CSE 131 Spring 2017 – Final Exam 1

Name:

PID:

Email:

Only answers on this sheet will be graded.
Per-question points are in italics. There are 125 total points, plus 10 extra credit points.
Please also write your PID on each page of the exam, in case pages get separated.

1. *(2)*	16. *(2)*
2. *(2)*	17. *(3)*
3. *(2)*	18. *(4)*
4. *(3)*	19. *(4)*
5. *(3)*	20. *(12)*
6. *(3)*	21. *(10)*
7. *(6)*	22. *(8)*
8. *(6)*	23. *(4)*
9. *(6)*	24. *(5)*
10. *(3)*	25. *(4)*
11. *(3)*	26. *(10)*
12. *(3)*	27. *(3)*
13. *(4)*	28. *(3)*
14. *(2)*	29. *(3)*
15. *(2)*	**EC** *(10)*

If an answer says "Choose ALL that apply" put all the letters that you choose, one after another, in the box, like `ABD`. If the answer is "None of the above," make sure to answer the appropriate letter for that response.

"Choose ALL that apply" questions grant partial credit (e.g. you get points for choosing correct answers and for *not* choosing incorrect ones).

If an answer says "Enter your answer directly in the answer key," follow the instructions in the question for the answer. You might write a single number, a fragment of code, or a variable name.

This exam is largely structured as a walkthrough of a compiler similar to the snake languages we used in homeworks, asking questions along the way.

There is a lot of OCaml code provided in the exam. There aren't any "gotcha"s in the OCaml code that you need to find. The code should look familiar, and is intended to work in conjunction with your experience to help you answer the questions.

There are a few global assumptions you can make in this exam. Like the compiler code, these are not intended to be tricky; they are intended to make questions clear, and re-use knowledge you have from the programming assignments efficiently.

- The start of the heap for generated programs, and the initial value in `EBX` at the start of a program, is always `0x100`
- The start of the stack for generated programs, and the initial value in `ESP` at the start of a program, is always `0x2000`
- All instructions are aligned at 4-byte addresses, so any reference to an instruction will always end in 0, 4, 8, or c (in hexadecimal).
- Don't worry about the presence or absence of size modifiers like `DWORD` – assume all `mov` instructions always move 4 bytes.
- Registers and memory locations are used for the same purposes as in the compilers we wrote this quarter – `EAX` holds the most recent "answer", `ESP` tracks the current stack frame, `EBX` holds the heap pointer, and by default values take up 4 bytes (either stored in a register or somewhere in memory). Other registers may be used for storing temporary values in some questions.
- The exam builds up a compiler from mostly scratch, and asks you questions about it. Any code that starts with line numbers, like:

```
1  let this_is_compiler_code = "yup!" in
2  let meaning_an_ocaml_implementation = true in
3  printf "OK!\n"
```

  is part of the running OCaml implementation of the compiler. Other code examples will be either assembly (which is obvious by its format), or programs in the language we're compiling.

- When writing out value in memory frames, we put `----` for words that are irrelevant and not set by the program. For values that have a meaningful decimal interpretation, we write them as, e.g.

  `0x0000000c (12)`

  For values that represent addresses, we dash out the decimal part to indicate that the decimal interpretation is irrelevant, e.g.

  `0x00002000 (--)`

Consider the following expression language, which we'll call **Indigo**:

```
type expr =
  | EBox of expr
  | EGetBox of expr
  | ESetBox of expr * expr
  | EBegin of expr list
  | EApp of expr * expr
  | EFun of string * expr
  | ENum of int
  | EPlus of expr * expr
  | EMinus of expr * expr
  | ELet of string * expr * expr
  | EId of string
```

With the following concrete syntax:

```
expr := <number>                  # ENum
      | <id>                      # EId
      | box <expr>                # EBox
      | *<expr> = <expr>          # ESetBox
      | *<expr>                   # EGetBox
      | begin <expr>; ... end     # EBegin
      | <expr>(<expr>)            # EApp
      | fun <id>: <expr> end      # EFun
      | <expr> + <expr>           # EPlus
      | <expr> - <expr>           # EMinus
      | let x = <expr> in <expr>  # ELet
```

There are two features in this language that are *like* features we saw in class and PAs, but not quite the same.

1. **Boxes**, which are like single-element, mutable tuples. In fact, that description captures the idea of a *pointer*, which is also a useful analogy.
2. **First-class functions**, which are defined with `fun` and act like closures that don't store any free variables, and take only a single argument.

This exam will introduce these features by example, and you will answer questions relating to the implementation.

First, we need to make sure we understand how the concrete and abstract syntax correspond. Here are some examples to get us started:

Concrete Syntax	**Abstract Syntax**
`14`	`ENum(14)`
`14 + x`	`EPlus(ENum(14), EId("x"))`
`fun x:  1 end`	`EFun("x", ENum(1))`
`let f = fun x:  1 end in f(10)`	`ELet("f", EFun("x", ENum(1)), EApp(EId("f"), ENum(10)))`
`begin box 5; box 10 end`	`EBegin([EBox(ENum(5)); EBox(ENum(10))])`

`let b = box 10 in`
`begin`
`*b = 11;`
`*b;`
`end`
	`ELet("b", EBox(ENum(10)),`
	`EBegin([`
	`ESetBox(EId("b"),ENum(11));`
	`EGetBox(EId("b"));`
	`]))`

What concrete program corresponds to the following `expr`s? Enter your answer directly in the answer sheet:

1. `ELet("x", EId("y"), EPlus(EId("x", "y")))`
2. `EFun("y", EBox(EId("y")))`
3. `ESetBox(EId("b"), EGetBox(EId("b2")))`

   Fill in the blanks in the following `expr`s for the given concrete syntax:

4. Concrete program:

   ```
   let x = 10 in
   let y = x in
   x + y
   ```

   The corresponding `expr`:

   ```
   ELet("x", ENum(10),
     ELet("y", __________,
     EPlus(EId("x"), EId("y"))))
   ```

   Choose from:

   (a) `"x"`
   (b) `EId("x")`
   (c) `ENum(10)`
   (d) `10`
   (e) None of the above

5. Concrete program:

```
let b = box 10 in
begin
  *b = 15;
  box *b
end
```

The corresponding `expr`:

```
ELet("b", EBox(ENum(10)),
  EBegin([
    ESetBox(EId("b"), ENum(15));
    ______________
  ])
```

Choose from

(a) `EGetBox(EBox(EId("b")))`

(b) `EBox(EGetBox(EId("b")))`

(c) `EBox(EId("b"))`

(d) `EGetBox(EId("b"))`

(e) None of the above

6. Concrete program:

```
let f = fun y: 1 + y end in
f(f(10))
```

The corresponding `expr`:

```
ELet("f", EFun("y", EPlus(ENum(1), EId("y")))
_____________________)
```

Choose from

(a) `EApp(EId("f"), EApp(EId("f"), ENum(10)))`

(b) `EApp("f", EApp("f", ENum(10)))`

(c) `EApp(EId("f"), EId("f"), ENum(10))`

(d) `EApp("f", EId("f"), ENum(10))`

(e) None of the above

Consider the following implementation of well-formedness. Note that there is a hole in the `EFun` case, which is the only non-standard case compared to the other snake languages:

```
let rec well_formed_e (e : expr) (env : bool envt) =
  match e with
    | ENum(_) -> []
    | EId(x) ->
      begin match find env x with
        | None -> ["Unbound identifier: " ^ x]
        | Some(_) -> []
      end
    | EApp(f, arg) ->
      (well_formed_e f env) @ (well_formed_e arg env)
    | ELet(x, bind, body) ->
      (well_formed_e bind env) @ (well_formed_e body ((x, true)::env))
    | EBox(e) -> (well_formed_e e env)
    | EGetBox(e) -> (well_formed_e e env)
    | EPlus(e1, e2)
    | EMinus(e1, e2)
    | ESetBox(e1, e2) -> (well_formed_e e1 env) @ (well_formed_e e2 env)
    | EBegin(es) -> List.flatten (List.map (fun e -> well_formed_e e env) es)
    | EFun(arg, body) -> (well_formed_e body _____________)
and find ls x =
  match ls with
    | [] -> None
    | (y,v)::rest ->
      if y = x then Some(v) else find rest x
```

By "report a well-formedness error" below, we mean that a call to `well_formed_e` would return a non-empty list. Consider these programs:

(a) `let x = 10 in fun y:  x + y end`

(b) `let x = 10 in fun y:  x end`

(c) `let x = 10 in fun y:  10 end`

(d) `let x = 10 in fun y:  y end`

(e) `let f = fun y:  f + y end`

(f) `fun y:  let x = 10 in x + y end`

7. If we chose `[(arg, true)]` to fill in the blank in the program, which of these programs would report well-formedness errors? Choose ALL that apply.

8. If we chose `(arg, true)::env` to fill in the blank in the program, which of these programs would report well-formedness errors? Choose ALL that apply.

9. If we chose `env` to fill in the blank in the program, which of these programs would report well-formedness errors? Choose ALL that apply.

There are three kinds of values. Let's start with the expressions that generate those:

```
let count = ref 0
let gen_temp base =
  count := !count + 1; sprintf "%s%d" base !count
let rec compile_expr (e : expr) (env : (string * int) list) (si : int) : instr list =
  match e with
    | ENum(n) -> [IMov(Reg(EAX), Const(n * 2))]
    | EBox(e) -> (compile_expr e env si) @
      [IMov(RegOffset(EBX, 0), Sized(DWORD_PTR, Const(0)));
       IMov(RegOffset(EBX, 4), Reg(EAX));
       IMov(Reg(EAX), Reg(EBX));
       IAdd(Reg(EBX), Const(8));
       IAdd(Reg(EAX), 1)]
    | EFun(arg, body) ->
      let body_is = compile_expr body ((arg, 1)::env) 2 in
      let jump_to = gen_temp "after" in
      let name = gen_temp "function" in
      [IJmp(Label(jump_to));
       ILabel(name)] @ body_is @
      [IRet;
       ILabel(jump_to);
       IMov(Reg(EAX), Label(name));
       IAdd(Reg(EAX), Const(3))]
```

Assuming the compiler above, what code would be generated for the following examples, given both in concrete and abstract syntax? Assume for these examples that `env` is the empty list, and `si` is `0`.

10. `5` `ENum(5)`

    Choose the letter that corresponds to the generated code and enter it in the answer sheet:

    (a) `mov eax, 5`
    (b) `mov eax, 10`
    (c) `mov [eax], 10`
    (d) `mov eax, 11`

11. `box 10` `EBox(ENum(10))`

    Fill in the blank in the following generated code (write the answer directly in the answer sheet):

```
mov eax, 20
mov [ebx+0], DWORD 0
mov [ebx+4], eax
mov eax, ebx
add ebx, __________
add eax, 1
```

12. `fun x:  6 end` `EFun("x", ENum(6))`

    Fill in the two blanks in the following generated code (write BOTH answers in the answer cell separated by a comma):

```
jmp after1
function2:
mov eax, ________
ret
after1:
mov eax, ________
add eax, 3
```

13. `box fun z:  31 end` `EBox(EFun("z", ENum(31)))`

    Choose the letter that corresponds to the correct answer:

    (a)
    ```
    jmp after1
    function2:
    mov eax, 62
    ret
    after1:
    mov eax, function2
    add eax, 3
    mov [ebx+0], 0
    mov [ebx+4], eax
    mov eax, ebx
    add ebx, 8
    add eax, 1
    ```

    (b)
    ```
    mov [ebx+0], 0
    mov [ebx+4], eax
    mov eax, ebx
    add ebx, 8
    add eax, 1
    jmp after1
    function2:
    mov eax, 62
    ret
    after1:
    mov eax, function2
    add eax, 3
    ```

    (c)
    ```
    jmp after1 mov [ebx+0], 0
    mov [ebx+4], eax
    mov eax, ebx
    add ebx, 8
    add eax, 1
    function2:
    mov eax, 62
    ret
    after1:
    mov eax, function2
    add eax, 3
    ```

    For each of the following questions, enter your answer as two binary digits, or NA if the answer won't always be the same two bits.

14. Assuming each function label is aligned on a 4-byte boundary, what will the two least significant bits of each function value be?

15. Assuming the value of `EBX` starts at 4-byte boundary, what will the two least significant bits of each box value be?

16. What will the two least significant bits of each number value be?

Next, let's consider simple arithmetic and variable bindings. In Indigo, they will work much the same as in the snake languages we used throughout the quarter.

```
let stackloc i      = RegOffset (-4 * i, ESP)
let save_to_stack i = [ IMov (stackloc i, Reg(EAX)) ]
let restore_stack i = [ IMov (Reg(EAX), stackloc i) ]

let rec compile_expr e env si =
  (* ... as before for ENum, EBox, EFun ... *)
  | EPlus(e1, e2) -> compile_op "plus" e1 e2 env si
  | EId(x) -> restore_stack (direct_find env x)
  | EMinus(e1, e2) -> compile_op "minus" e1 e2 env si
  | EPlus(e1, e2) -> compile_op "plus" e1 e2 env si
  | ELet(x, e, body) ->
    let eis = compile_expr e env si in
    eis @ (save_to_stack si) @ (compile_expr body ((x, si)::env) (si + 1))
and direct_find env x =
  match env with
    | [] -> failwith "Unbound id"
    | (y, i)::rest ->
      if x = y then i else find rest x
and compile_op op e1 e2 env si =
  let rhs_loc = stackloc (si+1) in
  let e1_is   = compile_expr e1 si     env @ save_to_stack si     in
  let e2_is   = compile_expr e2 (si+1) env @ save_to_stack (si+1) in
  let op_is   = match o with
    | "plus"  -> [ IAdd (Reg(EAX), rhs_loc) ]
    | "minus" -> [ ISub (Reg(EAX), rhs_loc) ]
```

For the next few questions, assume the initial value in `ESP` is `0x2000`, the starting environment is the empty list, and the starting stack index is `0`.

17. Consider compiling and running the generated instructions for the following program:

```
let x = 5 in
1 + x
```

Fill in the blank below for the value stored at `0x2000`. Answer as a decimal number (the format in parentheses below).

`0x1ff8`	0x0000000a (10)
`0x1ffc`	0x00000002 (02)
`0x2000`	**ANSWER HERE**

18. Consider compiling and running the generated instructions for the following program:

```
let x = 5 in
let y = 6 in
(x + y) - 13
```

Assuming the initial value in `ESP` is `0x2000`, the starting environment is the empty list, and the starting stack index is `0`, what would memory look like after the instructions for this expression are executed? Assume that a dashed-out word could be any value and is unused.

(a)

`0x1ff0`	0x0000001a (26)
`0x1ff4`	0x0000000c (12)
`0x1ff8`	0x0000000a (10)
`0x1ffc`	0x0000000c (12)
`0x2000`	0x0000000a (10)

(b)

`0x1ff0`	————
`0x1ff4`	0x0000001a (26)
`0x1ff8`	0x00000016 (22)
`0x1ffc`	0x0000000c (12)
`0x2000`	0x0000000a (10)

(c)

`0x1ff0`	————
`0x1ff4`	————
`0x1ff8`	0x0000001a (26)
`0x1ffc`	0x0000000c (12)
`0x2000`	0x0000000a (10)

(d)

`0x1ff0`	0x00000012 (18)
`0x1ff4`	0x00000010 (16)
`0x1ff8`	0x00000022 (34)
`0x1ffc`	0x0000000c (12)
`0x2000`	0x00000016 (22)

19. If we replaced `(x, si)::env` above with `(x, si + 1)::env`, which of the following problems could result? Choose ALL that apply.

    (a) The generated code would overwrite existing values on the stack that shouldn't be overwritten

    (b) The generated code would look up variables in the wrong locations

    (c) The `failwith` for "Unbound id" could occur in new situations

For the next few questions, assume that the starting value in the register `EBX` is `0x100` at the start of running these instructions, and the starting value in `ESP` is `0x2000`.

20. Consider this layout on the *heap*:

0x100	0x00000000 (00)
0x104	0x00000020 (32)
0x108	0x00000000 (00)
0x10c	0x00000101 (--)
0x120	0x00000000 (00)
0x124	0x00000101 (--)

Which of the following programs would produce this heap? Choose ALL that apply.

(a) `let x = 16 in let b = box x in let b2 = box b in box b`

(b) `let x = 16 in let b = box x in let b2 = box b in box x`

(c) `box box 16`

(d) `let b = box 16 in let b2 = box b in let b3 = box b in b3`

(e) `let b = box 16 in let b2 = b in box b2`

(f) `let b = box 16 in box 32`

21. Consider the following layout on the *stack*:

0x1ff8	0x00000105 (--)
0x1ffc	0x00000101 (--)
0x2000	0x00000101 (--)

Which of the following programs would produce this stack? Choose ALL that apply.

(a) `let x = box 4 in let y = x in let z = y in z`

(b) `let x = box 4 in let y = x in let z = box 3 in z`

(c) `let x = box 4 in let y = box 3 in let z = y in z`

(d) `let x = box 4 in let y = x in let z = box 10 in z`

(e) `let x = box 4 in let y = box 3 in let z = x in z`

Consider defining the calling convention for invoking functions defined with `fun`, by adding the following case for `EApp` in the compiler:

```
1  | EApp(f, arg) ->
2    let return = gen_temp "after_call" in
3    let fis = compile_expr f env si in
4    let argis = compile_expr arg env si in
5    fis @
6    [IMov(Reg(ECX), Reg(EAX))] @
7    argis @
8    [
9      IMov(RegOffset((si + 1) * -4, ESP), Sized(DWORD_PTR, Label(return)));
10     IMov(RegOffset((si + 2) * -4, ESP), Reg(EAX));
11     ISub(Reg(ESP), Const((si + 1) * 4));
12     ISub(Reg(ECX), Const(3));
13     IJmp(Reg(ECX));
14     ILabel(return);
15     IAdd(Reg(ESP), Const(si * 4))
16   ]
```

Reminder: We can use an instruction like

`jmp ecx`

to jump the instruction pointer to the address stored in that register. This is different than jumping to a label directly, since the address can be computed and stored in the register while the program is running, and isn't fixed at compile time.

22. Given this implementation, which of the following are stored on the stack in preparation for a call? Choose ALL that apply.

    (a) The value of the argument

    (b) A return pointer

    (c) The previous value stored in `ESP`

    (d) The value of the function

    (e) None of the above

23. Which line in the implementation above is responsible for generating the instruction(s) that restore the value of `ESP` to the value it held before the call? Enter the **line number** directly in the answer sheet.

24. Consider this stack layout:

0x1fe4	0x00000006 (006)
0x1fe8	0x00000066 (102)
0x1fec	0x00000066 (102)
0x1ff0	0x00000064 (100)
0x1ff4	0xSOMECODE (a return pointer)
0x1ff8	0x0000006c (108)
0x1ffc	0x00000054 (042)
0x2000	---------- (---)

Which of the following programs would produce this stack layout by the end of the program? Assume the starting stack index is `1` and the starting value in `ESP` is `0x2000`.

(a)
```
let f = fun x:
let y = x + 1 in
y + 3
end in
42 + (f(50))
```

(b)
```
let f = fun x:
let y = x + 1 in
let z = y + 3 in
z
end in
42 + (f(50))
```

(c)
```
let f = fun x:
let y = x + 1 in
y + 3
end in
(f(50)) + 42
```

(d)
```
let f = fun x:
let y = x + 1 in
let z = y + 3 in
z
end in
(f(50)) + 42
```

Consider adding operations to get and set the value within a box (similar to setting a value in a pointer). The following code introduces sequences of expressions, and these two operations:

```
1    (* ... in compile_expr, as before ... *)
2    | EBegin(es) ->
3      List.flatten (List.map (fun e -> compile_expr e env si) es)
4    | EGetBox(be) ->
5      let beis = compile_expr be env si in
6      beis @ [IMov(Reg(EAX), RegOffset(3, EAX))]
7    | ESetBox(be, ve) ->
8      let beis = compile_expr be env si in
9      let veis = compile_expr ve env (si + 1) in
10     beis @ (save_to_stack si) @ veis @
11     [
12       IMov(Reg(ECX), RegOffset(si * -4, ESP));
13       IMov(RegOffset(3, ECX), Reg(EAX))
14     ]
```

25. Given this implementation, what value will be in `EAX` after the following program runs? Assume the starting value in `EBX` is `0x100`.

```
let b = box 10 in
begin
  *b = box 4;
  *b;
end
```

(a) `0x109`

(b) `0x101`

(c) `0x008`

(d) `0x020`

(e) None of the above

26. Consider the following heap layout:

0x100	0x00000000 (00)
0x104	0x00000109 (--)
0x108	0x00000000 (00)
0x10c	0x00000101 (--)

Which of the following programs could generate this heap layout? Choose ALL that apply:

(a) `let b = box 10 in let b2 = box b in b2`

(b) `let b = box 10 in let b2 = box b in *b2 = b`

(c) `let b = box 10 in let b2 = box b in *b = b2`

(d) `let b = box 10 in let b2 = box b in *b = *b2`

(e) `let b = box box 10 in *(*b) = b`

Consider adding a mark/compact collector to Indigo. We won't duplicate the code for it here. Consider this starting stack and heap:

Stack

0x1ff0	0x00000121 (---)
0x1ff4	0xSOMECODE (a return pointer)
0x1ff8	0x00000008 (008)
0x1ffc	0x00000109 (---)
0x2000	---------- (---)

Heap

0x100	0x00000000 (00)
0x104	0x00000119 (--)
0x108	0x00000000 (00)
0x10c	0x00000111 (--)
0x110	0x00000000 (00)
0x114	0x00000109 (--)
0x118	0x00000000 (00)
0x11c	0x00000008 (08)
0x120	0x00000000 (--)
0x124	0x0000000a (10)

After a round of mark/compact, the new layout is:

Stack

0x1ff0	FILL ANSWER (---)
0x1ff4	0xSOMECODE (a return pointer)
0x1ff8	0x00000008 (008)
0x1ffc	FILL ANSWER (---)
0x2000	---------- (---)

Heap

0x100	0x00000000 (00)
0x104	FILL ANSWER (--)
0x108	0x00000000 (00)
0x10c	0x00000101 (--)
0x110	0x00000000 (00)
0x114	0x00000010 (--)
0x118	---------- (--)
0x11c	---------- (--)
0x120	---------- (--)
0x124	---------- (--)

For the next few questions, consider the following values:

(a) `0x101`

(b) `0x109`

(c) `0x10d`

(d) `0x121`

(e) `0x008`

27. What value should be at `0x1ff0` after the collection? Choose only one, though more than one answer may apply.

28. What value should be at `0x104` after the collection? Choose only one, though more than one answer may apply.

29. What value should be at `0x1ffc` after the collection? Choose only one, though more than one answer may apply.

EXTRA CREDIT The code for `EApp` has a serious, but subtle, bug in its use of `ECX`. Which of the following tests could fail if we used this implementation? Choose ALL that apply.

(a) `let f = fun x:  x + 1 end in let g = fun y:  y + 200 end in f(g(10))`
(which should produce 211)

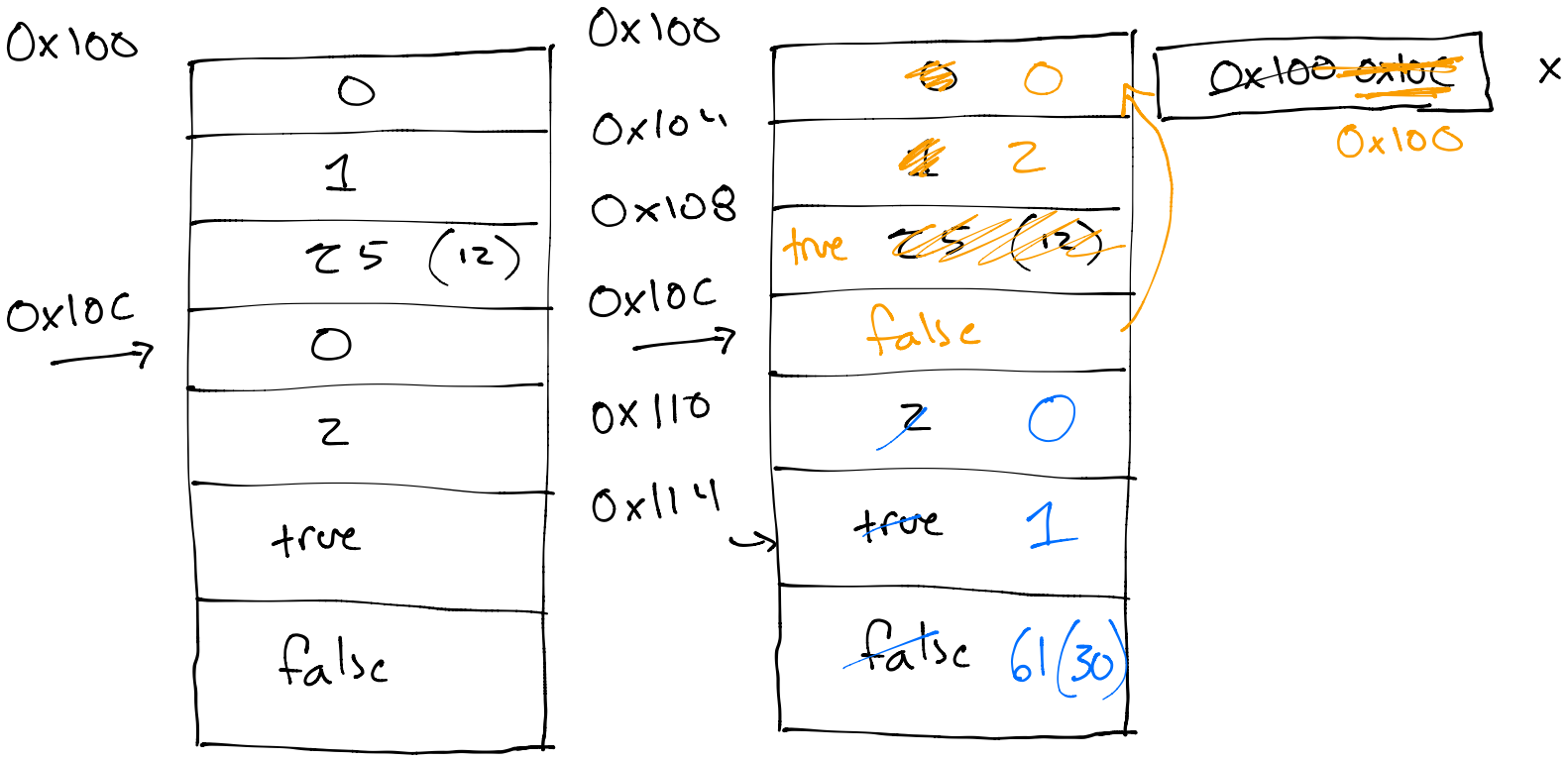
(b) `let f = fun x:  x + 1 end in let g = fun f2:  f2(200) end in g(f)`
(which should produce 201)

(c) `let f = fun x:  x + 1 end in let g = fun f2:  f2(200) + (f2(300)) end in g(f)`
(which should produce 502)

(d) `let f = fun x:  x + 1 end in f(f(5))`
(which should produce 7)

(e) `let f = fun x:  x + 1 end in f(5) + (f(7))`
(which should produce 14)

```
((def our_main (input)
  (let ((x (tup 12))) (begin (x:= (tup true false)) (x:= (tup 30)) x))))
```



```
<expr> :=
  |(mprint <expr list>)

<expr list> :=
  | <expr>
  | <expr> <expr list>

type expr =
  | EMPrint of expr list
```

| (let (<bind list>) <expr>)

```
((define (our_main input) (mprint (1 true 2 false (tup 1 2 3)))))
```

((define our_main (input) mprint 1 true 2 false (tup 1 2 3)))

```
((define our_main(input)
  (let ((x (tup 1 2 3)) (a (tup (tup 1) (tup 2))))
    (begin
      (x := (tup 4 5 6))
      (x := true)
      (x := (tup 1 2 3 4 5 6 7 8 9))
      a))))
```

can we re-use any space before alloc?

A: Yes
B: No

can we re-use any space before alloc?

A: Yes
B: No

21