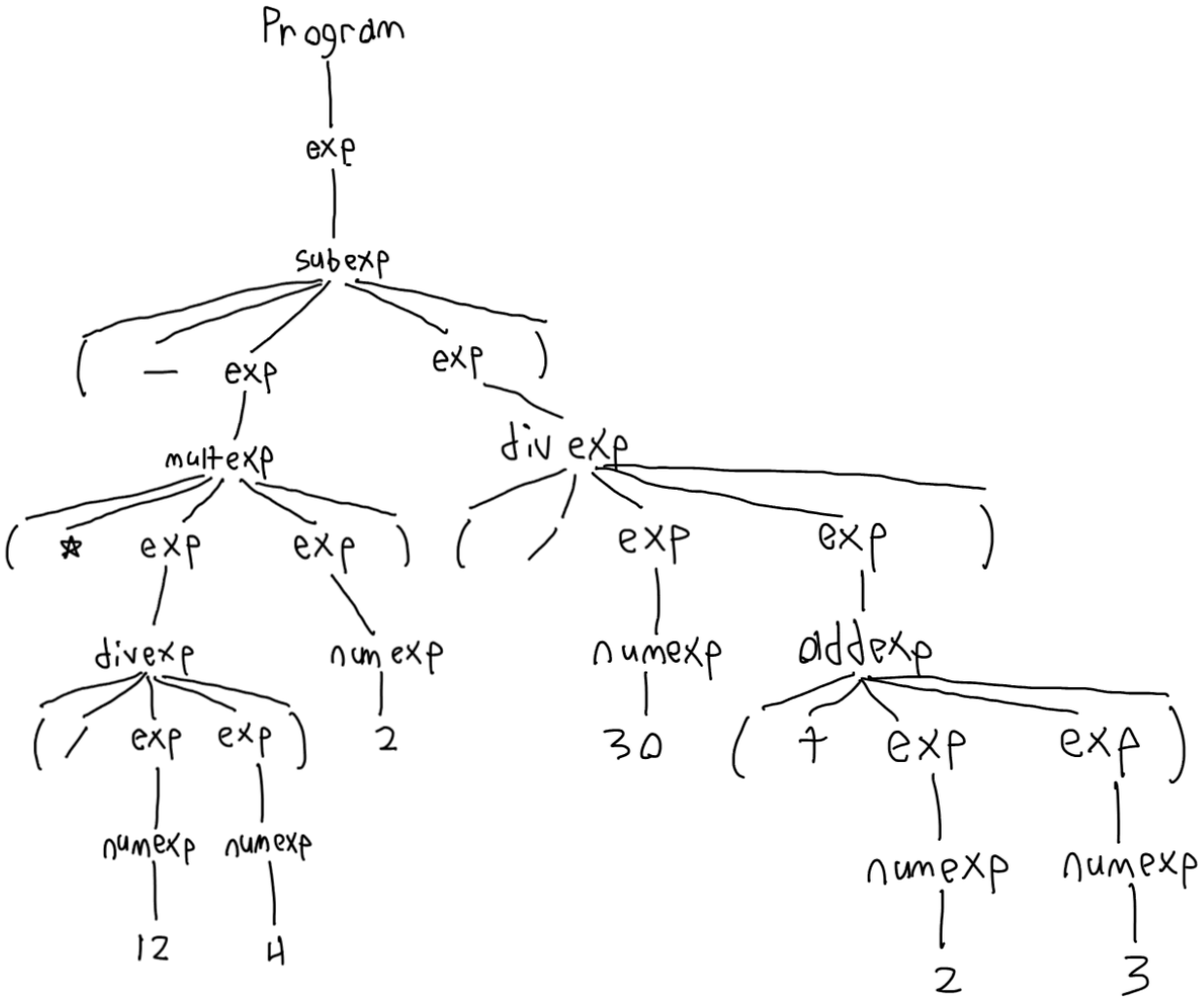
Questions:

1. (3 pt) [Writing ArithLang programs] Write one Arithlang program that computes the value: *3420.0*. You must use each of the ArithLang operators at least once.

   (- (* 3 (/ (+ 54 79) 2) (- 12 5) 3) 769.5)

2. (3 pt) [Arithlang syntax] Get familiar with the syntax of Arithlang by going through the grammar file at: *arithlang/src/main/antlr/arithlang/parser/ArithLang.g4*. Then construct a parse tree for the following program: (- (* (/ 12 4) 2) (/ 30 (+ 2 3)))



3. (6 pt) [ArithLang interpreter understanding] Get familiar with the ArithLang source code:

   (a) (3 pt) List the methods invoked in the ArithLang interpreter when it processes the expression: (+ 3 (- 4 2). The methods do not need to be ordered and, repeatedly invoked methods are only needed to be included once. Do not include any methods called in the auto-generated Parser class from ANTLR (*ArithLangParser.java*) or methods in Java standard library classes.
   For example:

   Interpreter::main(String[] args)
   Reader::Reader()
   Evaluator::Evaluator()

```
Printer::Printer()
Reader::read()
Reader::readNextProgram()
Reader::runFile(String programText) // Input: (+ 3 (- 4 2)
Reader::parse(String programText) // Input: (+ 3 (- 4 2)
Reader::getLexar(CharStream s)
ArithLandLexer::ArithLandLexer(CharStream input)
LexerATNSimulator::LexerATNSimulator(Lexer recog, ATN atn,DFA[] decisionToDFA,
        PredictionContextCache sharedContextCache)
Reader::getParser(CommonTokenStream s)
Evaluator::valueOf(Program p)
Program::accept(Visitor visitor)
Evaluator::visit(Program p)
AddExp::accept(Visitor visitor)
Evaluator::visit(AddExp e)
CompoundArithExp::all()
NumExp::accept(Visitor visitor)
Evaluator::visit(NumExp e)
NumVal::NumVal(Double v)
NumVal::v()
SubExp::accept(Visitor visitor)
Visitor::visit(SubExp e)
Printer::pr      int(Value e)
```

(b) (3 pt) Extract and describe the code snippets that implements the visitor pattern for evaluating an expression.

This snippet is the implementation for the visitor pattern with an AddExp input. It starts by extracting all the operands of the expression and iterates over each one. For each operand, it calls its accept method to visit the expression recursively. This way it traverses down each expressions parse tree until arriving at a final value, or a NumExp whos visit returns a NumVal. After getting a NumVal from each operand, NumVal::v() grabs the double value, and we add it to the total result. This way we accumulate the results of evaluating every expression or operand of the AddExp. Lastly, the final result it returned.

```
@Override
public Value visit(AddExp e) {
    double result = 0;
    List<Exp> operands = e.all();
    for (Exp operand : operands) {
        NumVal val = (NumVal) operand.accept(this);
        result += val.v();
    }
    return new NumVal(result);
}
```

4. (18 pt) Extend the ArithLang to add the *Power* operation. The operation takes integers as input and computes an exponential function. See example scripts below:

$ (Power 2 3)

$ 8 // 2^3
$ (Power -1 5)
$ -1 // $(-1)^5$
$ (Power 2 3 2)
$ 64 // $(2^3)^2$
$ (+ (Power 6 2) 4)
$ 40
$ (Power x 2)
error // can be any error message
$ (Power 2)
error // can be any error message

5. (26 pt) Implement an interpreter for AbstractLang described as follows. You can start modifying from the ArithLang code

   (a) This language contains only five terminals, p, n, z, u, *, +

   (b) p represents positive numbers, n represents negative numbers, z represents 0, and u represent unknown values

   (c) There are two operators * and + that can be applied on the terminals, their syntactic rules are similar to * and + in ArithLang, except that each operator only can take two operands

   (d) The semantics of AbstractLang are defined by the following rules (columns in the table rep resents the first operand and the rows in the table represents the second operand of the operation):

+	p	n	z	u
p	p	u	p	u
n	u	n	n	u
u	u	u	u	u
z	p	n	z	u

*	p	n	z	u
p	p	n	z	u
n	n	p	z	u
u	u	u	z	u
z	z	z	z	u