LAST NAME (please print)	
First name (please print)	
Student Number	

WESTERN UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

CS3342: Organization of Programming Languages – Winter 2021
– Midterm Exam –

Wednesday, March 3, 3:30 - 5:30pm
Location: OWL

Instructor: Prof. Lucian Ilie

Submit your solutions as a *single PDF file*. Typeset answers are preferred but legible handwriting is also acceptable. Upload your solutions in *OWL* at most *30 minutes* after the end of the exam time (see above). If you have approved accommodation, add time to your exam accordingly and then submit also in OWL. Failure to submit within the allowed time will result in your exam being discarded.

This exam consists of 4 questions (6 pages, including this page), worth a total of 100 marks.
The exam is 120 minutes long and comprises 31% of your final grade.

(1) 20pt	
(2) 30pt	
(3) 20pt	
(4) 30pt	
Grade	

1. (20pt) Python strings are single-line character sequences surrounded by single or double quotation marks. Comments are either single-line character sequences that start with `#` and end at the end of the line, `\n`, or multi-line comments (technically they are multi-line strings but we call them comments here) surrounded by three consecutive double quotes, `"""`.

   (a) (14pt) Build a scanner for identifying Python strings and comments. Your scanner will be a deterministic finite automaton (such as the one on page 57 of textbook) that will identify only three types of tokens: (a) strings, (b) single-line comments, and (c) multi-line comments. That is, the scanner will not skip comments but rather identify those as tokens.

   (b) (6pt) The scanner reads as far as it can to identify the longest tokens. After finding a valid token, it starts the next token with the character right after the previous; if this is a white space (blank, tab, newline), then it skips all white spaces. The scanner finds an error when it cannot process the next character in a non-final state; in this case, all characters until the next white space and all consecutive white spaces that follow are ignored and the scanner resumes scanning with the next character.

   Give the output of the scanner for the following input; explain the output in detail using the scanner by running your finite automaton (show the states and transitions) on the input:

   ```
   "abc" " "" """ """" """""# ##\n
   ```

   (The last '`\n`' is to clarify there is a newline at the end.)

2. (30pt) Consider a language where assignments can appear in the same context as expressions; the value of `a = b = c` equals the value of `c`. The following grammar, G, generates such expressions that includes assignments in addition to additions and multiplications:

$$
\begin{array}{lrcl}
0. & P & \longrightarrow & E\texttt{\$\$} \\
1. & E & \longrightarrow & \texttt{id = } E \\
2. & E & \longrightarrow & TX \\
3. & X & \longrightarrow & \texttt{+ } TX \\
4. & X & \longrightarrow & \varepsilon \\
5. & T & \longrightarrow & FY \\
6. & Y & \longrightarrow & \texttt{* } FY \\
7. & Y & \longrightarrow & \varepsilon \\
8. & F & \longrightarrow & \texttt{(} E \texttt{)} \\
9. & F & \longrightarrow & \texttt{id}
\end{array}
$$

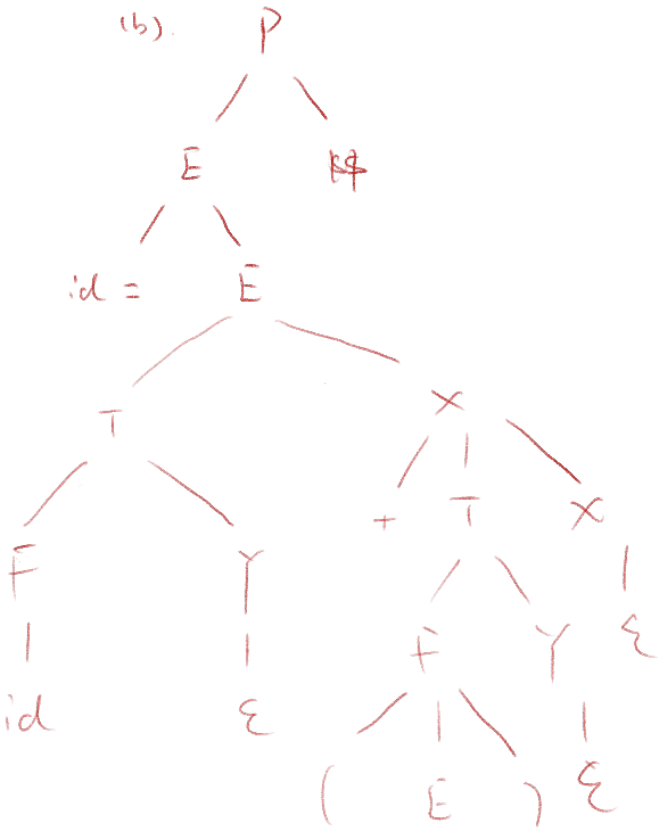
   (a) (2pt) What is the value of the expression: `a = 5 + (b = 2 + 6 * 4)$$`?

   (b) (3pt) Show a parse tree for the string: `id = id + (id = id + id * id)$$`.

   (c) (10pt) Compute, for each production $A \longrightarrow \alpha$, FIRST(α) and FOLLOW(A) using the algorithm on the next page; FIRST(α) is computed by the `string_FIRST` procedure. For each token added to each set, indicate the pair $(step, prod)$, where $1 \leq step \leq 3$ is the step in the algorithm (marked as 1, 2, 3; see next page) and $0 \leq prod \leq 9$ is the production involved; indicate $(0, -)$ when step 0 is used for terminals. (You can use jflap to help you, but you need to provide the information as mentioned. Information from jflap alone will not receive any points.)

   (d) (5pt) Using the information computed above, show that this grammar is not LL(1). (You need to use the definition on the last slide of the Syntax chapter.)

   (e) (10pt) Modify this grammar to make it LL(1). You can use jflap to check this; include the jflap table computation (include the jflap window).



```
-- EPS values and FIRST sets for all symbols:
    for all terminals c, EPS(c) := false; FIRST(c) := {c}  0
    for all nonterminals X, EPS(X) := if X ⟶ ϵ then true else false; FIRST(X) := ∅
    repeat
        ⟨outer⟩ for all productions X ⟶ Y_1 Y_2 ... Y_k,
            ⟨inner⟩ for i in 1..k
            1  add FIRST(Y_i) to FIRST(X)
                if not EPS(Y_i) (yet) then continue outer loop
            EPS(X) := true
    until no further progress

-- Subroutines for strings, similar to inner loop above:

    function string_EPS(X_1 X_2 ... X_n)
        for i in 1..n
            if not EPS(X_i) then return false
        return true

    function string_FIRST(X_1 X_2 ... X_n)
        return_value := ∅
        for i in 1..n
            add FIRST(X_i) to return_value
            if not EPS(X_i) then return

-- FOLLOW sets for all symbols:
    for all symbols X, FOLLOW(X) := ∅
    repeat
        for all productions A ⟶ α B β,
        2  add string_FIRST(β) to FOLLOW(B)
        for all productions A ⟶ α B
                or A ⟶ α B β, where string_EPS(β) = true,
        3  add FOLLOW(A) to FOLLOW(B)
    until no further progress

-- PREDICT sets for all productions:
    for all productions A ⟶ α
        PREDICT(A ⟶ α) := string_FIRST(α) ∪ (if string_EPS(α) then FOLLOW(A) else ∅)
```

3. (20pt) Consider the following pseudocode:

```
int x = 0
set_x (int n) { x = n }
print_x() { print(x) }
f() { set_x(1); print_x() }
g() { int x = 0; set_x(2); print_x() }

set_x(0); f(); print_x(); g(); print_x()
```

use global x

(a) (10pt) What is the output of the program if it uses static scoping?

(b) (10pt) What is the output of the program if it uses dynamic scoping? use local x.

Explain each answer by running the program statement by statement and detailing what happens.

(a). 1122

(b) 1121.

local change would affect global val if there's no corresponding local var.

4. (30pt) Consider the following SLR(1) grammar, G, for boolean expressions containing operands (`id`), operators (`and`, `or`), and parentheses:

$$
\begin{array}{lcl}
P & \longrightarrow & B\texttt{\$\$} \\
B & \longrightarrow & B \texttt{ or } D \\
B & \longrightarrow & D \\
D & \longrightarrow & D \texttt{ and } C \\
D & \longrightarrow & C \\
C & \longrightarrow & \texttt{(}B\texttt{)} \\
C & \longrightarrow & \texttt{id}
\end{array}
$$

   (a) (20pt) Construct an attribute grammar, based on G, that evaluates the value, `val`, of a boolean expression by avoiding unnecessary computations; e.g., if A is true, then A `or` B is true, so there is no need to evaluate B; similarly, if A is false, then A `and` B is false, so, again, there is no need to evaluate B. Each `id` has a known attribute `val` that is true or false. The attribute grammar must be designed such that the value of any expression can be computed by a single traversal of the parse tree.

   (b) (10pt) Using this attribute grammar, draw a decorated tree for the following boolean expression:

`(u or v) and (x or y) or z $$`,

where `u,v,x,y,z` are `id`'s such that `u.val = v.val = y.val = false`, `x.val = z.val = true`.

P.val ← B.val
B.val ← B.val or D.val
B.val ← D.val
D.val ← D.val and C.val
D.val ← C.val
C.val ←