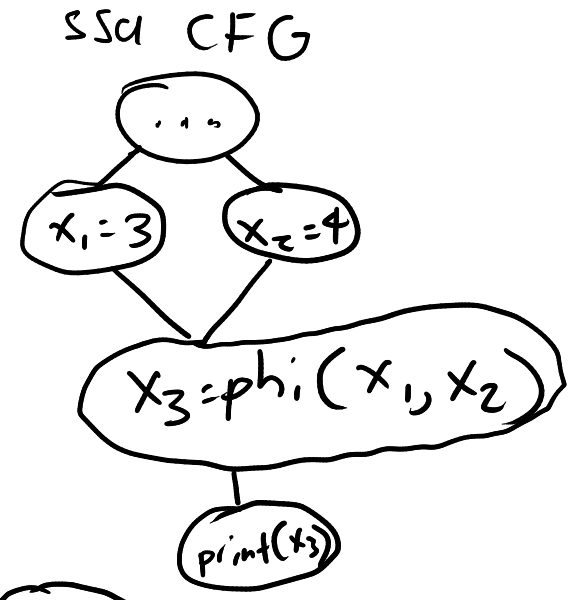
```
// not SSA
x = input()
x = x + 2
x = x * 3

// SSA
x1 = input()
x2 = x1 + 2
x3 = x2 * 3
```

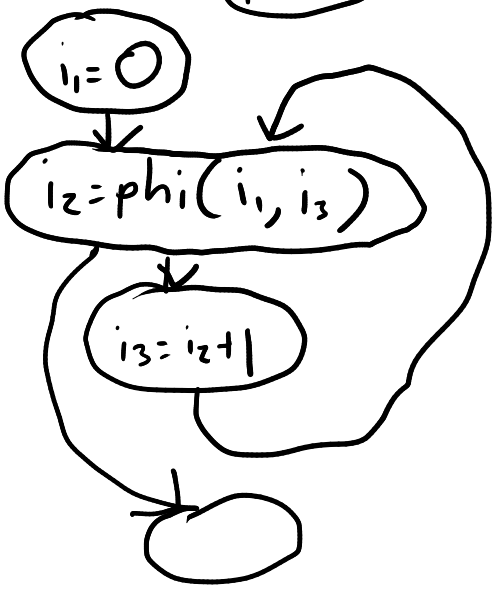
```
// not SSA
if (...) {
      x = 3
} else {
      x = 4
}
print(x)

// SSA
if (...) {
      x1 = 3
} else {
      x2 = 4
}
x3 = phi(x1, x2)
print(x3)
```




```
// not SSA
i = 0
while (i < 10) {
      i = i + 1
}

// SSA
i1 = 0
while (i2 = phi(i1, i3); i < 10) {
      i2 = i3 + 1
}
```




```
if (...) {
      x = 10
} else {
      x = 11
}
...
if (...) {
      print(x * 2)
} else {
      print(x * 3)
}
```

```
if (...) {
      x1 = 10
} else {
      x2 = 11
}
x3 = phi(x1, x2)
if (...) {
      print(x3 * 2)
} else {
      print(x3 * 3)
}
```

```
struct BasicBlock {
	std::vector<BasicBlock*> predecessors;
	std::map<std::string, Value*> variables;
	std::vector<Value*> operations;
	std::set<Value*> incomplete;
	std::variant<Return, Jump, Branch> terminator;
	bool sealed; // no more predecessors may be added
};

struct Return {
	Value* value;
};

struct Jump {
	BasicBlock* target;
};

struct Branch {
	Value* condition;
	BasicBlock* if_true;
	BasicBlock* if_false;
}


Value* read_variable(BasicBlock* block, std::string name) {
	// TODO
}

Value* write_variable(BasicBlock* block, std::string name, Value* value) {
	// TODO
}

void seal_block(BasicBlock* block) {
	// TODO
}


Value* codegen_expression(BasicBlock* current, Expr expr) {
	// creates or looks up and returns the previous value of expr
}


BasicBlock* codegen_stmt(BasicBlock* current, Stmt stmt) {
	// returns the basic block the next statement should continue in
}


 BasicBlock* codegen_body(BasicBlock* current, std::vector<Stmt> statements) {
	for stmt in statements {
		current = codegen_stmt(current, stmt);
	}
	return current;
 }
```

```
BasicBlock* codegen_statement(BasicBlock* current, Stmt stmt) {
    if stmt is an expression {
        Value* value = codegen_expression(current, stmt as expr);
        return current;
    } else if stmt is an assignment {
        Value* value = codegen_expression(current, stmt.rhs);
        write_variable(current, stmt.lhs, value);
        return current;
    } else if stmt is a return {
        if stmt.value == nullptr {
            current->terminator = Return { nullptr };
        } else {
            Value* value = codegen_expression(current, stmt.value);
            current->terminator = Return { value };
        }
        return nullptr;
    } else if stmt is an if-statement {
        Value* condition = codegen_expression(current, stmt.condition);
        BasicBlock* if_true = new BasicBlock();
        BasicBlock* if_false = new BasicBlock();
        BasicBlock* merge = new BasicBlock();
        current->terminator = Branch { condition, if_true, if_false };
        if_true->add_predecessor(current);
        seal_block(if_true);
        if_false->add_predecessor(current);
        seal_block(if_false);
        BasicBlock* if_true_end = codegen_body(if_true, stmt.if_true_body);
        if_true_end->terminator = Jump { merge };
        merge->add_predecessor(if_true_end);
        // don't seal yet
        BasicBlock* if_false_end = codegen_body(if_false, stmt.if_false_body);
        if_false_end->terminator = Jump { merge };
        merge->add_predecessor(if_false_end);
        // now seal
        seal_block(merge);
        return merge;
    } else if stmt is a while-loop {
        BasicBlock* header = new BasicBlock();
        BasicBlock* body = new BasicBlock();
        BasicBlock* after = new BasicBlock();
        current->terminator = Jump { header };
        header.add_predecessor(current);
        // don't seal yet
        Value* condition = codegen_expression(header, stmt.condition);
        header->terminator = Branch { value, body, after };
        body->add_predecessor(header);
        // don't seal yet
        after->add_predecessor(header);
        // don't seal yet
        BasicBlock* loop_body_end = codegen_body(body, stmt.body);
        loop_body_end->terminator = Jump { header };
        header->add_predecessor(loop_body_end);
        seal_block(header);
        seal_block(body);
        seal_block(after);
        return after;
    } else if stmt is a break {
        // TODO
    } else if stmt is a continue {
        // TODO
    }
}
```

```
Value* codegen_expression(BasicBlock* current, Expr expr) {
	if (expr is an IdentExpr) {
		return read_variable(current, expr.name);
	} else if (expr is a BinaryExpr) {
		Value* left = codegen_expression(current, expr.left);
		Value* right = codegen_expression(current, expr.right);
		return new BinaryValue(current, left, right);
	}
}


 Value* write_variable(BasicBlock* block, std::string name, Value* value) {
	block->variables[name] = value;
 }

 Value* read_variable(BasicBlock* block, std::string name) {
	if block->variables.contains(name) {
		return block->variables[name];
	} else if !block->sealed {
		Value* phi = new Phi(block); // add Phi to vector of operations for that block
		block->incomplete.insert(phi);
		return phi;
	} else if block.predecessors.size() == 1 {
		return read_variable(block.predecessors[0], name);
	} else {
		Value* phi = new Phi(block); // add Phi to vector of operations for that block
		write_variable(block, name, phi);
		for p in block->predecessors {
			phi->add_operand(read_variable(p, name));
		}
		return phi;
	}
 }

 void seal_block(BasicBlock* block) {
	for phi in block->incomplete {
		for p in block->predecessors {
			phi->add_operand(read_variable(p, name));
		}
	}
	block->sealed = true;
 }
```