THE AUSTRALIAN NATIONAL UNIVERSITY

Second Semester 2010

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 (NatDed)	Q2 (StrInd)	Q3 (Z)	Q4 (Hoare)	Q5 (WPs)

Q6 (FSA)	Q7 (TM)	Q8 (CFG)	Q9 (1 of 3)	Total

QUESTION 1 [12 marks] (Natural Deduction)

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) Give a natural deduction proof of $\dfrac{p \to p \wedge q}{p \vee q \to q}$, that is, $\dfrac{p \to (p \wedge q)}{(p \vee q) \to q}$.

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

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

(b) What follows is a proof of $(p \vee q) \rightarrow (\neg p \rightarrow q)$ with the justifications, and the layout showing assumptions, missing.

$p \vee q$
$\neg p$
p
$\neg q$
$p \wedge \neg p$
q
q
q
q
$\neg p \rightarrow q$
$(p \vee q) \rightarrow (\neg p \rightarrow q)$

Reproduce the proof below, adding justifications which show which rules are used on each statement. Use the Fitch proof style which shows the scope of each assumption. (Hint: one of the lines is simply a repeat of a previous line).

QUESTION 1(b) **[4 marks]**

1	$p \vee q$	
2	$\neg\, p$	
3	p	
4	$\neg\, q$	
5	$p \wedge \neg\, p$	$\wedge$-I, 2, 3
6	q	$\neg$E, 4–5
7	q	
8	q	R, 7
9	q	$\vee$-E, 1, 3–6, 7–8
10	$\neg\, p \rightarrow q$	$\rightarrow$-I, 2–9
11	$(p \vee q) \rightarrow \neg\, p \rightarrow q$	$\rightarrow$-I, 1–10

(c) Give a natural deduction proof of $\dfrac{(\forall x.\, P(x)) \vee Q}{\forall x.\, P(x) \vee Q}$ that is $\dfrac{(\forall x.\, P(x)) \vee Q}{\forall x.\, (P(x) \vee Q)}$

Take care with parentheses and variable names.

QUESTION 1(c) **[4 marks]**

1	$(\forall x.\, P(x)) \vee Q$	
2	$\quad \forall x.\, P(x)$	
3	$\quad a \mid P(a)$	$\forall$-E, 2
4	$\quad\quad P(a) \vee Q$	$\vee$-I, 3
5	$\quad \forall x.\, P(x) \vee Q$	$\forall$-I, 4
6	$\quad Q$	
7	$\quad a \mid P(a) \vee Q$	$\vee$-I, 6
8	$\quad \forall x.\, P(x) \vee Q$	$\forall$-I, 7
9	$\forall x.\, P(x) \vee Q$	$\vee$-E, 1, 2–5, 6–8

or this alternative proof

1	$(\forall x.\, P(x)) \vee Q$	
2	$a \mid \quad \forall x.\, P(x)$	
3	$\quad\quad P(a)$	$\forall$-E, 2
4	$\quad\quad P(a) \vee Q$	$\vee$-I, 3
5	$\quad\quad Q$	
6	$\quad\quad P(a) \vee Q$	$\vee$-I, 5
7	$\quad P(a) \vee Q$	$\vee$-E, 1, 2–4, 5–6
8	$\forall x.\, P(x) \vee Q$	$\forall$-I, 7

QUESTION 2 [13 marks] (Structural Induction)

(a) Give an inductive proof of the fact that mapping a function over a list, and then filtering the result, can be achieved by filtering the list first, and then mapping. That is:

```
filter p (map f xs)  =  map f (filter (p.f) xs)
```

The definitions of `map`, `filter` and compose `(.)` are:

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

(p . f) x    =  p (f x)                                            -- C

filter p []     = []                                               -- F1
filter p (x:xs) = if p x then x : filter p xs else filter p xs     -- F2
```

(i) State and prove the base case goal

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

Base case:

```
filter p (map f [])  =  map f (filter (p.f) [])
```

Proof:

```
filter p (map f [])
  = filter p []                -- by (M1)
  = []                         -- by (F1)
  = map f []                   -- by (M1)
  = map f (filter (p.f) [])    -- by (F1)
```

(ii) State the inductive hypothesis

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

```
filter p (map f xs)  =  map f (filter (p.f) xs)  -- (IH)
```

(iii) State and prove the step case goal: where the goal is of the form $P(x : xs)$ (based on inductive hypothesis $P(xs)$), you need deal only with the case where `p (f x)` is true; you may omit the case where `p (f x)` is false.

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

Step Case: Show

```
filter p (map f (x:xs))  =  map f (filter (p.f) (x:xs))
```

Proof:

```
filter p (map f (x:xs))
  = filter p (f x : map f xs)     -- by (M2)
  = f x : filter p (map f xs)     -- by (F2)
  = f x : map f (filter (p.f) xs) -- by (IH)
  = map f (x : filter (p.f) xs)   -- by (M2)
  = map f (filter (p.f) (x:xs))   -- by (F2)
```

The first `(F2)` step uses `p (f x)` true ;
the second `(F2)` step uses `(p . f) x)` true, which follows from `(C)`.

(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 turn a tree into a list containing the same entries with the tail recursive function `flat`, where `flat t acc` returns the result of flattening the tree `t`, appended to the front of the list `acc`. Thus, for example,

`flat (Node 5 (Node 3 Nul Nul) (Node 6 Nul Nul)) [1,2]` = `[3,5,6,1,2]`

```
flat :: Tree a -> [a] -> [a]
flat Nul acc = acc                                  -- (F1)
flat (Node a t1 t2) acc =
    flat t1 (a : flat t2 acc)                       -- (F2)
```

We can get the sum of entries in a list by the function `sumL`

```
sumL :: [Int] -> Int
sumL []      = 0                                    -- (S1)
sumL (x:xs)  = x + sumL xs                          -- (S2)
```

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

```
sumT :: Tree Int -> Int
sumT  Nul           = 0                             -- (T1)
sumT (Node n t1 t2) = n + sumT t1 + sumT t2         -- (T2)
```

This question is about proving, by structural induction on the structure of the tree argument, that for all `t` and `acc`,

```
sumL (flat t acc) = sumT t + sumL acc
```

(i) State and prove the base case goal.

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

Base case: `t = Nul`.
Show that `sumL (flat Nul acc) = sumT Nul + sumL acc`
Proof:

```
sumL (flat Nul acc) = sumL acc              -- by (F1)
     = 0 + sumL acc                         -- by arithmetic
     = sumT Nul + sumL acc                  -- by (T1)
```

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 hypotheses.

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

∀ acc

```
sumL (flat t1 acc) = sumT t1 + sumL acc    -- (IH1)
sumL (flat t2 acc) = sumT t2 + sumL acc    -- (IH2)
```

(iii) State and prove the step case goal.

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

Show (assuming the IHs), that ∀ acc,

```
sumL (flat (Node y t1 t2) acc =
  sumT (Node y t1 t2) + sumL acc
```

Proof:

```
sumL (flat (Node y t1 t2) acc)
  = sumL (flat t1 (y : flat t2 acc))     -- by (F2)
  = sumT t1 + sumL (y : flat t2 acc)     -- by (IH1)
  = sumT t1 + y + sumL (flat t2 acc)     -- by (S2)
  = sumT t1 + y + sumT t2 + sumL acc     -- by (IH2)
  = sumT (Node y t1 t2) + sumL acc       -- by (T2)
```

Note: (IH1) is instantiated so that `acc` becomes `(y : flat t2 acc)`, but the `acc` of (IH2) is just instantiated to the `acc` of the proof.
Also, the proof of the step case goal involves an implicit ∀-I step.
Note: the proof involves steps which use associativity or commutativity of +; these steps are not explicitly shown

QUESTION 3 [11 marks] (Specification in Z)

The system we deal with in this question is that of a *shop* which sells *items* that are acquired from a wholesaler. Each item has a description, brand and value (which is its price from the wholesaler).

Given types for this system are *Item*, *Description* and *Brand*. There are global constants which are tables giving properties of *items*. We introduce them axiomatically:

$$
\begin{array}{|l}
dTable : Item \rightarrow Description \\
bTable : Item \rightarrow Brand \\
priceList : Item \rightarrow \mathbb{R} \\
\hline
\forall\, i, j : Item \bullet \; (dTable(i) = dTable(j) \;\land\; bTable(i) = bTable(j)) \;\Rightarrow\; i = j
\end{array}
$$

(a) Why is it appropriate that *bTable* is a total function?

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

Because we wish to specify that every item has a brand.

(b) What property of items is enforced by the predicate part of the above axiomatic definition?

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

It is the property that in any state of the system no two items have both the same brand and the same description.

The above three functions (*dTable* etc.) are constants for dealing with the wholesaler whereas the following constant, *margin* is the basis of how this shop computes a price for sales to its customers. A typical value for *margin* might be 40 indicating that the shop hopes to make a 40% profit on each item it sells.

$$
\begin{array}{|l}
margin : \mathbb{N}
\end{array}
$$

The state schema for the shop follows. The global variables for the system *Shop* tell how many of each item is in stock, which ones are on sale and how much cash is in the bank for acquisitions.

Shop

$$
\begin{array}{|l}
stock : Item \pfun \mathbb{N} \\
saleItems : \mathbb{P}\, Item \\
capital : \mathbb{R} \\
\hline
capital \geq 0 \\
\forall\, n \in \operatorname{ran} stock \bullet \; n > 0
\end{array}
$$

The operation of the shop acquiring a quantity $qty?$ of some item $i?$ is captured thus:

stockUp

$$\Delta Shop$$
$$i? : Item$$
$$qty? : \mathbb{N}$$

$$capital \geq qty? * priceList(i?)$$
$$i? \notin \mathrm{dom}\, stock \Rightarrow stock' = stock \cup \{i? \mapsto qty?\}$$
$$i? \in \mathrm{dom}\, stock \Rightarrow stock' = stock \setminus \{i? \mapsto stock(i?)\} \cup \{i? \mapsto stock(i?) + qty?\}$$
$$saleItems' = saleItems$$
$$capital' = capital - (qty? * priceList(i?))$$

(c) Express in English what the predicate part of the above schema for stock acquisition specifies, noting what the preconditions and postconditions are.

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

- The first predicate, the only precondition for the operation, specifies that there must be sufficient cash available for the purchase of stock.
- The 2nd and 3rd predicates, both postconditions, together specify that the stock table is updated to reflect the fact that the number of items $i?$ in stock is increased by $qty?$.
  Two clauses are necessary for the two cases depending on whether there was initially instances of items $i?$ in stock or not.
- The 4th predicate, a postcondition, specifies that the list of items on sale is unchanged by the operation.
- The 5th predicate, also a postcondition, specifies that the available cash is decreased by the product of a single item's price and the number acquired.

The global variable *saleItems* indicates what items are *'on sale'* at any particular time. The effect of being in this set is that the usual markup is halved when the shop sells such an item.

(d) Write a schema $EndSale_o$ for the operation of removing some set of items from the sale list. (This is the error-free case.)

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

$$
\begin{array}{|l}
\hline
EndSale_o \\
\Delta Shop \\
items? : \mathbb{P}\, Item \\
\hline
items? \subseteq saleItems \\
saleItems' = saleItems \setminus items? \\
stock' = stock \\
capital' = capital \\
\hline
\end{array}
$$

(e) Write a schema $Sell_o$ for the operation of selling a single item.
(Hint: Introduce a local variable, *cost*, which will be the price the customer will pay.)

QUESTION 3(e) **[4 marks]**

$$
\begin{array}{|l}
\hline
Sell_o \\
\Delta Shop \\
i? : Item \\
cost : \mathbb{N} \\
\hline
i? \in \mathrm{dom}\, stock \\
i? \notin saleItems \Rightarrow cost = priceList(i?) \times (1 + margin/100) \\
i? \in saleItems \Rightarrow cost = priceList(i?) \times (1 - margin/200) \\
stock(i?) = 1 \Rightarrow stock' = stock \setminus (i? \mapsto 1) \\
stock(i?) > 1 \Rightarrow stock' = stock \setminus (i? \mapsto stock(i?)) \cup (i? \mapsto (stock(i?) - 1)) \\
capital' = capital + cost \\
\hline
\end{array}
$$

QUESTION 4 [11 marks] (Hoare Logic)

The following piece of code, called *Divide* does integer division. It computes the integer part of the quotient of two non-negative numbers, *x* and *y* using repeated addition:

```
t := y;          } Init
q := 0;          }
while (t ≤ x)                          } Loop      } Divide
   t := t + y;   } Body                 }           }
   q := q + 1;   }                      }           }
```

We wish to use Hoare Logic (Appendix 2) to show that q is the quotient $x \div y$ – that is, $q * y$ is the largest multiple of y that does not exceed x. Formally, we want to prove

$$\{True\}\ Divide\ \{q * y \leq x \quad \wedge\ (q+1) * y > x\}$$

Because there is a loop involved, we need to capture our intuition about the loop as an invariant. The predicate $(t = (q+1) * y \quad \wedge \quad q * y \leq x)$, which we will call *INV*, has been suggested.

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 initialisation assignments as *Init*.

Assume that the computation domain is that of the natural numbers (i.e. *no negative integers*).

(a) Prove that the initialisation code will establish the alleged loop invariant, *INV*, starting in an arbitrary state. Be sure to properly justify each step of your proof.

QUESTION 4(a) **[3 marks]**

(b) Prove that *INV* really is a suitable loop invariant for *Loop*. That is,

$$\{INV \wedge (t \leq x)\}\ Body\ \{INV\}$$

All uses of precondition strengthening and postcondition weakening must be explicit but the arithmetic facts that they rely on need not be proved, although they must clearly be true.

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

(c) Using the previous results and some more proof steps show that

$$\{True\}\ Divide\ \{q * y \leq x \quad \wedge\ (q+1) * y > x\}$$

Be sure to properly justify each step of your proof.

QUESTION 4(c) **[4 marks]**

QUESTION 5 [11 marks] (Weakest Precondition Calculus)

The following piece of code, called *Divide* does integer division. It computes the integer part of the quotient of two non-negative numbers, x and y using repeated addition:

```
t := y;          } Init
q := 0;
while (t ≤ x)
   t := t + y;   } Body      } Loop      } Divide
   q := q + 1;
```

We wish to use the computation rules of the Weakest Precondition Calculus (Appendix 3) to show that q is the quotient $x \div y$; in other words, $q * y$ is the largest multiple of y that does not exceed x. Formally, we want to compute the weakest precondition for this program to compute the required answer – the required postcondition, *Post*, is $q * y \leq x \ \wedge \ (q+1) * y > x$.

(a) First of all we want to compute the weakest precondition applying just before the loop that guarantees the 'right state' when it exits. This exit condition must reflect not only *Post* (the program postcondition) but also the final state of the temporary variable t.

Use as the loop exit condition, Q, the predicate $t - y \leq x \ \wedge \ t > x \ \wedge \ t = (q+1) * y$.

Clearly, Q implies *Post*, but better describes the state. In fact,

$$Q \ \equiv \ Post \ \wedge \ t = (q+1) * y.$$

(i) Give, as simply as you can, an expression for P_0 for this loop.

QUESTION 5(a)(i) **[1 mark]**

$$\begin{aligned} P_0 \ &\equiv \ \neg\,(t \leq x) \ \wedge \ Q \\ &\equiv \ t > x \ \wedge \ t - y \leq x \ \wedge \ t > x \ \wedge \ t = (q+1) * y \\ &\equiv \ t - y \leq x \ \wedge \ t > x \ \wedge \ t = (q+1) * y \end{aligned}$$

(ii) Prove, in the usual way, that P_1 is the predicate

$$t \leq x \ \wedge \ t + y > x \ \wedge \ t = (q+1) * y.$$

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

$$\begin{aligned} P_1 \ &\equiv \ t \leq x \ \wedge \ wp(Body, \ P_0) \\ &\equiv \ t \leq x \ \wedge \ wp(\texttt{t:=t+y}, \ wp(\texttt{q:=q+1}, \\ &\qquad\qquad t - y \leq x \ \wedge \ t > x \ \wedge \ t = (q+1) * y \ ))) \\ &\equiv \ t \leq x \ \wedge \ t + y - y \leq x \ \wedge \ t + y > x \\ &\qquad\qquad\qquad\qquad \wedge \ t + y = ((q+1)+1)) * y \ ))) \\ &\equiv \ t \leq x \ \wedge \ t + y > x \ \wedge \ t = (q+1) * y \end{aligned}$$

(iii) Derive, in the same, an expression for P_2 for this loop.

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

$$\begin{aligned} P_2 &\equiv t \le x \ \wedge \ wp(Body,\ P_1) \\ &\equiv t \le x \ \wedge \ wp(\texttt{t:=t+y},\ wp(\texttt{q:=q+1}, \\ &\qquad\qquad t \le x \ \wedge \ t+y > x \ \wedge \ t = (q+1)*y)) \\ &\equiv t \le x \ \wedge \ t+y \le x \ \wedge \ t+y+y > x \ \wedge \ t+y = (q+1+1)*y \\ &\equiv t+y \le x \ \wedge \ t+2*y > x \ \wedge \ t = (q+1)*y \end{aligned}$$

(iv) Give (without the appropriate inductive argument) an expression for P_k.

QUESTION 5(a)(iv) **[1 mark]**

$$P_k \equiv t+(k-1)*y \le x \ \wedge \ t+k*y > x \ \wedge \ t = (q+1)*y$$

(v) Give an expression for $wp(Loop,\ Q)$. (Hint: Don't try to remove the quantifier yet, but otherwise simplify it.)

QUESTION 5(a)(v) **[1 mark]**

$$\begin{aligned} wp(Loop, Q) &\equiv \exists k \ge 0.\ P_k \\ &\equiv \exists k \ge 0.\ t+(k-1)*y \le x \ \wedge \ t+k*y > x \ \wedge \ t = (q+1)*y \\ &\equiv t = (q+1)*y \ \wedge \ \exists k \ge 0.\ t+(k-1)*y \le x \ \wedge \ t+k*y > x \end{aligned}$$

(b) Using logic and the rules for computing *wp*, derive a simple expression (without the quantifier) for $wp(Divide, Q)$.

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

$$\begin{aligned} wp(Divide, Q) &\equiv wp(\texttt{t:=y},\ wp(\texttt{q:=0},\ wp(Loop, Q))) \\ &\equiv wp(\texttt{t:=y},\ wp(\texttt{q:=0}, \\ &\qquad\qquad t = (q+1)*y \ \wedge \ \exists k \ge 0.\ t+(k-1)*y \le x \ \wedge \ t+k*y > x \\ &\equiv y = (0+1)*y \ \wedge \ \exists k \ge 0.\ y+(k-1)*y \le x \ \wedge \ y+k*y > x \\ &\equiv y > 0 \end{aligned}$$

(c) Describe how do we might proceed to derive an expression for $wp(Divide, Post)$. (Hint: What other result would help?)

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

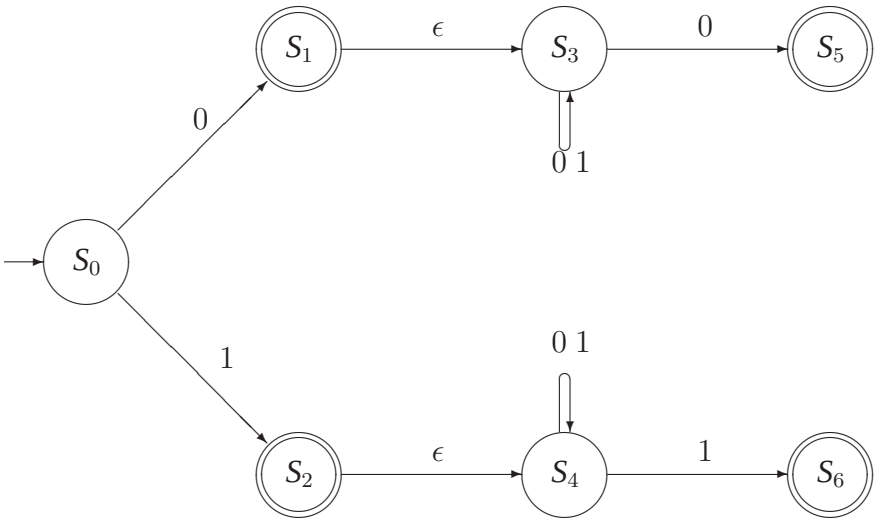
We have that $wp(Divide,\ Post \ \wedge \ t = (q+1)*y) \equiv y > 0$
Therefore $wp(Divide,\ Post) \ \wedge \ wp(Divide,\ t = (q+1)*y) \equiv y > 0$
By propositional logic, we deduce $y > 0 \ \Rightarrow \ wp(Divide,\ Post) \equiv True$
Since the program doesn't terminate for $y = 0$) we can say
$y = 0 \ \Rightarrow \ wp(Divide,\ Post) \equiv False$
That is, $wp(Divide,\ Post) \equiv y > 0$

(Nontermination for $y = 0$ is shown by proving $wp(Divide,\ True) \equiv y > 0$)

QUESTION 6 [12 marks] (Finite State Automata)

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

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

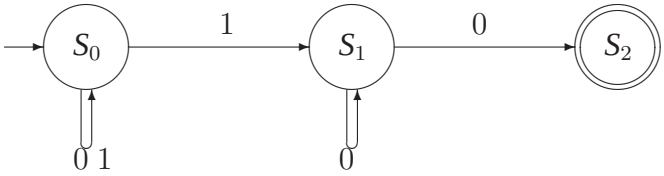


(b) Give a regular expression for this language.

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

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

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



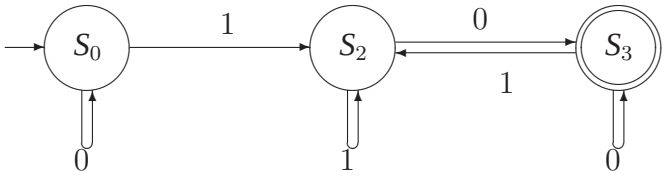
Describe the language L which is accepted by the automaton.

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

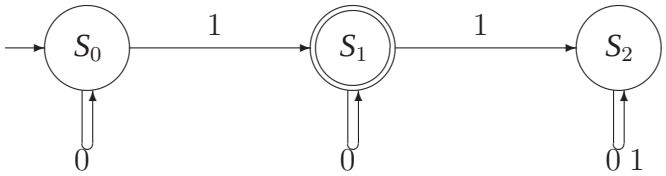
L is the set of strings over {0,1} which contain at least one '1' and finish with a '0'

(d) Design a *deterministic* FSA which recognises exactly the same language L.

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



(e) Consider the following DFA



Prove that the automaton does not accept strings ending in 11;
that is, prove that for any $\alpha \in \Sigma^*$, $N^*(S_0, \alpha 11)$ is not a final state.

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

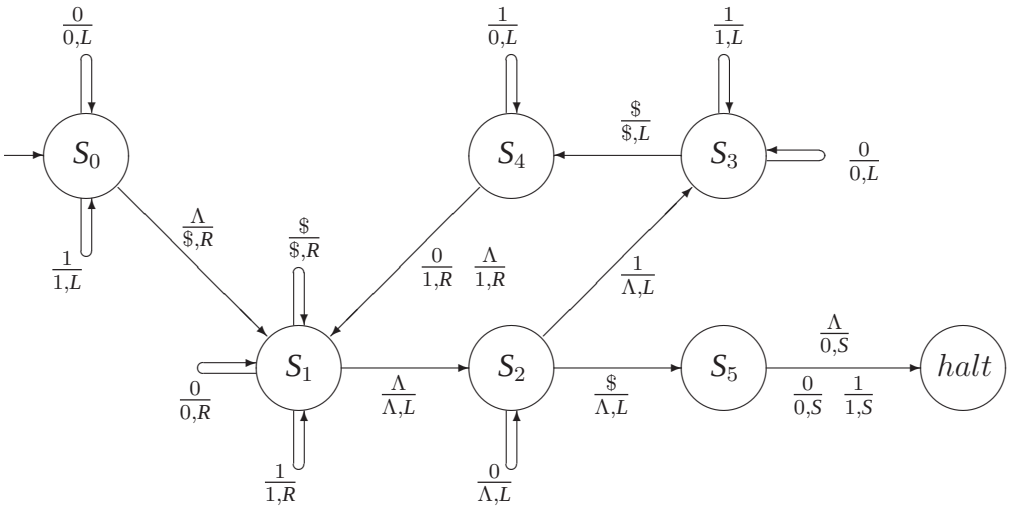
By the "Append" theorem, $N^*(S_0, \alpha 11) = N^*(N^*(S_0, \alpha), 11)$
Now $N^*(S_0, \alpha)$ is S_0, S_1 or S_2, so $N^*(S_0, \alpha 11) = N^*(S_i, 11)$ for $i = 0, 1$ or 2.
Also $N^*(S_i, 11) = N(N(S_i, 1), 1)$ by the definition of N^* (or you can think of it as like another instance of the "Append" theorem).
Thus $N^*(S_0, \alpha 11) = N(N(S_i, 1), 1)$ for some i. So we look at cases of S_i.

$$N(N(S_0, 1), 1) = N(S_1, 1) = S_2$$
$$N(N(S_1, 1), 1) = N(S_2, 1) = S_2$$
$$N(N(S_2, 1), 1) = N(S_2, 1) = S_2$$

So in all cases $N^*(S_i, 11) = S_2$, which is not in F.
That is, in all cases $N^*(S_0, \alpha 11) \notin F$, as required.

QUESTION 7 [12 marks] (Turing Machines)

(a) The following diagram shows a Turing machine, whose purpose is to perform some manipulation of a string of bits. Initially the head is somewhere over the input string which consists of '0's and '1's; the rest of the tape is blank.



(i) Give a general description of the purpose of state S_2.

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

From the right-most non-blank, proceed left, erasing 0 repeatedly, until the first 1, which is also erased.

(ii) If the given string is "101" what string is produced?

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

10

(iii) To what string is "00110" transformed?

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

10

(iv) Give a general description of the purpose of state S_4.

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

Add 1 to a binary number by starting at the right, moving left, changing 1 to 0, until a blank or 0 is encountered, which is changed to 1.

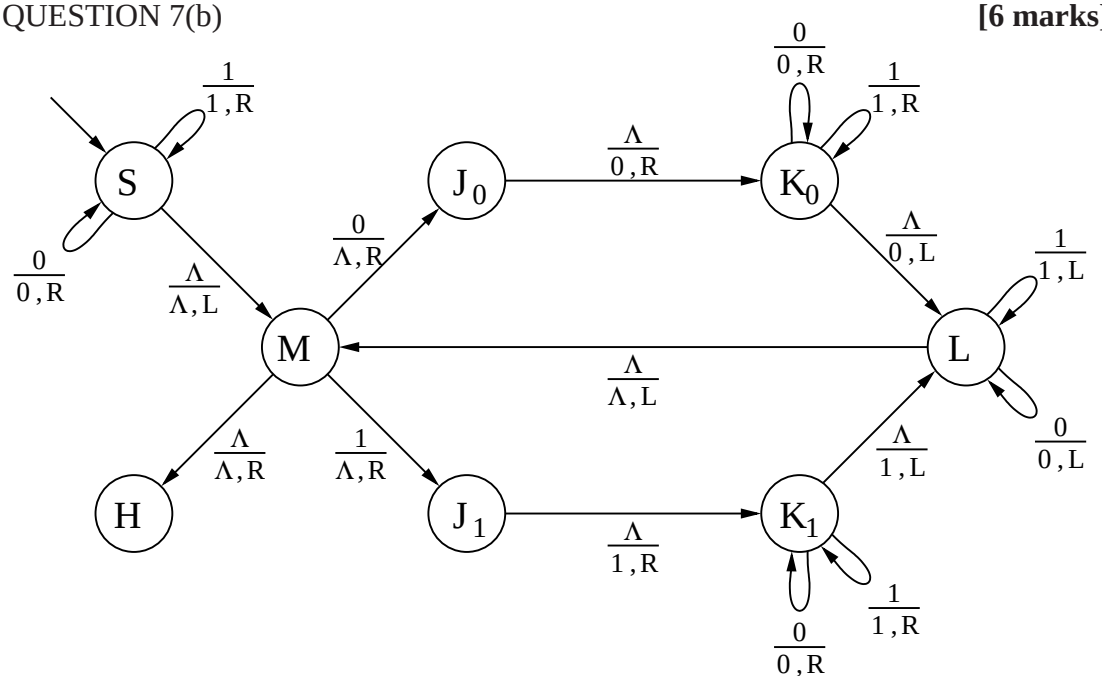
(v) Briefly describe what transformation this machine performs on an arbitrary number.

QUESTION 7(a)(v) **[2 marks]**

It counts the number of '1's in the initial string, and returns the string giving the count in binary

(b) Design a Turing Machine which constructs a palindrome from an arbitrary string over the alphabet $\{a, b\}$ by appending the reverse of the given string to the end of that given string. For example, "ba" will be transformed to "baab" and "bbab" will be replaced by "bbabbabb". Initially the given string is the only thing on the tape and the read head is positioned somewhere over it.

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



General idea: For an initial string $\alpha\beta$, an intermediate stage has $\alpha\Lambda\beta\beta'$ on the tape, where β' is the reverse of β. (The α is in its original location, the β is one cell to the right of its original location). Let $\alpha = \gamma i$. For one iteration of the main loop, starting from *M*, with the head pointing to the *i*, you

- erase *i*, but remember it by going to J_i
- overwrite the Λ with *i*
- go right, remembering *i* (you're at K_i) to beyond β'
- write *i*
- go left (state *L*) to the blank between the new α and β

(I've omitted detail of the start and finish).

QUESTION 8 [10 marks] (Context Free Grammars)

The following grammar describes a context free language:

$S ::= \epsilon \mid cS \mid Sc \mid aSbS$

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

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

Here are two parse trees for the string *c*

(b) How can you easily tell that this language is not a regular language?

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

The language contains all strings of the form $a^n b^n$, but does not contain any strings of the form $a^n b^m$ for $m \neq n$.

Imagine a DFA which accepts this language (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 '*b*'s would be accepted by the machine if it started from S_n.

Starting from S_n, the machine accepts b^n but does not accept b^m for any $m \neq n$.

That is, $N^*(S_n, b^n) = N^*(S_0, a^n b^n) \in F$, and
$N^*(S_n, b^m) = N^*(S_0, a^n b^m) \notin F$ for $m \neq n$.

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.

(c) Give a simple English description of the language that this grammar generates.

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

Strings over $\{a, b, c\}$ where

- the number of 'a's equals the number of 'b's
- whenever you divide the string into a left-hand part and a right-hand part, the left-hand part contains at least as many 'a's as 'b's

(d) Give a non-ambiguous grammar for the same language.

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

$$S ::= \epsilon \mid cS \mid aSbS$$

(e) Say why the language generated by the above grammar could be recognized by some *deterministic* PDA (and thus efficiently parsed). (You are not being asked to design the PDA.)

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

The only non-terminal is S. When S is on top of the stack (so non-terminal S is to be matched with the next portion of the input), the next terminal in the input needs to be c, for the production $S \rightarrow cS$, or a for the production $S \rightarrow aSbS$.

The other production is $S \rightarrow \epsilon$. For this production the next input symbol must be something that can immediately follow a string generated by S. Inspection of the right-hand sides of the productions shows that what follows S is either b or end of the input.

Thus we have

Next input symbol	Relevant production
a	$S \rightarrow aSbS$
c	$S \rightarrow cS$
b or $\dashv$	$S \rightarrow \epsilon$

So the next production to use at each step is determined by the next input symbol

QUESTION 9 [8 marks]

(Choice of *one* of three topics)

(a) Push-Down Automata

(b) Lambda Calculus and Types

(c) Prolog

Attempt just ONE of the three parts (a), (b) or (c) of this question (noting that each part has 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 table gives the transition function of a PDA with stack vocabulary $\{x, y, X, Z\}$ and input alphabet $\{x, y, \#\}$, where the hash symbol (#) marks the end of input. 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 $\{q_0, q_1, q_2\}$. The initial state is q_0 and the only final state is q_2. A string over $\{x, y\}^*$ is accepted by the PDA if the PDA consumes the string followed by a hash and ends up in the final state.

	x, Z	y, Z	x, x	y, x	x, y	y, y	x, X	y, X	$\#, Z$	$\#, x$	$\#, y$	$\#, Z$
q_0	q_0, xZ	q_0, yZ	q_1, x	q_0, X	q_0, X	q_0, yy	q_0, ϵ	q_0, Xy	q_2, ϵ	—	—	—
q_1	—	—	q_1, xx	q_0, ϵ	—	—	—	—	—	—	—	—
q_2	—	—	—	—	—	—	—	—	—	—	—	—

(Each entry in this table corresponds to one maplet of the transition function. For example, the top left entry corresponds to the mapping $\delta(q_0, x, Z) \rightarrow (q_0, xZ)$.)

(i) Give a trace of the acceptance of the input string *xyx*. (The input to the PDA, in this case will be *xyx#*.)

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

(ii) Give a trace of the acceptance of the input string *xxyyxx*.

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

(iii) What language does the PDA accept?

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

(iv) What does an *X* on the stack indicate?

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

(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:

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

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

$$(\lambda x.\ (\lambda v.\ \lambda y.\ (\lambda w.\ w)\ v)\ (\lambda u\ \lambda z.\ u)\ x)$$

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

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

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 substitututions 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:

$$(\lambda n.\ \lambda s.\ \lambda z.\ n\ s\ (s\ z))\ (\lambda s.\ \lambda z.\ s\ (s\ z))$$

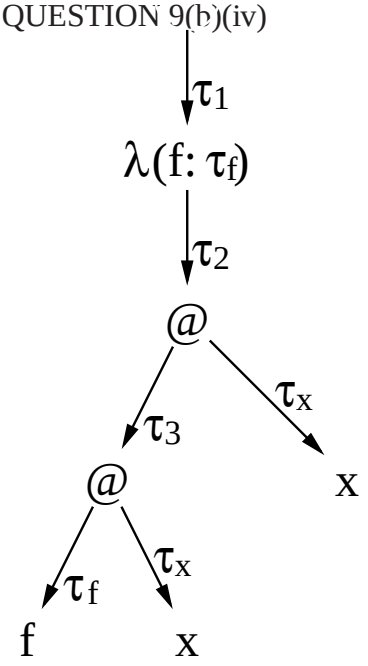
QUESTION 9(b)(iii) **[2 marks]**

It is easiest, and sometimes necessary, to rename repeated variables

$$\begin{aligned}(\lambda n.\ \lambda s.\ \lambda z.\ n\ s\ (s\ z))\ (\lambda s'.\ \lambda z'.\ s'\ (s'\ z')) \\ &= \lambda s.\ \lambda z.\ (\lambda s'.\ \lambda z'.\ s'\ (s'\ z'))\ s\ (s\ z) \\ &= \lambda s.\ \lambda z.\ (\lambda z'.\ s\ (s\ z'))\ (s\ z) \\ &= \lambda s.\ \lambda z.\ s\ (s\ (s\ z))\end{aligned}$$

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

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



$$\tau_1 = \tau_f \to \tau_2$$
$$\tau_3 = \tau_x \to \tau_2$$
$$\tau_f = \tau_x \to \tau_3$$

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

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

$$\tau_1 = (\tau_x \to \tau_x \to \tau_2) \to \tau_2$$

(c) Prolog

(i) Consider the following Prolog program:

```
alts([A|Xs],[A,B|Ys]) :- alts(Xs,Ys).     % (A1)
alts([B|Xs],[A,B|Ys]) :- alts(Xs,Ys).     % (A2)
alts([],[]).                              % (A3)
alts([A],[A]).                            % (A4)
```

Where `Ys` is given, the query `alts(Xs,Ys)` succeeds where `Xs` consists of one of the first two members of `Ys`, then one of the next two, and so on (then also, if `Ys` has odd length, the last member of `Ys`).

Thus the query `alts(Xs,[1,2,3,4,5]).` succeeds with the following four values for `Xs` (NOT in this order)

```
[2,4,5]    [1,4,5]    [1,3,5]    [2,3,5]
```

Show the correct order in which these solutions to the query appear, and explain why they appear in this order.

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

(ii) The Prolog program from the previous page is repeated here.

```
alts([A|Xs],[A,B|Ys]) :- alts(Xs,Ys).     % (A1)
alts([B|Xs],[A,B|Ys]) :- alts(Xs,Ys).     % (A2)
alts([],[]).                              % (A3)
alts([A],[A]).                            % (A4)
```

and we are considering the query `alts(Xs,[1,2,3,4,5]).`
One of its four solutions is `Xs = [1,4,5].`
Explain in detail the sequence of steps which leads to the the solution `Xs = [1,4,5].`
For each of these steps include the following information:

- The subgoal being attempted (such as, eg, "solve `alts(Ls,[1,3,4]`")
- Which rule (of (A1) to (A4)) is matched to that subgoal, in the course of producing the solution
- What solution is (ultimately) obtained for the variable (eg, `Ls`) in that subgoal

QUESTION 9(c)(ii) **[4 marks]**

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

```
arr(A, (arr(B, D), G))
arr(arr(int, G), (A, bool))
```

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)))`

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 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:

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

- 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

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

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

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

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

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