UCLA CS 131 Midterm, Spring 2018
100 minutes total, open book, open notes
closed computer. Exam is DOUBLE SIDED.

Name: Derek Chu Student ID: [redacted]

1	2	3	total
6+2 +8 = 30	23	11	64

1. Consider the following EBNF grammar for a subset of OCaml. The start symbol is "expr".

```
expr:
  constant
  '(' expr ')'
  expr {',' expr}+
  '[' expr {';' expr} [';'] ']'
  expr {expr}+
  '-' expr
  expr infix-op expr
  'if' expr 'then' expr ['else' expr]
  'match' expr 'with' pattern-matching
  'function' pattern-matching
  'fun' {parameter}+ '->' expr
  'let' ['rec'] let-binding 'in' expr

pattern-matching:
  ['|'] pattern '->' expr {'|' pattern '->' expr}

let-binding:
  pattern '=' expr

parameter:
  pattern

constant:
  INTEGER-LITERAL
  STRING-LITERAL
  'false'
  'true'
  '(' ')'
  '[' ']'

infix-op:
  '='
  '+'
  '-'
  '*'
  '/'

pattern:
  IDENTIFIER
  '_'
  constant
  '(' pattern ')'
  pattern {',' pattern}+
  '[' pattern {';' pattern}  [';'] ']'
```

1a (2 minutes). What are the nonterminals of this grammar?

1b (6 minutes). Show that the grammar is ambiguous, even if you remove the infix-op rule so that no program can contain infix-ops.

1c (10 minutes). Translate this grammar to BNF. Make as few changes as possible. Write your BNF in the same style of the grammar.

1d (10 minutes). Convert the grammar to syntax diagram form. Make the diagram as concise and clear as you can, and eliminate nonterminals when possible.

1e (8 minutes). If you took the BNF version of this grammar, converted it to a form suitable for Homework 2, and submitted it to a correct solution to Homework 2, an infinite loop could result. Briefly explain why.

1f (8 minutes). Fix the BNF version of this grammar so that it does not make Homework 2 loop forever.

1g (10 minutes). Fix the BNF version of this grammar so that it is no longer ambiguous. (Do not worry about looping forever.)

2. Consider the following OCaml definitions, which is a simplified version of the hint code for Homework 2 except with a somewhat different API.

```
 1 type nucleotide = A | C | G | T
 2
 3 type fragment = nucleotide list
 4
 5 type pattern =
 6   | Frag of fragment
 7   | List of pattern list
 8   | Or of pattern list
 9
10
11 let match_empty accept frag = accept frag
12
13 let match_nothing _ _ = None
14
15 let match_nt nt accept = function
16   | [] -> None
17   | n::t -> if n == nt then accept t else None
18
19 let append_matchers matcher1 matcher2 accept =
20   matcher1 (matcher2 accept)
21
22 let make_append make_a_matcher ls =
23   let rec mams = function
24     | [] -> match_empty
25     | h::t -> append_matchers
26                 (make_a_matcher h) (mams t)
27   in mams ls
28 let rec make_or mm = function
29   | [] -> match_nothing
30   | h::t ->
31      let head_matcher = mm h
32      and tail_matcher = make_or mm t
33      in fun accept frag ->
34           match head_matcher accept frag with
35             | None -> tail_matcher accept frag
36             | something -> something
37
38 let rec make_matcher = function
39   | Frag frag -> make_append match_nt frag
40   | List pats -> make_append make_matcher pats
41   | Or pats -> make_or make_matcher pats
```

2a (14 minutes). Give the types of each function defined at the top level in this code.

2b (8 minutes). Suppose the line 16 '| [] -> None' in match_nt were changed to '| [] -> accept []'. How would this affect the behavior of the program? Briefly describe the effect at the level of the user who is calling make_matcher.

2c (8 minutes). Suppose instead that lines 24 and 29 were swapped. Explain what would go wrong; give two distinct examples, one for each line.

3. Java does not provide a mechanism to declare array elements as being volatile; accesses to array elements are normally considered to be normal accesses. Suppose we define a new language JavaV that is like Java, except that if you declare an array with the keyword 'volatile' immediately after the '[' of the array's type (e.g., 'long foo[volatile];'), accesses to that array's elements are volatile accesses.

3a (8 minutes). Give an example JavaV program that has well-defined behavior, whereas the same program in Java (i.e., without 'volatile' after '[') would have a race condition.

3b (8 minutes). In JavaV, should 'long [volatile]' be a subtype of 'long []', or vice versa, or should neither be a subtype of the other? Briefly explain.

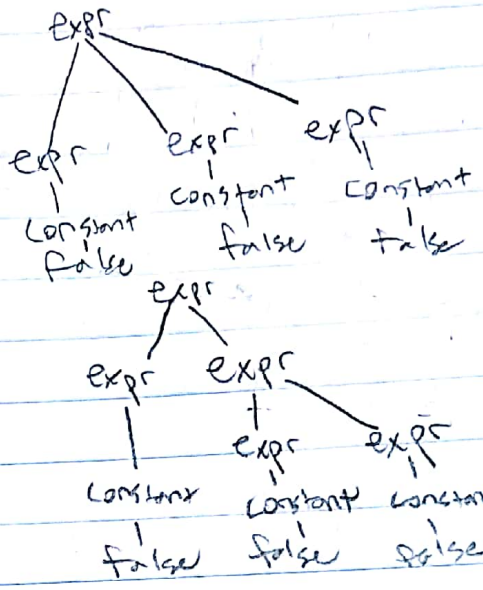
CS 131 Midterm

1. a. Non-terminals: expr, pattern-matching, let-binding, parameters, constant, infix-op, pattern

(2)

b. For the string "false false false"

we could have:



(6)

or we could have:

we have the grammar ambiguous since even with the infix-op rule removed, we can have multiple parse trees for the same tree.

c. BNF grammar:

missing case
expr → [expr]

```
expr = constant
expr = '(' expr ')'
expr = '[' expr semexp (';' expr)* sem ']'
semexp = semexp ';' expr | <empty>
sem = ';' | <empty>
expr = expr expr
expr = '-' expr
expr = expr infix-op expr
expr = 'if' expr 'then' expr | 'if' expr 'then' expr 'else' expr
expr = 'match' expr 'with' pattern-matching
expr = 'function' pattern-matching
expr = 'fun' param parameter '->' expr
param = param parameter | <empty>
expr = 'let' let-binding 'in' expr | 'let' 'rec' let-binding 'in' expr
```

missing
expr → expr{, expr}+

missing case pattern-matching → pattern → expr

pattern-matching = bar pattern `->` expr pat | `|` pattern `->` expr
pat = pat `|` pattern `->` expr | <empty>
bar = `|` | <empty>

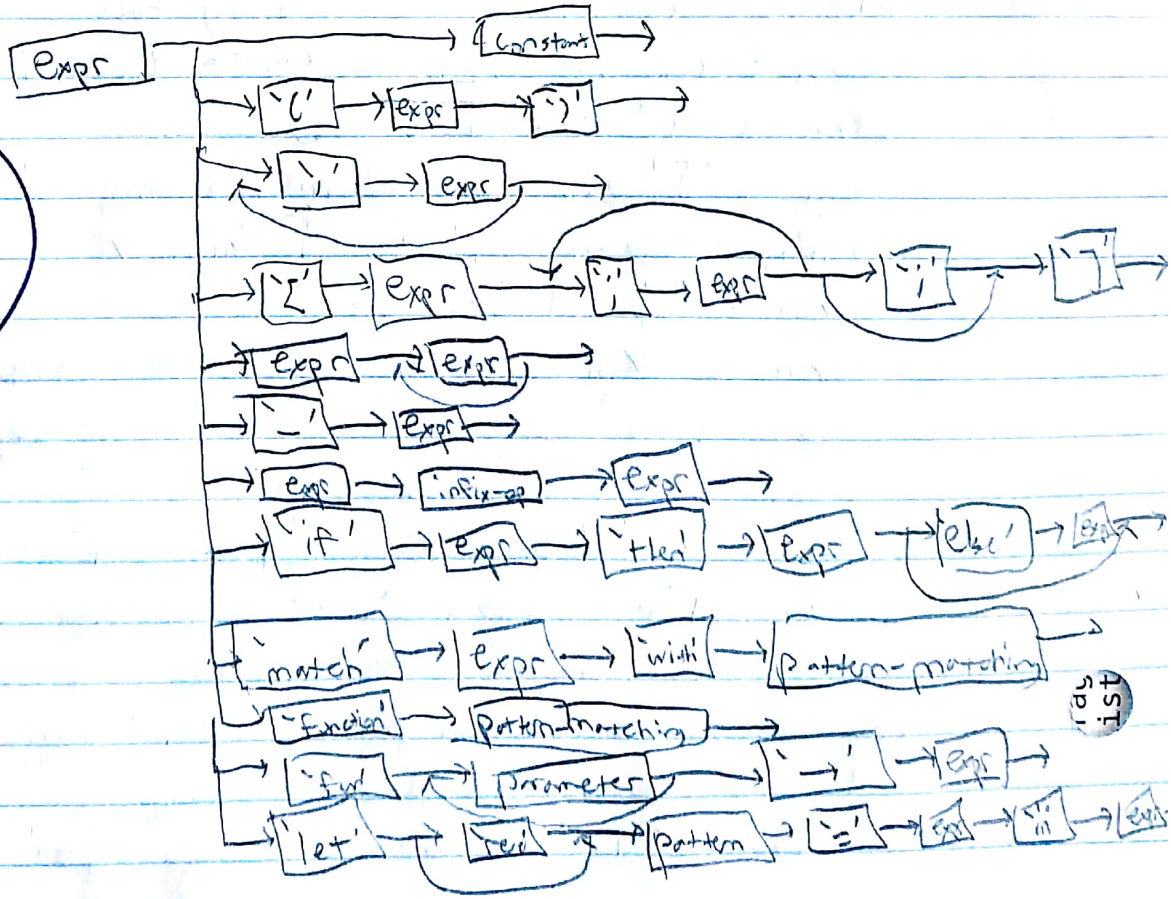
let-binding = pattern `=` expr
parameter = pattern
constant = INTEGER-LITERAL | STRING-LITERAL | `false` | `true` | `(` `)` | `[` `]`
infix-op = `=` | `+` | `-` | `*` | `/`
pattern = IDENTIFIER
pattern = `_`
pattern = constant
pattern = `(` pattern `)`
pattern = pattern otherpat `,` pattern
otherpat = otherpat `,` pattern | <empty>
pattern = `[` pattern semipat `;` pattern `]` | `[` pattern `]`
semipat = semipat `;` pattern | <empty>

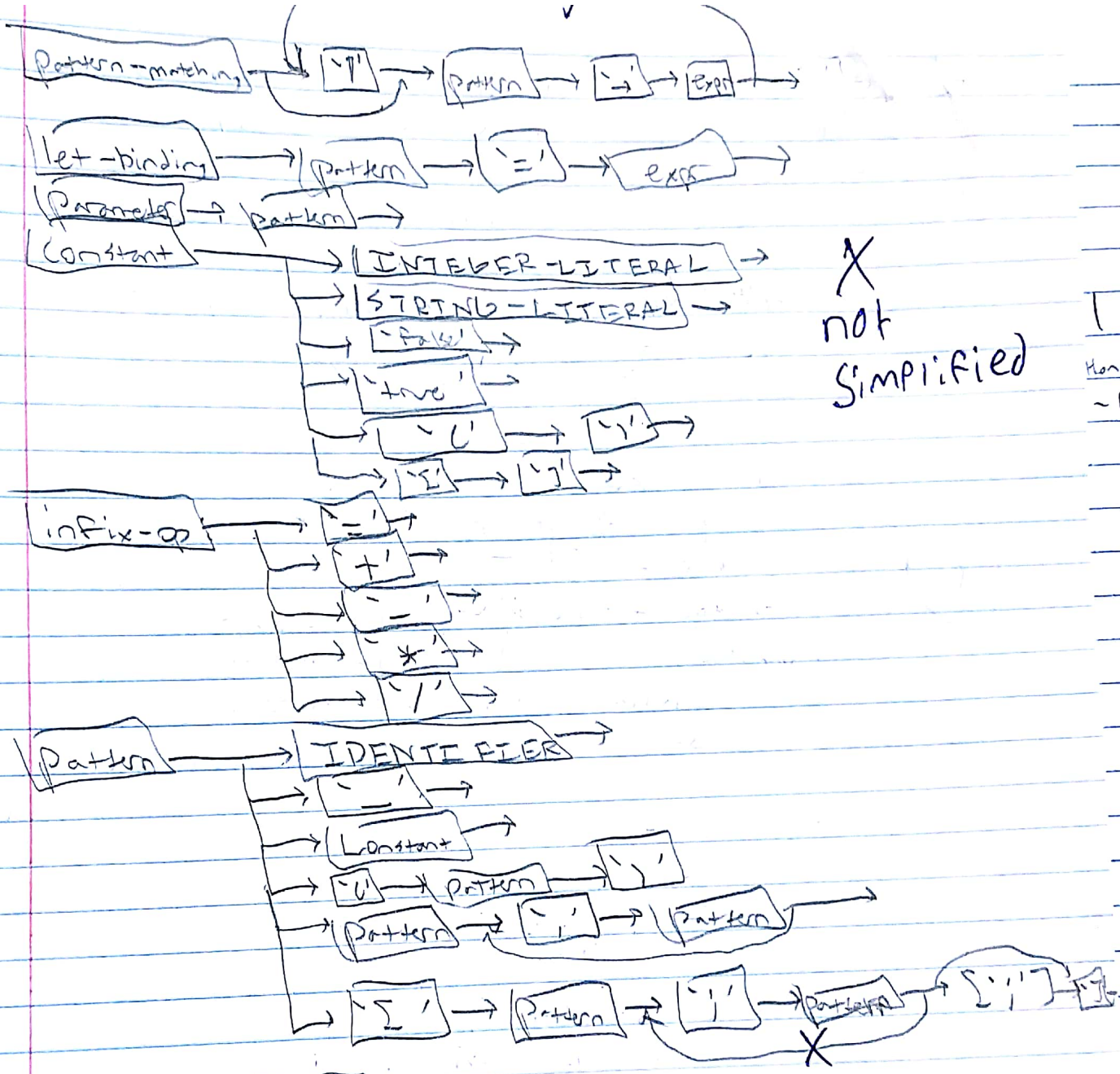
missing case pattern → [pattern]

↑ defined on prev. page

+6

d.





X not Simplified

e. expr = expr expr rule could result in us attempting to continually expand the left expr since our implementation does not support left-recursion. Thus, our Homework 2 would continue to attempt to expand this rule in an infinite loop as our implementation only works on rules that feature right-recursion. This is especially the case if all the rules that come before this rule are always rejected since then our only option would be to continue to use this rule.

+8

f. Make expr = expr expr the last possible rule for expr
This way, we can just try out every rule
before attempting that rule which prevents us
from entering an infinite loop that just attempts
to expand this rule. Also now pattern = pattern { ',' pattern }
doesn't stop all left recursion to be the last rule.

expr = T

g. expr = constant
T

expr = '(' expr ')'

T = expr

expr = expr { ',' expr }+

expr = 'Σ' T { ':' T } Σ':' ']' '['

expr = semexp ':' T | <empty>

sem = ';' | <empty>

expr =

(0)

2. a. match_empty: ('a → 'b) → 'a → 'b ✓

(11) match_empty: 'a → 'b → None option -1

match_one: 'a → ('a list → 'b option) → 'a list → 'b option ✓

append_matchers: ('b → 'c) → ('a → 'b) → 'a → 'c ✓

make_append: ('a → 'b) → 'a list → ('c → 'd) → 'c → 'd -1

make_or_matcher: (('a → 'b) → 'c → 'd) → 'a list → 'e → 'f → 'f option -1

make_matcher: pattern → ('a list → 'b option) → 'a list → 'b option

fragment fragment

(8) b. This would result in the user calling make_matcher to get the wrong result in the end. This is due to how if we pass in an empty list, then we would just call accept on the empty list which is not what is wanted. For example, if we wanted to match to a fragment that looks like this: ["A"; "C"; "G"] but then we pass the empty list instead, then we do not attempt to create a parse tree for this fragment and instead just attempt to see if the empty fragment can be passed into the accept function to get the correct result. Thus, the behavior of the program would no longer work with this kind of implementation.

(4) c. This would also result in wrong behavior. If line 29 were in line 24's place, then we would return +3 None always when we really just wanted some function that would be able to test frag using accept. Thus, this would result in the empty list always returning the wrong matcher function. If line 24 were in line 29's place, +1 then we would get a different matcher that does not follow the specific behavior that we want as we just want to return a matcher that returns None for an empty list.

13

3. a. class C{
long foo [volatile];
void m(){
int n = foo[0];
foo[0] = n;
if(foo[0] != n){
throw Error();
}
}
}

In a multithreaded environment, if we didn't have the array accesses as volatile accesses, then there would be a race condition as Java would reorder and take out code to suit its needs. We would then have multiple threads running this same method which results in the proper value not being correctly stored in this array element. As such, by declaring the array to be volatile, we tell the compiler that the array elements may change which necessarily prevents the compiler from performing reorderings and other optimizations. Thus, Java would work here as the code would not be thrown away while Java would have a race condition due to how the compiler's optimizations mess up our method. +7

b. Neither should be a subtype of the other as volatile simply just tells the compiler that the variable may change. Volatile has no extra operations and as such cannot be considered a subtype of a supertype since all that changes is that the keyword tells the compiler not to perform certain reorderings or optimizations. +8

+4