Family Name: .... **Other Names:** ....

Student ID: .... **Signature** ....

COMP 261 : Test 1

16 March 2023, ** WITH SOLUTIONS **

Instructions

- Time allowed: **50 minutes**
- Attempt **all** the questions. There are 50 marks in total.
- **In-person**: Write your answers in this test paper and hand in all sheets.
  Remote: Type your answers in the template file and submit to "Test 1 Remote" on the COMP 261 submission system.
- If you think a question is unclear, ask for clarification.
- This test contributes 10% of your final grade.
- You may use dictionaries and calculators.
- You may write notes and working on this paper, but make sure your answers are clear.

Questions		Marks	
1.	Grammars and Parse Trees	[10]	
2.	Abstract Syntax Trees.	[5]	
3.	Coding a Parser	[15]	
4.	Printing an Abstract Syntax Tree	[10]	
5.	LL(1) grammars and recursive descent parsing	[10]	
		TOTAL:	

Question 1. Grammars and Parse Trees **[10 marks]**

Consider the following grammar that describes a made-up language for specifying filters.

In this grammar

- Non-terminals are in uppercase; terminals are enclosed in quotation marks,
- `|` means OR.
- `[...]+` means one or more repetitions of what is in the brackets.
- `SIGNAL` matches any terminal that is a single letter followed by a single digit, such as "a1".
- `NUM` matches any terminal that is a non-negative integer.

```
FILTER ::= [ SPEC ]+
SPEC ::= MULTI | STATE
MULTI ::= "many"  NUM  SPEC  "ynam"
STATE ::= "(("  [ SIGNAL ]+  "))"
SIGNAL ::= matches "[a-z][0-9]"
NUM ::= matches "[0-9]+"
```

(a) **[5 marks]**

The following three sentences are almost, but not quite valid sentences of the grammar above. For each sentence, circle the first token where a parser could identify the error.

(i) `many 4 (( a3 ab )) ynam`

(ii) `(( d1 d2 d3 )) many a1 (( a2 )) ynam`

(iii) `many 1 many (( a3 b7 )) ynam ynam`

(Question 1 continued on next page)

(Question 1 continued)

(b) **[5 marks]** Draw the Concrete Parse Tree of the following filter according to the grammar above.

```
(( a1 )) many 4  many 3 ((  b2  b3  )) ynam ynam
```

```
               _________FILTER____________
              /                           \
        SPEC                              SPEC
         |                                 |
        STATE                       _____MULTI______
      /   |     \                  /    /      \      \
  "((" SIGNAL "))"         "many"  "4"   SPEC  "ynam"
          |                                 |
        "a1"                        ______MULTI_____
                                   /      /    |     \
                             "many"  "3"   SPEC  "ynam"
                                              |
                                     ______STATE______
                                    /     /       \      \
                                  ((  SIGNAL  SIGNAL  ))
                                        |        |
                                      "b2"     "b3"
```

Stage 1 grammar from Assignment 1 (RoboGame):

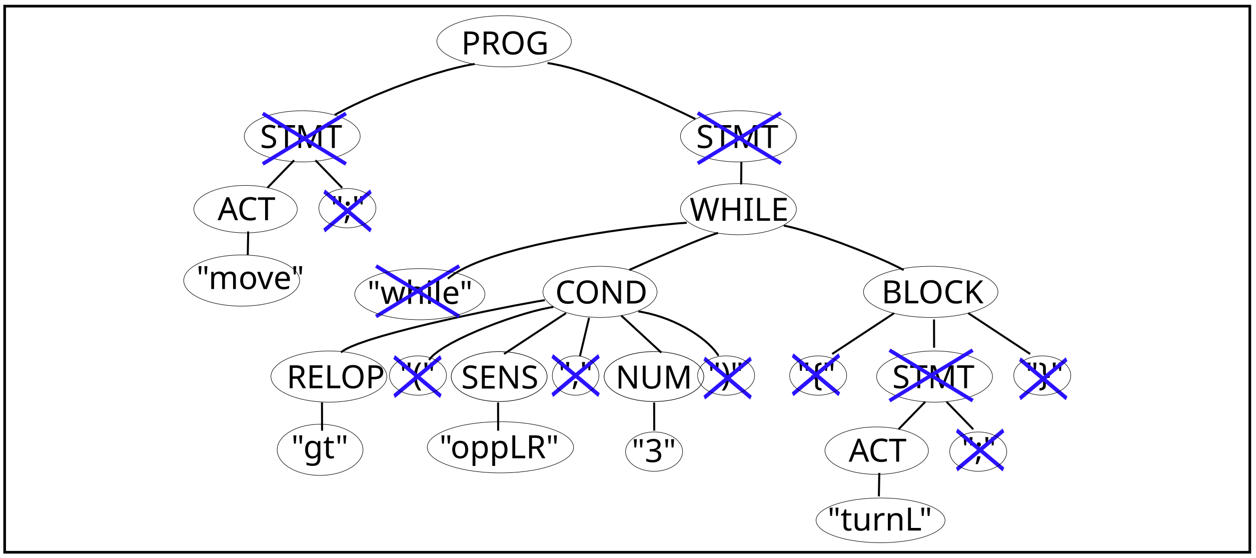
```
PROG  ::= [ STMT ]*
STMT  ::= ACT ";" | LOOP | IF | WHILE
ACT   ::= "move" | "turnL" | "turnR" | "turnAround" | "shieldOn" |
          "shieldOff" | "takeFuel" | "wait"
LOOP  ::= "loop" BLOCK
IF    ::= "if" "(" COND ")" BLOCK
WHILE ::= "while" "(" COND ")" BLOCK
BLOCK ::= "{" STMT+ "}"
COND  ::= RELOP "(" SENS "," NUM ")
RELOP ::= "lt" | "gt" | "eq"
SENS  ::= "fuelLeft" | "oppLR" | "oppFB" | "numBarrels" |
          "barrelLR" | "barrelFB" | "wallDist"
NUM   ::= "-?[1-9][0-9]*|0"
```

SPACE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

Question 2. Abstract Syntax Trees. **[5 marks]**

Consider the following Concrete Parse Tree of a program for the Robot in assignment 1.
(The Stage 1 grammar is given on the previous page.)



Identify nodes of the Concrete Parse Tree which could be removed to leave an Abstract Syntax Tree by drawing an X over the unnecessary nodes in the diagram.

Give a brief explanation of why removing those nodes does not lose any information that would be needed to either print out or to execute the program.

The tokens that are removed do not carry any information -
they are just there to help specify the structure
The STMT nodes that are removed are unnecessary because
they only contain a single subnode and there is no additional
information attached to the STMT node

Question 3. Coding a Parser **[15 marks]**

Suppose you are writing a parser for the grammar in question 1 which should return an Abstract Syntax Tree of FNodes, or throw an exception if a program is invalid.

Your parser program (below) already includes some constants, parseSignal(..) and parseNum(..) methods, utility methods (require(..) and fail(..)), and classes defining the different kinds of FNodes.

You are to complete the three methods on the next page (in Java, not pseudocode).

```
//-------- constants (patterns) -------------------
    static final Pattern MANY_PAT = Pattern.compile("many");
    static final Pattern YNAM_PAT = Pattern.compile("ynam");
    static final Pattern LEFT_PAT = Pattern.compile("\\(\\("); // matches ((
    static final Pattern RIGHT_PAT = Pattern.compile("\\)\\)"); // matches ))
    static final Pattern SIG_PAT  = Pattern.compile("[a-z][0-9]");

//-------- parse... methods -------------------
     :
    public int parseNum(Scanner s){
        if (s.hasNext("[0-9]")) {return s.nextInt();}
        fail ("Expecting integer"); return -1;
    }
    public FNode parseSignal(Scanner s){
        if (s.hasNext(SIG_PAT)) {return new SignalNode(s.next());}
        fail ("Expecting signal"); return null;
    }
//-------- Utility methods -------------------
    public void require(Pattern pat, Scanner s){
        if (s.hasNext(pat)) {s.next(); return;}
        fail ("expecting "+ pat);
    }
    public void fail (String msg){ System.out.println(msg); throw new RuntimeException(msg);}

//-------- Node classes -------------------
    interface FNode{}

    class FilterNode implements FNode{
        final List<FNode> specs;
        public FilterNode( List<FNode> spcs){specs=spcs;}
    }
    class MultiNode implements FNode{
        final int num;
        final FNode spec;
        public MultiNode(int n, FNode spc){num=n; spec=spc;}
    }
    class StateNode implements FNode{
        final List<FNode> signals;
        public StateNode(List<FNode> sigs){signals=sigs;}
    }
    class SignalNode implements FNode{
        final String signalName;
        public SignalNode(String sig){signalName=sig;}
    }
```

(Question 3 continued on next page)

(Question 3 continued)

Complete the parseSpec(..), parseMulti(..), and parseState(..) methods below:
(Note: the AST does not need SPEC nodes.)

```
/** Parses the rule:   SPEC ::= MULTI |  STATE          */
public FNode parseSpec(Scanner s){
    if (s.hasNext(MANY_PAT)){return parseMulti(s);}
    if (s.hasNext(LEFT_PAT)){return parseState(s);}
    fail("invalid SPEC");
    return null;
}
/** Parses the rule:   MULTI ::= "many" NUM SPEC "ynam"          */
public FNode parseMulti(Scanner s){
    require(MANY_PAT, s);
    int num = parseNum(s); // or s.nextInt();
    FNode spec = parseSpec(s);
    require(YNAM_PAT, s);
    return new MultiNode(num, spec);
}
/** Parses the rule:   STATE ::= "((" [ SIGNAL ]+ "))"          */
public FNode parseState(Scanner s){
    List<FNode> signals = new ArrayList<FNode>();
    require(LEFT_PAT, s);
    do {
        signals.add(parseSignal(s));
    } while (!s.hasNext(RIGHT_PAT));    // or s.hasNext(SIG_PAT);
    require(RIGHT_PAT, s);
    return new StateNode(signals);
}
```

Grammar and example filter specification from question 1 repeated for convenience:

Grammar:

```
FILTER ::= [ SPEC ]+
SPEC ::= MULTI | STATE
MULTI ::= "many"  NUM  SPEC  "ynam"
STATE ::= "(("  [ SIGNAL ]+  "))"
SIGNAL ::= matches "[a-z][0-9]"
NUM ::= matches "[0-9]+"
```

Example filter:

```
(( a1 )) many 4  many 3 ((  b2  b3  )) ynam ynam
```

Question 4. Printing an Abstract Syntax Tree. **[10 marks]**

The parser in question 3 also needs toString() methods for the four node classes below so that the filter specification in an Abstract Syntax Tree of FNodes could be printed out in the same syntax as specified in the grammar. (The grammar is repeated on the previous page.)

Note: the String returned by toString() does not need to include newlines or indentation.

```
class FilterNode implements FNode{
    final List<FNode> specs;
    public FilterNode( List<FNode> spcs){specs=spcs;}
    public String toString(){
        String ans = "";
        for (FNode spec : specs) {ans = ans+spec.toString()+" ";}
        return ans;



    }
}
class MultiNode implements FNode{
    final int num;
    final FNode spec;
    public MultiNode(int n, FNode spc){num=n; spec=spc;}
    public String toString(){
        return "many "+num+" "+spec.toString()+" ynam";

    }
}
class StateNode implements FNode{
    final List<FNode> signals;
    public StateNode(List<FNode> sigs){signals=sigs;}
    public String toString(){
        String ans = "(( ";
        for (FNode node : signals) {ans = ans+node.toString()+" ";}
        return ans+"))";



    }
}
class SignalNode implements FNode{
    final String signalName;
    public SignalNode(String sig){signalName=sig;}
    public String toString(){
        return signalName;


    }
}
```

Question 5. LL(1) grammars and recursive descent parsing **[10 marks]**

Ambiguous grammars (where a text can have multiple different parse trees) cannot be parsed by deterministic top-down recursive descent parsers, like the parsers described in the lectures.

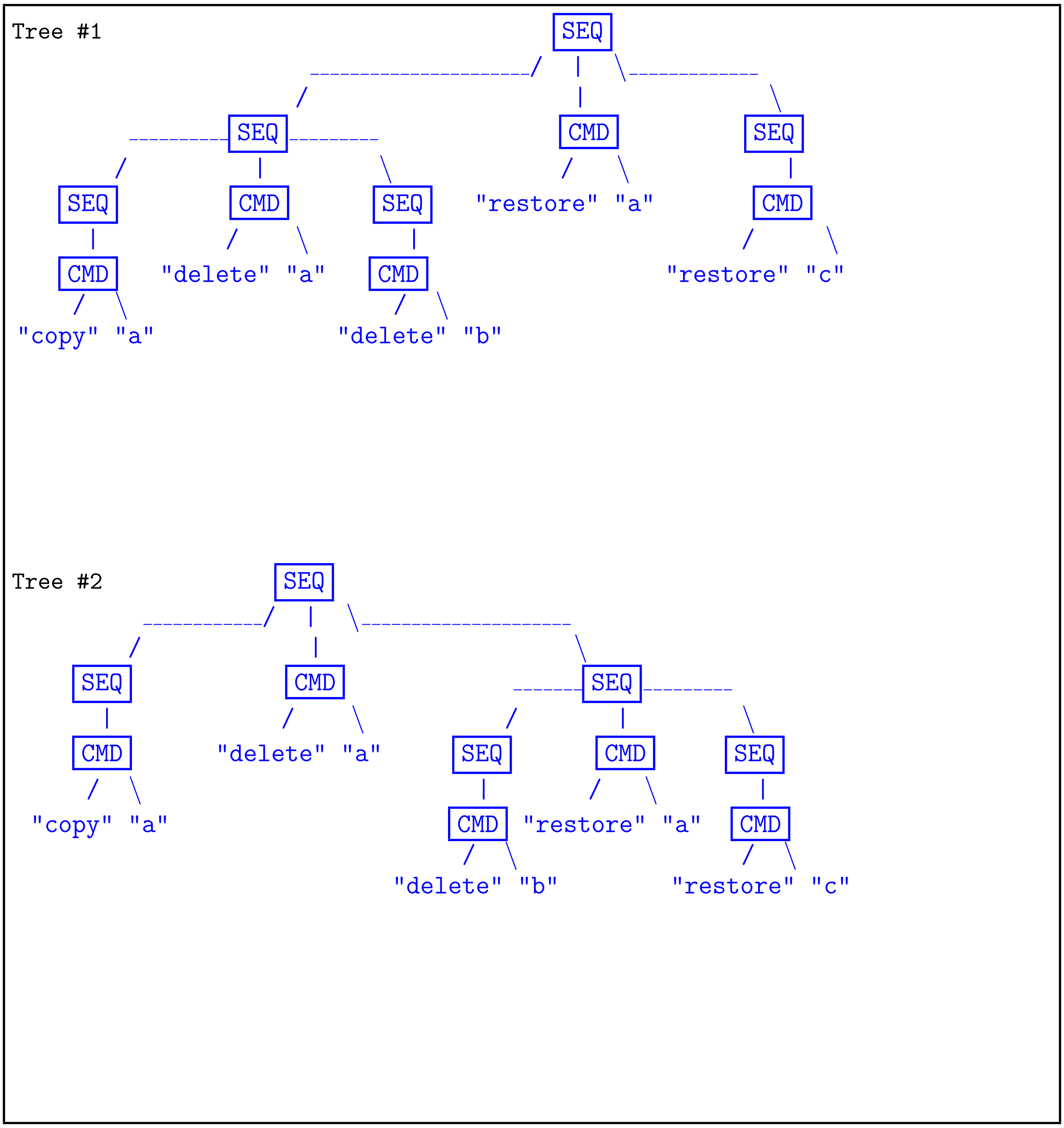
In some cases, it is possible to rewrite the rules of a grammar so that it describes the same language but is no longer ambiguous.

The following grammar for sequences of file commands is ambiguous.

```
SEQ  ::=  CMD | SEQ CMD SEQ
CMD  ::=  "copy" FILE | "delete" FILE | "restore" FILE
FILE ::=  matches "[a-z]"
```

(a) **[3 marks]** Draw the two alternative parse trees of the following sequence according to this grammar:

`copy a delete a delete b restore a restore b`



(Question 5 continued on next page)

(Question 5 continued)

(b) **[2 marks]** Rewrite the rule for `SEQ` (and any additional rules you need) so that the grammar covers the same language but is no longer ambiguous.

```
SEQ   ::= CMD | CMD CMD SEQ
   or ::= CMD [ CMD CMD ]*


CMD ::= "copy" FILE | "delete" FILE | "restore" FILE
FILE ::= matches "[a-z]"
```

(c) **[2 marks]** The following grammar for Instructions is not LL(1) and cannot be parsed by a deterministic recursive descent parser with just 1 token look ahead.

```
INSTR     ::= SELDIR "backup" | SELDRIVE "copy"
SELDIR    ::= "select" DIR
SELDRIVE ::= "select" DRIVE
DIR       ::= matches "[a-z/]+/"
DRIVE     ::= matches "[A-Z]:"
```

Explain briefly what makes this grammar not LL[1].

The initial token of both SELDIR and SELFILE is "select" so
the parser cannot tell which of the two options of INSTR will apply
based on the next token

(d) **[3 marks]** Rewrite the grammar for "Instructions" so that it still describes the same language, but is now LL(1). (you do not need to write the rules for `DIR` and `DRIVE`).

```
INSTR ::=  "select" ARG
ARG ::= DIR "backup" | DRIVE "copy"


DIR ::= matches "[a-z/]+/"
DRIVE ::= matches "[A-Z]:"
```

* * * * * * * * * * * * * * *

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.