NAME: ____________________________

STUDENT ID: ____________________________

SIGNATURE: ____________________________

The University of New South Wales

Final Examination

June 2022

COMP3131/COMP9102

Programming Languages and Compilers

Time allowed: **2 hours**

Total number of questions: **5**

Answer **all** questions

The questions are **not** of equal value

Marks for this paper total **100**

This paper may **not** be retained by the candidate

No examination materials

Answers must be written in ink.

Question 1. Regular Expressions and Finite Automata *[15 marks]*

Consider the following regular expression:

$$1(0|1)^*$$

(a) Use **Thompson's construction** to convert this regular expression into an NFA.

[6 marks]

(b) Use the **subset construction** to convert the NFA of (a) into a DFA.

[6 marks]

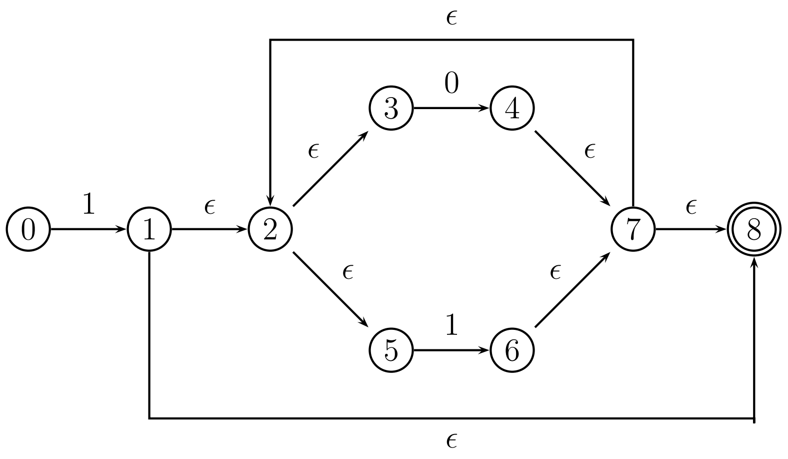
(c) Convert the DFA of (b) into a minimal-state DFA.

[3 marks]

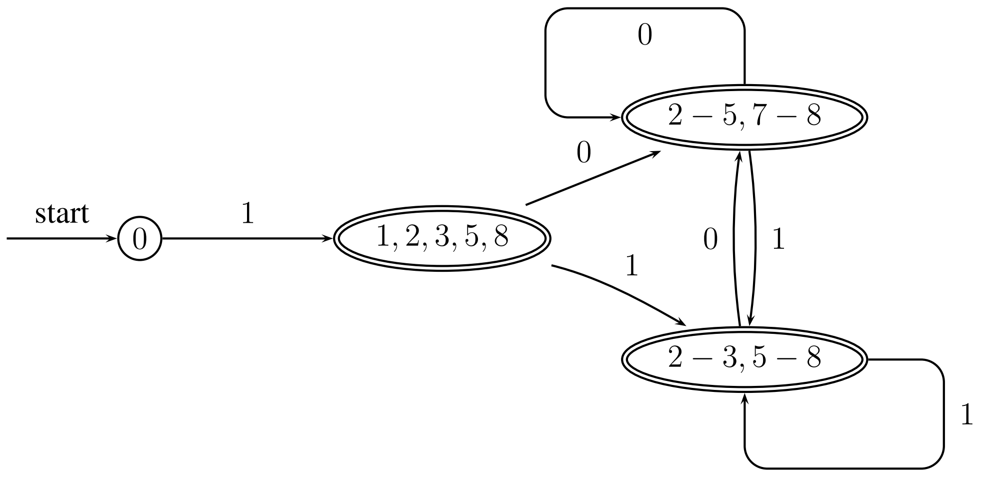
You are required to apply exactly Thompson's construction algorithm in (a) and the subset construction algorithm in (b) to solve those two problems.

********* ANSWER *********

(a)

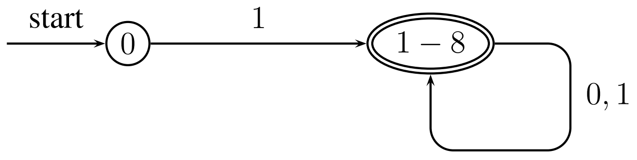


(b)



(c)

The minimal-state DFA is:



Question 2. Context-Free Grammars *[20 marks]*

Consider the following context-free grammar for describing arithmetic expressions:

$$
\begin{array}{lcl}
\textit{expr} & \rightarrow & \textit{term}\ \mathbf{+}\ \textit{expr} \ \mid \textit{term} \\
\textit{term} & \rightarrow & \textit{factor}\ \mathbf{*}\ \textit{term} \mid \textit{term}\ \mathbf{/}\ \textit{factor} \ \mid \textit{factor} \\
\textit{factor} & \rightarrow & \mathbf{INTLITERAL}
\end{array}
$$

where the the non-terminals are given in *italics* and the terminals in **boldface** (including '+', '*', '/', and **INTLITERAL** (representing integer constants)).

(a) Write a leftmost derivation for the sentence $5 + 4 * 3 \ / \ 2$.

[3 marks]

(b) Draw a parse tree for this sentence.

[3 marks]

(c) If the operators **+**, ***** and **/** represent the operations of integer addition, integer multiplication and integer (truncated) division, respectively, what would be the value implied by your parse tree found in (b)?

[4 marks]

(d) Is this grammar ambiguous? Justify your answer.

[4 marks]

(e) Answer the following true or false questions about this grammar:

1. **+** is always right-associative.
2. ***** is always left-associative.
3. **/** is always right-associative.
4. **+** must always have higher precedence than ***** and **/**.
5. ***** may have higher precedence than **/**.
6. **/** may have higher precedence than *****.

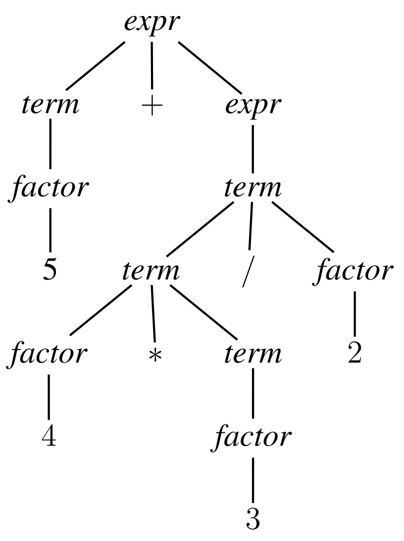
[6 marks with 1 mark for each true/false question]

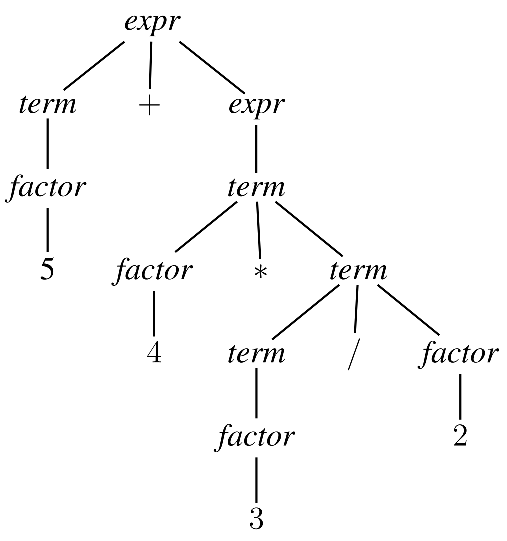
********* ANSWER *********

(a) Two different leftmost derivations:

$$\begin{aligned}
\textit{expr} &\Rightarrow_{\mathrm{lm}} \textit{term} + \textit{expr}\\
&\Rightarrow_{\mathrm{lm}} \textit{factor} + \textit{expr}\\
&\Rightarrow_{\mathrm{lm}} 5 + \textit{expr}\\
&\Rightarrow_{\mathrm{lm}} 5 + \textit{term}\\
&\Rightarrow_{\mathrm{lm}} 5 + \textit{term} \,/\, \textit{factor}\\
&\Rightarrow_{\mathrm{lm}} 5 + \textit{factor} * \textit{term} \,/\, \textit{term}\\
&\Rightarrow_{\mathrm{lm}} 5 + 4 * \textit{term} \,/\, \textit{term}\\
&\Rightarrow_{\mathrm{lm}} 5 + 4 * \textit{factor} \,/\, \textit{term}\\
&\Rightarrow_{\mathrm{lm}} 5 + 4 * 3 \,/\, \textit{factor}\\
&\Rightarrow_{\mathrm{lm}} 5 + 4 * 3 \,/\, 2
\end{aligned}
\qquad
\begin{aligned}
\textit{expr} &\Rightarrow_{\mathrm{lm}} \textit{term} + \textit{expr}\\
&\Rightarrow_{\mathrm{lm}} \textit{factor} + \textit{expr}\\
&\Rightarrow_{\mathrm{lm}} 5 + \textit{expr}\\
&\Rightarrow_{\mathrm{lm}} 5 + \textit{term}\\
&\Rightarrow_{\mathrm{lm}} 5 + \textit{factor} * \textit{term}\\
&\Rightarrow_{\mathrm{lm}} 5 + 4 * \textit{term}\\
&\Rightarrow_{\mathrm{lm}} 5 + 4 * \textit{term} \,/\, \textit{factor}\\
&\Rightarrow_{\mathrm{lm}} 5 + 4 * \textit{factor} \,/\, \textit{factor}\\
&\Rightarrow_{\mathrm{lm}} 5 + 4 * 3 \,/\, \textit{factor}\\
&\Rightarrow_{\mathrm{lm}} 5 + 4 * 3 \,/\, 2
\end{aligned}$$

(b) Two corresponding parse trees:





(c) The values for the two parse trees are: 11 and 9.

(d) YES due to the existence of two different leftmost derivations or parse trees.

(e) 1. TRUE 2. FALSE 3. FALSE 4. FALSE 5. TRUE 6. TRUE

Question 3. Recursive Descent Parsing *[25 marks]*

Consider the following context-free grammar:

$$\begin{array}{lccl} 1. & S & \rightarrow & SS* \\ 2. & S & \rightarrow & SS+ \\ 3. & S & \rightarrow & \textbf{INT} \\ 4. & S & \rightarrow & \textbf{ID} \end{array}$$

where S is the non-terminal, and ‘*****’, ‘**+**’, **INT** (representing integer constants) and **ID** (representing identifiers) are the four terminals.

(a) Eliminate the left recursion in the grammar.

[5 marks]

(b) Do left-factorisation of the grammar produced in (a).

[5 marks]

(c) Compute the FIRST and FOLLOW sets for every non-terminal in the grammar produced in (b).

[6 marks]

(d) Construct an LL(1) parsing table for the grammar produced in (b), based on the FIRST and FOLLOW sets computed in (c).

[5 marks]

(e) The sentence $4\ x + *$ is NOT syntactically legal (since it is NOT in the language defined by the grammar). Explain concisely how $4\ x + *$ can be detected by an LL(1) table-driven parser for the language, by showing the moves of the parser on this sentence based on the LL(1) parsing table produced in (d), as shown in Week 9’s Wednesday Lecture:

```
Stack              Input                  Production
$                  INT ID + *
```

[4 marks]

********* ANSWER *********

(a)

```
S  --> INT S′
S  --> ID S′
S′ --> S⋆S′
S′ --> S+S′
S′ --> ϵ
```

(b)

```
1. S  --> INT S′
2. S  --> ID S′
3. S′ --> S A
4. S′ --> ϵ
5. A  --> ⋆S′
6. A  --> +S′
```

(c)

	FIRST	FOLLOW
S	{INT, ID}	{+, ⋆, $}
S′	{INT, ID, ϵ}	{+, ⋆, $}
A	{⋆, +}	{+, ⋆, $}

(d)

	INT	ID	⋆	+	$
S	1	2			
S′	3	3	4	4	4
A			5	6	

(e)

Stack	Input	Production	Left Derivation
$S	INT ID + *	S -> INT S’	
$S’ INT	INT ID + *	pop & scanner	
$S’	ID + *	S’ --> S A	
$A S	ID + *	S --> ID S’	
$A S’ ID	ID + *	pop & scanner	
$A S’	+ *	S’ --> epsilon	
$A	+ *	A --> + S’	
$S’ +	+ *	pop & scanner	
$S’	*	S’ --> epsilon	
$	*	error	

Question 4. Attribute Grammars *[25 marks]*

Consider the following context-free grammar:

program	$\rightarrow$	**func** *stmt-list*
stmt-list	$\rightarrow$	*stmt stmt-list*
stmt-list	$\rightarrow$	*stmt*
stmt	$\rightarrow$	**for ID = INTLITERAL upto INTLITERAL begin** *stmt-list* **end**
stmt	$\rightarrow$	*assignment*

where the non-terminals are given in *italics* and the terminals in **boldface**.

This grammar describes a simple language, which allows iterating through a sequence of statements, ultimately assignments. A **for**-loop is specified by the range of its loop variable (identified by **ID**), given by integer constants (identified by **INTLITERAL**), representing its lower and upper bounds. In other words, a **for**-loop, with its lower and upper bounds being L and U, respectively, will execute its loop body, i.e., *stmt-list* exactly $U - L + 1$ times. No production for assignments is given (as it is irrelevant here). Therefore, assignments, represented by *assignment*, are treated here as terminals.

A sample program is given below (with its loop executed exactly 20 times):

```
func
assignment
for i = 1 upto 20
begin
  assignment
  assignment
  assignment
end
assignment
```

(a) Write an attribute grammar that determines for each assignment how many times the assignment will be executed when running the program. You can assume that for each loop, its lower bound is no larger than its upper bound (so that you do not have to check this in your solution).

[20 marks]

(b) Describe whether each attribute used is synthesised or inherited.

[5 marks]

This question can be a bit involved, as students need to figure out to use an inherited attribute to solve this problem.

********* ANSWER *********

(a)

```
program -> func stmt-list
{
  program.t = 1
  prog.t = 1
  stmt-list.t = 1
}

stmt-list1 -> stmt stmt-list2
{
   stmt.t = stmt-list1.t
   stmt-list2.t = stmt-list1.t
}

stmt-list -> stmt
{
  stmt.t = stmt-list.t
}

stmt -> for ID = INTLITERAL1 upto INTLITERAL2 begin stmt-list end
{
stmt-list.t = stmt.t x (INTLITERAL2.val - INTLITERAL1.val + 1)
}

stmt -> assignment
{
        assignment.t = stmt.t
}
```

(b) Inherited

Question 5. Code Generation *[15 marks]*

Suppose we introduce a **do-while** (loop) statement into our VC language:

do stmt **while** "(" expr ")"

where *expr* and *stmt* are defined exactly as in our VC grammar. Note that how these two nonterminals are defined is irrelevant to this question.

The **do-while** statement has exactly the same semantics as that in C/C++/Java. It executes *stmt* repeatedly until the value of `expr` is `false`.

Suppose we use the following AST class to represent a **do-while** statement in the AST representation of a VC program:

```
package VC.ASTs;

import VC.Scanner.SourcePosition;

public class DoWhiletmt extends Stmt {

  public Expr E;
  public Stmt S;

  public DoWhileStmt (Expr eAST, Stmt sAST, SourcePosition Position)
  {
    super (Position);
    E = eAST;
    S = sAST;
    E.parent = S.parent = this;
  }


  public Object visit(Visitor v, Object o)
  {
    return v.visitDoWhileStmt(this, o);
  }
}
```

Write Emitter.visitDoWhileStmt in Java for generating Jasmin code for the **do-while** statement by using the visitor design pattern as you did in Assignment 5 (for translating the **for** and **while** loops in VC).

You do not need to include code for computing the operand stack size required. However, you are required to include code to generate appropriate labels that can be used for translating any **break** or **continue** statement that may be contained in a **do-while** loop, similarly as you did in Assignment 5.

********* ANSWER *********

```
public Object visitDoWhileStmt(DoWhileStmt ast, Object o) {

        Frame frame = (Frame)o;

        String firstLabel = frame.getNewLabel();
        String secondLabel = frame.getNewLabel();

        frame.conStack.push(firstLabel);
        frame.brkStack.push(secondLabel);

        emit(firstLabel + ":");

        ast.S.visit(this, o);

        ast.E.visit(this, o);
        emit(JVM.IFEQ, secondLabel);
        emit(JVM.GOTO, firstLabel);
        emit(secondLabel + ":");

        frame.conStack.pop();
        frame.brkStack.pop();

        return null;
}
```