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

(DIS 1D)

Name: Michael Liu Student ID: 104778500

1	2	3	total
7 to +16 / 37	17	7	61

1 hr

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}  [';'] ']'
```

(expr)
expr, expr
[expr; expr]
expr expr

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.

2

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.

Michael Liu (104778500) pg 1A/4A

CS131: Midterm

(2) 1 a) expr, constant, infix-op, pattern-matching, let-binding, parameter, pattern, INTEGER-LITERAL, STRING-LITERAL, IDENTIFIER

* b) dangling else. if expr1 then if expr2 then expr3 else expr4.
parse tree (6) A → if expr1 then (if expr2 then expr3) else expr4
B → if expr1 then (if expr2 then expr3 else expr4)
both A & B are possible in grammar → ambiguous.

c) expr:
(7) constant
'(' expr ')'
expr expr
expr ',' expr
'[' expr ']'
'[' expr expr-sc ']'
expr infix-op expr
'if' expr 'then' expr
'if' expr 'then' expr 'else' expr
'match' expr 'with' pattern-matching
'function' pattern-matching
'fun' parameter param-list '->' expr
'let' let-binding 'in' expr
'let' 'rec' let-binding 'in' expr

optional ;

expr-sc:
';' expr
| ';' expr expr-sc
| ';' expr ';'

param-list:
empty
parameter (repeat)

empty:

pattern-matching:
pattern '->' expr pat-mat-cont
'|' pattern '->' expr pat-mat-cont

pat-mat-cont:
'|' pattern '->' expr (repeat)
empty

(cont'd) c) let-binding:
pattern '=' expr

parameter:
pattern

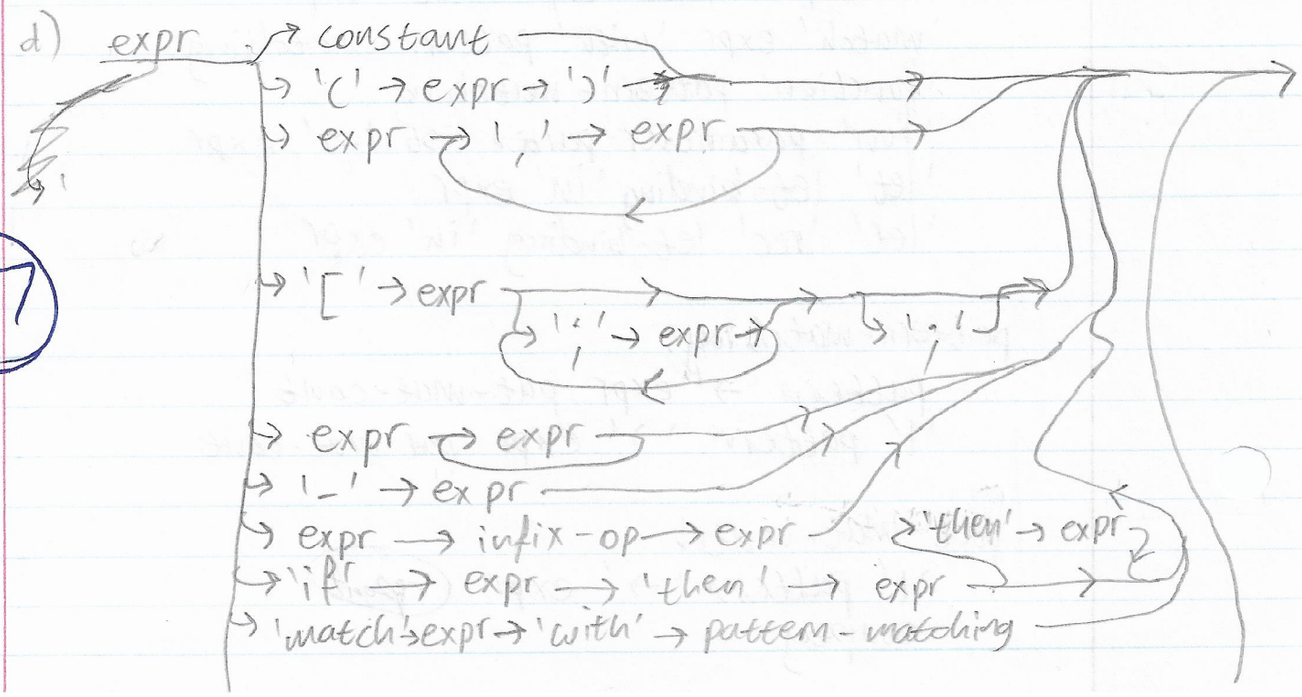
constant: (same)

infix-op: (same)

pattern:
IDENTIFIER
'_'
constant
'(' pattern ')'
pattern ',' pattern
'[' pattern pat-sc-opt ']'
'[' pattern pat-sc-opt ';' ']'

pat-sc-opt:
empty
';' pattern pat-sc-opt

d) expr

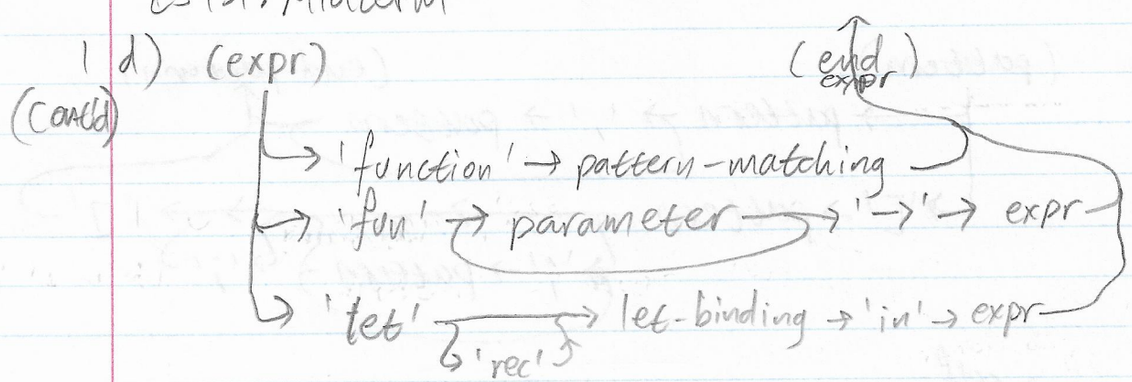


+7

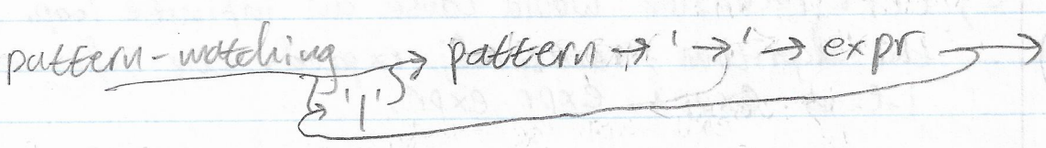
Michael Liu (10477850)

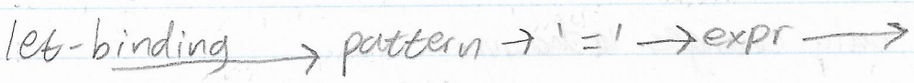
CS 131: Midterm

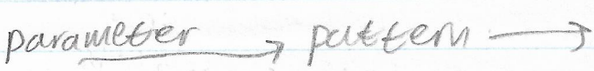
1 d) (expr)



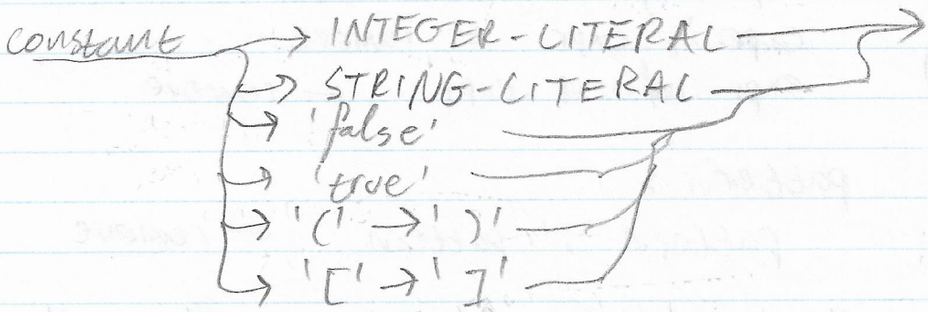
(cont'd)

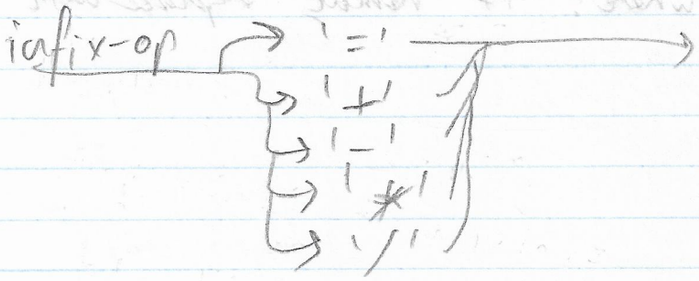


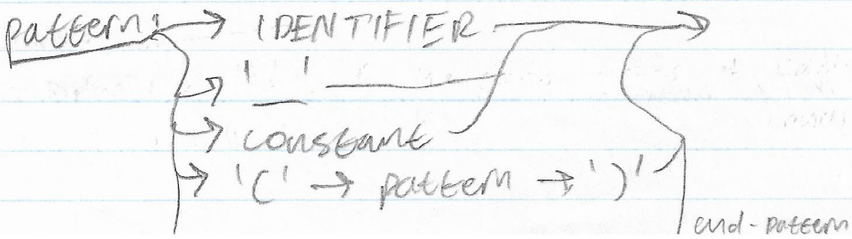




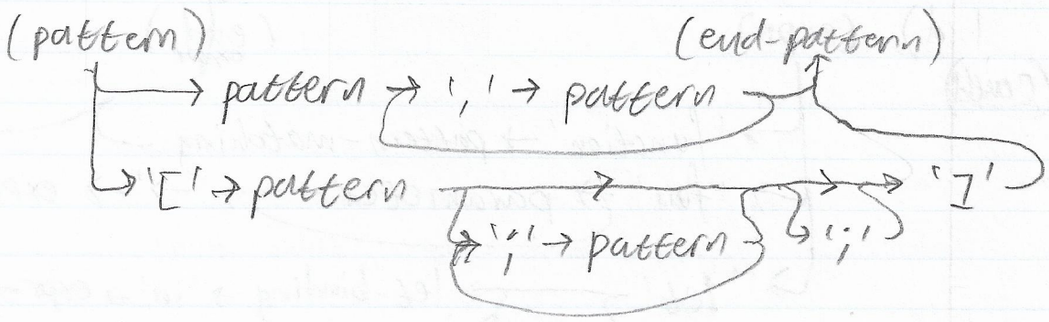
X not simplified







(cont'd) d)



+7

e) Left self-recursion would cause an infinite loop, as the algorithm recursively expands the left side first. i.e. in expr: expr expr.

* f) only changes shown.

migrate instead of remove?

```
expr:
    expr expr          ← remove
    expr ',' expr      ← remove
    expr infix-op expr ← remove

pattern:
    pattern ',' pattern ← remove
```

Migrate where? ~~if~~ remove, replace with what?

Michael Liu (104778500)
CS 131: Midterm

pg3A/4A

1 g) only changes shown.

* any other ambigs beside dang else?

(8)

```
expr:
  'if' expr 'then' expr                          ← remove
  'if' expr 'then' expr 'else' expr              ← remove
  'if' expr 'then' full-stmt 'else' expr
  'if' expr 'then' expr
```

What abt. match, infix?

```
full-stmt:
    full-if
    expr

full-if:
  'if' expr 'then' full-stmt 'else full-stmt
```

2 a) match_empty: ('a → 'b) → 'a → 'b ✓ 1

(11)

match_nothing: 'a → 'b → 'c option ✓

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

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

make_append: (('a → 'b) → 'a → 'b) → 'a list → (('a → 'b) → 'a → '
option -1

make_or: ('a → 'b) → 'a list → ('a → 'b option)
-1

make_matcher: pattern → (('a → 'b option) → 'a → 'b option)
-1

2 b) This means that the behavior of match_nt when given an empty list would depend on whether the acceptor accepts empty lists.

(4)

The user would have to be mindful of the acceptor's behavior in this case, whereas before, he can be confident that match-nt always returns None if given an empty list.

c) match-empty is really the identity matcher.

(2)

Make append would falsely return None ~~sometim~~ when given an empty list to append (it should just return the old matcher). Also, ~~for~~ make_or would now falsely return the old matcher when given an empty list (it should return None).

+1 +1

3 a)

```
int data[volatile] = new Array (10);
void reader () {
    for (int i=0; ; i = (i+1)%10) {
        printf (data[i]);
    }
}

void writer() {
    for (int i=0; ; i=(i+1)%10) {
        int d = receiveDataFromNetwork();
        data[i] = d;
    }
}
```

+4

run one thread per function. JMM states in this case that the volatile ~~means there is no conflict~~. It reads from the memory

b) ~~long[vol] should be a subtype of long[]~~ long[] long[v], ~~because we can perform optimizations on long[] that's not allowed on long[volatile], i.e. Out-of-order execution.~~ Also, casting a long[volatile] into a long[] is always ~~safe, but not~~ vice-versa.

Michael Liu (104778500) Pg 4A/4A

CS 131: Midterm

3 b) long[volatile] should be a subtype of long[], because it's always safe to cast a long[volatile] into a long[]; but not vice-versa (causes undefined behavior).

x3

~~long[] is a subtype of long~~

~~func long fn (long[volatile]) {~~

It is not safe to 'cast' the long[volatile] to long[].

The context (i.e. the program correctness) defines the order of long[] and long[volatile].