PA2

A59011251 Hsin-Yuan Chen

1. In my implementation, I leave the right most 2 bits to represent True/False and None. None is store as 1 in my implementation, and I choose True's value to be 3 and False's value to be 2, which makes it very easy to implement and operator and or operator because the behavior of these two functions are aligned with behaviors in i32 functions. To deal with the number, I will left shift 2 of the real value in WASM, and I will right shift 2 when I need to get the real value.

Run!

```
True
```

```
(module
  (func $print_num (import "imports" "print_num") (param i32) (result i32)
  (func $print_bool (import "imports" "print_bool") (param i32) (result i3
  (func $print_none (import "imports" "print_none") (param i32) (result i3

  (func (export "_start") (result i32)
    (local $scratch i32)
(i32.const 3)
(local.set $scratch)
    (local.get $scratch)
  )
)
```

```
Result: True
```

Run!

```
15
```

```
(module
  (func $print_num (import "imports" "print_num") (param i32) (result i32))
  (func $print_bool (import "imports" "print_bool") (param i32) (result i32))
  (func $print_none (import "imports" "print_none") (param i32) (result i32))

  (func (export "_start") (result i32)
    (local $scratch i32)
(i32.const 60)
(local.set $scratch)
    (local.get $scratch)
  )
)
```

```
Result:15
```

Run!

```
None
```

```
(module
  (func $print_num (import "imports" "print_num") (param i32) (result i32))
  (func $print_bool (import "imports" "print_bool") (param i32) (result i32))
  (func $print_none (import "imports" "print_none") (param i32) (result i32))

  (func (export "_start") (result i32)
    (local $scratch i32)
(i32.const 1)
(local.set $scratch)
    (local.get $scratch)
  )
)
```

```
Result: None
```

Run!

```
True and False
```

```
(module
  (func $print_num (import "imports" "print_num") (param i32) (result i32))
  (func $print_bool (import "imports" "print_bool") (param i32) (result i32))
  (func $print_none (import "imports" "print_none") (param i32) (result i32))

  (func (export "_start") (result i32)
    (local $scratch i32)
(i32.const 3)
(i32.const 2)
(i32.and)
  (i32.const 2)
  (i32.or)

(local.set $scratch)
    (local.get $scratch)
  )
)
```

```
Result: False
```

2.

```
return `
  (module
    (func $print_num (import "imports" "print_num") (param i32) (result i32))
    (func $print_bool (import "imports" "print_bool") (param i32) (result i32))
    (func $print_none (import "imports" "print_none") (param i32) (result i32))
    ${varDecls}
    ${allFuns}
    (func (export "_start") ${retType}
      ${main}
      ${retVal}
    )
  )
`;
```

The WASM codes will be generated from the above, where varDecls will declare global variables(Figure1), and allFuns will have the function definition WASM codes where includes the declaration of both parameter and variable defined inside functions.

```
export function compile(source : string) : string {
  let ast = parseProgram(source);
  ast = tcProgram(ast);
  const emptyEnv = new Map<string, boolean>();
  const [vars, funs, stmts] = varsFunsStmts(ast);
  const funsCode : string[] = funs.map(f => codeGenStmt(f, emptyEnv)).map(f => f.join("\n"));
  const allFuns = funsCode.join("\n\n");
  const varDecls = vars.map(v => `(global $${v} (mut i32) (i32.const 0))`).join("\n");

  const allStmts = stmts.map(s => codeGenStmt(s, emptyEnv)).flat();
```

Figure 1

```
function variableNames(stmts: Stmt<Type>[]) : string[] {
  const vars : Array<string> = [];
  stmts.forEach((stmt) => {
    if(stmt.tag === "var") { vars.push(stmt.name); }
  });
  return vars;
}
```

The underlined part of the code define the global variables in WASM. “vars” come from the return value of variableNames() which consists of all statements whose tag is “var” in the first level statements, and the program will map each vars to WASM code for generating global variables. The global variables are stored in global in WASM.

```
export function codeGenStmt(stmt : Stmt<Type>, locals : Env) : Array<string> {
  console.log(stmt)
  switch(stmt.tag) {
    case "define":
      const withParamsAndVariables = new Map<string, boolean>(locals.entries());

      // Construct the environment for the function body
      const variables = variableNames(stmt.body);
      variables.forEach(v => withParamsAndVariables.set(v, true));
      stmt.params.forEach(p => withParamsAndVariables.set(p.name, true));

      // Construct the code for params and variable declarations in the body
      const params = stmt.params.map(p => `(param $${p.name} i32)`).join(" ");
      const varDecls = variables.map(v => `(local $${v} i32)`).join("\n");

      const stmts = stmt.body.map(s => codeGenStmt(s, withParamsAndVariables)).flat();
      const stmtsBody = stmts.join("\n");
      return [`(func $${stmt.name} ${params} (result i32)
        (local $scratch i32)
        ${varDecls}
        ${stmtsBody}
        (i32.const 0))`];
```

Figure 2

Both parameters and variables defined inside functions are generated in the codeGenStmt function. The red line in Figure2 shows how to make params stored as param in WASM. The blue line in Figure2 shows that variables defined inside a functions are stored as local in WASM.

Run!

```
a:int = 3
def haha(b:int):
  c:int=5
  print(a+b+c)
haha(4)
```

```
    (module
      (func $print_num (import "imports" "print_num") (param i32) (result i32))
      (func $print_bool (import "imports" "print_bool") (param i32) (result i32))
      (func $print_none (import "imports" "print_none") (param i32) (result i32))
      (global $a (mut i32) (i32.const 0))
      (func $haha (param $b i32) (result i32)
        (local $scratch i32)
        (local $c i32)
        (i32.const 20)
(local.set $c)
(global.get $a)
(local.get $b)
(i32.add)
(local.get $c)
(i32.add)
(call $print_num)
(local.set $scratch)
        (i32.const 0))
      (func (export "_start") (result i32)
        (local $scratch i32)
```

3. The web page got stuck in the infinite loop and the print statement seems to malfunction. My computer got stuck because of this, and I am forced to close to page because of this.

Run!

```
a:int=2
while a>0:
  a = a+1
  print(a)
```

```
    (module
      (func $print_num (import "imports" "print_num") (param i32) (result i32))
      (func $print_bool (import "imports" "print_bool") (param i32) (result i32))
      (func $print_none (import "imports" "print_none") (param i32) (result i32))
      (global $a (mut i32) (i32.const 0))

      (func (export "_start")
        (local $scratch i32)
(i32.const 8)
(global.set $a)

        (loop $my_loop
          (global.get $a)
(i32.const 0)
(i32.gt_s)
    (i32.const 2)
    (i32.or)

          i32.const 3
          i32.eq
          (if
            (then
              (global.get $a)
(i32.const 4)
(i32.add)
(global.set $a)
(global.get $a)
(call $print_num)
(local.set $scratch)
              br $my_loop
            )
          )
        )

      )
    )
```

4.

(4-1)

Run!

```
def haha(a:int)->int:
  return a+1
print(haha(2)+true)
```

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

(4-2)

Run!

```
def haha(x:int):
  while(x):
    print(x)
    x=x-1
haha(2)
```

```
Error: Condition expression cannot be type of int
```

(4-3)

Run!

```
def haha(x:int):
  while(x>0):
    cool(x)
    x=x-1
def cool(x:int):
  print(x)
haha(2)
```

```
    (module
      (func $print_num (import "imports" "print_num") (param i32) (result i32))
      (func $print_bool (import "imports" "print_bool") (param i32) (result i32))
      (func $print_none (import "imports" "print_none") (param i32) (result i32))

      (func $haha (param $x i32) (result i32)
        (local $scratch i32)


        (loop $my_loop
          (local.get $x)
(i32.const 0)
(i32.gt_s)
    (i32.const 2)
    (i32.or)

          i32.const 3
          i32.eq
          (if
            (then
              (local.get $x)
(call $cool)
(local.set $scratch)
(local.get $x)
(i32.const 4)
(i32.sub)
(local.set $x)
              br $my_loop
            )
          )
        )
        (i32.const 0))

(func $cool (param $x i32) (result i32)
        (local $scratch i32)

        (local.get $x)
(call $print_num)
(local.set $scratch)
        (i32.const 0))
      (func (export "_start") (result i32)
        (local $scratch i32)
(i32.const 8)
(call $haha)
(local.set $scratch)
        (local.get $scratch)
      )
    )
```

```
2 1 Result:0
```

(4-4)

```
def haha(x:int)->int:
  while (x>0):
    if x==1:
      return x
    x = x-1
    return x
print(haha(3))
```

Run!

```
    (module
      (func $print_num (import "imports" "print_num") (param i32) (result i32))
      (func $print_bool (import "imports" "print_bool") (param i32) (result i32))
      (func $print_none (import "imports" "print_none") (param i32) (result i32))

      (func $haha (param $x i32) (result i32)
        (local $scratch i32)


        (loop $my_loop
          (local.get $x)
(i32.const 0)
(i32.gt_s)
    (i32.const 2)
    (i32.or)

          i32.const 3
          i32.eq
          (if
            (then

        (local.get $x)
(i32.const 4)
(i32.eq)
    (i32.const 2)
    (i32.or)

        i32.const 3
        i32.eq
        (if
          (then
            (local.get $x)
return
          )


        )

(local.get $x)
(i32.const 4)
(i32.sub)
(local.set $x)
(local.get $x)
return
              br $my_loop
            )
          )
        )
        (i32.const 0))
      (func (export "_start") (result i32)
        (local $scratch i32)
(i32.const 12)
(call $haha)
(call $print_num)
(local.set $scratch)
        (local.get $scratch)
      )
    )


2 Result:2
```

(4-5)

Run!

```
a:int=5
b:bool=true
print(a)
print(b)
```

```
    (module
      (func $print_num (import "imports" "print_num") (param i32) (result i32))
      (func $print_bool (import "imports" "print_bool") (param i32) (result i32))
      (func $print_none (import "imports" "print_none") (param i32) (result i32))
      (global $a (mut i32) (i32.const 0))
(global $b (mut i32) (i32.const 0))

      (func (export "_start") (result i32)
        (local $scratch i32)
(i32.const 20)
(global.set $a)
(i32.const 3)
(global.set $b)
(global.get $a)
(call $print_num)
(local.set $scratch)
(global.get $b)
(call $print_bool)
(local.set $scratch)
        (local.get $scratch)
      )
    )
```

```
5 True Result: True
```

(4-6)

```
def fib(x:int)->int:
  if x==0:
    return 0
  elif x==1:
    return 1
  else:
    return fib(x-1)+fib(x-2)
print(fib(6))
```

Run!

```
    (module
      (func $print_num (import "imports" "print_num") (param i32) (result i32))
      (func $print_bool (import "imports" "print_bool") (param i32) (result i32))
      (func $print_none (import "imports" "print_none") (param i32) (result i32))

      (func $fib (param $x i32) (result i32)
        (local $scratch i32)


        (local.get $x)
(i32.const 0)
(i32.eq)
    (i32.const 2)
    (i32.or)

        i32.const 3
        i32.eq
        (if
          (then
            (i32.const 0)
return
          )

            (else
              (local.get $x)
(i32.const 4)
(i32.eq)
    (i32.const 2)
    (i32.or)

              i32.const 3
              i32.eq
              (if
                (then
                  (i32.const 4)
return
                )

               (else
                (local.get $x)
(i32.const 4)
(i32.sub)
(call $fib)
(local.get $x)
(i32.const 8)
(i32.sub)
(call $fib)
(i32.add)
return
               )

              )
            )


        )

        (i32.const 0))
      (func (export "_start") (result i32)
        (local $scratch i32)
(i32.const 24)
(call $fib)
(call $print_num)
(local.set $scratch)
        (local.get $scratch)
      )
    )


8 Result:8
```

(4-7)

Run!

```
def printOdd(x:int):
  print(x)
  printEven(x-1)
def printEven(x:int):
  print(x)
  if x!=0:
    printOdd(x-1)
printOdd(5)
```

```
    (module
      (func $print_num (import "imports" "print_num") (param i32) (result i32))
      (func $print_bool (import "imports" "print_bool") (param i32) (result i32))
      (func $print_none (import "imports" "print_none") (param i32) (result i32))

      (func $printOdd (param $x i32) (result i32)
        (local $scratch i32)

        (local.get $x)
(call $print_num)
(local.set $scratch)
(local.get $x)
(i32.const 4)
(i32.sub)
(call $printEven)
(local.set $scratch)
        (i32.const 0))

(func $printEven (param $x i32) (result i32)
        (local $scratch i32)

        (local.get $x)
(call $print_num)
(local.set $scratch)

        (local.get $x)
(i32.const 0)
(i32.ne)
    (i32.const 2)
    (i32.or)

        i32.const 3
        i32.eq
        (if
          (then
            (local.get $x)
(i32.const 4)
(i32.sub)
(call $printOdd)
(local.set $scratch)
          )


        )

        (i32.const 0))
      (func (export "_start") (result i32)
        (local $scratch i32)
(i32.const 20)
(call $printOdd)
(local.set $scratch)
        (local.get $scratch)
      )
    )


5 4 3 2 1 0 Result:0
```

5. I would like to use (2). The error occurs in the condition expression of the while loop during type checking, and the expression type should be bool instead of int.

```
export function tcStmt(s : Stmt<any>, functions : FunctionsEnv, variables : BodyEnv, currentRe
  switch(s.tag) {
    case "while": {
      const cond = tcExpr(s.cond, functions, variables);
      if (cond.a!=="bool") throw new Error("Condition expression cannot be type of "+cond.a);
      const stmtBody = s.stmtBody.map(bs => tcStmt(bs, functions, variables, currentReturn));
      return {...s, cond, stmtBody};
    }
```