Open book exam. Duration: 1h30m. First midterm exam

```
prog foo (n, t) {
1: push 0
2: dup
3: store r
4: store i
5: load i
6: load n
7: ifg 17
8: load r
9: load i
10: add
11: store r
12: load i
13: load t
14: add
15: store i
16: goto 5
17: load r
18: return
}
```

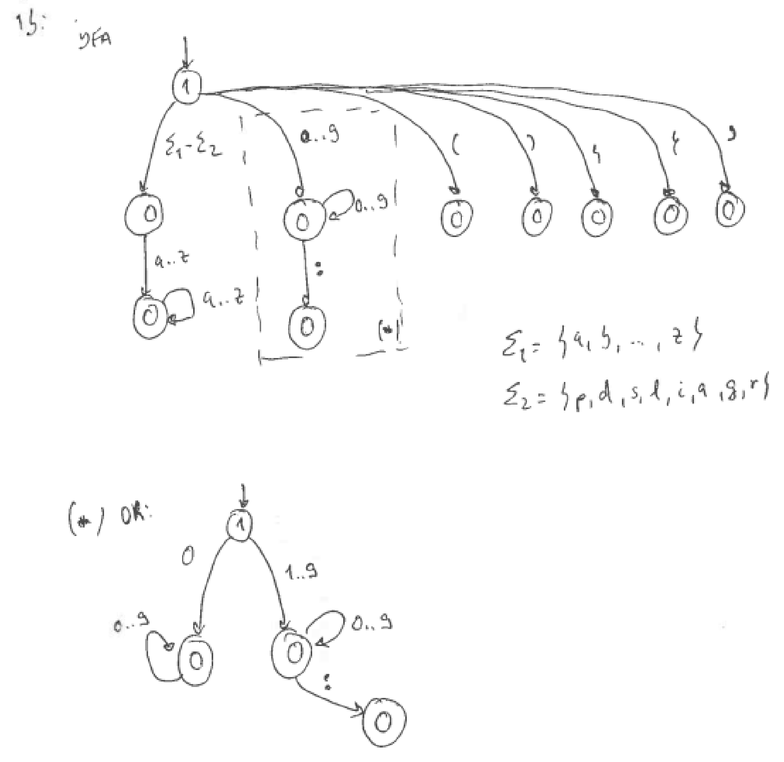
Grupo 1. (9 points)

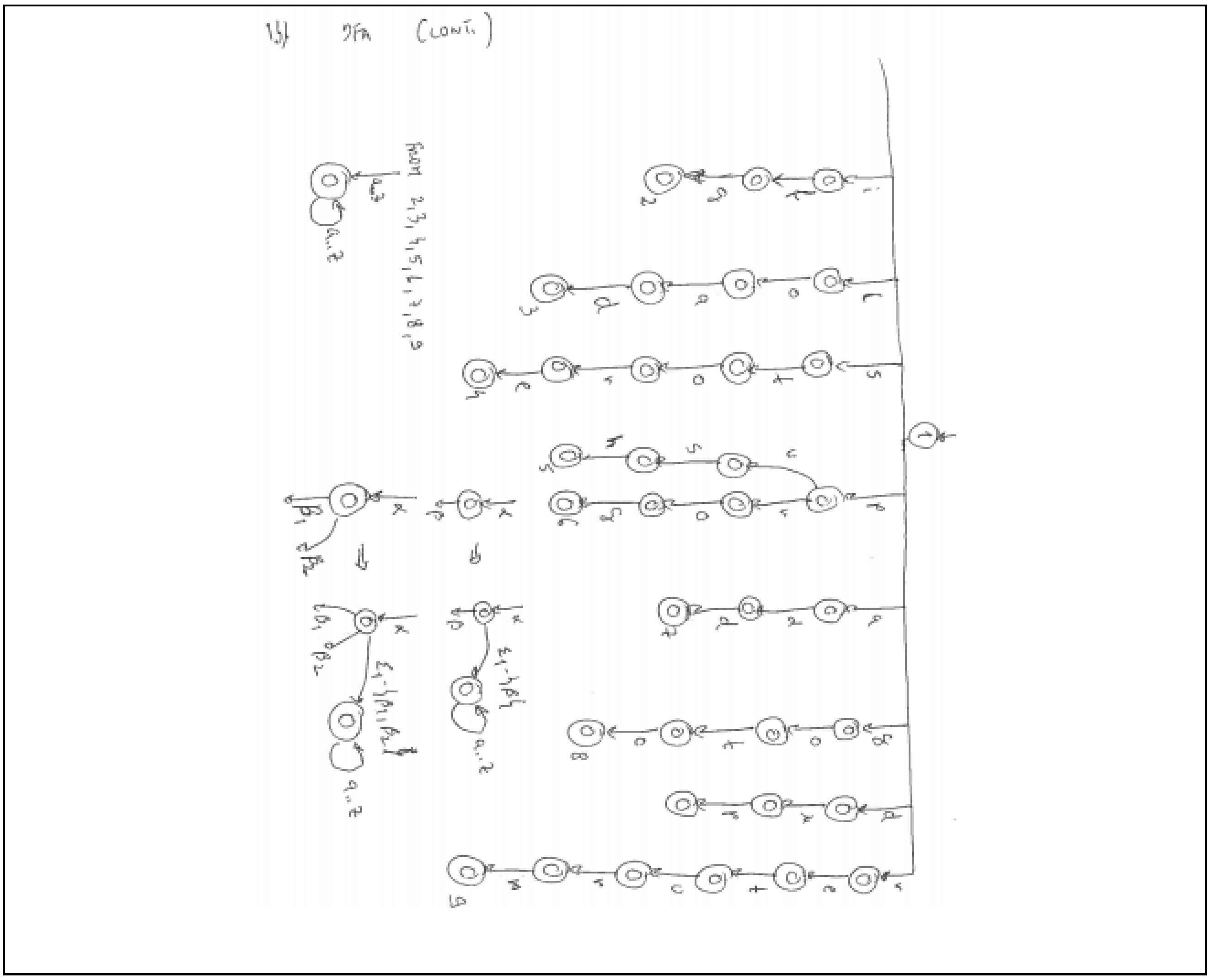
Consider the code example presented on the left. Suppose this code includes all the instructions of a stack-based machine and *n*, *t*, and the letters after the instructions *load* and *store* identify variables. Assume the variables can be identified by a letter from the alphabet.

1.a) [1pt] Present the definitions of the terminals for the language.

PUSH="push"	RETURN="return"	COMMA=","
DUP=dup"	LINENUM=[0-9]+":"	OPEN1="("
STORE=store"	PROG="prog"	CLOSE1=")"
LOAD=load"	NAME=[a-z]+	OPEN2="{"
IFG=ifg"	VAR=[a-z]	CLOSE2="}"
ADD="add"		CONST=[0-9]+
GOTO="goto"		

1.b) [2pt] Draw a DFA for the lexical analysis.





1.c) [3pt] Considering the definitions of the terminals of a), give a CFG (*context free grammar*) which accepts sequences of instructions of the machine. Note that the presented code on the left is an example of a sequence of instructions for the machine.

Start → PROG NAME OPEN1 Params CLOSE1 OPEN2 Statements CLOSE2
Params → VAR {COMMA VAR}
Statements → {LINENUM Inst}
Inst → PUSH CONST | DUP | STORE VAR | LOAD VAR | IFG CONST | ADD | GOTO CONST | RETURN

1.d) [2pt] Modify the grammar to accept programs on this language, but without the numbering of the instructions and using *labels* for the conditional and unconditional branches.

LABEL = [a-z][0-9a-z]*
COLON=":"

Start → PROG NAME OPEN1 Params CLOSE1 OPEN2 Statements CLOSE2
Params → VAR {COMMA VAR}
Statements → {<u>(LABEL COLON)?</u> Inst}
Inst → PUSH CONST | DUP | STORE VAR | LOAD VAR | IFG <u>LABEL</u> | ADD | GOTO <u>LABEL</u> | RETURN

1.e) [1pt] Can we consider that the code presented is in a high-level intermediate representation? Justify your answer.

No, because the code:
- does not reflect the constructs of typical programming languages such as FOR, WHILE and IF-THEN.
- uses low-level information according to the target machine (use of stack-based instructions).

Grupo 2. (11 points)

Consider the following grammar and example.

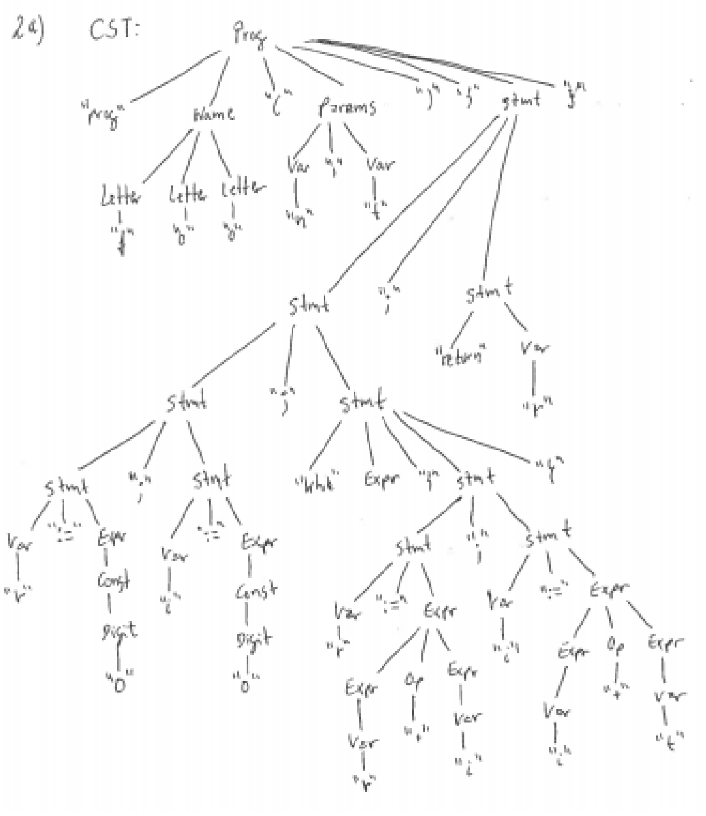
Grammar

Prog	::= “prog” Name “(“ Params “)” “{“ Stmt “}”
Params	::= Var {“,” Var}
Expr	::= Const | Var | Expr Op Expr
Op	::= “+” | “<=”
Stmt	::= Var “:=” Expr
	| Stmt “;” Stmt
	| “if” Expr “then” Stmt “else” Stmt
	| “while” Expr “{“ Stmt “}”
	| “return” Var
Var	::= “r” | “t” | “n” | “i” | “j” | “x” | “y” | “z” | “u”
Const	::= Digit { Digit }
Digit	::= “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”
Name	::=Letter { Letter }
Letter	::=”a” | “b” | … | “z”

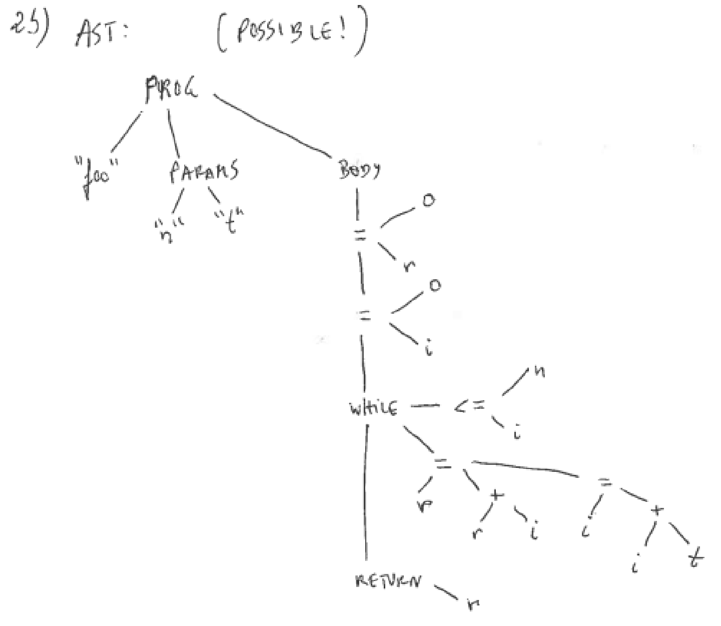
Example

```
prog foo (n, t) {
  r := 0 ;
  i := 0 ;
  while i <= n {
    r := r+i ;
    i := i+t
  };
  return r
}
```

2.a) [1pt] Show a possible concrete syntax tree obtained for the example using *leftmost derivation*.



2.b) [1pt] Show a possible AST (*abstract syntax tree*) for the example.



2.c) [1pt] This grammar can be implemented by an LL(k) syntactic analyser? If yes, give the value of k it needs to use.

No.
(The grammar has left recursion.)

2.d) [4pt] Write a grammar specifying the same language, without left recursivity, and possible to be implemented with an LL(k) syntactic analyser. (Note: you only need to present the productions that must be modified or added)

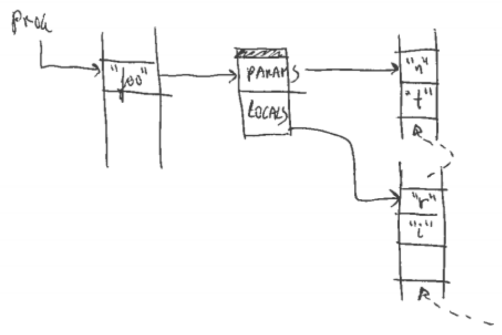
Original	Modified
Expr ::= Const | Var | Expr Op Expr	Expr ::= **Term (Op Expr)?** **Term** ::= Const | Var
Stmt ::= Var ":=" Expr | Stmt ";" Stmt | "if" Expr "then" Stmt "else" Stmt | "while" Expr "{" Stmt "}" | "return" Var	Stmt ::= **Stmt1** (";" Stmt)? **Stmt1** ::= Var ":=" Expr | "if" Expr "then" Stmt "else" Stmt | "while" Expr "{" Stmt "}" | "return" Var

2.e) [1pt] Show the modifications needed to apply to the original grammar in order the programs in that language can have zero or more parameters.

Original	Modified
Params ::= Var {"," Var}	Params ::= Var {"," Var} Params ::= ε or: Params ::= (Var {"," Var})?

2.f) [1pt] Assuming that the variables used in a program can only be parameters or local variables, show a possible symbol table for the example.

2.f) POSSIBLE SYMBOL TABLE:



2.g) [2pt] Describe possible semantic verifications that can be included in the semantic analysis for this language and the way they can be implemented.

Identification if a variable used has been initialized before. This would require a traverse from the beginning to the end (by the order the instructions are in the program) of the intermediate representation (IR), checking at each use of a variable, if this variable has been previously initialized. During the traversing of the IR, we can use an hierarchical symbol table to mark initialization of variables and to lookup for each use. Variables not initialized before entering to an "IF" scope are only considered initialized after the "IF" if they are initialized in all IF branches. During the traversing of the IR, the initialization of variables, when not in the scope of WHILE constructs, should be propagated to all the higher levels of the table symbol hierarchy.

(End.)