Victoria University of Wellington

DEGREE EXAMINATIONS — 1998

COMP 202

END OF YEAR

COMP 202

Formal Methods of Computer Science

Time Allowed: 3 Hours

Instructions: Candidates should attempt **all** questions.

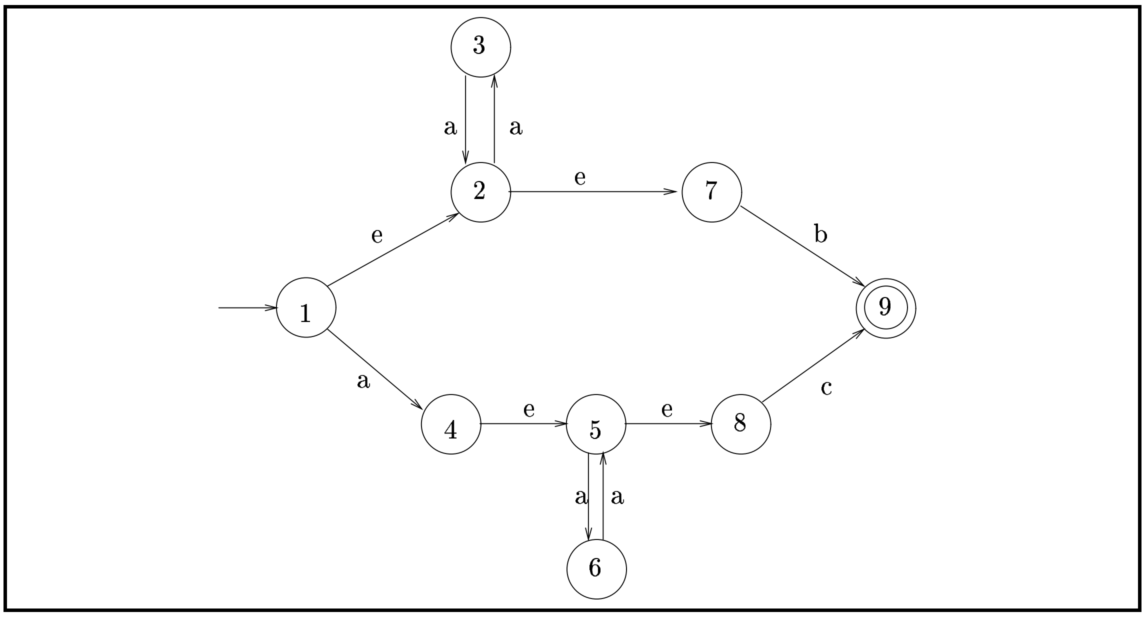
This exam will be marked out of 100.

Question 1. [15 marks]

Consider the regular expression $(aa)^*b \mid a(aa)^*c$.

(a) Construct a transition diagram for an NFA to recognise the language defined by this regular expression, using the "top-down construction". [5 marks]

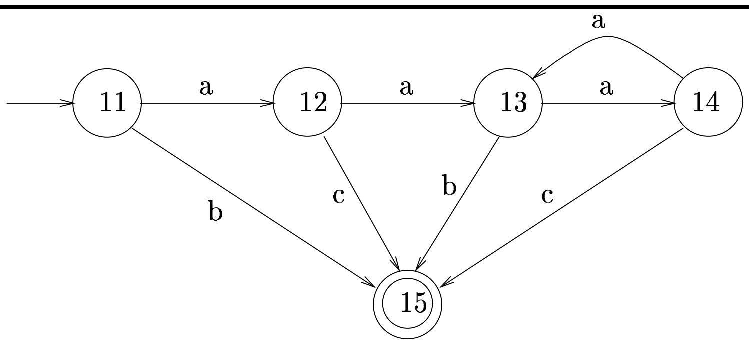
You do not need to show all of the steps in the construction, but you **should** show all of the states and transitions resulting from using the top-down construction.



(b) Construct a transition diagram for a DFA, equivalent to your NFA in part **(a)**, using the "subset construction".

Show the relationship between states in your DFA and those in the NFA.

[5 marks]



The correspondence is:

DFA	*NFA*
11	$\{1, 2, 6\}$
12	$\{3, 4, 5, 8\}$
13	$\{2, 6, 7\}$
14	$\{3, 5, 8\}$
15	$\{9\}$

(c) Write a Regular Grammar, equivalent to the original regular expression, based on your DFA in part **(b)**.

[5 marks]

$$\begin{array}{lll} S & \rightarrow & aT \mid bW \\ T & \rightarrow & aU \mid cW \\ U & \rightarrow & aV \mid bW \\ V & \rightarrow & AU \mid cW \\ W & \rightarrow & \lambda \end{array}$$

continued...

Question 2. [40 marks]

Suppose you wish to write a program to solve sets of algebraic equations. The first thing you need is a parser that allows you to read a list of equations and build a representation of them which your solver can operate on.

Suppose we define the syntax of equation lists as follows:

$$\begin{array}{lll} \textit{eq-list} & \rightarrow & \textit{eqn} \mid \textit{eq-list} \text{ ``;'' } \textit{eqn} \\ \textit{eqn} & \rightarrow & \textit{exp} \text{ ``='' } \textit{exp} \\ \textit{exp} & \rightarrow & \textit{id} \mid \textit{id} \text{ ``('' } \textit{exp-list} \text{ ``)''} \\ \textit{exp-list} & \rightarrow & \textit{exp} \mid \textit{exp-list} \text{ ``,'' } \textit{exp} \end{array}$$

where *id* is an identifier.

(a) Explain what it means for a grammar to be in LL(1) form, and why this property is important in the construction of recursive descent parsers. [5 marks]

A grammar is LL(1) if:

(i) For any two (distinct) rules $N \rightarrow \alpha$ and $N \rightarrow \beta$, $first(\alpha) \cap first(\beta) = \emptyset$, where $first(\gamma)$ is the set of terminals that start strings produced from γ.

(ii) If G contains a rule $N \rightarrow \alpha$, where $\alpha \Rightarrow^* \lambda$, then $first(\alpha) \cap follow(\alpha) = \emptyset$, where $follow(\alpha)$ is the set of terminals that can follow a string produced from N in a sentence.

These conditions are important, because ensure that a recursive descent parser can always chose the appropriate rule to use at every step, based on a one-symbol lookahead (i.e. just by looking a the next input symbol).

(b) Identify any places where the above grammar fails to meet the LL(1) conditions. [3 marks]

The rules for *eq-list*, *exp* and *exp-list* all break the first LL(1) condition. In each case the first sets for the two rules are identical.
Specifically (though not expected):

$$first(\textit{eqn}) = \{\textit{id}\} = first(\textit{eq-list} ; \textit{eqn})$$

$$first(\textit{id}) = \{\textit{id}\} = first(\textit{id} \ (\ \textit{exp-list} \ ))$$

$$first(\textit{exp}) = \{\textit{id}\} = first(\textit{exp-list} \ , \ \textit{exp})$$

(c) Write a (plain) CFG in LL(1) form, equivalent to the grammar above.

Show that your grammar is in LL(1) form, by constructing the necessary $first$ and $follows$ sets.

Explain why constructing an LL(1) grammar in this way often has undesirable consequences. [7 marks]

eq-list	$\rightarrow$	*eqn rest-eq-list*	(1)
rest-eq-list	$\rightarrow$	λ | ";" *eq-list*	(2,3)
eqn	$\rightarrow$	*exp* "=" *exp*	(4)
exp	$\rightarrow$	*id rest-exp*	(5)
rest-exp	$\rightarrow$	λ | "(" *exp-list* ")"	(6,7)
exp-list	$\rightarrow$	*exp rest-exp-list*	(8)
rest-exp-list	$\rightarrow$	λ | "," *exp-list*	(9,10)

This is usually undesirable, because it changes the structure of the grammar. In particular, we have to replace left recursion with right recursion.
(Insert first/follows sets and explanation of why it's bad.)

(d) Construct an LL(1) parser table, based on your grammar in part **(c)**. [5 marks]

	id	;	=	(	)	,	$
eq-list	1	-	-	-	-	-	-
rest-eq-list	-	3	-	-	-	-	2
eqn	4	-	-	-	-	-	-
exp	5	-	-	-	-	-	-
rest-exp	-	-	6	7	-	-	-
exp-list	8	-	-	-	-	-	-
rest-exp-list	-	-	-	-	9	10	-

(e) Extended Context Free Grammars (ECFGs) provide an attractive basis for constructing recursive descent parsers, because repetition can be handled by loops, both in the grammar and in the parser.

Write an ECFG, in LL(1) form, equivalent to the grammar given above.

You should make best use of the features of ECFGs to obtain a compact and intelligible grammar, reflecting the structure of the original grammar as closely as possible. [5 marks]

eq-list	$\rightarrow$	*eqn* { ";" *eqn* }
eqn	$\rightarrow$	*exp* "=" *exp*
exp	$\rightarrow$	*id* ["(" *exp-list* ")"]
exp-list	$\rightarrow$	*exp* { "," *exp* }

(f) Write the procedures required for a recursive descent parser to recognise lists of equations, based on your grammar from part **(e)**.

You should assume the availability of a scanner, recognising the terminal symbols used in this grammar (i.e. ";", "=", "(", ")", "," and *id*). [10 marks]

(INSERT)

(g) Explain how you would extend your parser in part **(f)** to build a representation of the list of equations recognised.

You may assume any reasonable representation for lists of equations, provided you explain the operations used in its construction. [5 marks]

(INSERT)

continued...

Question 3. [25 marks]

(a) Explain briefly the difference between *strong equivalence* and *weak equivalence* of programs. [4 marks]

Standard "book work".

(b) Consider the following three while programs, where B_1 and B_2 are any Boolean expressions and S_1, S_2, S_3 are any statements:

```
P_1:  if B_1 then S_1 else S_2 fi;
      if B_2 then S_3 else S_4 fi

P_2:  if B_1 then
         S_1; if B_2 then S_3 else S_4 fi
      else
         S_2; if B_2 then S_3 else S_4 fi
      fi

P_3:  if B_1 then
         if B_2 then S_1; S_3 else S_1; S_4 fi
      else
         if B_2 then S_2; S_3 else S_2; S_4 fi
      fi
```

(i) Show that P_1 and P_2 are *strongly* equivalent. [4 marks]

Call the programs P_1 and P_2, and let $Paths(P)$ be the RE for all potential paths in P. Then:
$Paths(P_1) = (B_1\,S_1|\overline{B_1}\,S_2)(B_2\,S_3|\overline{B_2}\,S_4) = (B_1\,S_1 S\,(B_2\,S_3|\overline{B_2}\,S_4)|\overline{B_1}\,S_2\,(B_2\,S_3|\overline{B_2}\,S_4)) = Paths(P_2)$
The two REs are equal, by distributivity.
(NB: This is blurring the distinction between S_1 and $Paths(S_1)$, etc.)

(ii) Under what circumstances will P_1 and P_3 be *weakly* equivalent? [2 marks]

When the value of B_2 is not affected by executing S_1 or S_3.

(c) Show that, for any Boolean expression B and any statements S_1 and S_2, there is a while program which is *strongly* equivalent to the following flowchart program: [5 marks]

```
1:  S_1;
    if B then 2 else 3;
2:  S_2;
    goto 1;
3:  skip
```

The following while program is strongly equivalent to the given flowchart program (for any ...):

```
$S_1$;
**while** $C$ **do**
    $S_2$;
    $S_1$
**od**
```

The paths RE in both cases is: $S_1(B\,S_2\,S_1)^*\overline{B}$

(d) Show that, for any while program, there is a *weakly* equivalent while program containing only one **while** statement. [5 marks]

(You may assume results about program transformations that were proved in the course notes.)

Translate the while program into a (strongly equivalent) flowchart program, using the T_{wf} function defined in the notes. Then translate that into a (weakly equivalent) while program using the T_{fw} function defined in the notes.
The resulting program is weakly equivalent to the original program (by a sort of transitivity).
The resulting program also has only one **while** statement (because T_{fw} constructs a program with a single **while** statement whose body is a multiway if statement).

(e) Demonstrate the effect of the transformation you described in part **(d)** on the following while program: [5 marks]

```
$k := 1$;
**while** $n \neq 0$ **do**
    **while** $A[k] \neq x$ **do**
        $k := k + 1$
    **od**;
    $n := n - 1$; $k := k + 1$
**od**
```

Applying T_{wf} to the given while program gives (the line numbers on the left are used in the next step):

```
(1)        $k := 1$;
(2)   1:   **if** $n \neq 0$ **then** 2 **else** 3;
(3)   2:   **skip** ;
(4)   4:   **if** $A[k] \neq x$ **then** 5 **else** 6;
(5)   5:   **skip** ;
(6)        $k := k + 1$
(7)        **goto** 1;
(8)   6:   **skip**
(9)        $n := n - 1$;
(10)       $k := k + 1$;
(11)       **goto** 1;
(12)  3:   **skip**
```

Applying T_{fw} to the flowchart program gives:

```
$s := 1$;
**while** $s \leq 12$ **do**
    **if** $s = 1$ **then** $k := 1$; $s := 2$
    **elsif** $s = 2$ **then** **if** $n \neq 0$ **then** $s := 3$ **else** $s := 12$
    **elsif** $s = 3$ **then** **skip** ; $s := 4$
    **elsif** $s = 4$ **then** **if** $A[k] \neq x$ **then** $s := 5$ **else** $s := 8$
    **elsif** $s = 5$ **then** **skip** ; $s := 6$
    **elsif** $s = 6$ **then** $k := k + 1$; $s := 7$
    **elsif** $s = 7$ **then** $s := 4$
    **elsif** $s = 8$ **then** **skip** ; $s := 9$
    **elsif** $s = 9$ **then** $n := n - 1$; $s := 10$
    **elsif** $s = 10$ **then** $k := k + 1$; $s := 11$
    **elsif** $s = 11$ **then** $s := 2$
    **elsif** $s = 12$ **then** **skip** ; $s := 13$
    **fi**
**od**
```

continued...

Question 4. **[20 marks]**

The following is an algorithm to count the number of index positions at which two strings s and t have identical elements, assuming that $|s| = m$ and $|t| = n$.

```
$k := 0$; $c := 0$;
**while** $k \neq m \wedge k \neq n$ **do**
    **if** $s[k+1] = t[k+1]$ **then**
        $c := c + 1$
    **fi**;
    $k := k + 1$
**od**
```

If we write *count*(i, S, P) for the number of elements, i, in set S, satisfying property P (i.e. $count(i, S, P) = |\{\, i \mid i \in S \wedge P \,\}|$), and $u \mathrel{..} v$ for the set $\{j \mid u \leq j \leq v\}$, we can express the postcondition for this program as:

$$k = count(\, i,\ 1 \mathrel{..} min(m, n),\ s[i] = t[i]\,)$$

(a) Explain what is meant by a "loop invariant", and how a loop invariant can be used in proving that a loop satisfies a given specification. [5 marks]

$\boxed{\text{This is standard "book work".}}$

(b) Give a loop invariant that could be used to verify the loop in the above program. [5 marks]

$\boxed{0 \leq k \leq min(m, n) \ \wedge\ c = count(i, 1..k, s[i] = t[i])}$

(c) Use your loop invariant from part **(b)** to prove that the program correctly computes the number of index positions at which s and t have identical elements.

You should give the verification conditions that must hold in order for the program to be correct, and clearly identify any mathematical properties of strings and/or of counting that your proof relies upon. [10 marks]

(i) The loop invariant holds on entry to the loop. [2 marks]

$$\begin{aligned} & k = 0 \wedge c = 0 \Rightarrow 0 \le k \le min(m,n) \wedge c = count(i, 1..k, s[i] = t[i]) \\ \equiv \quad & 0 \le 0 \le min(m,n) \wedge 0 = count(i, 1..0, s[i] = t[i]) \\ \equiv \quad & true \end{aligned}$$

(ii) The loop invariant is preserved by the loop body. [6 marks]

We need to consider two cases, according to whether $s[k+1] = t[k+1]$ holds or not. The resulting verification conditions are:

(i) $0 \le k \le min(m,n) \wedge c = count(i, 1..k, s[i] = t[i]) \wedge$
$k \ne m \wedge k \ne m \wedge s[k+1] = t[k+1] \Rightarrow$
$0 \le k+1 \le min(m,n) \wedge c+1 = count(i, 1..k+1, s[i] = t[i])$

This holds, since $s[k+1] = t[k+1] \Rightarrow count(i, 1..k+1, s[i] = t[i]) = count(i, 1..k, s[i] = t[i]) + 1$

(ii) $0 \le k \le min(m,n) \wedge c = count(i, 1..k, s[i] = t[i]) \wedge$
$k \ne m \wedge k \ne m \wedge s[k+1] \ne t[k+1] \Rightarrow$
$0 \le k+1 \le min(m,n) \wedge c = count(i, 1..k+1, s[i] = t[i])$

This holds, since $s[k+1] \ne t[k+1] \Rightarrow count(i, 1..k+1, s[i] = t[i]) = count(i, 1..k, s[i] = t[i])$

Note that we can obtain these verification conditions via several routes: the details of how they are obtained is not so important as what they are.

(iii) The postcondition $k = count(i, 1..min(m,n), s[i] = t[i])$ holds when the loop exits with the loop invariant true. [2 marks]

The verification condition is:

$0 \le k \le min(m,n) \wedge c = count(i, 1..k, s[i] = t[i]) \wedge$
$\neg(k \ne m \wedge k \ne m) \Rightarrow$
$c = count(i, 1..min(m,n), s[i] = t[i])$

This holds, since $0 \le k \le min(m,n) \wedge \neg(k \ne m \wedge k \ne m) \Rightarrow k = min(m,n)$.

(iv) The loop terminates for any strings s and t. [2 marks]

Take the bound function to be $t \triangleq min(m,n) - k$.

Every iteration of the loop increases k, and the loop must exit when $min(m,n) - k = 0$, i.e. $k = min(m,n)$.

****