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.

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) :

AND r0 r3 #2 *;*

Question #2.2

Disassemble by hand the following instruction, given in binary :

0000110000000111 *;*

<table>
<tr><th>syntaxe</th><th>action</th><th>NZP</th><th colspan="16">codage</th></tr>
<tr><td></td><td></td><td></td><td colspan="4">opcode</td><td colspan="12">arguments</td></tr>
<tr><td></td><td></td><td></td><td>F</td><td>E</td><td>D</td><td>C</td><td>B</td><td>A</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>NOT DR,SR</td><td>DR ← not SR</td><td>*</td><td colspan="4">1 0 0 1</td><td colspan="3">DR</td><td colspan="3">SR</td><td colspan="6">1 1 1 1 1 1</td></tr>
<tr><td>ADD DR,SR1,SR2</td><td>DR ← SR1 + SR2</td><td>*</td><td colspan="4">0 0 0 1</td><td colspan="3">DR</td><td colspan="3">SR1</td><td>0</td><td colspan="2">0 0</td><td colspan="3">SR2</td></tr>
<tr><td>ADD DR,SR1,Imm5</td><td>DR ← SR1 + SEXT(Imm5)</td><td>*</td><td colspan="4">0 0 0 1</td><td colspan="3">DR</td><td colspan="3">SR1</td><td>1</td><td colspan="5">Imm5</td></tr>
<tr><td>AND DR,SR1,SR2</td><td>DR ← SR1 and SR2</td><td>*</td><td colspan="4">0 1 0 1</td><td colspan="3">DR</td><td colspan="3">SR1</td><td>0</td><td colspan="2">0 0</td><td colspan="3">SR2</td></tr>
<tr><td>AND DR,SR1,Imm5</td><td>DR ← SR1 and SEXT(Imm5)</td><td>*</td><td colspan="4">0 1 0 1</td><td colspan="3">DR</td><td colspan="3">SR1</td><td>1</td><td colspan="5">Imm5</td></tr>
<tr><td>LEA DR,label</td><td>DR ← PC + SEXT(PCoffset9)</td><td>*</td><td colspan="4">1 1 1 0</td><td colspan="3">DR</td><td colspan="9">PCoffset9</td></tr>
<tr><td>LD DR,label</td><td>DR ← mem[PC + SEXT(PCoffset9)]</td><td>*</td><td colspan="4">0 0 1 0</td><td colspan="3">DR</td><td colspan="9">PCoffset9</td></tr>
<tr><td>ST SR,label</td><td>mem[PC + SEXT(PCoffset9)] ← SR</td><td></td><td colspan="4">0 0 1 1</td><td colspan="3">SR</td><td colspan="9">PCoffset9</td></tr>
<tr><td>LDR DR,BaseR,Offset6</td><td>DR ← mem[BaseR + SEXT(Offset6)]</td><td>*</td><td colspan="4">0 1 1 0</td><td colspan="3">DR</td><td colspan="3">BaseR</td><td colspan="6">Offset6</td></tr>
<tr><td>STR SR,BaseR,Offset6</td><td>mem[BaseR + SEXT(Offset6)] ← SR</td><td></td><td colspan="4">0 1 1 1</td><td colspan="3">SR</td><td colspan="3">BaseR</td><td colspan="6">Offset6</td></tr>
<tr><td>BR[n][z][p] label</td><td>Si (cond) PC ← PC + SEXT(PCoffset9)</td><td></td><td colspan="4">0 0 0 0</td><td>n</td><td>z</td><td>p</td><td colspan="9">PCoffset9</td></tr>
<tr><td>NOP</td><td>No Operation</td><td></td><td colspan="4">0 0 0 0</td><td>0</td><td>0</td><td>0</td><td colspan="9">0 0 0 0 0 0 0 0 0</td></tr>
<tr><td>RET (JMP R7)</td><td>PC ← R7</td><td></td><td colspan="4">1 1 0 0</td><td colspan="3">0 0 0</td><td colspan="3">1 1 1</td><td colspan="6">0 0 0 0 0 0</td></tr>
<tr><td>JSR label</td><td>R7 ← PC ; PC ← PC + SEXT(PCoffset11)</td><td></td><td colspan="4">0 1 0 0</td><td>1</td><td colspan="11">PCoffset11</td></tr>
</table>

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 :

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

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.*

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

Question #3.2

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

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.

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

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

Question #3.4

Apply these rules on the example.

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 :

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

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

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.

(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

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

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

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 :

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

Question #5.1

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

Question #5.2

Give rules for substraction and multiplication.

Question #5.3

Give rules for the sequence and test (if).

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 $\mathcal{A}[a]\sigma < 0$.

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 $(\Gamma', \sigma') \in \mathcal{R}$.

Question #5.6

What does that mean for our typing system ?