Programming Assignment-2

1. Description of the representation of values (integers, booleans, and None)

Integers: I represent integers and booleans as i64. I consider integers the way they are because they always can be represented by 32 bits.

Booleans: I represent booleans as i64 too. I need to distinguish between booleans and integers for typechecking and printing i.e. I have to print "True" for a bool True. I distinguish booleans from integers by making the 33rd bit from right as 1 for booleans. I represent True as 2^32 + 1 and False as 2^32. Therefore, during printing or any type checking I decide that it's a boolean based on its 33rd bit as mentioned below.

```
if (arg < 2**32) { // This is a number
    elt.innerText = arg;
    return BigInt(arg);
}
else if (BigInt(arg) >= 2**32 && BigInt(arg) < 2**33) {
    // Bools are added with 2^32.
    arg = BigInt(arg) & BigInt(1);
    if (arg == 1) {
        elt.innerText = "True";
    }
    else if (arg == 0) {
        elt.innerText = "False";
    }
    else {
        throw Error("Something other than True/False appeared.")
    }
    return arg;
}
```

None: My current implementation doesn't require a "none" literal. However, I have a separate type called "none" that is used as the return type by the functions that don't return anything. So, the possible types are {int, bool, none}.

2. Give an example of a program that uses At least one global variable, At least one function with a parameter, At least one variable defined inside a function.

Python code is:

```
def func(y:int) -> int:
    i:int = 1
```

```
    i = i + a + y
    return i

a:int = 9
y:int = 2
```

Generated Wasm code:

```
(module
    (func $print (import "imports" "print") (param i64) (result i64))
    (import "js" "memory" (memory 1))
    (func $abs (import "imports" "abs") (param i64) (result i64))
    (func $max (import "imports" "max") (param i64) (param i64) (result
i64))
    (func $min (import "imports" "min") (param i64) (param i64) (result
i64))
    (func $pow (import "imports" "pow") (param i64) (param i64) (result
i64))

    (func $func (param $y i64) (result i64)
        (local $localScratchVar i64)
        (local $i i64)
        (i64.const 1)
        (local.set $i)
        (local.get $i)
        (i32.const 8)
        (i64.load)
        (i64.add)
        (local.get $y)
        (i64.add)
        (local.set $i)
        (local.get $i)
        (return)
    )
    (func (export "exported_func") (result i64)
        (i32.const 8) ;; a
        (i64.const 9)
        (i64.store)
        (i32.const 16) ;; y
        (i64.const 2)
        (i64.store)
        (i32.const 0) ;; $scratchVar
        (i32.const 16)
        (i64.load)
        (call $func)
        (i64.store)
        (i32.const 0)
        (i64.load)
```

```
      )
)
```

In my implementation, I store all the global variables inside the memory and my function variables are stored on stack using `local.set`. Therefore, when the function is executed, all the function variables will also be deleted from the stack. In the above program, `a` is a global variable and is accessed inside the function `func`, `y` is a function parameter and `i` is a variable defined inside the function. We can observe from the first 3 lines in `exported_func` that `a` is stored in the memory and is loaded from memory in `$func` at line-6 in the function. The function parameter and variable declared inside the function are not needed once the function is executed. Therefore, I save them on function stack using `local.set`. On the second line of `$func` I declare the function variable `i` and I access the function parameter `y` at line 9 using `local.get`.

I briefly discuss my implementation here.

My ast is as follows:

```
export type Type = "bool" | "int" | "none"

export type Parameter =
    { name: string, type: Type }


export type Stmt =
  | { tag: "define", name: string, value: Expr }
  | { tag: "expr", expr: Expr }
  | { tag: "globals" }
  | {tag: "init", name: string, type: Type, value: Expr}
  | {tag: "if", ifcond: Expr, ifthn: Array<Stmt>, elifcond: Expr, elifthn:
Array<Stmt>, else: Array<Stmt>}
  | {tag: "while", cond: Expr, body: Array<Stmt>}
  | { tag: "funcdef", name: string, decls: Array<Stmt>, parameters:
Array<Parameter>, body: Array<Stmt> , return: Type }
  | { tag: "return", value: Expr }


export type Expr =
    { tag: "literal", value: number, type: Type }
  | {tag: "uniop", value: Expr, name: string}
  | { tag: "id", name: string }
  | { tag: "builtin1", name: string, arg: Expr }
  | { tag: "binop", name: string, arg1: Expr, arg2: Expr}
  | { tag: "builtin2", name: string, arg1: Expr, arg2: Expr}
  | { tag: "call", name: string, arguments: Array<Expr> }
```

The parsed function parameters are part of `Array<Parameter>` inside `funcDef` statement. Since I fill all the new function variables declared in `decls`, the parsed function variables can be found here.

My global environment variable `GlobalEnv` is as follows:

```
export type GlobalEnv = {
  globals: Map<string, number>;
  offset: number;
  types: Map<string, Type>;
  functypes: Map<string, Map<string, Type>>;
  funcDef: Map<string, Stmt>;
  funcStr: string;
  localVars: Set<any>;
}
```

I explain each key inside `GlobalEnv` here.

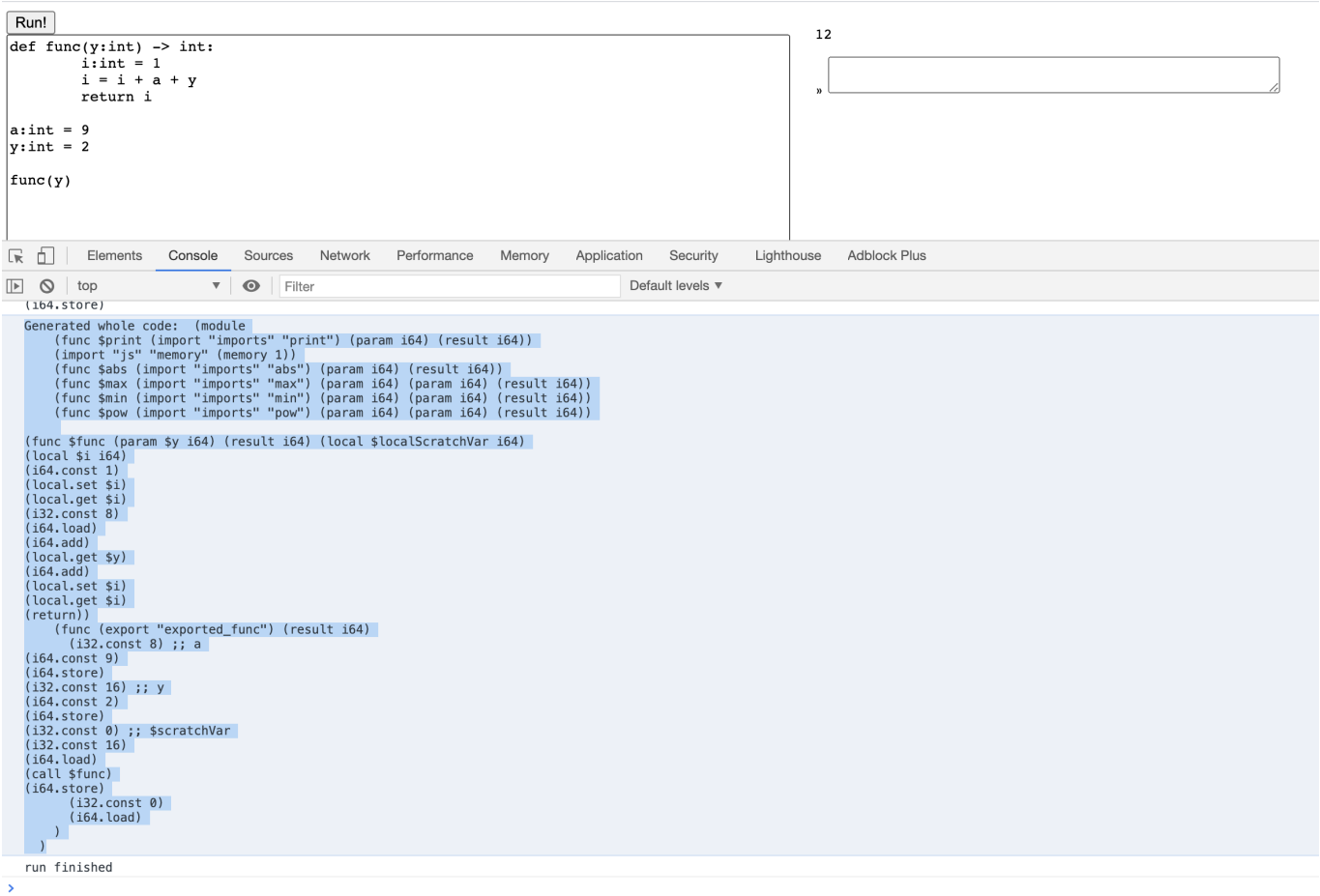
- **globals**: The `globals` is a mapping from global variable name to the memory location in the memory.
- **offset**: `offset` carries the information about the current offset in memory from which empty space starts.
- **types**: `types` is a mapping between variable name and its type. The `types` map is used during typechecking to retrieve the type of a variable. For example, `i = 5`, in this case, I need the type of `i` to check whether `=` is a valid operation.
- **functypes**: `functypes` is a mapping from function name to a types map that is specific to that function. I made this mapping to typecheck statements with function variables. However, I don't use it in my current implementation now because while typechecking a function and its statements, now I just make a copy of `types` in the GlobalEnv and populate it with function variables and its types and use the updated map for typechecking. I don't save it, thus the original `types` doesn't get corrupted and so does `GlobalEnv`.
- **funcDef**: `funcDef` is a map between function name and its ast. This information is used to check whether function returns anything and if it returns, it is used to get the return type of a function. This return type is a useful information while typechecking and code generating. Since a return statement could be anywhere i.e. inside an if-else statement or inside a while statement, saving an ast comes handy to query anything anytime.
- **funcStr**: `funcStr` is the wasm code generated for functions that is passed through multiple runs. This is useful in rendering the functions during REPL.
- **localVars**: `localVars` is a set of local variables in a function. Whenever I generate a code for a variable used in function, my implementation needs to know whether it's a function variable or global variable because function variables are available on stack and global variables are in memory. Therefore, I store this information and query while generating code for function.

I follow the usual pipeline parse-typecheck-codeGen. Since the code for functions is appended at the top separately in wasm code, I separate code for functions first and then global next. So, the pipeline is parse-typecheck-codeGenFunc-codeGen. `codeGenFunc` in turn calls codeGen to generate code for function body.

The function header of `codeGen` and a sample `define` handling is as follows:

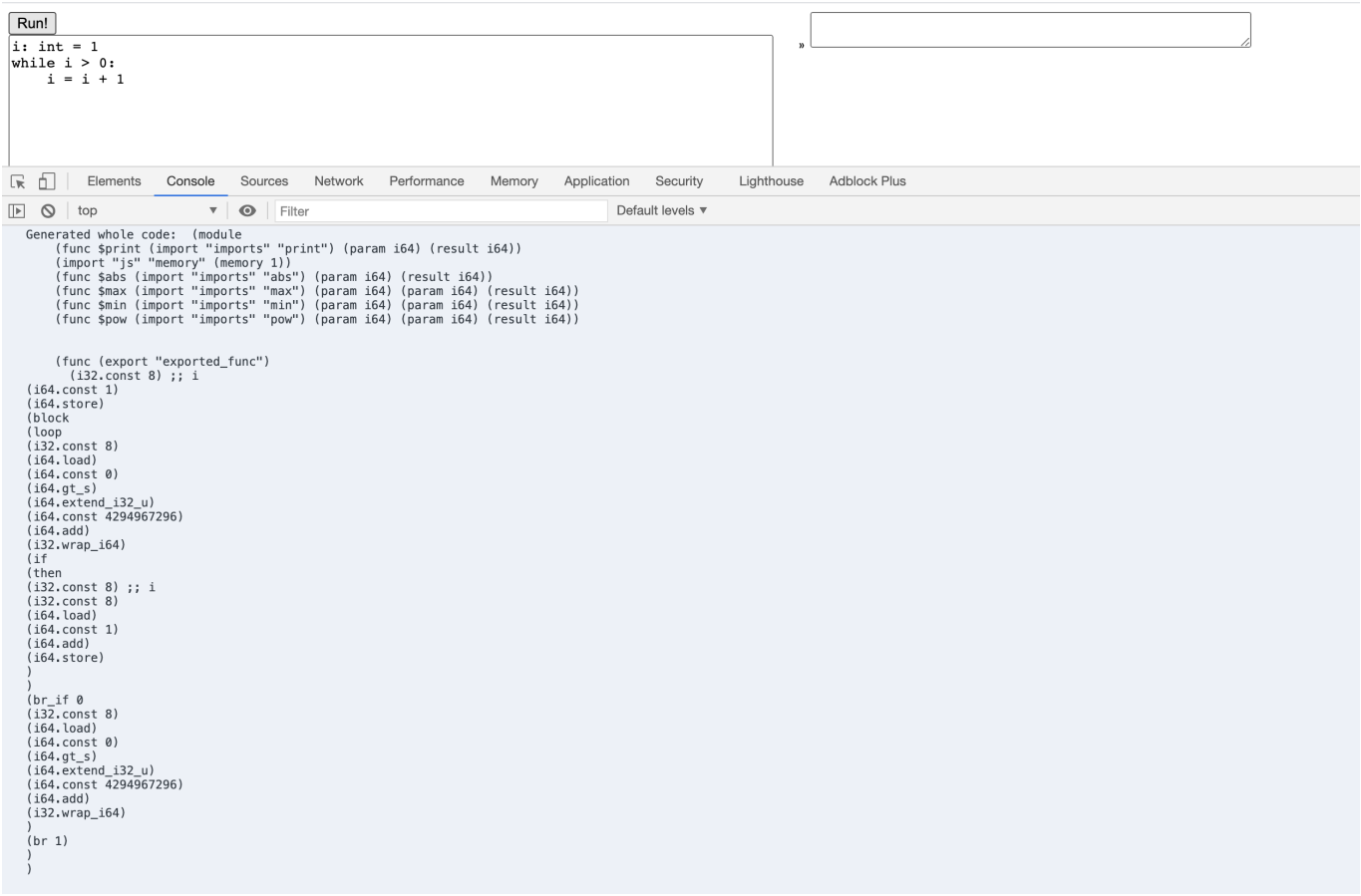
```
function codeGen(stmt: Stmt, env: GlobalEnv, isFunc: boolean = false) :
Array<string> {
  switch(stmt.tag) {
    case "define":
      if (isFunc && isFunctionVar(stmt.name, env)) {
        var valStmts = codeGenExpr(stmt.value, env, isFunc);
        return valStmts.concat([`(local.set $${stmt.name})`]);
      }
      else {
        const locationToStore = [`(i32.const ${envLookup(env, stmt.name)})
;; ${stmt.name}`];
        var valStmts = codeGenExpr(stmt.value, env, isFunc);
        return locationToStore.concat(valStmts).concat([`(i64.store)`]);
      }
  }
}
```

As you can observe, I keep track of the call to `codeGen` whether it's called from a global code or a function code. This is needed because my function variables are on stack and if `codeGen` is called from a function, I look up the variable in `localVars` inside the environment to confirm whether it's a function variable. If I don't find it, then it's a global variable and I use `globals` to obtain the memory location and generate code accordingly.



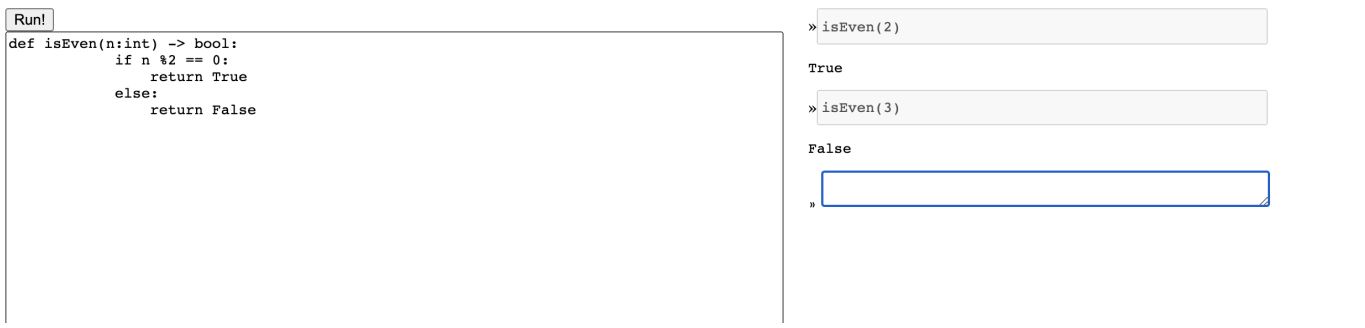
3. Write a Python program that goes into an infinite loop. What happens when you run it on the web page using your compiler?

My pc's fan starts running, the webpage hangs and my browser crashes after a while.



4. Scenarios

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



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

Run!

```
def isEven(n:int) -> bool:
            if n %2 == 0:
                return True
            else:
                return False
```

```
def isOdd(n:int) -> bool:
            if isEven(n):
                return False
            else:
                return True
```

»

Run!

```
def isEven(n:int) -> bool:
            if n %2 == 0:
                return True
            else:
                return False
```

» def isOdd(n:int) -> bool: if isEven(n):

» isOdd(5)

True

» isOdd(6)

False

» isEven(3)

False

» isEven(4)

True

»

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

Run!

```
i:int = 0
j:bool = True
k:int = 5
k = i + j
```

Cannot apply operator `+` on types `int` and `bool`

»

A program that has a type error in a conditional position

Run!

```
i:int = 0
j:bool = True
if i == j:
    print(45)
```

Cannot apply operator `==` on types `int` and `bool`

»

A program that calls a function from within a loop

Run!

```
def isEven(n:int) -> bool:
    if n %2 == 0:
        return True
    else:
        return False

i: int = 0
while i < 10:
    print(isEven(i))
    i = i + 1
```

```
True
False
True
False
True
False
True
False
True
False
»
```

Printing an integer and a boolean

Run!

```
print(5 == 2)
print(5 == 5)
print(98)
```

```
False
True
98
98
»
```

A recursive function.

Run!

```
def Fibonacci(n:int) -> int:
    if n==0:
        return 0
    elif n==1:
        return 1
    else:
        return Fibonacci(n-1)+Fibonacci(n-2)
```

```
» Fibonacci(9)
34
» Fibonacci(5)
5
»
```

Two mutually-recursive functions.

Adapted from: http://www.idc-online.com/technical_references/pdfs/information_technology/Mutual_Recursion_in_Python.pdf

Run!

```
def isEven(n:int) -> bool:
    if n %2 == 0:
        return True
    else:
        return isOdd(n-1)


def isOdd(n:int) -> bool:
    if n%2 == 0:
        return False
    else:
        return isEven(n-1)
```

```
» isOdd(5)
True
» isOdd(6)
False
» isEven(2)
True
» isEven(3)
False
»
```

Collaborators: I disucssed with Amanda, Edwin, Hema while doing this assignment.