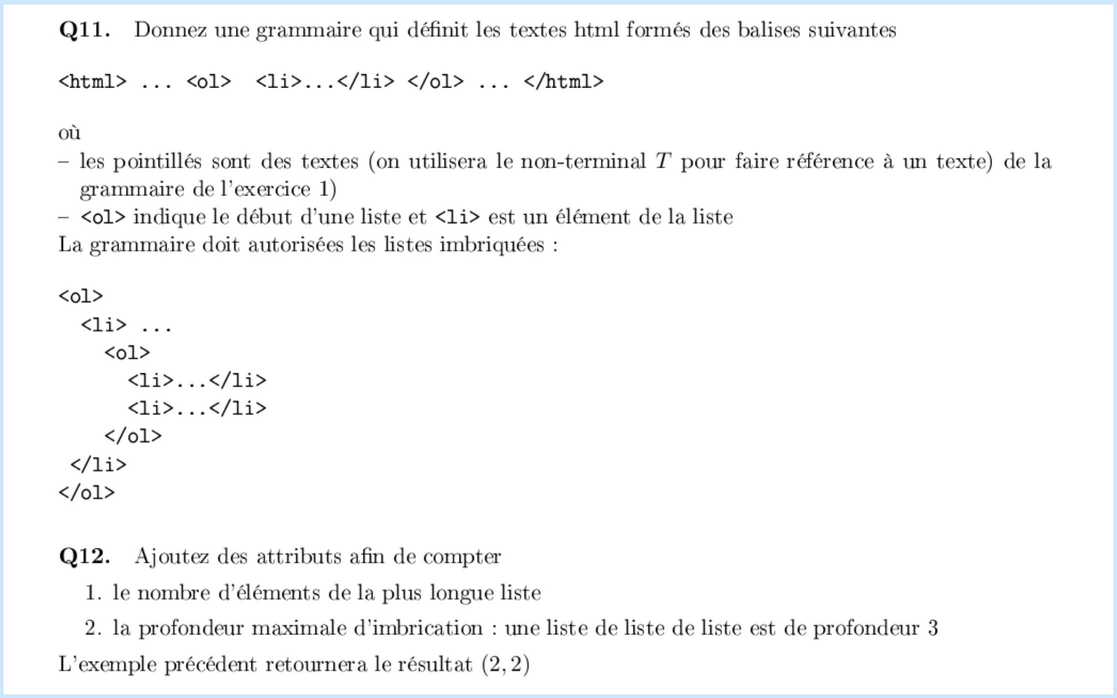
Question #1.2
Give an attribution that permits to count the number of elements in the longest list of the file (pseudo code)

Question #1.3
The maximal list depth : a list of list of list is of depth 3.

Question #1.4
Implement these two attributions as a visitor file (Fill Appendix A) that would be called with :

```
visitor = MyHtmlVisitor()
nbelmax, profmax = visitor.visit(prog)
print("nb␣max␣="+str(nbelmax)+";␣profmax="+str(profmax))
```

Solution:



Q11. Donnez une grammaire qui définit les textes html formés des balises suivantes

`<html> ... <ol>  <li>...</li> </ol> ... </html>`

où
– les pointillés sont des textes (on utilisera le non-terminal T pour faire référence à un texte) de la grammaire de l'exercice 1)
– `<ol>` indique le début d'une liste et `<li>` est un élément de la liste
La grammaire doit autorisées les listes imbriquées :

```
<ol>
  <li> ...
    <ol>
      <li>...</li>
      <li>...</li>
    </ol>
 </li>
</ol>
```

Q12. Ajoutez des attributs afin de compter

1. le nombre d'éléments de la plus longue liste
2. la profondeur maximale d'imbrication : une liste de liste de liste est de profondeur 3

L'exemple précédent retournera le résultat $(2, 2)$

Solution code dans le repertoire `HtmlVisitor` :

```
from HtmlVisitor import HtmlVisitor


class MyHtmlVisitor(HtmlVisitor):

    def visitProg(self, ctx):
        return(self.visit(ctx.listo()))

    def visitListorec(self, ctx):
        el1, pr1 = self.visit(ctx.listi())
        el2, pr2 = self.visit(ctx.listo())
        return (max(el1, el2), max(pr1+1, pr2))

    def visitListobase(self, ctx):
        return (0, 0)

    def visitListibase(self, ctx):
        return self.visit(ctx.onei())

    def visitListirec(self, ctx):
        nbi, pli = self.visit(ctx.onei())
        nbell, pl = self.visit(ctx.listi())
        return (max(nbi, nbell+1), max(pli, pl))

    def visitListiel(self, ctx):
        el, pr = self.visit(ctx.listo())
        return (1, pr)
```

Exercise #2 ▶ **Program equivalence (10 min)**

Question #2.1

Express program equivalence for the natural semantics of mini-while.

Question #2.2

Prove the equivalence of the following two programs :

```
P1 : if (b1 and b2) then S1 else S2;
```

and

```
P2 : if b1 then (if b2 then S1 else S2);
```

Please be precise in your justifications (use semantic rules and clean semantic proof trees).

Exercise #3 ▶ **Mini-While : typing + code generation (20 min)**

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

```
var x2: int;
x2 := 13;
while (x2 > 8) do
   x2 := x2 - 1;
done
```

Question #3.1

Show that this program is well-typed (declarations, statements). If some rules are missing in the companion file, invent them and report them on your sheet.

Solution:
Flemme du correcteur

Question #3.2

Generate the LEIA 3-address code[1] for the given program according to the code generation rules. *Recursive calls, auxiliary temporaries, code, must be separated and clearly described.* .

Question #3.3

Replace the conditional JUMP by a regular sequence of LEIA code with a `snif` (with temporaries).

Solution: flemme

1. We recall that the LEIA three address code has the same instruction set as the LEIA regular code except for conditions which use the idiom `condJUMP(label,t1,condition,t2)` and temporaries/virtual registers instead of regular registers). This code can use the `.LET` preprocessing command if you want.

Exercise #4 ▶ Variable initialisation with typing (20min)

Adapted from a compilation exam from J-C. Filliâtre, ENS Paris.

In Java, a local variable should be initialized (by a value) before being used. In this exercise, we propose to statically check this condition using Typing. For this purpose, we consider the same while language as in the course.

We also suppose that there is no double variable declaration and all programs are "regularly typed" with the typing rules of the course.

Question #4.1

What could be the interest for the compiler to know that a given variable is initialised before being used ?

Solution: For instance, he avoids execution crashes due to non correctly initialised variables (in C). Or in Java, it avoid filling memory with default values when using contructors.

Question #4.2

Give an example of a well-typed program and a non well-typed program for the variable initialisation condition.

Solution:

```
var x,y:int;
x := 2;
y := 42 + x;
```

is well-typed, but not :

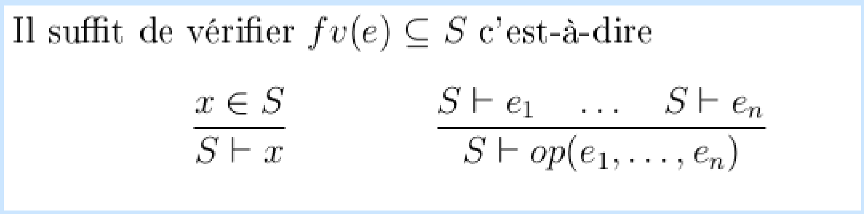
```
var x,y:int;
y := 42 + x;
```

Let S be a variable set. Given an expression e, $S \vdash e$ denotes the judgment "expression e only uses variables that appear in S".

Question #4.3

Give inference rules for $S \vdash e$, defined by induction on the abstract syntax of *numerical* expressions.

Solution: Adapt these to the language we have.



Il suffit de vérifier $fv(e) \subseteq S$ c'est-à-dire

$$\frac{x \in S}{S \vdash x} \qquad \frac{S \vdash e_1 \quad \dots \quad S \vdash e_n}{S \vdash op(e_1, \dots, e_n)}$$

To verify that a given instruction *stm* is correct, let us define $S \vdash smt \rightarrow S'$ which means "supposing that all variables in S are already initialised, the execution of *stm* only accesses initialised variables, and at the end of the execution, all variables in S' are initialised, with $S \subseteq S'$".

Question #4.4

Give the 5 inference rules for this typing judgment. Apply on your two examples.

Solution:

:

$$\frac{S \vdash e}{S \vdash x{=}e; \to S \cup \{x\}} \qquad \frac{}{S \vdash \texttt{int}\ x\texttt{;} \to S}$$

$$\frac{S \vdash e \quad S \vdash s_1 \to S_1 \quad S \vdash s_2 \to S_2}{S \vdash \texttt{if(}\ e\ \texttt{)}\ s_1\ \texttt{else}\ s_2 \to S_1 \cap S_2}$$

$$\frac{S \vdash e \quad S \vdash s \to S'}{S \vdash \texttt{while(}\ e\ \texttt{)}\ s \to S} \qquad \frac{S \vdash s_1 \to S_1 \quad S_1 \vdash s_2 \to S_2}{S \vdash s_1\ \texttt{;}\ s_2 \to S_2}$$

Exercise #5 ▶ A new mini-while construction (15 min)

We augment the mini-while language with "guarded commands" of the form : "`case F esac`" with F a (non empty) list of elements of the kind :"$[e] \to S$;" : Here is an example :

```
case  [x=1] ->  x:=x+1 ;
      [y=3] ->  y:=2 ;
esac
```

the conditions must evaluate to a boolean, and the command which is executed is the first command whose associated condition evaluates to true. If no conditions are true, then the case behave like a skip.

This new command is well-typed if for each sublist the e expr is of type bool, and S is of type `void`.

Question #5.1

Augment the mini while abstract syntax (use regular grammar definition idioms : + and * are forbidden).

Solution:

$$\begin{array}{lcll} smt & ::= & x := e & \text{assign} \\ & | & \dots & \\ & | & \texttt{case}\ condstmtlist\ \texttt{esac} & \text{case} \end{array}$$

where :

$$\begin{array}{lcl} condsmtlist & ::= & (bexpr, smt)\ condsmtlist \\ & | & (bexpr, smt) \end{array}$$

Question #5.2

Give new typing rules for this construction.

Solution: todo

Question #5.3

Give new natural operational rules for this construction.

Question #5.4

Give new 3 address code generation rules for this construction.

Question #5.5

Illustrate the previous solutions on the example within an appropriate environnement. The details of the rules used in the process of code generation is not mandatory.

Exercise #6 ▶ Operational semantics, a debugger (40min)

Adapted from a compilation exam, Grenoble, 2008

We consider the following abstract grammar for a (low level) imperative language :

$$\begin{array}{rcll} S(Smt) & ::= & x := e & \textit{assign} \\ & | & \texttt{if } b \texttt{ goto } \ell & \textit{test} \\ & | & \texttt{goto } \ell & \textit{goto} \end{array}$$

The configurations are triples (pc, C, σ) where pc is the program counter, C is the code (a sequence of statements labelled by $\mathbb{N}$), and σ is the memory state ($State = Val \to \mathbb{Z}$). The semantics of arithmetic expressions as well as boolean expressions is the natural semantics seen in the course.

Statement Semantics We denote by $\triangleright$ the small step semantics of this language :

- $(pc, C, \sigma) \triangleright (pc + 1, C, \sigma[x \mapsto \mathcal{A}[a]\sigma])$ if $C(pc) = x := a$.
- $(pc, C, \sigma) \triangleright (\ell, C, \sigma)$ if $C(pc) = \texttt{goto } \ell$.
- $(pc, C, \sigma) \triangleright (pc + 1, C, \sigma)$ if $C(pc) = \texttt{if } b \texttt{ goto } \ell$ and $\mathcal{B}[b]\sigma = false$.
- $(pc, C, \sigma) \triangleright (\ell, C, \sigma)$ if $C(pc) = \texttt{if } b \texttt{ goto } \ell$ and $\mathcal{B}[b]\sigma = true$.

The evaluation of a program is done from the initial configuration with an empty memory : $(0, C, [])$.

Question #6.1

Compute the semantics ($\triangleright$) of the following program :

```
1:x=8;
2:y:=12;
3:if x<y goto 5;
4:x:=y-x;
5:x:=42;
```

Solution: We denote C the above code and supposed that the program starts in an empty memory with $pc = 0$

$$(0, C, []) \triangleright (1, C, [x \mapsto 8]) \triangleright (3, C, [x \mapsto 8, y \mapsto 12]) \triangleright (5, C, [x \mapsto 8, y \mapsto 12]) \triangleright (6, C, [x \mapsto 42, y \mapsto 12])$$

Debugger Now we consider a given program and its interaction with a *debugger*, which provides the following *commands* :

— `setbrk`(n) : sets a breakpoint at statement labeled by n : where the program counter is equal to n, the execution stops.
— `rmbrk`(n) : remove the breakpoint set at statement labeled by n.
— `step` : performs a single execution step.
— `cont` : resumes program execution up to the next breakpoint.
— `set`(x, v) : sets value v to variable x.
— `print`(x) : prints the value of variable x.
— `exit` : exits from the debugger.

Now a given configuration is composed of : a program configuration (pc, C, σ), a set of breakpoints β and optionally a debugger command. An initial configuration is of the form $((1, C, \sigma), \emptyset, \texttt{cmd})$. We denote by $\rightarrow$ the transition relation $((pc, C, \sigma), \beta, \texttt{cmd}) \rightarrow ((pc', C, \sigma'), \beta)$.

The following program :

```
1: x:=8
2: y:=12
3: if x=y goto 9
4: if x<y goto 7
5: x:=x-y
6: goto 3
7: x:=y-x
8: goto 3
9:
```

was designed to compute the *gcd* of the two variables x and y, but it has a bug.

On this program named C, we give the expected steps of our semantics :

— $((1, C, [x \mapsto 0, y \mapsto 0]), \emptyset, \texttt{setbrk}(3))$ produces configuration $((1, C, [x \mapsto 0, y \mapsto 0]), \{3\})$.
— Configuration $((3, C, [x \mapsto 8, y \mapsto 12]), \{3\}, \texttt{cont})$ produces configuration $((3, C, [x \mapsto 4, y \mapsto 12]), \{3\})$.

Question #6.2

Give the operational semantic rules ($\rightarrow$)for `setbrk(n)` and `rmbrk(n)` and show their effect with appropriate examples similar to the previous ones. *Your semantic rules will be be written in a* Natural Semantics style *similar to the one described in the course. Please explain and justify them.*

Solution: $((pc, C, \sigma), \beta, \texttt{setbrk}(n)) \rightarrow ((pc, C, \sigma), \{n\} \cup \beta)$ and $((pc, C, \sigma), \{n\} \cup \beta, \texttt{rmbrk}(n)) \rightarrow ((pc, C, \sigma), \beta)$.
Configuration $((3, C, [x \mapsto 8, y \mapsto 12]), \{3\}, \texttt{setbrk}(6))$ produces configuration $((3, C, [x \mapsto 8, y \mapsto 12]), \{3, 6\})$.
Configuration $((3, C, [x \mapsto 8, y \mapsto 12]), \{3, 6\}, \texttt{rmbrk}(6))$ produces configuration $((3, C, [x \mapsto 8, y \mapsto 12]), \{3\})$.

Question #6.3

Same questions for `set(x,v)` and `print(x)`.

Solution: $((pc, C, \sigma), \beta, \texttt{set(}x\texttt{,}v\texttt{)}) \rightarrow ((pc, C, \sigma[x \mapsto v]), \beta)$ and $((pc, C, \sigma), \beta, \texttt{print(}x\texttt{)}) \rightarrow ((pc, C, \sigma, \beta)$.

Configuration $((3, C, [x \mapsto 8, y \mapsto 12]), \{3\}, \texttt{set(}x\texttt{,}2)$ produces configuration $((3, C, [x \mapsto 2, y \mapsto 12]), \{3\})$.

Configuration $((3, C, [x \mapsto 8, y \mapsto 12]), \{3, 6\}, \texttt{print(}x\texttt{)})$ produces configuration $((3, C, [x \mapsto 8, y \mapsto 12]), \{3\})$ and prints 8.

Question #6.4

Same questions for `step`.

Solution: $((pc, C, \sigma), \beta, \texttt{step}) \rightarrow ((pc', C, \sigma'), \beta)$ where $(pc, C, \sigma) \rhd (pc', C, \sigma')$

Configuration $((3, C, [x \mapsto 8, y \mapsto 12]), \{3\}, \texttt{step})$ produces configuration $((4, C, [x \mapsto 2, y \mapsto 12]), \{3\})$.

Question #6.5

Same questions for `cont`.

Solution: There are two rules, a base one if the next control point is a breakpoint, and a recursive one if this is not the case. The recursive one makes one step in the semantics of the program, and re-executes the debugger from it. $((pc, C, \sigma), \beta, \texttt{cont}) \rightarrow ((pc', C, \sigma'), \beta)$ if $pc' \in \beta$ and $(pc, C, \sigma) \rhd (pc', C, \sigma')$.

$$\frac{(pc, C, \sigma) \rhd (pc'', C, \sigma''), \quad , ((pc'', C, \sigma'')), \beta, \texttt{cont}) \rightarrow ((pc', C, \sigma'), \beta)}{((pc, C, \sigma), \beta, \texttt{cont}) \rightarrow ((pc', C, \sigma'), \beta')} \text{ if } pc'' \notin \beta.$$

Configuration $((1, C, []), \{3\}, \texttt{cont})$ produces configuration $((3, C, [x \mapsto 8, y \mapsto 12]), \{3\})$.

Now we define a debugging sequence as a sequence of configurations :

$$((1, C, \sigma), \emptyset, \texttt{cmd}), (pc_1, C, \sigma_1), \beta_1, \texttt{cmd}_1) \dots ((pc_i, C, \sigma_k), \beta_k, \texttt{cmd}_k)$$

such that each step is conform to the debugging semantics ($\rightarrow$) :

$$((pc_i, C, \sigma_i), \beta_i, \texttt{cmd}_i) \rightarrow ((pc_{i+1}, C, \sigma_{i+1}), \beta_{i+1})$$

A maximal debugging sequence is of the form :

$$((1, C, \sigma), \emptyset, \texttt{cmd}), (pc_1, C, \sigma_1), \gamma_1, \texttt{cmd}_1) \dots ((pc_n, C, \sigma_n), \beta_n, \texttt{exit})$$

Question #6.6

Give the debugging sequence obtained by applying the following debugging commands on the example program :

```
setbrk(3), cont, cont, cont, cont
```

Solution:

$((1, C, \sigma), \emptyset, \texttt{setbrk(3)}),$
$((1, C, \sigma), \{3\}, \texttt{cont}),$
$((3, C, [x \mapsto 8, y \mapsto 12]), \{3\}, \texttt{cont}),$
$((3, C, [x \mapsto 4, y \mapsto 12]), \{3\}, \texttt{cont}),$
$((3, C, [x \mapsto 8, y \mapsto 12]), \{3\}, \texttt{cont}),$
$((3, C, [x \mapsto 4, y \mapsto 12]), \{3\}, \texttt{cont})$

Question #6.7

Same question with :

`setbrk(3), step, step.`

Solution: $((1, C, \sigma), \emptyset, \texttt{setbrk(3)}), ((2, C, \sigma), \{3\}, \texttt{step}), ((3, C, [x \mapsto 8]), \{3\}, \texttt{step})$

Question #6.8

Where is the bug ?

Solution: line 7 should be $y := y - x$ instead of $x := y - x$.

A Visitor

```
from HtmlVisitor import HtmlVisitor

class MyHtmlVisitor(HtmlVisitor):

    def visitProg(self, ctx):




    def visitListorec(self, ctx):




    def visitListobase(self, ctx):



    def visitListibase(self, ctx):




    def visitListirec(self, ctx):




    def visitListiel(self, ctx):




#
```