Worksheet 6A

Quiz 1: *Tail calls*

Circle the calls that are tail calls.

```
(fun (fac n acc)
  (if (= n 0)
    acc
    (if (= n 2)
        (* 2 (fac (sub1 n) (* acc n)))
        (fac (sub1 n) (* acc n))
    )
  )
)
```

Quiz 2: *Which expressions can have tail calls?*

- **Circle** the expression that *can have* tail calls
- **Cross out** the expressions that *cannot have* tail calls

```
e ::= n | true | false | input | x | (add1 e) | (print e) | (+ e1 e2) | (= e1 e2)
    | (let (x e1) e2) | (if e1 e2 e3) | (set x e) | (block e1...en) | (loop e) | (break e)
    | (f e1) | (f e1 e2)
```

Quiz 3: *Identify Tail Calls*

```
fn compile_expr(e: &Expr, env: &Stack, funs: &FunEnv, lbl:&str, tr: bool) -> String {
  match e {
    Add1(e1) => {
      let e1_code = compile_expr(e1, env, funs, lbl,            );
      ...
    }
    Plus(e1, e2) => {
      let e1_code = compile_expr(e1, env, funs, lbl,        );
      let e2_code = compile_expr(e2, env, funs, lbl,        );
      ...
    }
    Let(x, e1, e2) => {
      let e1_code = compile_expr(e1, env, funs, lbl,        );
      let e2_code = compile_expr(e2, &newenv, funs, lbl,    );
      ...
    }
    If(cond, e_then, e_else) => {
      let cnd_code = compile_expr(e_cnd,  env, funs, lbl,       );
      let thn_code = compile_expr(e_then, env, funs, lbl,       );
      let els_code = compile_expr(e_else, env, funs, lbl,       );
      ...
    }
    Set(x, e) => {
      let e_code = compile_expr(e, env, funs, lbl,       );
      ...
    }
    Block(es) => {
      let n = es.len();
      let e_codes: Vec<String> = es.iter().enumerate()
          .map(|(i, e)| compile_expr(e, env, funs, lbl,        ))
          .collect();
      ...
    }
    Loop(e) => {
      let e_code = compile_expr(e, env, funs, &loop_exit,       );
      ...
    }
    Break(e) => {
      let e_code = compile_expr(e, env, funs, lbl,         );
      ...
    }
```

```
    Print(e) => {
      let e_code = compile_expr(e, env, funs, lbl,          );
      ...
    }
    Call2(f, e1, e2) => {
      let e1_code = compile_expr(e1, env, funs, lbl,        );
      let e2_code = compile_expr(e2, env, funs, lbl,        );
      ...
    }
  }
}
```

Quiz 4: *Modify Assembly to Use `jmp` for Tail Calls*

```
fn compile_defn(&mut self, defn: &expr::Defn) -> String {
  let label = entry_label(&defn.name);

  let entry = compile_entry(&defn.body);
  let stack = Stack::new(&defn.params[..]);
  let exit = exit_label();
  let body = self.compile_expr(&defn.body, &stack, &exit, true);
  let exit = compile_exit();
  format!(
    "{label}:\n\
     {entry}\n\

     {body}\n\
     {exit}"
  )
}

fn compile_expr(&mut self, e: &Expr, env: &Stack, lbl: &str, tr: bool) -> String {
  match e { // ...
    Call2(f, e1, e2) => {
      let e1_code = self.compile(e1, env, lbl, false);
      let e2_code = self.compile(e2, env, lbl, false);
      let call_code = if tr {
        format!(
          "{e1_code}\n\

           {e2_code}\n\



          "
        )
      } else { ... };
    }
  }
}
```

Quiz 5: *Your turn!*

What is something you found confusing in today's lecture (or earlier)?