## Admin

- Two alternative tutorials

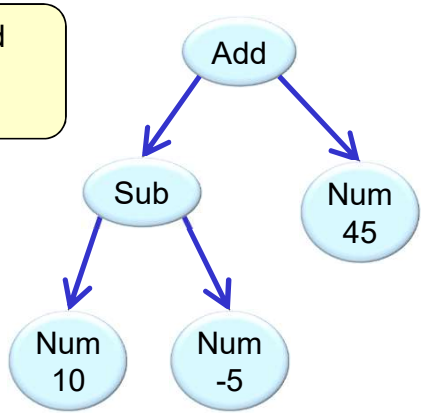
- Thursday 5:10-6pm in TTR104 ( room is confirmed. Please go to this one if you are signed up for Tuesday 2-3 )
- Friday 10:00-11:50am AM101 (room is confirmed. Please go to this one if you are signed up for Tuesday 4-5).

# What we can do with an AST: 1 - print it out

```
public interface ExprNode { }

class NumNode implements ExprNode {
   final int value;
   public NumNode(int v){ value = v; }
   public String toString() { return value + ""; }
}
class AddNode implements ExprNode {
   final ExprNode left;
   final ExprNode right;
   public AddNode(ExprNode lt, ExprNode rt){ left=lt; right=rt; }
   public String toString() { return "("+left+"+"+right+")"; }
}
class SubNode implements ExprNode {
   final ExprNode left;
   final ExprNode right;
   public SubNode(ExprNode lt, ExprNode rt){ left=lt; right=rt; }
   public String toString() { return "("+left+"-"+right+")"; }
}
```

A toString() method lets us print out the program




Prints in regular infix notation (with brackets)

calls the toString() method automatically

# What we can do with an AST : 2 - evaluate/execute

- We can evaluate/execute parse trees in AST form

```
interface ExprNode {
   public int evaluate();
}
```

Every ExprNode must have an evaluate() method that returns the value of the sub-expression

```
class NumNode implements ExprNode{
   ...
   public int evaluate() { return this.value; }
}
```

```
class AddNode implements ExprNode{
   ...
   public int evaluate() {return left.evaluate() + right.evaluate(); }
}
```

Recursive DFS evaluation of expression tree

```
class SubNode implements ExprNode{
   ...
   public int evaluate() {return left.evaluate() - right.evaluate(); }
}
```

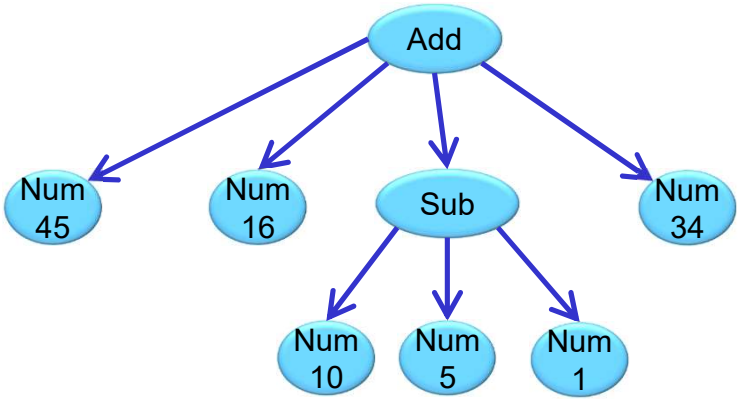
# Extending the Language 1

- Extend the language to allow 2 or more arguments:

```
Expr  ::=  Num   | Add  |  Sub  | Mul  |  Div
Add   ::=  “add”  “(”  Expr  [ “,”  Expr ]+  “)”
Sub   ::=  “sub”  “(“  Expr  [ “,”  Expr ]+  “)”
Mul   ::=  “mul”  “(”  Expr  [ “,”  Expr ]+  “)”
Div   ::=  “div”  “(“  Expr  [ “,”  Expr ]+  “)”
```

sub(16, 8, 2, 1) = 16 – 8 – 2 – 1

“add(45, 16, sub(10, 5, 1), 34)”



# Extending the language 1: Change the Node Classes

```
class AddNode implements ExprNode {
   final List<ExprNode> operands;
   public AddNode(List<ExprNode> ops){
      operands = ops;
   }
   public String toString(){
      String ans = "(" + operands.get(0);
      for (int i=1; i<operands.size(); i++){
        ans += " " + "+ operands.get(i); }
      return ans + ")";
   }
   public int evaluate(){
      int ans = 0;
      for (ExprNode op : operands) { ans += op.evaluate(); }
      return ans;
   }
```

Add ::= “add” “(” Expr [ “,” Expr ]+ “)”

# Extending the language 1: Node Classes (using StringBuilder)

```
class AddNode implements ExprNode {
   final List<ExprNode> operands;
   public AddNode(List<ExprNode> ops){
      operands = ops;
   }
   public String toString(){
      StringBuilder ans = new StringBuilder("(");
      ans.append(operands.get(0));
      for (int i=1; i<operands.size(); i++){
        ans.append(" + ").append(operands.get(i)); }
      return ans.append(")").toString();
   }
   public int evaluate(){
     ......
```

Add ::= “add” “(” Expr [ “,” Expr ]+ “)”

StringBuilder is better than adding lots of Strings together.

# Extending the language 1: the parse…. methods

Add ::= “add” “(” Expr [ “,” Expr ]+ “)”

```
public ExprNode parseAdd(Scanner s) {
   List<ExprNode> operands = new ArrayList<ExprNode>();
   require(ADD_PAT, "Expecting 'add'", s);
   require(OPEN_PAT, "Missing '('", s);
   operands.add(parseExpr(s));
   do {
      require (COMMA_PAT, "Missing ','", s);
      operands.add(parseExpr(s));
   } while (!s.hasNext(CLOSE_PAT));
   require(CLOSE_PAT, "Missing ')'", s);
   return new AddNode(operands);
}
```

?

# Examples of multiple arguments.

Expr: add(10 ,-8, 2)

Print → (10 + -8 + 2)
Value → 4

Expr: add(sub(10 ,-7), mul(div(45, 5), 6), 3)

Print → ((10 – –7) + ((45 / 5.0) * 6) + 3)
Value → 74

Expr: add(14, sub(mul(div (1, 28), 17), mul(3, div(5, sub(7, 5))))) no of arguments?

Print → (14 + (((1 / 28) * 17) – (3 * (5 / (7 – 5)))))
Value → 8

# Extending the language 2: Conditional expressions

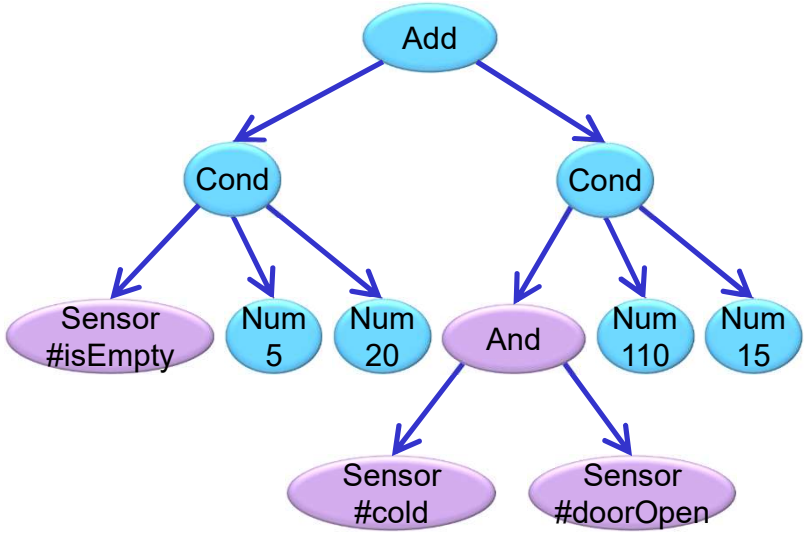
- Suppose the expression language used for customizing what is displayed for a smart home system which includes a number of sensors.
  - The sensors report on the state of the house: #isEmpty, #nighttime, #cold, #windowsOpen, ….
  - The expressions specify what values should be calculated and displayed
  - The expressions should be able to include the sensors using conditional expressions such as:

    mul(34, add(15, if(#isEmpty, 5, 30), if(and(#cold, #doorOpen), 110, 10)))

- To include sensors, if-expressions, boolean operators, need to
  - extend the grammar
  - define new node classes (Including a new category of nodes)
  - define new parse…. methods

# Extending the language 2: Conditional expressions

```
Expr    ::= Num  | Add  |  Sub  | Mul  |  Div  | Cond
Add     ::= “add”  “(”  Expr  [ “,”  Expr ]+  “)”
Sub     ::= “sub”  “(”  Expr  [ “,”  Expr ]+  “)”
Mul     ::= “mul”  “(”  Expr  [ “,”  Expr ]+  “)”
Div     ::= “div”  “(”  Expr  [ “,”  Expr ]+  “)”
Cond    ::= “if”  “(”  Bool “,”  Expr  “,”  Expr  “)”
Bool    ::= Sensor  |  And  |  Or  |  Not
And     ::= “and”  “(“  Bool  [ “,”  Bool ]+  “)”
Or      ::= “or”  “(“  Bool  [ “,”  Bool ]+  “)”
Not     ::= “not”  “(“  Bool  “)”
Num     ::=  matches “[-+]?[0-9]+”
Sensor  ::= matches "#[a-zA-Z]+"
```




add(if(#isEmpty, 5, 20), if(and(#cold, #doorOpen), 110,15))

# Extending the language 2: Node classes: BoolNode

- Bool nodes (for the if statement) are different from ExprNodes:
  - ExprNodes have an evaluate() method that returns an int
  - BoolNodes have an evaluate() method that returns a boolean
  - Therefore, they can't be the same interface

```
public interface ExprNode {
   public int evaluate();
}
public interface BoolNode {
   public boolean evaluate();
}
```

# Extending the language 2: Node classes: CondNode

Cond ::= “if” “(” Bool “,” Expr “,” Expr “)”

```
class CondNode implements ExprNode {
   final BoolNode condition;
   final ExprNode trueExp;
   final ExprNode falseExp;

   public CondNode(BoolNode cnd, ExprNode texp, ExprNode fexp){
     condition = cnd;  trueExp=texp; falseExp=fexp;
   }

   public String toString() {
     return "if("+condition+" then "+trueExp+" else "+falseExp+")";
   }

   public int evaluate() {
     if (condition.evaluate()){ return trueExp.evaluate(); }
     else                     { return falseExp.evaluate(); }
   }
}
```

# Extending the language 2: Node classes: SensorNode

Sensor ::= *matches* "#[a-zA-Z]+"

```
class SensorNode implements BoolNode {
   final String sensorName;
   public SensorNode(String sname){
     sensorName = sname;
   }
   public String toString() {
     return "sensor:"+sensorName;
   }
   public boolean evaluate () {
     return houseSystem.getSensorValue(sensorName);
   }
}
```

# Extending the language 2: Node Classes: AndNode

And ::= “and” “(” Bool [ “,” Bool ]+ “)”

```
class AndNode implements BoolNode {
   final List<BoolNode> conjuncts;
   public AddNode(List<BoolNode> cnjcts){ conjuncts = cnjcts; }
   public String toString(){
      StringBuilder ans = new StringBuilder("(");
      ans.append(conjuncts.get(0));
      for (int i=1; i<args.size(); i++){
        ans.append(" & ").append(conjuncts.get(i));}
      ans.append(")");
      return ans.toString();
   }
   public boolean evaluate(){
      for (BoolNode conjunct : conjuncts) {
        if (!conjunct.evaluate()) {return false; }
      }
      return true;
   }
```

Similar to an AddNode, except BoolNodes instead of ExprNodes

# Extending the language 2: Node Classes: OrNode, NotNode

```
class OrNode implements BoolNode {
   final List<BoolNode> disjuncts;
   ...[similar to AndNode]...
 }
```

Or ::= “or” “(” Bool [ “,” Bool ]+ “)”

```
 class NotNode implements BoolNode {
   final BoolNode expr;
   public NotNode(BoolNode> exp){ expr = exp; }
   public String toString(){
     return "!"+ expr;
   }
   public boolean evaluate(){
     return !expr.evaluate();
   }
```

Not ::= “not” “(” Bool “)”

# Extending the language 2: the parse… methods: parseBool

Bool ::= Sensor | And | Or | Not

```
public BoolNode parseBool(Scanner s) {
   if (!s.hasNext())              { fail("Empty Boolean expr",s); }
   if (s.hasNext(SENSOR_PAT))     { return parseSensorNode(s);}
   if (s.hasNext(AND_PAT))        { return parseAndNode(s); }
   if (s.hasNext(OR_PAT))         { return parseOrNode(s); }
   if (s.hasNext(NOT_PAT))        { return parseNotNode(s); }
   fail("not a Boolean expression", s);
   return null;
}
```

# Extending the language 2: the parse…. methods: parseAnd

And ::= “and” “(” Bool [ “,” Bool ]+ “)”

```
public BoolNode parseAnd(Scanner s) {
   List<BoolNode> conjuncts = new ArrayList<BoolNode>();
   require(AND_PAT, "Expecting 'and'", s);
   require(OPEN_PAT, "Missing '('", s);
   conjuncts.add(parseBool(s));
   do {
      require (COMMA_PAT, "Missing ','", s);
      conjuncts.add(parseBool(s));
   } while (!s.hasNext(CLOSE_PAT));
   require(CLOSE_PAT, "Missing ')'", s);
   return new AndNode(conjuncts);
}
```

Just like parseAdd, but parseBool instead of parseExpr

## Summary: building a parser (for a "nice" grammar)

- interfaces for each category of node
  - Different return types of the evaluate/execute method => different category
- classes for each node type (corresponding to each non-terminal)
  - fields for the components and a constructor
  - toString() to print out nicely (including the subcomponents); [StringBuilder to build up strings]
  - evaluate() or execute() method, recursively called on the subcomponent nodes.
- methods to parse each non-terminal
  - "choice" non-terminals: peek at next token and call appropriate parse method
  - require(..) for each structural token (like "add" or ",")
  - recursive calls for the components.
  - loops if there are repeated components (need to work out when to stop the loop!)
  - build and return the node