Faculty of Engineering, Built Environment and Information Technology
Department of Computer Science
COS341 ***CompilerConstruction***
SPECIAL Exam **O**pportunity **3**

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

1st of February 2023

Student: *Full* ***Name:*** __

Student: ***Number:*** ____

INSTRUCTIONS:

- *Any electronic devices* (cell phones, laptop computers, tablet computers, pocket calculators, etc.) are *strictly* ***forbidden***.
- Paper-based auxiliary materials (printed books and/or hand-scribbled crib notes) are *allowed up to a maximum weight* of 20 kilogram per student.
- Your answers *must* be written in ***indelible ink*** into the separate answer-booklet provided together with this question-paper. Answers *written with pencil (or other types of erasable ink) will not be marked* (= null points).
- You have ***3 hours*** work-time to complete this exam. (Extra time is *only* for students with a qualifying letter issued by the university's office for the disabled students.)
- All in all there are ***six Questions*** with a total value of ***24 Points***. *The exam is passed if at least 50% = 12 Points are achieved* (whereby any previous semester-marks are *no* longer taken into account).
- ***Wait*** until the invigilator gives you permission to start working.
- *Read* each question carefully and thoroughly before attempting to answer it.
- The invigilators in the exam room are *not allowed to provide any hints* which could possibly lead to the solution of a question that the student is supposed to answer.
- ***Return this question-paper*** (with the marking-grid displayed below) *together* with your answer-booklet.

33% = F

MARKING:

Question	Q1	Q2	Q3	Q4	Q5	Q6	SUM
Maximum	4	3	5	2	5	5	24
Result	X	X	4	X	1	3	8

Question 1 .... **[*4 Points*]**

Given is the alphabet

Σ := { **A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z,** **→ , ** }

a) ... *[1 Point]*
Provide a *regular expression* (*over* Σ) by means of which *any rule* of a context-free grammar can be produced. (Advice: assume that a rule of a context-free grammar is structured as shown in Chapter 2 of our book.)

b) ... [1 Point]
Provide another *regular expression* (over Σ) by means of which *any context-free grammar as a whole* can be produced, whereby you assume that two subsequent rules of such a grammar are separated from each other by means of the symbol \ (new line) which the given Σ contains. Note, thereby, that the "alternative" symbol | (from Chapter 2 of our book, page 41) is *not* provided as *object*-symbol inside Σ.

c) ... [2 Points]
Provide a *minimal DFA* (over Σ) which can *also* produce any context-free grammar, whereby again (as above) the symbol \ (new line) shall be used to separate two subsequent rules of the grammar from each other. (Note: *only* the final result needs to be shown - *not* all the steps of construction.)

Question 2 .... [3 Points]

As usual the set of all non-negative natural numbers is $N := \{\ \mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}, ...etc...\ \}$

Now let P be the the *largest possible sub*-set of N in such a way that *every* natural number n in P is of *palindromic* syntactic form (see book Chapter 2) when beheld as a string.

On these premises, provide a context-free grammar G such that $L(G) = P$, whereby the numbers n in $L(G)$ are beheld as strings.

Question 3 .... [5 Points]

From previous study-years you know that disjunctions ($\vee$) in combination with negations ($\neg$) are sufficient to implement all propositional Boolean logical functions.

Thus with the following context-free grammar G we can produce *all formulæ of propositional logic*; (for easier reference each of the eight rules of the grammar is numbered). Thereby, *Formula* is the start-symbol of G.

1. *Formula* → *Literal*
2. *Formula* → *Disjunction*
3. *Formula* → *Negation*
4. *Disjunction* → *Literal* ∨ *Literal*
5. *Disjunction* → *Literal* ∨ *Disjunction*
6. *Disjunction* → (*Disjunction*)
7. *Negation* → ¬(*Disjunction*)
8. *Negation* → ¬(*Disjunction*) ∨ *Formula*
9. *Negation* → *Formula* ∨ ¬(*Disjunction*)
10. *Negation* → ¬(*Formula* ∨ *Negation*)
11. *Negation* → ¬(*Formula* ∨ *Negation*) ∨ *Formula*
12. *Negation* → *Formula* ∨ ¬(*Formula* ∨ *Negation*)
13. *Negation* → ¬(*Negation* ∨ *Formula*)
14. *Negation* → ¬(*Negation* ∨ *Formula*) ∨ *Formula*
15. *Negation* → *Formula* ∨ ¬(*Negation* ∨ *Formula*)
16. *Literal* → *Atom*
17. *Literal* → ¬*Atom*
18. *Atom* → **v***Number*
19. *Number* → *Digit*
20. *Number* → *Digit Number*
21. *Digit* → **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

a) ... [3 Points]
Analyse by using Look-Ahead sets (LA) whether this grammar G **belongs to the class of LL1.**

b) ... [2 Points]

- IF your answer to the question (a) of above was "YES", then provide the **pseudo-code** of a **recursive descent parser** for *G*.
- IF your answer to the question (a) of above was "NO", then answer the following question and sub-questions:
  - Is it *certain* (*guaranteed*) that a correct other (non-**LL1**) parsing-algorithm *exists* for *G*?
    - IF "YES", then explain *why*.
    - IF "NO", then explain *why not*.

Note: An explanation must be provided (otherwise zero points)!

Question 4 .... ***[2 Points]***

A *tautology* is a logical formula which *always* evaluates to "true" (regardless of whether its *atoms* are considered to be true or false).

For example: the formula **¬(v18 ∨ v37) ∨ ¬v9 ∨ (v37 ∨ v18)**, which can indeed be produced by the grammar *G* of Question 3 of above, *is* a tautology.

Thus the question arises whether any context-free-grammar G could be written which produces *all* tautologies (none omitted) of propositional logic, *without* at the same time also producing any *non*-tautology? To have such a tautology-grammar *would* be very nice, because in such a case we would only need a context-free parsing algorithm in order to find out whether any given Boolean formula is a tautology – which, in turn, could be very beneficial for program code optimisation by a "smart" compiler.

However, our textbook states that *"the language of strings that can be constructed by repeating a string twice is not context-free"* [page 84 of 1st edition], or –which is the same– that *"the language of strings that can be constructed by concatenating a string with itself is not context-free"* [page 90 of 2nd edition].

Use the above-mentioned assertion from the book to ***prove convincingly*** that there *cannot* exist any context-free grammar *G* for propositional logical formulæ such that *L*(*G*) contains *no more and no less than all* those formulæ that are tautologies.

- **Advice:** Construct the proof as a "proof by contradiction": *Assume* that such a tautology-grammar *would* exist; then demonstrate that such a (wrong) assumption is *not* consistent with the already established knowledge.

Question 5 .... **[5 *Points*]**

The formula string **¬(v26 ∨ ¬v12)** is a valid member of the language *L*(*G*) with G being the logic formulæ grammar of Question 3 of above.

On the basis of Sub-Section 6.6.1 of our book, show *in details, step-by-step,* how this given formula gets *translated into intermediate code.*

At the end of all your detailed explanations, *also highlight* the *final result* of the translation process.

Question 6 .... **[5 *Points*]**

From Section 7.3 of our book you have learned that *"an instruction-set description is a list of pairs where each pair consists of a pattern (a sequence of intermediate-language instructions) and a replacement (a sequence of machine-code instructions)"*. In this question we want to simulate such a scenario by means of a few simple character-strings.

Given (on the next page) is now the following simplistic *matching table:*[1]

1 For comparison see Fig.7.1 in our textbook.

Source Pattern	*Target Pattern*
ABA	X
AC	Y
CAB	Z

On these premises are now requested to write a **high-level pseudo-code** function that fulfills *all* the requirements specified below:

- The function <u>must</u> be implemented *purely recursively* (NO while-loop, NO for-loop);
- The function *must* take as its *input-parameter* a *string* of arbitrary length;
- The function attempts to *match* the Source Patterns of above onto its given input string and
  - <u>returns</u> a string composed of the corresponding Target Patterns *IF* the matching-attempt was successful;
  - *otherwise* <u>returns</u> the string "-" to indicate that no match was possible.[2]

Examples and Further Advice:

- For input string "CABABACABAC" the function would <u>return</u> the string "ZXZY".
- For input string "ACACABAB" the function would <u>return</u> the string "-" (no match possible).
- *To keep your pseudo-code high-level* (i.e.: without needing to implement too many details at great length) please assume that the function's input-string can be split into two parts, "head" and "tail", such that the source pattern matches can be attempted on the "head" whereas the recursion would continue with the "tail". You are allowed to use "==" for string comparison, e.g.: ***if(string1==string2)...*** Moreover you may utilize an "append" operator ++ by means of which two strings can be "glued" together, e.g.: "Z"++"X" = "ZX".

END OF PAPER: THERE ARE NO FURTHER QUESTIONS

2 Ideally, in a correctly implemented compiler, such a mis-match *should never* occur. If such a mis-match can occur then there must be a "bug" anywhere in the compiler's software: perhaps in the front-end that produces intermediate code, perhaps in the back-end that produces target-code, or perhaps even in the design of the table in which all the pattern-matching pairs of code-snippets are defined.