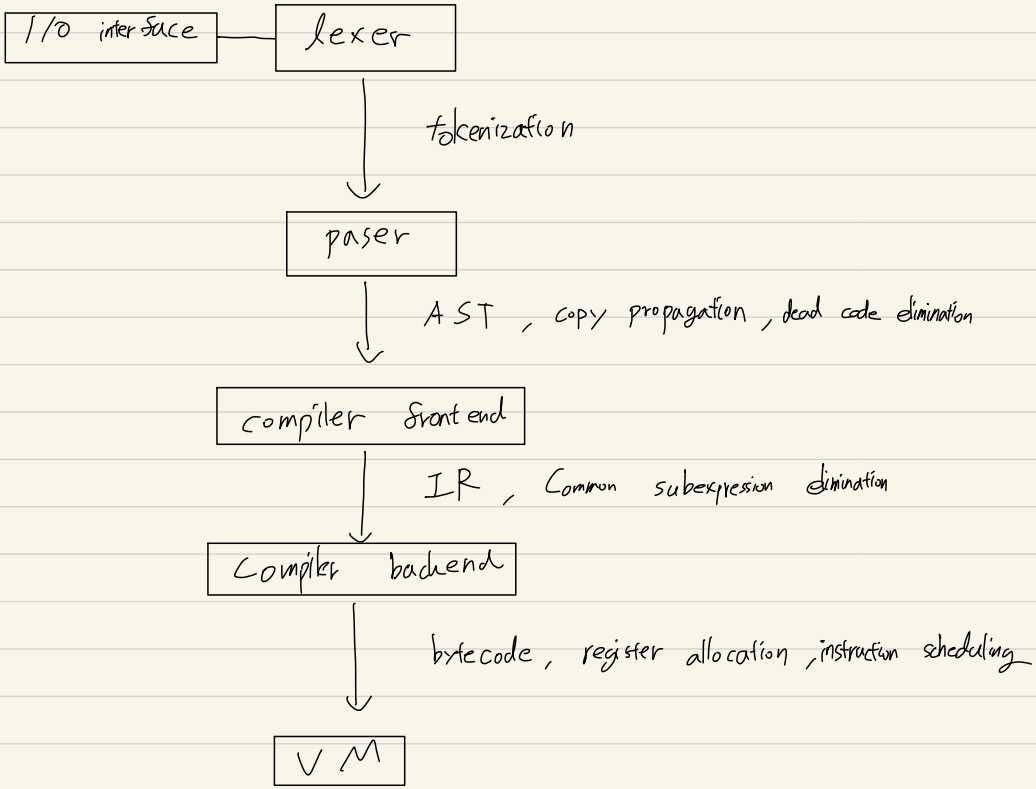


Forward pass : AST, copy propagation

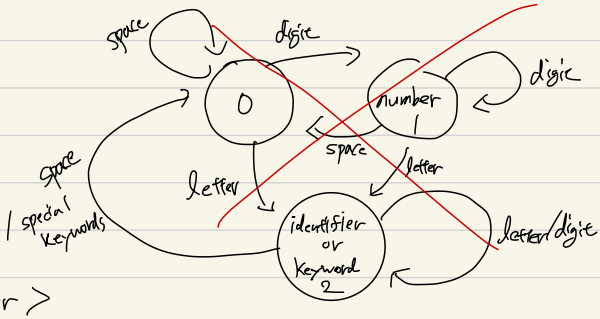
Forward pass : IR(SSA), Common subexpression elimination

{ Backward Pass : Register allocation
  Forward Pass : Instruction scheduling } × n where n = 0, 1, ...

I/O interface and lexer

<I/O interface>

read_words (filename) : return a list of words separated by space, tab, new line



special keywords : [,], (,), {, }, ←, ,, ;, :, .

<lexer>

① Read words

② Tokenization

@ data class

Token { }

Identifier (Token) { value : string }

Number (Token) { value : int }

Relation (Token)

Eq (Relation), Neq (Relation), G (Relation), Ge (Relation), L (Relation), Le (Relation)

Operator (Token)

Plus (Operator), Minus (Operator), Mul (Operator), Div (Operator)

Keyword (Token)

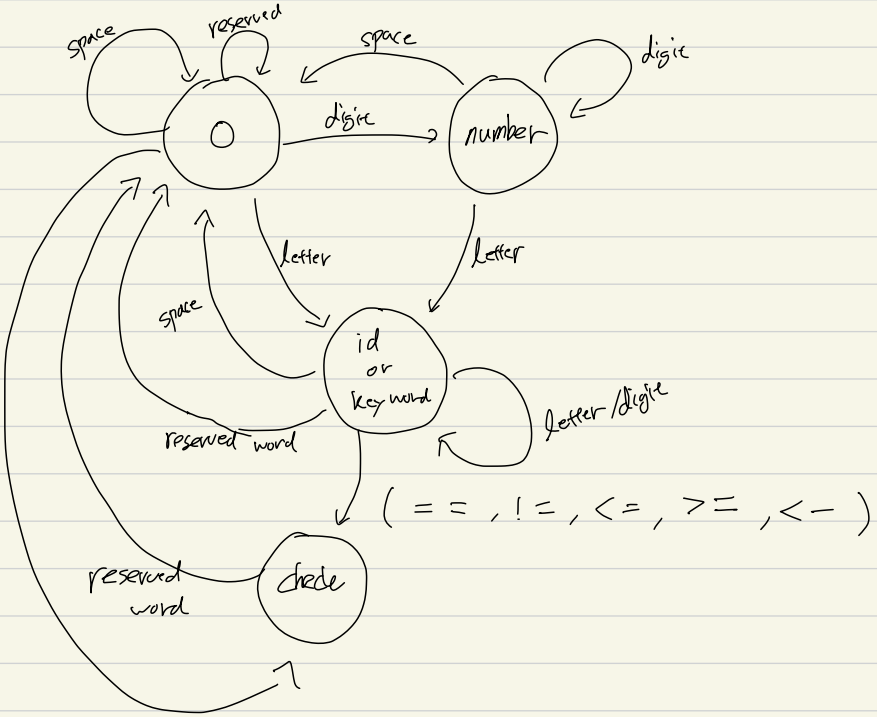
child class of keyword class ⇒ { 0: right_brace, 1. left_bracket, 2: right_paren, 3: left_paren
4: right_brace, 5: left_brace, 6: let, 7: assign, 8: call
9: comma, 10: if, 11: then, 12: else, 13: fi, 14: while, 15: do
16: od, 17: return, 18: semicolon, 19: var, 20: array, 21: void,
22: function, 23: main, 24: dot }

Input Num (Token) { // No value }

Output Num (Token) { string } → id

Output Newline (Token) { // No value }

<I/O interface State machine>



Parser

@dataclass

```
Computation () { var_decl : list [VarDecl]
                 func_decl : list [FuncDecl]
                 stats_sequence : list [Statement]
               }
VarDecl () { type_decl : TypeDecl
             id : string
           }
TypeDecl () { }
Var (TypeDecl)
Array (TypeDecl) { idx_list : list [int]
                   }
FuncDecl () { void : boolean      // True only if void is stated
              id : string
              formal_param : list [string]
              func_body : FuncBody
            }
FuncBody () { var_decl : list [VarDecl]
              stats_sequence : list [Statement]
            }
Statement () { Statement_inherit : Boolean } // True if child is used for Statement
Assignment (Statement) { designator : Designator
                         expression : Expression
                       }
Designator (Factor) { id : string
                      idx_list : list [Expression]
                    }
Expression (Factor) { terms : list [Term]
                      ops : list [Op]
                    }
```

Factor, assignment

Should I distinguish two usages?

Maybe I should

solution: use inherit variable

```
Op () { }
Plus (Op) { }
Minus (Op) { }
Mul (Op) { }
Div (Op) { }
Term () {   factors : list [Factor]
            ops : list [Op]
        }
Factor () { factor_inherit : Boolean }          // True if child is used for factor
Number (Factor) { value: int
        }
Func Call (Factor, Statement) {   id : Identifier
                                  arg_list : list [Expression]
                                }
Identifier) { value: string
        }
If Statement (Statement) { relation : Condition
                           then_block: list [Statement]
                           else_block: list [statement]
                         }
While Statement (Statement) { relation : Condition
                              do_block: list [Statement]
                            }
Return Statement (Statement) { value : None | Expression
                             }
```

for dead code elimination

Condition True (Relation) { }

Condition False (Relation) { }

```
Relation () { left : Expression
              rel_op : Rel Op
              right : Expression
            }
```

RelOP () { }
Eq (RelOp) { }
Neq (RelOp) { }
G (RelOp) { }
Ge (RelOp) { }
L (RelOp) { }
Le (RelOp) { }

< Copy propagation >
Inside Term :

---

number "*" | "/" number ⟼ number

Inside Expression :

---

number "+" | "−" number ⟼ number

Inside Relation :

---

number relOp number ⟼ True | False

< Dead code elimination >
① If Statement

---

if True then stateSequence1 [else stateSequence2] fi ⟼ StateSequence1

---

if False then stateSequence1 else stateSequence2 fi ⟼ StateSequence 2

---

if False then stateSequence fi ⟼ ∅

② while

while False do stat Sequence od ↦ ∅

③ assignment

remove assigment which is not used → using SSA Table

④ func Call ✗

remove funcCall which is not called

⑤ function inlining (Optional) ✗

Compiler front end

Later, line numbers should be assigned for each inst and all branch instructions should be updated before register allocation

```
BasicBlock {  ~~prevs : list[BasicBlock]~~
              nexts : list[BasicBlock]
              SSA_table : dict(id, Instruction | (int, list[int]))   # for array
              CSE_table : dict(inst_name, inst)
              instructions : list[Instruction | InstructionSet]
           }

InstructionSet() {  instructions : list[Instruction]
                 }

Instruction() { x: int | Instruction | string (FP), y : int | Instruction | string,   # for function call
  assigned_reg : boolean, offset : int (?), id : string (for CSETable), call_id : string, call_arg : list[int|Instruction], line_number : int }
```

Neg(Instruction), Add(Instruction), Sub(Instruction), Mul(Instruction), Div(Instruction), Cmp(Instruction)
adda(Instruction), Load(Instruction), Store(Instruction), Phi(Instruction), End(Instruction), Bra(Instruction)
Bne(Instruction), Beq(Instruction), Ble(Instruction), Blt(Instruction), Bge(Instruction), Bgt(Instruction)
Read(Instruction), Write(Instruction), WriteNL(Instruction), Call(Instruction), Return(Inst), PUSH?

We need Ordered dict because of while?

```
SSATable(dict) {   dict(id (str), Instruction | (base_addr, total_idx) | offset | int)
                       # for function params — I might not need this?
                       # ↳ stack에 박을거면 필요
                       #    ↳ register allocation 간단
                   get(id) -> Instruction   // if there is no id then raise
                                            else if instruction is None raise unitialized
                                            → emit zero  How?
                   update(id, Instruction) -> None
               }

CSETable(dict) {  dict(inst_id (str), Node)
                  class Node {  value : Instruction
                                next : Node
                             }
                  lookup(Instruction) -> None | Instruction
                  add(Instruction)
                  lookup_return(Instruction) -> list[Instruction], Instruction
                  kill()  // How to check addr?   or kill all
               }
```

<Operational semantics>

Current : current basic block

operate_computation :=

This works on list [func_decl]

$$\frac{\text{operate\_var\_decl}(\text{var\_decl}) \mapsto \text{SSATable}}{\text{Computation}(\text{var\_decl}, \text{func\_decl}, \text{state\_sequence}) \mapsto \text{SSATable}, \text{func\_decl}, \text{state\_sequence}}$$

$$\frac{\text{operate\_func\_decl}(\text{func\_decl}) \mapsto \text{list}[(\text{id}, \text{BasicBlock})]}{\text{SSATable}, \text{func\_decl}, \text{state\_sequence} \mapsto \text{SSATable}, \text{list}[(\text{id}, \text{BasicBlock})], \text{state\_sequence}}$$

$$\frac{\text{operate\_state\_sequence}(\text{SSATable}, \text{CSETable}, \text{state\_sequence}) + |\text{list}| \mapsto \text{list2}}{\text{SSATable}, \text{list}[(\text{id}, \text{BasicBlock})], \text{state\_sequence} \mapsto \text{list2}[(\text{id}, \text{BasicBlock})]}$$

operate_var_decl :=

$$\frac{}{\text{list}[\text{VarDecl}] \longmapsto \text{SSATable}}$$

~~all element of array should be inserted into SSATable~~

No, SSATable should store base address for array and idx_list

operate_func_decl :=

call 했을 때 업데이트 되어야 한다.

$$\frac{\text{operate\_formal\_param}(\text{formal\_param}) \mapsto \text{SSATable}}{\text{FunDecl}(\text{void}, \text{id}, \text{formal\_param}, \text{func\_body}) \mapsto \text{void}, \text{id}, \text{SSATable}, \text{func\_body}}$$

$$\frac{\text{operate\_func\_body}(\text{SSATable}, \text{func\_body}) \longmapsto \text{BasicBlock}}{\text{void}, \text{id}, \text{SSATable}, \text{func\_body} \mapsto \text{void}, \text{id}, \text{BasicBlock}}$$

$$\frac{}{\text{void}, \text{id}, \text{BasicBlock} \longmapsto (\text{id}, \text{BasicBlock})}$$

operate_formal_param :=

$$\frac{}{\text{FormalParam} \mapsto \text{SSATable}}$$

operate_func_body :=

$$\frac{\text{operate\_var\_decl}(\text{var\_decl}) \mapsto \text{SSATable}|}{\text{FuncBody}(\text{var\_decl}, \text{state\_sequence}), \text{SSA2} \longmapsto \text{SSATable}|+\text{SSA2}, \text{state\_sequence}}$$

Result = int | Instruction

first의 argument 3 int 생략가능

operate_state_sequence (SSATable, CSETable, state_sequence) ⟼ Basic Block

BB = Basic Block

SSATable, state_sequenc ⟼ Basic Block

$x \in$ { assignment, funccall, is-statement, while-statement, return statement }

operate_state_sequence :=

SSATable, CSETable, state_sequence ⟼ SSATable, CSETable, x :: tail

operate_assignment (SSATable, CSETable, Assignment) ⟼ BB

SSATable, CSETable, Assignment :: tail ⟼ BB, tail

operate_func_call (SSATable, CSETable, Funccall) ⟼ BB

SSATable, CSETable, Funccall :: tail ⟼ BB, tail

operate_is_statement (SSATable, CSETable, IfStatement) ⟼ BB

SSATable, CSETable, IfStatement :: tail ⟼ BB, tail

operate_while_statement (SSATable, CSETable, WhileStatement) ⟼ BB

SSATable, CSETable, WhileStatement :: tail ⟼ BB, tail

operate_return_statement (SSATable, CSETable, ReturnStatement) ⟼ BB

SSATable, CSETable, ReturnStatement :: tail ⟼ BB, tail

operate_state_sequence (BB.SSATable.copy(), BB.CSETable.copy(), tail) ⟼ BB2

BB, tail ⟼ BB.nexts.append(BB2)

Be careful when to copy SSATable and CSETable and when not to.

operate_assignment (SSATable, CSETable, Assignment(designator, expression)) → BB :

list1, result1 = operate_expression (SSATable, CSETable, expression)

list2, result2 = operate_designator (SSATable, CSETable, designator, result1)

assert result1 is result2 ? (x)

return BB(list1 + list2, SSATable, CSETable)

operate_func_call (SSATable, CSETable, FuncCall(id, arg_list)) → BB | List[inst], Result :

if FuncCall.factor_inherit :

result = [], args = []

for expression in arg_list

l, r = operate_expression (SSATable, CSETable, expression)

result += l

arg.append(r)

inst = Call(id, args)

return result + inst, inst

elif FuncCall.statement_inherit :

result = [], args = []

for expression in arg_list

l, r = operate_expression (SSATable, CSETable, expression)

result += l

arg.append(r)

inst = Call(id, args)

return BB(result + inst, SSATable, CSETable), None

operate_if_statement (SSATable, CSETable, IfStatement(relation, then, else) → BB, BB

joint = BB()

cond_bb, cond_brn = operate_relation (SSA, CSE, relation)

SSA1 = SSA.copy() , CSE1 = CSE.copy()

SSA2 = SSA.copy() , CSE2 = CSE.copy()

then_bb = operate_stat_sequence (SSA1, CSE1, then),

if else is not None :

else_bb = operate_stat_sequence (SSA2, CSE2, else)

// Compare SSA1 and SSA2 to generate phi function

// Update SSA with phi function

// Add phi functions to joint

// Update bra to then block, insert bra to else block

// Link BBs and return joint or return cond? or return start BB and end BB?

O(n)

time complexity ?

too much overhead

// also should check kill is on in CSE

↳ all load

연결해야 할 것들

```
operate_relation (SSATable, CSETable, relation (lest, rel_op, right)) → BB, bra :
    list1, result1 = operate_expression (SSATable, CSETable, lest)
    list2, result2 = operate_expression (SSATable, CSETable, right)
    list3, cmp = CSETable.lookup_return (CMP(result1, result2))
    bra ← select branch inst (cmp, empty)   according to rel_op
    return BB (list1 + list2 + list3 + bra, SSA, CSE), bra
                                  ↑ (location to jump cannot be resolved yet)

operate_while_statement (SSATable, CSETable, WhileStatement (relation, do)) → BB, BB :
    joint_bb, cond_bra = operate_relation (SSA, CSE, relation)
    SSA1 = SSA.copy(), CSE1 = CSE.copy()
    do_bb = operate_state_sequence (SSA1, CSE1, do)
    // compare SSA1 and SSA to generate phi with empty right  ⇒ SSA2
    new_joint_bb, new_cond_bra (SSA2, CSE, relation)
    SSA3 = SSA2.copy()   CSE2 = CSE.copy()
    new_do_bb = operate_state_sequence (SSA3, CSE2, do)   // check kill is on in CSE2
    // compare SSA3 and SSA2 update right sides of phi functions
    // put phi into the front of new_joint_bb
    // insert loopback bra into new_do_bb
    // insert (add 0 0) and update new_cond_bra
    // Link BBs and return   // attach empty block at the end

operate_return_statement (SSA, CSE, ReturnStatement (value)) → BB :
    return_bb = BB()
    if value is expression :
        list, r = operate_expression (SSA, CSE, value)
        BB.append (list)
        BB.append (Return (r))
    else :
        BB.append (Return())
```

for assignment

```
operate_designator (SSATable, CSETable, designator(id, idx_list), value: Result) → List[inst], result:
    assert designator.factor_inherit == True and value is None
    assert designator.factor_inherit == False and value is not None
    is_array = len(idx_list) > 0
    if designator.factor_inherit == True:
        if is_array:
            base_addr, total_idxs = SSATable.get(id)
            l, base = CSETable.lookup_return(Add("FP", base_addr))
            for i, expression in enumerate(designator.factor.idx_list):
                list1, Result = operate_expression(SSATable, CSETable, expression)
                // compute element_size with total_idxs[i:]
                list2, x = CSETable.lookup_return(Mul(Result, element_size))
                l += list1
                l += list2
                if i > 0:
                    list3, x = CSETable.lookup_return(Add(y, x))
                    l += list3
                y = x
            list1, addr = CSETable.lookup_return(Adda(base, x))
            list2, ld = CSETable.lookup_return(load(addr))
            l += list1, list2
            return l, ld
        else:
            inst = SSATable.get(id)
            return [], inst
    else:
        if is_array:
            base_addr, total_idx = SSATable.get(id)
```

extract method

// function parameter 일 경우

if inst == offset

// compute addr and load

```
l, base = CSETable.lookup_return(Add("FP", base_addr))
for i, expression in enumerate(designator.factor.idx_list):
    list1, Result = operate_expression(SSATable, CSETable, expression)
    // compute element_size with total_idxs[i:]
    list2, x = CSETable.lookup_return(Mul(Result, element_size))
    l += list1
    l += list2
    if i > 0:
        list3, x = CSETable.lookup_return(Add(y, x))
        l += list3
    y = x
list1, addr = CSETable.lookup_return(Adda(base, x))
list2, st = CSETable.lookup_return(store(value, addr))
l += list1, list2
return l, value
else:
    SSATable.update(id, value)
    return [], value
```

```
operate_expression(SSATable, CSETable, expression(terms, ops)) -> list[Instruction], Result :
    list1, x = operate_term(SSATable, CSETable, terms[0])
    for term, op in zip(terms[1:], ops):
        list2, y = operate_term(SSATable, CSETable, term)
        list1 += list2
        list3, x = compute(op, x, y, SSATable, CSETable)
        list1 += list3
    return list1, x
```

CSETable method

```
compute(op, x, y, SSATable, CSETable) -> list[Instruction], Result :
    if isinstance(x, int) and isinstance(y, int):
        if op == Plus : return [], x+y
        elif op == Minus : return [], x-y
        elif op == Mul : return [], x*y
        else : return [], x/y
    else :
        if op == Plus :
            return lookup_return(Add(x, y, id="add"))
        elif op == Minus :
            return lookup_return(Sub(x, y, id="sub"))
        elif op == Mul :
            return lookup_return(Mul(x, y, id="mul"))
        elif op == Div :
            return lookup_return(Div(x, y, id="div"))
        else : raise
```

parent에서 처리됨

CSETable

```
def lookup_return(Inst):
    old = CSETable.lookup(Inst)
    if old is None :
        CSETable.add(Inst)
        return [Inst], Inst
    else :
        return [], old
```

```
operate_term (SSATable, CSETable, term(factors, ops)) -> list[Instruction], Result:
    list1, x = operate_factor (SSATable, CSETable, factors[0])
    for factor, op in zip(factors[1:], ops):
        list2, y = operate_factor (SSATable, CSETable, factor)
        list1 += list2
        x = compute(op, x, y, SSATable, CSETable)
        if isInstance(x, Instruction):
            list1.append(x)
    return list1, x

operate_factor (SSATable, CSETable, factor) -> list[Instruction], Result:
    if isInstance(factor, Designator):
        return operate_designator (SSATable, CSETable, factor, None)
    elif isInstance(factor, Number):
        return [], Number.value  -> Add Inst
    elif isInstance(factor, Expression):
        return operate_expression (SSATable, CSETable, expression)
    elif isInstance(factor, FuncCall[id, arg_list]):

        result = [], args = []
        for expression in arg_list
            l, r = operate_expression (SSATable, CSETable, expression)
            result += l
            arg.append(r)
        inst = Call(id, args)
        return result + inst, inst
```

All of this can be done in code generation?

this should be updated as line number later

<Resolve call>

Insert push operation and jump to function. I need a function table (id → inst) global

go back here

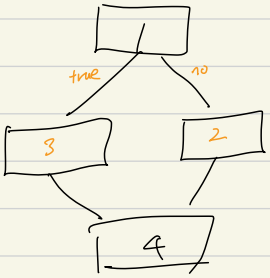
<Stack allocation>

When should this be done? before line numbering

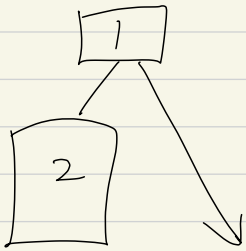
<line numbering>

Before the register allocation, each line should be assigned a line number. For this, we should sequentialize every BB.

- If



- while



<Resolve branch>

After line numbers are assigned to each instruction, every branch should store proper destination as a line number. other than branch? call

<Resolve cmp and branch inst>

Cmp should be converted to sub.

Should it be done now?

Compiler backend

① Register allocation
   a) interference graph
   b) graph coloring + cost function design
   c) assign register + resolve phi function

. cost function
   cost (i) := number of apperance of i

② Instruction scheduling => basic block based (block pruning is needed)
   a) topological ordering

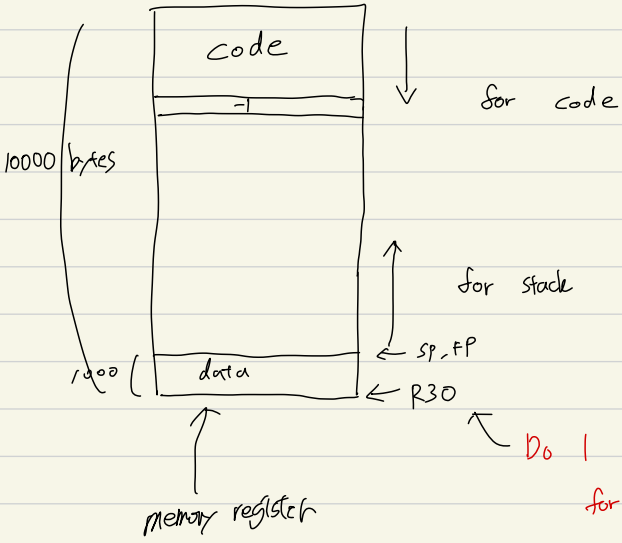
③ iteration

④ Code generator (using DLX.py definition)
. Keywords: "FP" = R28 ... frame pointer
   "SP" = R29 ... stack pointer
. cmp → sub

D L X

R0 = 0

sp = 9000
FP = 9000



Do I need a global variable?
for now let's make it only accessible from main => R30 is now useless

binary