**Victoria University of Wellington**

# DEGREE EXAMINATIONS — 1999



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.
Foreign language dictionaries are permitted.

## Question 1. [20 marks]

**(a)** Design a DFA that correctly recognizes the language $L$ over the alphabet $\Sigma = \{a, b\}$

$$L = \{x \in \Sigma^* : \exists\, x_1, x_2, x_3, x_4 \in \Sigma^* \text{ such that } x = x_1\, a\, x_2\, b\, x_3\, a\, x_4\}$$

(In other words, $L$ consists of all words in $\Sigma^*$ that contains $aba$ as a subsequence.)
[5 marks]

**(b)** Give a regular expression that describes $L$. [5 marks]

**(c)** Use the Myhill-Nerode Theorem to prove that the language of squares over $\Sigma = \{a, b\}$ is not regular. The language of squares is:

$$L = \{x \in \Sigma^* : \exists\, y \in \Sigma^* \text{ such that } x = y\, y\}$$

[5 marks]

**(d)** Define the relation $R$ on $\Sigma^*$ for $\Sigma = \{a, b\}$ by:

$$xRy \text{ if and only if } n_a(x) - n_b(x) = n_a(y) - n_b(y)$$

Justify your answer to the following questions:

**(i)** Is $R$ a right congruence?

**(ii)** Does $R$ have a finite number of equivalence classes?

[5 marks]

## Question 2. [12 marks]

The following Context Free Grammar defines the language of well-formed formulas (wffs) of propositional logic:

$$W \quad \rightarrow \quad atom \mid \neg\, W \mid W \wedge W \mid W \vee W \mid W \Rightarrow W \mid W \equiv W \mid (\, W \,)$$

where $W$ denotes a wff and *atom* is an atomic wff consisting of a propositional symbol. We will assume that $p$, $q$ and $r$ are propositional symbols.

**(a)** Explain what it means for a Context Free Grammar to be *ambiguous*, and show that the above grammar is ambiguous. [4 marks]

**(b)** Write an unambiguous Context Free Grammar for the language of wffs, assuming the following properties for the operators:

| Operator | Associativity | Precedence |
|---|---|---|
| $\neg$ | Associative | High (performed first) |
| $\wedge$ | Left-associative | |
| $\vee$ | Left-associative | |
| $\Rightarrow$, $\equiv$ | Non-associative | Low (performed last) |

[5 marks]

**(c)** Give a parse tree showing that the string "$(p \wedge q \Rightarrow r) \equiv (\neg p \vee q \vee r)$" is a wff, using your grammar from part **(b)**. [3 marks]

## Question 3. [30 marks]

Consider the following Context Free Grammar:

$$
\begin{array}{lcl}
S & \rightarrow & \mathit{Exps}\ \mathit{Exp} \\
\mathit{Exps} & \rightarrow & \mathit{Exp}\ \text{“;”}\ \mathit{Exps} \mid \lambda \\
\mathit{Exp} & \rightarrow & \mathit{number} \mid \mathit{id} \mid \mathit{id}\ \text{“(”}\ \mathit{EL}\ \text{“)”} \\
\mathit{EL} & \rightarrow & \mathit{Exp} \mid \mathit{EL}\ \text{“,”}\ \mathit{Exp}
\end{array}
$$

This grammar defines a language in which each sentence is a non-empty list of expressions, separated by semicolons. An expression is either a number, an identifier, or a function application of the form $id(e_1, \cdots, e_n)$, where $n > 0$ and $e_1, \cdots, e_n$ are expressions. We will assume that a number is a non-empty sequence of digits and an identifier is a non-empty sequence of letters.

**(a)** Explain what it means for a grammar to be in LL(1) form, and why this property is important in the construction of an LL(1) parser. [5 marks]

A grammar is LL(1) if:

(i) For any two (distinct) rules $N \rightarrow \alpha$ and $N \rightarrow \beta$, $first(\alpha) \cap first(\beta) = \emptyset$, where $first(\gamma)$ is the set of terminals that start strings produced from $\gamma$.

(ii) If $G$ contains a rule $N \rightarrow \alpha$, where $\alpha \Rightarrow^* \lambda$, then $first(\alpha) \cap follow(\alpha) = \emptyset$, where $follow(\alpha)$ is the set of terminals that can follow a string produced from $N$ in a sentence.

These conditions are important, because ensure that an LL(1) parser can always chose the appropriate rule to use at every step, based on a one-symbol lookahead (i.e. just by looking a the next input symbol).

**(b)** Identify any places where the above grammar fails to meet the LL(1) conditions. [3 marks]

The rules for *eq-list*, *exp* and *exp-list* all break the first LL(1) condition. In each case the first sets for the two rules are identical.
Specifically (though not expected):

$$first(eqn) = \{id\} = first(eq - list\ ;\ eqn)$$
$$first(id) = \{id\} = first(id\ (\ exp - list\ ))$$
$$first(exp) = \{id\} = first(exp - list\ ,\ exp)$$

**(c)** Write a (plain) CFG in LL(1) form, equivalent to the grammar above. Show that your grammar satisfies the LL(1) conditions. [7 marks]

| | | | |
|---|---|---|---|
| *eq-list* | → | *eqn rest-eq-list* | (1) |
| *rest-eq-list* | → | $\lambda$ \| ";" *eq-list* | (2,3) |
| *eqn* | → | *exp* "=" *exp* | (4) |
| *exp* | → | *id rest-exp* | (5) |
| *rest-exp* | → | $\lambda$ \| "(" *exp-list* ")" | (6,7) |
| *exp-list* | → | *exp rest-exp-list* | (8) |
| *rest-exp-list* | → | $\lambda$ \| "," *exp-list* | (9,10) |

**(d)** Construct an LL(1) parser table, based on your grammar in part **(c)**. [5 marks]

| | id | ; | = | ( | ) | , | $ |
|---|---|---|---|---|---|---|---|
| *eq-list* | 1 | - | - | - | - | - | - |
| *rest-eq-list* | - | 3 | - | - | - | - | 2 |
| *eqn* | 4 | - | - | - | - | - | - |
| *exp* | 5 | - | - | - | - | - | - |
| *rest-exp* | - | - | 6 | 7 | - | - | - |
| *exp-list* | 8 | - | - | - | - | - | - |
| *rest-exp-list* | - | - | - | - | 9 | 10 | - |

**(e)** Show the steps taken by an LL(1) parser, using your table from part **(d)**, in parsing the string "$f(1, a); b$".

At each step you should show the current contents of the stack, the remaining input, and the action taken by the parser, including which rule is used in each "expand" step. You may use suitable abbreviations for non-terminal symbols, provided the intended meaning is clear. [5 marks]

**(f)** Write an Extended Context Free Grammar (ECFG), equivalent to the grammar given above.

You should make best use of the ECFG notation to obtain a compact and intelligible grammar, reflecting the structure of the language as clearly as possible. [5 marks]

| | | |
|---|---|---|
| *eq-list* | → | *eqn* { ";" *eqn* } |
| *eqn* | → | *exp* "=" *exp* |
| *exp* | → | *id* [ "(" *exp-list* ")" ] |
| *exp-list* | → | *exp* { "," *exp* } |

## Question 4. [20 marks]

**(a)** Explain briefly the difference between *strong equivalence* and *weak equivalence* of programs. [4 marks]

Standard "book work".

**(b)** Show that the following while programs are *strongly* equivalent, where $B_1$ and $B_2$ are any Boolean expressions and $S_1$, $S_2$, $S_3$ are any statements:

$P_1$: **if** $B_1$ **then** $S_1$ **else** $S_2$ **fi**;
**if** $B_2$ **then** $S_3$ **else** $S_4$ **fi**

$P_2$: **if** $B_1$ **then**
$S_1$; **if** $B_2$ **then** $S_3$ **else** $S_4$ **fi**
**else**
$S_2$; **if** $B_2$ **then** $S_3$ **else** $S_4$ **fi**
**fi**

[4 marks]

Call the programs $P_1$ and $P_2$, and let $Paths(P)$ be the RE for all potential paths in $P$. Then:

$$
\begin{aligned}
&Paths(P_1)\\
&= (B_1\, S_1 | \overline{B_1}\, S_2)(B_2\, S_3 | \overline{B_2}\, S_4)\\
&= (B_1\, S_1 S\, (B_2\, S_3 | \overline{B_2}\, S_4) | \overline{B_1}\, S_2\, (B_2\, S_3 | \overline{B_2}\, S_4))\\
&= Paths(P_2)
\end{aligned}
$$

The two REs are equal, by distributivity.
(NB: This is blurring the distinction between $S_1$ and $Paths(S_1)$, etc.)

**(c)** We can translate any while program, $P$, into a while program, $P'$, containing no **if** statements, by applying the following transformation to each **if** statement:

**if** $B$ **then** $S_1$ **else** $S_2$ **fi** $\longrightarrow$
$v := 1$;
**while** $B \wedge v = 1$ **do** $S_1'$; $v := 0$ **od**;
**while** $v = 1$ **do** $S_2'$; $v := 0$ **od**

where $S_1'$ and $S_2'$ are the results of applying this transformation to $S_1$ and $S_2$, respectively, and a new variable $v$, not appearing anywhere else in the program, is used for each **if** statement transformed.

**(i)** Show that $P'$ is not strongly equivalent to $P$. [2 marks]

**(ii)** Show that $P'$ is weakly equivalent to $P$. [5 marks]

**(iii)** Show the result of applying this transformation to the following program:

```
i := 1; r := 0;
while i ≤ n ∧ r = 0 do
    if a[i] = b[i] then
        if b[i] = c[i] then
            r := 1
        else
            i := i + 1
        fi
    else
        i := i + 1
    fi
od
```

[5 marks]

```
i := 1; r := 0;
while i ≤ n ∧ r = 0 do
    v1 := 1;
    while a[i] = b[i] ∧ v1 = 1 do
        v2 := 1;
        while b[i] = c[i] ∧ v2 = 1 do
            r := 1; v2 := 0
        od;
        while v2 = 1 do
            i := i + 1; v2 := 0
        od;
        v1 := 0
    od;
    while v1 = 1 do
        i := i + 1
    od
od
```

## Question 5. [18 marks]

The following program sums the elements in an array segment $A[\,1..n\,]$, by adding the first and last elements, then the second and second to last elements, etc.

$\{\ even(n)\ \}$
$p := 1;\ q := n;\ s := 0;$
**while** $p < q$ **do**
    $s := s + A[\,p\,] + A[\,q\,];$
    $p := p + 1;\ q := q - 1$
**od**
$\{\ s = \sum_{i=1}^{n} A[\,i\,]\ \}$

We want to show that this program does correctly compute the sum of $A[\,1..n\,]$, provided that $n$ is even. To do this, we first need to find a loop invariant. Consider the following condition, which is a possible loop invariant:

$$I \triangleq even(n)\ \wedge\ 1 \le p \le n{+}1\ \wedge\ 0 \le q \le n\ \wedge\ s = \textstyle\sum_{i=1}^{p-1} A[\,i\,] + \sum_{j=q+1}^{n} A[\,i\,]$$

**(a)** Show that $I$ is invariant in the loop; i.e. show (i) that $I$ holds on entry to the loop, and (ii) that $I$ is preserved by the loop.

Give the conditions that must be satisfied, and give a brief argument to show why they hold, identifying any properties of summation of arrays required in the proof. [5 marks]

**(b)** Unfortunately, $I$ is not strong enough to allow us to show that the required post-condition, $s = \sum_{i=1}^{n} A[\,i\,]$, holds on exit from the loop.

Explain why this is, and give an example to show that $s = \sum_{i=1}^{n} A[\,i\,]$ may not hold in a state where $I \wedge \neg(p < q)$ holds. [3 marks]

**(c)** Give an additional condition which, when added to $I$ to obtain a new loop invariant, does allow us to show that $s = \sum_{i=1}^{n} A[\,i\,]$ holds on exit from the loop.

Show that the additional condition is invariant in the loop (i.e. that it holds on entry to the loop and is preserved by the loop).

Show that the postcondition holds on exit from the loop, using the new invariant.

Give the conditions that must be satisfied, and give a brief argument to show why they hold, identifying any properties of summation of arrays required in the proof. [7 marks]

**(d)** Explain, using an example, why the program will not always correctly compute the sum of $A[\,1..n\,]$ if $n$ is odd.

Show how the program can be made to work correctly for any $n$, by adding one statement. [3 marks]

**********