Programming Assignment-2.5

Submission 61741567

1. For each of the compilers you are reviewing, choose two programs that run successfully

Program-1:

```
i:int = 4
while i > 0:
    i = i - 1
print(i)
```

The ast that is required for while-statement is mentioned below. Note that, this is not the complete ast.

```
export type Stmt = { tag: "while", expr: Expr, stmts: Array<Stmt>}

export type Expr = { tag: "literal", value: Literal} | { tag: "id", name:
string } | { tag: "binop", expr1: Expr, op: Op, expr2: Expr}

export type Op =
    {tag: "add"}
  | {tag: "sub"}
  | {tag: "mul"}
  | {tag: "div_s"}
  | {tag: "rem_s"}
  | {tag: "eq"}
  | {tag: "ne"}
  | {tag: "le_s"}
  | {tag: "ge_s"}
  | {tag: "lt_s"}
  | {tag: "gt_s"}
  | {tag: "is"}
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" }
```

The parser parses it and adds into the ast of while statement. The relevant snippet of parser is attached below:

```
case "WhileStatement":
      c.firstChild(); // focus on while
      c.nextSibling(); // focus on while cond
      const whileExpr = traverseExpr(c, s);
      c.nextSibling(); // focus on body
      c.firstChild(); // focus on :

      var whileStmts = [];
      while (c.nextSibling()) {
        console.log(traverseStmt(c,s))
        whileStmts.push(traverseStmt(c, s));
      }
      c.parent()  // pop up to body
      c.parent() // pop to while
      return {
        tag: "while",
        expr: whileExpr,
        stmts: whileStmts
      }
```

They implement typechecking as a step in the `codeGenExpr`. The code flow is as follows: `codeGen` -> `codeGenExpr` for condition expression in while -> recursively call `codeGen` for statements inside while.

The `codeGen` code corresponding to while is as follows:

```
case "while":
      var wCond = codeGenExpr(stmt.expr, env);
      var condStmts: string[] = []
      condStmts.push(wCond.join("\n"));
      condStmts.push(`(i64.const ${TRUE}) \n (i64.ne)\n`) // Only
necessary when it's actually True false in cond

      var exprStmts: string[] = [];
      //console.log(stmt.stmts);
      stmt.stmts.forEach(st => exprStmts.push(codeGen(st,
env).join("\n")));
      //console.log(exprStmts);

      let whileStmts = `(block\n (loop \n ${condStmts.join("\n")} (br_if
1) ${exprStmts.join("\n")} (br 0)) )`
      return [whileStmts]
```

The codeGenExpr performs type checking first and generate code based on expression types. The code for codeGenExpr is as follows:

```
function codeGenExpr(expr: Expr, env: GlobalEnv): Array<string> {
  switch (expr.tag) {
    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
      }
    case "literal":
      const val = expr.value
      switch (val.tag) {
        case "None":
          return [`(i64.const ${NONE})`]
        case "number":
          return ["(i64.const " + val.value + ")"];
        case "False":
          return [`(i64.const ${FALSE})`]
        case "True":
          return [`(i64.const ${TRUE})`]
      }
    // Cases for binary operation and bultin2
    case "binop":
      checkTypeOp(expr.expr1, expr.op, "left side", env)
      checkTypeOp(expr.expr2, expr.op, "right side", env)
      var stmts = codeGenExpr(expr.expr1, env);
      //const stmts2 = codeGenExpr(expr.expr2)
      stmts = stmts.concat(codeGenExpr(expr.expr2, env))
      stmts = stmts.concat(["(i64." + expr.op.tag + ")"])
      // If result is int don't need to signextend
      if (resultIsInt(expr.op)) { return stmts }
      // Sign extend possible boolean result
      return stmts.concat([(`(if (result i64) (then (i64.const ${TRUE}))
(else (i64.const ${FALSE})))`)])
    case "uniop":
      //TODO
      checkTypeOp(expr.expr, expr.uniop, "", env)
      var stmts: string[] = []
      if (expr.uniop.tag === "neg") {
        expr = {
          tag: "binop",
          expr1: { tag: "literal", value: { tag: "number", value: 0, type:
{ tag: "int" } } },
          expr2: expr.expr,
          op: { tag: "sub" }
        }
        stmts = codeGenExpr(expr, env);
      }
      return stmts
    /*
    var stmts = codeGenExpr(expr.expr, env);
    stmts = stmts.concat(["(i64." + expr.uniop.tag + ")"])
    return stmts.concat(["(i64.extend_i32_s)"])
    */
    case "call":
      var valStmts: string[] = []
```

```
      expr.arguments.forEach(arg => valStmts.push(codeGenExpr(arg,
env).join("\n")))
      valStmts.push(`(call $${expr.name})`);
      return valStmts
  }
}
```

Program-2:

```
def foo() -> bool:
  return True

foo()
```

The ast that is required for function definition and calling is mentioned below. Note that, this is not the complete ast.

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

export type Stmt = { tag: "define", name: string, parameters:
Array<Parameter>, body: Array<Stmt>, ret: Type}

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

export type Expr = { tag: "call", name: string, arguments: Array<Expr> }
```

The parser parses it and adds into the ast of function definition statement and call statement. The relevant snippet of parser is attached below:

```
case "FunctionDefinition":
      c.firstChild();  // Focus on def
      c.nextSibling(); // Focus on name of function
      var funcName = s.substring(c.from, c.to);
      c.nextSibling(); // Focus on ParamList
      var parameters = traverseParameters(c, s);
      c.nextSibling() // focus on body/ret type
      // Has return type
      var retType= null;
      // parse return type
      if(s.substring(c.from,c.to)[0] === '-'){
        c.firstChild();
        retType = traverseType(c, s);
        c.parent();
      }
      c.nextSibling(); // Focus on Body
```

```
      c.firstChild();  // Focus on :

      var bodyStmt = []
      // determine if init came first and func declare not inside function
      while (c.nextSibling()) {
        bodyStmt.push(traverseStmt(c, s));
      }
      c.parent();      // Pop to Body
      c.parent();      // Pop to FunctionDefinition
      var ret: Type = { tag: "int" } // todo
      return {
        tag: "define",
        name: funcName, parameters, body:bodyStmt, ret:retType
      }

  case "CallExpression":
      c.firstChild();
      const callName = s.substring(c.from, c.to);
      c.nextSibling(); // focus on arglist
      c.firstChild(); //focus on (
      var argList = []
      while(c.nextSibling()){
        if(s.substring(c.from, c.to)===","|| s.substring(c.from,
c.to)===")") continue
        var expr = traverseExpr(c, s);
        argList.push(expr)
      }
      c.parent() // pop arglist
      c.parent() // expressionstmt
      return {tag:"call", name: callName, arguments:argList}
```

The compile function first generates code for functions and then for the rest. They maintain a global flag `isFunc` to indicate whether they are generating code for function or for main. The compile function is shown below:

```
export function compile(source: string, env: GlobalEnv): CompileResult {
  const ast = parse(source);
  console.log(ast);
  const definedVars = new Set();
  const withDefines = augmentEnv(env, ast);
  // Check if init or func def came before all other
  var cameBefore = true
  var otherAppear = false
  ast.forEach(s => {
    if (s.tag !== "init" && s.tag !== "define") {
      otherAppear = true
    }
    if (otherAppear && (s.tag === "init" || s.tag === "define")) {
      cameBefore = false
    }
  })
```

```
  // If not defined before
  if (!cameBefore) throw new Error("Program should have var_def and
func_def at top")

  // Function definition
  const funcs: Array<string> = [];
  ast.forEach((stmt) => {
    if (stmt.tag === "define") { isFunc = true; funcs.push(codeGen(stmt,
withDefines).join("\n")); }
  });
  isFunc = false;
  const allFuns = funcs.join("\n\n");
  const stmts = ast.filter((stmt) => stmt.tag !== "define");
  ast.forEach(s => {
    switch (s.tag) {
      case "init":
        definedVars.add(s.name);
        break;
    }
  });
  const scratchVar: string = `(local $$last i64)`;
  const localDefines = [scratchVar];
  definedVars.forEach(v => {
    localDefines.push(`(local $${v} i64)`);
  })

  const commandGroups = stmts.map((stmt) => codeGen(stmt,
withDefines).join("\n"));
  const commands = localDefines.concat([].concat.apply([],
commandGroups));
  //const commands = commandGroups.join("")
  console.log("Generated: ", commands.join("\n"));
  return {
    declFuncs: allFuns,
    wasmSource: commands.join("\n"),
    newEnv: withDefines
  };
}
```

Let's move on to the code generation of function. The codeGen snippet relevant to code generation for function is shown below:

```
case "define":
      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 $${p.name}
i64)`).join(" ");
      const funcVarDecls: Array<string> = [];
      //funcVarDecls.push(`(local $$last i64)`);
      // Initialize function var def
      funcBody.forEach(stmt => {
        if (stmt.tag == "init") {
          funcVarDecls.push(`(local $${stmt.name} i64)`);
        }
      });
      // Treat all local
      // Generate stmts code for func
      var funcStmtsGroup = funcBody.map(stmt => codeGen(stmt, env))
      const funcStmts = [].concat([].concat.apply([], funcStmtsGroup));
      return [`(func $${stmt.name} ${params} (result i64) \n
${funcVarDecls.join("\n")} ${funcStmts.join("\n")})`];
```

It assumes that all variables in a function are local here. But while generating code for `"id"` type expression(code described in the previous problem), it first looks up whether a variable is in global, if yes, it uses global memory to load/store otherwise it uses stack's memory using local.get/set. This leads to a bug here. When there is a function variable with same name as global variable, it would use the value of global variable.

The codeGen code for `init` and `assign` statement checks whether they are generating code for function using `isFunc` variable and based on that they save on stack or global memory. The snippet is shown below:

```
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 "assign":
      if (isFunc) {
        var valStmts = codeGenExpr(stmt.value, env);
```

```
          // Do type check here
          valStmts.push(`(local.set $${stmt.name})`);
          return valStmts;
        }
        const locationToStore = [`(i32.const ${envLookup(env, stmt.name)})
;; ${stmt.name}`];
        var valStmts = codeGenExpr(stmt.value, env);
        let tmp = locationToStore.concat(valStmts).concat([`(i64.store)`]);
        console.log(tmp);
        return tmp
```

The codeGenExpr code for call expression is attached above. We can observe that they don't do any typechecking to confirm whether the return type mentioned and actual function return is the same.

Bugs, Missing Features, and Design Decisions

A bug in this submission is it doesn't use function variables when they have same name as global variables and uses global values instead.

I will diagnose this in-detail and suggest the places where the code need to be updated to handle it.

A sample program that doesn't work:

```
a:int = 6

def foo(a:int) -> int:
  i:int = 4
  i = i + a
  return i

foo(9)
```

The above program returns 10.

To handle this, I would add a `localVars` list/set inside the environment to identify local variables and would update it before generating code for a function. So, my updated environment would look like following:

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

Before I generate code for any function, I would update it: I would create a codeGenFunc that first separates local variables and global variables and populates `localVars` in env and then calls codeGen to generate the code for function body. This is illustrated below as pseudo code below:

```
ast.forEach((stmt) => {
    if (stmt.tag === "define") {
      isFunc = true;
      funcs.push(codeGenFunc(stmt, withDefines).join("\n"));
      }
  });

function codeGenFunc(stmt: Stmt, env: GlobalEnv) : Array<string> {
  // Get the returnStmt
  // Get the params statement
  // Go through ast: Using init statements, make a list of variables that
are initialized inside a function, let's call them definedVars.
  // Generate wasm code for these definedVars: definedVars.forEach(v =>
{localDefines.push(`(local $${v} i64)`);})
  // Add local defined variables and also function parameters to your
localVars inside env.
  // You have to add function parameters as well because they are present
on stack and can be fetched and saved using local.get/set the same way as
locally defined variables.
  // After populating this env, use this to generate code for functions.
```

Once the localVars is populated and stored in env, I would change the load/store. Basically, I will first check whether it's a function using `isFunc` variable and if it's a function, while loading/storing, I would first check whether it's in the localVars set and if it's present then I would use local.get/set otherwise I would get the memory address from global environment and load/store there. The modified "id" handling would look like following:

```
case "id":
      if (isFunc && isFunctionVar(expr.name, env)) {
        return [`(local.get $${expr.name})`]
      }
      else {
        return [`(i32.const ${envLookup(env, expr.name)})`, `(i64.load)`]
      }
```

where isFunctionVar just checks whether variable is in the `localVars`.

```
function isFunctionVar(varName: string, env: GlobalEnv) : boolean {
  return env.localVars.has(varName);
}
```

The updated wat code for the above mentioned sample python program would look like:

```
(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 $a i64) (result i64)
      (local $i i64)
      (i64.const 4)
      (local.set $i)
      (local.get $i)
      (local.get $a)
      (i64.add)
      (local.set $i)
      (local.get $i)
      return
    )
    (func (export "exported_func") (result i64)
      (local $$last i64)
      (local $a i64)
      (i32.const 0) ;; a
      (i64.const 6)
      (i64.store)
      (i64.const 9)
      (call $foo)
      (local.set $$last)
      (local.get $$last)
    )
)
```

Adding New Features

I will describe how to add global declarations inside functions for this submission.

The global declarations allow a function to modify a global variable i.e. saving it. Therefore, we need to keep track of global variables inside function that are declared with a `global` keyword. And also, we have to typecheck while saving a global variable in a function whether it is declared with a `global` keyword. Note that, `"init"` statements inside function could be after `global` declarations. So, we need to handle that as well. And a `global` declared variable must be declared somewhere before in the code. So, we have to check that too. And, if a variable is declared `global` and a local variable is initialized in the function with the same name, we should throw an error.

Firstly, I would add global statement in 'ast' and would add handle it in parser.

The addition to `ast` will be:

```
export type Stmt = { tag: "global", name: string }
```

Similarly, the parser needs to handle `ScopeStatement` and populate `ast`. We need to modify `traverseStmt` and add an additional handling for `ScopeStatement`.

```
export function traverseStmt(c: TreeCursor, s: string): Stmt {
  .
  .
  .
  case "ScopeStatement":
    //handle it and add to ast for statement
    return {tag: "global", name: variableName}
}
```

Next, I need to keep track of global variables declared with `global` keyword. For this, I would follow the same design as local variables mentioned before. Therefore, I would add a `functionGlobalVars` list/set inside the environment to identify global variables declared inside function and would update it before generating code for a function. So, my updated environment would look like following:

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

Note that, I am continuing from my previous explanation that keeps track of local variables in a function whose code is generated using `localVars`.

Before I generate code for any function, I would update it by checking whether there are any global variables. Following the same codeGenFunc mentioned before, I would separate local variables, global declared variables, and global non-declared variables and populate `localVars` in env with local variables and `functionGlobalVars` with global declared variables and then call codeGen to generate the code for function body. This is illustrated below as pseudo code below:

```
ast.forEach((stmt) => {
    if (stmt.tag === "define") {
      isFunc = true;
      funcs.push(codeGenFunc(stmt, withDefines).join("\n"));
      }
  });

function codeGenFunc(stmt: Stmt, env: GlobalEnv) : Array<string> {
  // Check if inits are the first statements other than Global declared
statements
  // Get the returnStmt
  // Get the params statement
  // Go through ast: Using init statements, make a list of variables that
are initialized inside a function, let's call them definedVars.
  // Generate wasm code for these definedVars: definedVars.forEach(v =>
{localDefines.push(`(local $${v} i64)`);})
```

```
  // Add local defined variables and also function parameters to your
localVars inside env.
  // You have to add function parameters as well because they are present
on stack and can be fetched and saved using local.get/set the same way as
locally defined variables.
  // Go through ast: Using global statements, make a list of global
declared variables, let's call them globalDeclVars.
  // Add globalDeclVars to functionGlobalVars inside env.
  // Typecheck: (1) Ensure that all global declared variables exist
somewhere globally. we can do this by checking in globals in the map.
  // Typecheck: (2) Ensure there is no function variable declared with
same name as global declared variable. We can do this by ensuring that the
intersection between globalDeclVars and localVars is null.
  // After populating this env, use this to generate code for functions.
```

Once the `functionGlobalVars` is populated and stored in env, I would change the store. Basically, I will first check whether it's a function using `isFunc` variable and if it's a function, while storing, I would first check whether it's in the `localVars` set and if it's present then I would use local.get/set, if not I would check whether it's in `functionGlobalVars` and if it's present then I would allow saving and save in global memory otherwise I would throw an error. The modified "assign" handling where a variable is store, would look like following:

```
case "assign":
      if (isFunc) {
        if (isFunctionVar(expr.name, env)) {
          var valStmts = codeGenExpr(stmt.value, env);
          valStmts.push(`(local.set $${stmt.name})`);
          return valStmts;
        }
        elif (isFunctionGlobalVar(expr.name, env)) {
          const locationToStore = [`(i32.const ${envLookup(env,
stmt.name)}) ;; ${stmt.name}`];
          var valStmts = codeGenExpr(stmt.value, env);
          let tmp =
locationToStore.concat(valStmts).concat([`(i64.store)`]);
          console.log(tmp);
          return tmp
        }
        else {
          throw Error("Saving non-declared global variable is not allowed
in a function.")
        }

      }
      const locationToStore = [`(i32.const ${envLookup(env, stmt.name)})
;; ${stmt.name}`];
      var valStmts = codeGenExpr(stmt.value, env);
      let tmp = locationToStore.concat(valStmts).concat([`(i64.store)`]);
      console.log(tmp);
      return tmp
```

where isFunctionVar and isFunctionGlobalVar just checks whether variable is in the `localVars` and `functionGlobalVars` respectively.

```
function isFunctionVar(varName: string, env: GlobalEnv) : boolean {
  return env.localVars.has(varName);
}

function isFunctionGlobalVars(varName: string, env: GlobalEnv) : boolean {
  return env.functionGlobalVars.has(varName);
}
```

The updated wat code for the following sample python program would look like:

```
b:int = 6

def foo(a:int) -> int:
  global b
  b = a + b
  return a

foo(9)
```

```
(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 $a i64) (result i64)
        (i32.const 0) ;; b
        (local.get $a)
        (i32.const 0)
        (i64.load)
        (i64.add)
        (i64.store)
        (local.get $a)
        return
    )
    (func (export "exported_func") (result i64)
        (local $$last i64)
        (local $b i64)
        (i32.const 0) ;; b
        (i64.const 6)
        (i64.store)
        (i64.const 9)
        (call $foo)
        (local.set $$last)
        (local.get $$last)
```

```
    )
)
```

Lessons and Advice

1. **Better design decision than mine**

They have a separate possible op list declared before, which is helpful.

2. **Worse design decision than mine**

They don't keep track of Decl statements separately which are used widely everywhere later.

3. **What's one improvement you'll make to your compiler based on seeing this one?**

The way they check whether init cames before is interesting and would like to adopt that.

4. **What's one improvement you recommend this author makes to their compiler based on reviewing it?**

Better keep track of declaration statements and use env to keep track of function local variables and global variables.

Submission 61875546

1. For each of the compilers you are reviewing, choose two programs that run successfully

Program-1:

```
i:int = 4
while i > 0:
    i = i - 1
print(i)
```

The ast that is required for while-statement is mentioned below. Note that, this is not the complete ast.

```
export type Stmt = { tag: "while", expr: Expr, stmts: Array<Stmt> }

export type Expr =
    { tag: "literal", value: Literal }
  | { tag: "id", name: VarName }
  | { tag: "binaryop", expr1: Expr, expr2: Expr, op: Op}
  | { tag: "unaryop", expr: Expr, op: Op}
  | { tag: "call", name: VarName, args: Array<Expr> }

export type VarName = string
export type Op = string
export type Type = string
```

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

The parser parses it and adds into the ast of while statement. The relevant snippet of parser is attached below:

```
case "WhileStatement": {
      c.firstChild();  // while
      c.nextSibling();  // expr
      const expr = traverseExpr(c, s);
      c.nextSibling();  // Body
      c.firstChild();
      c.nextSibling();
      const stmts = [];
      do {
        stmts.push(traverseStmt(c, s));
      } while (c.nextSibling())
      c.parent();  // Body
      c.parent();  // WhileStatement
      return {tag: "while", expr, stmts};
    }
```

The typechecker returns `TypedExpr` which contains expression and its type. It is defined as follows:

```
export type TypedExpr = {
  expr: Expr,
  type: ClassType,
}
```

The typechecking corresponding to while statement is as follows:

```
case "while": {
      let t = tcExpr(s.expr);
      if (t.type.name !== "bool") {
        throw new Error("Expect bool type, get type " + t.type.name);
      }

      return;
    }
```

The `codeGenStmt` code corresponding to while is as follows:

```
case "while": {
      wasms = wasms.concat(
        [`(block \n(loop`],
        codeGenExpr(s.expr),
        [`(i32.const 1)`, `(i32.xor)`, `(br_if 1)`],
      )
      s.stmts.forEach(stmt => {
        wasms = wasms.concat(codeGenStmt(stmt));
      });
      wasms = wasms.concat(
        [`(br 0)\n)\n)`]
      )
      break;
    }
```

The `codeGenExpr` generates code for the condition expression. The code for `codeGenExpr` corresponding to binary operation is as follows:

```
case "binaryop": {
      const expr1Stmts = codeGenExpr(expr.expr1);
      const expr2Stmts = codeGenExpr(expr.expr2);
      wasms = wasms.concat(
        expr1Stmts,
        expr2Stmts,
        binaryOpToWASM.get(expr.op),
      )
      break;
    }
```

Here the `binaryOpToWASM` is a mapping from operation to its corresponding statement in wasm as below:

```
export const binaryOpToWASM: Map<string,Array<string>> = new Map([
  .
  .
  .
  ["+", ["(i32.add)"]],
  ["-", ["(i32.sub)"]],
  ["*", ["(i32.mul)"]],
  .
  .
  .

])
```

Program-2:

```
def foo() -> bool:
  return True

foo()
```

The ast that is required for function definition and calling is mentioned below.

```
export type Program = {
  defs: PreDef,
  stmts: Array<Stmt>
}

export type PreDef = {
  varDefs: Array<VarDef>,
  funcDefs: Array<FuncDef>,
}

export type VarDef = { tvar: TypedVar, value: Literal }
export type TypedVar = { name: VarName, type: Type}

export type FuncDef = {
  name: VarName,
  params: Array<TypedVar>,
  retType: Type,
  body: FuncBody,
}

export type FuncBody = {
  defs: PreDef,
  stmts: Array<Stmt>
}

export type Stmt =
    { tag: "assign", name: VarName, value: Expr }
  | { tag: "if", exprs: Array<Expr>, blocks: Array<Array<Stmt>> }
  | { tag: "while", expr: Expr, stmts: Array<Stmt> }
  | { tag: "pass" }
  | { tag: "return", expr: Expr }
  | { tag: "expr", expr: Expr }

export type Expr =
    { tag: "literal", value: Literal }
  | { tag: "id", name: VarName }
  | { tag: "binaryop", expr1: Expr, expr2: Expr, op: Op}
  | { tag: "unaryop", expr: Expr, op: Op}
  | { tag: "call", name: VarName, args: Array<Expr> }

export type VarName = string
export type Op = string
export type Type = string
```

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

As we can observe, they have separated function definitions from normal definitions. They parse function definitions and normal variable definitions first before parsing any other statements as shown below:

```
export function traverse(c : TreeCursor, s : string) : Program {
  switch(c.node.type.name) {
    case "Script":
      c.firstChild();
      const [defs, end] = traverseDefs(c, s);
      const stmts: Array<Stmt> = [];
      if (!end) {
        return {defs, stmts};
      }
      do {
        stmts.push(traverseStmt(c, s));
      } while(c.nextSibling())
      console.log("traversed " + stmts.length + " statements ", stmts,
"stopped at " , c.node);
      return {defs, stmts};
    default:
      throw new Error("Could not parse program at " + c.node.from + " " +
c.node.to);
  }
}
```

This `traverseDefs` calls `traverseFuncDef` which parses function definition.

```
export function traverseFuncDef(c : TreeCursor, s : string) : FuncDef {
  c.firstChild();  // def
  c.nextSibling();  // VariableName
  const name = s.substring(c.from, c.to);
  const params = [];
  c.nextSibling();  // ParamList
  c.firstChild();  // (
  c.nextSibling();  // VariableName
  while (c.node.type.name != ")") {
    params.push(traverseTypedVar(c, s))
    c.nextSibling();  // , | )
    c.nextSibling();
  }
  c.parent();  // ParamList
  c.nextSibling();  // TypeDef or Body

  let retType = "<None>";
```

```
  if (c.type.name === "TypeDef") {
    c.firstChild();  // VariableName
    retType = s.substring(c.from, c.to);
    c.parent();  // TypeDef
    c.nextSibling();  // Body
  }

  const body = traverseFuncBody(c, s);
  c.parent();

  return {name, params, retType, body};
}
```

The snippet relevant to parsing call expression is shown below:

```
case "CallExpression": {
      c.firstChild();
      const name = s.substring(c.from, c.to);
      c.nextSibling(); // go to arglist
      c.firstChild(); // go into arglist
      const args = [];
      while (c.nextSibling() && c.node.type.name !== ")") {
        args.push(traverseExpr(c, s));
        c.nextSibling();
      }
      c.parent(); // pop arglist
      c.parent(); // pop CallExpression
      return {
        tag: "call", name, args
      };
    }
```

Similarly, they generate code for variable definition first, function definition next and then all other statements.

```
function codeGenProgram(p: Program): Array<Array<string>> {

  let varWASM = codeGenVarDef(p.defs.varDefs);
  let funcWASM = codeGenFuncDef(p.defs.funcDefs);
  let stmtsWASM: Array<string> = new Array();

  p.stmts.forEach(stmt => {
    stmtsWASM = stmtsWASM.concat(
      codeGenStmt(stmt)
    )
  })

  return [varWASM, funcWASM, stmtsWASM];
}
```

They recursively generate code for function definition by generating for the function variable declarations first and body next and then recursively for the functions inside body.

```
function codeGenFuncDef(fds: FuncDef[]): Array<string> {
  let wasms: Array<string> = new Array();
  fds.forEach(fd => {
    let funcName = curEnv.name + "." + fd.name;
    curEnv = envMap.get(funcName);
    wasms = wasms.concat(
      [`(func $${funcName} (result i32)`],
      codeGenVarDef(fd.body.defs.varDefs),
    )
    fd.body.stmts.forEach(stmt => {
      wasms = wasms.concat(codeGenStmt(stmt));
    })
    wasms = wasms.concat(
      [`)`]
    );

    wasms = wasms.concat(codeGenFuncDef(fd.body.defs.funcDefs));
    curEnv = curEnv.parent;
  })

  return wasms;
}
```

From the above recursive code, we can observe that they support nested functions.

They save all the variables in heap. And retrieve memory locations of variables from heap.

Bugs, Missing Features, and Design Decisions

A bug in this submission is it doesn't use parameters at all i.e. any function that takes parameters and use them wouldn't work fine.

I will diagnose this in-detail and suggest the places where the code need to be updated to handle it.

A sample program that doesn't work:

```
def foo(a:int) -> int:
  return a

foo(9)
```

The above program returns 0.

To handle this, I would add a `functionParams` list/set inside the environment to identify function parameters and would update it before generating code for a function. So, my updated environment

would look like following:

```
export type GlobalEnv = {
  name: "",
  parent: null,
  nameToVar: new Map(),
  paramsName: new Array(),
  functionVars: Set<any>;
}
```

Before I generate code for any function, I would update environment and after I generate code for environment, I would empty `functionVars`. Therefore, my codeGenFunc would be modified to:

```
function codeGenFuncDef(fds: FuncDef[]): Array<string> {
  let wasms: Array<string> = new Array();
  fds.forEach(fd => {
    let funcName = curEnv.name + "." + fd.name;
    curEnv = envMap.get(funcName);
    // Add parameters to curEnv
    wasms = wasms.concat(
      [`(func $${funcName} (result i32)`],
      codeGenVarDef(fd.body.defs.varDefs),
    )
    fd.body.stmts.forEach(stmt => {
      wasms = wasms.concat(codeGenStmt(stmt));
    })
    wasms = wasms.concat(
      [`)`]
    );

    wasms = wasms.concat(codeGenFuncDef(fd.body.defs.funcDefs));
    curEnv = curEnv.parent;
  })

  return wasms;
}
```

And, while generating code for `id`, I would check whether it's a function parameter and if it is, I will use local.get otherwise get from heap.

```
case "id": {
      wasms = wasms.concat(getPointerWithOffset(generalPointer.get("DL"),
-1));  // pointer to current SL
      let iterEnv = curEnv;
      if (isFunc && curEnv.functionParams.has(varName)) {
        // Get variable using local.get
        wasms = wasms.concat([local.get ${variableName}])
      }
```

```
      else {
        let counter = 0;
        while (!iterEnv.nameToVar.has(expr.name)) {
          counter += 1;
          iterEnv = iterEnv.parent;
          wasms = wasms.concat([`(i32.load)`])
        }
        // at SL now
        let idInfo = iterEnv.nameToVar.get(expr.name);
        wasms = wasms.concat([
          `(i32.const ${- (idInfo.offset - 1) * 4})`,
          `(i32.add)`,
          `(i32.load)`,
        ])
      }
      break;
    }
```

Adding New Features

I will describe how to add `and/or` for this submission.

I will add `and` and `or` operator along with their `i32.and` and `i32.or`. Since `True` is 1 and `False` is 0, I can use i32.and/or operations for boolean `and/or`. The `binaryOpToWasm` which maintains a mapping between binary operation and its wasm code, should be modified to:

```
export const binaryOpToWASM: Map<string,Array<string>> = new Map([
  ["+", ["(i32.add)"]],
  ["-", ["(i32.sub)"]],
  ["*", ["(i32.mul)"]],
  ["//", ["(i32.div_s)"]],
  ["%", ["(i32.rem_s)"]],
  ["==", ["(i32.eq)"]],
  ["!=", ["(i32.ne)"]],
  ["<=", ["(i32.le_s)"]],
  [">=", ["(i32.ge_s)"]],
  ["<", ["(i32.lt_s)"]],
  [">", ["(i32.gt_s)"]],
  ["is", ["(i32.eq)"]],
  ["and", ["(i32.and)"]],
  ["or", ["(i32.or)"]],
])
```

We need to add handling for `and/or` in the typechecker verifying whether the operands are booleans. So, we have to add `and/or` to `boolOp`, a variable that keeps track of operations that takes booleans as operands.

```
const boolOp: Array<FuncType> = [
    { name: "__eq__", paramsType: [boolType], returnType: boolType },
```

```
    { name: "__neq__", paramsType: [boolType], returnType: boolType },
    { name: "__and__", paramsType: [boolType], returnType: boolType },
    { name: "__or__", paramsType: [boolType], returnType: boolType },
    { name: "__not__", paramsType: [], returnType: boolType },
  ]
```

These changes are sufficient to typecheck it and generate code for them.

This is the sample wat code that is generated for an `or` operation:

```
True or False
```

```
(module
    (import "js" "mem" (memory 10))  ;; memory with one page(64KB)
    (func $builtin_print (import "imports" "print") (param i32)(param i32)
(result i32))

    (func (export "exported_func") (result i32)
      (local $$last i32)
      (i32.const 4)
      (i32.const 4)
      (i32.load)
      (i32.const 655352)
      (i32.add)
      (i32.store)
      (i32.const 8)
      (i32.const 8)
      (i32.load)
      (i32.const 655356)
      (i32.add)
      (i32.store)
      (i32.const 1)
      (i32.const 0)
      (i32.or)
      (local.set $$last)
      (local.get $$last)
    )
  )
```

Lessons and Advice

1. **Better design decision than mine**

I liked the way they kept track of caller/callee and its hierarchical structure which helped them with nested functions easily.

2. **Worse design decision than mine**

I found their memory handling really difficult to understand and hard to keep track of. This is just my personal opinion. For example, a 1-line python code generates such long wasm code with lots of unnecessary loads and stores:

```
i:int = 9
```

```
(module
    (import "js" "mem" (memory 10))  ;; memory with one page(64KB)
    (func $builtin_print (import "imports" "print") (param i32)(param i32)
(result i32))
    (func (export "exported_func")
        (local $$last i32)
        (i32.const 4)
        (i32.const 4)
        (i32.load)
        (i32.const 655352)
        (i32.add)
        (i32.store)
        (i32.const 8)
        (i32.const 8)
        (i32.load)
        (i32.const 655356)
        (i32.add)
        (i32.store)
          (i32.const 4)
        (i32.const 4)
        (i32.load)
        (i32.const -8)
        (i32.add)
        (i32.store)
        (i32.const 4)
        (i32.load)
        (i32.const 0)
        (i32.add)
        (i32.const 2)
        (i32.store)
        (i32.const 4)
        (i32.load)
        (i32.const 4)
        (i32.add)
        (i32.const 9)
        (i32.store)
    )
  )
```

3. **What's one improvement you'll make to your compiler based on seeing this one?**

I will try to keep track of caller/callee and BinaryOpToWasm so that by just changing 2 lines, I am able to handle another boolean expression the way they did.

4. **What's one improvement you recommend this author makes to their compiler based on reviewing it?**

I think they're using lot of memory by unnecessary loads/stores. Please keep track of it.