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!

Adding Strings to the Language

Let's consider adding heap-allocated strings as a new datatype to Snek.

Concrete Syntax: In a Snek program, a `str` is a sequence of Unicode code points surrounded by single quotes. **Examples**: `'abc'`, `'hello world'`

Representation: A `str` is represented as an 8-byte aligned address, tagged with the last *three* bits `0b111`. At that address, the words on the heap store first the number of characters in the string, then a sequence of words, one per character, storing `char`s.[1]

Overall, to summarize the tags in this version of Snek:

- The tag for strings is `0b111`
- The tag for booleans is `0b101`, with `true` is represented as `0b1101` (decimal `13`) and `false` as `0b101` (decimal `5`).
- The tag for pairs is `0b001`
- The tag for `char`s is `0b011`
- The tag for numbers is `0b0` (and they are represented as shifted)

String Expression Compilation

This code in the compiler implements the string case. Fill in the blanks according to the specification above. Assume there is enough space to create the string on the heap.

```
1   Expr::Str(s) => {
2       let mut instrs = String::new();
3       instrs.push_str(&format!("mov qword (A)                    , {}\n", s.len()));
4       let mut i = 1;
5       for c in s.chars() {
6           instrs.push_str(&format!("mov rax, {}\n", ((c as u64) << 32) + 3));
7           instrs.push_str(&format!("mov [r15 + {}], rax\n", i * 8));
8           i += 1;
9       }
10
11      instrs.push_str(&format!("mov rax, (B)                    \n"));
12      instrs.push_str(&format!("add rax, (C)                    \n"));
13      instrs.push_str(&format!("add (D)                    , {}\n", i * 8));
14      instrs
15  }
```

[1] A `char` is a 64 bit word with the most significant digits storing a single Unicode code point and tagged with `0b011` in the lower 32 bits; this is the same representation as on the first exam.

Printing Strings

String values need to be printed, which means `snek_str` should be updated to print them as well. Fill in the blanks below to implement `snek_str` function for strings:

```
fn snek_str(val : i64, seen : &mut Vec<i64>) -> String {
  if val == 0b1101 { "true".to_string() }
  else if val == 0b101 { "false".to_string() }
  else if val == 1 { "nil".to_string() }
  else if val % 2 == 0 { format!("{}", val >> 1) }
  else if [(A)                                  ] {
    let addr = [(B)                             ] as *const i64;
    let len = unsafe { [(C)                     ] } as isize;
    let mut result = String::new();
    result.push('\'');
    for i in 0..len {
      let ch = unsafe { [(D)                    ] };
      result.push(char::from_u32((ch >> 32) as u32).unwrap());
    }
    result.push('\'');
    return result;
  }
  else if val & 3 == 3 {
    let ch = (val - 3) as u64;
    let mut result = String::new();
    result.push(char::from_u32((ch >> 32) as u32).unwrap());
    return result;
  }
  else if val & 1 == 1 {
    if seen.contains(&val)  { return "(pair <cyclic>)".to_string() }
    seen.push(val);
    let addr = (val - 1) as *const i64;
    let fst = unsafe { *addr };
    let snd = unsafe { *addr.offset(1) };
    let result = format!("(pair {} {})", snek_str(fst, seen), snek_str(snd, seen));
    seen.pop();
    return result;
  }
  else {
    format!("Unknown value: {}", val)
  }
}
```

Indexing Strings

Consider adding an *indexing* operation on strings, with concrete syntax `(char-at <expr> <expr>)`. The first expression should evaluate to a string and the second to a number. The returned value is the character at the given position (0-indexed) in the string, so `(char-at 'abc' 1)` evaluates to the single character `'b'`.[2]

Fill in this implementation of `CharAt`. **Assume** you can use any additional registers you need without saving or restoring them, that `e1` evaluates to a string and `e2` evaluates to a (Snek) number that is in-bounds for the string. So *only* fill in the instruction(s) needed to do the indexing, no error checking.

```
Expr::CharAt(e1, e2) => {
  let e1_instrs = compile_expr(e1, si, env, brake, l);
  let e2_instrs = compile_expr(e2, si + 1, env, brake, l);
  let stack_offset = si * 8;
  format!("{e1_instrs}
           mov [rsp + {stack_offset}], rax
           {e2_instrs}
           mov rbx, [rsp + {stack_offset}]
           [(A)                                  ] (Can be multiple lines)
         ")
}
```

[2]Note a slight abuse of notation, this returns a single character, not a string that's single character long

Concatenating Strings

Consider adding a *concatenation* operator on strings, with concrete syntax `(cat <expr> <expr>)`. Both subexpressions are expected to evaluate to strings, and the result is a new string with the characters from the first followed by the characters from the second. As an example `(cat 'cse' '231')` would produce `'cse231'`.

Because there's a nontrivial pair of loops involved, it's useful to implement `cat` using a runtime function that coordinates with generated code. Fill in the blanks in the code below to generate an implementation of `cat` according to the specification above. You can assume:

1. Both expressions evaluate to strings
2. At the start of each generated function in the Snek language, `sub rsp, <depth>` is evaluated and `add rsp, <depth>` is evaluated at the end, where `<depth>` is the number of locals used in the function body times 8.
3. Generated function calls to Snek language functions store their (1 or 2) arguments on the stack
4. The Snek language does not use the registers `rdi`, `rsi`, or `rdx`, so their values do not need to be saved.[3]
5. `extern "C"` functions use the standard x86_64 calling convention

As extra assumptions for this exam, do not consider any architectural requirements of alignment for `rsp`, and assume there is enough space to allocate the string.

```
// in runtime/start.rs
#[no_mangle]
#[export_name = "\x01snek_concat"]
/// Expects str1 and str2 to be string values.
/// Expects heap to be a pointer to the next free space for allocation
/// Creates the new string with the correct length and characters starting
/// at the address given by heap. Returns the address immediately after
/// the newly created string.
extern "C" fn snek_concat(str1 : i64, str2 : i64, heap: *mut i64) -> *mut i64 {
  // implementation omitted, refer to the comment above for behavior
  // you DO NOT need to implement this function, just use it
}
```

```
// in src/main.rs
Expr::Concat(e1, e2) => {
  let e1_instrs = compile_expr(e1, si, env, brake, l);
  let e2_instrs = compile_expr(e2, si + 1, env, brake, l);
  let stack_offset = si * 8;
  format!(
      "
    {e1_instrs}
    mov [rsp + {stack_offset}], rax
    {e2_instrs}
    (A)                                  (Can be multiple lines)
    call snek_concat
    (B)                                  (Can be multiple lines)
"
  )
}
```

[3] If you're concerned about `input`, `rdi` is stored in `r14` to use for the value of `input`.