

CSE231 Project Assignment 2

Chao-Te Chou (c3chou@ucsd.edu)

Due: April 13 11:59 PM

1 Question 1

Representation of values:

- integers: they are represented in their original value and are saved in literal `{a?: A, tag: "num", value: number}`.
- boolean: In the parser and the parsed program, boolean values are represented in their original format using literal `({a?: A, tag: "bool", value: boolean})`. When compiling the program, "True" is compiled to be 1 and "False" is compiled to be 0.
- None: In the parser and the parsed program, None is represented as a literal `({a?: A, tag: "none"})`. When compiling, it is represented with the value 0.

When translating the code to WASM, all the types are translated into integers according to the rules above. Thus, when compiling the program, I should assign different print function for different type. That is I have `print_num` to print integer, `print_bool` to print boolean value and `print_none` to print None. Those print functions are implemented in JavaScript and customized to print different type according to the given numerical values. Part of the codes for printing boolean values is shown below.

```
1 print_bool: (arg : any) => {
2     if(arg === 0) { display("False"); }
3     else { display("True"); }
4     return arg;
5 }
```

Listing 1: The print function in wenstart.ts to print boolean

```
1 if(expr.name === 'print') {
2
3     switch(expr.args[0].a) {
4         case Type.int:
5             codes.push('(call $print_num)');
6             break;
7         case Type.bool:
8             codes.push('(call $print_bool)');
9             break;
10        case Type.none:
11            codes.push('(call $print_none)');
12            break;
13        default:
14            throw new Error("COMPILE ERROR: unknow literal type");
15    }
16 } else {
17     codes.push('(call $$ {expr.name})');
18 }
```

Listing 2: The code to decide which print functions to use.

2 Question 2

The code according to the specification of the question is shown below. The result is shown in Fig. 1.

```
1 x: int = 1
2 def add2(a: int) -> int:
3     b: int = 2
4     return a + b
5
6 x = add2(x)
7 x
```

Listing 3: Code for Question2.

Let's first show the data structure for the parsed program and the function definition.

```
1 export type Program<A> = {a?: A,
2                             varInits: VarInit<A>[],
3                             funcDefs: FuncDef<A>[],
4                             stmts: Stmt<A>[]}
```

Listing 4: The data structure for the parsed program.

```
1 export type FuncDef<A> = {a?: A,
2                             name: string,
3                             params: TypedVar<A>[],
4                             retType: Type,
5                             varInits: VarInit<A>[],
6                             stmts: Stmt<A>[]}
```

Listing 5: The data structure for the function definitions.

parsing. A program stores the global variables (varInits), the function definitions (funcDefs) and the main body (stmts). A funcDef stores the parameters (params), the function name (name), the return type (retType), the local variables (varInits) and the function body (stmts). Thus, x, which is the global variable, is stored in program.varInits. The add2 function is stored in program.funcDefs. b, which is a local variable, is stored in funcDef.varInits.

compiling. When compiling the function call. We pass the global variable list and the function definitions into the function, which compiles the function call. In that function, it compiles the code line by line and decide whether a given variable is a global one or a local one to decide whether to initialize the variable and get it or directly get it from the global variables. Part of the code is shown below.

```
1 export type GlobalEnv = {
2     vars: Map<string, VarInit<Type>>,
3     funcs: Map<string, FuncDef<Type>>,
4     loopDepth: number
5 }
6 function codeGenCall(expr: Expr<Type>, globalEnv: GlobalEnv): string[] {
7     if(expr.tag !== 'call') {
8         throw new Error("COMPILER ERROR: the input expression to codeGenCall should be call");
9     }
10
11     let codes: string[] = [];
12
13     // collect arguments
14     for(let idx = 0; idx < expr.args.length; ++idx) {
15         const arg = expr.args[idx];
16         codes = [...codeGenExpr(arg, globalEnv), ...codes];
17     }
18
19     // call the function
20     if(expr.name === 'print') {
21
22         switch(expr.args[0].a) {
23             case Type.int:
24                 codes.push('(call $print_num)');
25                 break;
26             case Type.bool:
27                 codes.push('(call $print_bool)');
28                 break;
29             case Type.none:
30                 codes.push('(call $print_none)');
31                 break;
```

```

x: int = 1
def add2(a: int) -> int:
    b: int = 2
    return a + b

x = add2(x)
x

(global $x (mut i32) (i32.const 1))
(func $add2 (param $a i32) (result i32)
(local $scratch i32)
(local $b i32)
(local.set $b (i32.const 2))
(local.get $a)
(local.get $b)
(i32.add)
return
(i32.const 0))
(func (export "exported_func") (result i32)(local $$last i32)
(local $x i32)
(global.get $x)
(call $add2)
(global.set $x)
(global.get $x)
(local.set $$last)(local.get $$last))
3

```

Figure 1: Result of Questions.

```

32     default:
33         throw new Error("COMPILE ERROR: unknow literal type");
34     }
35 } else {
36     codes.push('(call ${expr.name})');
37 }
38
39
40 return codes;
41 }

```

Listing 6: The code to compile a function call for Question 2.

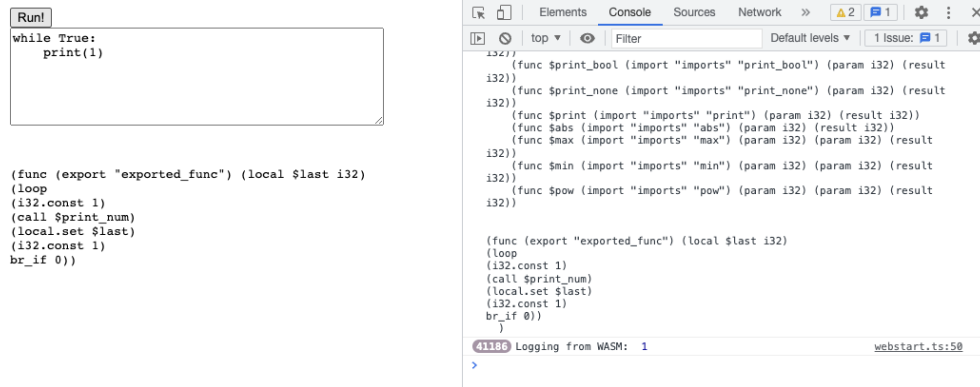


Figure 2: **Question 3.** Infinite loop.

3 Question 3

Fig. 2 shows the code, the compiled WASM and the developer windows when I run the infinite loop. When running the infinite loop with a print function inside, the number in the light brown oval keep increasing, the browser is slow and it takes time for the browser to respond to some of my request (e.g. closing).

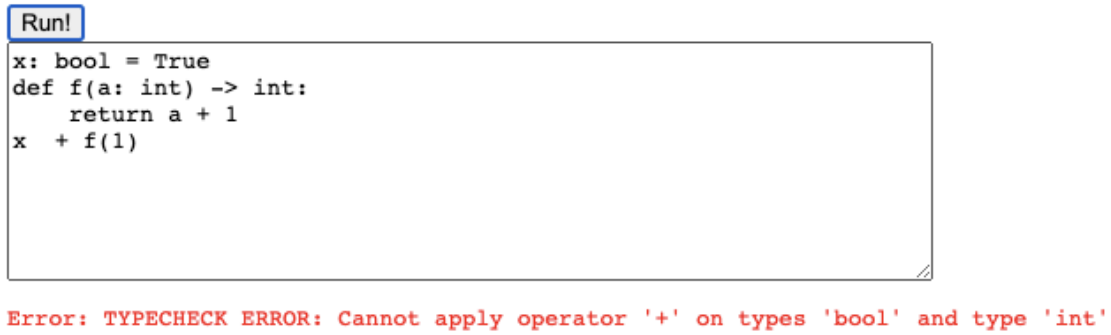


Figure 3: Question 4.1.

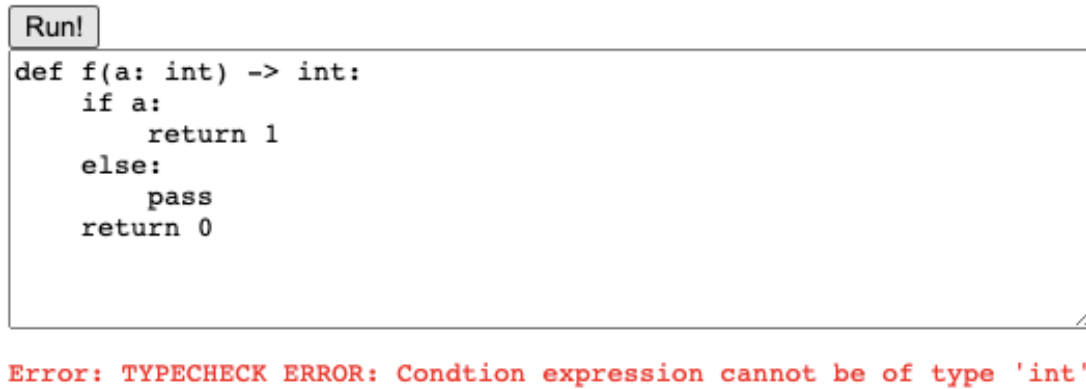


Figure 4: Question 4.2.

4 Question 4

4.1 Question 4.1

Please refer to Fig. 3

4.2 Question 4.2

Please refer to Fig. 4

Run!

```
x: int = 0
while x < 4:
    print(x)
    x = x + 1
```

```
(global $x (mut i32) (i32.const 0))

(func (export "exported_func") (local $last i32)
(loop
(global.get $x)
(call $print_num)
(local.set $last)
(global.get $x)
(i32.const 1)
(i32.add)
(global.set $x)
(global.get $x)
(i32.const 4)
(i32.lt_s)
br_if 0))

0

1

2

3
```

Figure 5: Question 4.3.

4.3 Question 4.3

Please refer to Fig. 5

Run!

```

def f() -> bool:
    x: int = 0
    while x < 4:
        if x == 2:
            return True
        x = x + 1
    return False

r: bool = False
r = f()
print(r)

```

```

(global $r (mut i32) (i32.const 0))
(func $f (result i32)
(local $last i32)
(local $x i32)
(local.set $x (i32.const 0))
(loop
(local.get $x)
(i32.const 2)
(i32.eq)
(if
(then
(i32.const 1)
return
)
else
(i32.const 0)
(if
(then
nop
)
else
nop
))))
(local.get $x)
(i32.const 1)
(i32.add)
(local.set $x)
(local.get $x)
(i32.const 4)
(i32.lt_s)
br_if 0)
(i32.const 0)
return
(i32.const 0))
(func (export "exported_func") (result i32)(local $last i32)
(local $r i32)
(call $f)
(global.set $r)
(global.get $r)
(call $print_bool)
(local.set $last)(local.get $last))

```

True1

Figure 6: Question 4.4.

4.4 Question 4.4

Please refer to Fig. 6

Run!

```
print(4)
print(True)
```

```
(func (export "exported_func") (result i32)(local $last i32)
(i32.const 4)
(call $print_num)
(local.set $last)
(i32.const 1)
(call $print_bool)
(local.set $last)(local.get $last))

4True1
```

Figure 7: Question 4.5.

4.5 Question 4.5

Please refer to Fig. 7

Run!

```

def f(x: int):
    print(x)
    if x == 0:
        return
    x = x - 1
    f(x)
f(4)

```

```

(func $f (param $x i32) (result i32)
(local $last i32)

(local.get $x)
(call $print_num)
(local.set $last)
(local.get $x)
(i32.const 0)
(i32.eq)
(if
(then
(i32.const 0)
return
)
else
(i32.const 0)
(if
(then
nop
)
else
nop
))))
(local.get $x)
(i32.const 1)
(i32.sub)
(local.set $x)
(local.get $x)
(call $f)
(local.set $last)
(i32.const 0)
(func (export "exported_func") (result i32)(local $last i32)
(i32.const 4)
(call $f)
(local.set $last)(local.get $last))

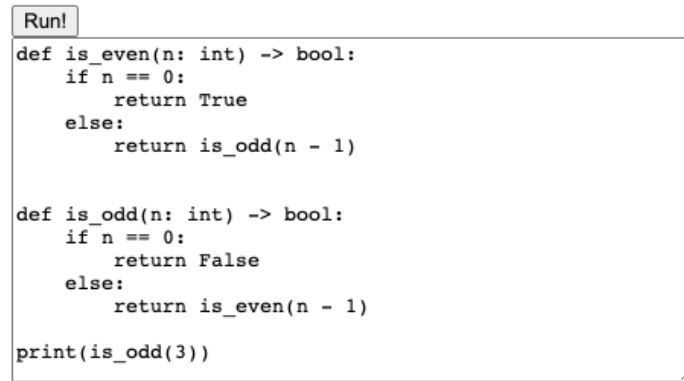
432100

```

Figure 8: Question 4.6.

4.6 Question 4.6

Please refer to Fig. 8



```
def is_even(n: int) -> bool:
    if n == 0:
        return True
    else:
        return is_odd(n - 1)

def is_odd(n: int) -> bool:
    if n == 0:
        return False
    else:
        return is_even(n - 1)

print(is_odd(3))
```

Figure 9: Question 4.7.1.

4.7 Question 4.7

Please refer to Fig. 9 and Fig. 10.

5 Question 5

I will choose Question 4.1. I show part of the code for type checking expressions, statements and binary operations below.

```
1 export function typeCheckStmts(stmts: Stmt<null>[], env: TypeEnv): Stmt<Type>[] {
2   const typedStmts: Stmt<Type>[] = [];
3
4   stmts.forEach(stmt => {
5     switch(stmt.tag) {
6       case "assign": // e.g. a = 0
7         if(!env.vars.has(stmt.name)) { // the variable should have been initied
8           throw new Error('TYPE ERROR: Not a variable ${stmt.name}');
9         }
10        const typedValue = typeCheckExpr(stmt.value, env);
11        if(typedValue.a !== env.vars.get(stmt.name)) {
12          throw new Error('TYPE ERROR: Expected type ${env.vars.get(stmt.name)}; got type ${
typedValue.a}');
13        }
14
15        typedStmts.push({...stmt, a: Type.none, value: typedValue});
16        break;
17      case "expr" :
18        const typedExpr = typeCheckExpr(stmt.expr, env);
19        typedStmts.push({...stmt, expr: typedExpr, a: Type.none});
20        break;
21      case "return":
22        const typedRet = typeCheckExpr(stmt.expr, env);
23        if(typedRet.a !== env.retType) {
24          throw new Error('TYPE ERROR: return expected type ${env.retType}; got type ${
typedRet.a}');
25        }
26        typedStmts.push({...stmt, expr: typedRet, a: typedRet.a});
27        break;
28      case "pass" :
29        typedStmts.push({...stmt, a: Type.none});
30        break;
31      case "while":
32        const typedWhile = typeCheckWhile(stmt, env);
33        typedStmts.push({...typedWhile, a: Type.none});
34        break;
35      case "if":
36        const typedIf = typeCheckIf(stmt, env);
37        typedStmts.push({...typedIf, a: Type.none});
38        break;
39    }
40  });
41  return typedStmts;
```

Listing 7: Type Check Statement.

```
1 export function typeCheckExpr(expr: Expr<null>, env: TypeEnv): Expr<Type> {
2   switch (expr.tag) {
3     case "id": // if the variable has been defined
4       if(!env.vars.has(expr.name)){
5         throw new Error('TYPE ERROR: not a variable ${expr.name}');
6       }
7       const idType = env.vars.get(expr.name);
8       return {...expr, a: idType};
9     case "binop" :
10      return typeCheckBinOp(expr, env);
11    case "uniop":
12      return typeCheckUniOp(expr, env);
13    case "literal" :
14      return {...expr, a: typeCheckLiteral(expr.literal).a}
15    case "call":
16      const typedCall = typeCheckCall(expr, env);
17      return typedCall
18  }
19 }
```

Listing 8: Type Check Expression.

```
1 export function typeCheckBinOp(expr: Expr<null>, env: TypeEnv): Expr<Type> {
2   if(expr.tag !== "binop") {
3     throw new Error("TYPECHECK ERROR: typeCheckBinOp only take binary operation");
4   }
5 }
```

```

5
6  switch(expr.op) {
7      case BinOp.Plus : // work for int
8      case BinOp.Minus:
9      case BinOp.Mul   :
10     case BinOp.Div   :
11     case BinOp.Mod   :
12     case BinOp.Seq   :
13     case BinOp.Leq   :
14     case BinOp.Sml   :
15     case BinOp.Lrg   :
16         const leftTyped = typeCheckExpr(expr.left, env);
17         const rightTyped = typeCheckExpr(expr.right, env);
18         if(leftTyped.a !== rightTyped.a || (leftTyped.a !== Type.int || rightTyped.a !== Type.int))
19     {
20         throw new Error('TYPECHECK ERROR: Cannot apply operator \''{expr.op}\'' on types \''{
leftTyped.a}\' and type \''{rightTyped.a}\'');
21     }
22     if(expr.op === BinOp.Seq || expr.op === BinOp.Leq || expr.op === BinOp.Sml || expr.op ===
BinOp.Lrg) {
23         return {...expr, left: leftTyped, right:rightTyped, a: Type.bool};
24     }
25     return {...expr, left: leftTyped, right:rightTyped, a: Type.int};
26     case BinOp.Eq    : // wrk fpr both int and bool
27     case BinOp.Neq   :
28         const leftTypedEq = typeCheckExpr(expr.left, env);
29         const rightTypedEq = typeCheckExpr(expr.right, env);
30         if(leftTypedEq.a !== rightTypedEq.a ) {
31             throw new Error('TYPECHECK ERROR: Cannot apply operator \''{expr.op}\'' on types \''{
leftTypedEq.a}\' and type \''{rightTypedEq.a}\'');
32         }
33         return {...expr, left: leftTypedEq, right: rightTypedEq, a: Type.bool}
34     case BinOp.Is    : // work for none
35         const leftTypedNone = typeCheckExpr(expr.left, env);
36         const rightTypedNone = typeCheckExpr(expr.right, env);
37         if(leftTypedNone.a !== Type.none || rightTypedNone.a !== Type.none) {
38             throw new Error('TYPECHECK ERROR: Cannot apply operator \''{expr.op}\'' on types \''{
leftTypedNone.a}\' and type \''{rightTypedNone.a}\'');
39         }
40         return {...expr, left: leftTypedNone, right: rightTypedNone, a: Type.bool}
41     }
}

```

Listing 9: Type Check Binary Operation.

The error message is thrown when we are checking the binary operation ('+'). We first type check its left expression, which has type bool. Then, we type check its right expression, which has type int. Since the plus operator can only operate on number, the error is thrown.

```

(func $is_even (param $n i32) (result i32)
(local $last i32)

(local.get $n)
(i32.const 0)
(i32.eq)
(if
(then
(i32.const 1)
return
)
else
(i32.const 0)
(if
(then
nop
)
else
(local.get $n)
(i32.const 1)
(i32.sub)
(call $is_odd)
return
))))
(i32.const 0))
(func $is_odd (param $n i32) (result i32)
(local $last i32)

(local.get $n)
(i32.const 0)
(i32.eq)
(if
(then
(i32.const 0)
return
)
else
(i32.const 0)
(if
(then
nop
)
else
(local.get $n)
(i32.const 1)
(i32.sub)
(call $is_even)
return
))))
(i32.const 0))
(func (export "exported_func") (result i32)(local $last i32)
(i32.const 3)
(call $is_odd)
(call $print_bool)
(local.set $last)(local.get $last))

True1

```

Figure 10: Question 4.7.2.