**Module Title: Introduction to Theoretical Computer Science**
**Exam Diet (Dec/April/Aug): April 2021/2**
**Brief notes on answers:**
**Section A**

All section A questions are either bookwork or simple applications similar to questions done in coursework or tutorials. The marking scheme is generic: drop 1 for minor slips, give about half for an answer with significant gaps, give a token 1 for an answer that is wrong but suggests at least some understanding.

1. (a) (i). Regular Languages are closed under $\oplus$ as $L_1 \oplus L_2$ could be rephrased as $(L_1 \cup L_2) \cap \overline{(L_1 \cap L_2)}$, and as regular languages are closed under intersection, union, and complement, they are closed under $\oplus$.

   (ii). Context-free languages are not closed under $\oplus$ as, if they were, then the complement of a context-free language $L$ could be given by $L \oplus \Sigma^*$ and thus would be context-free, but we know that the context-free languages are not closed under complement.

   (b) (i). It describes the language of matching parentheses.

   (ii). $S \to \texttt{(}S\texttt{)} \mid SS \mid \varepsilon$

   (c) Consider the infinite sequence of strings $u_i = \texttt{(}^i$. For any $i$ and $j$ where $i \neq j$ we have a string $w_{ij} = \texttt{)}^i$. Then $u_i w_{ij} \in \mathcal{L}(P)$ but $u_j w_{ij} \notin \mathcal{L}(P)$, thus there are an infinite number of equivalence classes and thus the language is not regular by the Myhill-Nerode theorem.

2. [easy problem-solving]

   (a) The function $\langle h :: t \rangle$ is $\langle h, t \rangle_2$, so roughly the code of a sequence of length $n$ is ${}^{n}3$ (the $n$-fold exponentiation of 3 to the 3, alias tetration). The value of $h$ can be roughly ignored - if it is small, it adds about 1 to the exponent of 3, which is insignificant.

   According to the coding in lectures, a machine state is a pair of program and register contents. The program is a sequence of length 100, and the register contents a sequence of length 5; these will add. So a very rough estimate is ${}^{105}3$.

   Give 3 marks for an answer like this, 4 marks for anything more careful. 2 marks for anything that at least gets the tower of exponents.

   (b) This is vastly better, because a sequence of length $n$ with values of size $m$ codes to about $m^n$. So a similar rough estimate would be around $3^{105}$ (or $10^{105}$ to be more pessimistic), which is huge but manageable.

   (c) Need to show we can translate programs both ways. From classical, $\text{DECJZ}(i, j)$ is equivalent to $\text{JZ}(i, j'); \text{DEC}(i)$. where $j'$ is the location in the translated program corresponding to $j$ in the original. From alternative, $j_0 : \text{DEC}(i)$ is equivalent to $j'_0 : \text{DECJZ}(i, j'_0 + 1)$, and $\text{JZ}(i, j)$ is equivalent to $\text{DECJZ}(i, j'); \text{INC}(i)$. One mark for each basic translation, plus one for remembering that instruction numbers need relocated.

3. (a) The rightmost $\lambda$-abstraction can be eliminated with the $\eta$ rule, giving:

$$(\lambda n.\ \lambda f.\ \lambda x.\ f\ (n\ f\ x))\ (\lambda f.\ f)$$

(b) The number of $\beta$-reductions to normalise $t_i$ is $\mathcal{O}(i)$, i.e. polynomial (linear) time.

(c) The size of the normal form of $t_i$ as a string is exponential in $i$.

(d) A Turing-machine must move the tape head to a tape cell in order to write to it. Therefore, there is a relationship between space and time: Any algorithm that uses exponential space must use exponential time. This simulation is not reasonable with respect to the cost models we used earlier, because the space required to write the normal form of $t_i$ grows exponentially with $i$ but the model used for time (number of $\beta$-reductions) grows only linearly. It is impossible for a Turing machine to simulate $\lambda$-calculus with those complexity characteristics.

## Section B

1. (a) We know that the word is of length $\leq n$, therefore we may simply nondeterministically guess a word $w$ by nondeterministically guessing each symbol up to the length $n$ [2] and (in linear time) check if it is accepted by a DFA [1]. Thus this problem is in NP.

   (b) Consider a clause $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$. We will construct a DFA that recognises satisfying assignments for this clause. [1] Each of the literals will be either a variable $V_k$ or its negation $\neg V_k$. If there is a positive variable $V_k$ in the clause, our DFA will accept if it encounters a 1 at position $k$. If there is a negation, i.e. $\neg V_k$, our DFA will accept if it encounters a 0 at position $k$. [1] Such a DFA can be constructed using a number of states linear in the number of variables. We have now constructed a DFA $A_i$ for each clause $C_i$. Each of these DFAs accepts a finite language corresponding to the satisfying assignments of the clause. [1] Thus, the intersection of these DFAs accepts exactly the satisfying assignments for the whole formula. [1] Thus we can solve 3-SAT problems by reducing them to finite language DFA intersection problems. [1]

   (c) The construction of each DFA requires a linear number of states, and thus could be carried out in polynomial time. [2] Thus DFAInterF is NP-Hard, and because it is also in NP, it is thus NP-Complete. [2]

   (d) We are given DFAs $A_1, \ldots, A_k$. Construct a combined DFA $A$ by adding a transition from all final states of $A_n$ to all initial states of $A_{n+1}$ [2]. The transition should be labelled by the special symbol #. The final states of $A_k$ should have a similar transition to a new final state [2]. This DFA would recognise the language that consists of words from each language in order, with each word terminated by a #. Clearly, this construction could be done in polynomial time [2]. Note that the word $w$ is in the intersection of the languages of our input DFAs if our constructed DFA $A$ accepts $w$# repeated $k$ times. Thus we have a polynomial-time reduction from DFA intersection (and thus DFAInterF) to KPow, and thus KPow is NP-hard. [2]

2. None of the problems are hard in themselves, but they require a reasonably well developed understanding of the course material in order to apply it in a slightly different context.

   (a) [bookwork+] Want to reduce$\underline{(\lambda x.\lambda y.xy)(\lambda x.y)w}$, but need to $\alpha$-convert to avoid variable capture, so:
   $(\lambda x.\underline{\lambda y.xy})(\lambda x.y)w \overset{\alpha}{\to} (\lambda x.\underline{\lambda z.xz})(\lambda x.y)w$, then
   $\underline{(\lambda x.\lambda z.xz)(\lambda x.y)w} \overset{\beta}{\to} (\underline{\lambda z.(\lambda x.y)z})w$ then
   $\underline{(\lambda z.(\lambda x.y)z)w} \overset{\beta}{\to} \underline{(\lambda x.y)w}$ then
   $\underline{(\lambda x.y)w} \overset{\beta}{\to} \underline{y}$
   3 for the reduction steps, 1 for commenting on $\alpha$.

   (b) [problem-solving] $\mathsf{SKK} \overset{\beta}{=} \mathsf{I}$ because $\mathsf{SKK}X \overset{\beta}{\to} \mathsf{K}X(\mathsf{K}X) \overset{\beta}{\to} X \overset{\beta}{\leftarrow} \mathsf{I}X$ [1 mark].

   To show that $\mathsf{SKK} \overset{\beta}{\neq} \mathsf{I}$, we observe that there is no $\beta$-rule that applies to $\mathsf{SKK}$ or to $\mathsf{I}$ [1 mark], and so they have no common redex, and are not equal by Church–Rosser [2].

(c) [problem] $\mathsf{SB}\ulcorner 0\urcorner XY \equiv \mathsf{SB}(\mathsf{KI})XY \overset{\beta}{\to} \mathsf{B}X(\mathsf{KI}X)Y \overset{\beta}{\to} \mathsf{B}X(I)Y \overset{\beta}{\to} X(IY) \overset{\beta}{\to} XY \overset{\beta}{\leftarrow} IXY$.

(d) [problem] Base case 0: $\mathsf{KI}XY = \mathsf{I}Y = X^0Y$ [1 mark].

Base case 1: done in part (c) [1 mark].

Inductive step: $\ulcorner n+1\urcorner XY = \mathsf{SB}(\ulcorner n\urcorner)XY = \mathsf{B}X(\ulcorner n\urcorner X)Y = X(\ulcorner n\urcorner XY) = X(X^nY)$. [2 marks]

(e) [problem, synthesis] The strategy is to convert a machine into a combinatory expression which rewrites into a coding of the final state (if any) of the machine. We might say that it rewrites into the numeral for the final contents of register 0, for example.

We can represent register contents directly as the corresponding combinatory numerals. They might carried as arguments.

The program can be represented in various ways. Probably the conceptually easiest would be to carry the (codes of) instructions throughout the rewriting as arguments to the main expression. Alternatively, the program can be packed up into a numeral, and instructions extracted on demand as with univeral machine constructions.

The execution of the increment and decrement is simple; executing a jump instruction after a successful zero test means rewriting to execute the target instruction, however they are labelled.

On executing the halt instruction, the combinary expression could rewrite into a simple numeral, or some other flagged state.

Then the halting of the original machine corresponds to the combinatory expression being $\beta$-rewritable to a flagged state.

Give roughly one mark for each of these (or other similar) points.