THE AUSTRALIAN NATIONAL UNIVERSITY

Second Semester 2011

COMP2600
(Formal Methods for Software Engineering)

Writing Period: 3 hours duration

Study Period: 15 minutes duration

Permitted Materials: One A4 page with hand-written notes on both sides

Answer ALL questions
Total marks: 100
Note internal choice in Question 9

WITH SOME SAMPLE SOLUTIONS

The questions are followed by labelled blank spaces into which your answers are to be written.

Additional answer panels are provided (at the end of the paper) should you wish to use more space for an answer than is provided in the associated labelled panels. If you use an additional panel, be sure to indicate clearly the question and part to which it is linked.

The following spaces are for use by the examiners.

Q1 (StrInd)	Q2 (Z)	Q3 (Hoare)	Q4 (WPs)	Q5 (FSA)

Q6 (NatDed)	Q7 (TM)	Q8 (CFG)	Q9 (1 of 4)	Total

QUESTION 1 [12 marks] (Structural Induction)

In all proofs indicate the justification (eg, the line of a definition used) for each step.

(a) The function `revA` provides an efficient way to reverse a list: `revA xs ys` reverses the list `xs` and appends the result to (the front of) the list `ys`.

If we wish to modify members of these lists with `map f` and also to apply `revA` to the two lists, show that these operations can be done in either order. That is, prove

```
map f (revA xs ys) = revA (map f xs) (map f ys)
```

The definitions of `map` and `revA`:

```
map f []     = []                                    -- M1
map f (x:xs) = f x : map f xs                        -- M2

revA [] ys     = ys                                  -- RA1
revA (x:xs) ys = revA xs (x : ys)                    -- RA2
```

(i) State and prove the base case goal

QUESTION 1(a)(i) **[2 marks]**

We use induction on `xs` (not `ys`): the clue here is that the definition of `revA xs ys` is recursive on its first argument `xs`.

Base case: `xs = []`
Show that `map f (revA [] ys) = revA (map f []) (map f ys)`

Proof:

```
  map f (revA [] ys)
     = map f ys                          -- by (RA1)
     = revA [] (map f ys)                -- by (RA1)
     = revA (map f []) (map f ys)        -- by (M1)
```

In the remaining parts of your solution, indicate the use of ∀ quantifiers in the property to be proved and in the inductive hypotheses, and indicate how these are instantiated in proving the step case goal.

(ii) State the inductive hypothesis

QUESTION 1(a)(ii) **[1 mark]**

The property *P* we prove by induction will be
`P(xs) = ∀ys.  map f (revA xs ys) = revA (map f xs) (map f ys)`
For proving the step case with `xs = a:as`,
the inductive hypothesis is: *for all* `ys`

```
  map f (revA as ys) = revA (map f as) (map f ys)    -- (IH)
```

(iii) State and prove the step case goal:

QUESTION 1(a)(iii) **[4 marks]**

Step case: (for `xs = a:as`)
Assuming the inductive hypothesis (IH) above, show that:
for all `ys`,
`map f (revA (a:as) ys) = revA (map f (a:as)) (map f ys)`

Proof: Let `zs` be given (we will generalise `zs` to `∀ys`)

```
map f (revA (a:as) zs)
  = map f (revA as (a:zs))                    -- by (RA2)
  = revA (map f as) (map f (a:zs))            -- by (IH) (*)
  = revA (map f as) (f a : map f zs)          -- by (M2)
  = revA (f a : map f as) (map f zs)          -- by (RA2)
  = revA (map f (a:as)) (map f zs)            -- by (M2)
```

`(*)` Note - the `ys` in (IH) is instantiated to `a:zs`

Then we can generalise (using the ∀-I rule) `zs` to `ys`, to get
`∀ys. map f (revA (a:as) ys) = revA (map f (a:as)) (map f ys)`
which completes the proof.

(b) Binary trees can be represented in Haskell with the following algebraic data type:

```
data Tree a = Nul | Node a (Tree a) (Tree a)
```

We can get the sum of entries in a tree by the function `sumT`

```
sumT :: Tree Int -> Int
sumT  Nul           = 0                          -- (ST1)
sumT (Node n t1 t2) = n + sumT t1 + sumT t2      -- (ST2)
```

We can reverse a tree (get its mirror image) by the function `revT`

```
revT :: Tree a -> Tree a
revT  Nul           = Nul                             -- (RT1)
revT (Node n t1 t2) = Node n (revT t2) (revT t1)      -- (RT2)
```

This question is about proving, by structural induction on the structure of the tree argument, that for all trees `t :: Tree Int`,

```
sumT (revT t) = sumT t
```

(i) State and prove the base case goal.

QUESTION 1(b)(i) **[1 mark]**

Base case: `t = Nul`
Show that `sumT (revT Nul) = sumT Nul`

Proof:

```
  sumT (revT Nul) = sumT Nul                  -- by (RT1)
```

(ii) State the inductive hypotheses.

QUESTION 1(b)(ii) **[1 mark]**

for the step case `t = Node y t1 t2`:

```
sumT (revT t1) = sumT t1                    -- (IH1)
sumT (revT t2) = sumT t2                    -- (IH2)
```

(iii) State and prove the step case goal.

QUESTION 1(b)(iii) **[3 marks]**

Step case: (for `t = Node y t1 t2`)
Assuming the inductive hypotheses (IH1) and (IH2) as above, prove
`sumT (revT (Node y t1 t2)) = sumT (Node y t1 t2)`

Proof:

```
sumT (revT (Node y t1 t2))
  = sumT (Node y (revT t2) (revT t1))       -- by (RT2)
  = y + sumT (revT t2) + sumT (revT t1)     -- by (ST2)
  = y + sumT t2 + sumT t1                   -- by (IH1,IH2)
  = y + sumT t1 + sumT t2                   -- by arith
  = sumT (Node y t1 t2)                     -- by (ST2)
```

QUESTION 2 [12 marks] (Specification in Z)

The system that is partially modelled in this question is one in which a number of *agencies* of a *limousine service* are each able to provide a car to transport a person from one place to another. The cost for the hire is proportional to the *distance* that the hired vehicle needs travel at an agency-dependant *rate*. Each agency manages its own stock of cars. Vehicles are retired when they have accumulated a large number of kilometres; and so *records* are kept of all *trips* made.

The state schema for the system is called *LimoService* and is shown below.

[Vehicle]
[Place]
[Agency]

$$addr : Agency \rightarrow Place$$

$$\forall c_1, c_2 : Agency \bullet (c_1 = c_2) \Leftrightarrow (addr(c_1) = addr(c_2))$$

$$dist : Place \times Place \rightarrow \mathbb{R}$$
$$trip : (Place \times Place \times Place) \rightarrow \mathbb{R}$$

$$\forall x, y : Place \bullet dist(x, y) \geq 0$$
$$\forall x, y : Place \bullet dist(x, y) = 0 \Leftrightarrow x = y$$
$$\forall x, y, z : Place \bullet dist(x, y) + dist(y, z) \geq dist(x, z)$$
$$\forall x, y, z : Place \bullet trip(x, y, z) = dist(x, y) + dist(y, z) + dist(z, x)$$

TripRecord

$$car : Vehicle$$
$$distance : \mathbb{R}$$
$$cost : \mathbb{R}$$
$$distance > 0 \ \wedge \ cost > 0$$

LimoService

$$cars : Agency \rightarrow \mathbb{P}\, Vehicle$$
$$rate : Agency \rightarrow \mathbb{R}$$
$$history : Agency \rightarrow \mathbb{P}\, TripRecord$$
$$\forall a_1, a_2 : Agency \bullet a_1 \neq a_2 \Rightarrow cars(a_1) \cap cars(a_2) = \varnothing$$
$$\forall a : Agency \bullet rate(a) > 0$$
$$\forall a : Agency, t : TripRecord \bullet t \in history(a) \Rightarrow car(t) \in cars(a)$$

(a) In the above prelude to the specification of operations and queries, which identifiers are

- the *given types*,
- the *defined types?*

QUESTION 2(a) **[1 mark]**

- the *given types* are *Vehicle*, *Place* and *Agency*.
- the one *defined type* is *TripRecord*.

(b) In the above prelude to the specification of operations and queries, which identifiers are

- the *axiomatically introduced constants*,
- the *global variables* for the system?

QUESTION 2(b) **[1 mark]**

- the *axiomatically introduced constants* are *addr*, *dist* and *trip*.
- the one *global variables* are *cars*, *rate* and *history*.

(c) Describe in simple terms what the system invariants are.

QUESTION 2(c) **[2 marks]**

- No car belongs to two agencies
- Regardless of agency the cost per kilometre is greater than zero.
- The car mentioned in any trip record belongs to the agency having that trip record.

(d) What does the function $trip(b, f, t)$ compute?

QUESTION 2(d) **[1 mark]**

The round-trip distance from b to f to t and back to b.

(e) Write the state initialisation schema for the system *LimoService*.

QUESTION 2(e) **[1 mark]**

Initialize

LimoService

$$\forall a : Agency \bullet cars(a) = \varnothing$$
$$\forall a : Agency \bullet rate(a) = defaultRate$$
$$\forall a : Agency \bullet history(a) = \varnothing$$

AddNewCar$_o$

$\Delta LimoService$
$a? : Agency$
$v? : Vehicle$

$\neg\ \exists a : Agency \bullet v? \in cars(a)$
$\forall a : Agency \bullet\ a \neq a?\ \Rightarrow$
$\quad cars'(a) = cars(a)$
$cars'(a?) = cars(a)\ \cup\ \{v?\}$

GetQuote$_o$

$\Xi LimoService$
$a? : Agency$
$from?,\ to? : Place$
$quote! : \mathbb{N}$
$kms : \mathbb{R}$

$kms = trip(addr(a?),\ from?,\ to?)$
$quote! = rate(a?) \times kms$

(f) The specification (above) of the operation *AddNewCar*$_o$ to add a new car to an agency's stock is erroneous. How is it underspecified?

QUESTION 2(f) **[1 mark]**

It doesn't give postconditions involving $rate'$ or $history'$. The following assertions would be appropriate:

$rate' = rate$
$history' = history$

(g) The schema *GetQuote*$_o$ (above) specifies the query which gives, in the output variable *quote*!, the cost of hiring a limo to go from one place to another using agency $a?$. The following schema, *BestQuote*$_o$, attempts to get the best quote, considering all agencies, but is seriously defective. Explain what the problem is or how you would fix it.

BestQuote$_o$

$\Xi LimoService$
$from?,\ to? : Place$
$quote! : \mathbb{N}$
$a! : Agency$
$kms : \mathbb{R}$

$kms = trip(addr(a!),\ from?,\ to?)$
$quote! = rate(a!) \times kms$
$\forall a : Agency \bullet a \neq a!\ \Rightarrow$
$\quad kms = trip(addr(a),\ from?,\ to?)\ \wedge$
$\quad quote! \leq rate(a) \times kms$

QUESTION 2(g) **[2 marks]**

The given predicate part asserts that the variable *kms* has a value which is equal to every possible round trip distance. (Note: there are no *assignment statements* in Z predicates – only statements of equality.)

It is easily fixed by replacing the last two lines with:

$quote! \leq rate(a) \times trip(addr(a),\ from?,\ to?)$

(h) Complete the schema $AddTrip_o$ by augmenting the predicate part appropriately.

Note that, as well as the three input variables, there are two local variables that you may find useful. Remember to give appropriate precondition(s) for the operation to be error-free.

QUESTION 2(h) **[3 marks]**

$AddTrip_o$

$\Delta LimoService$
$v? : Vehicle$
$from?, to? : Place$
$a : Agency$
$t : TripRecord$

$from? \neq to?$
$\exists x : Agency \bullet v? \in cars(x)$

$v? \in cars(a)$
$cars' = cars$
$rate' = rate$
$car(t) = v?$
$distance(t) = dist(addr(a), from?, to?)$
$cost(t) = distance(t) \times rate(a)$
$\forall x : Agency \bullet x \neq a \Rightarrow history'(x) = history(a)$
$history'(a) = history(a) \cup \{t\}$

QUESTION 3 [11 marks] (Hoare Logic)

The following code fragment *Average* takes two numbers *x* and *y*, typed integer, and assigns a (possibly fractional) number to the variable *a*.

```
while (x < y) do
   x := x + 1;
   y := y - 1;
if (x = y)
  then a := y
  else a := y + 1/2
```

Body (lines 2–3); *Loop* (lines 1–3); *Cond* (lines 4–6); *Average* (lines 1–6)

We wish to use Hoare Logic (Appendix 2) to show that *a* is the average of the initial values of *x* and *y*, so long as a certain precondition holds.

Specifically, let

$$\begin{array}{lcl} \mathit{Pre} & \equiv & (\ (\mathit{sum} = x + y) \wedge (x \leq y + 1)\ ) \\ \mathit{Post} & \equiv & (\ a = \frac{\mathit{sum}}{2}\ ). \end{array}$$

We wish to prove that

$$\{\mathit{Pre}\}\ \mathit{Average}\ \{\mathit{Post}\}.$$

In the questions below (and your answers), we may refer to the loop code as *Loop*, the body of the loop as *Body*, and the if-then-else statement as *Cond*.

(a) Using the rules of Hoare Logic, prove that

$$\{(\mathit{sum} = x + y) \wedge (y \leq x \leq y + 1) \wedge (x = y)\}\ \texttt{a := y}\ \{\mathit{Post}\}.$$

Be sure to properly justify each step of your proof. Any use of precondition strengthening or postcondition weakening must be explicit, but the arithmetic facts that you rely on need not be proved, although they must clearly be true.

QUESTION 3(a) **[2 marks]**

(b) Prove that

$$\{(sum = x + y) \wedge (y \leq x \leq y + 1)\}\ Cond\ \{Post\}.$$

You may use the result of question (a) here. You may also use the fact that x and y are integers (whole numbers).

QUESTION 3(b) **[3 marks]**

(c) We must find an invariant for our while loop. One suggestion is to use *Pre*. Prove that this is indeed a suitable loop invariant for *Loop*. That is,

$$\{Pre \wedge (x < y)\}\ Body\ \{Pre\}.$$

QUESTION 3(c) **[3 marks]**

(d) Using the result of part (c), prove that

$$\{Pre\}\ Loop\ \{(sum = x + y) \wedge (y \leq x \leq y + 1)\}.$$

QUESTION 3(d) **[2 marks]**

(e) Using any of the results from this question that you wish, prove that

$$\{Pre\}\ Average\ \{Post\}.$$

Remember to justify your reasoning by citing each previous answer you are using by its number.

QUESTION 3(e) **[1 mark]**

QUESTION 4 [11 marks] (Weakest Precondition Calculus)

(a) Suppose that, given a code fragment *S*, we used the rules of the Weakest Precondition Calculus (Appendix 3) to find that

$$wp(S,\ True)\ \equiv\ False$$

What can we say about *S*?

QUESTION 4(a)	**[2 marks]**

(b) The following code fragment *SquareSum* calculates the sum of the first *n* square numbers.

```
s := 0;
x := 0;
while (x ≠ n) do
   x := x + 1;
   s := s + (x * x)
```

Body *Loop* *SquareSum*

All the variables are typed integer. In the questions below (and your answers), we may refer to the loop code as *Loop* and the body of the loop as *Body*

In these questions we will calculate the weakest precondition of *SquareSum* with respect to the postcondition

$$Post\ \equiv\ (\ s = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6}\ )$$

(i) We first need to find $wp(Loop, Post)$. State P_0, the predicate expressing success after zero loop iterations.

QUESTION 4(b)(i)	**[1 mark]**

(ii) Assume that

$$P_k \equiv (\, x + k = n \;\wedge\; s = \frac{x^3}{3} + \frac{x^2}{2} + \frac{x}{6} \,)$$

Use this to find P_{k+1}, the predicate expressing success after $k + 1$ loop iterations. Simplify your answer as far as possible.

Hint: You might find the following algebraic expansions useful when simplifying:

$$(a+b)^2 = a^2 + 2ab + b^2$$
$$(a+b)^3 = a^3 + 3a^2b + 3ab^2 + b^3$$

QUESTION 4(b)(ii) **[4 marks]**

(iii) Using the fact from the previous questions that, for all integers $k \geq 0$,

$$P_k \equiv (\, x + k = n \ \wedge \ s = \frac{x^3}{3} + \frac{x^2}{2} + \frac{x}{6} \,)$$

find

$$wp(Loop, Post)$$

Simplify your answer as much as possible.

QUESTION 4(b)(iii) **[2 marks]**

(iv) Using the previous answer, find

$$wp(SquareSum, Post)$$

Remember to simplify your answer as much as possible.

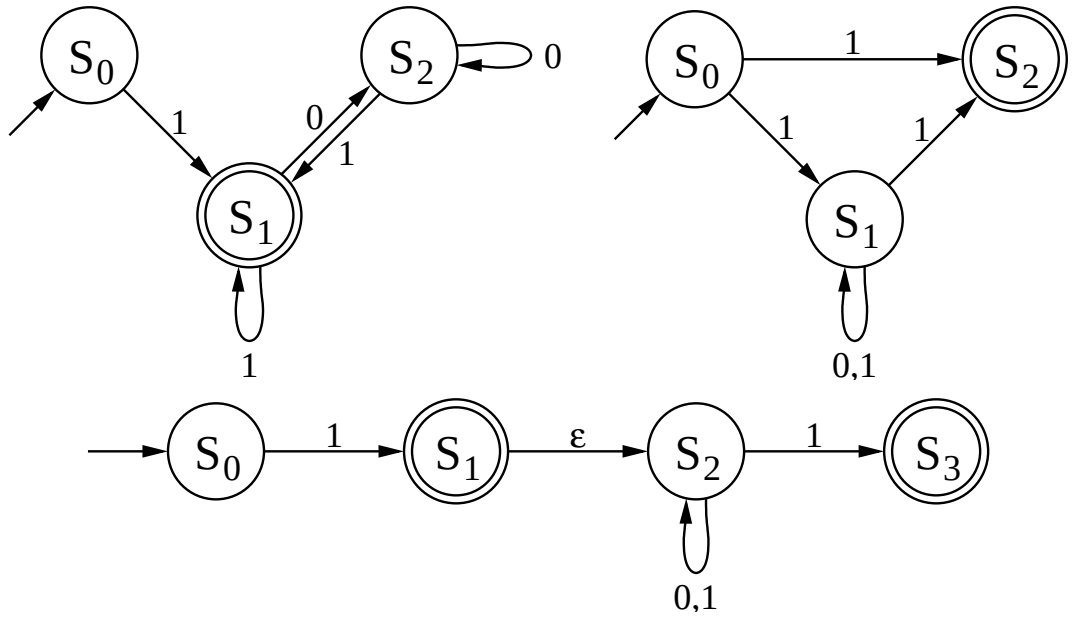
QUESTION 4(b)(iv) **[2 marks]**

QUESTION 5 [13 marks] (Finite State Automata)

(a) Design a finite state automaton to accept bit strings which both start and finish with 1. That is, your FSA should accept strings $\alpha \in \{0,1\}^*$ where the first digit and the last digit are both 1. (Hint: note that the single digit string 1 is included in this description, and should be accepted by your automaton).

QUESTION 5(a) **[2 marks]**

Here are three possible solutions



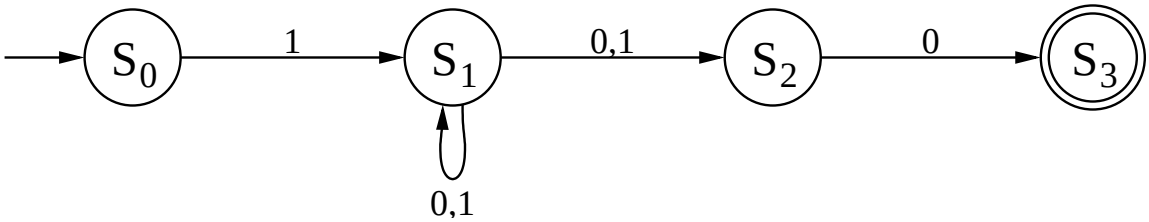
(b) Give a regular expression for this language.

QUESTION 5(b) **[1 mark]**

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

A common error was to omit the $\mid\ 1$.
Ensure your hand-written 1 and $\mid$ can be distinguished.

(c) The following FSA accepts a language L consisting of certain bit strings, ie, $L \subseteq \{0,1\}^*$.



Describe (in English) the language L which is accepted by the automaton.

QUESTION 5(c) **[2 marks]**

The language contains those strings over $\{0,1\}^*$ which

- start with 1
- finish with 0, and
- are of length at least 3 symbols

NOTE: correct terminology: a *language* is a set of *strings*

(d) Give a regular grammar (which may be right-linear *or* left-linear) which generates the language L of part (c).

QUESTION 5(d) **[2 marks]**

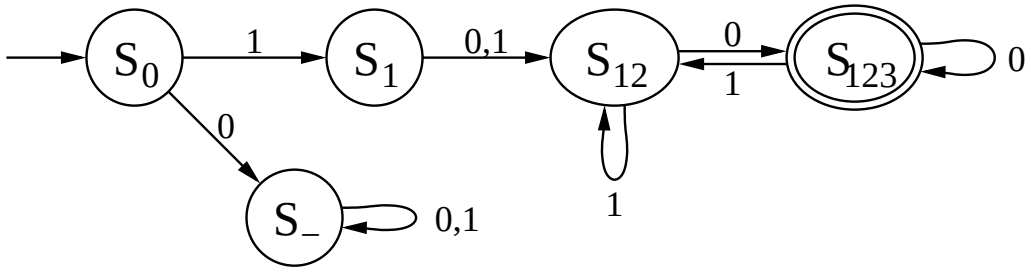
Left-linear		**Right-linear**
Start symbol S_3		Start symbol S_0
$S_0 \to \epsilon$		$S_3 \to \epsilon$
$S_1 \to S_0\ 1$		$S_2 \to 0\ S_3$
$S_1 \to S_1\ 0$		$S_1 \to 0\ S_2$
$S_1 \to S_1\ 1$	or	$S_1 \to 1\ S_2$
$S_2 \to S_1\ 0$		$S_1 \to 0\ S_1$
$S_2 \to S_1\ 1$		$S_1 \to 1\ S_1$
$S_3 \to S_2\ 0$		$S_0 \to 1\ S_1$

Don't forget to identify the start symbol (unless you use the conventional S).
But remember also that we're using a definition of a regular grammar which does not allow transitions of the form $A \to B$.
A common error was to forget the productions $S_3 \to \epsilon$ or $S_0 \to \epsilon$

(e) Design a *deterministic* FSA which recognises exactly the same language L of part (c).

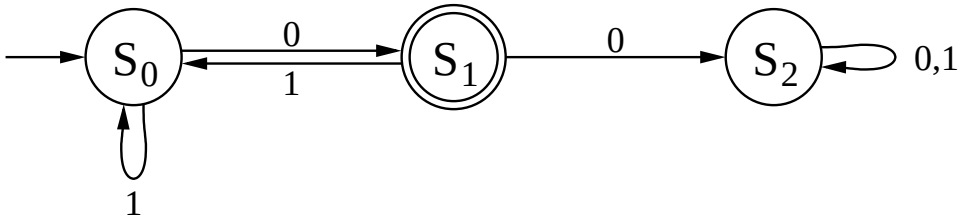
QUESTION 5(e) **[3 marks]**

The following solution was got using the general method for deriving a DFA from an NFA, where the states here labelled $S_0, S_1, S_{12}, S_{123}$ and S_- are the states which would be called $\{S_0\}, \{S_1\}, \{S_1, S_2\}, \{S_1, S_2, S_3\}$ and $\{\}$ according to that general method



A common error was to omit the state S_-, and the edges to and from it.
Note that a DFA must have no missing transitions.

(f) Consider the following DFA



Show that the automaton does not accept strings ending in 00, that is, for any $\alpha \in \Sigma^*$, $N^*(S_0, \alpha 00)$ is not a final state.

QUESTION 5(f) **[3 marks]**

By the "Append" theorem, $N^*(S_0, \alpha 00) = N^*(N^*(S_0, \alpha), 00)$
Now $N^*(S_0, \alpha)$ is S_0, S_1 or S_2, so $N^*(S_0, \alpha 00) = N^*(S_i, 00)$ for $i = 0, 1$ or 2.
Also $N^*(S_i, 00) = N(N(S_i, 0), 0)$ by the definition of N^* (or you can think of it as like another instance of the "Append" theorem).
Thus $N^*(S_0, \alpha 00) = N(N(S_i, 0), 0)$ for some i. So we look at cases of S_i.

$$N(N(S_0, 0), 0) = N(S_1, 0) = S_2$$
$$N(N(S_1, 0), 0) = N(S_2, 0) = S_2$$
$$N(N(S_2, 0), 0) = N(S_2, 0) = S_2$$

So in all cases $N^*(S_i, 00) = S_2$, which is not in F.
That is, in all cases $N^*(S_0, \alpha 00) \notin F$, as required.

QUESTION 6 [11 marks] (Natural Deduction)

Some of the following questions ask for proofs using natural deduction. Present your proofs in the Fitch style as used in lectures. Use only the introduction and elimination rules given in Appendix 1. Number each line and include justifications for each step in your proofs.

(a) Use truth tables to determine whether the following two propositions are (always) equivalent, or not; if not, give an example of values of p, q, r for which they are not equivalent.

$$p \vee (q \rightarrow r) \qquad (p \vee q) \rightarrow (p \vee r)$$

QUESTION 6(a) **[2 marks]**

p	q	r	$q \rightarrow r$	$p \vee (q \rightarrow r)$	$p \vee q$	$p \vee r$	$(p \vee q) \rightarrow (p \vee r)$
T	T	T	T	T	T	T	T
T	T	F	F	T	T	T	T
T	F	T	T	T	T	T	T
T	F	F	T	T	T	T	T
F	T	T	T	T	T	T	T
F	T	F	F	F	T	F	F
F	F	T	T	T	F	T	T
F	F	F	T	T	F	F	T

So the two propositions are equivalent for all values of p, q, r;
that is, they are always equivalent.
NOTE: A common error was to forget to state this conclusion!

(b) Give a natural deduction proof of $\dfrac{(p \to r) \wedge (q \to r)}{(p \vee q) \to r}$

QUESTION 6(b) **[3 marks]**

1	$(p \to r) \wedge (q \to r)$	
2	$p \to r$	$\wedge$-E, 1
3	$q \to r$	$\wedge$-E, 1
4	$p \vee q$	
5	p	
6	r	$\to$-E, 2, 5
7	q	
8	r	$\to$-E, 3, 7
9	r	$\vee$-E, 4, 5–6, 7–8
10	$(p \vee q) \to r$	$\to$-I, 4–9

(c) Give a natural deduction proof of $\dfrac{\neg\,(p \wedge \neg\, q)}{(p \to q)}$

QUESTION 6(c) **[3 marks]**

1	$\neg\,(p \wedge \neg\, q)$	
2	p	
3	$\neg\, q$	
4	$p \wedge \neg\, q$	$\wedge$-I, 2, 3
5	$(p \wedge \neg\, q) \wedge \neg\,(p \wedge \neg\, q)$	$\wedge$-I, 4, 1
6	q	$\neg$E, 3–5
7	$p \to q$	$\to$-I, 2–6

(d) Give a natural deduction proof of $\dfrac{\exists x.\ Q \to P(x)}{Q \to (\exists x.\ P(x))}$ (x does not appear free in Q).

Take care with parentheses and variable names.

QUESTION 6(d) **[3 marks]**

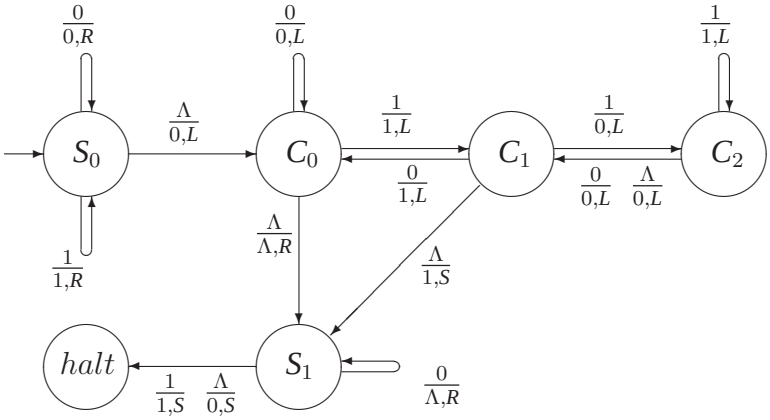
1	$\exists x.\ Q \to P(x)$	
2	$\quad Q$	
3	$\quad a \mid Q \to P(a)$	
4	$\quad\quad P(a)$	$\to$-E, 3, 2
5	$\quad\quad \exists x.\ P(x)$	$\exists$-I, 4
6	$\quad \exists x.\ P(x)$	$\exists$-E, 1, 3–5
7	$Q \to (\exists x.\ P(x))$	$\to$-I, 2–6

or this alternative proof

1	$\exists x.\ Q \to P(x)$	
2	$a \mid Q \to P(a)$	
3	$\quad Q$	
4	$\quad P(a)$	$\to$-E, 2, 3
5	$\quad \exists x.\ P(x)$	$\exists$-I, 4
6	$Q \to (\exists x.\ P(x))$	$\to$-I, 3–5
7	$Q \to (\exists x.\ P(x))$	$\exists$-E, 1, 2–6

QUESTION 7 [11 marks] (Turing Machines)

(a) The following diagram shows a Turing machine which performs an arithmetic operation on a binary number. Initially the head is somewhere over the input string which consists only of '0's and '1's; the rest of the tape is blank.



(i) Give a general description of the purpose of state S_0. (Mention the position of the head when the TM goes to state C_0.)

QUESTION 7(a)(i) **[1 mark]**

The head is moved to the right until a blank is encountered. It replaces that blank by 0, then it moves left and goes to state C_0. So the machine enters C_0 with the head pointing to the rightmost digit of the original (input) binary number.

(ii) If the given string is "00" what string is produced?

QUESTION 7(a)(ii) **[1 mark]**

0

(iii) To what string is "11" transformed?

QUESTION 7(a)(iii) **[1 mark]**

10010

(iv) Briefly describe what computation this machine performs.

QUESTION 7(a)(iv) **[1 mark]**

The binary number is multiplied by 6.
Initially, an extra zero is written on the right-hand end of the number, multiplying it by 2.
Then the states C_i represent a value $i = 0, 1, 2$ "carried in": starting from C_i, the binary number whose rightmost digit is under the head is multiplied by 3, and is then incremented by i.

(v) Give a general description of the purpose of state S_1. (What difference would there be if it was replaced by the halt state?)

QUESTION 7(a)(v) **[1 mark]**

The machine enters state S_1 with the head pointing to the leftmost binary digit. In state S_1 it replaces leading zeroes by blanks. On leaving S_1 it writes a single zero in the case of a number which was entirely zeroes and so has been replaced entirely by blanks.

(b) Design a Turing Machine which, given a binary string, will

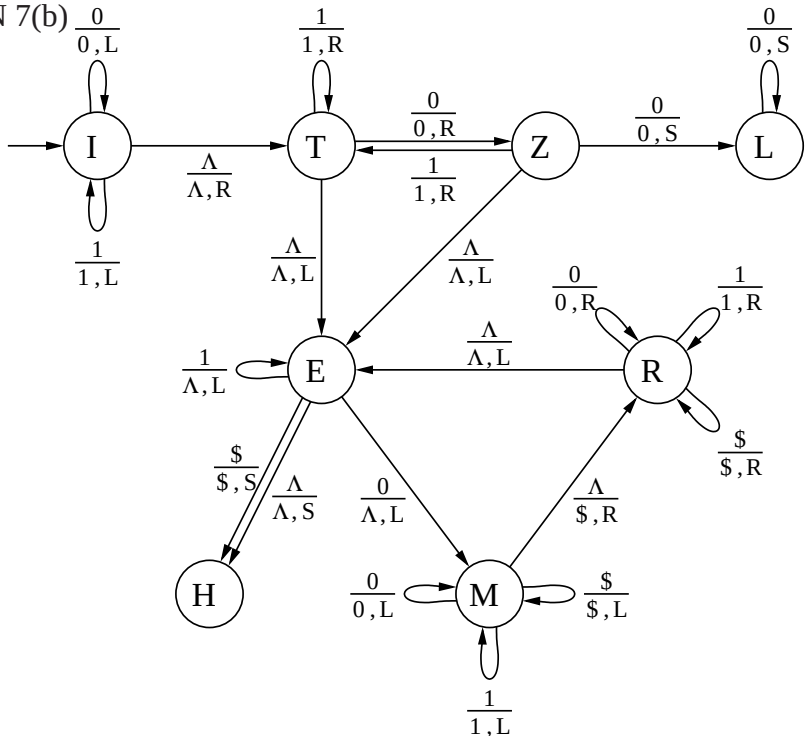
i) accept the string by halting *if and only if* there are no consecutive zeroes;

ii) for a string which is accepted, delete all the ones and replace every zero by a dollar.

For example, the machine should transform 01011101111 to $\$\$\$$ and halt. On the other hand, it should not halt on 1010011.

(You will get 3 marks if you give a Turing Machine that halts *if and only if* the input belongs to the given language.)

QUESTION 7(b) **[6 marks]**



- State I: moves to the leftmost bit
- State T, Z: moves right, looking for two consecutive zeroes; when 00 found, goes to state L (infinite loop)
- State E: at rightmost bit, goes left, erasing 1. When 0 found, erases it and moves to state M.
- State M: moves past left-hand end of string and writes \$, then
- State R: moves back to right-hand end of string, back to state E

QUESTION 8 [11 marks] (Context Free Grammars)

The following grammar describes a context free language L over the alphabet $\{a, x, q\}$:

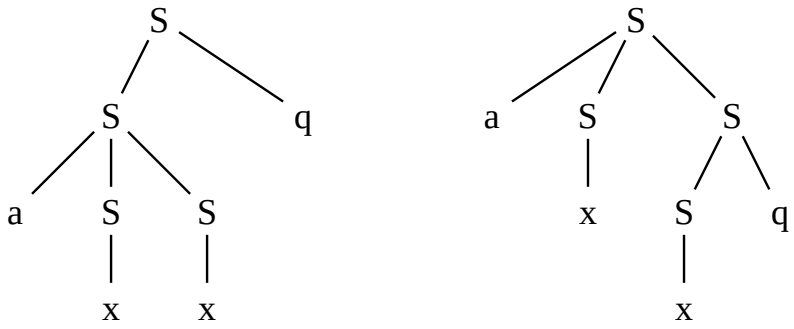
$$S \rightarrow a\,S\,S \mid S\,q \mid x$$

This grammar is one that is intended to describe a language of expressions involving a binary prefix operator (a) and a unary postfix operator (q); it illustrates the representation problems associated with mixing postix and prefix operators.

Thus, if a means "Add" and q means "sQuared", and if you think of eq as e^2, then you could take the sentence $axqx$ of the grammar to mean `add` x^2 x.

(a) With the aid of a couple of parse trees, show that the grammar is ambiguous.

QUESTION 8(a) **[2 marks]**



The string $axxq$ can be parsed in two different ways, which shows that the grammar is ambiguous.

(b) Give a non-ambiguous grammar for the language L. (Hint: let the operator q bind more tightly than the operator a).

QUESTION 8(b) **[3 marks]**

Two solutions shown: the first one conforms to the hint

$S \rightarrow Q$	$S \rightarrow A$
$Q \rightarrow x$	$A \rightarrow a\,S\,A$
$Q \rightarrow Q\,q$	$A \rightarrow x$
$S \rightarrow a\,S\,S$	$S \rightarrow S\,q$

In the first solution, the non-terminal Q expands to xq^n (for any n)

(c) Give a grammar which is not left-recursive for the language L

QUESTION 8(c) **[3 marks]**

Three solutions shown, got from the grammars given and shown in part (b). But of course in the second one, you could replace the first two productions by $S \rightarrow x\ R$. In the third you could replace the production $A \rightarrow a\ S\ A$ by $A \rightarrow a\ S\ S$ (which would make it ambiguous). In each case the non-terminal R generates q^n (for any n).

$S \rightarrow a\ S\ S\ R$	$S \rightarrow Q$	$S \rightarrow A\ R$
$S \rightarrow x\ R$	$Q \rightarrow x\ R$	$A \rightarrow a\ S\ A$
$R \rightarrow q\ R$	$R \rightarrow q\ R$	$A \rightarrow x$
$R \rightarrow \epsilon$	$R \rightarrow \epsilon$	$R \rightarrow q\ R$
	$S \rightarrow a\ S\ S$	$R \rightarrow \epsilon$

Note that a grammar can be indirectly left-recursive, which does **not** give a satisfactory solution eg

$$S \rightarrow S'q \qquad S' \rightarrow S$$

(d) Prove that the language L is not a regular language. (Hint: consider the relative number of occurrences of a and of x in any sentence of L).

QUESTION 8(d) **[3 marks]**

Each sentence of the language contains one more 'x' than the number of 'a's. In fact the string $a^n x^m \in L$ *if and only if* $m = n + 1$.

Imagine a DFA which accepts the language L (with starting state S_0).

Let S_n be the state reached after reading a^n from the input, ie $S_n = N^*(S_0, a^n)$.

Now, consider which strings consisting wholly of 'x's would be accepted by the machine if it started from S_n.

Starting from S_n, the machine accepts x^{n+1} but does not accept x^m for any $m \neq n + 1$.

That is, $N^*(S_n, x^{n+1}) = N^*(S_0, a^n x^{n+1}) \in F$, and
$N^*(S_n, x^m) = N^*(S_0, a^n x^m) \notin F$ for $m \neq n + 1$.

So the states S_n, for various values of n, must all be different. Yet there are infinitely many of them, which is not possible in a *finite* state machine.

NOTE: It is **wrong** to answer the question by saying that the grammar presented is not regular. That is so, but it is possible, in general, that a language L is generated by many different grammars, some regular, some not.

A language is regular if and only if *there exists* a regular grammar which generates it.

NOTE: A common wrong answer was simply to say that any sentence containing n 'a's must contain exactly $n + 1$ 'x's. That is so, but it is also true of the regular language given by the regular expression $(ax)^*x$. The key point here is that L contains the strings $a^n x^{n+1}$: when reading *these* strings, the FSA must keep count of the 'a's.

QUESTION 9 [8 marks]

(Choice of *one* of four topics)

(a) Push-Down Automata

(b) Lambda Calculus and Types

(c) Prolog

(d) Computability

Attempt just ONE of the four parts (a), (b), (c) or (d) of this question (noting that some parts have several subparts, totalling 8 marks). (If you start to write answers to more than one part, indicate CLEARLY which one is to be marked).

(a) Push-Down Automata

The following list of transitions specifies the transition relation δ of a PDA with stack vocabulary $\{<, Z\}$ and input alphabet $\Sigma = \{<, x, >\}$. As usual Z, used to indicate the bottom of the stack, is the only item in the stack initially.

The set of control states of the PDA is $\{q_s, q_<, q_x, q_>, q_f\}$. The initial state is q_s and the only final state is q_f. A string over Σ^* is accepted by the PDA if the PDA consumes the string and ends up in the final state with an empty stack.

$$\begin{aligned}
\delta(q_s, <, Z) &\mapsto q_</ < Z \\
\delta(q_<, <, <) &\mapsto q_</ << \\
\delta(q_<, x, <) &\mapsto q_x/ < \\
\delta(q_x, x, <) &\mapsto q_x/ < \\
\delta(q_x, >, <) &\mapsto q_>/\epsilon \\
\delta(q_>, >, <) &\mapsto q_>/\epsilon \\
\delta(q_>, <, Z) &\mapsto q_</ < Z \\
\delta(q_>, <, <) &\mapsto q_</ << \\
\delta(q_>, \epsilon, Z) &\mapsto q_f/\epsilon
\end{aligned}$$

(i) Give a trace of the acceptance of the input string $< xx >$.

QUESTION 9(a)(i) **[2 marks]**

$$\begin{aligned}(q_s, < x\,x >,\ Z) &\Rightarrow (q_<,\ x\,x >, < Z)\\ &\Rightarrow (q_x,\ x >, < Z)\\ &\Rightarrow (q_x, >, < Z)\\ &\Rightarrow (q_>,\ \epsilon,\ Z)\\ &\Rightarrow (q_f,\ \epsilon,\ \epsilon)\end{aligned}$$

NOTE: A *trace* is a sequence of *configurations*,
each of which consists of (*state, remaining input, stack contents*)

(ii) Give a trace of the acceptance of the input string $<< x >>$.

QUESTION 9(a)(ii) **[2 marks]**

$$\begin{aligned}(q_s, << x >>,\ Z) &\Rightarrow (q_<, < x >>, < Z)\\ &\Rightarrow (q_<,\ x >>, << Z)\\ &\Rightarrow (q_x, >>, << Z)\\ &\Rightarrow (q_>, >, < Z)\\ &\Rightarrow (q_>,\ \epsilon,\ Z)\\ &\Rightarrow (q_f,\ \epsilon,\ \epsilon)\end{aligned}$$

(iii) Describe (in English) the language which the PDA accepts.

QUESTION 9(a)(iii) **[2 marks]**

The language consists of strings over $\Sigma = \{<, x, >\}$ where

- there are equal numbers of '$<$' and '$>$', where each '$>$' can be matched to a *preceding* '$<$' (like properly matched parentheses)
- but $<>$ cannot appear adjacent to each other, one or more 'x' *must* occur in between them
- 'x' may not appear other than between '$<$' and '$>$'

(Note that the states $\{q_<, q_x, q_>\}$ are used to record the last input symbol)

(iv) What does a $<$ on the stack indicate?

QUESTION 9(a)(iv) **[2 marks]**

Pushing '$<$' onto the stack is used to count the number of '$<$' seen in the input, minus the number of '$>$' seen. Thus '$<$' on the stack means that the input so far has contained more '$<$' than '$>$'.

(b) Lambda Calculus and Types

(i) Use α-conversion to change the following to an equivalent expression in which distinct bound variables are represented by different letters:

$$(\lambda x.\ (\lambda x.\ (\lambda x.\ x)\ (\lambda y.\ x))\ (\lambda x.\ \lambda y.\ x)\ x)$$

QUESTION 9(b)(i) **[1 mark]**

$$(\lambda x.\ (\lambda b.\ (\lambda a.\ a)(\lambda y.\ b))(\lambda c.\ \lambda y.\ c)x)$$

(ii) Unify the following two type expressions, so as to obtain the most general unifier. Show clearly the steps in the algorithm you use.

$$(\alpha, \delta \to \beta) \to \gamma \qquad\qquad (\text{Bool}, \gamma) \to (\text{Int} \to \alpha)$$

In your answer you should show the substitutions you obtain for the type variables, and the result of applying this substitution to the given types.

QUESTION 9(b)(ii) **[2 marks]**

In the following solution, at each step we attack the first of the the list of pairs yet to be unified. For this we either decompose the pair of terms, or, if one of the terms is a variable we make that the next new substitution. For each new substitution, you need to apply that substitutution to (1) the substitutions previously obtained, and (2) the remaining pairs to be unified.

Pairs yet to be unified	**Substitutions found so far**
$((\alpha, (\delta \to \beta)) \to \gamma) = ((\text{Bool}, \gamma) \to (\text{Int} \to \alpha))$	
$(\alpha, (\delta \to \beta)) = (\text{Bool}, \gamma)$; $\gamma = (\text{Int} \to \alpha)$	
$\alpha = \text{Bool}$; $(\delta \to \beta) = \gamma$; $\gamma = (\text{Int} \to \alpha)$	
$(\delta \to \beta) = \gamma$; $\gamma = (\text{Int} \to \text{Bool})$	$\alpha = \text{Bool}$
$(\delta \to \beta) = (\text{Int} \to \text{Bool})$	$\gamma = (\delta \to \beta)$; $\alpha = \text{Bool}$
$\delta = \text{Int}$; $\beta = \text{Bool}$	$\gamma = (\delta \to \beta)$; $\alpha = \text{Bool}$
$\beta = \text{Bool}$	$\delta = \text{Int}$; $\gamma = (\text{Int} \to \beta)$; $\alpha = \text{Bool}$
	$\beta = \text{Bool}$; $\delta = \text{Int}$; $\gamma = (\text{Int} \to \text{Bool})$; $\alpha = \text{Bool}$

so the substitutions found are
$\beta = \text{Bool}$; $\delta = \text{Int}$; $\gamma = (\text{Int} \to \text{Bool})$; $\alpha = \text{Bool}$
and the terms unify to $((\text{Bool}, (\text{Int} \to \text{Bool})) \to (\text{Int} \to \text{Bool}))$

(iii) Reduce the following expression to normal form, showing each step in the reduction:

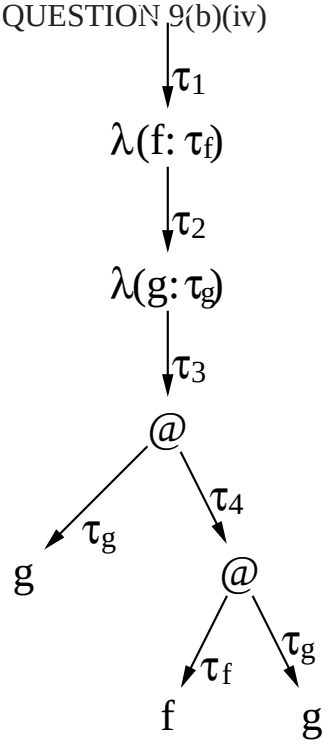
$$(\lambda\, x\, y\, z.\ x\ z\ (y\ z))\ f\ (\lambda\, x.\ x)$$

QUESTION 9(b)(iii) **[2 marks]**

$$\begin{aligned}(\lambda\, x\, y\, z.\ x\ z\ (y\ z))\ f\ (\lambda\, x.\ x) \\ &= (\lambda\, y\, z.\ f\ z\ (y\ z))\ (\lambda\, x.\ x) \\ &= \lambda\, z.\ f\ z\ ((\lambda\, x.\ x)\ z) \\ &= \lambda\, z.\ f\ z\ z\end{aligned}$$

(iv) Draw the syntax tree for the expression $\lambda f.\ \lambda g.\ g\ (f\ g)$ and extract type constraints from your syntax tree.

QUESTION 9(b)(iv) **[2 marks]**



$$\tau_1 = \tau_f \rightarrow \tau_2$$
$$\tau_2 = \tau_g \rightarrow \tau_3$$
$$\tau_g = \tau_4 \rightarrow \tau_3$$
$$\tau_f = \tau_g \rightarrow \tau_4$$

(v) Solve these type constraints to yield the most general type for the whole expression.

QUESTION 9(b)(v) **[1 mark]**

$$\tau_1 = \big((\tau_4 \rightarrow \tau_3) \rightarrow \tau_4\big) \rightarrow (\tau_4 \rightarrow \tau_3) \rightarrow \tau_3$$

(c) Prolog

This question concerns four different Prolog programs, called **A,B,C,D**.

For all four programs, the fact database contains the following two facts.

```
parent(a,b).
parent(b,c).
```

Each of the four programs also contains two rules, as follows.

A
$$\begin{cases} \texttt{ancestor(X,Y)} :- \texttt{ancestor(X,Z)}, \texttt{parent(Z,Y)}. \\ \texttt{ancestor(X,Y)} :- \texttt{parent(X,Y)}. \end{cases}$$

B
$$\begin{cases} \texttt{ancestor(X,Y)} :- \texttt{parent(X,Y)}. \\ \texttt{ancestor(X,Y)} :- \texttt{ancestor(X,Z)}, \texttt{parent(Z,Y)}. \end{cases}$$

C
$$\begin{cases} \texttt{ancestor(X,Y)} :- \texttt{parent(Z,Y)}, \texttt{ancestor(X,Z)}. \\ \texttt{ancestor(X,Y)} :- \texttt{parent(X,Y)}. \end{cases}$$

D
$$\begin{cases} \texttt{ancestor(X,Y)} :- \texttt{parent(X,Y)}. \\ \texttt{ancestor(X,Y)} :- \texttt{parent(Z,Y)}, \texttt{ancestor(X,Z)}. \end{cases}$$

For each program, the query

```
ancestor(A,B).
```

is run (and after a result is given, the system is prompted for further results). The four programs give these four different results.

1
$$\begin{cases} \texttt{A} = \texttt{a}, \texttt{B} = \texttt{b} \\ \texttt{A} = \texttt{b}, \texttt{B} = \texttt{c} \\ \texttt{A} = \texttt{a}, \texttt{B} = \texttt{c} \\ \text{finishes, no more results} \end{cases}$$

2
$$\begin{cases} \texttt{A} = \texttt{a}, \texttt{B} = \texttt{c} \\ \texttt{A} = \texttt{a}, \texttt{B} = \texttt{b} \\ \texttt{A} = \texttt{b}, \texttt{B} = \texttt{c} \\ \text{finishes, no more results} \end{cases}$$

3 returns no results, just runs forever

4
$$\begin{cases} \texttt{A} = \texttt{a}, \texttt{B} = \texttt{b} \\ \texttt{A} = \texttt{b}, \texttt{B} = \texttt{c} \\ \texttt{A} = \texttt{a}, \texttt{B} = \texttt{c} \\ \text{after these results, runs forever} \end{cases}$$

(i) State which of the four programs give each of the four results.

QUESTION 9(c)(i) **[3 marks]**

(ii) Give reasons for your answer to the last part. You need to show how the differences between the four observed behaviours are related to the differences between the four programs.

QUESTION 9(c)(ii) **[3 marks]**

(iii) In the course of executing a Prolog program, the Prolog engine needs to unify the following two terms:

```
arr(C, p(B, arr(D, A)))
arr(arr(cat, B), p(dog, C))
```

Unify these two terms, so as to obtain the most general unifier. Show clearly the steps in the algorithm you use.

QUESTION 9(c)(iii) **[2 marks]**

In the following solution, at each step we attack the first of the the list of pairs yet to be unified. For this we either decompose the pair of terms, or, if one of the terms is a variable we make that the next new substitution. For each new substitution, you need to apply that substitutution to (1) the substitututions previously obtained, and (2) the remaining pairs to be unified.

Pairs yet to be unified	**Substitutions found so far**
`arr(C,p(B,arr(D,A))) = arr(arr(cat,B),p(dog,C))`	
`C = arr(cat,B) ; p(B,arr(D,A)) = p(dog,C)`	
`p(B,arr(D,A)) = p(dog,arr(cat,B))`	`C = arr(cat,B)`
`B = dog ; arr(D,A) = arr(cat,B)`	`C = arr(cat,B)`
`arr(D,A) = arr(cat,dog)`	`B = dog ; C = arr(cat,dog)`
`D = cat ; A = dog`	`B = dog ; C = arr(cat,dog)`
`A = dog`	`D = cat ; B = dog ; C = arr(cat,dog)`
	`A = dog ; D = cat ; B = dog ; C = arr(cat,dog)`

so the substitutions found are `A = dog ; D = cat ; B = dog ; C = arr(cat,dog)` and the terms unify to `arr(arr(cat,dog),p(dog,arr(cat,dog)))`

(d) Computability Assume that the Halting Problem has no solution.

Consider a program *canHaltTest* which takes an (encoded) program *P* and returns a result indicating whether there is *some* input on which *P* halts. Show that no such program can be written.

QUESTION 9(d) **[8 marks]**

Additional answers. Clearly indicate the corresponding question and part.

Additional answers. Clearly indicate the corresponding question and part.

Additional answers. Clearly indicate the corresponding question and part.

Additional answers. Clearly indicate the corresponding question and part.

Appendix 1 — Natural Deduction Rules

$$(\wedge I)\quad \frac{p \qquad q}{p \wedge q} \qquad\qquad (\wedge E)\quad \frac{p \wedge q}{p} \qquad \frac{p \wedge q}{q}$$

$$(\vee I)\quad \frac{p}{p \vee q} \qquad \frac{p}{q \vee p} \qquad\qquad (\vee E)\quad \frac{p \vee q \qquad \begin{array}{c}[p]\\ \vdots\\ r\end{array} \qquad \begin{array}{c}[q]\\ \vdots\\ r\end{array}}{r}$$

$$(\rightarrow I)\quad \frac{\begin{array}{c}[p]\\ \vdots\\ q\end{array}}{p \rightarrow q} \qquad\qquad (\rightarrow E)\quad \frac{p \qquad p \rightarrow q}{q}$$

$$(\neg I)\quad \frac{\begin{array}{c}[p]\\ \vdots\\ q \wedge \neg q\end{array}}{\neg p} \qquad\qquad (\neg E)\quad \frac{\begin{array}{c}[\neg p]\\ \vdots\\ q \wedge \neg q\end{array}}{p}$$

$$(\forall I)\quad \frac{P(a) \qquad (a \text{ arbitrary})}{\forall x.\, P(x)}$$

$$(\forall E)\quad \frac{\forall x.\, P(x)}{P(a)}$$

$$(\exists I)\quad \frac{P(a)}{\exists x.\, P(x)}$$

$$(\exists E)\quad \frac{\exists x.\, P(x) \qquad \begin{array}{c}[P(a)]\\ \vdots\\ q\end{array} \qquad (a \text{ arbitrary})}{q \quad (a \text{ is not free in } q)}$$

Appendix 1a — Truth Table Values

p	q	$p \vee q$	$p \wedge q$	$p \rightarrow q$	$\neg p$	$p \leftrightarrow q$
T	T	T	T	T	F	T
T	F	T	F	F	F	F
F	T	T	F	T	T	F
F	F	F	F	T	T	T

Appendix 2 — Hoare Logic Rules

- Precondition Strengthening:

$$\frac{\{P_w\}\ S\ \{Q\} \qquad P_s \rightarrow P_w}{\{P_s\}\ S\ \{Q\}}$$

- Postcondition Weakening:

$$\frac{\{P\}\ S\ \{Q_s\} \qquad Q_s \rightarrow Q_w}{\{P\}\ S\ \{Q_w\}}$$

- Assignment:

$$\{Q(e)\}\ \texttt{x} := \texttt{e}\ \{Q(x)\}$$

- Sequence:

$$\frac{\{P\}\ S_1\ \{Q\} \qquad \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1;\ S_2\ \{R\}}$$

- Conditional:

$$\frac{\{P \wedge b\}\ S_1\ \{Q\} \qquad \{P \wedge \neg\, b\}\ S_2\ \{Q\}}{\{P\}\ \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{Q\}}$$

- While Loop:

$$\frac{\{P \wedge b\}\ S\ \{P\}}{\{P\}\ \textbf{while}\ b\ \textbf{do}\ S\ \{P \wedge \neg\, b\}}$$

Appendix 3 — Weakest Precondition Rules

$$
\begin{aligned}
wp(\texttt{x := e}, Q(x)) &\equiv Q(e) \\
wp(S_1;\ S_2, Q) &\equiv wp(S_1, wp(S_2, Q)) \\
wp(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, Q) &\equiv (b \rightarrow wp(S_1, Q)) \wedge (\neg\, b \rightarrow wp(S_2, Q)) \\
&\equiv (b \wedge wp(S_1, Q)) \vee (\neg\, b \wedge wp(S_2, Q))
\end{aligned}
$$

P_k is the weakest predicate that must be true before while b do S executes, in order for the loop to terminate after exactly k iterations in a state that satisfies Q.

$$
\begin{aligned}
P_0 &\equiv \neg\, b \wedge Q \\
P_{k+1} &\equiv b \wedge wp(S, P_k) \\
wp(\texttt{while } b \texttt{ do } S, Q) &\equiv \exists k.(k \geq 0 \wedge P_k)
\end{aligned}
$$

Appendix 4 — Lambda Calculus Typing Rules

$$\frac{x : \tau \ \in\ \Gamma}{\Gamma \ \vdash\ x \ ::\ \tau} \qquad \text{(Variable)}$$

$$\frac{\Gamma,\ x : \tau_1 \ \vdash\ e_2 \ ::\ \tau_2}{\Gamma \ \vdash\ \lambda(x : \tau_1).\ e_2 \ ::\ \tau_1 \rightarrow \tau_2} \qquad \text{(Abstraction)}$$

$$\frac{\Gamma \ \vdash\ e_1 \ ::\ \tau_{11} \rightarrow \tau_{12} \qquad \Gamma \ \vdash\ e_2 \ ::\ \tau_{11}}{\Gamma \ \vdash\ e_1\, e_2 \ ::\ \tau_{12}} \qquad \text{(Application)}$$