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

Name: [redacted]

1	2	3	total
8+1+10 = 28	10	10	48

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'
  '(' ')'
  '[' ']'
```

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.

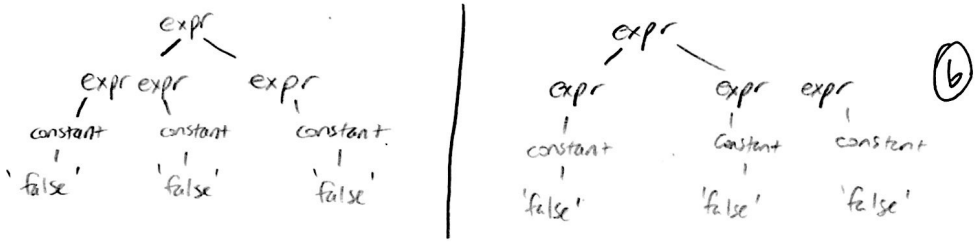
3. Java does not provide a mechanism to declare array elements as being volatile; accesses to array elements are normally considered to be normal accesses. Suppose we define a new language JavaV that is like Java, except that if you declare an array with the keyword 'volatile' immediately after the '['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

a) The nonterminals are expr, constant, pattern, pattern-matching, let-binding, parameter, INTEGER-LITERAL, STRING-LITERAL, infix-op, and IDENTIFIER. (2)

b) It's ambiguous because we can get two parse trees for the same expression.



(6)

The expression 'false' 'false' 'false' gives us these 2 parse trees

c) expr → constant | '(' expr ')' | expr, expr | expr | '[' arr ']' | expr expr | '-' expr | expr infix-op expr | 'if' expr 'then' expr | 'if' expr 'then' expr 'else' expr | 'match' expr 'with' pattern-matching | 'function' pattern-matching | 'fun' parameter-list '->' expr | 'let' 'rec' let-binding 'in' expr | 'let' let-binding 'in' expr (8)

pattern-matching → '|'? (optional) pp

pp → pattern '->' expr | pattern '->' expr pattern-matching

let-binding → pattern '=' expr

parameter → pattern

constant → (* remains the same *)

infix-op → (* remains the same *)

pattern → IDENTIFIER | '_' | constant | '(' pattern ')' | pattern ',' pattern | pattern | '[' pattern-list ']'

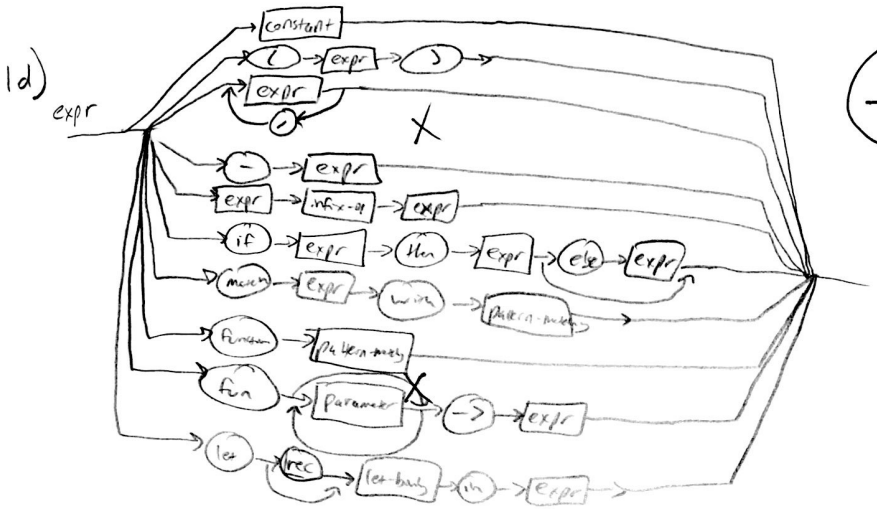
↓ continued on back

arr → expr | expr ';' | expr ';' arr

parameter-list → parameter | parameter parameter-list

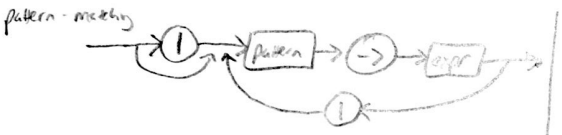
pattern-list → pattern | pattern ';' | pattern ';' pattern-list

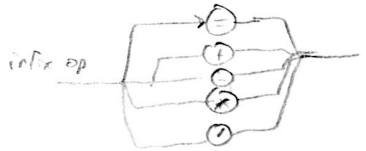
1d)

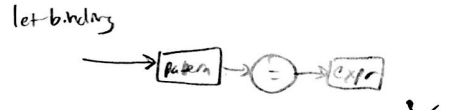


+6

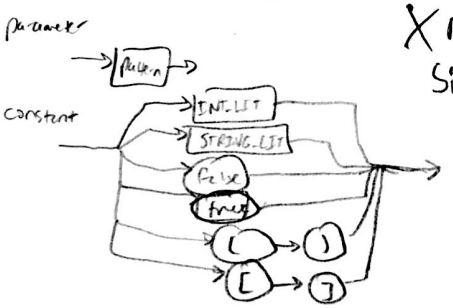
X

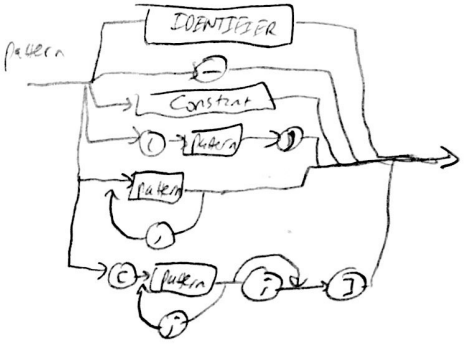






X not Simplified





[redacted] would fail because there are several blind alley rules. For example, expr → expr so when HW2 attempts to go down the parse tree for certain parses, if the correct parse was something below Expr → expr, then we would loop forever, because it would keep exploring the parse tree for expr → expr instead of going down a later tree like '-' expr which is a later option. Also we have undefined non-terminals like IDENTIFIER.

X

+3

1f) We need to remove the rule expr → expr from the grammar and also move expr → expr expr to the last option on the RHS of expr so it only checks that in the last possible case. Assuming our implementation of HW2 cannot match an array of NT values that is longer than the length of our terminal string, this for norms. We need to do a similar thing for pattern → pattern and also define our undefined NT values

This is not required for hw2, so is not a safe assumption

1g) change expr → expr expr into

expr → Lexpr Rexpr

Lexpr → list of expressions ok on left side

Rexpr → list of values ok on right side.

(2)

if-else, match, infix ?

[redacted]

(5)

make_matcher

pattern -> fragment option. ✗

make_or

same as make_append ✗

match_empty

(fragment -> fragment option) -> fragment -> fragment option

too concrete -1

match_nothing

'a -> 'b -> 'c option ✓

match_nt

fragment -> (fragment -> fragment option) -> fragment list -> fragment option -1

append_matchers

((pattern -> pattern option) -> pattern list -> pattern option) -> ((pattern -> pattern option) -> pattern list -> pattern option) -> (fragment -> fragment option) -> fragment -> option ✗

make_append

(pattern -> (pattern -> pattern option) -> pattern list -> pattern option) -> pattern -> fragment option

-1

b) This would allow for the function to match w/ an empty suffix. Before we could not call the accept function on [] but now if the accept function returns true on an empty list we will return that value.

(4)

c) The function (which?) will return None everytime, since everytime we reach the end of our fragment, we will get the matcher match_nothing which always returns None, regardless of input.

(1)

3)

```
int[] a = new int[1000];
```

a)

thread 1	thread 2
`for (int i = 0; i < 500; i++)` `a[2*i] = 1;`	`for (int i = 0; i < 500; i++)` `a[2*i+1] = 2;`

This program should set all even indexed array elements to 1 and all odd indexed to 2. Due to cache blocks being pulled into different cores on this multithreaded application in normal Java, this probably wouldn't succeed due to race conditions. However, in JavaV, since all reads to this array would have to pull in a fresh copy from RAM, we wouldn't have to worry about race conditions.

+4

b) long[] should be a subtype of long[volatile] since long[volatile]'s functionality is a subset of long[]'s functionality. For example, long[] can do everything that long[volatile] can do except it has the added freedom that it can do writes/reads whenever it wants, as opposed to have to wait for locks to synchronize.

+6

+5