UCLA CS 131 Midterm, Fall 2017
100 minutes total, open book, open notes
closed computer

148

Name: Yinxue Xiao Student ID: 904-581-627

1	2	3	4	5	6	total
14	5	7	23	6	7	

1a (10 minutes). Write an OCaml function merge_sorted that merges two sorted lists. Its first argument should be a comparison function 'lt' that compares two list elements and returns true if the first element is less than the second. Its second and third arguments should be the lists to be merged. For example, (merge_sorted (<) [21; 49; 49; 61] [-5; 20; 25; 49; 50; 100]) should yield [-5; 20; 21; 49; 49; 49; 50; 61; 100].

1b (3 minutes). What is the type of merge_sorted?

1c (3 minutes). What does the following expression yield, and what is its type?

```
merge_sorted (fun a b -> List.length a < List.length b)
```

1d (8 minutes). Is your implementation of merge_sorted tail-recursive? If so, briefly say why it won't have any problem with stack overflow. If not, briefly say why not, and explain any problems you would have in rewriting your implementation to make it tail-recursive.

2 (9 minutes). Consider the following top-level OCaml definitions:

```
let f f = f 1 1
let g g = g 0.0 g
let h h = h f "x"
```

For each identifier declared in this code, give the identifier's scope and type. Or, if there is a scope or type error, briefly explain the error.

3a (5 minutes). In Java, is the subtype relation transitive? That is, if A is a subtype of B and B is a subtype of C, is A a subtype of C? If so, explain why; if not, give a counterexample.

3b (5 minutes). In Java, is the graph of the subtype relation a tree? If so, explain why, and say what the root is; if not, give a counterexample.

4. Consider the following grammar for declarations in a subset of C. Ths grammar uses a form of EBNF in which the left hand side is not indented and is followed by ":", each right hand alternative is indented, nonterminals are strings of letters and "-", terminal symbols are either surrounded by single quotes or are INT (meaning an integer constant) or ID (meaning an identifier), and X? stands for zero or one instances of X.

```
declaration:
  declaration-specifiers init-declarator-list? ';'

declaration-specifiers:
  storage-class-specifier declaration-specifiers?
  type-specifier declaration-specifiers?
  type-qualifier declaration-specifiers?
  function-specifier declaration-specifiers?

storage-class-specifier:
  'typedef'
  'static'

type-qualifier:
  'const'
  'volatile'

type-specifier:
  'void'
  'char'
  'int'

function-specifier:
  'inline'
  '_Noreturn'

init-declarator-list:
  init-declarator
  init-declarator-list ',' init-declarator

init-declarator:
  declarator
  declarator '=' initializer

declarator:
  pointer? direct-declarator

direct-declarator:
  ID
  '(' declarator ')'
  direct-declarator '[' INT ']'
  direct-declarator '(' 'void' ')'

pointer:
  '*' type-qualifier-list? pointer?

type-qualifier-list:
  type-qualifier-list? type-qualifier

initializer:
  ID
  INT
```

4a (2 minutes). What makes this grammar EBNF and not simply BNF?

4b (8 minutes). Give an example declaration that is syntactically correct (i.e., it is produced by this grammar) but is semantically incorrect for C. Prove that it is syntactically correct. Briefly explain why it is semantically incorrect.

4c (5 minutes). Suppose we changed the grammar by replacing the ruleset for type-qualifier-list with the following:

```
type-qualifier-list:
  type-qualifier type-qualifier-list?
```

Would this cause any problems? If so, describe a problem and give an example. If not, briefly explain why not.

4d (10 minutes). Suppose we changed the original grammar by replacing the two rulesets for declarator and direct-declarator with the following single ruleset:

```
declarator:
  pointer? declarator
  ID
  '(' declarator ')'
  declarator '[' INT ']'
  declarator '(' 'void' ')'
```

Would this cause any problems? If so, describe a problem and give an example. If not, briefly explain why not.

4e (10 minutes). Draw a syntax chart for the original grammar.

5 (10 minutes). Suppose we write Java code in a purely functional style, in that we never assign to any variables except when initializing them. That is, we always initialize local variables and never assign to them later, and we always initialize instance variables once at the start of constructors and never assign to them later.

In our purely-functional Java programs, is the Java Memory Model still relevant, or can we ignore it? If it's still relevant, explain which parts of it still apply and give an example. If not, briefly explain why not.

6 (12 minutes). Consider the following code, taken from the answer to the older version of Homework 2.

```
let match_empty frag accept = accept frag

let match_nothing frag accept = None

let rec match_star matcher frag accept =
  match accept frag with
    | None ->
        matcher frag
                (fun frag1 ->
                   if frag == frag1
                   then None
                   else match_star matcher frag1 accept)
    | ok -> ok

let match_nucleotide nt frag accept =
  match frag with
    | [] -> None
    | n::tail -> if n == nt then accept tail else None

let append_matchers matcher1 matcher2 frag accept =
  matcher1 frag (fun frag1 -> matcher2 frag1 accept)

let make_appended_matchers make_a_matcher ls =
  let rec mams = function
    | [] -> match_empty
    | head::tail -> append_matchers (make_a_matcher head) (mams tail)
  in mams ls
```

In this code, a matcher is a curried function taking two arguments: first, a fragment 'frag' and second, an acceptor 'accept'. Suppose we change the API for matchers by interchanging their arguments, so that the acceptor comes first (all the functions remain curried). Rewrite the above code to use the altered API, and simplify the resulting code as much as possible.

Yinxue Xiao
904-581-627

CS131 Midterm

1a. let merge_sorted lt list1 list2 = match list1 with
| [] -> list2
| x::xs -> match list2 with
| [] -> list1
| y::ys -> if (lt x y) then x :: merge_sorted (xs, y::ys)
else y :: merge_sorted (x::xs, ys)

1b. if we assume the list inputs are all int lists, then it's
(int -> int -> bool) -> int list -> int list -> int list.
if it's any type, then it's
('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list

1c. It takes 2 sorted list, a and b. If a's length is less than b's length, the merged list will be [a's list element; b's list elements] vice versa.
int list -> int list -> int list
(or if unknown type: 'a list -> 'a list -> 'a list)

2) ① ② ③
2. let f f = f 1 1

① first f is the function's definition (int -> int -> 'a) -> (int -> int -> 'a)
② second f is in the scope of the funcion
③ third f is in the scope of the function's parameter. int -> int -> 'a

⇓ but the second f doesn't take any arguments

2) ① ② ③ ④
let g g = g 0.0 g

① first g is function's definition
('float -> 'g -> 'a) -> (float -> 'g -> 'a) error

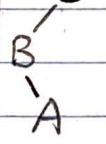
② second g is in the scope of the function

③④ third & fourth g is in the scope of function's parameter.
float -> 'g -> 'a

3). let h h = h f "x"
① ② ③

① g is function definition ('a -> string -> 'b) -> ('a -> string -> 'b)
② g is in the scope of the function
③ g is in the scope of function's parameter.
'a -> string -> 'b

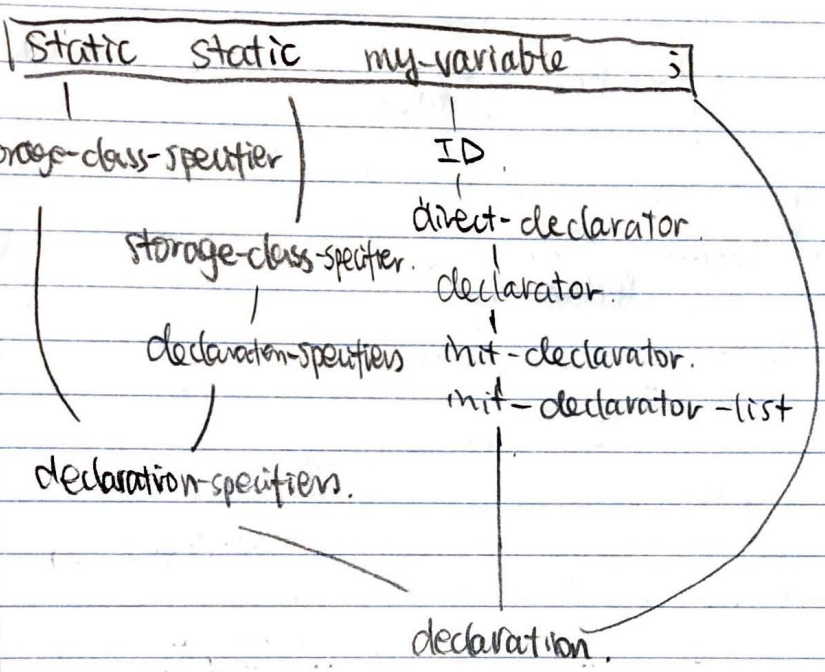
3a Yes it is. If A is a subtype of B, it supports operations of supertype B. Similarly, if B is subtype of C, it supports operations of supertype C. Thus, A should be able to support operations of its supertype C, and be a subtype of C. Another way to think about this is ∵ B is subtype of C, B has all features of C, if A is subtype of B, A has all features of B. Thus, A has all features of C, and thus a subtype of C.

3b


Yes it is. Because a subclass is a subtype, and one class can't inherit from more than one superclass, so it can have at most one superclass. While a superclass can have multiple subclass, it can have multiple subtype. Thus it works like a tree. The superclass acts like a root and branches out for the subtypes. It has to be a tree because there are no cycles due to the fact that one subclass can only have one superclass.

4a the "?" symbol which stands for zero or one instance.
2

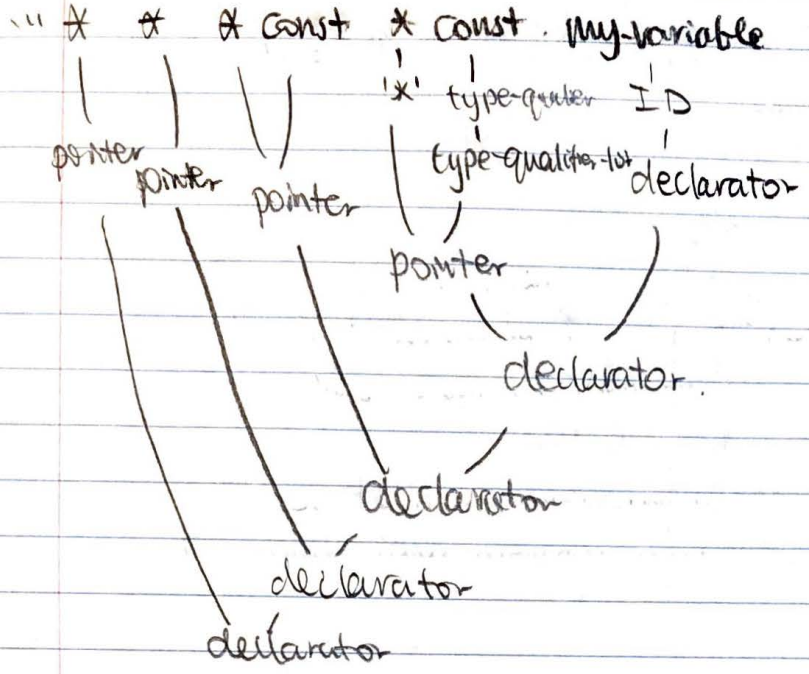
4b
8



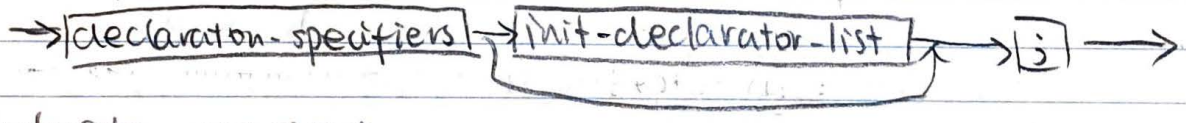
it's syntactically correct as analyzed above.
but it's semantically incorrect since we can't declare static twice in c. It doesn't make sense.

4c This changes the type-qualifier-list from left recursive to right recursive. It wouldn't cause any problems. Before it would be volatile const, now it would be volatile constant.
4 not true. -1
If there's a problem that has too many repeated words, both of them would have the same problem.

4d Yes it will. Because "declarator: pointer? declarator", it would just recurse on itself, thus adding infinite number of pointers to itself. For example, lets consider the case: ****const.
4 The old way we did it wouldn't cause this problem (next page) since it separates it out using a direct-declarator.

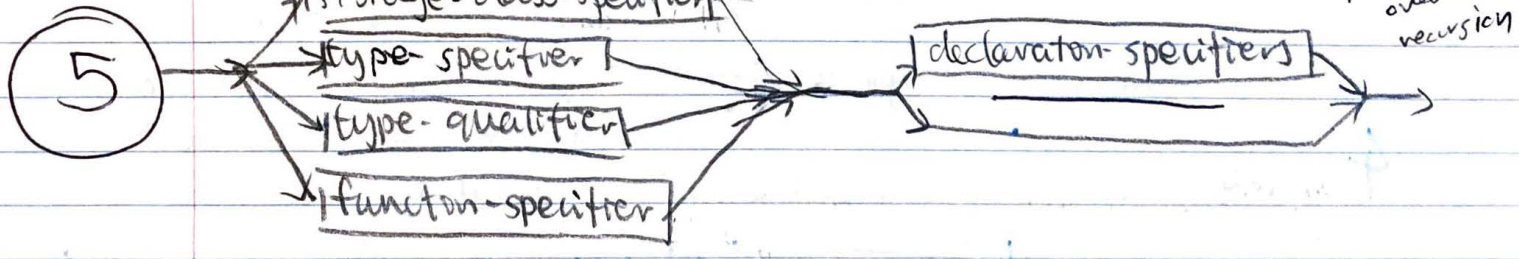


4e declaration.

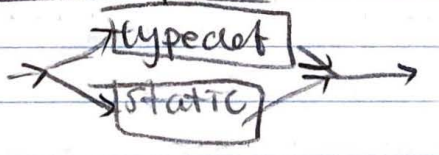


-1 could have condensed diagram

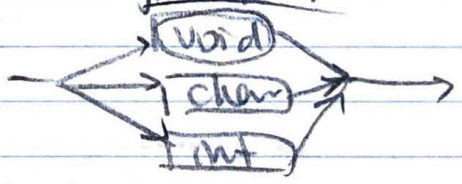
declaration-specifiers



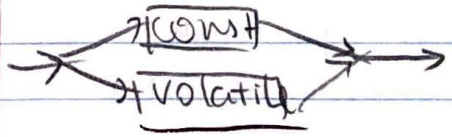
storage-class-specifier



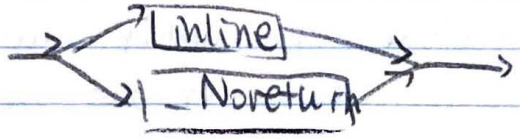
type-specifier



type-qualifier



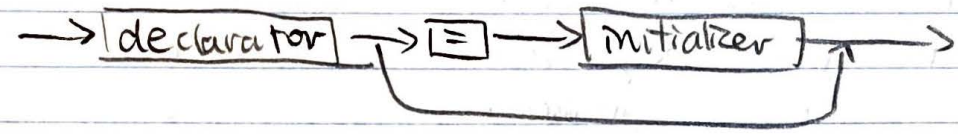
function specifier



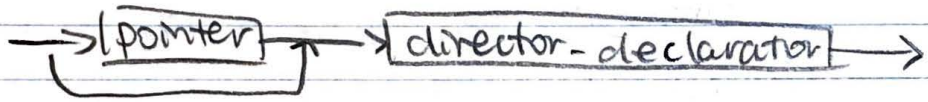
init-declarator-list :

-1 doesn't need recursion

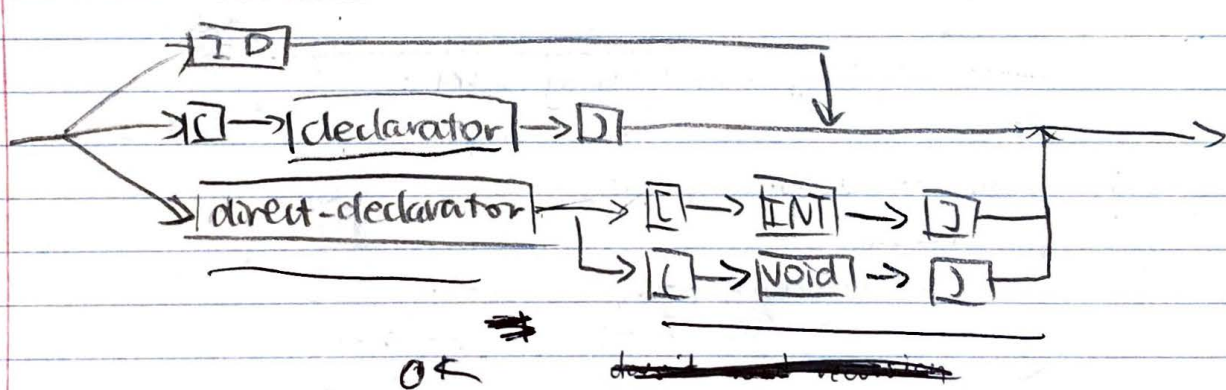
init-declarator



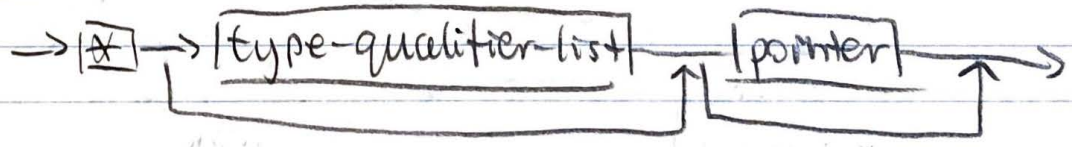
declarator



direct-declarator

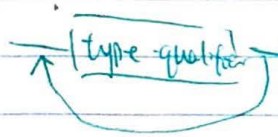


pointer



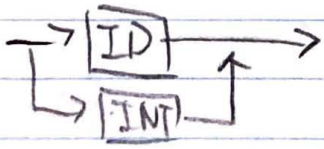
type-qualifier-list

-1 "



-1 repetition or recursion

initializer



5 Consider the following case:

T1
```
bool a = true
...
a = false
```

T2
```
while (a == true) {
...
}
```

(cont'd) if java is purely functional, we cannot change its value later on in a thread. Thus, it won't affect another thread's running. For the java memory model, volatile keeps the variables updated since it puts them in memory and access it from memory. However, since we can't change a variable's value after initialization, we will be certain that the variable is up to date, so we don't need java memory model anymore.

6
```
let match_empty accept frag = accept frag
let match_nothing accept frag = None
let rec match_star matcher accept frag =
    match accept frag with
    | None ->
        matcher accept frag
            (fun accept1 ->
                if accept1 == accept
                then None
                else match_star matcher accept1 frag)
    | ok -> ok
let match_nucleotide nt accept frag = match frag with
    | [] -> None
    | n :: tail -> if n == nt then accept tail else None
let append_matchers matcher1 matcher2 accept frag =
    matcher1 (fun frag1 -> matcher2 accept frag1) frag
let make_appended_matchers make_a_matcher ls =
    let rec mams = function
        | [] -> match_empty
        | head :: tail -> append_matchers (make_a_matcher head) (mams tail)
    in mams ls
```

7