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;
}
```

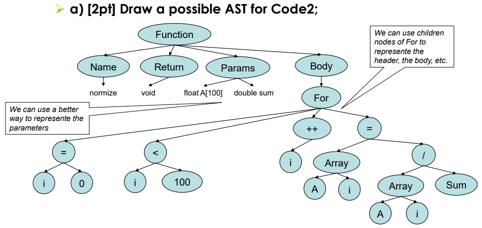

Group 2. AST, Symbol Table and High-Level Representation (7 pts)

- 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;
}
```

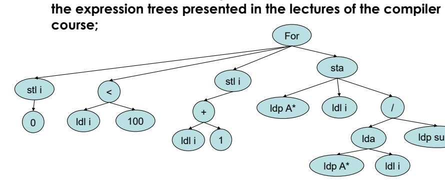

Group 2. AST, Symbol Table and High-Level Representation (7 pts)

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



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

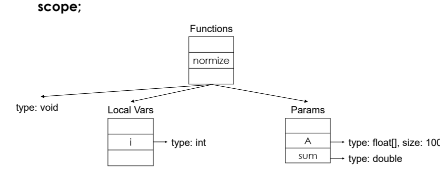
- **b) [2pt] Draw a possible high-level representation based on the expression trees presented in the lectures of the compiler course;**



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

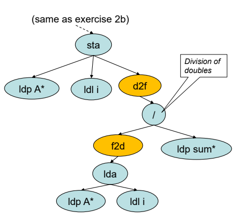
Group 2. AST, Symbol Table and High-Level Representation (7 pts)

- **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)

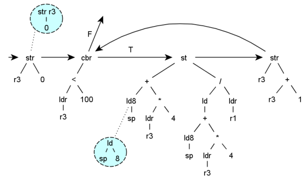
Code1:
```
for(i=0; i < 100; i++)
    A[i] = A[i]/sum;
```

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)

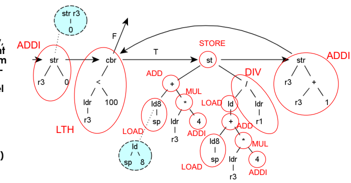
- **a) [3pt] Indicate the LLIR for the section of the example Code1 based on the LLIR presented in the lectures of the course, which is based on expression trees, and the type and storage of the variables given by the table below.**



* 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)

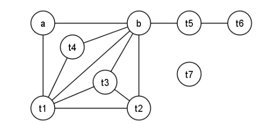
- **b) [4pt] Consider a target machine with 32 32-bit registers (R0 stores 0) including the instructions presented in the table below, where Ri, Rk, and Rj represent registers of the machine (from R0 to R31), C represents a 16-bit signed integer constant, and label1 represents a label identifying the 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)

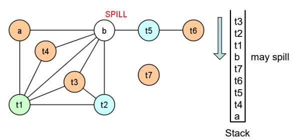
- **a) [2pt] By only inspecting the code, present the interference graph for Code2, which would result from liveness analysis.**

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.**

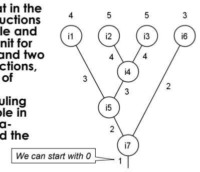


R1={a, t4, t3, t7, t6}
R2={t2, t5}
R3={t1}

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
```



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.)**

Possible Answers for Final Exam (June 21, 2017)

Compilers course

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

U.PORTO
João M. P. Cardoso
Universidade do Porto FEUP Faculdade de Engenharia

Dep. de Engenharia Informática
Faculdade de Engenharia (FEUP), Universidade do Porto,
Porto, Portugal
Email:jmpc@acm.org

PART I (10 PTS)

Group 1. (5 pts)

- Considering the following grammar (parentheses and numbers on the right identify the productions):
  - S → xB | yC (1, 2)
  - B → Aa (3)
  - C → Aba (4)
  - A → b | ε (5, 6)
- **1a) [1pt] Indicate the First and Follow sets for the grammar variables.**

First(S)={x, y}
First(B)={a,b}
First(C)={b}
First(A)={b} // {b, ε}

Follow(S)={}
Follow(B)={}
Follow(C)={}
Follow(A)={b,a}

Group 1. (5 pts)

- Considering the following grammar (parentheses and numbers on the right identify the productions):
  - S → xB | yC (1, 2)
  - B → Aa (3)
  - C → Aba (4)
  - A → b | ε (5, 6)
- **1b) [1pt] Indicate if this grammar is LL(1) and show the LL(1) parsing table.**

	x	y	a	b
S	S → xB	S → yC		
B			B → Aa	B → Aa
C				C → Aba
A			A → ε	A → b A → ε

The gramar is not LL(1) as there is at least a cell in the LL(1) parsing table with conflicts

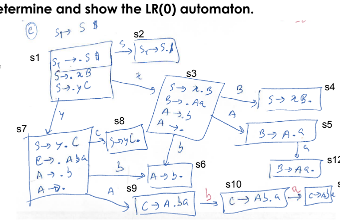
Group 1. (5 pts)

- Considering the following grammar (parentheses and numbers on the right identify the productions):
  - S → xB | yC (1, 2)
  - B → Aa (3)
  - C → Aba (4)
  - A → b | ε (5, 6)
- **1c) [1pt] Determine and show the LR(0) automaton.**

Group 1. (5 pts)

- **1c) [1pt] Determine and show the LR(0) automaton.**

Add:
S1 → S $
To the gramar (other option would be to add $ to end of every rule in S)



Group 1. (5 pts)

- Considering the following grammar (parentheses and numbers on the right identify the productions):
  - S → xB | yC (1, 2)
  - B → Aa (3)
  - C → Aba (4)
  - A → b | ε (5, 6)
- **1d) [1pt] Is the grammar LR(0)? Justify your answer indicating the LR(0) parsing table.**

Group 1. (5 pts)

- **1d) [1pt] Is the grammar LR(0)? Justify your answer indicating the LR(0) parsing table.**

State	x	y	a	b	$	S	B	C	A
s1	shift s3	shift s7	error	error	error	goto s2			
s2	error	error	error	error	accept				
s3	red. (6)	red. (6)	red. (6)	red. (6) shift s6	red. (6)		Goto s4		Goto s5
s4	red. (1)	red. (1)	red. (1)	red. (1)	red. (1)				
s5	error	error	shift s12	error	error				
s6	red. (5)	red. (5)	red. (5)	red. (5)	red. (5)				
s7	red. (6)	red. (6)	red. (6)	shift s6 red. (6)	red. (6)			Goto s8	Goto s9
s8	red. (2)	red. (2)	red. (2)	red. (2)	red. (2)				
s9				shift s10					
s10			Shift s11						
s11	red. (4)	red. (4)	red. (4)	red. (4)	red. (4)				
s12	red. (3)	red. (3)	red. (3)	red. (3)	red. (3)				

The grammar is not LR(0). There are shift/reduce conflicts in the LR(0) parsing table.

Group 1. (5 pts)

- **1e) [1pt] The following grammar is not LL(k). Indicate changes to the grammar that make it LL(k).**
  - ID = [a-z][0-9a-z]*
  - S → E = E ;
  - S → E ;
  - E → E * E
  - E → E + E
  - E → E (E)
  - E → ID

New grammar:
S → E S1
S1 → ; | = E;
E → ID E1
E1 → ε | + E E1 | * E E1 | (E) E1

Group 2. (3 pts)

- The goal is to define a programming language with a grammar that forces applying certain semantic rules. A team designing the language finds itself discussing the possibilities to verify if the data type defined as return in a function header coincides with the data type in the *return* instructions used in the code of the same function.
  - **2a) [1pt] Indicate the necessary aspects and possible restrictions that need to be taken into account in terms of such programming language to make the semantic rules mentioned above implemented by the grammar;**
  - **2b) [2pt] Indicate possible grammar rules that illustrate the way those semantic rules can be implemented considering the int, float, and double data types. Notice that it is only necessary to indicate sections of the grammar that illustrate the way the semantic rules can be implemented by the grammar.**

Group 2. (3 pts)

- **2a) [1pt] Indicate the necessary aspects and possible restrictions that need to be taken into account in terms of such programming language to make the semantic rules mentioned above implemented by the grammar;**

- Possibilities:
  - Declaration of the return variable
  - Use of a reserved word for the variable to be return and forcing by grammar rules the adequate type
  - Identifiers with the identification of the type
  - Enforcing the use of casts associated with the return type

Group 2. (3 pts)

- **2b) [2pt] Indicate possible grammar rules that illustrate the way those semantic rules can be implemented considering the int, float, and double data types. Notice that it is only necessary to indicate sections of the grammar that illustrate the way the semantic rules can be implemented by the grammar.**

Example using the declaration of a specific return variable:
FunctionHeader int → FunctionName(…) { … DeclareRetVar OtherStatements Return }
DeclareRetVar → int ret;
OtherStatements → …
Return → return ret;
…
FunctionHeader double → FunctionName(…) {… DeclareRetVar OtherStatements Return }
DeclareRetVar → double ret;

Group 3. (2 pts)

- **3a) [2pt] Comment the following sentence: "The high level intermediate representation obtained right after semantic analysis is just an AST with variable identifiers replaced by the access type (e.g., *load* array, *store* local variable, *load* local variable, *load* function parameter)."**

- The sentence is generally false as the IR after semantic analysis may have, according to the programming language, operations regarding the conversion between types and thus might be very different from the AST (even in the cases this IR is based on tree of expressions).

PART II (10 PTS)

Group 4. (8 pts)

- **4a) [2pt] Indicate the low level intermediate representation (LLIR) for the section of the code in the example below (where N represents a 32-bit *int* constant) based on expression trees, but considering the *Jouette*+ processor (instruction set presented in annex), and the type and storage of the variables given by the following table.**

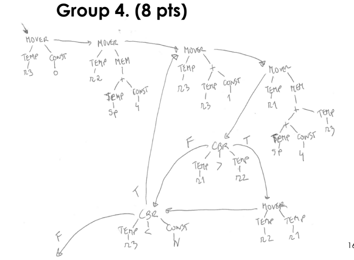
```
//in={}
1 i = 0;
2 m = A[0];
do {
3    i=i+1;
4    x = A[i];
5    if (x > m)
6        m = x;
7 } while(i<N);
// out = {m}
```

Variable	Type	Storage
A	int8 A[N] (array of integers (8-bit))	array stored in the stack starting at SP + 4
i	int i (32-bit scalar variable)	Variable stored in register r3
m	int m (32-bit scalar variable)	Variable stored in register r2
x	int x (32-bit scalar variable)	Variable stored in register r1

Group 4. (8 pts)

- **4a)**

```
//in={}
1 i = 0;
2 m = A[0];
do {
3    i=i+1;
4    x = A[i];
5    if (x > m)
6        m = x;
7 } while(i<N);
// out = {m}
```



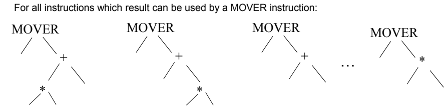
Group 4. (8 pts)

- **4b) [2pt] In this LLIR for the *Jouette*+ processor, code of the type A = B + C, supposing that R2, R3, and R4 are the registers that store A, B and C, respectively, is represented by two instructions, such as illustrated by the following example: MOVER: R2 ← R5; ADD: R5 ← R3 + R4. However, it is possible to make the selection of instructions resulting in the direct generation (i.e., without optimizations and/or elimination of instructions after selection) of the code ADD: R2 ← R3 + R4. In order to do this, indicate the pattern trees of the IR that make the selection of instructions more efficient in the context of the *Jouette*+ processor.**

Group 4. (8 pts)

- **4b)**

For all instructions which result can be used by a MOVER instruction:



Group 4. (8 pts)

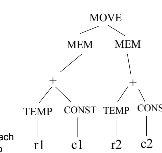
- **4c) [2pt] Taking into account the costs per instruction of the *Jouette*+ processor presented in the table below, indicate an example in which the Maximal Munch algorithm is unable to achieve a selection of instructions that corresponds to the minimum global cost. Indicate which would be the minimum global cost for that example and indicate how the use of dynamic programming can obtain a selection of instructions with that cost.**

Instruction	Cost	Instruction	Cost	Instruction	Cost	Instruction	Cost
MADD	5	LOAD	3	ADD	2	MOVEM	4
MUL	3	STORE	2	ADDI	1	Other instructions	1

Group 4. (8 pts)

- **4c)**

The example corresponding to: M[r1+c1] = M[r2+c2]
(another example: M[0] = M[r1])



Maximal Munch ⇒ 4+1+1 = 6 (cost)
Dynamic Programming ⇒ 2+3 = 5 (cost)

Dynamic Programming:
Starting at the bottom and keeping the best selection for each node (or each tile for the subtree with that node as root) allows dynamic programming to select a LOAD and a STORE instead of the MOVEM and two ADDI's. (solution can use the example and show the steps of the algorithm)

Instruction	Cost	Instruction	Cost	Instruction	Cost	Instruction	Cost
MADD	5	LOAD	3	ADD	2	MOVEM	4
MUL	3	STORE	2	ADDI	1	Other instructions	1

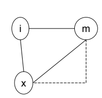
Group 4. (8 pts)

- **4d) [2pt] Draw the interference graph for the local scalar variables used in the section of code presented above by direct inspection. Indicate a possible allocation of registers to variables using the graph coloring algorithm explained in class and supposing the utilization of 2 registers (*R1*, e *R2*). Show the content of the stack immediately after the simplification of the interference graph. In the case of having to perform *spilling*, specify an efficient criterion that you suggest for the selection of variables for *spilling* and the order of selection determined by that criterion. Show the result of the first graph coloring (i.e., without repeating the process).**

4d)

Possible cost function to select the variable(s) to spill:

```
//in={}
1 i = 0;
2 m = A[0];
do {
3    i=i+1;
4    x = A[i];
5    if (x > m)
6        m = x;
7 } while(i<N);
// out = {m}
```



Variable	#use	#def	#load	#store	Total (#load+ #store)	Total/(#def+ #use)
i	3	2	3N	1+N	1+4N	(1+4N)/5
x	2	1	2N	N	3N	N
m	1	2	N	1+N	1+2N	(1+2N)/3

Ordering to spill m, then i, then x

Top of stack
i
x
may-spill m

R1={i}
R2={x}
Spill m

Note: Other possibilities can use the dynamic range (minimum to maximum) considering the evaluation of the condition i<N OR probabilities of the condition to be evaluated as true and as false (how?).

Group 5. (2 pts)

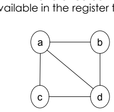
- **5a) [2pt] Suppose there is a need to pack variables in the same register, even if the lifetime of those variables interferes (for instance, 32-bits register can store the value of two 16-bits variables). Assuming 32-bits registers and variables with 8, 16 e de 32-bits data types, indicate what would have to be changed in the register allocations based in graph coloring so that packaging is used to reduce the number of registers required. Use the examples that you think are adequate to illustrate that process.**

- One possibility is to keep the simplification process and to modify the assignment of variables to registers based on the possible slots of contiguous bytes free in a 32-bit register. Now two variables with interference can be assigned to the same register if there are slots available in the register to store the two variables. Example:

Group 5. (2 pts)

- **5a)**
  - One possibility is to keep the simplification process and to modify the assignment of variables to registers based on the possible slots of contiguous bytes free in a 32-bit register. Now two variables with interference can be assigned to the same register if there are slots available in the register to store the two variables. Example:

Assume:
a, c: 8-bit variables
b: 16-bit variable
d: 32-bit variable
and the interference graph on the right.
Let's do register allocation considering 2 32-bit registers.
The stack is obtained after the simplification process.
We now start coloring.
Pop d and assign it to R1=1111 (1 identifies a byte occupied and 0 a byte free in the register)
Pop c and assign it to R2=0001
Pop b and assign it to R2=0011
Pop a and assign it to R2=0100



Top of stack
d
c
b
mayspill-a

Exercises About LL(K) Grammars

Compilers course

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

João M. P. Cardoso

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Dep. de Engenharia Informática
Faculdade de Engenharia (FEUP)
Universidade do Porto,
Porto, Portugal
Email:jmpc@acm.org

Exercise 1

E → E + T | T
*T → T * F | F*
F → (E) | NUM

- **Group 1. *First*, *Follow* Sets and *Parsers* LL(k) (5 pts)**
- Consider the CFG1 in the left. **NUM** is a terminal symbol representing numbers.
- **1.a)** [1pt] Give the *First* and *Follow* sets for each grammar variable;
- **1.b)** [2pts] Is the grammar LL(1)? Show the table for the *parser* LL(1);
- **1.c)** [2pts] Modify the grammar in order it can be implemented as a top-down recursive parser with K=1 (*lookahead*).

Source: First midterm exam (MT1): April 6, 2017

Exercise 1

E → E + T | T
*T → T * F | F*
F → (E) | NUM

- **1.a)** [1pt] Give the *First* and *Follow* sets for each grammar variable;
- First(E)={(,NUM}
- First(T)={(,NUM}
- First(F)={(,NUM}
- Follow(E)={+,)}
- Follow(T)={*,+,)}
- Follow(F)={*,+,)}

Source: First midterm exam (MT1): April 6, 2016

Exercise 1

Note that here you need to determine the First of each production (not the first of each variable) and the follow of the variables that have empty productions.

E → E + T | T
*T → T * F | F*
F → (E) | NUM

- **1.b)** [2pts] Is the grammar LL(1)? Show the table for the *parser* LL(1);
  - The grammar is not LL(1), there are cases with more than one production for a given input symbol as can be seen below

	+	*	(	)	NUM
E			E → E + T E → T		E → E + T E → T
T			T → T * F T → F		T → T * F T → F
F			F → (E)		F → NUM

Source: First midterm exam (MT1): April 6, 2016

Exercise 1

What we need?
- Eliminate ambiguity
- Eliminate left recursion
- Do left factorization (may need to consider modifications of productions across different variables)

- **1.c)** [2pts] Modify the grammar in order it can be implemented as a top-down recursive parser with K=1 (*lookahead*).

E → E + T | T
*T → T * F | F*
F → (E) | NUM

The grammar is non-ambiguous

Source: First midterm exam (MT1): April 6, 2016

Exercise 1

What we need?
- Eliminate ambiguity
- Eliminate left recursion
- Do left factorization (may need to consider modifications of productions across different variables)

- **1.c)** [2pts] Modify the grammar in order it can be implemented as a top-down recursive parser with K=1 (*lookahead*).

E → E + T | T
*T → T * F | F*
F → (E) | NUM

The grammar is non-ambiguous

1st: eliminate left recursion

E → T + E | T
*T → F * T | F*
F → (E) | NUM

2nd: eliminate for each variable the existence of productions with First sets with one or more equal symbols

E → T E1
E1 → + E | ε
T → F T1
*T1 → * T | ε*
F → (E) | NUM

E → T [+ E]
T → F [T]*
F → (E) | NUM

Source: First midterm exam (MT1): April 6, 2016

Exercise 2

E → T E1
E1 → + E | ε
T → F T1
*T1 → * T | ε*
F → (E) | NUM

- **1d)** Show the grammar is LL(1) by using the table for the *parser* LL(1)

Exercise 2

- **1d)** Show the grammar is LL(1) by using the table for the *parser* LL(1)
  - The table presented below has at most one production per cell and thus the grammar is LL(1)

E → T E1
E1 → + E | ε
T → F T1
*T1 → * T | ε*
F → (E) | NUM

	+	*	(	)	NUM
E			E → T E1		E → T E1
E1	E1 → + E				
T			T → F T1		T → F T1
T1	T1 → ε	T1 → * T			
F			F → (E)		F → NUM

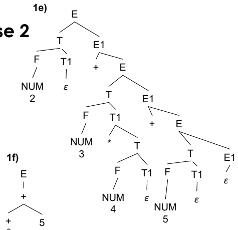
Source: First midterm exam (MT1): April 6, 2016

Exercise 2

E → T E1
E1 → + E | ε
T → F T1
*T1 → * T | ε*
F → (E) | NUM

- **1e)** Show the concrete syntax tree (CST) for 2+3*4+5
- **1f)** Show a possible abstract syntax tree (AST) for 2+3*4+5

1e)



1f)