OLLSCOIL NA hÉIREANN
THE NATIONAL UNIVERSITY OF IRELAND, CORK

COLÁISTE NA hOLLSCOILE, CORCAIGH
UNIVERSITY COLLEGE, CORK

2014/2015

Semester 1– Winter 2015

CS4150: Principles of Compilation

Dr Helen Purchase
Professor Cormac Sreenan
Dr Kieran Herley

$1\frac{1}{2}$ Hours

Answer All Questions
Total marks 100%

PLEASE DO NOT TURN THIS PAGE UNTIL INSTRUCTED TO DO SO

PLEASE ENSURE THAT YOU HAVE THE CORRECT EXAM

Question 1 *[30%]*

(i) Give a regular expression for the set of binary strings where the number of 1 bits is divisible by 3 *e.g.* the language includes strings 00000000, 100010000001 and 1110101010, but not 1010 or 1100111. (*5%*)

(ii) Give a deterministic finite automaton for language of Part (i). (*5%*)

(iii) Give a non-deterministic finite automata capable of recognizing validly formatted email addresses (a simplified version, as specified below) and indicate how the automaton might be used to scan a text document and flag every email address that occurs within.

We will assume that each email address `local_part@domain` comprises a *local part*, an at symbol (@) and a *domain*. The local part consists of upper or lowercase Latin letters ($\{a, \cdots, z, A, \cdots, Z\}$), digits ($\{0, \cdots, 9\}$) and the underscore (denoted by $\ominus$) and dot symbols (denoted by $\odot$); a dot may not occur as the first or last symbol and no two dots may appear one after the other. The domain comprises letters, digits, and dots. You may use $\mathcal{L}$ and $\mathcal{D}$ as shorthand for "any letter" or "any digit" respectively in your expression and the notation $\bar{S}$ to denote the complement of any set S i.e. those elements not in S. (*10%*)

(iv) Give a regular expression to capture the valid email addressed as described above. (*10%*)

Question 2 *[20%]*

(i) Consider the following context-free grammar for a simplified set of regular expressions over the alphabet $\{a, b\}$. To avoid any potential confusion between regular expression symbols and context free grammar symbols, we put boxes around the regular expression operator symbols (e.g. $\boxed{*}$ for the star operator and so on), so, for example, the set of strings that begin with symbol 'a' and end with 'b' is captured by the expression

$$a \boxed{\cdot} \boxed{(} a \boxed{|} b \boxed{)} \boxed{*} \boxed{\cdot} b$$

A context-free grammar for regular expressions is as follows.

$\langle \mathrm{R} \rangle \rightarrow \langle \mathrm{R} \rangle \boxed{|} \langle \mathrm{R} \rangle$
$\langle \mathrm{R} \rangle \rightarrow \langle \mathrm{R} \rangle \boxed{\cdot} \langle \mathrm{R} \rangle$
$\langle \mathrm{R} \rangle \rightarrow \langle \mathrm{R} \rangle \boxed{*}$
$\langle \mathrm{R} \rangle \rightarrow \boxed{(} \langle \mathrm{R} \rangle \boxed{)}$
$\langle \mathrm{R} \rangle \rightarrow \mathbf{a}$
$\langle \mathrm{R} \rangle \rightarrow \mathbf{b}$

Show that the grammar is ambiguous. Explain why ambiguous grammars are generally regarded as undesirable in a parsing context. (*10%*)

(ii) Give a complete parse tree for the regular expression given as an example in Part (i). (*10%*)

Question 3 *[30 %]*

(i) The following is a modified grammar for the regular expression language of Question 2.

$\langle \text{R} \rangle \rightarrow \langle \text{E} \rangle \ \langle \text{A} \rangle$
$\langle \text{A} \rangle \ \rightarrow \boxed{*} \mid \epsilon$
$\langle \text{E} \rangle \ \rightarrow \langle \text{T} \rangle \ \langle \text{E'} \rangle$
$\langle \text{E'} \rangle \ \rightarrow \boxed{|} \ \langle \text{T} \rangle \ \langle \text{E'} \rangle \ \mid \epsilon$
$\langle \text{T} \rangle \ \rightarrow \langle \text{F} \rangle \ \langle \text{T'} \rangle$
$\langle \text{T'} \rangle \ \rightarrow \boxed{\cdot} \ \langle \text{F} \rangle \ \langle \text{T'} \rangle \ \mid \epsilon$
$\langle \text{F} \rangle \ \rightarrow \boxed{(} \ \langle \text{R} \rangle \ \boxed{)} \mid \mathbf{a} \mid \mathbf{b}$

Indicate which non-terminals in the grammar are nullable (3%), the first set for each non-terminal (6%) and the follow set for each nonterminal (6%). (*15%*)

(ii) Based on the above, indicate the value (if any) in the LL(1) parser table for the following cells. In the event that there is no value in the cell in question just write "empty".

- Row $< F >$, column (
- Row $< F >$, column $\boxed{*}$
- Row $< T' >$, column $\boxed{\cdot}$
- Row $< T' >$, column $\boxed{|}$

(*10%*)

(iii) Give a crisp summary, in words or pseudocode, of the stack-based, table-driven LL(1) parsing algorithm. (*5%*)

Question 4 *[20 %]*

(i) Give a brief outline in words of a recursive code-generating algorithm that takes a parse tree for a Tiny program as input and that generates the equivalent three-address code (TAC) as output. You need not provide code, just a few sentences that convey the main ideas embodied in the approach. The Tiny grammar and a summary of the TAC notation are provided below for reference. (*5%*)

(ii) Write a pseudo-code fragment that generates TAC from a subtree of a parse tree representing a Tiny repeat-until loop. State any assumptions you make. (*10%*)

(iii) Illustrate your approach by showing the code generated in respect of the following Tiny repeat loop. In your answer show both the Tiny source and the TAC in a manner that displays the correspondence between each element of the Tiny code the associated TAC code. (*5%*)

```
sum := 0;
n := 100;
i := 0;
repeat
    sum := sum + i;
    i := i + 1
until i = n
```

Appendix

A Grammar for Tiny

⟨program ⟩ → ⟨stmtseq⟩
⟨stmtseq ⟩ → ⟨stmtseq⟩ **;** ⟨statement⟩ | ⟨statement⟩
⟨statement⟩ → ⟨ifstmt⟩ | ⟨repeatstmt⟩ | ⟨assignstmt⟩
| ⟨readstmt⟩ | ⟨writestmt⟩
⟨ifstmt⟩ → **if** ⟨exp⟩ **then** ⟨stmtseq⟩ **end**
| **if** ⟨exp⟩ **then** ⟨stmtseq⟩ **else** ⟨stmtseq⟩ **end**
⟨repeatstmt ⟩ → **repeat** ⟨stmtseq⟩ **until** ⟨exp⟩
⟨assignstmt ⟩ → **id :=** ⟨exp⟩
⟨readstmt ⟩ → **read id**
⟨writestmt ⟩ → **write** ⟨exp⟩
⟨exp⟩ → ⟨simplexp⟩ ⟨comp⟩ ⟨simplexp⟩ | ⟨simplexp⟩
⟨comp ⟩ → **<** | **=**
⟨simplexp⟩ → ⟨simplexp⟩ ⟨addop⟩ ⟨term⟩ | ⟨term⟩
⟨addop ⟩ → **+** | **-**
⟨term⟩ → ⟨term⟩ ⟨mulop⟩ ⟨factor⟩ | ⟨factor⟩
⟨mulop ⟩ → ***** | **/**
⟨factor ⟩ → **(** ⟨exp⟩ **)** | **num** | **id**

Three-Address Code

x := y op z	assignment
x := op y	unary assignment
x := y	copy
goto L	unconditional jump
if x relop y goto L	conditional jump
param x	procedure call
call p n	procedure call
return y	procedure call
x := y[i]	indexed assignment
x[i] := y	indexed assignment