1. [8 marks] Recall the following definitions of map and foldr in ML-like languages.

```ml
  fun map f [] = []
  | map f (h::t) = (f h)::(map f t)
  fun foldr f e [] = e
  | foldr f e (h::t) = f (h, foldr f e t)
```

Consider the function

```ml
  fun mapr g L = foldr (fn (x,y) => (g x) :: y) nil L
```

Prove that for all lists \(L: \text{`a list}\) and functions \(g: \text{`a -> `a}\)

\[
\text{mapr } g \ L = \text{map } g \ L
\]

**Solution.**

**Proof:** Let \(f = \text{fn } (x, y) => (g x) :: y\)

**Basis.** \(\text{mapr } g \ [] = \text{foldr } f \ nil \ [] = \text{nil} = \text{map } g \ []\)

**Induction Hypothesis (IH).**

\[
\text{For all lists } L \text{ of length less than } n > 0, \ \text{mapr } g \ L = \text{map } g \ L
\]

**Induction Step.** Let \(L = z::t\) be a list of length \(n > 0\). Then

\[
\begin{align*}
  &= \text{mapr } g \ (z::t) \\
  &= \text{foldr } f \ nil \ (z::t) \\
  &= \text{foldr } (\text{fn}(x,y) => (g x) :: y) \ nil \ (z::t) \\
  &= (\text{fn}(x,y) => (g x)::y) \ (z,\text{foldr } f \ nil \ t) \\
  \{\text{Ind.hyp.}\} &= (\text{fn}(x,y) => (g x)::y) \ (z,\text{map } g \ t) \\
  &= (g z)::(\text{map } g \ t) \\
  &= \text{map } g \ (z::t)
\end{align*}
\]

QED

2. [5 + 6 = 11 marks] Each comment in the C language

- begins with the characters “//” and ends with the newline character, or
- begins with the characters “/*” and ends with “*/” and may run across several lines.

If the character sequences “//”, “/*” and “*/” are allowed to appear in ’quoted’ form as “//”, “/*” and “*/” respectively within a C comment, then give

(a) a regular expression for the modified C comments and
(b) a corresponding DFA for modified C comments

**Solution.** Here is a classic case of what reality some times looks like!
3. [10 marks] Recall that the Church numerals are defined as follows:

\[ 0 \overset{df}{=} \lambda f \ x[x], \quad 1 \overset{df}{=} \lambda f \ x[(f \ x)], \quad 2 \overset{df}{=} \lambda f \ x[(f \ (f \ x))], \ldots, \quad n+1 \overset{df}{=} \lambda f \ x[(f^{n+1} \ x)], \ldots \]

and the addition operation on them is defined as

\[ \text{add} \overset{df}{=} \lambda m \ n \ f \ x[((m \ f) \ (n \ f \ x))] \]
Prove that \( \text{add} \ u \ v =_\beta \text{add} \ v \ u \) for all Church numerals \( u \) and \( v \).

**Solution.**

Let \( u \equiv i \equiv \lambda g \ y \ (g^i \ y) \) and \( v \equiv j \equiv \lambda h \ z \ (h^j \ z) \) be any Church numerals representing the natural numbers \( i, j \geq 0 \) respectively. We then have

\[
\begin{align*}
\text{(add} \ u \ v) & \rightarrow^*_\beta \lambda f \ x[(u \ f) (v \ f \ x)] \\
& \rightarrow^*_\beta \lambda f \ x[(\lambda y[(f^i \ y)] \ (\lambda z[(f^j \ z)] \ x))]
\end{align*}
\]

Similarly we have

\[
\begin{align*}
\text{(add} \ v \ u) & \rightarrow^*_\beta \lambda f \ x[(v \ f) (u \ f \ x)] \\
& \rightarrow^*_\beta \lambda f \ x[(\lambda z[(f^j \ z)] \ (\lambda y[(f^i \ y)] \ x))]
\end{align*}
\]

From (??) and (??) it follows that \( \text{add} \ u \ v =_\beta \text{add} \ v \ u \).

4. \([6 + 6 = 12 \text{ marks}]\) Consider the term algebra defined by the following grammar

\[ t, u ::= \varepsilon \mid a \mid b \mid t.u \]

where “.” is an associative binary operator

\[
\begin{align*}
a.b & \rightarrow^\delta \varepsilon \\
\varepsilon.t & \rightarrow^\delta t \\
t.\varepsilon & \rightarrow^\delta t
\end{align*}
\]

(a) Give rules for the compatible closure of the rewrite rules.

(b) Prove that \( t \rightarrow^*_\delta \varepsilon \) if and only if the following conditions are satisfied

i. \( \#a(t) = \#b(t) \) and

ii. \( \#a(s) \geq \#b(s) \) for every prefix \( s \preceq t \)

where \( \#a(t) \) denotes the number of occurrences of \( a \) in \( t \).

**Solution.**

(a) The compatible closure defines a 1-step \( \delta \)-reduction

\[
\begin{align*}
\delta_1 & & \frac{t \rightarrow^\delta u}{t \rightarrow^\delta_i u} & \frac{t \rightarrow^\delta t'}{t \rightarrow^\delta_i t' \ u'} & \frac{u \rightarrow^\delta t}{t \rightarrow^\delta \delta_i u'}
\end{align*}
\]

(b) The terms of the algebra may be regarded as non-empty strings. It is useful to use a separate symbol \( \varepsilon \) to denote the empty string since \( \varepsilon \) already denotes a constant symbol in the algebra.

Let \( \#t = \#a(t) + \#b(t) + \#\varepsilon(t) \). The following claim then follows from the \( \delta \)-rules and the compatibility rules.

**Claim 0.1**

i. \( t \rightarrow^\delta t' \) implies \( \#t = \#t' + 1 \)
5. [5+5+5=15 marks] The Python language allows a “multiple-assignment” command. For example the following two programs (which compute the largest Fibonacci number under 100) are equivalent in Python.

```
#!/usr/bin/python
a, b = 0, 1
while b < 100:
    a, b = b, a+b
```

Assume that in addition to the simple assignment command, the pure WHILE language also has the following multiple-assignment command.

```
x1, x2 := e1, e2
```

(a) Define an operational rule of inference for the above command.

(b) From your operational rule prove that the following two programs are equivalent whenever x1 and x2 are distinct.

```
x1, x2 := e1, e2  |  x2, x1 := e2, e1
```

---

### Lemma 0.2

$t \rightarrow \varepsilon \Rightarrow \text{one (or more) of the following holds.}$

**Case (0)** $t = \varepsilon$

**Case (1)** $t = u.\varepsilon.v$

**Case (2)** $t = u.a.v.b.w$

where $u \neq \varepsilon \Rightarrow u \rightarrow \varepsilon$, $v \neq \varepsilon \Rightarrow v \rightarrow \varepsilon$ and $w \neq \varepsilon \Rightarrow w \rightarrow \varepsilon$.

**Proof:** We proceed by induction on $\#t$. Notice that for any actual term $t$, $\#t > 0$.

**Basis.** $\#t = 1$. Then $t \rightarrow \varepsilon$ iff $t \equiv \varepsilon$ which is merely case (1) with $u = \varepsilon = v$.

**Induction Hypothesis (IH).**

For any term $s$ with $1 \leq \#s < n$, $s \rightarrow \varepsilon \Rightarrow \text{one (or more) of the cases (0-2) holds.}$

**Induction Step.**

Let $\#t = n > 1$. Then $t \rightarrow \varepsilon$ iff for some term $t_1$, $t \rightarrow t_1 \rightarrow \varepsilon$ where $t_1$ satisfies the induction hypothesis. If $t \rightarrow t_1$ because of an application of either rule (??) or (??), then Case (1) applies. If $t \rightarrow t_1$ because of an application of rule (??) then Case (2) applies.

QED

**Corollary 0.3** $t \rightarrow \varepsilon$ implies $\#a(t) = \#b(t) \land \forall s \leq t[\#a(s) \geq \#b(s)]$

**Lemma 0.4** $\#a(t) = \#b(t) \land \forall s \leq t[\#a(s) \geq \#b(s)]$ implies $t \rightarrow \varepsilon$ for each term $t$.

**Proof:** We proceed by induction on $\#a(t)$ assuming that $\#a(t) = \#b(t) \land \forall s \leq t[\#a(s) \geq \#b(s)]$.

**Basis.** $\#a(t) = \#b(t) = 0$. then clearly $t \equiv \varepsilon$, for some $k \geq 1$. By applying rules (??) or (??), $k-1$ times the required result follows.

**Induction Hypothesis (IH).**

For every term $u$, $0 \leq \#a(u) = \#b(u) < n \land \forall s \leq u[\#a(s) \geq \#b(s)]$ implies $u \rightarrow \varepsilon$.

**Induction Step.** Assume $\#a(t) = \#b(t) = n > 0$. Then there exists a left-most occurrence of $b$ in the string $t$, $t = x.b.y$ for some terms $x$ and $y$ ($y$ could be empty, but $x$ is not). Since $x.b \leq t$ we have $\#a(x) = \#a(x.b) \geq \#b(x.b) = 1$ and $\#b(x) = 0$. Consider the right-most occurrence of $a$ in $x$. We then have for some terms $u$, $v$ (either or both of which could be empty) $x = u.a.v$ such that $\#a(v) = 0$. Since $\#b(x) = 0$ we also have $\#b(v) = 0$ which implies $v = \varepsilon$ for some $i \geq 0$. Hence $t \equiv u.a.\varepsilon.b.y \rightarrow t_1$ $u.a.b.y \rightarrow t_1$ $u.y \equiv t'$, where $\#a(t') = \#a(t) - 1 = \#b(t) - 1 = \#b(t')$. By the induction hypothesis $t' \rightarrow \varepsilon$. Hence $t \rightarrow \varepsilon$.

QED
(c) Give an example in the WHILE language to show that the following three programs all yield
different final states, even if they all begin execution from the same initial state, even if $x_1$
and $x_2$ are distinct.

\[
x_1, x_2 := e_1, e_2 \mid x_1 := e_1; x_2 := e_2 \mid x_2 := e_2; x_1 := e_1
\]

Solution.

(a) Assume in the following that $m_1$ and $m_2$ are normal forms of values of the appropriate type
compatible with the types of variables $x_1$ and $x_2$.

Then \textbf{Assgn2} is the required rule of inference.

\[
\begin{array}{c}
\langle \sigma, e_1 \rangle \rightarrow^* (\sigma, m_1) \\
\langle \sigma, e_2 \rangle \rightarrow^* (\sigma, m_2)
\end{array}
\]

\[
\gamma \vdash \langle \sigma, x_1, x_2 := e_1, e_2 \rangle \rightarrow^* \gamma(x_2) \mapsto m_2 \mid \gamma(x_1) \mapsto m_1 | \sigma
\]

\textbf{Assgn2} may be derived from the following three rules which outline exactly how the evaluation
of the expressions and the subsequent change of store takes place.

\[
\begin{array}{c}
\text{Assgn2.0} \\
\gamma \vdash \langle \sigma, x_1, x_2 := m_1, m_2 \rangle \rightarrow^* \gamma(x_2) \mapsto m_2 \mid \gamma(x_1) \mapsto m_1 | \sigma
\end{array}
\]

\[
\begin{array}{c}
\text{Assgn2.1} \\
\gamma \vdash \langle \sigma, e_2 \rangle \rightarrow \gamma(x_2) \mapsto \gamma(x_2) \mapsto m_2 \mid \gamma(x_1) \mapsto m_1 | \sigma
\end{array}
\]

\[
\begin{array}{c}
\text{Assgn2.2} \\
\gamma \vdash \langle \sigma, x_1, x_2 := e_1, e_2 \rangle \rightarrow^* \gamma(x_2) \mapsto m_2 \mid \gamma(x_1) \mapsto m_1 | \sigma
\end{array}
\]

- The two expressions $e_1$ and $e_2$ are evaluated in order. But the values stored in the
  locations of the variables $x_1$ and $x_2$ does not change until both expressions $e_1$ and $e_2$
  have been completely evaluated.
- It is possible to derive the rule \textbf{Assgn2} by induction on the sum of the number of steps it
takes to evaluate expressions $e_1$ and $e_2$ in sequence using the rules \textbf{Assgn2.2}, \textbf{Assgn2.1}
and \textbf{Assgn2.0}.

(b) Since $x_1$ and $x_2$ are distinct variables (they are not aliases of each other) and hence in any
environment $\gamma$, $\gamma(x_1) \neq \gamma(x_2)$. It follows that

\[
[\gamma(x_2) \mapsto m_2] \mid \gamma(x_1) \mapsto m_1 | \sigma = [\gamma(x_1) \mapsto m_1] \mid \gamma(x_2) \mapsto m_2 | \sigma
\]

Applying rule \textbf{Assgn2} to the programs

\[
x_1, x_2 := e_1, e_2 \mid x_2, x_1 := e_2, e_1
\]

and by the identity (??) we see that the two programs are equivalent.

(c) Let $\sigma_0$ be the initial state with $\sigma_0(\gamma(x_1)) = 1$ and $\sigma_0(\gamma(x_2)) = 2$. Let $e_1 \equiv x_1 + x_2 + 10$ and
$e_2 \equiv x_1 + x_2 + 20$. Then it is easy to see that the final states $\sigma_f$ are given by the following table.

\[
\begin{array}{cccccc}
\hline
\sigma_0 & \gamma(x_1) & \gamma(x_2) & \text{Program} & \sigma_f & \gamma(x_1) & \gamma(x_2) \\
\hline
1 & 2 & x_1, x_2 := x_1+x_2+10, x_1+x_2+20 & 13 & 23 \\
1 & 2 & x_1 := x_1+x_2+10; x_2 := x_1+x_2+20 & 13 & 35 \\
1 & 2 & x_2 := x_1+x_2+20; x_1 := x_1+x_2+10 & 34 & 23 \\
\hline
\end{array}
\]

6. [14 marks]
Using the type rules

\[ \Gamma \vdash x : \Gamma(x) \triangleright \emptyset \]

\[ \text{Abs} \quad \Gamma, x : \sigma \vdash L : \tau \triangleright_T C \]

\[ \Gamma \vdash \lambda x[L] : \sigma \rightarrow \tau \triangleright_T C \]

\[ \text{App} \quad \Gamma \vdash L : \sigma \triangleright_{T_1} C_1 \]

\[ \Gamma \vdash M : \tau \triangleright_{T_2} C_2 \]

\[ \Gamma \vdash (L M) : \tau \rightarrow \sigma \triangleright_{T'} C' \]  
(Conditions 1. and 2.)

where

- **Condition 1.** \( T_1 \cap T_2 = T_1 \cap TVar(\tau) = T_2 \cap TVar(\sigma) = \emptyset \)
- **Condition 2.** \( a \notin T_1 \cup T_2 \cup TVar(\sigma) \cup TVar(\tau) \cup TVar(C_1) \cup TVar(C_2) \).

- \( T' = T_1 \cup T_2 \cup \{a\} \)
- \( C' = C_1 \cup C_2 \cup \{\sigma = \tau \rightarrow a\} \)

give a formal proof of the principal type of the combinator \( S \equiv \lambda x \ y \ z[(x \ z) \ (y \ z)] \).

**Solution.**

1. \( x : 'a, y : 'b, z : 'c \vdash x : 'a \triangleright \emptyset \) \quad (Var)
2. \( x : 'a, y : 'b, z : 'c \vdash y : 'b \triangleright \emptyset \) \quad (Var)
3. \( x : 'a, y : 'b, z : 'c \vdash z : 'c \triangleright \emptyset \) \quad (Var)
4. \( x : 'a, y : 'b, z : 'c \vdash (y \ z) : 'd \triangleright \{a\} \) \quad \{b = 'c \rightarrow 'd\} \quad (App)
5. \( x : 'a, y : 'b, z : 'c \vdash (x \ z) : 'e \triangleright \{a, 'e\} \) \quad \{b = 'c \rightarrow 'd, 'a = 'c \rightarrow 'e\} \quad (App)
6. \( x : 'a, y : 'b, z : 'c \vdash ((x \ z) \ (y \ z)) : 'f \triangleright \{d, 'e, 'f\} \) \quad \{b = 'c \rightarrow 'd, 'a = 'c \rightarrow 'e, 'e = 'd \rightarrow 'f\} \quad (App)
7. \( x : 'a, y : 'b \vdash \lambda z[(x \ z) \ (y \ z)] : 'c \rightarrow 'f \triangleright \{a, 'e, 'f\} \) \quad \{b = 'c \rightarrow 'd, 'a = 'c \rightarrow 'e, 'e = 'd \rightarrow 'f\} \quad (Abs)
8. \( x : 'a \vdash \lambda y \ z[(x \ z) \ (y \ z)] : 'b \rightarrow 'c \rightarrow 'f \triangleright \{d, 'e, 'f\} \) \quad \{b = 'c \rightarrow 'd, 'a = 'c \rightarrow 'e, 'e = 'd \rightarrow 'f\} \quad (Abs)
9. \( \vdash \lambda x \ y \ z[(x \ z) \ (y \ z)] : 'a \rightarrow 'b \rightarrow 'c \rightarrow 'f \triangleright \{d, 'e, 'f\} \) \quad \{b = 'c \rightarrow 'd, 'a = 'c \rightarrow 'e, 'e = 'd \rightarrow 'f\} \quad (Abs)

Substituting from the constraints for \( 'a, 'b \) and \( 'e \) we get

\[ S : (c \rightarrow d \rightarrow e) \rightarrow (c \rightarrow d) \rightarrow c \rightarrow e \quad \] which is a principal type scheme for \( S \).