Final Exam
Compilation and Program Analysis (CAP)
January, 7th, 2020
Duration: 3 Hours

Instructions:

1. We give some typing/operational/code generation rules in a companion sheet.
2. Explain your proofs, semantic, typing and code generation rules!
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 (4-5 points)**

Solution: Idea of quadtrees Alexis Nasr.

We textually represent a square picture of size $N \times N$, N being a power of two, and the left/up coordinate is (0,0). Squares are cut so that each pieces of the cut have a unique color. 3 colors are available: r, g and b. During the cut process, either a subimage is of unique color, in which case we don't have to continue; either it is subdivided into 4 subimages. During the process, a tree can be constructed (leaves are subimages of a unique color, labeled by the color; a node depicts an image cut into 4, whose children are its subimages, in the order of a Z). An example of the process is depicted in Figure 1.

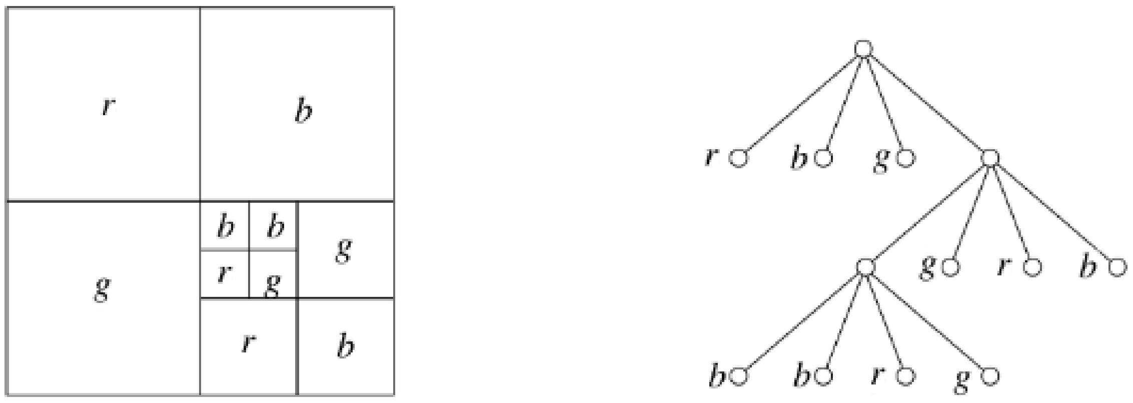

Figure 1: A quadtree

A quadtree can be represented as a parenthesis expression, for instance here `(r b g ((b b r g) g r b))`. The following grammar can generate such expressions:

```
S -> Q
Q -> ( Q Q Q Q )
Q -> r
Q -> g
Q -> b
```

For the following questions, you'll have to derive grammar attributes. For each computation, you have to give a name and a type, and say whether your attribute is **synthesized** (i.e computed from children to parents) or **inherited**, and give a computation by induction on the grammar

Question #1.1

Compute the size of each side of each sub-square (and sub-sub-square, and so on) of the picture.

Solution:

attribut hérité , avec N qui devient $\frac{N}{2}$ à chaque fois qu'on splitte.

Question #1.2

Compute the coordinates (x, y) of the left upper corner of each of these sub-squares.

Solution:

attribut hérité, (x, y) devient (x, y) ou $(x + N/2)$ etc, il faut donc propager la taille avec.

Question #1.3

Compute the number of pixels of color r of the picture.

Solution: attribut synthétisé qui vaut la surface $(N/2)^2$ si on dérive r et 0 si on dérive une autre couleur aux feuilles. Pour la règle de split on ajoute les 4 surfaces héritées.

Exercise #2 ▶ MiniC: 3 address Code generation (4-5 points)

Here is a piece of code in the MiniC language seen in the course:

```
int x; int y;
while (x > 3) {
   x = x - y;
}
```

Question #2.1

Generate the RISCV 3-address code for this code [1]. for the given program **according to the code generation rules**. *Recursive calls in the code generator, auxiliary temporaries, code, must be separated and clearly described. 0 pt for code only.*

Question #2.2

Generate the code (including register saving) for the function: *You may use ellipses, i.e. [...] for register saving/restoring the same way we did on the slides to avoid writting overly long repetitive code.*

```
int my_fun(int x, int y){
   while (x > 3) {
      x = x - y;
   }
   return x+1;
}
```

[1]We recall that the RISCV (our machine in 2019-20) three address code has the same instruction set as the RISCV regular code except for conditions which use the idiom `condJUMP(label,t1,condition,t2)` and temporaries/virtual registers instead of regular registers)

Question #2.3

Generate the code for the following call:

```
int toto ;
toto =  my_fun(12, 45);
```

Solution:

Voici le code généré par notre compilo:

```
python3 ../../../TP2019-20/TP04/MiniC-codegen/Main.py exam19.c --reg-alloc=none
```

```
##Automatically generated RISCV code, MIF08 & CAP 2019
##non executable 3-Address instructions version



##prelude

        .text
        .globl  my_fun
my_fun:
        addi sp, sp, -160
        sd ra, 0(sp)
        sd fp, 8(sp)
        addi fp, sp, 160



##Generated Code
        sd s1, 16(sp)
        sd s2, 24(sp)
        sd s3, 32(sp)
        sd s4, 40(sp)
        sd s5, 48(sp)
        sd s6, 56(sp)
        sd s7, 64(sp)
        sd s8, 72(sp)
        sd s9, 80(sp)
        sd s10, 88(sp)
        sd s11, 96(sp)
        mv temp_0, a0
        mv temp_1, a1
        # (stat (while_stat while (  (expr (expr (atom x)) > (expr (atom 3))) ) (stat_block
     { (block (stat  (assignment x = (expr (expr (atom x)) - (expr (atom y)))) ;)) })))
lbl_l_while_begin_0:
        li  temp_2, 3
        li  temp_3, 0
        ble temp_0, temp_2, lbl_end_relational_1
        li  temp_3, 1
lbl_end_relational_1:
        beq temp_3, zero, lbl_l_while_end_0
        # (stat (assignment x = (expr (expr (atom x)) - (expr (atom y)))) ;)
        sub temp_4, temp_0, temp_1
        mv temp_0, temp_4
        j  lbl_l_while_begin_0
lbl_l_while_end_0:
        # Return at end of function:
        li  temp_5, 1
        add temp_6, temp_0, temp_5
        mv a0, temp_6
        ld  s1, 16(sp)
        ld  s2, 24(sp)
        ld  s3, 32(sp)
        ld  s4, 40(sp)
        ld  s5, 48(sp)
        ld  s6, 56(sp)
        ld  s7, 64(sp)
        ld  s8, 72(sp)
        ld  s9, 80(sp)
        ld  s10, 88(sp)
        ld  s11, 96(sp)


##postlude

        ld  ra, 0(sp)
        ld  fp, 8(sp)
        addi sp, sp, 160
        ret
##Automatically generated RISCV code, MIF08 & CAP 2019
##non executable 3-Address instructions version


##prelude

        .text
        .globl  main
main:
        addi sp, sp, -160
        sd ra, 0(sp)
        sd fp, 8(sp)
        addi fp, sp, 160
```

```
##Generated Code
        sd s1, 16(sp)
        sd s2, 24(sp)
        sd s3, 32(sp)
        sd s4, 40(sp)
        sd s5, 48(sp)
        sd s6, 56(sp)
        sd s7, 64(sp)
        sd s8, 72(sp)
        sd s9, 80(sp)
        sd s10, 88(sp)
        sd s11, 96(sp)
        # (stat (assignment toto = (expr my_fun ( (expr_l (expr_l_nonempty (expr (
    atom 12)) , (expr_l (expr_l_nonempty (expr (atom 45)))))) ))) ;)
        li temp_1, 12
        li temp_2, 45
        sd t0, 104(sp)
        sd t1, 112(sp)
        sd t2, 120(sp)
        sd t3, 128(sp)
        sd t4, 136(sp)
        sd t5, 144(sp)
        sd t6, 152(sp)
        mv a0, temp_1
        mv a1, temp_2
        call my_fun
        ld t0, 104(sp)
        ld t1, 112(sp)
        ld t2, 120(sp)
        ld t3, 128(sp)
        ld t4, 136(sp)
        ld t5, 144(sp)
        ld t6, 152(sp)
        mv temp_3, a0
        mv temp_0, temp_3
        # Return at end of function:
        li temp_4, 0
        mv a0, temp_4
        ld s1, 16(sp)
        ld s2, 24(sp)
        ld s3, 32(sp)
        ld s4, 40(sp)
        ld s5, 48(sp)
        ld s6, 56(sp)
        ld s7, 64(sp)
        ld s8, 72(sp)
        ld s9, 80(sp)
        ld s10, 88(sp)
        ld s11, 96(sp)


##postlude

        ld ra, 0(sp)
        ld fp, 8(sp)
        addi sp, sp, 160
        ret
```

Exercise #3 ▶ **A parallel while with sleep (10-12 points)**

In this exercise, we want to extend the while language with threads and a sleep statement that pauses the execution for some units of time. The state of the program at execution contains thus additionally to the memory state, a global execution time.

We consider a mini-language with the following statements:

$$
\begin{array}{rcll}
S(Smt) & ::= & x := e & \text{assign} \\
 & | & skip & \text{do nothing} \\
 & | & S_1; S_2 & \text{sequence} \\
 & | & \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 & \text{test} \\
 & | & \texttt{newThread}[S] & \text{new thread} \\
 & | & \texttt{sleep}(c) & \text{sleep, i.e. pause for } c \text{ units of time}
\end{array}
$$

In the sequel, let $Prog$ be the following program:

```
X:=0;
newThread[sleep(1);sleep(2);X:=100;];
newThread[sleep(5);sleep(3);X:=52];
X:=102
```

Dynamic (small steps) semantics (approx 4 points) In this section we define three dynamic small steps semantics denoted by $\Rightarrow$ for our new language. In each case, configurations are of the form: $(S_1||S_2||\dots S_n, \sigma, t)$, where:

- $P = S_1||S_2||\dots S_n$ denotes a **non ordered set** $\{S_1, \dots S_n\}$ of statements.
  When $P = S_1||S_2||\dots S_n$, $S||P = S||S_1||S_2||\dots S_n$
- σ is the memory, as usual in Mini-while small step semantics.
- t is a positive natural number.

Intuitively, the elements of the non-ordered set are the statements "to be done in parallel", σ is the current value of variables, t is the time that progresses according to `sleep` instructions.

All three semantics have common starting configuration $(S, vars \mapsto \bot, 0)$ where S is the program to execute. Then, all rules in the small semantics of Mini-while are adapted in the following way:

$$\frac{(S,\sigma) \Rightarrow (S',\sigma')}{(S||P,\sigma,t) \Rightarrow (S'||P,\sigma',t)} \qquad \frac{(S,\sigma) \Rightarrow \sigma'}{(S||P,\sigma,t) \Rightarrow (P,\sigma',t)}$$

(Mini-while regular instructions have the same effect on memory and do not take time). Finally, they all share the rule for the `newThread` construction:

$$(\texttt{newThread}[S]; S'||P,\sigma,t) \Rightarrow (S||S'||P,\sigma,t) \qquad \textsc{NewThread}$$

The initial configuration to evaluate a program $Prog$ is $(Prog, \emptyset, 0)$. A final configuration is (σ, t) such that $(Prog, \emptyset, 0) \Rightarrow^* (\emptyset, \sigma, t)$. For the `sleep` semantics, we have three variants with their own rule(s):

1. NaiveSem: $(\texttt{sleep}(c); P||S_1 \ldots ||S_n, \sigma, t) \Rightarrow (P||S_1 \ldots ||S_n, \sigma, t+c)$.
2. MaxSleepSem: $(\texttt{sleep}(c_1); S_1||\texttt{sleep}(c_2); S_2 \ldots ||\texttt{sleep}(c_n); S_n, \sigma, t) \Rightarrow (S_1 \ldots ||S_n, \sigma, t{+}max_k\{c_k\})$.
3. MinSleepSem(two rules):
   $(\texttt{sleep}(0); P||S_1 \ldots ||S_n, \sigma, t) \Rightarrow (P||S_1 \ldots ||S_n, \sigma, t)$
   and $(\texttt{sleep}(c_1); S_1||\texttt{sleep}(c_2); S_2 \ldots ||\texttt{sleep}(c_n); S_n, \sigma, t) \Rightarrow (\texttt{sleep}(c_1 - c); S_1 \ldots ||\texttt{sleep}(c_n - c); S_n, \sigma, t+c)$ where $c = min_k\{c_k\}$).

Question #3.1

For each variant, say if we have a shared memory model or not.

For each variant, explain informally the semantics of sleep *in 2 lines maximum.*

Solution: Easy question: variables are shared, a unique sigma, thus shared memory.

Question #3.2

Show that there exists a unique derivation in MaxSleepSem to a final configuration for the program: `sleep(1)||x:=5;y:=7`.

Solution: You have to execute assigments in order to reduce to empty for the second process before being authorized to evaluate the `sleep`.

Question #3.3

What are the possible final configurations reached by $Prog$ according to the semantics NaiveSem.

Write the semantic rules for NaiveSem in order to exhibit an execution that leads to the **maximal final value** for `X` in $Prog$. *Like in the course, you can skip premises of the rules when you apply a small step and **only show the different states** reached by the program.*

Solution: The three possibilities for X are 100, 52 and 102. The final time is always 11. The reduction is with 11 steps, so quite long to write. To be detailed.

Question #3.4

Same question for MaxSleepSem.

Solution: Two possibilities, both start with:

- $X := 0$
- newThread
- newThread
- $X := 102$
- sleep(1) and sleep(5) for a waiting time of 5
- $X := 5$
- sleep(2) and sleep(3) for a waiting time of 3

Then it is $X := 100$ and $X := 52$ in any order, leading to the final configuration where X is worth 100 or 52, and t is 8. (Number of steps: 9)

Question #3.5

Same for MinSleepSem.

Solution: Only one possibility: 52, with final time 8, and 15 reduction steps.

- $X := 0$
- newThread
- newThread
- $X := 102$
- sleep(1) and sleep(5) for a waiting time of 1
- sleep(0)
- sleep(2) and sleep(5-1) for a waiting time of 2
- sleep(0)
- $X := 100$
- sleep(5-1-2) for a waiting time of 2
- sleep(0)
- $X := 5$
- sleep(3) for a waiting time of 3
- sleep(0)
- $X := 52$

Static Semantics (approx 4 points) The object of the next questions is to design a static semantics that infers the maximal sleep in a program, and to prove a certain notion of correctness.

Informally, we want :
" $\vdash S : \text{Sleep}(n)$ when n is the largest sleep that can be performed by the statement S, potentially 0 if no sleep is performed."

For example, for $S = \mathit{sleep}(3)\ ;\ \mathit{sleep}(2)$, we want to be able to prove $\vdash S : \text{Sleep}(3)$.

More formally, we look for a judgment $\vdash S : \text{Sleep}(n)$ such that

$$\vdash S : \text{Sleep}(n) \ \wedge\ (S, \sigma, t) \Rightarrow^* (\mathit{sleep}(n'); S' || P, \sigma, t) \implies n' \leq n \qquad \text{"sleep property"}$$

Question #3.6

Design the static semantics that defines the judgment described above for all statements of the program, we give you one simple rule to start with:

$$\vdash x := e : \text{Sleep}(0)$$

You have to provide the other rules.

Solution:

$$\mathit{skip} : 0 \qquad \frac{S_1 : n \qquad S_2 : n' \qquad m = \mathit{max}(n, n')}{S_1; S_2 : m} \qquad \frac{S_1 : n \qquad S_2 : n' \qquad m = \mathit{max}(n, n')}{\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 : m}$$

$$\texttt{sleep}(c) : c \qquad \frac{\vdash S : \text{Sleep}(n)}{\vdash \mathit{newThread}(S) : \text{Sleep}(n)}$$

Question #3.7

Use your rules to find n such that $\vdash S : \text{Sleep}(n)$
when $S = (x := 3; \texttt{if}\ x > 1\ \texttt{then}\ \texttt{sleep}(2)\ \texttt{else}\ x := 1)$ and when $S = \mathit{Prog}$.
Detail enough steps to illustrate the important rules.

Solution: sleep(5) pour prog et sleep(2) pour le if

Question #3.8

The `sleep` property only reasons on programs (or on initial configurations if you prefer) and not on configurations (i.e. not on parallel statements). If one wants to prove correctness of the static semantics, one would like to have a similar property for a running configuration, i.e. a property of the form:

$$\text{properties on } Si \wedge (S_1 || .. || S_n, \sigma, t) \Rightarrow^* \ldots \implies \ldots$$

Complete the property. You do not have to prove it but you should explain in 2 or 3 sentences how the proof will be organized to show that your statement is the right one.

Solution:

$$\forall i.S_i : sleep(n_i) \wedge \forall i.n_i \leq n \wedge (S_1||..||S_n, \sigma, t) \Rightarrow^* (sleep(n'); S||P, \sigma', t') \implies n' \leq n$$

A scheduler based on the static semantics (approx 4 points) We now decide that it is not useful to spawn a new thread for doing a task that does not introduce delays. We thus replace the new-thread rule by the following two rules:

$$\textsc{sched-0}\quad \frac{\vdash S : \text{Sleep}(0)}{(\texttt{newThread}[S]; S'||P, \sigma, t) \Rightarrow (S; S'||P, \sigma, t)} \qquad \textsc{Sched-1}\quad \frac{\vdash S : \text{Sleep}(n) \qquad n > 0}{(\texttt{newThread}[S]; S'||P, \sigma, t) \Rightarrow (S||S'||P, \sigma, t)}$$

We want to compare this new semantics with the former one in the three cases. We call $\Longrightarrow_s$ the SOS semantics using Sched-0 and Sched-1 but not NewThread.

Question #3.9

NaiveSem: Prove that when using the new thread creation rules (Sched-0 and Sched-1) we obtain *a possible reduction* of NaiveSem, i.e. if $(P, \sigma, t) \Longrightarrow_s^* (P', \sigma', t')$ by NaiveSem then $(P, \sigma, t) \Rightarrow^* (P', \sigma', t')$ by NaiveSem. Prove all useful cases.

Explain why we do not have the converse, i.e. $(P, \sigma, t) \Rightarrow^* (P', \sigma', t')$ by NaiveSem but (P', σ', t') cannot be obtained by $\Longrightarrow_s$ (and NaiveSem).

Question #3.10

MinSleepSem: Explain why the same applies to MinSleepSem, i.e. $\Longrightarrow_s$ and MinSleepSem provides only reductions that are possible with $\Rightarrow$ and MinSleepSem, but bot all of them.

Question #3.11

MaxSleepSem: This simulation of one semantics by another is not possible with MaxSleepSem. Give an example of a program for which the semantics of (and the final state obtained by) $\Longrightarrow_s$ and MaxSleepSem cannot be obtained by applying $\Rightarrow$ and MaxSleepSem. Hint: use $\texttt{sleep}(0)$.

Solution: Counter example:$\texttt{newThread}[\texttt{sleep}(0); \texttt{sleep}(0); x := 2]; \texttt{sleep}(1); x := 4$ by the sched semantics, we necessarily obtain $(x \mapsto 4)$ at the end and by the MaxSleepSem semantics we have necessarily $(x \mapsto 2)$

Question #3.12

Modify the static semantics and modify the rules Sched-0 and Sched-1 to allow a direct scheduling (i.e. no thread creation) in cases that do not raise incompatibilities with MaxSleepSem. You should notice that the problem is with $\texttt{sleep}(0)$ and should only prevent thread creation if *no sleep statement* can possibly exist in the spawned thread. Explain why your new reduction $\rightarrow_S$ ensures that $\rightarrow_S$ and MaxSleepSem provides reductions that are possible with $\Rightarrow$ and MaxSleepSem.

Solution: The static judgment should infer $\vdash S$: Nosleep if no sleep instruction exist and $\vdash S : \mathrm{Sleep}(n)$ if there might be a sleep (n is not necessary here). then scheduling rules are: