```
<prog> := <defn>+ <expr>

<defn> := (fun (<name> <name>) <expr>) | (fun (<name> <name> <name>) <expr>)

<expr> := … | (<name> <expr>) | (<name> <expr> <expr>)
```

Example

```
(fun (sumsquare x y)
  (+ (* x x) (* y y)))

(sumsquare (* 2 input) (+ 30 (* 3 input)))
```

Generated code for example

Sketch of compiler code (and any additional infrastructure)

```
fn compile_definition(d: &Definition, labels: &mut i32) -> String {
    match d {
        Fun1(name, arg, body) => { … like below but one arg … }
        }
        Fun2(name, arg1, arg2, body) => {
            let body_env = hashmap! {
                arg1.to_string() => -1,
                arg2.to_string() => -2
            };
            let body_is = compile_expr(body, 2, &body_env, &String::from(""), labels);
            format!(
                "{name}:
                {body_is}
                ret"
            )
        }
    }
}



… in compile_expr …
        Expr::Call2(name, arg1, arg2) => {
            let arg1_is = compile_expr(arg1, si, env, brake, l);
            let arg2_is = compile_expr(arg2, si + 1, env, brake, l);
            let curr_word = si * 8;
            let offset = (si * 8) + (2 * 8);
            // With this setup, the current word will be at [rsp+16], which is where arg1 is stored
            // We then want to get rdi at [rsp+16], arg2 at [rsp+8], and arg1 at [rsp], then call
            // Fill in the needed moves!
            format!(
                "
                {arg1_is}
                mov [rsp-{curr_word}], rax
                {arg2_is}
                sub rsp, {offset}









                call {name}
                mov rdi, [rsp+16]
                add rsp, {offset}
            "
            )
        }
```