

## CSE 131 Fall 2019 – Exam 1

**Answer sheet** Use the designated spaces; this is the only page that will be graded. The exam will be **45 minutes** and no additional study aids are allowed.

Name: \_\_\_\_\_ PID: \_\_\_\_\_

Write out “I excel with integrity” \_\_\_\_\_

Question 1 (2 points each)

Write a program

A.
B.
C.
D.
E.

Question 3 (2 points each)

Write SAME or a program

A.
B.
C.
D.
E.


**Question 2 (2 points each)**

Write the output

A.
B.
C.
D.
E.
F.
G.
H.

Question 4 (5 points)

Finish the implementation for EAnd



Question 4 (3 points per blank)

A, FILL 1.
A, FILL 2.
B.

Consider this compiler, which is similar to the ones you've seen in class and in PAs, but has a notion of *characters* as a type of value representing a single character, does not have booleans, and has the logical operators `and` and `or`. There are two files, `compile.ml` and `main.c`.

```

1  open Sexplib.Sexp
2  module Sexp = Sexplib.Sexp
3  open Printf
4
5  (*
6  expr := <number> | '<character>'
7         | (let (<name> <expr>) <expr>)
8         | (to-num <expr>) | (to-char <expr>)
9         | (or <expr> <expr>) | (and <expr> <expr>)
10        | (+ <expr> <expr>)
11  *)
12 type expr =
13   | ENum of int | EChar of char
14   | EOr of expr * expr | EAnd of expr * expr
15   | EToNum of expr | EToChar of expr
16   | EId of string
17   | ELet of string * expr * expr
18   | EPlus of expr * expr
19
20 let int_of_string_opt s =
21   try Some(int_of_string s) with Failure _ -> None
22
23 let rec sexp_to_expr (se : Sexp.t) : expr =
24   match se with
25   | Atom("true") -> ENum(1)
26   | Atom("false") -> ENum(0)
27   | Atom(s) ->
28     if ((String.length s) = 2) && ((String.get s 0) = '\') then
29       EChar(String.get s 1)
30     else
31       (match int_of_string_opt s with
32        | None -> EId(s)
33        | Some(i) -> ENum(i))
34   | List(sexps) ->
35     match sexps with
36     | [Atom("num"); arg1] -> EToNum(sexp_to_expr arg1)
37     | [Atom("chr"); arg1] -> EToChar(sexp_to_expr arg1)
38     | [Atom("+"); arg1; arg2] -> EPlus(sexp_to_expr arg1, sexp_to_expr arg2)
39     | [Atom("or"); arg1; arg2] -> EOr(sexp_to_expr arg1, sexp_to_expr arg2)
40     | [Atom("and"); arg1; arg2] -> EAnd(sexp_to_expr arg1, sexp_to_expr arg2)
41     | [Atom("let"); List([Atom(name); e1]); e2] ->
42       ELet(name, sexp_to_expr e1, sexp_to_expr e2)
43     | _ -> failwith "Parse error"
44
45 let parse (s : string) : expr = sexp_to_expr (Sexp.of_string s)
46
47 let count = ref 0
48 let gen_tmp str =
49   begin count := !count + 1; sprintf "%s%d" str !count end
50
51 let stackloc i = (i * 8)
52 let stackval i = sprintf "[rsp - %d]" (stackloc i)
53 type tenv = (string * int) list
54
55 let rec find (env : tenv) (x : string) : int option =
56   match env with
57   | [] -> None
58   | (y, i)::rest ->
59     if y = x then Some(i) else find rest x
60
61 let rec e_to_is (e : expr) (si : int) (env : tenv) =

```

```

62 match e with
63 | ENum(i) -> [sprintf "mov rax, %d" ((i * 2) + 1)]
64 | EChar(c) -> [sprintf "mov rax, %d" ((Char.code c) * 2)]
65 | EToNum(e) ->
66   let e_is = e_to_is e si env in
67   e_is @ [sprintf "sub rax, 1"]
68 | EToChar(e) ->
69   let e_is = e_to_is e si env in
70   e_is @ [sprintf "add rax, 1"]
71 | EPlus(e1, e2) ->
72   let e1_is = e_to_is e1 si env in
73   let e2_is = e_to_is e2 (si + 1) env in
74   let store_e1 = (sprintf "mov [rsp-%d], rax" (stackloc si)) in
75   let check = [
76     "mov rbx, rax; and rbx, 1"; sprintf "and rbx, [rsp-%d]" (stackloc si);
77     "cmp rbx, 0"; "je op_error"
78   ] in
79   e1_is @ [store_e1] @ e2_is @ check @ [
80     sprintf "and rax, 0xFFFFFFFFFFFFFFFE";
81     sprintf "add rax, [rsp-%d]" (stackloc si) ]
82 | EId(x) ->
83   (match find env x with
84   | None -> failwith "Unbound id"
85   | Some(i) -> [sprintf "mov rax, [rsp - %d]" (stackloc i)])
86 | ELet(x, v, body) ->
87   let vis = e_to_is v si env in
88   let bis = e_to_is body (si + 1) ((x,si)::env) in
89   vis @ [sprintf "mov [rsp - %d], rax" (stackloc si)] @ bis
90 | EOr(e1, e2) ->
91   let fin_lbl = gen_tmp "or_end" in
92   let e1_is = e_to_is e1 si env in
93   let store_e1 = (sprintf "mov [rsp-%d], rax" (stackloc si)) in
94   let e2_is = e_to_is e2 (si + 1) env in
95   e1_is @ [store_e1] @
96   ["cmp rax, 1"; sprintf "jne %s" fin_lbl] @
97   e2_is @ [fin_lbl ^ ":" ]
98 | EAnd(e1, e2) ->
99   let fin_lbl = gen_tmp "and_end" in
100   let e1_is = e_to_is e1 si env in
101   let store_e1 = (sprintf "mov [rsp-%d], rax" (stackloc si)) in
102   let e2_is = e_to_is e2 (si + 1) env in
103   e1_is @ [store_e1]
104   (* YOU WILL FILL THIS IN FOR A QUESTION *)
105
106 let compile (program : string) : string =
107   let ast = parse program in
108   let instrs = e_to_is ast 1 [] in
109   let instrs_str = (String.concat "\n" instrs) in
110   sprintf "
111 section .text
112 global our_code_starts_here
113 extern print_err_exit
114 our_code_starts_here:
115     %s
116     ret
117 op_error:
118     push 0
119     call print_err_exit
120     \n" instrs_str
121
122 let () =
123   let input_file = (open_in (Sys.argv.(1))) in
124   let input_program = (input_line input_file) in
125   let program = (compile input_program) in
126   printf "%s\n" program;;

```

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  extern int64_t our_code_starts_here() asm("our_code_starts_here");
5  extern void print_err_exit() asm("print_err_exit");
6
7  void print_err_exit() {
8      printf("Error\n");
9      exit(1);
10 }
11
12 int main(int argc, char** argv) {
13     int64_t result = our_code_starts_here();
14     if((result & 1)) {
15         printf("%lld\n", (result - 1) / 2);
16     }
17     else if(result < 512) {
18         // %c is a formatting character for printing single chars
19         printf("%c\n", (char)(result / 2));
20     }
21     else {
22         printf("Unrepresentable\n");
23     }
24     return 0;
25 }

```

## Directions and Assumptions

There is a lot of OCaml code provided above. There aren't any intentional gotchas in the code above 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.

- Don't worry about the presence or absence of size modifiers like `DWORD` – assume all instructions work with full 8-byte words.
- For this exam, don't consider overflow behavior (all the examples stay within small integer ranges)

Write all of your answers on the answer sheet at the front of the exam. You can detach it if you want (and it may make reading some of the code easier). There is some extra documentation provided in the exam, and the existing code serves as a broad set of examples for you to draw from. You are not allowed any additional study aids.

## Characters

The new kind of value in this compiler—characters—represent a single ASCII character (similar to the `char` type in C or Java). They are written in concrete syntax, and printed, with a single leading quote character. So the program `'d` is valid concrete syntax, and should evaluate to (and print) the value `'d`, represented as shown in the `EChar` case of the compiler.

1. For each of the following assembly programs, give an example in concrete syntax of a program in the language above that would produce these instructions when compiled. Write the expression directly in the answer sheet. There is a partial ASCII table on the next page for reference.

A.

```
mov rax, 1
mov [rsp-8], rax
cmp rax, 1
jne or_end1
mov rax, 15
or_end1:
```

B.

```
mov rax, 198
sub rax, 1
```

C.

```
mov rax, 194
mov [rsp - 8], rax
mov rax, [rsp - 8]
mov [rsp-16], rax
mov rax, 21
and rax, 1
and rax, [rsp-16]
cmp rax, 0
je op_error
and rax, 0xFFFFFFFFFFFFFFFE
add rax, [rsp-16]
```

D.

```
mov rax, 193
mov [rsp - 8], rax
mov rax, 15
mov [rsp-16], rax
and rax, 1
and rax, [rsp-16]
cmp rax, 0
je op_error
mov rax, [rsp - 8]
and rax, 0xFFFFFFFFFFFFFFFE
add rax, [rsp-16]
add rax, 1
```

E.

```
mov rax, 101
mov [rsp - 8], rax
mov rax, [rsp - 8]
mov [rsp-16], rax
and rax, 1
and rax, [rsp-16]
cmp rax, 0
je op_error
mov rax, [rsp - 8]
and rax, 0xFFFFFFFFFFFFFFFE
add rax, [rsp-16]
```

2. For each of the following programs, indicate what it would print when fully compiled and run through this compiler when linked against `main` for printing the result. Write your answer directly in the answer sheet. Your answer should look like what would be printed at the console – don't surround it in quotes. You can ignore any leading or trailing whitespace.

A fragment of an ASCII table is included for convenience below

- A. `(chr 'm)`
- B. `(num 'm)`
- C. `(+ 1 'j)`
- D. `(or 'k 1)`
- E. `(or 0 'b)`
- F. `(let (x 108) (chr (num (chr x))))`
- G. `(or (or 0 5) (+ 1 'g))`
- H. `(num 1000)`

---

a	97
b	98
c	99
d	100
e	101
f	102
g	103
h	104
i	105
j	106
k	107
l	108
m	109
n	110
o	111
p	112
q	113
r	114
s	115
t	116
u	117
v	118
w	119
x	120
y	121
z	122

3. Consider each of the following alternate implementations of the `EOr` case in the compiler. For each, either write `SAME` if this implementation would have the same behavior in terms of outputs and errors for all possible input programs, or give an example of an input program for which it would have different behavior.

A.

```
1 | EOr(e1, e2) ->
2   let fin_lbl = gen_tmp "or_end" in
3   let e1_is = e_to_is e1 si env in
4   let e2_is = e_to_is e2 si env in
5   e1_is @
6   ["cmp rax, 1"; sprintf "jne %s" fin_lbl] @
7   e2_is @ [fin_lbl ^ ":"]
```

B.

```
1 | EOr(e1, e2) ->
2   let fin_lbl = gen_tmp "or_end" in
3   let e1_is = e_to_is e1 si env in
4   let store_e1 = (sprintf "mov [rsp-%d], rax" (stackloc si)) in
5   let restore_e1 = (sprintf "mov rax, [rsp-%d]" (stackloc si)) in
6   let e2_is = e_to_is e2 si env in
7   e1_is @
8   ["cmp rax, 1"; sprintf "jne %s" fin_lbl] @
9   e2_is @ [store_e1; fin_lbl ^ ":"; restore_e1]
```

C.

```
1 | EOr(e1, e2) ->
2   let fin_lbl = gen_tmp "or_end" in
3   let e1_is = e_to_is e1 si env in
4   let e2_is = e_to_is e2 (si + 1) env in
5   e1_is @
6   ["cmp rax, 1"; sprintf "jne %s" fin_lbl] @
7   e2_is @ [fin_lbl ^ ":"]
```

D.

```
1 | EOr(e1, e2) ->
2   let fin_lbl = gen_tmp "or_end" in
3   let e1_is = e_to_is e1 si env in
4   let store_e1 = (sprintf "mov [rsp-%d], rax" (stackloc si)) in
5   let restore_e1 = (sprintf "mov rax, [rsp-%d]" (stackloc si)) in
6   let e2_is = e_to_is e2 si env in
7   e1_is @ [store_e1] @
8   ["cmp rax, 1"; sprintf "jne %s" fin_lbl] @
9   e2_is @
10  ["cmp rax, 1"; sprintf "jne %s" fin_lbl] @
11  [restore_e1; fin_lbl ^ ":"]
```

E.

```
1 | EOr(e1, e2) ->
2   let fin_lbl = gen_tmp "or_end" in
3   let e2_is = e_to_is e2 si env in
4   let e1_is = e_to_is e1 si env in
5   e2_is @
6   ["cmp rax, 1"; sprintf "jne %s" fin_lbl] @
7   e1_is @ [fin_lbl ^ ":"]
```

4. Assume that we would like the following behavior for **EAnd**

```
(and 4 5)           evaluates to      5
(and 'c 'b)         evaluates to      'b
(and 'c 0)          evaluates to      'b
(and 0 3)           evaluates to      0
(and 3 0)           evaluates to      0
(and (+ 0 0) 0)     evaluates to      0
(and 0 (+ 'c 4))    evaluates to      0
```

In English, if the first expression evaluates to 0, the result is 0 and the second expression isn't evaluated. If the first value is non-zero or a character, the second expression is evaluated and its result is the result of the whole expression. (This is the behavior of **and** in Python.)

The code for the **EAnd** case is incomplete in the given compiler. Fill in the remainder of the code for the **EAnd** case to implement this semantics. Write the necessary OCaml code directly in the answer sheet.

5. Your colleague proposes an idea for improving the compilation of **+** expressions to avoid some tag checks and extra storage. They propose adding an *extra match* case in the compiler, that comes before the existing **EPlus** case. The idea is that since the variable must already be stored on the stack, there's no need to generate instructions for it and give it new storage. Then **rbx** can be used for temporary storage of the checked value.

```
| EPlus(EId(x), e2) ->
begin match find env x with
| None -> failwith "Unbound id"
| Some(i) ->
  let e2_is = e_to_is e2 si env in
  let check = [
    "mov rbx, rax"; "and rbx, 1";
    sprintf "and rbx, [rsp-%d]" (stackloc i);
    "cmp rbx, 0"; "je op_error"
  ] in
  e2_is @ check @ [
    sprintf "and rax, 0xFFFFFFFFFFFFFFFE";
    sprintf "add rax, [rsp-%d]" (stackloc i) ]
end
```

You see promise in this idea and decide to extend it to other cases.

- A. The code below is the start of a similar attempt for when the first expression is known to be a number. Fill in the two blanks labelled **FILL** with OCaml code that will provide an implementation that has the same behavior as the general case, but uses **no memory operations** (instructions with arguments like **[rsp-8]**).<sup>1</sup>

```
| EPlus(ENum(n), e2) ->
  let e2_is = e_to_is e2 si env in
  let check = [
    (* FILL 1 *)
  ] in
  e2_is @ check @ [
    sprintf "and rax, 0xFFFFFFFFFFFFFFFE";
    (* FILL 2 *)
  ]
```

- B. The code below is the start of a similar attempt for when the first expression is known to be a character. Fill in a working implementation of this case that has the same behavior as the general case.

```
| EPlus(EChar(n), e2) -> (* FILL *)
```

<sup>1</sup>A reminder that for the purposes of this exam, ignore issues related to overflow.