Possible Answers for Midterm Exams (June 30, 2020)

Compilers Course

Masters in Informatics and Computing Engineering (MIEIC), 3rd Year

U.PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

DEI DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

MIDTERM EXAM 1 (45 MIN.)

Group 1. Lexical and Syntactic Analysis (9 pts)

- The CFG (Context-Free Grammar) G1 below represents a grammar of a simple programming language.

Some of the Tokens for G1:

```
REG = $[0-9][0-9]
IF = if
GOTO = goto
CMP = == | < | > | <= | >= | !=
CONST = [0-9]+
OP = + | - | * | /
LABEL = [a-zA-Z][a-zA-Z]*
```

Grammar G1:

```
S → Stmt (Stmt)*
Stmt → LABEL : | Assign | If | GOTO LABEL ;
Assign → Lhs = Operand (OP Operand)? ;
Operand → REG | CONST | Mem
If → IF Cond GOTO LABEL
Cond → Operand CMP Operand
Lhs → REG | Mem
Mem → “M[" Operand “]”
```

Code1:

```
 $2=0;
L2:
 if $2  >= 100 goto L1
 $3 = 4*$2;
 $4 = $1+$3;
 M[$4] = 0;
 $2 = $2+1;
 goto L2;
L1:
```

Group 1. Lexical and Syntactic Analysis (9 pts)

- **a) [1pt] Write the chain of the first 10 tokens resultant from the lexical analysis for Code1 below;**

- **REG("$2") » T1("=") » CONST("0") » T2(";") » LABEL("L") CONST("2") » T3(":") » IF("if") » REG("$2") » CMP(">=")**

Some of the Tokens for G1:

```
REG = $[0-9][0-9]
IF = if
GOTO = goto
CMP = == | < | > | <= | >= | !=
CONST = [0-9]+
OP = + | - | * | /
LABEL = [a-zA-Z][a-zA-Z]*
```

Code1:

```
 $2=0;
L2:
 if $2  >= 100 goto L1
 $3 = 4*$2;
 $4 = $1+$3;
 M[$4] = 0;
 $2 = $2+1;
 goto L2;
L1:
```

Group 1. Lexical and Syntactic Analysis (9 pts)

- **b) [2pt] Give the *First* and *Follow* sets for the grammar variables: *Assign*, *Lhs*, and *If*;**

- First(Assign) = First(Lhs) = {REG, “M[“ }
- First(if) = {IF}

- Follow(Assign) = Follow(if) = { LABEL, REG, “M[“, IF, GOTO}
- Follow(Lhs) = {“=”}

Group 1. Lexical and Syntactic Analysis (9 pts)

- **c) [2pt] Show the rows of the table for the parser LL(1) considering only the rows related to variables *Assign*, *Lhs*, and *If*, and the columns with the tokens whose cells are not empty in the considered section of the table;**

	REG	“M[“	IF
Assign	Assign → Lhs = Operand (OP Operand)?	Assign → Lhs = Operand (OP Operand)?	
Lhs	Lhs → REG	Lhs → Mem	
If			If → IF Cond GOTO LABEL

Group 1. Lexical and Syntactic Analysis (9 pts)

- **d) [1.5pt] Based on the section of the parser table you presented before, could you conclude that grammar G1 is not LL(1)? Why?**

- No. In those cells there aren't conflicts.

Group 1. Lexical and Syntactic Analysis (9 pts)

- **e) [2.5pt] Show the function, considering the grammar rule of line 5 (i.e., for variable *If*), and the respective pseudocode to implement it (considering only the syntax check and not the creation of the syntax tree), as a top-down recursive LL parser and considering the value of lookahead required for this grammar rule. Assume the existence of the lexical analyzer, which outputs the sequence of tokens, of the global variable *token*, and of the function *next()*, which returns the next token in the sequence of tokens (in the beginning the variable *token* identifies the first token in the sequence);**

```
boolean if () {
    if(token != IF) return false;
    token = next();
    if(!Cond()) return false;
    if(token !=GOTO) return false;
    token = next();
    if(token != LABEL) return false;
    token = next();
    return true;
}
```

Group 2. AST, Symbol Table and High-Level Representation (7 pts)

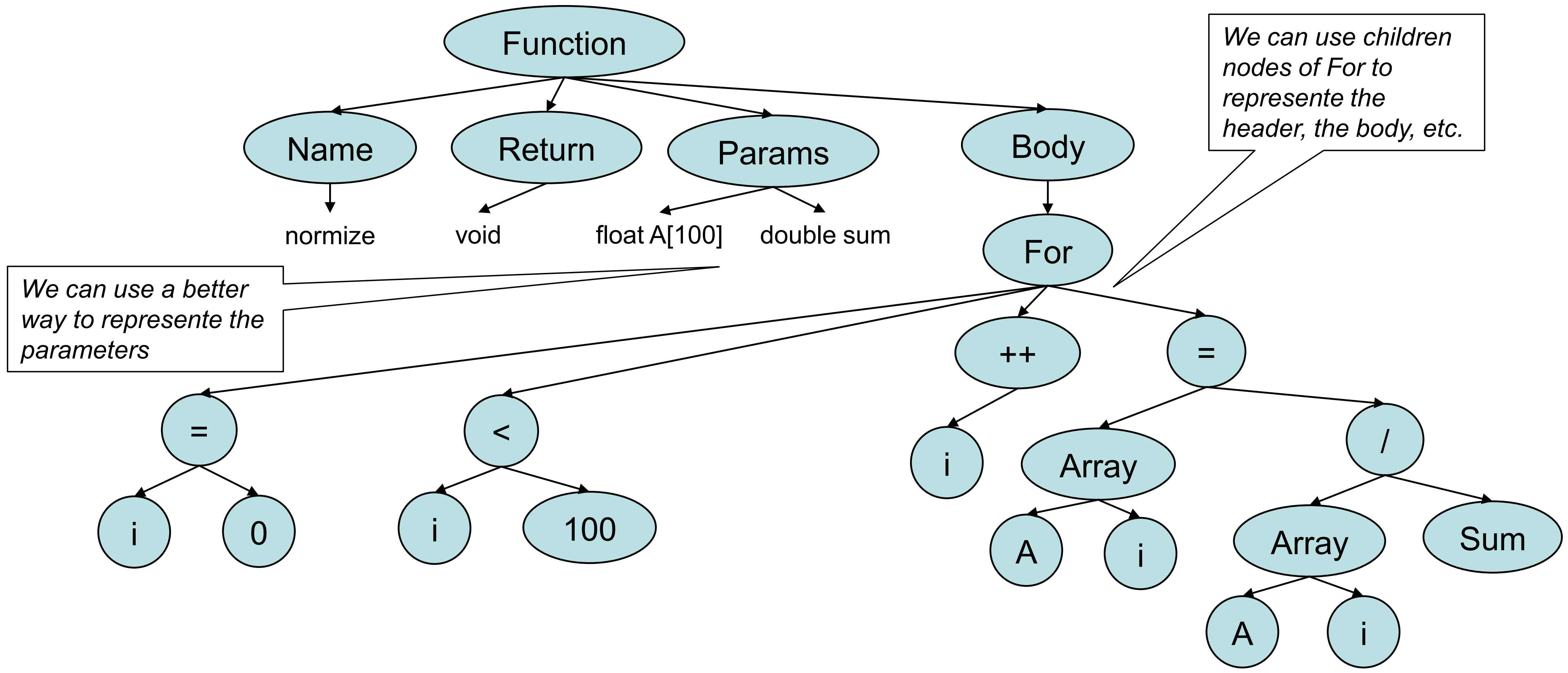
- Consider the function presented in Code2 based on the C programming language:

Code2:

```
void normize(float A[100],
double sum) {
  int i;
  for(i=0; i < 100; i++)
    A[i] = A[i]/sum;
}
```

Group 2. AST, Symbol Table and High-Level Representation (7 pts)

- **a) [2pt] Draw a possible AST for Code2;**

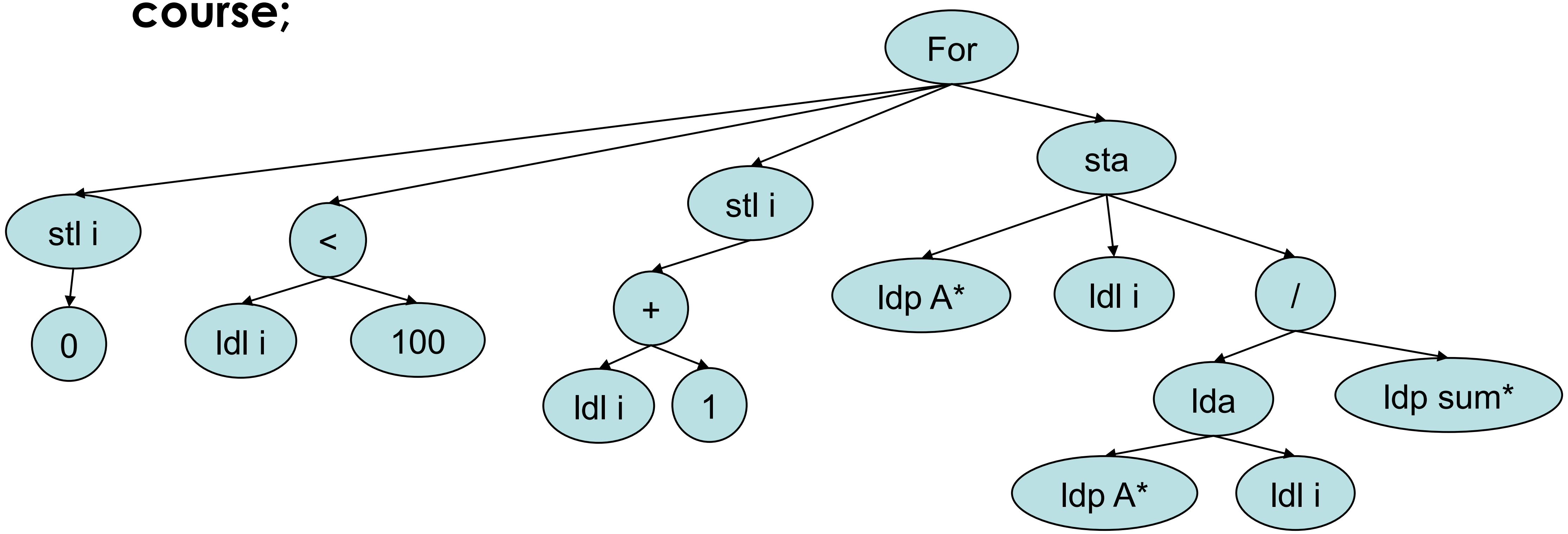


We can use children nodes of For to represente the header, the body, etc.

We can use a better way to represente the parameters

Group 2. AST, Symbol Table and High-Level Representation (7 pts)

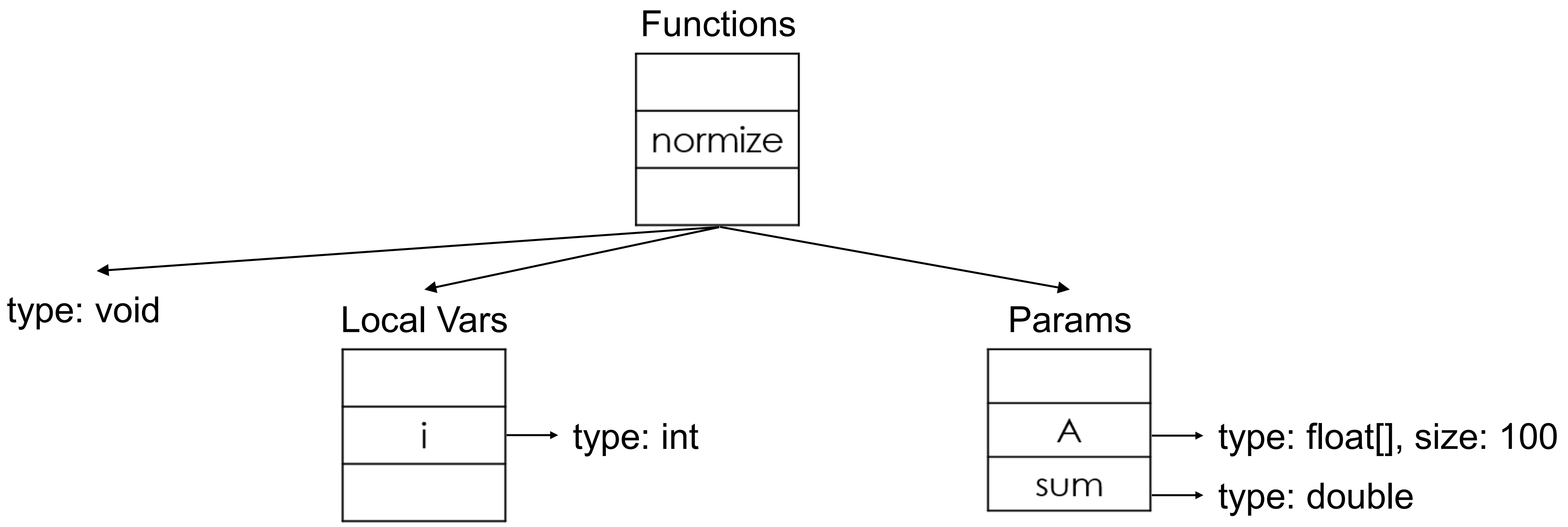
- b) [2pt] Draw a possible high-level representation based on the expression trees presented in the lectures of the compiler course;



* Would also accept "ldp <index>", e.g., ldp 1

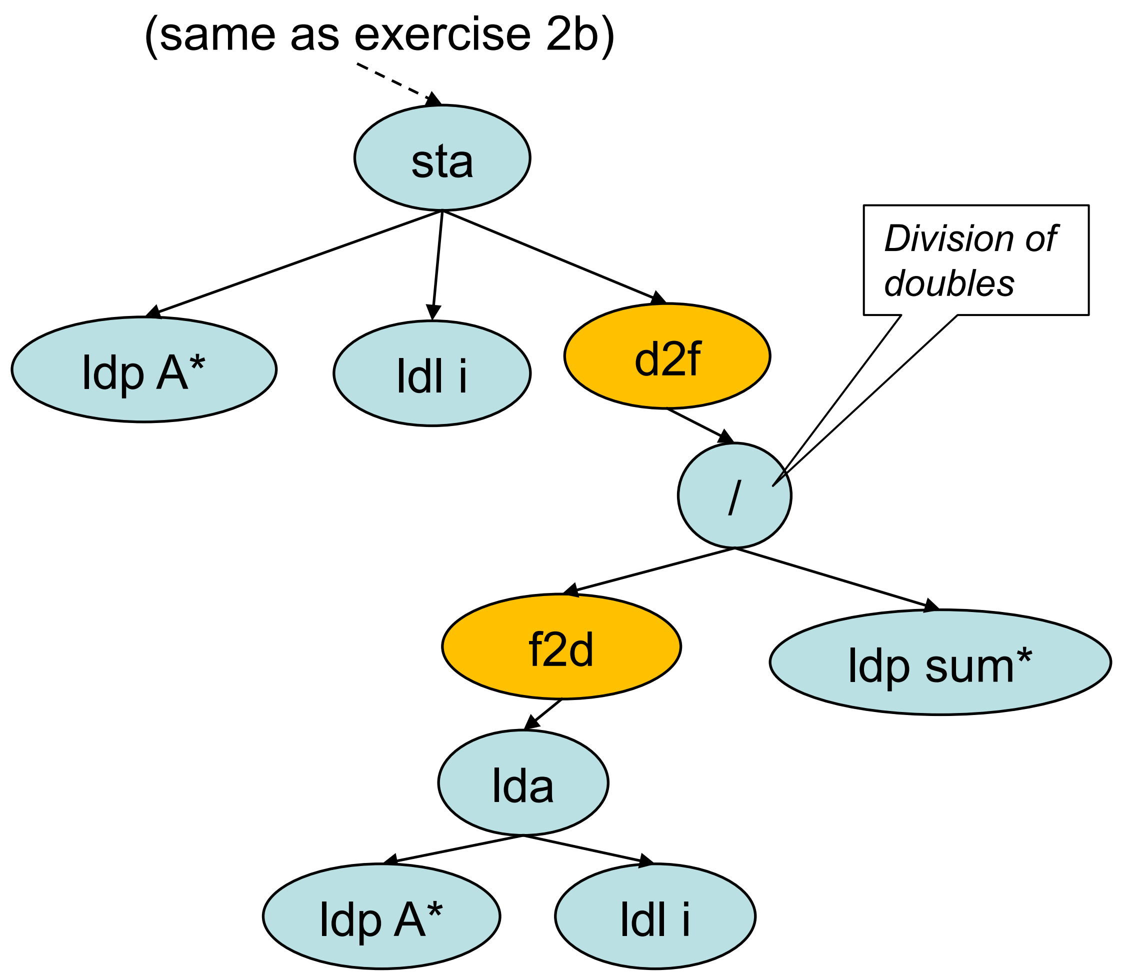
Group 2. AST, Symbol Table and High-Level Representation (7 pts)

- **c) [1.5pt] Draw a possible symbol table for the function considering that the language only considers a single local scope;**



Group 2. AST, Symbol Table and High-Level Representation (7 pts)

- **d) [1.5pt] Draw a possible high-level representation, based on the expression trees presented in the lectures of the compiler course, and after semantic analysis;**



Group 3. Comment the sentences below and justify why you consider each one true or false (4 pts)

- **a) [2pt] The semantic rules of a programming language are typically expressed in the grammar of the language used to build the frontend of the compiler.**

- *This sentence is false. Typically the semantic rules are checked in a semantic analysis stage after the syntactic analysis and the construction of the AST and of the Symbol Table. The main reason is the fact that it is typically not possible to express all the semantic rules in the grammar (e.g., to check the type of a variable or if it was initialized), but this would depend in the programming language as we may conceived a programming language where semantic rules can be expressed in the grammar. Anyway, there are some semantic rules typically addressed as grammar rules as is the case of the operator precedence of the language.*

Group 3. Comment the sentences below and justify why you consider each one true or false (4 pts)

- **b) [2pt] Given any context-free grammar, it is impossible to generate the AST (Abstract Syntax Tree) without generating first the CST (Concrete Syntax Tree).**
- *This sentence is false. Typically, the AST is generated at the parser level and without generating first the CST. For any CFG, one can always guide the construction of the tree in a way that it is anymore a CST and it is an AST. An example of that is when one defines rules to avoid some nodes or to build the tree in a different way in JJTree of JavaCC.*

- **(End.)**

MIDTERM EXAM 2 (45 MIN.)

Group 1. Low-Level Intermediate Representation (LLIR) and Instruction Selection (7 pts)

Code1:

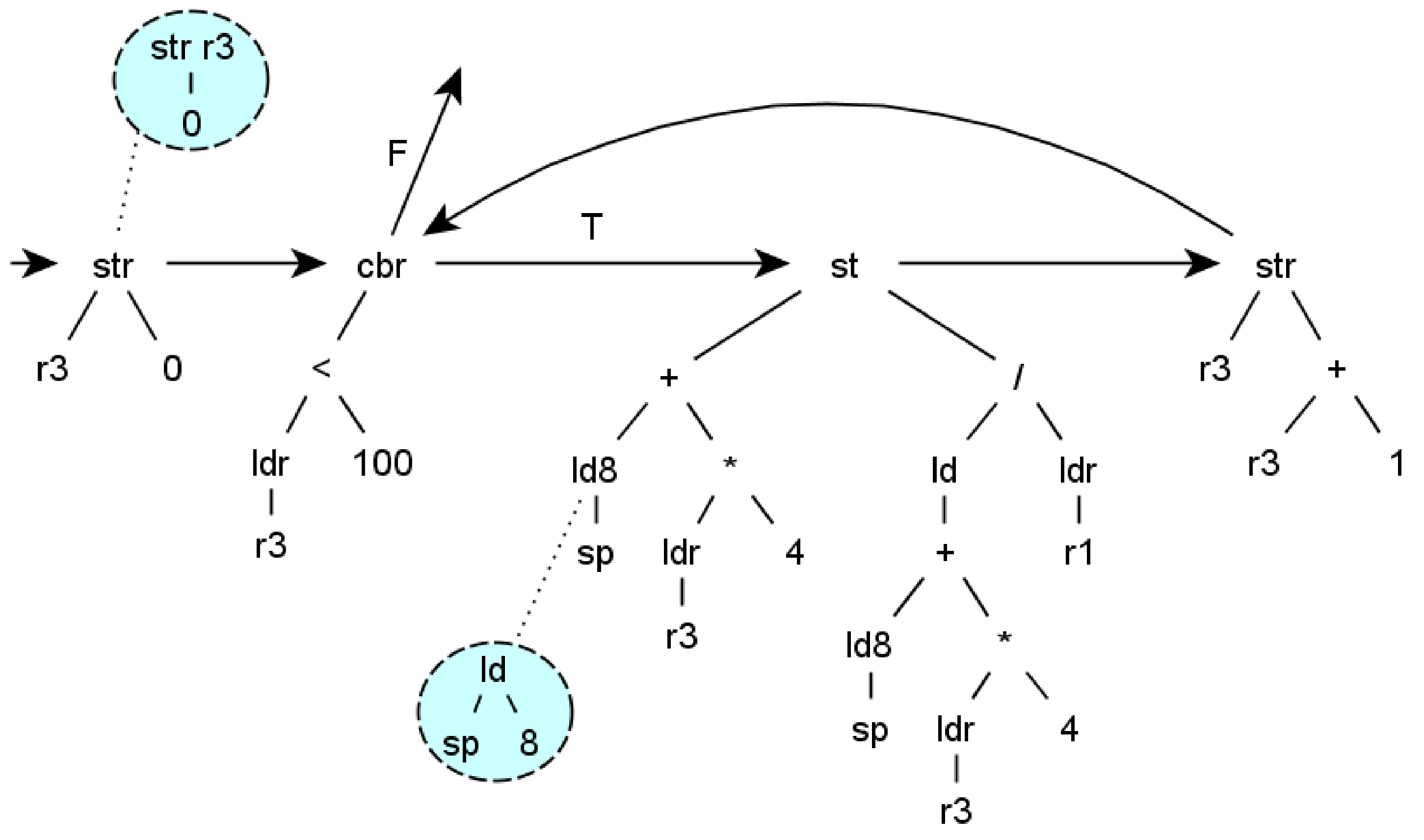
```
for(i=0; i < 100; i++)
      A[i] = A[i]/sum;
```

Var	Type	Storage
A	int32 A[100] (1D array of 32-bit integers)	Base address of A in the stack at SP + 8
i	int (32-bit scalar variable)	register r3
sum	int (32-bit scalar variable)	register r1

Instruction	Operation
load Ri, Rj, C	Ri ← Mem[Rj+C]
store Rj, C, Ri	Mem[Rj+C] ← Ri
add Ri, Rj, Rk	Ri ← Rj + Rk
addi Ri, Rj, C	Ri ← Rj + C
mul Ri, Rj, Rk	Ri ← Rj * Rk
div Ri, Rj, Rk	Ri ← Rj / Rk
lth Ri, Rj, label1	If(Ri < Rj) goto label1
gte Ri, Rj, label1	If(Ri >= Rj) goto label1
jump label1	goto label1

Group 1. Low-Level Intermediate Representation (LLIR) and Instruction Selection (7 pts)

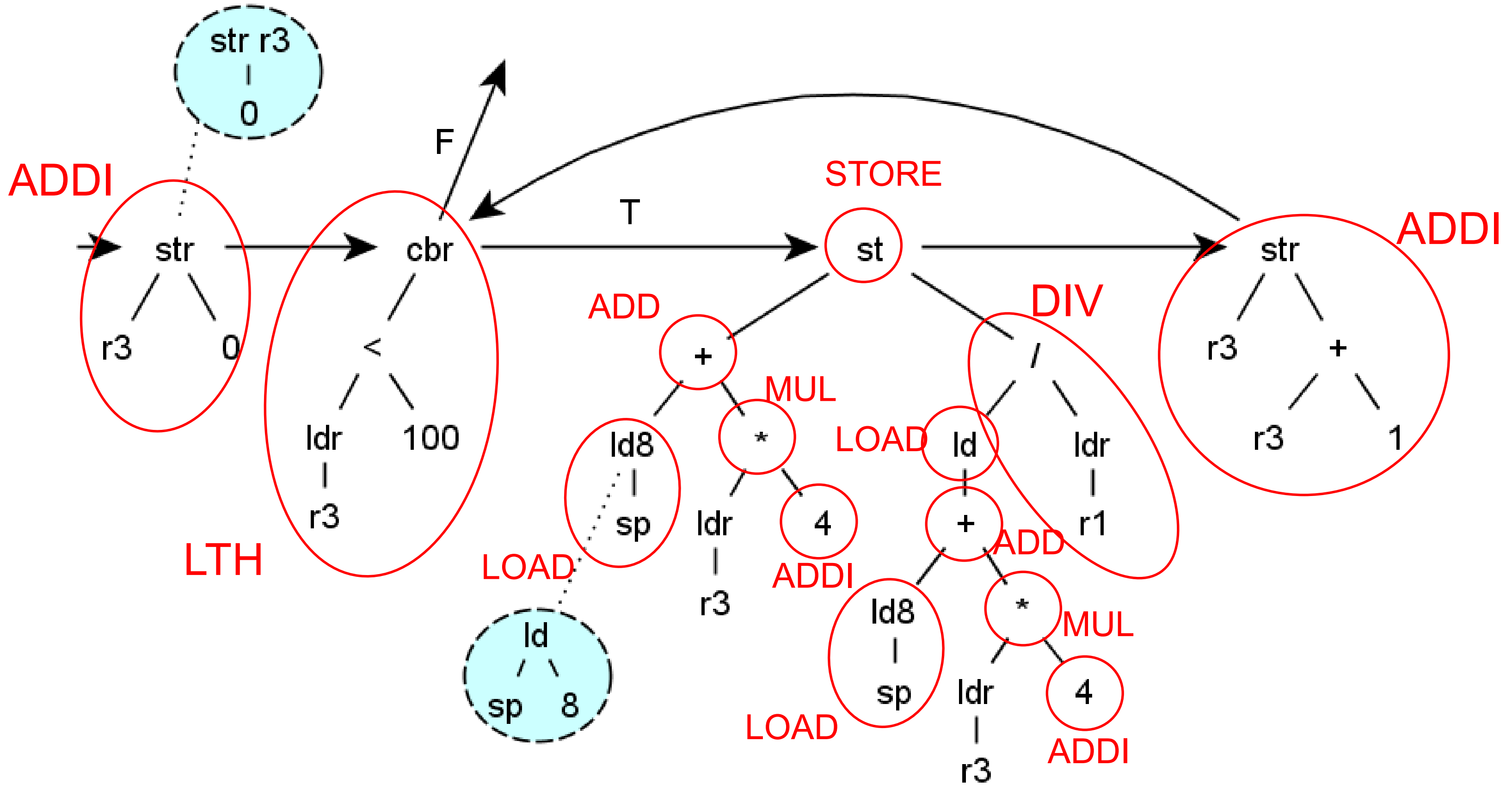
- **a) [3pt] Indicate the LLIR for the section c the code in the example Code1 based on the LLIR presented in the lectures of the course, which is based on expressio trees, and the type and storage of the variables given by the table below.**



* Would also accept reads of using only the register (i.e., values in registers without using **ldr**)

Group 1. Low-Level Intermediate Representation (LLIR) and Instruction Selection (7 pts)

- b) [4pt] Consider a target machine with 32 32-bit registers (R0 stores 0) including the instructions presented in the table below, where Ri, Rk, and Rj represent registers of the machine (from R0 to R31), C represents a 16-bit signed integer constant, and label1 represents a label identifying the target instruction of the jump. Using the Maximal Munch algorithm, present the instruction selection for the LLIR presented in 1.a) when targeting the machine of 1.b) (it is enough to draw in the LLIR the group of nodes for each selected instruction).

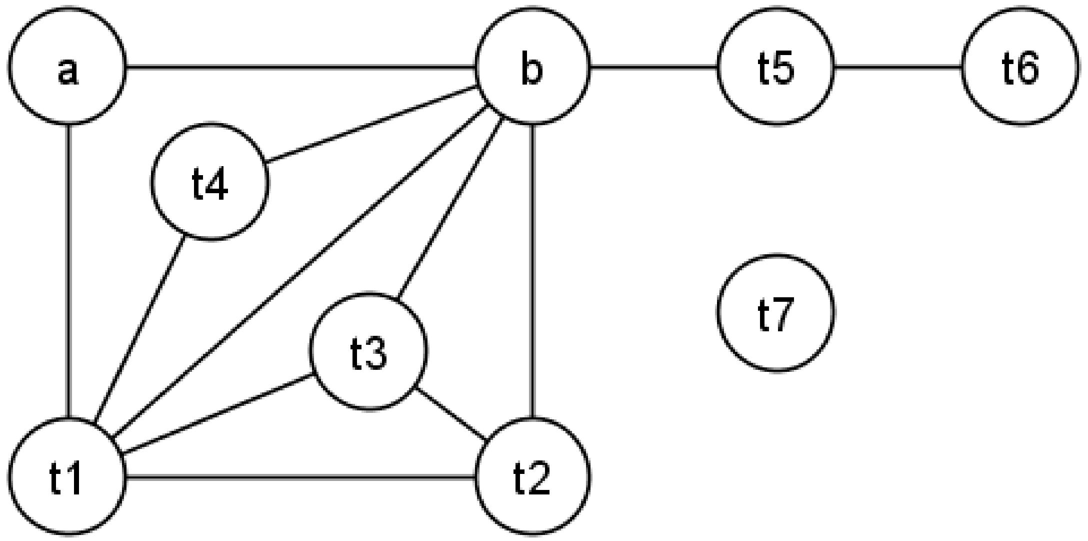


Group 2. Scheduling and Register Allocation (9 pts)

- a) [2pt] By only inspecting the code, present the interference graph for Code2, which would result from liveness analysis.

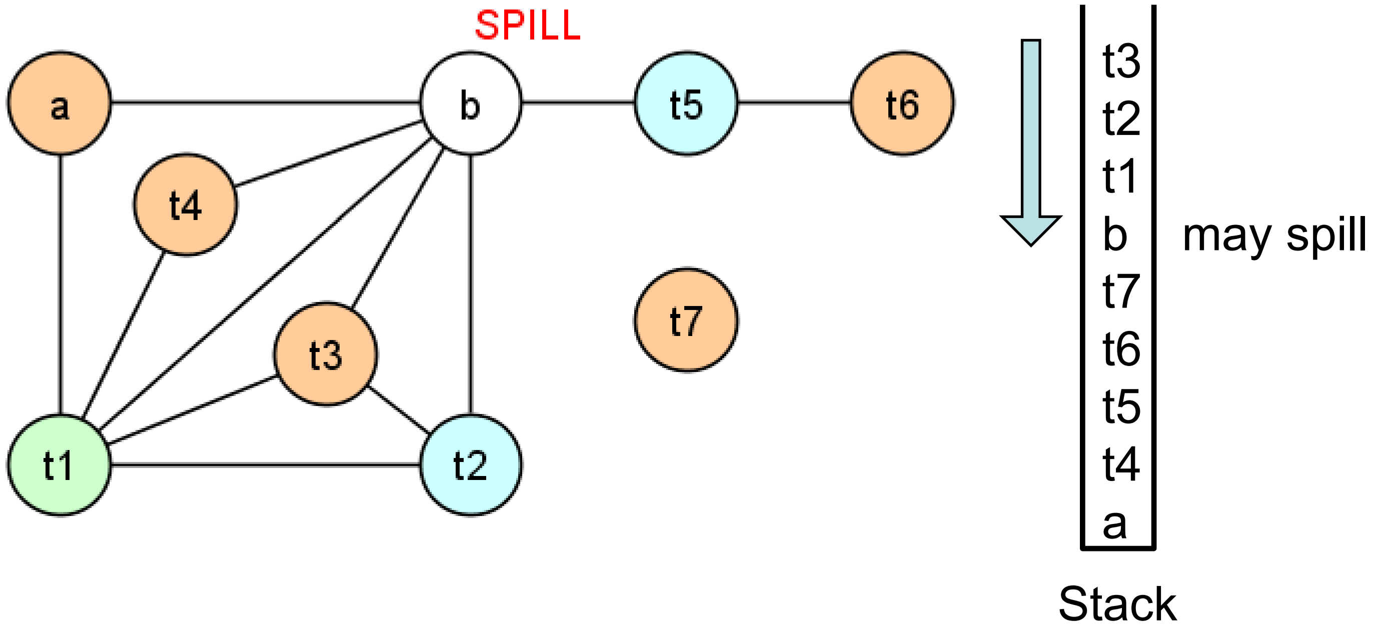
Code2:

```
   Live-in={a,b}
1.  t1 := a*a
2.  t2 := a*b
3.  t3 := 2;
4.  t4 := t3*t2
5.  t5 := t1+t4
6.  t6 := b*b
7.  t7 := t5+t6
   Live-out={t7}
```



Group 2. Scheduling and Register Allocation (9 pts)

- b) [3pt] Consider the interference graph (IG) of 2.a). Indicate a possible allocation of registers using the graph coloring based register allocation algorithm presented in the course lectures and considering a maximum of 3 registers (*R1, R2, and R3*). Show the content of the stack immediately after the simplification of the IG. In the case of needing to perform *spilling*, use the degree of each node to decide.



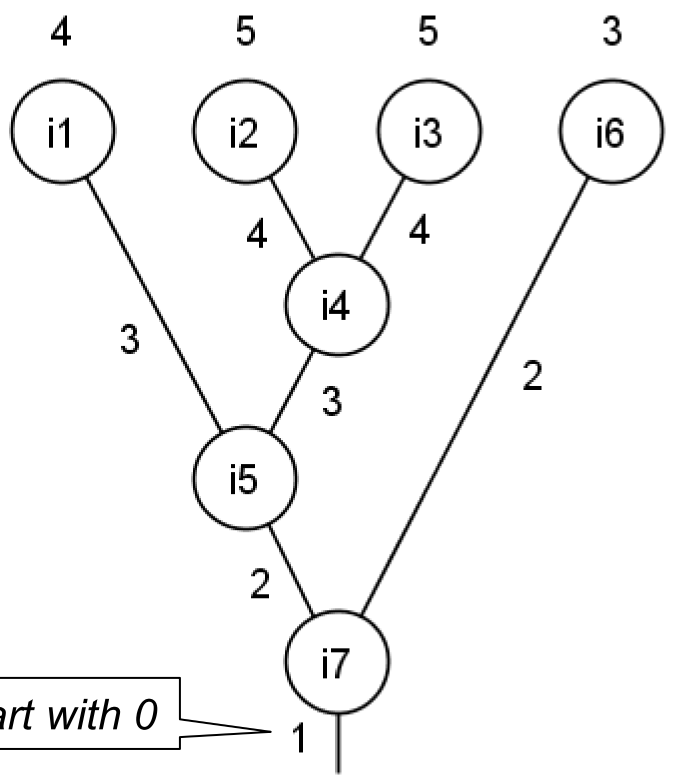
R1={a, t4, t3, t7, t6}
R2={t2, t5}
R3={t1}

Group 2. Scheduling and Register Allocation (9 pts)

- **c) [4pt] Considering that in the target machine all instructions execute in 1 clock cycle and the machine has one unit for load/store instructions and two units for the other instructions, present the scheduling of instructions resultant of applying the list-scheduling algorithm to the example in Code3. Present the data-dependence graph and the criterion to order the instructions ready for scheduling.**

Code3:

```
i1:        mul R1, R8, R8
i2:        mul R2, R8, R9
i3:        addi R3, R0, 2
i4:        mul R4, R3, R2
i5:        add R5, R1, R4
i6:        mul R6, R9, R9
i7:        add R7, R5, R6
```



We can start with 0

Cycle	FU1	FU2
1	i2	I3
2	i1	I4
3	i5	I6
4	i7	-

4 cycles

For the criterion to order the instructions, we used the longest path delay. The first set of instructions to be scheduled is ordered by i2, i3, i1, i6 (tie-break for deciding between i2 and i3?))

Group 3. Comment the sentences below and justify why you consider each one true or false (4 pts)

- **a) [2pt] It does not matter if you do register allocation before or after scheduling as in any way the code generated will be similar.**
- *This sentence is false*. Register allocation can be done before or after scheduling and there are advantages and disadvantages for doing one before the other.
- *See Section 10.2.4 Phase Ordering Between Register Allocation and Code Scheduling, page 715 of the Dragon Book (2nd edition):*

  "If registers are allocated before scheduling, the resulting code tends to have many storage dependences that limit code scheduling. On the other hand, if code is scheduled before register allocation, the schedule created may require so many registers that register spilling (storing the contents of a register in a memory location, so the register can be used for some other purpose) may negate the advantages of instruction-level parallelism. Should a compiler allocate registers before it schedules the code? Or should it be the other way round? Or, do we need to address these two problems at the same time?

  To answer the questions above, we must consider the characteristics of the programs being compiled. Many nonnumeric applications do not have that much available parallelism. It suffices to dedicate a small number of registers for holding temporary results in expressions…"

- *The best results could be achieved by applying both register allocation and instruction scheduling simultaneously. However, this makes the problem even more complex!*

Group 3. Comment the sentences below and justify why you consider each one true or false (4 pts)

- **b) [2pt] When doing dataflow analysis for liveness analysis, we do not consider array variables as it is impossible to determine their liveness analysis using the dataflow analysis considered in the compiler course.**
- *This sentence is false. In the course we do not consider the* **liveness analysis** of *array variables as arrays are stored in the heap or in the stack, and thus in that case we do not have the goal to assign to registers as many as possible variables in a method/function/procedure. However, it is possible to use the dataflow analysis technique (presented in the course) to calculate the liveness analysis of arrays (as a whole) or of array elements.*
- **(End.)**