NAME: _____

STUDENT ID: _____

SIGNATURE: _____

The University of New South Wales

Final Examination

November/December 2002

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

Answer Questions 1 – 2 in one book and Questions 3 – 5 in another book

No examination materials

Answers must be written in ink.

Question 1. Consider the following regular expression:

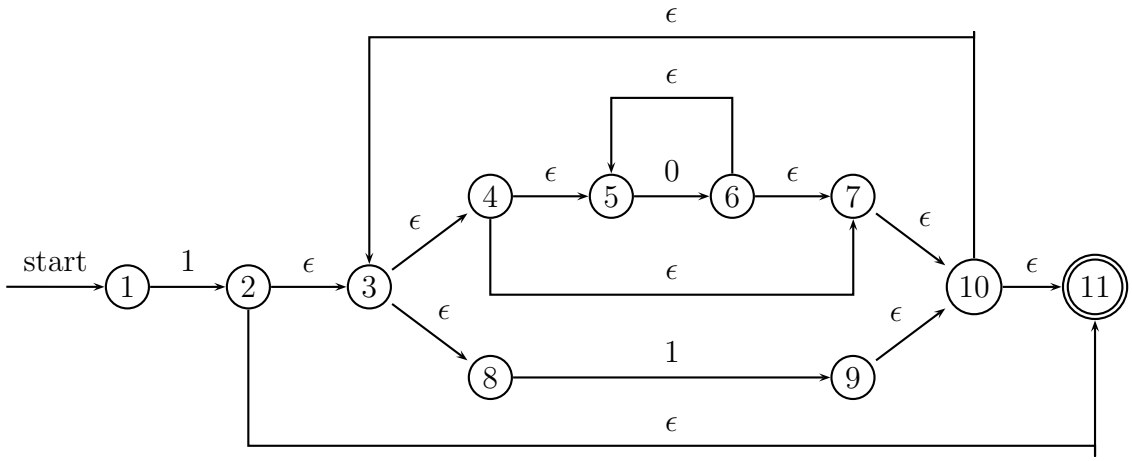
$$1\,(\,0^*\mid 1\,)^*$$

(a) Use *Thompson's construction algorithm* to obtain the NFA equivalent to the regular expression shown above.

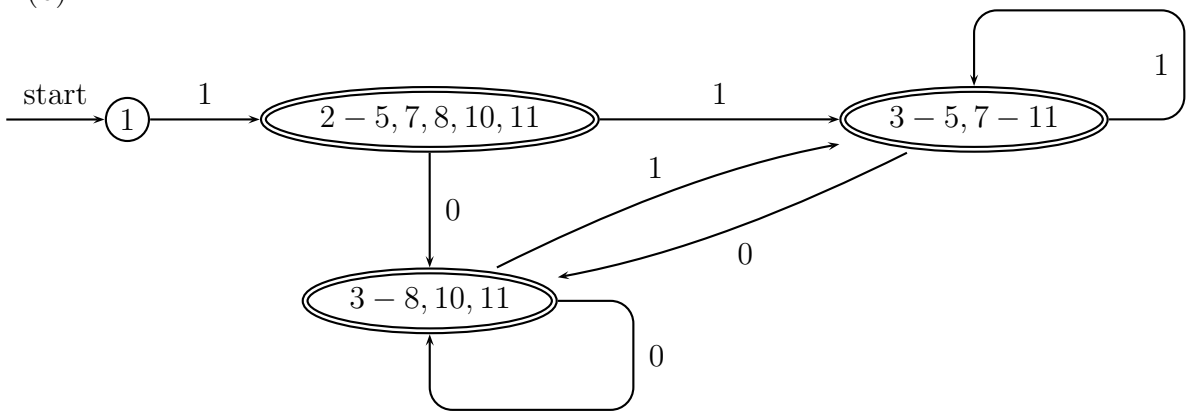
(b) Use the *subset construction* to obtain the DFA equivalent to the NFA found in (a). You need to indicate the NFA states that each DFA state contains.

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

(a)



(b)



Question 2. Consider the following CFG for describing numerical expressions:

$$\begin{array}{lcl} \mathit{expr} & \rightarrow & \mathit{term} \uparrow \mathit{expr} \mid \mathit{term} \\ \mathit{term} & \rightarrow & \mathit{factor} * \mathit{term} \mid \mathit{term} \,/\, \mathit{factor} \mid \mathit{factor} \\ \mathit{factor} & \rightarrow & \textbf{INTLITERAL} \end{array}$$

(a) Write a leftmost derivation for the sentence $3 \uparrow 2 * 6/4$.

(b) Draw a parse tree for this sentence.

(c) If the operators $\uparrow$, $*$ and / represent the operations of raising to a power, integer multiplication and integer (truncated) division, respectively, what would be the value implied by your parse tree found in (b)?

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

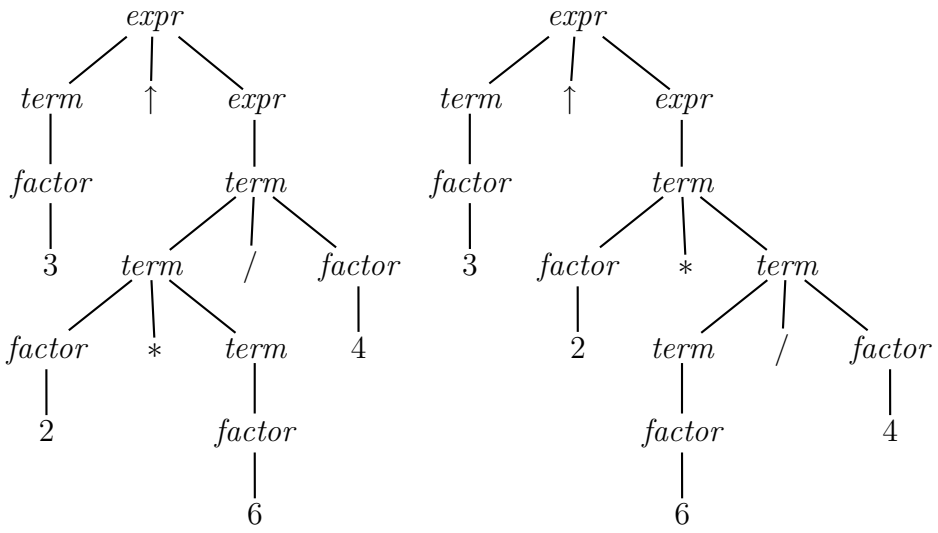
********* ANSWER *********

(a) Two different leftmost derivations:

$$\begin{array}{lll} \mathit{expr} & \Rightarrow_{\text{lm}} & \mathit{term} \uparrow \mathit{expr} \\ & \Rightarrow_{\text{lm}} & \mathit{factor} \uparrow \mathit{expr} \\ & \Rightarrow_{\text{lm}} & 3 \uparrow \mathit{expr} \\ & \Rightarrow_{\text{lm}} & 3 \uparrow \mathit{term} \\ & \Rightarrow_{\text{lm}} & 3 \uparrow \mathit{term} \,/\, \mathit{factor} \\ & \Rightarrow_{\text{lm}} & 3 \uparrow \mathit{factor} * \mathit{term} \,/\, \mathit{term} \\ & \Rightarrow_{\text{lm}} & 3 \uparrow 2 * \mathit{term} \,/\, \mathit{term} \\ & \Rightarrow_{\text{lm}} & 3 \uparrow 2 * \mathit{factor} \,/\, \mathit{term} \\ & \Rightarrow_{\text{lm}} & 3 \uparrow 2 * 6 \,/\, \mathit{factor} \\ & \Rightarrow_{\text{lm}} & 3 \uparrow 2 * 6 \,/\, 4 \end{array}$$

$$\begin{array}{lll} \mathit{expr} & \Rightarrow_{\text{lm}} & \mathit{term} \uparrow \mathit{expr} \\ & \Rightarrow_{\text{lm}} & \mathit{factor} \uparrow \mathit{expr} \\ & \Rightarrow_{\text{lm}} & 3 \uparrow \mathit{expr} \\ & \Rightarrow_{\text{lm}} & 3 \uparrow \mathit{term} \\ & \Rightarrow_{\text{lm}} & 3 \uparrow \mathit{factor} * \mathit{term} \\ & \Rightarrow_{\text{lm}} & 3 \uparrow 2 * \mathit{term} \\ & \Rightarrow_{\text{lm}} & 3 \uparrow 2 * \mathit{term} \,/\, \mathit{factor} \\ & \Rightarrow_{\text{lm}} & 3 \uparrow 2 * \mathit{factor} \,/\, \mathit{factor} \\ & \Rightarrow_{\text{lm}} & 3 \uparrow 2 * 6 \,/\, \mathit{factor} \\ & \Rightarrow_{\text{lm}} & 3 \uparrow 2 * 6 \,/\, 4 \end{array}$$

(b) Two corresponding parse trees:



(c) The values for the two parse trees are: 27 and 9.

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

NOTE: This question is designed to test their understanding about CFGs:

- parsing
- operator precedence and associativity implied by a grammar
- ambiguity

Question 3. Recall that LL(1) grammars describe a left-to-right reading of the input and produce a leftmost derivation. It turns out that not every one in the world reads from left to right! In this question, you will explore the class of RL(1) grammars. The parsers for these grammars read their input from right to left. Here are the LL(1) grammar rules discussed in class:

- No ambiguity
- No left recursion
- A grammar G is LL(1) if and only if whenever $A \to \alpha \mid \beta$ are two distinct productions of G, the following conditions hold:
  - for no terminal a do both α and β derive strings beginning with a (i.e., $\mathsf{FIRST}(\alpha)$ and $\mathsf{FIRST}(\beta)$ are disjoint)
  - at most one of α and β can derive the empty string
  - if $\beta \Rightarrow^* \epsilon$, then α cannot derive any string beginning with a terminal in $\mathsf{FOLLOW}(A)$ (i.e., $\mathsf{FIRST}(\alpha)$ and $\mathsf{FOLLOW}(A)$ are disjoint)

(a) List the rules for an RL(1) grammar. You may define new terminology if you find it useful.

(b) Consider our favorite grammar:

$$E \to E + E \mid E * E \mid (E) \mid \textsf{ID}$$

Modify this grammar so that the revised grammar is RL(1). In your revised grammar, + and $*$ are both left-associative and + has lower precedence than $*$.

(c) Calculate the select sets for all productions in your revised grammar found in (b) so that you are able to construct an RL(1) table-driven predictive parser.

(d) Construct the RL(1) parsing table for your revised grammar found in (b).

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

Let

$$\begin{aligned} \mathsf{LAST}(\alpha) &= \{a \mid S \Rightarrow^* \ldots a\} \\ \mathsf{PRECEDE}(A) &= \{a \mid S \Rightarrow^* \ldots aA \ldots\} \end{aligned}$$

(a)

- No ambiguity
- No **right** recursion
- A grammar G is **R**L(1) if and only if whenever $A \to \alpha \mid \beta$ are two distinct productions of G, the following conditions hold:
  - for no terminal a do both α and β derive strings **ending** with a (i.e., $\mathsf{LAST}(\alpha)$ and $\mathsf{LAST}(\beta)$ are disjoint)
  - at most one of α and β can derive the empty string
  - if $\beta \Rightarrow^* \epsilon$, then α cannot derive any string **ending** with a terminal in $\mathsf{PRECEDE}(A)$ (i.e., $\mathsf{LAST}(\alpha)$ and $\mathsf{PRECEDE}(A)$ are disjoint)

(b) We start with the following unambiguous grammar:

```
E -> E + T | T
T -> T * F | F
F -> ( E ) | ID
```

which defines correctly the precedence and associativity of the operators.

Both E and T have a common **suffix**. Eliminating these RL(1) parsing conflicts, we get the following RL(1) grammar:

1	E	$\to$	P T
2	P	$\to$	E +
3	P	$\to$	ϵ
4	T	$\to$	Q F
5	Q	$\to$	T *
6	Q	$\to$	ϵ
7	F	$\to$	(E)
8	F	$\to$	ID

(c)

Let $ mark the beginning of the sentence being parsed.

```
LAST(E)   = LAST(P T) = LAST(T) = LAST(Q F ) = LAST(F) = { ), ID }
LAST(P)   = LAST(E +) = { +, ϵ}
LAST(Q)   = LAST(F *) = { *, ϵ}
LAST(ID)  = { ID }
LAST((E)) = { ) }

PRECEDE(E) = PRECEDE(P) = { $, ( }
PRECEDE(T) = PRECEDE(Q) = LAST(P) U PRECEDE(E) = { +, $, ( }
PRECEDE(F) = LAST(Q) U PRECEDE(T) = { *, + , $, ( }

SELECT(E -> P T ) = LAST(T) = { ), ID }
SELECT(P -> E + ) = { + }
SELECT(P -> ϵ )   = PRECEDE(P) = { $, ( }
SELECT(T -> QF)   = LAST(F) = { ), ID }
SELECT(Q -> F * ) = { * }
SELECT(Q -> ϵ)    = PRECEDE(Q) = { +, $, ( }
SELECT(F -> (E) ) = { ) }
SELECT(F -> ID )  = { ID }
```

(d)

	+	*	(	)	ID	$
E				1	1	
P	2			3		3
T				4	4	
Q	6	5		6		6
F				7	8	

NOTE: This question tests their understanding about many concepts introduced in recursive-descent parsing, including:

- CFGs,
- first, follow and select sets
- LL(1) parsing conflicts
- table-driven parsing

Question 4. Consider the following expression grammar as it would be written for a predictive recursive-descent LL(1) parser with left recursion removed:

$$
\begin{array}{llll}
1 & \mathit{expr} & \rightarrow & \mathit{term}\ \mathit{expr}' \\
2 & \mathit{expr}' & \rightarrow & +\ \mathit{term}\ \mathit{expr}' \\
3 & \mathit{expr}' & \rightarrow & \epsilon \\
4 & \mathit{term} & \rightarrow & \mathit{factor}\ \mathit{term}' \\
5 & \mathit{term}' & \rightarrow & *\ \mathit{factor}\ \mathit{term}' \\
6 & \mathit{term}' & \rightarrow & \epsilon \\
7 & \mathit{factor} & \rightarrow & (\ \mathit{expr}\ ) \\
8 & \mathit{factor} & \rightarrow & \textsf{INTLITERAL}
\end{array}
$$

where the two operators + and ∗ are both left-associative.

Write an attribute grammar to compute the value of an expression derived by this expression grammar. You are **not** allowed to modify the the expression grammar given above.

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

Production	Semantic Rule
1 expr -> term expr'	{ expr.ptr = expr'.ptr = term.ptr }
2 expr1' -> + term expr2'	{ *expr2'.ptr = *expr1'.ptr + *term.ptr }
3 expr' -> ϵ	{ }
4 term -> factor term'	{ term.ptr = term'.ptr = factor.ptr }
5 term1' -> + factor term2'	{ *term2'.ptr = *term1'.ptr * *factor.ptr }
6 term' -> ϵ	{ }
7 factor -> (expr)	{ factor.ptr = expr.ptr }
8 factor -> INTLITERAL	{ factor.ptr = a pointer to INTLITERAL.val}

NOTE: The answer can be quite involved if non-pointer attributes are used. But the problem is still doable regardless.

Question 5. Suppose we introduced a **for** statement into our VC language:

for (`expr`$_1$; `expr`$_2$; `expr`$_3$) `stmt`

where *expr* and *stmt* are as defined in our VC grammar given in Appendix A.

This statement has the same semantics as in Java. It executes `expr`$_1$ (only once before the loop), then `expr`$_2$, *stmt* and `expr`$_3$ repeatedly until the value of `expr`$_2$ is `false`.

Suppose we used the following class to represent **for** statements in the AST:

```
/*============================ ForStmt.java  ======================  */
package VC.ASTs;

import VC.Scanner.SourcePosition;

public class ForStmt extends Stmt {

  public Expr E1;
  public Expr E2;
  public Expr E3;
  public Stmt S;

  public ForStmt(Expr e1AST, Expr e2AST, Expr e3AST, Stmt sAST,
                                              SourcePosition Position) {
    super (Position);
    E1 = e1AST;
    E2 = e2AST;
    E3 = e3AST;
    S = sAST;
    E1.parent = E2.parent = E3.parent = S.parent = this;
  }

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

Write Emitter.visitForStmt in Java for generating Jasmin code for the **for** statement.

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

```
public Object visitForStmt(ForStmt ast, Object o) {
  Frame frame = (Frame) o;
  String loopLabel = frame.getNewLabel();
  String testLabel = frame.getNewLabel();

  ast.E1.visit(this, o);

  // Discard the value of expr1 on top of stack, if any
  if (ast.E1 instanceof EmptyExpr)
     ;
  else if (ast.E1 instanceof AssignExpr)
     ;
  else if (ast.E instanceof CallExpr &&
     ((FuncDecl) ((CallExpr) ast.E).I.decl).T.equals(StdEnvironment.voidType))
     ;
  else
     frame.pop();

  emit(JVM.GOTO, testLabel);

  emit(loopLabel + ":");
  ast.S.visit(this, o);
  ast.E3.visit(this, o);

  emit(testLabel + ":");
  ast.E2.visit(this, o);
  emit(JVM.IFNE, loopLabel);

  return null;
}
```

NOTE: This question is quite straightforward. However, those who did not work **themselves** on their last programming assignment on code generation would usually get close to 0 marks.