Name: ______________________________ PID: ________________________ **(P. 1)**

INSTRUCTIONS

- Write your answers on the designated **answer sheet** in the specified areas – this is what we grade. If you need more space, raise your hand. Write your name and PID on this page, but **more importantly**, put your name and PID on the **answer sheet**. We grade what is on the answer sheet but collect all of the pages.
- We won't answer most questions about the exam during the exam time, and any questions we do answer will be posted on the projector for all to see. If you think a question is completely nonsense and unanswerable, you can write BAD QUESTION on the answer sheet. Generally, do your best to answer in the spirit of the question.
- We will give one opportunity to leave early at the 30min mark. Please **do not** leave before then, unless you have to use the restroom urgently, etc. It is distracting and disrespectful to your fellow students to have people walking around while they are trying to concentrate.
- Turn off and put away all cellphones, calculators, and other electronic devices. You may not access any electronic devices during the exam period. You cannot use external resources/notes for the exams.
- To receive full credit, your answers must be written legibly, and sufficiently darkly to scan well. Your solution will be evaluated both for correctness and clarity. Read the instructions for each part carefully to determine what is required for full credit.
- This exam is **45 minutes** long. Read all the problems first before you start working on any of them, so you can manage your time wisely.
- Stay calm and work methodically – you can do this!

Reference

x86_64 Registers We've Used

`rax`	Return values/expression results
`rsp`	Stack Pointer, refers to return address at start of function, used to look up variables
`rdi`	Holds 1st argument in "standard" calling convention
`rsi`	Holds 2nd argument in "standard" calling convention
`rdx`	Holds 3rd argument in "standard" calling convention
`rbx/rcx`	Used by us as temporary storage/for tag checking

x86_64 Instructions

`mov <reg>, <val>`	Move value to register
`mov <mem>, <val>`	Move value to memory (val can be register or immediate)
`push <val>`	Subtract 8 from `rsp` and store `<val>` at `[rsp]`
`pop <reg>`	Load value from `[rsp]` into `<reg>` and add 8 to `rsp`
`add/sub/imul <reg>, <val>`	Arithmetic
`and/or/xor <reg>, <val>`	Bitwise operators
`shr <reg>, <val>`	Shift `<reg>` right by `<val>` bits, filling with 0s
`sar <reg>, <val>`	Shift `<reg>` right by `<val>` bits, maintaining sign bits
`shl <reg>, <val>`	Shift `<reg>` left by `<val>` bits, filling with 0s
`test <reg>, <val>`	Bitwise and `<val>` and `<reg>` for condition codes, reg unchanged
`cmp <reg>, <val>`	Subtract `<val>` from `<reg>` and set condition codes, `<reg>` unchanged
`cmove/cmovl/cmovne/... <reg1>, <reg2>`	Move the value from `reg2` to `reg1` if the condition codes match
`<label>:`	Create a label (not really an instruction)
`jmp <label>`	Unconditional jump
`je/jne/jg/jge/jl/jle/jo <label>`	Conditional jumps based on condition codes
`call <label>`	Push (as with `push`) the address of next instruction and jump to `<label>`
`ret`	Pop the stack (as with `pop`) and jump to it

Rust Reference

`e >> n` Shift `e` to the right by `n` bits. Do signed/unsigned shift based on type (e.g. `i64` shifts signed, `u64` shifts unsigned)

`e1 & e2, e1 | e2` Bitwise operators

`e as t` Interpret the bits of the value `e` as type `t`. For example `let num_unsigned = num as u32;` when `num` is `i64` will reinterpret the lower 32 bits of the signed integer as an unsigned one.

`char` A type in Rust, a single Unicode "scalar value", 32 bits/4 bytes long.

`v[..]` Create a *slice* of a vector or string value `v`. Useful for pattern matching vectors and for getting a `&str` from a `String`.

Compiler Reference

The code below is the compiler we wrote in class, with some of the error checking for tags removed (that error checking is irrelevant for the questions on the exam). Specifically, it is mostly copy-pasted from the `cobra` branch of the lecture notes. You may use it as a reference; included is both the compiler from `src/main.rs` and the runtime from `runtime/start.rs`. Questions on the exam will ask about this code and its behavior, and about potential modifications to it.

```
use im::HashMap;
use sexp::Atom::*;
use sexp::*;
use std::env;
use std::fs::File;
use std::io::prelude::*;

enum Expr {
    Num(i32), True, False,
    Plus(Box<Expr>, Box<Expr>), Eq(Box<Expr>, Box<Expr>),
    Let(String, Box<Expr>, Box<Expr>), Id(String), Set(String, Box<Expr>)
    If(Box<Expr>, Box<Expr>, Box<Expr>),
    Loop(Box<Expr>), Break(Box<Expr>),
    Block(Vec<Expr>), Print(Box<Expr>),
}

fn parse_expr(s: &Sexp) -> Expr { ... elided ... }

fn new_label(l: &mut i32, s: &str) -> String {
    let current = *l;
    *l += 1;
    format!(”{s}_{current}”)
}

fn compile_expr(e: &Expr, si: i32, env: &HashMap<String, i32>, brake: &str, lbl: &mut i32) -> String {
  match e {
    Expr::Num(n) => format!(”mov rax, {}”, *n << 1),
    Expr::True => format!(”mov rax, {}”, 3),
    Expr::False => format!(”mov rax, {}”, 1),
    Expr::Id(s) if s == ”input” => format!(”mov rax, rdi”),
    Expr::Id(s) => format!(”mov rax, [rsp - {}]”, env.get(s).unwrap() * 8),
    Expr::Print(e) => {
      let e_is = compile_expr(e, si, env, brake, l);
      let index = if si % 2 == 1 { si + 1 } else { si };
      let offset = index * 8;
      format!(”
        {e_is}
        mov [rsp - {offset}], rdi
        sub rsp, {offset}
        mov rdi, rax
        call snek_print
        add rsp, {offset}
        mov rdi, [rsp - {offset}]
      ”)
    }
    Expr::Set(name, val) => {
      let offset = env.get(name).unwrap() * 8;

      let save = format!(”mov [rsp - {offset}], rax”);
      let val_is = compile_expr(val, si, env, brake, l);
      format!(”
        {val_is}
        {save}
        ”)
    }
    Expr::Break(e) => {
      let e_is = compile_expr(e, si, env, brake, lbl);
      format!(”
        {e_is}
        jmp {brake}
      ”)
    }
    Expr::Loop(e) => {
```

```
   let startloop = new_label(l, "loop");
   let endloop = new_label(l, "loopend");
   let e_is = compile_expr(e, si, env, &endloop[..], lbl);
   format!("
    {startloop}:
    {e_is}
    jmp {startloop}
    {endloop}:
   ")
  }
  Expr::Block(es) => {
   es.into_iter().map(|e| { compile_expr(e, si, env, brake, lbl) }).collect::<Vec<String>>().join("\n")
  }
  Expr::If(cond, thn, els) => {
   let end_label = new_label(l, "ifend");
   let else_label = new_label(l, "ifelse");
   let cond_instrs = compile_expr(cond, si, env, brake, lbl);
   let thn_instrs = compile_expr(thn, si, env, brake, lbl);
   let els_instrs = compile_expr(els, si, env, brake, lbl);
   format!("
    {cond_instrs}
    cmp rax, 1
    je {else_label}
     {thn_instrs}
    jmp {end_label}
    {else_label}:
     {els_instrs}
    {end_label}:
   ")
  }
  Expr::Eq(e1, e2) => {
   let e1_instrs = compile_expr(e1, si, env, brake, lbl);
   let e2_instrs = compile_expr(e2, si + 1, env, brake, lbl);
   let offset = si * 8;
   format!("
     {e1_instrs}
     mov [rsp - {offset}], rax
     {e2_instrs}
     cmp rax, [rsp - {offset}]
     mov rbx, 3
     mov rax, 1
     cmove rax, rbx
   ")
  }
  Expr::Plus(e1, e2) => {
   let e1_instrs = compile_expr(e1, si, env, brake, lbl);
   let e2_instrs = compile_expr(e2, si + 1, env, brake, lbl);
   let stack_offset = si * 8;
   format!("
    {e1_instrs}
    mov [rsp - {stack_offset}], rax
    {e2_instrs}
    add rax, [rsp - {stack_offset}]
   ")
  }
  Expr::Let(name, val, body) => {
   let val_is = compile_expr(val, si, env, brake, lbl);
   let body_is = compile_expr(body, si + 1, &env.update(name.to_string(), si), brake, lbl);
   let offset = si * 8;
   format!("
    {val_is}
    mov [rsp - {offset}], rax
    {body_is}
   ")
  }

 }
}

fn main() -> std::io::Result<()> {
  let args: Vec<String> = env::args().collect();
  let in_name = &args[1];
```

```
    let out_name = &args[2];
    let mut in_file = File::open(in_name)?;
    let mut in_contents = String::new();
    in_file.read_to_string(&mut in_contents)?;
    let expr = parse_expr(&parse(&in_contents).unwrap());
    let mut labels = 0;
    let result = compile_expr(&expr, 2, &HashMap::new(), &String::from(""), &mut labels);
    let asm_program = format!(
        "
section .text
global our_code_starts_here
our_code_starts_here:
  {}
  ret
",
        result
    );

    let mut out_file = File::create(out_name)?;
    out_file.write_all(asm_program.as_bytes())?;

    Ok(())
}
```

```
use std::env;
#[link(name = "our_code")]
extern "C" {
    // The \x01 here is an undocumented feature of LLVM that ensures
    // it does not add an underscore in front of the name.
    // Courtesy of Max New (https://maxsnew.com/teaching/eecs-483-fa22/hw_adder_assignment.html)
    #[link_name = "\x01our_code_starts_here"]
    fn our_code_starts_here(input : i64) -> i64;
}

#[no_mangle]
#[export_name = "\x01snek_print"]
fn snek_print(val : i64) -> i64 {
  if val == 3 { println!("true"); }
  else if val == 1 { println!("false"); }
  else if val % 2 == 0 { println!("{}", val >> 1); }
  else {
    println!("Unknown value: {}", val);
  }
  return val;
}

fn parse_arg(v : &Vec<String>) -> i64 {
  if v.len() < 2 { return 1 }
  let s = &v[1];
  if s == "true" { 3 }
  else if s == "false" { 1 }
  else { s.parse::<i64>().unwrap() << 1 }
}

fn main() {
    let args: Vec<String> = env::args().collect();
    let input = parse_arg(&args);

    let i : i64 = unsafe { our_code_starts_here(input) };
    snek_print(i);
}
```

Question 1: Compiler Behavior

For each of the following generated assembly snippets, give a Snek program that would have generated it if compiled with the compiler given above (ignore extra or missing whitespace). Write the Snek program directly in the answer sheet.

A.

```
mov rax, 3
```

B.

```
mov rax, 1
cmp rax, 1
je ifelse_1
mov rax, 1000
jmp ifend_0
ifelse_1:
mov rax, 14
ifend_0:
```

C.

```
mov rax, 20
mov [rsp - 16], rax
mov rax, [rsp - 16]
mov [rsp - 24], rax
mov rax, 20
add rax, [rsp - 24]
```

D.

```
loop_0:
mov rax, 74
mov [rsp - 16], rax
mov rax, [rsp - 16]
mov [rsp - 24], rax
mov rax, -2
add rax, [rsp - 24]
mov [rsp - 16], rax
jmp loop_0
loopend_1:
```

E.

```
mov rax, rdi
cmp rax, 1
je ifelse_1
  mov rax, 6
jmp ifend_0
ifelse_1:
  mov rax, 8
ifend_0:

mov [rsp - 16], rax

mov rax, [rsp - 16]
mov [rsp - 32], rdi
sub rsp, 32
mov rdi, rax
call snek_print
add rsp, 32
mov rdi, [rsp - 32]
```

Question 2: Adding char

Let's consider adding single characters as a new datatype to Snek.

Concrete syntax: In a Snek program, a char is a single unicode code point surrounded by single quotes. **Examples:** 'a', 'λ'.

Representation: A char is represented with a 32-bit unicode value in the upper part of the word, with the lower 32-bits being the tag 0x00000003. **Example:** The character 'a' is represented as 0x0000006100000003. The character 'λ' is represented as 0x000003BB00000003. You don't need to have ASCII (or Unicode) values memorized to complete the exam. Assume that all unicode characters fit in 32 bits.[1]

Because the representation 0x0000000000000003 would be ambiguous (true vs \0), we change the representation of true to be 0x5. false remains 0x1. This means the tag for booleans is that the low two bits are always 01. The representation of numbers from class remains unchanged (0 lowest bit, number n represented as $2n$ by shifting to the left).

Part 1: Literal and Printing

Fill in the blank below with code that generates assembly according to the specification above for the Char case of compile_expr:

```
  fn compile_expr(
      e: &Expr, si: i32, env: &HashMap<String, i32>,
      brake: &str, l: &mut i32) -> String {
    match e {
        Expr::Num(n) => format!("mov rax, {}", *n << 1),
        Expr::True => format!("mov rax, {}", 5),
        Expr::False => format!("mov rax, {}", 1),
        Expr::Char(c) => {
          // This line puts the character in the lower 32 bits of ch, which
          // has type u64. The upper 32 bits are all 0. For an input character
          // 'a', this would be 0x0000000000000061.
          let ch : u64 = c.chars().nth(1).unwrap() as u64;

          (A)                                 // Can be multiple lines
        },
        ...
    }
}
```

Fill in the blanks below with Rust code that prints a char if given its representation as val.

```
  fn snek_print(val : i64) -> i64 {
    if val == 5 { println!("true"); }
    else if val == 1 { println!("false"); }
    else if (B)                              {
      let code_point : u32 = (C)                              ;
      let c : char = char::from_u32(code_point).unwrap();
      println!("{}", c);
    }
    else if val % 2 == 0 { println!("{}", val >> 1); }
    else {
      println!("Unknown value: {}", val);
    }
    return val;
  }
```

[1] This also happens to be true!

Question 3: Adding continue

Let's add continue to our language. It should have the effect of moving control onto the next loop iteration when evaluated. For example, this loop, when evaluated with input equal to 10, would print the numbers from 9 to 1 in decreasing order, skipping 5 and 2:[2]

```
(let (n input)
  (loop
    (block
      (set! n (- n 1))
      (if (= n 0) (break 0) false)
      (if (= n 2) (continue) false)
      (if (= n 5) (continue) false)
      (print n))))
```

A few notes:

- The concrete syntax is (continue)
- The abstract syntax is a new variant of Expr, Continue, which has no fields (like True or False)

Consider the fragment of compiler implementation below, and come up with code to fill in each of the blanks to complete the implementation of continue. Put your answers as Rust code on the answer sheet.

```
fn compile_expr(
    e: &Expr, si: i32, env: &HashMap<String, i32>,
    brake: &str, (A)                              ,
    lbl: &mut i32,
) -> String {
  match e {
    ...
    Expr::Continue => {
      (B)                              // use multiple lines if needed
    }
    Expr::Break(e) => {
      let e_is = compile_expr(e, si, env, brake, (C)                              , lbl);
      format!(
          "
        {e_is}
        jmp {brake}
      ")
    }
    Expr::Loop(e) => {
        let start = new_label(l, "loop");
        let end = new_label(l, "loopend");
        let e_is = compile_expr(e, si, env, &end[..], (D)                              , lbl);
        format!(
            "
          {start}:
          {e_is}
          jmp {start}
          {end}:
        ")
    }
    ...
  }
}
```

[2] The false in the else branches of the if expressions don't have any real meaning, they're just used to approximate a "single-arm" if statement.

Question 4: Generating Labels

In the compiler from class, the last argument had type `&mut i32`. This was used in conjunction with the `new_label` helper to generate unique labels.

Consider instead a situation where we use just `lbl: i32` (not a mutable reference), and add 1 to it on each recursive call to `compile_expr` where new loop labels are added. That is, we write something like this, considering just the boolean, number, plus, and if cases (the relevant changes are boxed):

```
fn compile_expr(
    e: &Expr, si: i32,
    env: &HashMap<String, i32>, brake: &str,
    lbl: i32) -> String {
  match e {
    Expr::Num(n) => format!("mov rax, {}", *n << 1),
    Expr::True => format!("mov rax, {}", 3),
    Expr::False => format!("mov rax, {}", 1),
    Expr::Plus(e1, e2) => {
      let e1_instrs = compile_expr(e1, si, env, brake, lbl);
      let e2_instrs = compile_expr(e2, si + 1, env, brake, lbl);
      let stack_offset = si * 8;
      format!("
        {e1_instrs}
        mov [rsp - {stack_offset}], rax
        {e2_instrs}
        add rax, [rsp - {stack_offset}]
      ")
    }
    Expr::If(cond, thn, els) => {
      let end_label = format("ifend{}", lbl); // changed from use of use new_label
      let else_label = format("ifelse{}", lbl); // changed from use of new_label
      let cond_instrs = compile_expr(cond, si, env, brake, lbl + 1);
      let thn_instrs = compile_expr(thn, si, env, brake, lbl + 1);
      let els_instrs = compile_expr(els, si, env, brake, lbl + 1);
      format!(
          "
        {cond_instrs}
        cmp rax, 1
        je {else_label}
          {thn_instrs}
        jmp {end_label}
        {else_label}:
          {els_instrs}
        {end_label}:
     ")
    }
  ...
  }
}
```

Which of the following `snek` language programs would have problems with duplicated labels if we made this change? Choose all and only those that apply, put your answers on the answer sheet.

A. `(if true 1 false)`

B. `(if true (if false 3 4) (if true 5 6))`

C. `(if true 3 (if true 5 6))`

D. `(if true (if true 5 6) true)`

E. `(if (if true 4 5) (if true 5 6) true)`

F. `(+ (if true 4 5) (if true 5 6))`

G. `(if true (+ (if false 3 4) 9) 10)`

H. `(if true (+ (if false 3 4) 9) (if true 10 11))`