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

WESTERN UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

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

Tuesday, April 20, 7:00pm - 10:00pm
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 6 questions (7 pages, including this page), worth a total of 100 marks.
The exam is 180 minutes long and comprises 39% of your final grade.

(1) 15pt	
(2) 15pt	
(3) 15pt	
(4) 20pt	
(5) 15pt	
(6) 20pt	
Grade	

1. (15pt) Consider the following grammar for Prolog syntax (with some helpful comments on the right):

1.	T	$\rightarrow$	$A \mid N \mid V \mid S$	(term, atom, number, variable, structure)
2.	T_s	$\rightarrow$	$T \mid T$ `,` T_s	(terms)
3.	S	$\rightarrow$	A `(` T_s `)`	
4.	L	$\rightarrow$	`a` $\mid$ `b` $\mid \cdots \mid$ `z`	(lower case)
5.	U	$\rightarrow$	`A` $\mid$ `B` $\mid \cdots \mid$ `Z`	(upper case)
6.	D	$\rightarrow$	`0` $\mid$ `1` $\mid \cdots \mid$ `9`	(digit)
7.	C	$\rightarrow$	$L \mid U \mid D$	(character)
8.	I	$\rightarrow$	$CI \mid C$	(id)
9.	A	$\rightarrow$	LI	
10.	V	$\rightarrow$	UI	
11.	N	$\rightarrow$	$DN \mid D$	
12.	F	$\rightarrow$	T `.`	(fact)
13.	R	$\rightarrow$	T `:-` T_s `.`	(rule)
14.	Q	$\rightarrow$	`?-` T_s `.`	(query)

   (a) (5pt) Explain, using the definition (last slide of the Syntax chapter) why this grammar is not LL(1).

   (b) (10pt) Modify the grammar to be LL(1). You do not have to copy the entire grammar. Instead, replace only the rules that have problems. (Formal proof that the new grammar is LL(1) is not required.)

(a) not LL(1): $a \in First(CI)$ and $a \in First(C)$.

(b) Expand.

2. (15pt) The "mark-and-sweep" method for garbage collection (slides 37-38 of the Types chapter) can use "pointer reversal" to avoid the use of a stack, thus saving memory. The size of the stack is never larger than the longest chain through the heap. If we can store the heap as a balanced tree data structure, then the longest chain is the same as the maximum depth of the tree, which is $\mathcal{O}(\log n)$ for balanced trees. For instance, for 1GB of RAM, $\log n$ is quite small: $\log_2(1\text{ GB}) = \log_2(2^{30}) = 30$.

   (a) (5pt) Given a heap of 1 TB, what is the maximum size of the stack?

   (b) (10pt) Is it worth taking the trouble to implement the pointer reversal procedure? Explain your answer.

3. (15pt) Given the λ-expressions:

`True`	$\equiv$	$\lambda xy.x$	`car`	$\equiv$	$\lambda\ell.\ell$ `True`
`False`	$\equiv$	$\lambda xy.y$	`cdr`	$\equiv$	$\lambda\ell.\ell$ `False`
`if`	$\equiv$	$\lambda cte.c\ t\ e$	`null?`	$\equiv$	$\lambda\ell.\ell\ (\lambda xy.$`False`$)$
`cons`	$\equiv$	$\lambda xyf.f\ x\ y$	`nil`	$\equiv$	$\lambda\ell.$`True`

reduce the following expressions to their normal form:

(a) (5pt) `car` (`cons` A `nil`)

(b) (5pt) `cdr` (`cons` A `nil`)

(c) (5pt) `if` (`null?` `nil`) `True` `False`

Show all computations; indicate the reduction being performed by underlying the abstraction and argument: $(\underline{\lambda x}.M)\underline{N}$. The order of the reductions is not important.

(a)

3. (15pt) Given the λ-expressions:

`True`	$\equiv$	$\lambda xy.x$	`car`	$\equiv$	$\lambda\ell.\ell$ `True`
`False`	$\equiv$	$\lambda xy.y$	`cdr`	$\equiv$	$\lambda\ell.\ell$ `False`
`if`	$\equiv$	$\lambda cte.c\ t\ e$	`null?`	$\equiv$	$\lambda\ell.\ell\ (\lambda xy.$`False`$)$
`cons`	$\equiv$	$\lambda xyf.f\ x\ y$	`nil`	$\equiv$	$\lambda\ell.$`True`

reduce the following expressions to their normal form:

(a) (5pt) `car` (`cons` A `nil`)

=> $\lambda L.L\ \text{True}\ ((\lambda xyf.fxy)\ A\ (\lambda L.\text{True}))$

=> $((\lambda xyf.fxy)\ A\ (\lambda L.\text{True}))\ \text{True}$

=> $((\lambda yf.fAy)\ (\lambda L.\text{True}))\ \text{True}$

=> $(\lambda f.f\ A\ (\lambda L.\text{True}))\ \text{True}$

=> $\text{True}\ A\ (\lambda L.\text{True})$

=> $(\lambda xy.x)\ A\ (\lambda L.\text{True})$

=> $\lambda y.A\ (\lambda L.\text{True})$

=> $A\ (\lambda L.\text{True})$

=> $A\ (\lambda L.(\lambda xy.x))$

=> A

3. (15pt) Given the λ-expressions:

`True`	$\equiv$	$\lambda xy.x$	`car`	$\equiv$	$\lambda\ell.\ell$ `True`
`False`	$\equiv$	$\lambda xy.y$	`cdr`	$\equiv$	$\lambda\ell.\ell$ `False`
`if`	$\equiv$	$\lambda cte.c\ t\ e$	`null?`	$\equiv$	$\lambda\ell.\ell\ (\lambda xy.$`False`$)$
`cons`	$\equiv$	$\lambda xyf.f\ x\ y$	`nil`	$\equiv$	$\lambda\ell.$`True`

(b) (5pt) `cdr` (`cons` A `nil`)

=>

4. (20pt) This question concerns the `and` boolean operator of Scheme, which has variable arity and uses lazy evaluation.

   (a) (5pt) Define a binary operator, `bin-strict-and`, that corresponds to strict evaluation of `and`.

   (b) (5pt) Give a concrete example where `and` works and `bin-strict-and` does not, that is, `(and x y)` produces a value whereas `(bin-strict-and x y)` gives an error.

   (c) (10pt) Define an operator, `strict-and`, that corresponds to strict evaluation of `and`. This operator has variable arity.

   You are not allowed to use imperative features of Scheme.

5. (15pt) Consider the following Prolog facts and rules:

```
brothers(X, Y) :- mother(M, X), mother(M, Y), father(F, X), father(F, Y), boy(X), boy(Y).
```

Rewrite this rule in predicate calculus, using appropriate quantifiers:

(a) (5pt) using four universal quantifiers.

(b) (10pt) using two universal quantifiers and two existential quantifiers.

(a). $\forall x \forall y \forall M \forall F \quad mother(x, Y) \wedge father(F, x) \wedge father(f, Y) \wedge boy(x) \wedge boy(Y)$
$\rightarrow brothers(x, Y)$

(b)

6. (20pt) Consider the following Prolog facts and rules:

```
append([], Y, Y).
append([H|X], Y, [H|Z]) :- append(X, Y, Z).
member(X, [X|_]).
member(X, [_|T]) :- member(X, T).
member2(X, L) :- append(P, [X|_], L), not(member2(X, P)).
```

The `member` predicate returns all elements of a list when used as shown below (left). To produce each element only once, we can use `member2`; see below (middle). Note further that `member2` produces unique elements in the order of their first occurrence.

```
?- member(X, [a,b,c,b,a]).
X = a ;
X = b ;
X = c ;
X = b ;
X = a ;
false.
```

```
?- member2(X, [a,b,c,b,a]).
X = a ;
X = b ;
X = c ;
false.
```

```
?- member3(X, [a,b,c,b,a]).
X = c ;
X = b ;
X = a ;
false.
```

(a) (10pt) Write a new predicate `member3` which, when used as above, produces unique elements in the order of their last occurrence, as seen above (right).

(b) (10pt) Draw the Prolog tree for the computation below; like in the examples we discussed, you need to show, on each brach, the rule used and the substitution (if any), the branches not attempted because of cuts (`!` – indicate which cut prevented which branch), and the output produced (solutions and the `false`).

```
?- member3(X, [a,b,a]).
X = b ;
X = a ;
false.
```

```
1 append([], Y, Y).
2 append([H|X], Y, [H|Z]) :- append(X, Y, Z).
3 member(X, [X|_]).
4 member(X, [_|T]) :- member(X, T).
5 member2(X, L) :- append(P, [X|_], L), not(member2(X, P)).
```

(b) member3(X, [a,b,a])