SEMESTER 1 EXAMINATIONS 2019/2020

MODULE:	CA4003 - Compiler Construction
PROGRAMME(S):	
CASE	BSc in Computer Applications (Sft.Eng.)
CPSSD	BSc in ComputationalProblem Solv&SW Dev.
ECSAO	Study Abroad (Engineering & Computing)
YEAR OF STUDY:	4,O

EXAMINERS:

Dr. David Sinclair	(Internal)	(Ph:5510)
Dr. Hitesh Tewari	(External)	External
Prof. Brendan Tangney	(External)	External

TIME ALLOWED: 3 hours

INSTRUCTIONS: Answer 10 questions. All questions carry equal marks.

PLEASE DO NOT TURN OVER THIS PAGE UNTIL INSTRUCTED TO DO SO

The use of programmable or text storing calculators is expressly forbidden.
Please note that where a candidate answers more than the required number of questions, the examiner will mark all questions attempted and then select the highest scoring ones.

There are no additional requirements for this paper.

Note: In the following questions, non-terminal symbols are represented by strings starting with an upper case letter, e.g. A, Aa, Name, and terminal symbols are represented by either individual symbols (e.g. $+$) or sequence of symbols (e.g. $>=$), or by strings starting with a lower case letter, e.g. a, xyz. The ϵ symbol represents an empty symbol or null string as appropriate. The $ symbol represents the end-of-file.

QUESTION 1 ***[Total marks: 10]***

Regular Expressions & DFAs

1(a) [4 Marks]

Given a binary alphabet {0,1}, write a regular expression that recognises all words that have an odd number of '1's.

1(b) [6 Marks]

Use the subset construction method to derive a deterministic finite state automaton that recognises the language from part (a).

[End Question 1]

QUESTION 2 ***[Total marks: 10]***

Equivalent LL(1) Grammar

[10 Marks]

Consider the following grammar for a language where *print* is a keyword and *I* is a valid identifier for both variables and functions.

Convert the grammar into an LL(1) grammar which recognises the same language:

$$
\begin{array}{lcl}
S & \rightarrow & print(L) \\
L & \rightarrow & L, E \\
L & \rightarrow & E \\
E & \rightarrow & I \\
E & \rightarrow & I(L)
\end{array}
$$

[End Question 2]

QUESTION 3 *[Total marks: 10]*

FIRST & FOLLOW

[10 Marks]

Calculate the FIRST and FOLLOW sets for the following grammar.

$$\begin{aligned}
&S \to u\ A\ B\ v\\
&A \to A\ w\\
&A \to x\\
&B \to C\ D\\
&C \to y\\
&C \to \epsilon\\
&D \to z\\
&D \to \epsilon
\end{aligned}$$

[End Question 3]

QUESTION 4 *[Total marks: 10]*

LL(1)

[10 Marks]

Construct the LOOKAHEAD table and the LL(1) parse table for the following grammar and using the LL(1) parse table determine whether or not the grammar is a LL(1) grammar.

$$\begin{aligned}
&S \to I|o\\
&I \to i\ X\ t\ S\ E\\
&E \to e\ S\\
&E \to \epsilon\\
&X \to a|b
\end{aligned}$$

[End Question 4]

QUESTION 5 *[Total marks: 10]*

LR(1)

[10 Marks]

Construct the LR(1) parse table for the following grammar and using it determine whether or not the following grammar is LR(1):

$$
\begin{aligned}
&S' \rightarrow S \\
&S \rightarrow aBc \\
&S \rightarrow bCc \\
&S \rightarrow aCd \\
&S \rightarrow bBd \\
&B \rightarrow e \\
&C \rightarrow e
\end{aligned}
$$

[End Question 5]

QUESTION 6 ***[Total marks: 10]***

LALR(1)

[10 Marks]

Construct the LALR(1) parse table for the grammar in question 5 and using it determine whether or not the grammar is LALR(1).

[End Question 6]

QUESTION 7 ***[Total marks: 10]***

Intermediate Code

7(a) [7 Marks]

Convert the following source code into intermediate code using the syntax-directed approach given in the appendix. Assume that all variables are stored in 4 bytes.

```
sum = 0;
i = 0;
while (i < 10)
{
  sum = sum + a[i];
  i = i+ 1;
}
```

7(b) [3 Marks]

Generate a *Control Flow Graph* from the intermediate code generated in part (a). Clearly describe the rules used to generate the *Control Flow Graph*.

[End Question 7]

QUESTION 8 **[Total marks: 10]**

Liveness

8(a) [4 Marks]

Describe how *Data Flow Analysis* is used to calculate the liveness of variables.

8(b) [6 Marks]

For the following intermediate code, assuming variable d, k and j are live on exit from this code, calculate which variables are live on entry.

$$
\begin{aligned}
&t_1 = j + 12 \\
&g = a[t_1] \\
&h = k - 1 \\
&f = g * h \\
&t_2 = j + 8 \\
&e = a[t_2] \\
&t_3 = j + 16 \\
&m = a[t_3] \\
&b = a[f] \\
&c = e + 8 \\
&d = c \\
&k = m + 4 \\
&j = b
\end{aligned}
$$

[End Question 8]

QUESTION 9 **[Total marks: 10]**

Register Allocation & Graph Colouring

Consider the following basic block.

```
x = 5
a = x + 5
b = x + 3
v = a + b
a = x + 5
z = v + a
```

9(a) [5 Marks]

Calculate the *liveness* at each point in the basic block assuming that only z is live on exit.

9(b) [5 Marks]

From the *liveness* information in part (a), generate the interference graph and using the *graph colouring* algorithm allocate the variables to 3 registers.

[End Question 9]

QUESTION 10 ***[Total marks: 10]***

Visitor Pattern

[10 Marks]

With the aid of example code, describe the *Visitor* pattern. Why is it particularly suited to *"walking an abstract syntax tree"*?

[End Question 10]

APPENDICES

Syntax-directed definition approach to build the 3-address code

Production	Semantic Rule
$S \rightarrow \mathbf{id} = E;$	$gen(get(\mathbf{id}.lexeme)$ '=' $E.addr)$;
$S \rightarrow L = E;$	$gen(L.addr.base$ '[' $L.addr$ ']' '=' $E.addr)$;
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}Temp()$; $gen(E.addr$ '=' $E_1.addr$ '+' $E_2.addr)$;
$E \rightarrow \mathbf{id}$	$E.addr = get(\mathbf{id}.lexeme)$;
$E \rightarrow L$	$E.addr = \mathbf{new}Temp()$; $gen(E.addr$ '=' $L.array.base$ '[' $L.addr$ ']'$)$;
$L \rightarrow \mathbf{id}[E]$	$L.array = get(\mathbf{id}.lexeme)$; $L.type = L.array.type.elem$; $L.addr = \mathbf{new}Temp()$; $gen(L.addr$ '=' $E.addr$ '*' $L.type.width)$;
$L \rightarrow L_1[E]$	$L.array = L_1.array$; $L.type = L_1.type.elem$ $t = \mathbf{new}Temp()$; $L.addr = \mathbf{new}Temp()$; $gen(t$ '=' $E.addr$ '*' $L.type.width)$; $gen(L.addr$ '=' $L_1.addr$ '+' $t)$;
$B \rightarrow B_1 \vert\vert B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B_1.code \Vert label(B_1.false) \Vert B_2.code$
$B \rightarrow B_1 \&\& B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B_1.code \Vert label(B_1.true) \Vert B_2.code$
$B \rightarrow !B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \Vert E_2.code$ $\Vert gen($'if' $E_1.addr$ **rel** $E_2.addr$ 'goto' $B.true)$ $\Vert gen($'goto' $B.false)$
$B \rightarrow$ **true**	$B.code = gen($'goto' $B.true)$
$B \rightarrow$ **false**	$B.code = gen($'goto' $B.false)$

Production	Semantic Rule
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \| label(S.next)$
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \rightarrow \textbf{if}\ (\ B\ )\ S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \| label(B.true) \| S_1.code$
$S \rightarrow \textbf{if}\ (\ B\ )\ S_1\ \textbf{else}\ S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code \| label(B.true) \| S_1.code$ $gen(\text{'goto'}\ S.next) \| label(B.false) \| S_2.code$
$S \rightarrow \textbf{while}\ (\ B\ )\ S_1$	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \| B.code$ $\| label(B.true) \| S_1.code \| gen(\text{'goto'}\ begin)$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel$ $S_2.next = S.next$ $S_1.code \| label(S_1.next) \| S_2.code$

[END OF APPENDICES]

[END OF EXAM]