CS 131

Saketh Kasibatla

Today

Questions on HW6

Questions on Final

Sample Final

This Class

Last day to submit homework 6 late is **Today**

Deadline to evaluate instructors is this Saturday

HW6 Questions

Final

Final is Friday, Dec. 16 from 3PM to 6PM

Same room as lecture

Open book, open notes

- bring everything
- print out homeworks, solutions
- notes from class, sections
- textbook
- anything non-electronic

Final

Focused on content from after midterm

- 60% after, 40% before

there will be questions about the homework

there will be questions about topics discussed in lecture

Questions about the Final

Sample Final Question 1

Consider the following OCaml program.

```
let s x = x + 1

let rec i min max =
  if max < min
  then []
  else min :: i (s min) max

let rec f p = function
  | [] -> []
  | a::r -> let fpr = f p r in
    if p a then a::fpr else fpr

let x n =
  f (fun m -> not (m mod n = 0))
```

```
let t max =
  let rec f1 = function
    | [] -> []
    | n::r ->
      n :: (if max < n*n
        then r
        else f1 (x n r))
  in
    f1 (i 2 max)
```

what are the types of the top level identifiers `s`, `i`, `f`, `x`, `t`?

what does the expression "`t 20`" evaluate to?

Sample Final Question 2

The following statement is one of the disadvantages of event driven programming:

“As each callback must be finished as soon as possible, it is not possible to keep persistent state in function-local variables”

What does this remark mean? Give an example of why it might be nice to keep persistent state in a function-local variable, and describe what would happen to an asyncio server if you tried to do that.

Sample Final Question 3

Consider the following argument:

The programmers of the world are still using the first major imperative language (Fortran), the first major functional programming language (Lisp), and the first major logic-programming language (Prolog), but they no longer use the first major object-oriented programming language (Simula 67). That is because object-oriented languages are fundamentally different. Here’s why…

If you agree with this argument, finish it by adding some justifications and examples. If not, argue the contrary.

Sample Final Question 4

Consider the static-scoping semantics given in pages 512-513 of Webber.

a) Can these semantics be implemented easily on a machine in which all storage is allocated on the stack, or do they assume the existence of a heap? If the former, give a nontrivial example and show why the heap is not needed for it; if the latter, give an example of why the heap is necessary.

b) How about with dynamic-scoping semantics?

Sample Final Question 5

Generally speaking, languages that use call-by-name or call-by-need parameter-passing conventions avoid hangs and/or crashes that would occur if the same program were evaluated using call-by-value.

Can you think of a program that has well-defined behavior with call-by-value, but which hangs with call-by-name or call-by-need? If so, give a counterexample; if not, explain why it's not possible to give a counterexample

Sample Final Question 5

Call by name is an evaluation strategy where the arguments to a function are not evaluated before the function is called— they are substituted directly into the function body (using capture-avoiding substitution) and then left to be evaluated whenever they appear in the function

Call by need is a memoized version of call by name where, if the function argument is evaluated, that value is stored for subsequent uses. In a "pure" (effect-free) setting, this produces the same results as call by name; when the function argument is used two or more times, call by need is almost always faster. Haskell is the best known language that uses call by need evaluation.

Call by value (also referred to as pass by value) is the most common evaluation strategy, used in languages as different as C and Scheme. In call by value, the argument expression is evaluated, and the resulting value is bound to the corresponding variable in the function

Sample Final Question 6

Write implementations for the following Prolog Predicates

shift_left(L, R) succeeds if R is the result of "shifting left" the list L by 1. The leading element of L is lost. For example, shift_left([a,b,c], [b,c]).

shift_right(L, R) is similar, except it shifts right. For example, shift_right([a,b,c], [a,b]).

shift_left_circular(L, R) is like shift_left, except the leading element of L is reintroduced at the right. For example, shift_left_circular([a,b,c], [b,c,a]).

shift_right_circular(L, R) is similar, except it shifts right. For example, shift_right_circular([a,b,c], [c,a,b]).

Which of your implementation of the predicates might be nondeterministic? For each such predicate, give an example call that succeeds more than once.

Sample Final Question 7

Consider the following abstract syntax for scheme in the R5RS spec.

This syntax lists only constants, identifiers, procedure calls, lambda, if, and set!.
The expressions to the right aren't any of these things, so why are they valid expressions?
If they aren't valid, say why not

```
K ∈ Constant
I ∈ Identifier
E ∈ Expression
Γ ∈ Command = Expression
Exp → K | I | (E E*)
  | (lambda (I*) Γ* E)
  | (lambda (I* . I) Γ* E)
  | (lambda I Γ* E)
  | (if E E E) | (if E E)
  | (set! I E)
```

```
(let ((a '())) a)

(cond ((eq? '() '()) 1) (#t 0))

(1)
```

Sample Final Question 8

For each of the following seemingly-arbitrary language rules, explain what performance gains (if any) arise because of the rules. Analyze both time and space performance. List the rules roughly in decreasing order of importance, performance-wise.

Arrays are origin-zero, not origin 1 .

Arrays must be allocated statically or on the stack; they cannot be allocated on the heap.

Each object in an array must have the same size.

The array must have just one dimension

Subscript violations result in undefined behavior; an exception may or may not be raised.

The number of elements in an array is fixed at compile-time.

Arrays must always be accessed directly; there are no pointers to arrays, and arrays cannot be passed by reference to procedures.

Sample Final Question 9

Suppose we add a new statement "return[N]EXPR;" to our C implementation, as an extension to C.

N must be an integer constant expression with a nonnegative value, and

EXPR can be any valid C expression.

This new statement acts like "return EXPR;", except that it causes the Nth caller of the current function to return with the value of the expression EXPR.

For example, suppose F calls G which calls H, and suppose H then executes "return[2]5;".

This causes F to return immediately, with the value 5; G and H are silently exited in the process.

By definition, "return[0]EXPR;" is equivalent to "return EXPR;".

Sample Final Question 9

What problems do you see in implementing "return[N]EXPR;"?

Are these problems inherent to C, or can they be worked around by changing the semantics of the new statement slightly? Would it be easier to implement this new statement in some of the other languages that we have studied?

If so, which ones and why? If not, why not?