Partial Exam
Compilation and Program Analysis (CAP)
November, 8th, 2018
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.

Solution: In blue, correction remarks and not fully redacted answers.

Exercise #1 ▶ **A grammar attribution (20 min)**

Inspiration : Compiler exam, Grenoble, by S. Boulmé
Let us consider the grammar of prefix expressions :

$$\begin{array}{lcll} expr & ::= & id & variable \\ & | & + \; expr \; expr & plus \\ & | & - \; expr \; expr & minus \end{array}$$

and infix expressions with parenthesis :

$$\begin{array}{lcll} expr & ::= & id & variable \\ & | & expr + expr & plus \\ & | & expr - expr & minus \\ & | & (expr) & parenthesis \end{array}$$

We consider the translation of a prefix expression into an infix one : $+ab$ should be transformed into $a + b$.

Question #1.1

Define on the prefix grammar a naive **synthetised (from leaves to node)** attribution (types, computation rules) that computes this translation. Give an example where your translation give a non optimal number of parenthesis.

In the sequel, we will implement the idea that to avoid parenthesis under a minus operation, we inverse signs :

$$a - (b + c) = a - b - c \qquad a - (b - c) = a - b + c$$

Question #1.2

Define on the prefix grammar an attribution that inherits a boolean b and computes an infix expression e such that :

— if b is **false** then e has the same semantics as the analyzed chain.

— if b is **true** then by exchanging $+$ and $-$ symbols in e, we get an expression of the same semantics as the analyzed chain.

For instance, if b is true, for the prefix expression $+a - bc$ we would get $a - b + c$.

Question #1.3

What should be the initial value of b at the root ? Explain (draw the AST of the expression !) what would be the propagation of information on the initial chain $--a+-cd+ef-+gh-ij$.

Solution:

— be careful to write an attribution

— same

— initial value = false. Be careful an AST is not a derivation tree.

let us denote my att the attribution

```
att(e:expr) returns infixepr inherit[bool b]
x:id -> x
+ x y -> x.b = b, y.b = b;
       if b then  returns att(x) - att(y)
         else returns att(x)+att(y)
- x y -> x.b = b y.b = not(b);
      if b then returns att(x) + att (y)
         else returns att(x)-att(y)
```

examples : if b is true :

— $(+ab)$ should be equivalent to $a - b$

— for minus : $(-ab)$ should be equivalent to $a+b$. The right operand should be evaluating with a flipped b.

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

Question #2.1

Express program equivalence for the natural (big-steps) semantics of mini-while.

Question #2.2

Prove the equivalence of the following two programs :

`P1 : while e do C;`

and

`P2 : if e then (C ; while e do C) else skip;`

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

Solution: Program eq : P1 equiv P2 for all initial memory state σ, $(P_1, \sigma) \to \sigma' \Leftrightarrow (P_2, \sigma) \to \sigma'$). (and no derivation equiv no derivation)

A clean proof is by double implication : Suppose we have $(P_1, \sigma) \to \sigma'$, there are two cases : if $B(e)[\sigma] = ff$, then necessarily (thanks to the false while rule), $\sigma = \sigma'$, thus (insert a proof tree here) is a proof tree for $(P_2, \sigma) \to \sigma'$...

In particular, it is not sufficient to show two proof trees and say that they are equivalent.

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

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

```
var x,y: int;
x := 12;
```

```
y := 4;
if (y > x) do
   x := x - y;
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: Be careful to apply the declaration rules to get. $\Gamma = \{x \mapsto int, y \mapsto int\}$. And then a proof tree.

Question #3.2

Generate the TARGET18 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.* .

Solution: I was expecting "real" code generation, thus the recursive calls have to be instantiated and produce real code, link in the exercises we did in the course. I gave very few points to code coming with no explanation.

Question #3.3

Replace the conditional JUMP by a regular sequence of TARGET18 code (with temporaries).

Solution:

Voici le code généré par notre compilo :

```
;;Automatically generated TARGET code, MIF08 & CAP 2018
;;non executable 3-Address instructions version
        ;; (stat (assignment x = (expr (atom 12)) ;))
        leti temp_2 12
        let temp_1 temp_2
        ;; (stat (assignment y = (expr (atom 4)) ;))
        leti temp_3 4
        let temp_0 temp_3
        ;; (stat (if_stat if (condition_block (expr (atom ( (expr (expr (atom y)) > (expr (
    atom x))) ))) (stat_block (stat (assignment x = (expr (expr (atom x)) - (expr (atom y)))
    ;))))))
        leti temp_4 0
        ;; cond_jump lbl_end_relational_2 temp_0 sle temp_1
        cmp temp_0 temp_1
        jumpif sle lbl_end_relational_2
        ;; end cond_jump lbl_end_relational_2 temp_0 sle temp_1
        leti temp_4 1
lbl_end_relational_2:
```

1. We recall that the TARGET18 three address code has the same instruction set as the TARGET18 regular code except for conditions which use the idiom `condJUMP(label,t1,condition,t2)` and temporaries/virtual registers instead of regular registers).

```
        ;; cond_jump lbl_end_cond_1 temp_4 eq 0
        cmp temp_4 0
        jumpif eq lbl_end_cond_1
        ;; end cond_jump lbl_end_cond_1 temp_4 eq 0
        sub3 temp_5 temp_1 temp_0
        let  temp_1 temp_5
        jump lbl_end_if_0
lbl_end_cond_1:
lbl_end_if_0:


;;postlude
end:
        jump end
```

Exercise #4 ▶ **A variation on expressions and side-effects (30 min)**

Let us consider arithmetic expressions :

$$\begin{array}{rcll} expr & ::= & id & \textit{variable} \\ & | & n & \textit{const. value} \\ & | & expr + expr & \textit{plus} \\ & | & -\, expr & \textit{unary minus} \\ & | & \widetilde{f}(expr) & \textit{function} \end{array}$$

We also suppose that each $\widetilde{f}$ has an interpretation f as a partial function $\mathbb{N} \to \mathbb{N}$. We recall in the companion file the denotational semantics $\mathcal{A}$ for the subset of expressions without interpreted functions. Usual operations are interpreted in the standard way.

Question #4.1

Give an additional denotational rule for functions.

Question #4.2

Give an operational big steps semantics for expressions that we will denote by $(e, \sigma) \rightsquigarrow \sigma'$ (5 rules).

Question #4.3

Show the equivalence between these two semantics, *i.e.* :

$$(e, \sigma) \rightsquigarrow e' \Leftrightarrow \mathcal{A}[e]\sigma = e'$$

Now we add a new kind of expression, c **resultis** e, whose informal semantics is "we evaluate e in the environment obtained after evaluating the command c. Let us evaluate commands with the big steps semantics ($\to$) depicted in the companion file.

Question #4.4

Give the operational semantic rule for this new expression.

Question #4.5

Give the derivation tree for $(x := x + 1 \textbf{ resultis } x) + (y := x + x \textbf{ resultis } y)$ in the initial environment $\sigma : [x \mapsto 3]$.

Solution: $(e, \sigma) \rightsquigarrow \sigma'$ was a typo, of course expressions reduce to values in big steps semantics.. There was no real difficulty in this exercise.

Exercise #5 ▶ Typing with effects (40 min)

Adapted from JC Filliâtre, ENS Paris, and J. Goubault Larrecq, ENS Paris Saclay

We consider a variant of our mini-ML language seen in the course, whose reduction semantics is depicted in the companion file :

$$\begin{array}{rcll} e & ::= & & \\ & | & c & \text{cst only in } \mathbb{N} \text{ here} \ldots \\ & | & e + e & \text{plus} \\ & | & \texttt{fun } x \to e & \text{function} \\ & | & e\ e & \text{application} \end{array}$$

and we can use `let` $x = e_1$ `in` e_2 for syntactic sugar for $(\texttt{fun } x \to e_2)e_1$ This language is enriched with two new constructions :

$$\begin{array}{rcll} e & ::= & \text{print } e & \text{printing} \\ & | & \texttt{ifnul } e \texttt{ then } e \texttt{ else } e & \text{conditions} \end{array}$$

Question #5.1

Give a reduction rule for the `ifnul` construction, whose semantics is straighforward.

Solution: Be careful to use notations that are compatible with the provided semantics in the companion file. For this rule, you can choose to evaluate both branches of the if or not.

The informal semantics of the print expression is that it evaluates the expression and prints it on the standard output. Thus, now expressions have values (like before) as well as a printing side-effect.

Question #5.2

Give a new definition of reduction semantics that has side effects. *Explain notations, sets, and how rules are modified. Justify !*

Solution: A straighforward solution consists in changing configurations into $(value, \ell)$ where ℓ is the list of printed values. Then the printing rule :

Question #5.3

Compute the semantics of $print(2) - (print(60) - print(100))$ in your setting.

Solution: Here a proof tree. $[2, 60, 100]$ was one popular printing effect.

Now we will type our expressions with effects, ϕ, that can take three values :

— $\bot$ if the computation doesn't produce any printing.

— P if it may produce a printing, that doesn't depend of the evaluation order.

— $\top$ if it may produce a printing, that may depend of the evaluation order.

Effects are naturally ordered with $\bot \leq P \leq \top$, thus defining a `max` function on effects. An operation "upper bound" on effects is also defined by the three following equations :

— $\bot \sqcup \phi = \phi$,

— $\phi \sqcup \bot = \phi$,

— $\phi_1 \sqcup \phi_2 = \top$ else.

Typing judgments are now of the form $\Gamma \vdash e : \tau\&\phi$, which means "under the environment Γ, the expression e has type τ and effect ϕ". Types τ are either `int` or $\tau \xrightarrow{\phi} \tau$ for functions. Thus, function types contain their "latent effect", which will be the effect of its application.

We depict some of the typing rules here :

$$\frac{x \in dom(\Gamma)}{\Gamma \vdash x : \Gamma(x)\&\bot} \qquad \frac{\Gamma + x : \tau_1 \vdash e : \tau_2\&\phi}{\Gamma \vdash \texttt{fun}\ x \to e : \tau_1 \xrightarrow{\phi} \tau_2}$$

$$\frac{}{\Gamma \vdash n : \texttt{int}\&\bot}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int}\&\phi_1 \quad \Gamma \vdash e_2 : \texttt{int}\&\phi_2}{\Gamma \vdash e_1 + e_2 : \texttt{int}\&\phi_1 \sqcup \phi_2} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \xrightarrow{\phi} \tau_1\&\phi_1 \quad \Gamma \vdash e_2 : \tau_2\&\phi_2}{\Gamma \vdash e_1\ e_2 : \tau_2\&max(\phi_1 \sqcup \phi_2, \phi)}$$

Question #5.4

Explain the rule for function application *with the help of well-chosen examples.*

Solution: The difficult part here is to understand when max and union should be applied :

— the evaluation of the function e_1 or the argument e_2 can be in any order, thus if both of them have a side effect ($\phi_i \neq \bot$), we have $\top$ as the returned second element of the type.

— if only one has a non $\bot$ side effect, then it propagates into the result.

This explain the union.

+ examples

If then function has a latent effect ϕ, this will propagate to the result, hence the max.

Question #5.5

Give the rules for the constructions `print` and `test`.

Solution: Yes test was "if nul", sorry for the typo.

Question #5.6

Give 3 expressions e_1, e_2, e_3 such that :

— $\vdash e_1 : int\&\top$

— $\vdash e_2 : \big(int \xrightarrow{P} int\big)\&P$

— $\vdash e_3 : \big((int \xrightarrow{\top}) \xrightarrow{P} int\big)\&\bot$

Solution: typo in e_3, should add an int somewhere.

Question #5.7

What "correctness property" do we expect for this typing with respect to the semantics (2 propositions) ?

Solution:

— If a given expression types into $\tau\&P$, then ...

— If a given expression types into $\tau\&\bot$, then ...

Question #5.8

(Bonus) Prove these two propositions.