TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI
VICTORIA
UNIVERSITY OF WELLINGTON

EXAMINATIONS — 2011

END-OF-YEAR

SWEN 224
Formal Foundations of Programming
WITH ANSWERS

Time Allowed: 3 Hours

Instructions:

- Answer all **four** questions.
- The exam will be marked out of one hundred and eighty (180).
- Calculators ARE NOT ALLOWED.
- Non-electronic foreign language dictionaries are allowed.
- No other reference material is allowed.

Question 1. Assertions and Verification [60 marks]

(a) [15 marks : 5 for each part] Write an *assertion* formalising each of the following statements, where A is an array of size N. You may use either JML or ordinary mathematical notation.

(i) All elements of A are different.

$\forall i,j : 0 \leq i,j < N \wedge i \neq j : A[i] \neq A[j]$
or $\forall i,j : 0 \leq i < j < N : A[i] \neq A[j]$

(ii) A contains exactly one occurrence of z.

$\exists i : 0 \leq i < N : A[i] = z \wedge (\forall j : 0 \leq j < N \wedge i \neq j : A[j] \neq z)$
or $|\{i \mid 0 \leq i < N \wedge A[i] = z\}| = 1$

(iii) Between any two occurrences of z in A, there is at least one occurrence of y.

$\forall i,j : 0 \leq i < j < N : A[i] = z \wedge A[i] = z \Rightarrow (\exists k : i < k < j : A[j] = y)$

(b) [15 marks : 5 for each part] For each of the following correctness assertions, write down the *verification condition(s)* that must hold in order for the correctness assertion to be valid, and give a brief explanation of why these verification conditions hold.

(i) $\{ k = 0 \}\, s := 0 \,\{ s = \sum_{i=0}^{k-1} A[i] \}$

$k = 0 \Rightarrow 0 = \sum_{i=0}^{k-1} A[i]$

This holds because $\sum_{i=0}^{-1} A[i] = 0$.

(ii) $\{ x \leq y \}$ **if** $x > z$ **then** $x := z$ **else skip fi** $\{ x \leq y \wedge x \leq z \}$

$x \leq y \wedge x > z \Rightarrow z \leq y \wedge z \leq z$

This holds because: $x \leq y \wedge x > z$ implies $z \leq y$ (by transitivity), and $z \leq z$ always holds.

$x \leq y \wedge \neg\, (x > z) \Rightarrow x \leq y \wedge x \leq z$

The holds because $x \leq y$ implies $x \leq y$, and $\neg\, (x > z)$ is equivalent to $x \leq z$.

(iii) $\{ 0 \leq k < n-1 \wedge s = \sum_{i=0}^{k-1} A[i] \}\, k := k+1; s := s+A[k] \,\{ 0 \leq k < n \wedge s = \sum_{i=0}^{k-1} A[i] \}$

$0 \leq k < n \wedge s = \sum_{i=0}^{k-1} A[i] \wedge k \neq n-1 \Rightarrow 0 \leq k+1 < n \wedge s + a[k+1] = \sum_{i=0}^{k} A[i]$

This holds because:

- $0 \leq k$ implies $0 \leq k+1$
- $k < n \wedge k \neq n-1$ implies $k+1 < n$
- $s = \sum_{i=0}^{k-1} A[i]$ implies $s + a[k+1] = \sum_{i=0}^{k} A[i]$
  ($0 \leq k < n \wedge k \neq n-1$ ensures that $a[k+1]$ is well-defined)

(c) Consider the following Java method, which counts the number of times 0 occurs in an integer array A.

```
//@ requires A != null;
//@ ensures \result == (\num_of int k; 0 <= k && k < A.length; A[k] == 0);
int countZeroes(int[] A) {
  int i = 0;
  int c = 0;
  while (i < A.length) {
    if ( A[i] == 0 ) c = c + 1;
    i = i + 1;
  }
  return c;
}
```

(i) [4 marks] The `requires` and `ensures` annotations give the pre and postconditions for the method. What are the pre and postconditions for the loop?

Precondition:

```
//@ assert A != null && i == 0 && c == 0;
```

Postcondition:

```
//@ assert c == (\num_of int k; 0 <= k && k < A.length; A[k] == 0);
```

(ii) [5 marks] Give a loop invariant that can be used to verify the loop in this method.

```
/*@ loop_invariant A != null && 0 <= i && i <= A.length &&
          c == (\num_of int k; 0 <= k && k < i; A[k] == 0); @*/
```

(iii) [15 marks] State the three verification conditions that must be proved in order to show that the loop is partially correct, and give a brief argument to show that each of them holds. (You may use ordinary mathematical notation instead of JML if you prefer.)

- The precondition for the loop implies the loop invariant.

  $A \neq null \wedge i = 0 \wedge c = 0 \Rightarrow$
  $A \neq null \wedge 0 \leq i \wedge i \leq A.length \wedge c = (count\ k : 0 \leq k \wedge k < i : A[k] = 0)$

  $c = (count\ k : 0 \leq k \wedge k < i : A[k] = 0)$ holds because $c = 0$, $i = 0$, and *count* over an empty range is 0. The other conjuncts follow directly from the precondition, and the fact that array size is a natural number.

- The loop invariant is preserved when the loop test holds.

  When the if test holds:

  $A \neq null \wedge 0 \leq i \wedge i \leq A.length \wedge c = (count\ k : 0 \leq k \wedge k < A.length : A[k] = 0) \wedge i < A.length \wedge A[i] = 0 \Rightarrow$
  $A \neq null \wedge 0 \leq i+1 \wedge i+1 \leq A.length \wedge c+1 = (count\ k : 0 \leq k \wedge k < i+1 : A[k] = 0)$

This holds, because $(count\ k : 0 \leq k \wedge k < i+1 : A[k] = 0) = (count\ k : 0 \leq k \wedge k < A.length : A[k] = 0) + 1$ when $A[i] = 0$.

When the if test fails:

$A \neq null \wedge 0 \leq i \wedge i \leq A.length \wedge c = (count\ k : 0 \leq k \wedge k < i : A[k] = 0) \wedge i < A.length \wedge A[i] \neq 0 \Rightarrow$
$A \neq null \wedge 0 \leq i+1 \wedge i+1 \leq A.length \wedge c = (count\ k : 0 \leq k \wedge k < i+1 : A[k] = 0)$

This holds, because $(count\ k : 0 \leq k \wedge k < i+1 : A[k] = 0) = (count\ k : 0 \leq k \wedge k < A.length : A[k] = 0)$ when $A[i] \neq 0$.

- The loop invariant implies the postcondition when the loop test fails.

  $A \neq null \wedge 0 \leq i \wedge i \leq A.length \wedge c = (count\ k : 0 \leq k \wedge k < i : A[k] = 0) \wedge \neg\ (i < A.length) \Rightarrow$
  $c = (count\ k : 0 \leq k \wedge k < A.length : A[k] = 0)$

  This holds because $(count\ k : 0 \leq k \wedge k < i : A[k] = 0) = (count\ k : 0 \leq k \wedge k < A.length : A[k] = 0)$ when $i = A.length$.

(iv) [6 marks] Give a brief argument to show that the method terminates properly, i.e. that it does not give a run-time error or exception and does not loop forever. (You may ignore the possibility of arithmetic overflow.)

The only way the program could get a run-time error (ignoring the possibility of arithmetic overflow) is by getting an array index error. This cannot occur, however, because the loop invariant and loop test ensure that $0 \leq i < A.length$ in the if statement.

The program cannot loop forever because it terminates after $A.length$ iterations. We can prove this formally using $A.length - i$ as a loop variant. This value decreases on each iteration of the loop, and cannot go negative, since the loop will exit when it becomes zero.

Question 2. Alloy [20 marks]

Consider the following Alloy model for Nondeterministic Finite Acceptors:

```
sig Symbol {}

sig State {}

sig NFA {
  initial: State,
  next: State -> Symbol -> State,
  final: set State
}

sig Config {
  state: State,
  input: seq Symbol
}

pred move[m: NFA, c1, c2: Config] {
  c2.state = m.next[c1.state][c1.input.first]  &&
  c2.input = c1.input.rest
}

pred accepts[m: NFA, s: seq Symbol] {
  some ss: seq Config |
    ss.first.state = m.initial && ss.last.state in m.final &&
    (all i: ss.inds-#ss | move[m, ss[i], ss[i+1]])
}
```

(a) [3 marks] Write a predicate to determine whether an NFA has no final state.

```
pred noFinal[m: NFA] {
  no m.final
}
```

(b) [3 marks] Write a predicate to determine whether an NFA has at least two final states.

```
pred twoPlusFinals[m: NFA] {
  #m.final <= 2
}
```

(c) [3 marks] Write a predicate to determine whether an NFA is deterministic.

```
pred isDeterministic[m: NFA] {
  all s: State | all a: Symbol | #m.next[s][a] <= 1
}
```

(d) [3 marks] Write a predicate to determine whether an NFA accepts the empty string.

```
pred acceptsEmpty[m: NFA] {
  m.initial in m.final
}
```

(e) [3 marks] Write a predicate to determine whether the languages accepted by two NFAs have any strings in common.

```
pred accceptNonDisjointLanguages[m1: NFA, m2: NFA] {
  some s: seq Symbol | accepts[m1, s] and accepts[m2, s]
}
```

(f) [5 marks] Write a predicate to determine whether two NFAs has any common states; i.e. states that are reachable from the initial states of both machines.

```
pred reachable[m: NFA, s: State] {
  some ss: seq State |
    ss.first = s && ss.last = m.initial &&
    (all i: ss.inds-#ss | some a: Symbol | ss[i+1] in m.next[ss[i]][a])
}

pred CommonStates[m1, m2: NFA] {
  some s: State | reachable[m1, s] and reachable[m1, s]
}
```

or:

```
pred CommonStates'[m1, m2: NFA] {
  some s: State |
    some ss1, ss2: seq State |
      ss1.first = m1.initial && ss1.last = s &&
      ss2.first = m1.initial && ss2.last = s &&
      (all i: ss1.inds-#ss1 | some a: Symbol | ss1[i+1] in m1.next[ss1[i]][a]) &&
      (all i: ss2.inds-#ss1 | some a: Symbol | ss1[i+1] in m2.next[ss1[i]][a])
}
```

Question 3. Regular Languages [55 marks]

(a) [8 marks : 4 for each part] Write a *Regular Expression* or *Regular Grammar* to describe each of the following languages, over the alphabet $\{a, b, c\}$:

(i) All strings in which all a's come before all b's and all c's.

$a^*(b \mid c)^*$

(ii) All strings of even length in which all a's come before all b's.

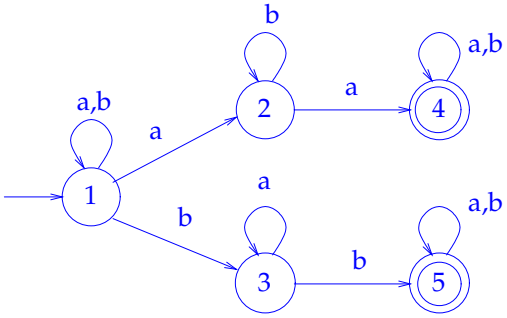
$((a \mid c)(a \mid c))^*((b \mid c)(b \mid c))^* \mid (a \mid c)((a \mid c)(a \mid c))^*(b \mid c)((b \mid c)(b \mid c))^*$

or: $((a \mid c)(a \mid c))^*(ab \mid \lambda)((b \mid c)(b \mid c))^*$

(b) Consider the NFA $M = (Q, q_I, A, N, F)$, where:

$$
\begin{aligned}
Q &= \{1, 2, 3, 4, 5\} \\
q_I &= 1 \\
A &= \{\mathrm{a}, \mathrm{b}\} \\
N &= \{(1,a,1), (1,a,2), (1,b,1), (1,b,3), (2,a,4), (2,b,2), \\
&\quad (3,a,3), (3,b,5), (4,a,4), (4,b,4), (5,a,5), (5,b,5)\} \\
F &= \{4, 5\}
\end{aligned}
$$

(i) [4 marks] Draw a transition diagram for M.



(ii) [4 marks] Show the sequence of configurations that M passes through in accepting the input *ababa*.

Note that you should show *all* states that M may be in after accepting part of the input.

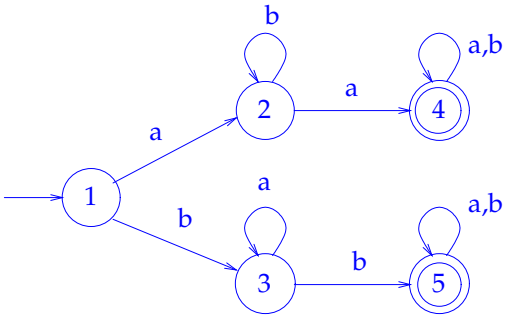
States	*Input*
$\{1\}$	*ababa*
$\{1,2\}$	*b*
$\{1,2,3\}$	*aba*
$\{1,2,3,\underline{4}\}$	*ba*
$\{1,2,3,\underline{4},\underline{5}\}$	*a*
$\{1,2,3,\underline{4},\underline{5}\}$	*Accept*

(iii) [4 marks] Write a regular expression describing the language accepted by M.

$(a \mid b)^*(ab^*a \mid ba^*b)(a \mid b)^*$

(i.e. all strings over a's and b's with either (at least) two a'a two b's.)

(iv) [8 marks] Draw a transition diagram for the DFA obtained by applying the *subset construction* to M. Show the correspondence between states of the DFA and those of the NFA. You only need to show reachable states.



The state correspondence is:

DFAState	*NFAStates*
1	$\{1\}$
2	$\{1,2\}$
3	$\{1,2,3\}$
4	$\{1,2,3,4\}$
5	$\{1,2,3,4\}$

(c) [10 marks] Given an NFA $M = (Q, q_I, A, N, F)$, show how to construct an NFA, $M' = (Q', q'_I, A', N', F')$, which accepts the language consisting of all strings in $\mathcal{L}(M)$ enclosed in a pair of a's. For example, if $\mathcal{L}(M) = \{a, aa, aaa\}$, then $\mathcal{L}(M') = \{aaa, aaaa, aaaaa\}$, and if $\mathcal{L}(M) = \{\lambda, b, c, bcb\}$, then $\mathcal{L}(M') = \{aa, aba, aca, abcba\}$.

Give a brief argument to show that the resulting NFA does in fact accept the required language.

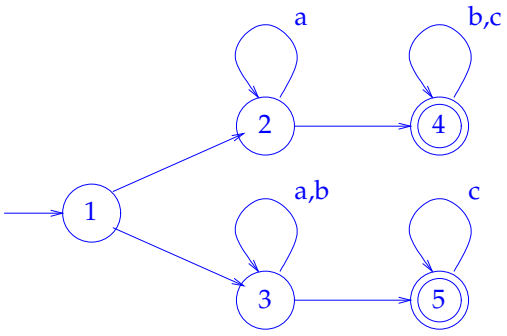
$M' = (Q', q'_I, A', N', F')$

where: $Q' = Q \cup \{q'_1, q'_2\}$, where q'_1 and q'_2 are new states, not in Q
$q'_I = q'_1$
$A' = A \cup \{a\}$
$N' = N \cup \{(q_1, a, q_I)\} \cup \{(q_f, a, q_2) \mid q_f \in F\}$
$F' = \{q_2\}$

M accepts a string γ iff M can traverse a path from q_I to some state $q_n \in F$, while consuming γ. But in this case, M' can consume $a\gamma a$ while passing through states $q'_1, q_I, ..., q_f, q'_2$, since M' has all of the transitions of M, and has transitions (q'_1, a, q_I), and (q_f, a, q'_2). So M' accepts $a\gamma a$.

Similarly, M' accepts a string γ iff γ is of the form $a\gamma' a$, where γ' is consumed on a path from q_I to some state $q_f \in F$, since the only transition out of q'_1 is (q'_1, a, q_I), and the only transitions leading to q'_2 are of the form (q_f, a, q'_2). But in this case, M can consume α while traversing the same path from q_I to $q_f F$. So M accepts γ.

(d) [6 marks] Draw a transition diagram for an NFA$_\epsilon$ that accepts the language defined by the regular expression $a^*(b \mid c)^* \mid (a \mid b)^* c^*$.



(e) [3 marks] Explain briefly why allowing null transition makes it easier to construct an NFA from a regular expression.

Allowing null transitions allows us to build the NFA in a modular way, using null transitions to "glue together" NFAs built from the subexpressions.

(f) [8 marks] Prove that, for any regular expressions x, y and z, $x(y \mid z) = xy \mid xz$.

We have to show that $\mathcal{L}(x(y \mid z)) = \mathcal{L}(xy \mid xz)$. We prove this as follows:

$$
\begin{array}{lll}
 & \mathcal{L}(x(y \mid z)) & \\
= & \mathcal{L}(x) \frown (\mathcal{L}(y) \cup \mathcal{L}(z)) & \text{Definition of } \mathcal{L} \\
= & \{s \frown v \mid s \in \mathcal{L}(x) \wedge v \in (\mathcal{L}(y) \cup \mathcal{L}(z))\} & \text{defn of } \frown \\
= & \{s \frown v \mid s \in \mathcal{L}(x) \wedge (v \in \mathcal{L}(y) \vee v \in \mathcal{L}(z))\} & \text{defn of } \cup \\
= & \{s \frown v \mid (s \in \mathcal{L}(x) \wedge v \in \mathcal{L}(y)) \vee (s \in S \wedge v \in \mathcal{L}(z))\} & \wedge \text{ distributes over } \vee \\
= & \{s \frown v \mid s \in \mathcal{L}(x) \wedge v \in \mathcal{L}(y)\} \cup \{s \frown v \mid s \in \mathcal{L}(x) \wedge v \in \mathcal{L}(z)\} & \text{defn of } \cup \\
= & (\mathcal{L}(x) \frown \mathcal{L}(y)) \cup (\mathcal{L}(x) \frown \mathcal{L}(z)) & \text{defn of } \frown \\
= & \mathcal{L}(xy \mid xz) & \text{Definition of } \mathcal{L}
\end{array}
$$

(The main part of this proof was given in the solutions to Assignment 4.)

Question 4. Context-Free Languages [45 marks]

(a) [10 marks : 5 for each part] Write a *Context Free Grammar* to describe each of the following languages. You are not required to give a full formal definition of these grammars — just write the list of rules.

(i) All strings consisting of one or more d's, optionally followed by an a and one or more further d's. For example, d, ddd, dad and $ddadd$ are in this language, but add, dda and $dada$ are not.

$$S \to T \mid T\,a\,T \qquad T \to d \mid d\,T$$

or

$$S \to T\,U \qquad T \to d \mid d\,T \qquad U \to \lambda \mid a\,T$$

(ii) All strings formed by concatenating two non-empty palindromes over $\{a, b, c\}$, where a palindrome is a strings that reads the same forwards and backwards (e.g. a, aa, aba, $abcabccbacba$). Thus, aa, $abaaba$, $abab$, $bbaaa$ and $abcabccbacbaaaa$ are in this language, but a, aba, $abbc$ and $abcabc$ are not.

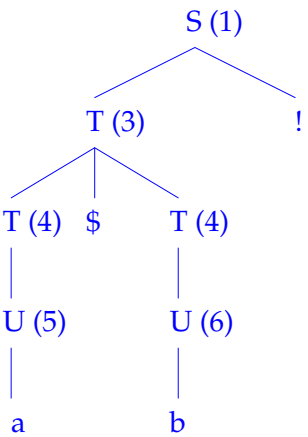
$$S \to T\,T$$
$$T \to U \mid a\,T\,a \mid b\,T\,b \mid c\,T\,c$$
$$U \to \lambda \mid a \mid b \mid c$$

(b) Consider the following grammar, where "!", "$", "$a$", b", "(" and ")" are terminal symbols:

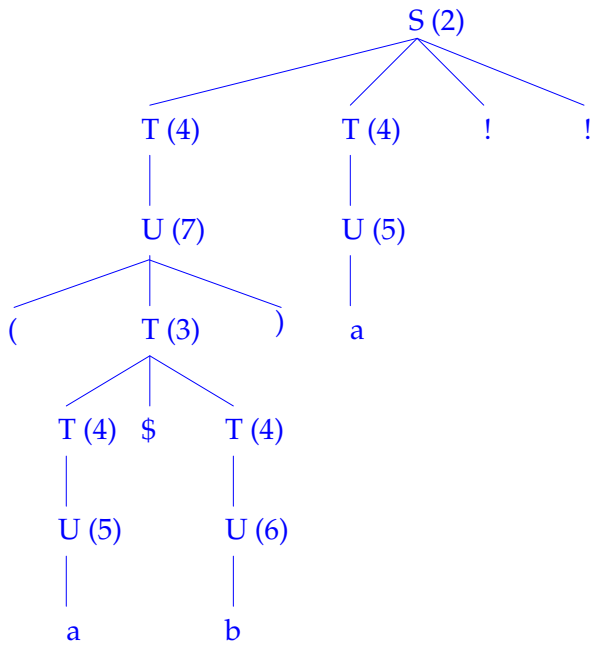
$$S \to T\,! \mid T\,T\,!\,! \quad (1,2)$$
$$T \to T\,\$\,T \mid U \quad (3,4)$$
$$U \to a \mid b \mid (\,T\,) \quad (5,6,7)$$

(i) [3 marks] Construct a parse tree for "$a\,\$\,b\,!$".

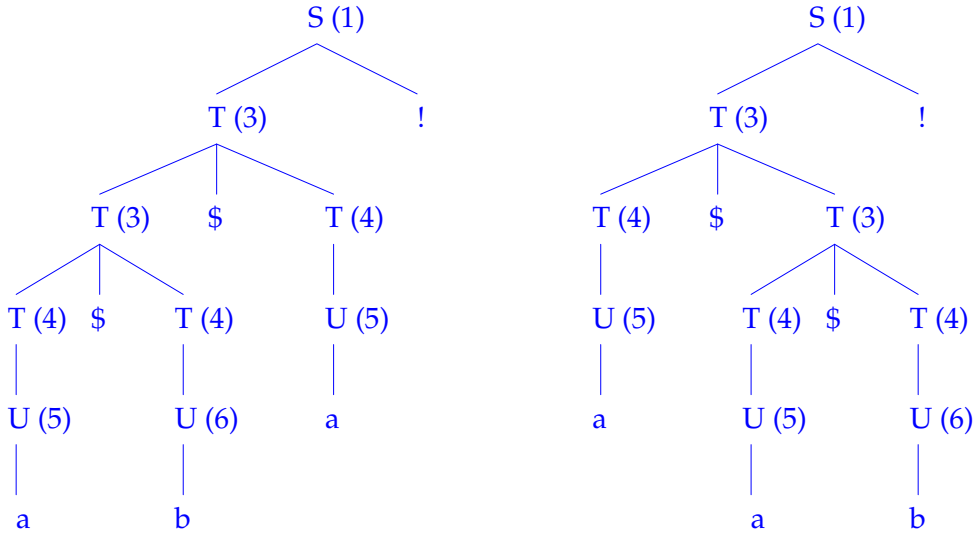
Write the number of the rule applied beside each nonterminal in the parse tree; likewise for parts **(ii)** and **(iii)**.



(ii) [4 marks] Construct a parse tree for "(a \$ b) a ! !".



(iii) [6 marks] Demonstrate that the grammar is ambiguous by drawing two differrent parse trees for "a \$ b \$ a !".



(iv) [6 marks] Write an equivalent, non-ambiguous grammar, treating "\$" as *left-associative*.

$$
\begin{array}{ll}
S \to T\,! \mid T\,T\,!\,! & (1,2) \\
T \to T\,\$\,U \mid U & (3,4) \\
U \to a \mid b \mid (\,T\,) & (5,6,7)
\end{array}
$$

(v) [8 marks] Write an equivalent LL(1) grammar, and show that it is LL(1).

$$
\begin{array}{ll}
S \to T\,S' & (1)\\
S' \to \,!\mid T\,!\,! & (2,3)\\
T \to U\,T' & (3,4)\\
T' \to \lambda \mid \$\,T' & (5,6)\\
U \to a \mid b \mid (\,T\,) & (7,8,9)
\end{array}
$$

In the definition of S', $first(!) = \{!\}$ and $first(T!!) = \{a, b, (\}$, which are disjoint.

In the definition of T', $first(\lambda) = \{\}$ and $first(\$T') = \{\$\}$, which are disjoint.

In the definition of U, $first(a) = \{a\}$, $first(b) = \{b\}$ and $first((T)) = \{(\}$, which are disjoint.

$T' \Rightarrow^* \lambda$, so we have to show that $first(T') \cap follow(T') = \varnothing$. $first(T') = \{\$\}$ and $follow(T') = \{!,)\}$, which are indeed disjoint.

(c) [8 marks] Prove that the union of two context-free languages is context-free.

Hint: Recall that a language is context-free if and only if it can be defined using a context-free grammar.

Suppose languages L_1 and L_2 are defined by CFG's $G_1 = (N_1, T_1, S_1, P_1)$ and $G_2 = (N_2, T_2, S_2, P_2)$, we show how to construct a grammar G_3 which defines $L_3 = L_1 \cup L_2$.

First, we rename nonterminals to ensure that N_1 and N_2 are disjoint. We can do this because renaming nonterminals does not alter the language defined by a gramamr.

The required grammar has all of the terminals, non-terminals and rules of the given grammars, along with one new nonterminal, S_3, which is the start symbol, and one new rule, allowing S_3 to be rewritten as either S_1 or S_2, i.e.:

$G_3 = (N_1 \cup N_2 \cup \{S_3\}, T_1 \cup T_2, S_3, P_1 \cup P_2 \cup \{S_3 \to S_1 \mid S_2\})$

To show that $\mathcal{L}(G_3) = L_1 \cup L_2$, we must show that for any string $\alpha \in (V_T \cup V_T')^*$ there is a parse tree for α from G_3 iff there is a parse tree for α from G_1 or from G_2.

Suppose that α is in L_1, then there must be a parse tree T for α from G_1, with S_1 as its root, and α as its fringe. But in that case, we can extend T to obtain a parse tree from α from G_3, by adding a new root, labelled S_3, with a single edge to the root of T. This is a parse tree from G_3 because it has S_3 as its root; the edge under the root corresponds to an application of the rule $S_3 \to S_1$, and the rest of the tree uses rules form G_1 which are also in G_3. Similarly, if α is in L_2, there is a parse tree from α from G_2, which can be extended to obtain a parse tree for α from G_3 by adding an application of the rule $S_3 \to S_2$.

In the other direction, suppose that T is a parse tree for α from G_3. Since the only rule in G_3 with S_3 on the left hand side is $S_3 \to S_1 \mid S_2$, the root of T has a single child, labelled either S_1 or S_2. Thus, if we delete the root and that edge, the resulting tree is either a parse tree for α from G_1 or a parse tree for α from G_2, so α is either in L_1 or in L_2, i.e. it is in $L_1 \cup L_2$.

****