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

WESTERN UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

CS3342: Organization of Programming Languages – Winter 2020
– Midterm Exam –

Wednesday, Mar. 11, 2020, 3:30 - 5:30pm
WSC-55: Abdel - Liang
SSC-2050: Lim - Zinn

Instructor: Prof. Lucian Ilie

This exam consists of 6 questions (8 pages, including this page), worth a total of 100 marks. **No other materials are allowed, such as cheat-sheets (or any other sheets), books, or electronic devices.** All answers are to be written in this booklet. If you continue your answer on a different page, indicate this clearly. If you submit answers on loose pages, write your name and question number on each. Scrap work can be done anywhere on the empty pages. The exam is 120 minutes long and comprises 31% of your final mark.

(1) 15pt	
(2) 25pt	
(3) 10pt	
(4) 15pt	
(5) 15pt	
(6) 15pt	
Grade (+5)	

1. (15pt) Write a regular expression that represents the set of all possible programming language identifiers satisfying the following rules:

   - identifiers are strings over the alphabet: $\{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}, \mathtt{A}, \mathtt{B}, \ldots, \mathtt{Z}\} \cup \{\mathtt{0}, \mathtt{1}, \ldots, 9\} \cup \{\_\}$,
   - start with a letter,
   - do not contain two consecutive underscores (__),
   - do not end with an underscore,
   - only a letter can come after an underscore,
   - the digit 0 cannot come right after a letter.

Solution:

$$
\begin{array}{rcl}
\mathit{letter} & \longrightarrow & \mathtt{a} \mid \mathtt{b} \mid \ldots \mid \mathtt{z} \mid \mathtt{A} \mid \mathtt{B} \mid \ldots \mid \mathtt{Z} \\
\mathit{p\_digit} & \longrightarrow & \mathtt{1} \mid \mathtt{2} \mid \ldots \mid 9 \\
\mathit{digit} & \longrightarrow & \mathtt{0} \mid \mathit{p\_digit} \\
\mathit{id} & \longrightarrow & \mathit{letter}\ (\_\ \mathit{letter} \mid \mathit{letter} \mid \mathit{p\_digit} \mid \mathit{pdigit}\ \mathit{digit}^*)^*
\end{array}
$$

2. (25pt) Consider the following context-free grammar, G, for the language of balanced parentheses:

$$\begin{array}{lcl} P & \longrightarrow & S\$\$ \\ S & \longrightarrow & SS \\ S & \longrightarrow & \texttt{(}S\texttt{)} \\ S & \longrightarrow & \texttt{()} \end{array}$$

(a) (5pt) Prove that G is not $LL(1)$.

(b) (10pt) Construct an $LL(1)$ grammar G_1, equivalent with G. Compute the sets $\textsc{predict}(A \longrightarrow \alpha)$, for all productions of G_1, and explain why there is no conflict.

(c) (5pt) Show a parse tree for the string `(())()$$` in G_1.

(d) (5pt) Show a step-by-step, top-down parsing of the string `(())()$$` in G_1. You need to show, at each step, the remaining input string, the parse stack, and the predicted rule or matched token.

(*Hint:* You need to eliminate left recursion and common prefixes. If you are not able to construct an $LL(1)$ grammar, solve still (c) for the grammar given, G. Of course, (d) does not work for G.)

Solution:

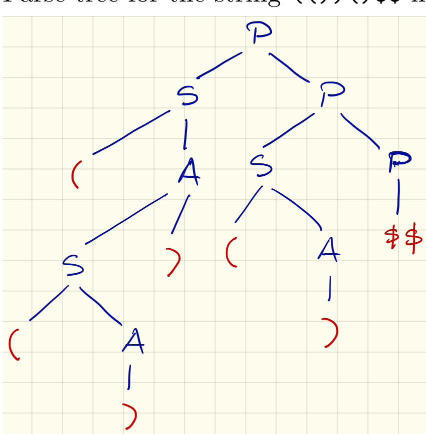
(a) G is not $LL(1)$ because $'\texttt{(}' \in \textsc{predict}(S \longrightarrow SS) \cap \textsc{predict}(S \longrightarrow \texttt{(}S\texttt{)}) \cap \textsc{predict}(S \longrightarrow \texttt{()})$.

(b) G_1 is:

	Production rules	PREDICT()
1.	$P \longrightarrow SP$	$\{\texttt{(}\}$
2.	$P \longrightarrow \$\$$	$\{\$\$\}$
3.	$S \longrightarrow \texttt{(}A$	$\{\texttt{(}\}$
4.	$A \longrightarrow S\texttt{)}$	$\{\texttt{(}\}$
5.	$A \longrightarrow \texttt{)}$	$\{\texttt{)}\}$

There are no conflicts since any two rules with the same LHS have disjoint $\textsc{predict}()$ sets.

(c) Parse tree for the string `(())()$$` in G_1.



(d) Top-down parsing of `(())()$$` in G_1:

Remaining input string	Parse stack	Predicted rule / match
`(())()$$`	P	1
`(())()$$`	SP	3
`(())()$$`	`(`AP	match `(`
`())()$$`	AP	4
`())()$$`	S`)`P	3
`())()$$`	`(`A`)`P	match `(`
`))()$$`	A`)`P	5
`))()$$`	`))`P	match `(`
`)()$$`	`)`P	match `(`
`()$$`	P	1
`()$$`	SP	3
`()$$`	`(`AP	match `(`
`)$$`	AP	5
`)$$`	`)`P	match `(`
`$$`	P	2
`$$`	`$$`	match `$$`

3. (10pt) Consider the following program (written using Pascal syntax):

```
program main;
    var x : integer;
    procedure sub1;
       begin
           write(x);
           x := 7
       end;
    procedure sub2;
       var x : integer;
       begin
           sub1;
           x := x + 2
       end;
begin
    x := 31;
    sub2;
    sub1
end.
}
```

What is printed by the above program:

(a) (5pt) under static scoping rules?

(b) (5pt) under dynamic scoping rules?

Explain your answers.

Solution:

(a) under static scoping rules: `31, 7`.
`sub1` always refers to the global `x`.

(b) under dynamic scoping rules: `?, 31`.
`sub1` called from `sub2` prints the local `x`, which has not been initialized; `sub1` called from the main program prints the global `x`.

4. (15pt) Consider the following grammar for reverse Polish arithmetic expressions:

$$\begin{array}{lll}
\textit{expr} & \longrightarrow & \textit{expr}\ \ \textit{expr}\ \ \textit{oper} \\
\textit{expr} & \longrightarrow & \texttt{id} \\
\textit{oper} & \longrightarrow & \texttt{+} \\
\textit{oper} & \longrightarrow & \texttt{-} \\
\textit{oper} & \longrightarrow & \texttt{*} \\
\textit{oper} & \longrightarrow & \texttt{/}
\end{array}$$

(a) (10pt) Assuming that each `id` has a synthesized attribute `name` of type string, and that each *expr* and *oper* has an attribute `val` of type string, write an attribute grammar that computes in the `val` attribute of the root of the parse tree the infix translation of the postfix expression represented by the tree. For example, if the parse tree is for the string `A B / C D E - * +`, then the `val` attribute of the root will equal `((A / B) + (C * (D - E)))`.

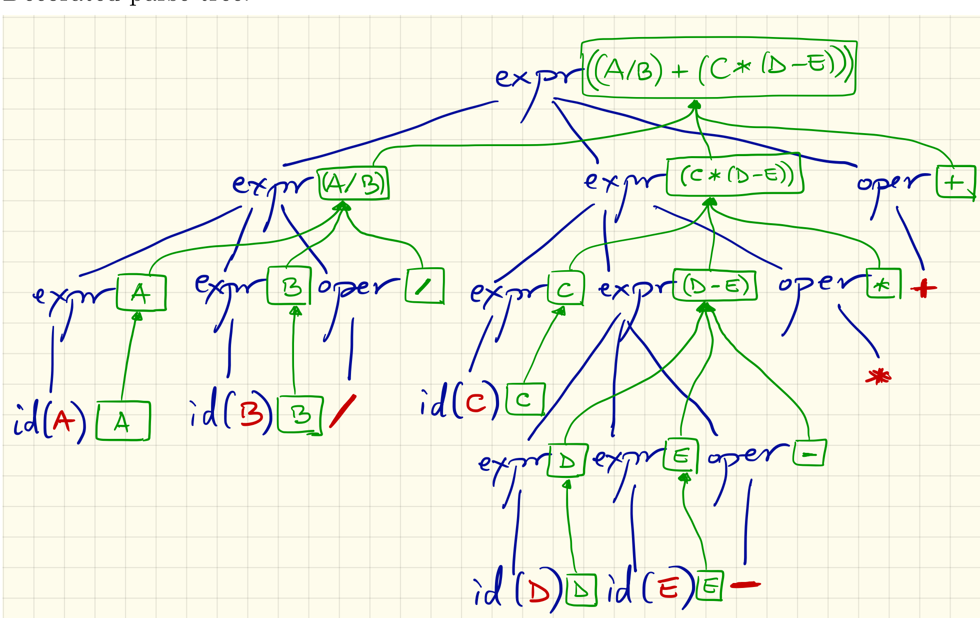
(b) (5pt) Show the decorated parse tree for the string `A B / C D E - * +`.

Solution:

(a) The attributed grammar:

$$\begin{array}{llll}
\textit{expr}_1 & \longrightarrow & \textit{expr}_2\ \ \textit{expr}_3\ \ \textit{oper} & \triangleright\ \textit{expr}_1\texttt{.val} = \texttt{concat("(",}\textit{expr}_2\texttt{.val},\textit{oper}\texttt{.val},\textit{expr}_3\texttt{.val,")")} \\
\textit{expr} & \longrightarrow & \texttt{id} & \triangleright\ \textit{expr}\texttt{.val} = \texttt{id.name} \\
\textit{oper} & \longrightarrow & \texttt{+} & \triangleright\ \textit{oper}\texttt{.val} = \texttt{"+"} \\
\textit{oper} & \longrightarrow & \texttt{-} & \triangleright\ \textit{oper}\texttt{.val} = \texttt{"-"} \\
\textit{oper} & \longrightarrow & \texttt{*} & \triangleright\ \textit{oper}\texttt{.val} = \texttt{"*"} \\
\textit{oper} & \longrightarrow & \texttt{/} & \triangleright\ \textit{oper}\texttt{.val} = \texttt{"/"}
\end{array}$$

(b) Decorated parse tree:



5. (15pt) Assume the following code in a language that uses static scoping:

```
procedure A
    procedure B
        procedure C
        begin ... (* body of C *) ... end
        procedure D
        begin ... (* body of D *) ... end
    begin ... (* body of B *) ... end
    procedure E
        procedure F
        begin ... (* body of F *) ... end
    begin ... (* body of E *) ... end
begin ... (* body of A *) ... end
procedure H
begin ... (* body of H *) ... end
```

For each of the stack configurations below, say whether it is possible or not. If it is not possible, then explain why. (The stack has the bottom on the left and top on the right.)

(a)	(5pt)	H	A	H	A	E	B	C	B	C	D		⇄
(b)	(5pt)	A	B	E	C	D	B	C	D	B	C		⇄
(c)	(5pt)	F	E	F	E	A	H	H	H	H	H		⇄

Solution:

(a) Possible.

(b) Impossible: `E` cannot call `C`.

(c) Impossible: `F` is not visible from the main program.

6. (15pt) Assume a programming language X which uses short-circuit evaluation for Boolean expressions and another programming language Y which does not use short-circuit evaluation.

   (a) (5pt) Consider the following piece of code in the X language:

   ```
   if (A == 0) or (B < C) and (A != C) then stmt_1
   else stmt_2
   ```

   Assuming `and` has higher precedence than `or`, describe, in words, the behaviour of the above code.

   (b) (10pt) Simulate the above code in the Y language. You are not allowed to use any additional variables (e.g., to store intermediate Boolean results). The Y language is assumed to have similar syntax.

Solution:

(a) The expression `(A == 0)` is evaluated first. If it is true, then `stmt_1` is executed. Otherwise, `(B < C)` is evaluated. If it is false, then `stmt_2` is executed. Otherwise, `(A != C)` is evaluated. If it is true, then `stmt_1` is executed, otherwise, `stmt_2` is executed.

(b) The Y code corresponds to the above description:

```
if (A == 0) then stmt_1
else if (B >= C) then stmt_2
     else if (A != C) then stmt_1
          else stmt_2
```