The University of New South Wales

Final Examination

Term 1, 2022

COMP3131/COMP9102

Programming Languages and Compilers

1. **Time Allowed:** 3 hours (including 10 min **reading time**)
2. **Total Number of Pages**: 7 (including cover page and Appendix A)
3. **Total Number of Questions:** 5
4. **Total Marks Available:** 100
5. Marks available for each question are shown in the examination paper.
6. The questions are not of equal value.
7. Answer all questions.
8. Submit your answers via **give** or **Webcms3**.
9. The answers to Q1 can be submitted as jpeg/gif/tiff/png/pdf files but the answers to Q2 – Q5 must be submitted as ASCII text files:
   - Q1 and Q2: *.*suffix*, where *suffix* is jpeg, gif, tiff, png, or pdf
   - Q3: *.txt
   - Q4: *.txt
   - Q5: *.txt
10. No examination materials allowed.

Question 1. Regular Expressions to Finite Automata *[15 marks]*

Consider the following regular expression:

$$1(0|1)^*$$

(a) Use **Thompson's construction** to convert this regular expression into an NFA.

[6 marks]

(b) Use the **subset construction** to convert the NFA of (a) into a DFA.

[6 marks]

(c) Convert the DFA of (b) into a minimal-state DFA.

[3 marks]

You are required to apply exactly Thompson's construction algorithm in (a) and the subset construction algorithm in (b) to solve those two problems.

Question 2. Context-Free Grammars *[20 marks]*

Consider the following context-free grammar for describing arithmetic expressions:

$$
\begin{array}{lcl}
\textit{expr} & \rightarrow & \textit{term} + \textit{expr} \mid \textit{term} \\
\textit{term} & \rightarrow & \textit{factor} * \textit{term} \mid \textit{term} / \textit{factor} \mid \textit{factor} \\
\textit{factor} & \rightarrow & \textbf{INTLITERAL}
\end{array}
$$

where the the non-terminals are given in *italics* and the terminals in **boldface** (including '+', '*', '/', and **INTLITERAL** (representing integer constants)).

(a) Write a leftmost derivation for the sentence $5 + 4 * 3 \ / \ 2$.

[3 marks]

(b) Draw a parse tree for this sentence.

[3 marks]

(c) If the operators +, * and / represent the operations of integer addition, integer multiplication and integer (truncated) division, respectively, what would be the value implied by your parse tree found in (b)?

[4 marks]

(d) Is this grammar ambiguous? Justify your answer.

[4 marks]

(e) Answer the following true or false questions about this grammar:

1. + is always right-associative.
2. * is always left-associative.
3. / is always right-associative.
4. + must always have higher precedence than * and /.
5. * may have higher precedence than /.
6. / may have higher precedence than *.

[6 marks with 1 mark for each true/false question]

Question 3. Recursive-Descent LL(1) Parsing *[25 marks]*

Consider the following context-free grammar:

$$
\begin{array}{lll}
1. & S & \rightarrow SS* \\
2. & S & \rightarrow SS+ \\
3. & S & \rightarrow \textsf{INT} \\
4. & S & \rightarrow \textsf{ID}
\end{array}
$$

where S is the non-terminal, and '*****', '**+**', **INT** (representing integer constants) and **ID** (representing identifiers) are the four terminals.

(a) Eliminate the left recursion in the grammar.

[5 marks]

(b) Do left-factorisation of the grammar produced in (a).

[5 marks]

(c) Compute the FIRST and FOLLOW sets for every non-terminal in the grammar produced in (b).

[6 marks]

(d) Construct an LL(1) parsing table for the grammar produced in (b), based on the FIRST and FOLLOW sets computed in (c).

[5 marks]

(e) The sentence $4\ x + *$ is NOT syntactically legal (since it is NOT in the language defined by the grammar). Explain concisely how $4\ x + *$ can be detected by an LL(1) table-driven parser for the language, by showing the moves of the parser on this sentence based on the LL(1) parsing table produced in (d), as shown in Week 9's Wednesday Lecture:

```
Stack                Input                Production
$                    INT ID + *
```

[4 marks]

Question 4. Attribute Grammars *[25 marks]*

Consider the following context-free grammar:

program	$\rightarrow$	**func** *stmt-list*
stmt-list	$\rightarrow$	*stmt stmt-list*
stmt-list	$\rightarrow$	*stmt*
stmt	$\rightarrow$	**for ID = INTLITERAL upto INTLITERAL begin** *stmt-list* **end**
stmt	$\rightarrow$	*assignment*

where the non-terminals are given in *italics* and the terminals in **boldface**.

This grammar describes a simple language, which allows iterating through a sequence of statements, ultimately assignments. A **for**-loop is specified by the range of its loop variable (identified by **ID**), given by integer constants (identified by **INTLITERAL**), representing its lower and upper bounds. In other words, a **for**-loop, with its lower and upper bounds being L and U, respectively, will execute its loop body, i.e., *stmt-list* exactly $U - L + 1$ times. No production for assignments is given (as it is irrelevant here). Therefore, assignments, represented by *assignment*, are treated here as terminals.

A sample program is given below (with its loop executed exactly 20 times):

```
1 func
2 assignment
3 for i = 1 upto 20
4 begin
5   assignment
6   assignment
7   assignment
8 end
9 assignment
```

(a) Write an attribute grammar that determines for each assignment how many times the assignment will be executed when running the program. You can assume that for each loop, its lower bound is no larger than its upper bound (so that you do not have to check this in your solution).

[20 marks]

(b) Describe whether each attribute used is synthesised or inherited.

[5 marks]

Question 5. Code Generation *[15 marks]*

Suppose we introduce a **do-while** (loop) statement into our VC language:

do stmt **while** "(" expr ")"

where *expr* and *stmt* are defined exactly as in our VC grammar. Note that how these two nonterminals are defined is irrelevant to this question.

The **do-while** statement has exactly the same semantics as that in C/C++/Java. It executes *stmt* repeatedly until the value of `expr` is `false`.

Suppose we use the following AST class to represent a **do-while** statement in the AST representation of a VC program:

```
package VC.ASTs;

import VC.Scanner.SourcePosition;

public class DoWhiletmt extends Stmt {

  public Expr E;
  public Stmt S;

  public DoWhileStmt (Expr eAST, Stmt sAST, SourcePosition Position)
  {
    super (Position);
    E = eAST;
    S = sAST;
    E.parent = S.parent = this;
  }


  public Object visit(Visitor v, Object o)
  {
    return v.visitDoWhileStmt(this, o);
  }
}
```

Write Emitter.visitDoWhileStmt in Java for generating Jasmin code for the **do-while** statement by using the visitor design pattern as you did in Assignment 5 (for translating the **for** and **while** loops in VC).

You do not need to include code for computing the operand stack size required. However, you are required to include code to generate appropriate labels that can be used for translating any **break** or **continue** statement that may be contained in a **do-while** loop, similarly as you did in Assignment 5.

Appendix A. Jasmin assembly (i.e., JVM) instructions for Question 5.

```
public final class JVM {
    // Arithmetic instructions
    IADD = "iadd",
    ISUB = "isub",
    IMUL = "imul",
    IDIV = "idiv",

    // Loads (for loading a local variable into operand stack)
    ILOAD = "iload",
    ILOAD_0 = "iload_0",
    ILOAD_1 = "iload_1",
    ILOAD_2 = "iload_2",
    ILOAD_3 = "iload_3",

    // Stores (for storing the top operand in operand stack into a local variable)
    ISTORE = "istore",
    ISTORE_0 = "istore_0",
    ISTORE_1 = "istore_1",
    ISTORE_2 = "istore_2",
    ISTORE_3 = "istore_3",

    // Loads (for loading a constant into operand stack)
    ICONST_M1 = "iconst_m1",
    ICONST_0 = "iconst_0",
    ICONST_1 = "iconst_1",
    ICONST_2 = "iconst_2",
    ICONST_3 = "iconst_3",
    ICONST_4 = "iconst_4",
    ICONST_5 = "iconst_5",

    // Control transfer instructions
    GOTO = "goto",
    IFEQ = "ifeq",
    IFNE = "ifne",
    IFLE = "ifle",
    IFLT = "iflt",
    IFGE = "ifge",
    IFGT = "ifgt",
    IF_ICMPEQ = "if_icmpeq",
    IF_ICMPNE = "if_icmpne",
    IF_ICMPLE = "if_icmple",
    IF_ICMPLT = "if_icmplt",
    IF_ICMPGE = "if_icmpge",
    IF_ICMPGT = "if_icmpgt",

    // Operand stack management instructions
    DUP_X2 = "dup_x2",
    DUP = "dup",
    POP = "pop",
    ...
}
```

========== END OF PAPER ==========