CANDIDATE: please attach Student Support Unit sticker, if relevant

THE UNIVERSITY OF SUSSEX

BSc and MComp SECOND YEAR EXAMINATION

Compilers and Computer Architecture
G5035

You can start this exam at a time of your choosing within a 24 hour window. Once started you will have a set exam duration in which to complete it (note: the assessment will close at end of the 24 hour window; start with sufficient time to complete).

If you have extra time due to Reasonable Adjustments this is additional to the exam duration below and has been added to your assessment on Canvas.

Date: Thursday, 6 January 2022
24 Hour Window starts at: 09:30
Exam Duration: 3 hours (including time for scanning, collating, uploading)

Candidates should answer TWO questions out of THREE.
If all three questions are attempted only the first two answers will be marked.
Each question is worth 50 marks.

Write or type your answers on A4 paper, scan and save as a single PDF file and upload to Canvas.

Please make sure that your submission includes the following:

Your candidate number (Do not put your name on your paper)
The title of the module and the module code.

Read Academic Integrity Statement
You MAY access online materials, notes etc. during this examination. You must complete this assessment on your own and in your own words. DO NOT discuss this assessment with others before the end of its 24 hour window. By submitting this assessment you confirm that your assessment includes no instances of academic misconduct, for example plagiarism or collusion. Any instance of academic misconduct will be thoroughly investigated in accordance with our academic misconduct regulations.

1. This question is about *lexer and parsers*, *regular expressions and languages*, and *context-free grammars (CFGs)*.

   (a) For each of the following statements about languages (i.e. sets of words), state whether they are true or false. Correct answers are awarded 2 marks; wrong answers are penalised by 1 mark (to a minimum of 0 marks for this question). Omitted answers do not contribute to the marks. [20 marks]

   i. If $A^* = A$, then $A = \{\epsilon\}$.
   ii. For all A and B, $(AB)^* = A^*B^*$.
   iii. For all A, B, C, $(A \cup B)C = (AC) \cup (BC)$.
   iv. For all A, B, C, $(AB) \cup C = (A \cup C)(B \cup C)$.
   v. $L = \{a^n b^n : 0 \leq n \leq 4\}$ is regular.
   vi. If α and β are regular expressions, then $(\alpha\beta)^*\alpha = \alpha(\beta\alpha)^*$.
   vii. $L = \{w = xyzy : x, y, z \in \{0, 1\}^+\}$ is regular.
   viii. If A and B are not regular, then $A \cup B$ is not regular.
   ix. If A and C are regular and $A \subseteq B \subseteq C$, then B must be regular.
   x. If A^* is regular, then A is regular.

   (b) Explain why we usually use CFGs rather than regular expressions or finite state automata to specify the *syntactic structures* of programming languages (as opposed to the lexical structures of programming languages). [5 marks]

   (c) The lexer and parser of a compiler can be written "by hand". What is a popular alternative to hand-writing lexer and parser? List the advantages and shortcomings of either approach. [10 marks]

   (d) Consider the following context-free grammar:

```
value
    : 'IF' expr 'THEN' expr 'ELSE' expr
    | 'IF' expr 'THEN' expr
    | expr
;

expr
    : expr '+' term
    | term
;

term
    : 'a' | 'b' | 'c' | 'd' | 'e'
;
```

   Ignoring whitespaces, draw a parse tree for

```
IF a THEN b + c + d ELSE e
```

   [5 marks]

   (e) The grammar has several undesirable features—what are they? Illustrate them by examples as necessary, and explain why they are problematic. [5 marks]

   (f) What could be done to overcome the problems in the previous question? [5 marks]

2. This question is about *code generation*.

   (a) What data structure is used as input to *code generators*? What feature of it makes the implementation of code generators simple and efficient? [5 marks]

   (b) Some compilers have a semantic analysis phase after lexing and parsing but before code generation. What is the purpose of this phase, and why is semantic analysis not done in the parsing stage? What about languages that don't have semantic analysis at all (e.g. Python or JavaScript)? [5 marks]

   (c) Explain the difference between *type checking* and *type inference*. [10 marks]

   (d) Many compilers don't generate machine code directly after lexing, parsing and semantic analysis. Instead they first generate *machine-independent code*, which is later, possibly after optimisation, used to generate code for the target architecture. Explain the characteristics of machine-independent code, and the advantages of using them as an intermediate layer in compilation. [10 marks]

   (e) Describe a scheme for the translation of *function definitions* and *function calls* into low-level code (e.g., RISC-V or stack machine code) based on a stack, as discussed in the lectures and the third task of the coursework. Assume that in the source language function calls are of the form

   ```
   f(e1, ..., en)
   ```

   where `e1, ..., en` are expressions, function definitions are of the form

   ```
   f(x1, ..., xn) = BLOCK
   ```

   where `BLOCK` itself is an expression, and any expression evaluates to a 32-bit (i.e. 4-byte) integer. We also assume that code generation for other kinds of expressions is readily available, e.g., one can use `genExp(BLOCK)` and `genExp(e1)` directly. If you use uncommon instructions, please explain their semantics briefly. If you use activation records in your explanation, describe the layout of your activation records. Note that your answer comprises two parts: code generation for function definitions, and code generation for function calls. [20 marks]

3. This question is about *computer architecture* and *low-level programming*.

   (a) Compare and contrast CISC with RISC architectures. [10 marks]

   (b) What is the meaning of *data locality* in the execution of a program? Give an example of a program fragment that exhibits a high degree of data locality. Explain why your example exhibits data locality. [10 marks]

   (c) Real CPUs have a small number of fast registers, typically between 8 and 256. Explain why this small number of registers makes compilation more complicated in comparison with compilation to stack machines. Outline a compilation strategy for machines with a finite number of registers. You don't have to present a full code generator, just explain the key features and ideas. [10 marks]

   (d) In this question, you will write RISC-V assembly programs for two different tasks. Your programs may only use the registers `x0`, `a0`, `t0`, `t1`, and `sp`, and the following RISC-V assembly commands:

`lw`	*register*$_1$	*offset* (*register*$_2$)		load a word of data at *offset* from an address stored in *register*$_2$, into *register*$_1$.
`sw`	*register*$_1$	*offset* (*register*$_2$)		store the value of *register*$_1$ into memory at *offset* from an address stored in *register*$_2$.
`add`	*register*$_1$	*register*$_2$	*register*$_3$	add the values stored in *register*$_3$ and *register*$_2$, and store the result in *register*$_1$.
`addi`	*register*$_1$	*register*$_2$	*constant*	add the signed 16-bit value *constant* to the value stored in *register*$_2$ and and store the result in *register*$_1$.
`mul`	*register*$_1$	*register*$_2$	*register*$_3$	multiply the values stored in *register*$_3$ and *register*$_2$, and store the (lower 32-bit of the) result in *register*$_1$.
`b`	*address*			jump to the instruction at *address* .
`beq`	*register*$_1$	*register*$_2$	*address*	if the contents of *register*$_1$ and *register*$_2$ are equal, jump to the instruction at *address* .
`bne`	*register*$_1$	*register*$_2$	*address*	if the contents of *register*$_1$ and *register*$_2$ are different, jump to the instruction at *address* .

   Each instruction is to be written on a separate line. Use the convention that the stack grows 'downwards' (i.e. the top of the stack is at a lower address than the rest of the stack) and that `sp` points at an unused word in memory just over the top of the stack. We also ignore issues such as overflow or alignment.

   - Write a program that pops two integers from the stack, adds them, and pushes the result back onto the stack. [10 marks]
   - Write a program that pops an integer n from the stack and computes the *factorial* of n, i.e. $n! = n*(n-1)*...*2*1$ (assume that $n > 0$), and pushes it back onto the stack. [10 marks]

End of Paper