Victoria University of Wellington

DEGREE EXAMINATIONS — 1997

COMP 202

END OF YEAR

COMP 202

Formal Methods of Computer Science

Time Allowed: 3 Hours

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

This exam will be marked out of 100.

continued...

Question 1. [24 marks]

For each of the languages described below:

(i) Say whether the language is regular, context free but not regular, or not context free.

(ii) If the language is regular, write a **regular expression** which defines it.

If the language is context free but not regular, write an **extended context free grammar** which defines it.

If the language is not context free, write a **context free covering grammar**. Explain what constraints on sentences in the language are not imposed by your grammar, and why these constraints cannot be imposed using a context free grammar.

Your covering grammar should enforce any constraints that can reasonably be imposed using a context free grammar.

The languages are:

(a) The set of all strings over $\{a, b, c\}$ where each string contains exactly two c's, and there are no b's between the two c's. For example, "*cc*", "*acac*", "*bbbccaaa*" and "*abcaacbba*" are in this language, but "*aca*", "*cbc*" and "*cccb*" are not. [4 marks]

(i) Regular
(ii) $(a|b)^*ca^*c(a|b)^*$

(b) The set of all strings over $\{a, b, c\}$ where each string contains exactly one c, and the number of a's before the c is equal to the number of a's after the c. For example, "*c*", "*aca*", "*abbbaabacbbaabbbbaa*" and "*ababcaabbb*" are in this language, but "*cc*", "*aaca*" and "*bbbca*" are not. [4 marks]

(i) Context Free
(ii) $S \rightarrow aSa \mid bS \mid Sb \mid c$
Note that this grammar is ambiguous. I think the following is an unambiguous grammar for this — an unambiguous grammar is certainly more complicated!

$$\begin{aligned} S &\rightarrow aSa \mid bSb \mid bU \mid Vb \mid c \\ U &\rightarrow aSa \mid bU \mid c \\ V &\rightarrow aSa \mid Vb \mid c \end{aligned}$$

(c) The set of all strings over $\{a, b, c\}$ where each string contains exactly one c, and the sequence of a's and b's before the c is identical to that following the c. For example, "*c*", "*aca*", "*aaacaaa*" and "*ababbacababba*" are in this language, but "*cc*", "*aaca*", "*bbbca*", "*abbbaabacbbaabbbbaa*" and "*ababcaabbb*" are not. [4 marks]

(i) Not context free
(ii) Covering grammar is $S \rightarrow ASA \mid c$
$A \rightarrow a \mid b$
This ensures that we get the same number of symbols before and after then c, but doesn't ensure that they are the same symbols. We can't impose that requirement using a CFG, because it can't be checked by pairing symbols in a nested fashion.

(d) The set of all strings of the form $a^k b^l c^m d^n$, where $k, l, m, n \geq 0$, and $k + l = m + n$. For example, "*ac*", "*abcd*", "*aacd*", "*aabccd*" and "*aaabccdd*" are in this language, but "*aabc*", "*badc*" and "*aaabbccdd*" are not. [4 marks]

(i) Context free
(ii)
$$\begin{aligned} S &\rightarrow aSd \mid aTc \mid bUd \mid V \\ T &\rightarrow aTc \mid V \\ U &\rightarrow bUd \mid V \\ V &\rightarrow bVc \mid \lambda \end{aligned}$$

(e) The set of all strings of the form $\alpha_1\beta_1 \pm \cdots \pm \alpha_n\beta_n$, where $n > 0$, each $\pm$ is either "+" or "−", and for each $i = 1, \cdots, n$, α_i is an optional natural number (a sequence of zero or more digits) and β_i is an identifier (a sequence of one or more letters). For example, "$2x$", "$x - y$" and "$23ab + 4x - 143x$" are in this language, but "$x2 - 3y$" and "$2x + -3y$" are not. [4 marks]

In defining this language, you should use "d" to denote a digit and "l" to denote a letter.

(i) Regular
(ii) $d^*l^+((+|-)d^*l^+)^*$

(f) The set of all strings over $\{0, 1\}$ where the distances between successive 1's form a strictly increasing sequence. For example, "11", "000", "000101000100" and "1101001" are in this language, but "111", "0010101" and "1010001001" are not. [4 marks]

(i) Not context free
(ii) The language has no interesting context-free structure, so the best we can do is to describe it by a regular expression (in the form of a one rule ECFG). Here are three possible ways to write the definition:

$$S \rightarrow \{0\}\{1\{0\}\}$$
$$S \rightarrow \{0\}\{1\{0\}\}[1\{0\}]$$
$$S \rightarrow \{0\}[1\{\{0\}1\}]\{0\}$$

These grammars doesn't impose the constraint that the sequences of 0s have to be of increasing length — we can't check that by pairing symbols in a nested fashion. (These are also equivalent to $S \rightarrow \{1|0\}$, but that doesn't allow us to extract sequences of 0.)

Question 2. [21 marks]

Let L_1 be the set of all strings over $\{a, b, c\}$ that start with "ab", and L_2 be the set of all strings over $\{a, b, c\}$ that end with "ba".

L_1 can be described by the regular expression $ab(a|b|c)^*$, and L_2 can be described by the regular expression $(a|b|c)^*ba$.

(a) Draw a transition diagram for a DFA to recognise L_1. [2 marks]

Transition table is:

	a	b	c
1	2	-	-
2	-	3	-
3*	3	3	3

(b) Draw a transition diagram for a DFA to recognise L_2. [2 marks]

You are not required to use a specific method to construct these DFAs.

Transition table is:

	a	b	c
1	1	2	1
2	3	2	1
3*	1	2	1

(c) Construct a DFA to recognise the intersection of L_1 and L_2 (i.e. $L_1 \cap L_2$, the set of all strings that begin with "ab" and end with "ba"), by applying the "pair machine" construction to the two DFAs in parts (a) and (b). [8 marks]

Transition table is:

	a	b	c
1:(1,1)	2:(2,1)	–	–
2:(2,1)	–	3:(3,2)	–
3:(3,2)	5:(3,3)	3:(3,2)	4:(3,1)
4:(3,1)	4:(3,1)	3:(3,2)	4:(3,1)
5:(3,3)*	4:(3,1)	3:(3,2)	4:(3,1)

(d) What is the shortest string accepted by your machine in part (c)? [1 mark]

"*aba*"

(e) Give a set of equations relating regular expressions describing the sets of strings that can be accepted starting in each state of your DFA in part (c). (These were called "*From* sets" in the Course Notes).

Solve these equations to obtain a regular expression describing the language accepted by your DFA. [8 marks]

You should simplify your regular expression, so as to write it in its simplest form.

Equations:
Writing A as the RE for state 1, B as the RE for state 2, etc.
$A = aB$
$B = bC$
$C = aE|bC|cD$
$D = aD|bC|cD$
$E = aD|bC|cD|\lambda$
Solving these (with a bit of simplification), gives:
$A = ab((a|c)^*b)^*a$
which is equivalent to $ab[(a|b|c)^*b]a$

Question 3. [30 marks]

Consider a variant of the while program language, called the "loop language", with the following syntax:

program	$\rightarrow$	*statement-list*
statement-list	$\rightarrow$	*statement* { ";" *statement* }
statement	$\rightarrow$	*assignment-statement*
	$\mid$	*if-statement*
	$\mid$	*do-statement*
	$\mid$	*exit-statement*
assignment-statement	$\rightarrow$	*variable-name* ":=" *arithmetic-expression*
if-statement	$\rightarrow$	"**if**" *Boolean-expression* "**then**" *statement-list* ["**else**" *statement-list*] "**fi**"
do-statement	$\rightarrow$	"**do**" *statement-list* "**od**"
exit-statement	$\rightarrow$	"**exit**"

In this language, a *do-statement*, **do** S **od**, repeatedly executes the enclosed statement, S, until terminated by an *exit-statement*. An *exit-statement* causes the closest enclosing *do-statement* to terminate. (Input and output statements are omitted for simplicity.)

For example, the following is a loop program which compares two character strings, A and B, which are assumed to be both of length n. The program returns (as the value of r) 0 if A and B are identical, -1 if A would come before B in an alphabetical listing, and 1 if A would come after B in an alphabetical listing.

```
k := 0; r := 0;
do
    if k = n then exit fi;
    k := k + 1;
    if A[k] < B[k] then r := −1; exit fi;
    if A[k] > B[k] then r := 1; exit fi
od
```

(a) Suppose you wanted to translate loop programs into strongly equivalent flowchart programs. Explain how you would translate *if-statements*, *do-statements* and *exit-statements*. [5 marks]

You should describe the transformation in a similar manner to the transformation T_{wf} given in the Course Notes, if you can. It will be possible to get full marks for a clear description of the transformation expressed in English.

(b) Show the flowchart program obtained by applying your translation from part (a) to the loop program given above. [5 marks]

(c) Show that the flowchart program you gave in part (b) is strongly equivalent to the given loop program, by finding a regular expression describing the set of possible execution paths for each program. [5 marks]

(d) Write the procedures you would use in a recursive descent parser to parse *if-statements*, *do-statements* and *exit-statements*, based on the grammar given on page 6.

Extend these procedures so that they implement the translation described in part (a).

[10 marks]

You do not need to give the plain/unextended procedures separately, so long as the extensions are clearly marked and explained. In implementing the translation, you should assume suitable functions are available to create the structures required in the flowchart, such as labels and various statement types.

(e) Suppose we extend the loop language to provide multi-level loop exits. We allow a *do-statement* to be labelled, and allow a label to be provided in an *exit-statement* to indicate which loop is to be exited.

For example, we might write a program to search for a given value, x, in a $n \times n$ array A as follows:

```
$i := 1$; $r := 0$;
outer:
    **do**
        **if** $i > n$ **then exit fi**;
        $j := 1$;
        **do**
            **if** $j > n$ **then exit fi**;
            **if** $A[i, j] = x$ **then** $r := 1$; **exit** outer **fi**;
            $j := j + 1$
        **od**;
        $i := i + 1$
    **od**
```

Explain how you would modify you parser procedures in part (d) to implement multi-level exits. [5 marks]

Question 4. [25 marks]

The following is an algorithm to compute (as the value of z) the intersection of two sets, X and Y:

```
$x := X$; $y := Y$; $z := \emptyset$;
while $x \neq \emptyset \ \wedge \ y \neq \emptyset$ do
    if $min(x) < min(y)$ then
        $x := x - \{ min(x) \}$
    elsif $min(x) > min(y)$ then
        $y := y - \{ min(y) \}$
    else
        $z := z \cup \{ min(x) \}$;
        $x := x - \{ min(x) \}$;
        $y := y - \{ min(y) \}$
    fi
od
```

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

You should think carefully about the relationships between the variables that this algorithm relies upon. In particular, think about why the algorithm only needs to compare the minimum of one set with the minimum of the other, rather than comparing it with every element of the other set. This has something to do with the relationships between the parts of X and Y that have already been examined (i.e. $X - x$ and $Y - y$) and the parts that have yet to be examined (i.e. x and y).

$u \subseteq x \ \wedge \ v \subseteq y \ \wedge \ x - u \cup y - v < u \cup v \ \wedge \ z = (x - u) \cap (y - v)$
where $s_1 < s_2$ means $(\forall p \in s_1, q \in s_2) p < q$.

(b) Use your loop invariant from part (a) to prove that the program correctly computes the intersection of x and y, by showing that:

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

When $u = x \ \wedge \ v = y \ \wedge \ z = \emptyset$, we clearly have

- $u \subseteq x$, since $u = x$
- $v \subseteq y$, since $v = y$
- $x - u \cup y - v < u \cup v$, since $x - u \cup y - v = \emptyset$
- $z = (x - u) \cap (y - v)$, since $z = \emptyset$ and $(x - u) \cap (y - v) = \emptyset$ because $(x - u) = (y - v) = \emptyset$.

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

continued...

Since the loop body is entered, we can assume that $u \neq \emptyset \ \wedge \ v \neq \emptyset$ holds.
We need to consider three cases, according to which branch of the **if** is taken.

- When $min(u) < min(v)$ we just remove $min(u)$ from u.

  In this case, we have:

  - $u \subseteq x$, because we have just removed an element from u which was already a subset of x,
  - $v \subseteq y$, because neither has changed,
  - $x - u \cup y - v < u \cup v$, because we have added one element to $x - u$ which is smaller that anything else in u and smaller than everything in v, and
  - $z = (x - u) \cap (y - v)$, because we have added one element to $x - u$ which is not in y (and therefore not in $y - v$).

- When $min(u) > min(v)$, we remove $min(v)$ from v.

  In this case, we have:

  - $u \subseteq x$, because neither has changed,
  - $v \subseteq y$, because we have just removed an element from v which was already a subset of y,
  - $x - u \cup y - v < u \cup v$, because we have added one element to $y - v$ which is smaller that anything else in v and smaller than everything in u, and
  - $z = (x - u) \cap (y - v)$, because we have added one element to $y - v$ which is not in x (and therefore not in $x - u$).

- When $min(u) = min(v)$, we add $min(u)$ to z, remove $min(u)$ from u and $min(v)$ from v.

  In this case, we have:

  - $u \subseteq x$, because we have just removed an element from u which was already a subset of x,

    $v \subseteq y$, because we have just removed an element from v which was already a subset of y,
  - $x - u \cup y - v < u \cup v$, because we have added one element to $x - u$ which is smaller that anything else in u and smaller than everything in v and added one element to $y - v$ which is smaller that anything else in v and smaller than everything in u, and
  - $z = (x - u) \cap (y - v)$, because we have added one element to a, $x - u$ and $y - v$. I.e. $z \cup \{p\} = ((x - u) \cup \{p\}) \cap ((y - v) \cup \{p\})$.

(iii) The postcondition $z = X \cap Y$ holds when the loop exits with the loop invariant true. [2 marks]

If the loop exits with the invariant true, we have:
$u \subseteq x \wedge v \subseteq y \wedge x-u \cup y-v < u \cup v \wedge z = (x-u) \cap (y-v) \wedge (u = \emptyset \vee v = \emptyset)$
If $u = \emptyset$, the remaining elements in v cannot be in x (because ...), so we have $z = x \cap y$.
Similarly, if $v = \emptyset$, the remaining elements in u cannot be in y (because ...), so we have $z = x \cap y$.

(iv) The loop terminates for any finite sets X and Y. [2 marks]

Take the bound function to be $t \triangleq |u| + |v|$.
Every iteration of the loop removes one element from u or from v, or both, so $|u| + |v|$ must decrease. The loop will terminate when $|u| + |v| = 0$, or before, since this can only hold when $u = \emptyset \wedge v = \emptyset$.

(c) The following is a version of the above algorithm, using arrays to represent the sets. Sets X and Y are represented by the array segments $a[1..M]$ and $b[1..N]$, respectively, where $M = |X|$ and $N = |Y|$; z is represented by an array segment $c[1..k]$, where k is the current size of z. All arrays are assumed to be at least as big as the set being represented. Elements of a and b are stored in strictly ascending order.

```
$i := 1$; $j := 1$; $k := 0$;
while $i \leq m \wedge j \leq n$ do
    if $a[i] < b[j]$ then
        $i := i + 1$
    elsif $a[i] > b[j]$ then
        $j := j + 1$
    else
        $k := k + 1$;
        $c[k] := a[i]$;
        $i := i + 1$;
        $j := j + 1$
    fi
od
```

(i) Give an *abstraction relation* explaining the relationship between the variables used in this version of the program and those in the earlier version. [4 marks]

(ii) Use your *abstraction relation* to explain how $a[i]$ and $b[j]$ in this version correspond to $min(x)$ and $min(y)$ in the earlier version, and why the operations $i := i + 1$ and $j := j + 1$ correctly implement the operations $x := x - \{min(x)\}$ and $y := y - \{min(y)\}$ in the earlier version. [4 marks]

****