1. The representation of values (intergers, boolean and None) are followed by the language description listed to PA2. I have a literal type in the Expr and then literal can be like on the right:

```
export type Literal =
    { tag: "None" }
  | { tag: "True", value: boolean, type: Type}
  | { tag: "False", value: boolean, type: Type}
  | { tag: "number", value: number, type: Type}

export type Type =
    { tag: "int" }
  | { tag: "bool" }
```

As you can see from the image, each literal has a type related to it. The reason I did this is because this gives me the power to know the type of literal in the generated AST and I am able to determine some of the simple type error in the AST phase. Assume we are adding 1+True. It would be interpreted by the parser as on the right. And from here we know the type of the operation on the left side and type of the operation on the right, we can do the type checking depending on the operation we are executing. As for the codegen part of the values. I have at the top defined some i64 value for true and false and None, with this I am able to generate the literal based on this. Integer would just be a normal (i64.const value). But for True, False, and None I would push the i64 number onto the stack and later when used, I compared the value on the stack True in codegen case “if” and push the result of the comparison to the stack.

```
{tag: “binop”,
 expr1: {tag:”literal”,
          value: {tag: “number”, value : 1, type:{tag:”int”}}},
 expr2: {tag:”literal”,
          value: {tag: “True”, value : true, type:{tag:”bool”}}},
 op: {tag:”add”}
}
```

```
export const TRUE = BigInt(1) << BigInt(32)
export const FALSE = BigInt(2) << BigInt(32)
export const NONE = BigInt(4) << BigInt(32)
```

```
x:int = 1

def foo(y:int):
  z:int = 2
  return x+ y + z
```

2. This is a sample program that uses one global variable one function with parameters and one variable defined inside a function.

```
export type GlobalEnv = {
  types: Map<string, string>
  globals: Map<string, number>;
  offset: number;}
```

Global is defined in a global Env. When every time compile is called, we would populate the globalEnv when we find the variable declaration statements. In the program it’s done by a function called **augmentEnv**. But this is only for the location and types of the global variable. We would actually store the value into a webassembly memory for repl to work. And here I am saying is that if the function has a variable declaration in it, we would set local value rather than go all the way into memory.

```
switch (stmt.tag) {
  case "init":
    if (isFunc) {
      var valStmts = codeGenExpr(stmt.value, env)
      valStmts.push(`(local.set $${stmt.name})`)
      return valStmts
    } else {
      const locationToSt = [`(i32.const ${envLookup(env, stmt.name)}) ;; ${stmt.name}`];
      var valStmts = codeGenExpr(stmt.value, env);
      return locationToSt.concat(valStmts).concat([`(i64.store)`]);
    }
```

```
case "id":
  if (env.globals.has(expr.name)) {
    return [`(i32.const ${envLookup(env, expr.name)})`, `(i64.load)`]
  }
  else {
    return [`(local.get $${expr.name})`] // take cares of parameters and local def
  }
```

And then when we are trying to refer to the defined variables. We would first find it in the glbal variables field and then we try to get the local defined variables. If the local or global does not have the definition anywhere, an error would be thrown.

As for the parameters, we would define all the params at the beginning of the function definition. And later on in the func body stmts when we are using the parameters, we can simple call (local.get ${name}) to get to the value of the parameter.

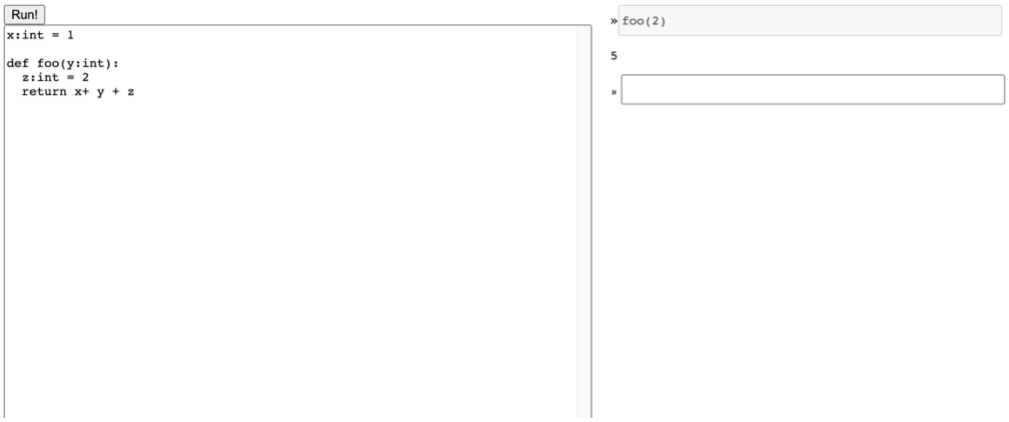
```
const funcBody = stmt.body
// Check if init or func def came before all other
var cameBefore = true
var otherAppear = false
funcBody.forEach(s => {
  if (s.tag === "define") { throw new Error("no function declare inside function body") };
  if (s.tag !== "init") {
    otherAppear = true
  }
  if (otherAppear && s.tag === "init") {
    cameBefore = false
  }
})
if (!cameBefore) { throw new Error("var_def should preceed all stmts") }

var params = stmt.parameters.map(p => `(param $${p.name} i64)`).join(" ");      You, a day ag
const funcVarDecls: Array<string> = [];
funcVarDecls.push(`(local $$last i64)`);
// Initialize function var def
funcBody.forEach(stmt => {
  if (stmt.tag == "init") {
    funcVarDecls.push(`(local $${stmt.name} i64)`);
  }
});
```

Following is the wasm code generated for the simple function here.

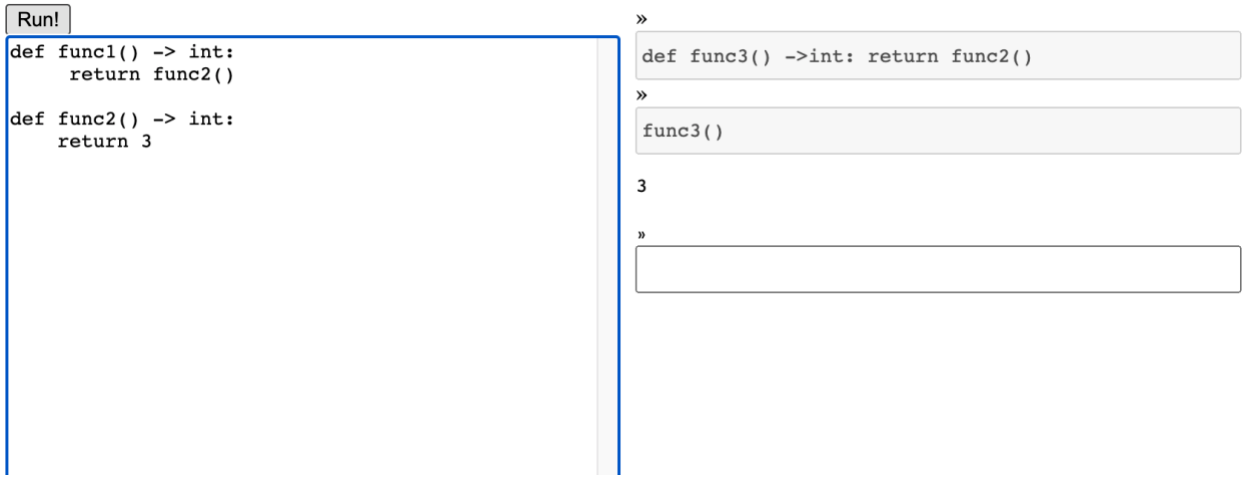
```
(module
   (func $print (import "imports" "imported_func") (param i64))
   (func $printglobal (import "imports" "print_global_func") (param i64) (param i64))
   (import "js" "memory" (memory 1))
   (func $foo (param $y i64) (result i64)
 (local $$last i64)
(local $z i64) (i64.const 2)
(local.set $z)
(i32.const 0)
(i64.load)
(local.get $y)
(i64.add)
(local.get $z)
(i64.add)
return)
   (func (export "exported_func")
     (local $$last i64)
(local $x i64)
(i32.const 0) ;; x
(i64.const 1)
(i64.store)
```

3. I used the while true to make the infinite loop. When I was trying to run the program, the webpage just stuck, and nothing is responding. I think I am still able to click the button and such, but nothing really happened. Even the close tab doesn't work for a while.

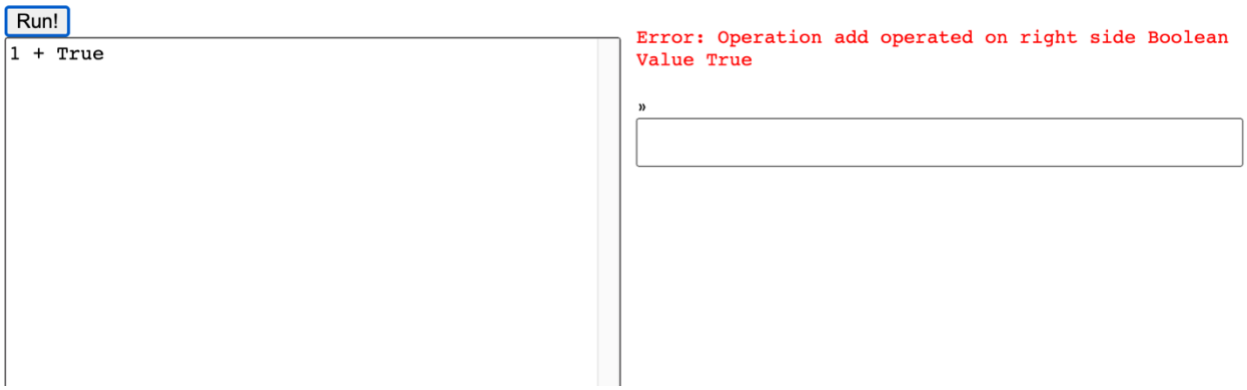
4.1 A function defined in the main program and later called from the interactive prompt



4.2 A function defined at the interactive prompt, whose body contains a call to a function from the main program, called at a later interactive prompt

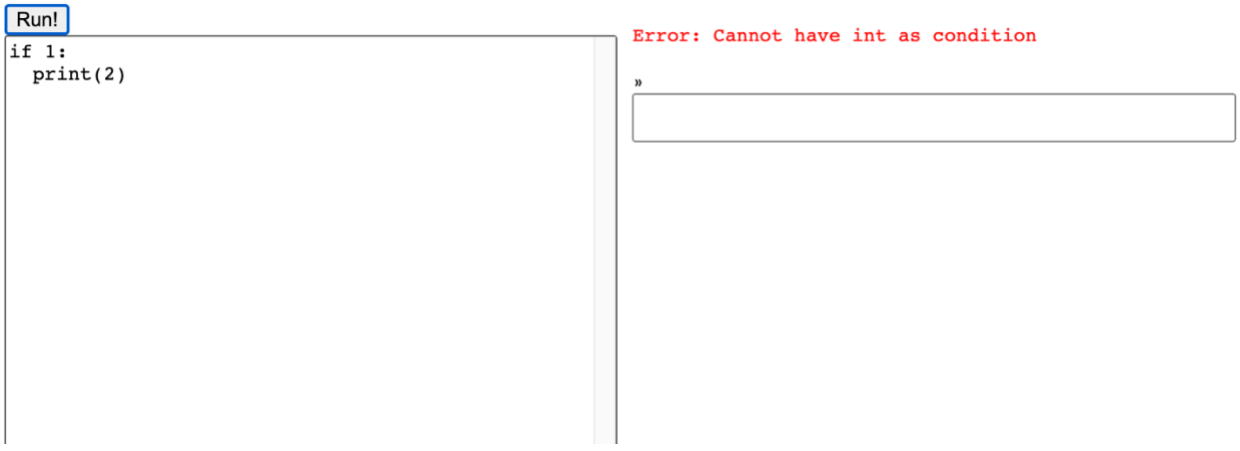


4.3 A program that has a type error because of a mismatch of booleans and integers on one of the arithmetic operations



4.4 A program that has a type error in a conditional position

I have some limitation here. If the condition is not an int or id, if it's an expression, I am not able to type check that. In order to fix this, I think I might need to have something in the codegen that specificly check the type of condition.



4.5 A program that calls a function from within a loop. I believe with this fun is only called twice.

Run!

```
x:int =1

def fun(x:int):
  y:int = 0
  y = x+1
  return y

while x < 5:
  x = fun(x) +1
```

```
» x
5
»
```

4.6

Run!

```
print(3)
```

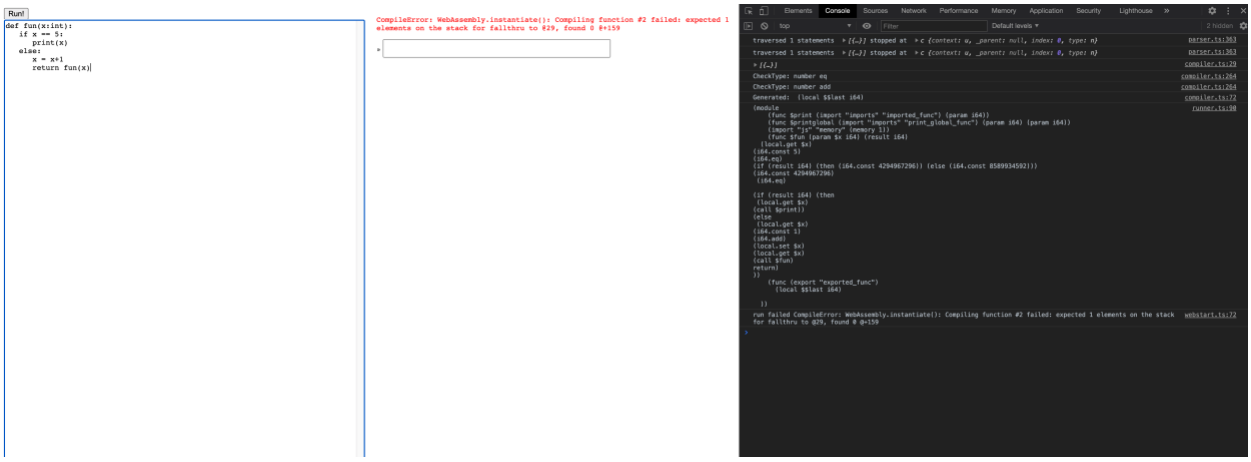
```
3
»
```

Run!

```
True
```

```
True
»
```

4.7



A recursive function in this way would not work because the problem stated. It seems that the parser has some error when generating the else statement. I've already spent too much time on this project but it's certain that some part of this can get improved.

4.8