The Open University of Sri Lanka
Faculty of Engineering Technology
Department of Electrical and Computer Engineering

Study Programme	: Bachelor of Software Engineering in Honours
Name of the Examination	: Final Examination
Course Code and Title	: **EEX6363 Compiler Construction** **EEX6563 Software Construction**
Academic Year	: 2020/21
Date	: 18th January 2022
Time	: 09:30 – 12:30 hrs.
Duration	: **3 hours**

General Instructions

1. Read all instructions carefully before answering the questions.
2. This question paper consists of **four (4)** questions in **four (4)** pages.
3. Answer **all** questions in **Section A** and any **TWO** questions from **Section B.**
4. Answer for each question should commence from a new page.
5. Answers should be in clear handwriting and do not use Red colour pen.
6. Clearly state your assumptions, if any.
7. This is a **Closed Book Test** (CBT).

Section A – Answer all questions [60 marks]

Q1 The following section presents the description of a language, called **SumCalc** for summing numerical values.

SumCalc is a very simple language that accommodates two forms of numerical data types (float and integer), allows computation and printing of numerical values, and offers a small set of variable names to hold the results of computations. As most programming languages, the conversion from integer type to float type is accomplished automatically in **SumCalc**. The production rules for the grammar of this language are given below.

1	PROG	→	DCLS STMTS *$*
2	DCLS	→	DCL DCLS
3		|	ε
4	DCL	→	*floatdcl id*
5		|	*intdcl id*
6	STMTS	→	STMT STMTS
7		|	ε
8	STMT	→	*id assign* VAL EXPR
9		|	*print id*
10	EXPR	→	*plus* VAL EXPR
11		|	*minus* VAL EXPR
12		|	ε
13	VAL	→	*id*
14		|	*intnum*
15		|	*floatnum*

where ε denotes the empty string, CAPITAL terms are non-terminals (PROG is the start symbol) while all others are terminals including the special symbol *$* represents the end of the input stream.

The specification of tokens in **SumCalc** is accomplished by associating a regular expression with each token, as shown below.

Terminal	Regular Expression
floatdcl	f
intdcl	i
print	p
id	$[a-e] \mid [g-h] \mid [j-o] \mid [q-z]$
assign	=
plus	+
minus	-
intnum	$[0-9]^+$
floatnum	$[0-9]^+ . [0-9]^+$
blank	$(\text{“ ”})^+$

(a) Write the input stream for this language that satisfies the following requirement. Your input stream should be syntactically valid. [10]

"Declare two numbers, one is integer type and other is float type, then assign values and getting the summation of those two values, finally print the result."

(b) Validate the input stream written in (a) using the grammar of **SumCalc**. [10]

(c) Construct the non-deterministic finite automata for the input obtained in (a). [04]

(d) Draw the parse tree for the input stream written in (a). [08]

(e) Draw the abstract syntax tree (AST) for the parse tree obtained in (d). You can assume whatever variables and numerical values from the specification of tokens. [08]

(f) Re-draw the AST drawn in (e) after applying semantic analysis. [04]

(g) Write LEX implementation syntax for the language **SumCalc**. [10]

(h) Explain how the grammar of **SumCalc** enables you to answer the following questions.

i) Can **SumCalc** program contain only declarations (and without statements)? [03]
ii) Can a print statement precede all assignment statements? [03]

<u>**Section B** – Answer any **TWO** questions</u> [40 marks]

Q2 Five production rules (R1 to R5) for a specific grammar, **G₂** are given by

R1: FCTR → *id* NEST NEST
R2: NEST → . *id* LIST NEST
R3: NEST → ε
R4: LIST → *[expr]* LIST
R5: LIST → ε

where CAPITAL terms are non-terminals (three) while all others are terminals (six) and the starting term is FCTR.

(a) Construct the FIRST and FOLLOW sets for the grammar G_2. [03]

(b) Construct the LL(1) top-down predictive parsing table for this grammar. [06]

(c) Convert the given grammar G_2 into Chomsky Normal Form (CNF). [08]

(d) Use the CNF obtained in (c) to verify whether the input string "*id.id[expr].id*" is belong to the grammar G_2 or not. [03]

Q3 Consider the grammar, $\mathbf{G_3}$ with three production rules as given below:

R1: S → aSa
R2: S → a
R3: S → b

where S is a non-terminal while a and b are terminals.

(a) Construct the Simple SLR item sets that correspond to this grammar. [04]

(b) Draw a deterministic finite automaton that corresponds to the item sets found in your answer (a). [04]

(c) Construct the LR(1) bottom-up parsing table for this grammar. [04]

(d) Convert the grammar $\mathbf{G_3}$ into Greibach Normal Form (GNF). [05]

(e) Use the GNF obtained in (d) to verify whether the input string "*aabaa*" is belong to the grammar $\mathbf{G_3}$ or not. [03]

Q4

(a) Consider the following statement:

total = *sum* + 1000 * (*k* – *func*(*k*)); /* recomputed total*/

Explain how the above statement is analyzed and transformed (the input and output) by the first four phases of a typical compiler. [08]

(b) Consider the following expression to answer parts i) and ii):

$$a \times (b + c - d)/a - e[4] + 2$$

i) Draw syntax tree and convert it into directed acyclic graph (DAG). [06]

ii) Write optimized three-address codes using the DAG obtained in i). [06]

-- End --