Parsing, ASTs, Pretty Printing, and the Visitor Design Pattern

Recitation 6

S-expressions

- Used to represent data of many types, especially lists.
- Invented for Lisp

S-expression examples

- (1)
- (“hello” “world”)
- (1 (2 3 “hello”) (2 (1 (1 2))))
- (“function_call” (“print” (“hello world”)))

An S-expression inspired grammar

expr → (list) | atom

list → expr*

atom → <**num literal**> | <**string literal**>

Note: terminals are in <> and bolded

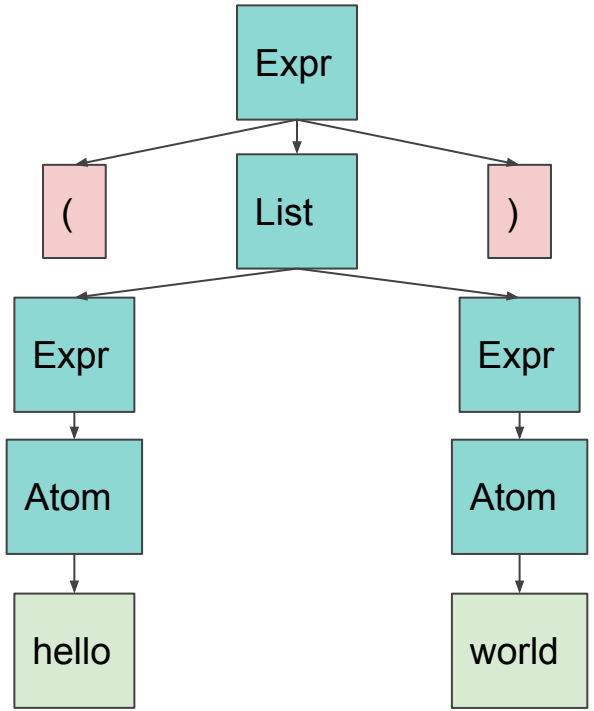
Naive Concrete Syntax Tree

(“hello” “world”)

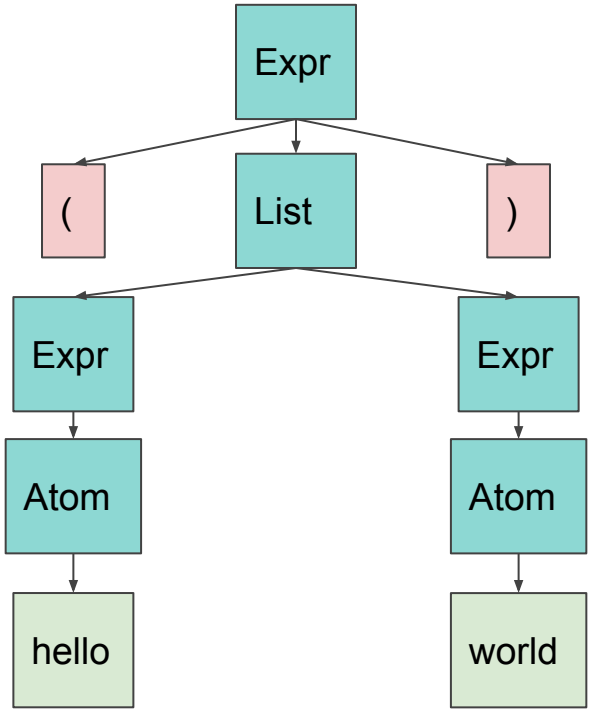
expr → (list) | atom

list → expr*

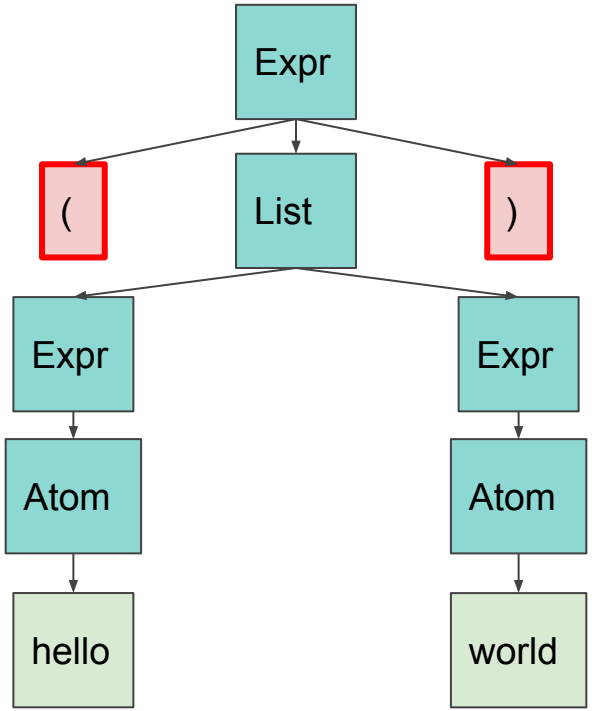
atom → <**num literal**> | <**string literal**>



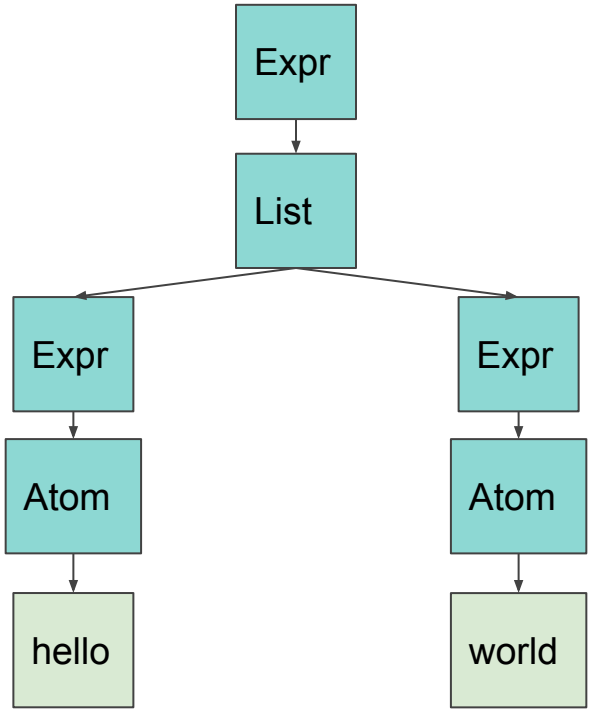
Making an AST



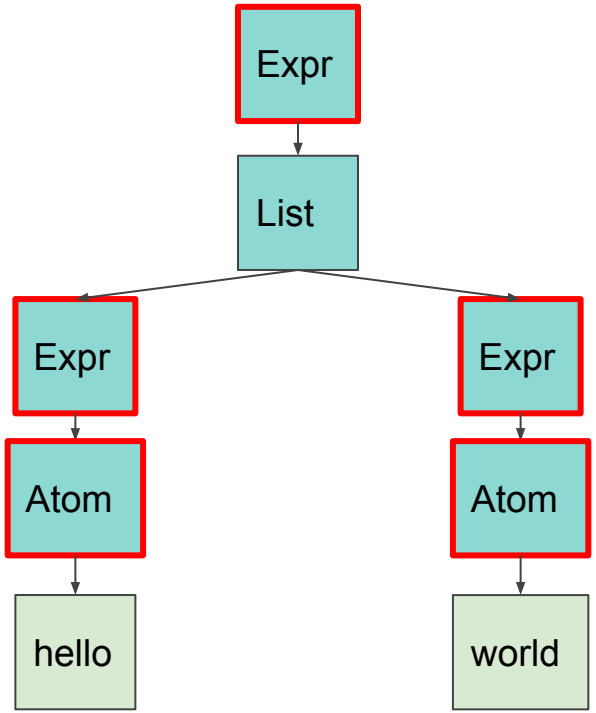
Making an AST



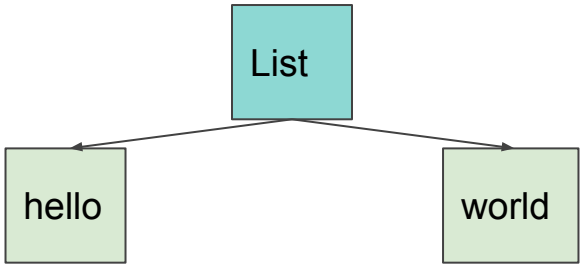
Making an AST



Making an AST

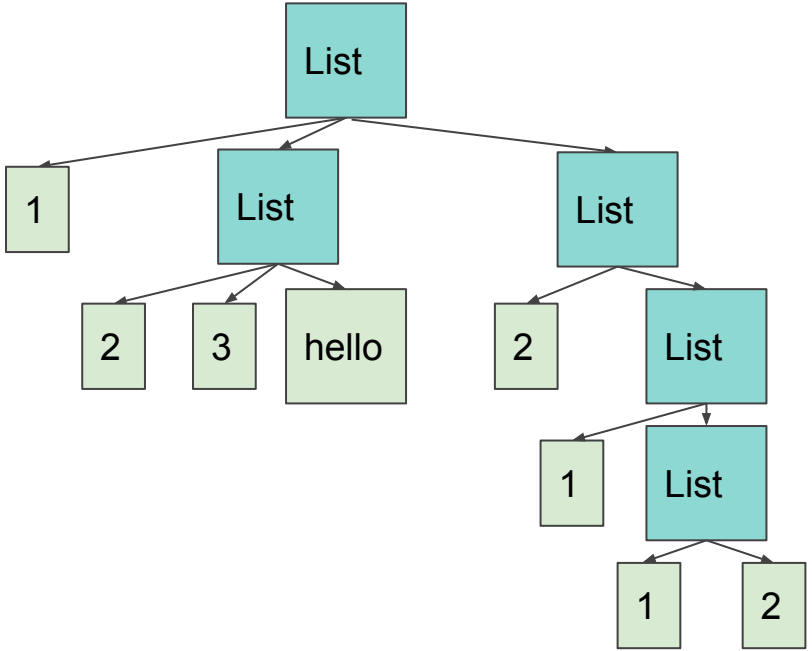


Making an AST



Another Example

(1 (2 3 “hello”) (2 (1 (1 2))))



Parsing

- Use EBNF (Extended Backus–Naur form) as a model
- Make a separate method for most rows

Parsing

EBNF Grammar	Implementation
expr → (list) | atom	Expr parseExpr(); // will call parseList or parseAtom
list → expr*	List parseList(); // will call parseExpr
atom → <**num literal**> | <**string literal**>	Atom parseAtom(); // won't call the other two

Demo

A simple grammar

expr → term PLUS term | term

term → factor MUL factor | factor

factor → base EXP factor | base

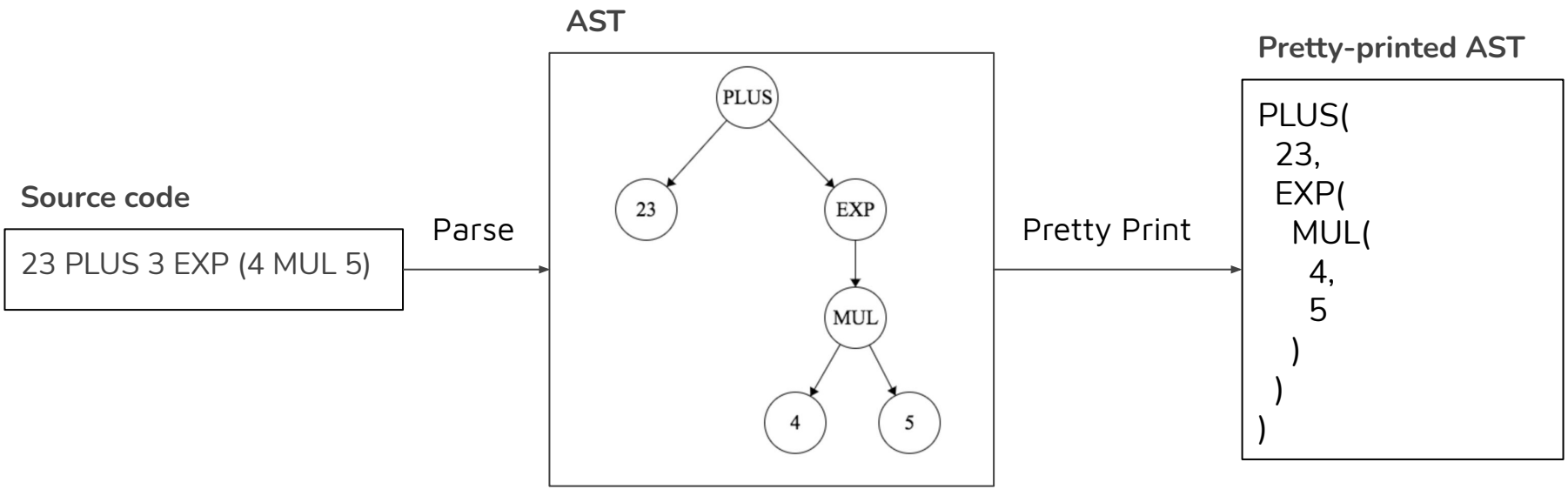
base → <num> | (expr)

Example Expressions

- 5
- 3 PLUS 4 MUL 5
- 23 PLUS 3 EXP (4 MUL 5)
- 2 EXP (2 EXP (2 PLUS 2))

Pretty Printing

Goal: print out a human readable representation of an AST



Pretty Printing with Indentation

```
class PrettyPrinter {
  int level = 0;

  private final StringBuilder sb = new StringBuilder();

  void addLine(String line) {
    sb.append(“ ”.repeat(level * 2))
      .append(line)
      .append(‘\n’);
  }
}

printer.level++;
printer.addLine(“this line is indented”)
printer.level--;
```

Pretty Printing with Indentation

```
class PrettyPrinter {
  // other code...

  void indented(??? printerCode) {
    level++;
    // TODO: run the printer code
    level--;
  }
}
```

Pretty Printing with Indentation

```
class PrettyPrinter {
  // other code...

  // Runnable is an interface with only one method: void run();
  void indented(Runnable printerCode) {
    level++;
    printerCode.run();
    level--;
  }
}

printer.addLine(“foo”);
printer.indented(() -> printer.addLine(“I’m indented!”));
printed.addLine(“bar”);
```

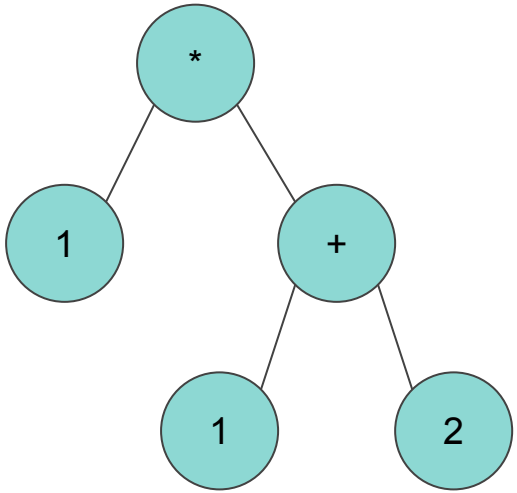
Pretty Printing and Parentheses

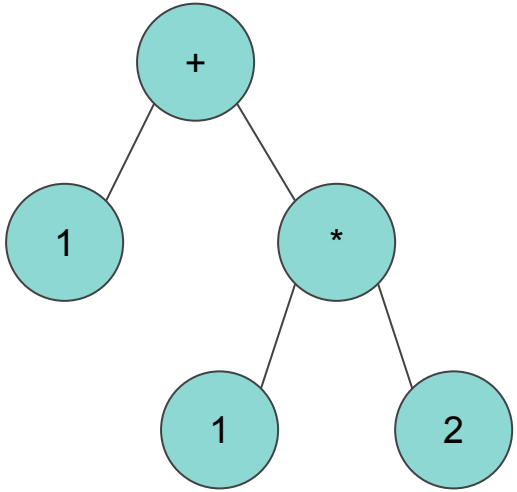
1*(1+2)

1+1*2

1+(1*2) 😢

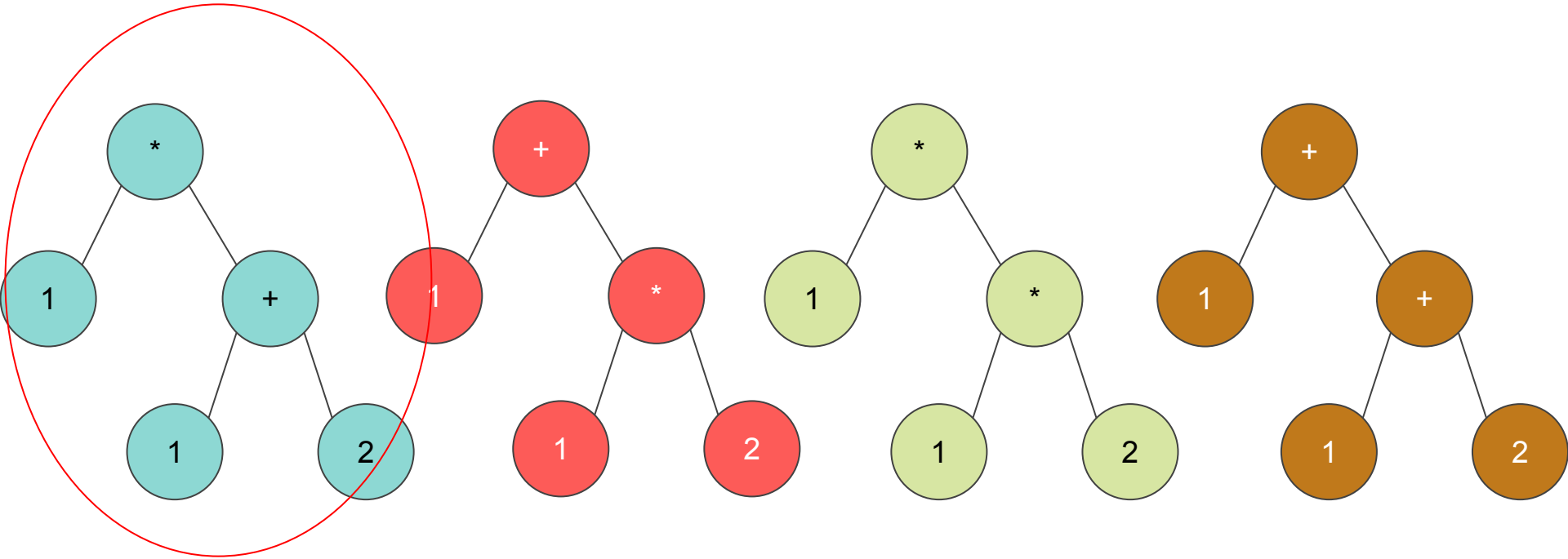
(1 + (1 * 2)) 😢





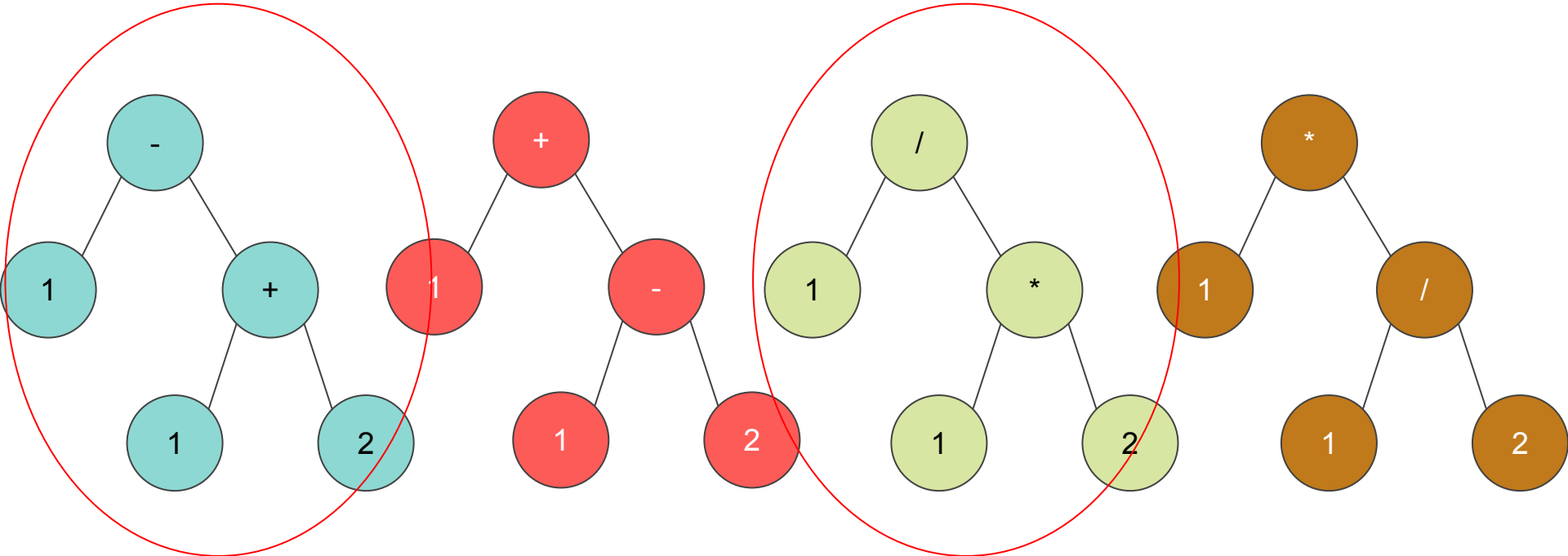
Can we avoid parentheses whenever possible?

Which ones must we add parenthesis?



What if we also have - and /

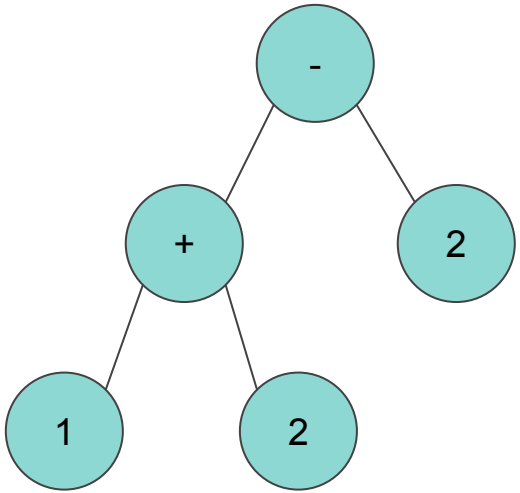
Which ones must we add parenthesis?



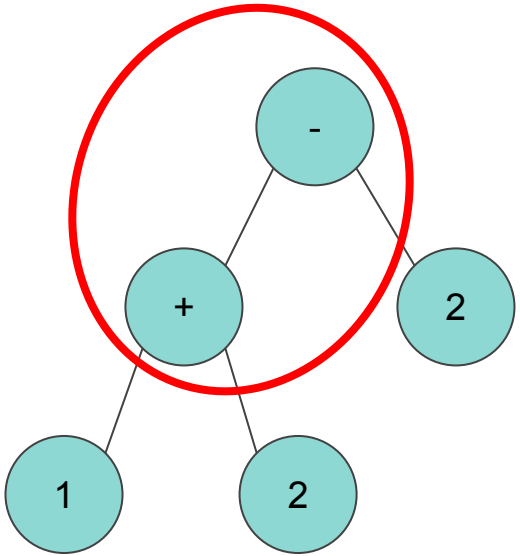
What about child on the left?

You don’t need parenthesis!

(a+b)-c == a+b-c

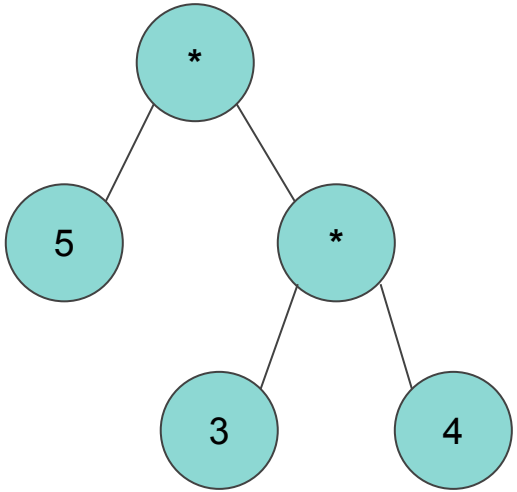


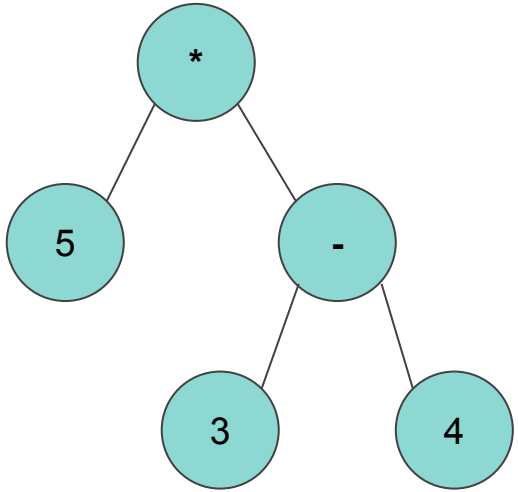
Challenge: Complete this function with the rules




```
boolean needParentheses(
  Operator parentOperator,
  Operator childOperator,
  boolean isLeftChild // refers to childOperator
) {
  // Your code here
  // Add more helper functions as you see fit
}
```

How Much Child Info Do We Need?





Traversing ASTs

Review: Single Dispatch (Polymorphism)

```
public interface Animal {

    void makeSound();

}
public class Cat implements Animal{

    public void makeSound() {
        System.out.println("meow");
    }

}
public class Dog implements Animal{

    public void makeSound() {

        System.out.println("bark");
    }

}
```

```
public static void main(String[] args) {

    Animal animal = new Cat();
    animal.makeSound();

    animal = new Dog();
    animal.makeSound();
}
```

Example Problem: Credit Card Cash Back

	Bronze	Silver	Gold
Gas	1%	2%	4%
Food	2%	3%	6%
Hotel	1%	1.5%	5%
Other	1%	1%	3%

How could we implement this?

- Using logic

```
if (instanceof(Gold)) {
     if (instanceof(GasOffer)) {}
     else if (instanceof(HotelOffer)) {}
else if (instanceof(Silver)) {
     if (instanceof(GasOffer)){ }
     else if (instanceof(HotelOffer)) {} ….
```

- Hard to Maintain

How could we implement this?

- Within the credit card class

```
class GoldCreditCard implements CreditCard() {

    // functions in Credit Card

    computeGasCashBack(){}

    computeHotelCashBack() {}
```

- Results in a lot of code that has doesn't have anything fundamentally to do with Credit Cards to be in Credit Card class
- Also hard to maintain

A Better Way:
Visitor Design Pattern Demo

How this is useful for your project

- We want to traverse the AST and perform different actions depending on NodeType and traversal type
  - ActionNode & pretty-print
  - RelationNode & parse
  - CommandNode & execute
- A solution is to have traversal code in nodes' classes
- IntegerNode has pretty_print code, evaluation code, etc.
- Code for given traversal is spread through all classes; difficult to read and maintain
- Use Visitor Pattern to do this instead

Tying it Together

```
public class StringNode extends AbstractNode {
    public void accept(Visitor v){
        v.visit(this);
    }
}

public class Interpret extends Visitor {
    public void visit(StringNode n) {
        // Know we are interpreting String Node
    }
    public void visit(IntNode n) {
        //Interpreting IntNode
    }
}
```

Drawbacks of Visitor Pattern

- Adding new node requires updating all visitors
- Each node class has to contain an accept method
- Need a new visitor class for every action
- Logic for a specific element is spread out across all visitor classes

If we have implemented the Visitor Pattern, what changes to the AST node (i.e. Action) do we need to make if we add a new way to traverse the AST (i.e. implement a new mutation)?

A. No changes
B. We need to add a new accept method to the AST Node
C. We need to implement this traversal in the node class
D. I'm confused

What is another advantage of the Visitor Pattern?

A. Because the way function calls are structured in Java, the Visitor Pattern is faster
B. Given the same number of nodes and traversals, the Visitor Pattern requires less code
C. If we forget to implement a method (either accept or visit), the compiler will often tell us
D. Implementing the visitor pattern will give you extra credit

Introduction to λ-Calculus

- A formal system in mathematical logic for expressing computation
- Universal model of computation that can simulate any Turing Machine
- Introduced in the 1930s
- Functional programming languages like OCaml implement λ-Calculus
- See more in CS 3110, CS 4110, and CS 6110

Some Key Ideas

Functions are anonymous:

- Instead of squared (x) = x^2, we write x -> x^2
- Instead of identity(x) = x, we write x -> x

Functions only have one input:

- Instead of (x,y) -> x^2 + y^2:
- x-> (y -> x^2 + y^2)
- We first provide the x input, say 2:
- Returns a new function (y -> 2^2 + y^2)
- Now provide the next input, say 5:
- 2^2 + 5^2 = 29

Lambda Calculus

Three types of expressions:

- Variables
  - x, y, z
- Functions
  - λx.x, λx.y
- Application
  - (x y), (λx.x) y

Example expressions

- (λx.x) y
- λx.λy.x
- (λx.λy.x y) (λx.x) (z)

Grammar

expr → function | var | app | (expr)

function → λvar.expr

app → expr expr

var → <variable name$_1$>

1. where a variable name is a string of alphabetic latin characters