Admin

- Term Test1 this week
- Thursday 5pm-6pm
  - MCLT101 (A-F),
  - MCLT103 (G-N),
  - KKLT303 (P-Z)
- Do the previous tests/exams, Assign1 (at least Stage 0)

- Help desks start this week, open to all
  - Wed, Thur, Fri

- Also tutorials, comp261-help@ecs,vuw,ac,nz

- My office hour for this week: Tuesday 9-10

Extending the language 2: Conditional expressions

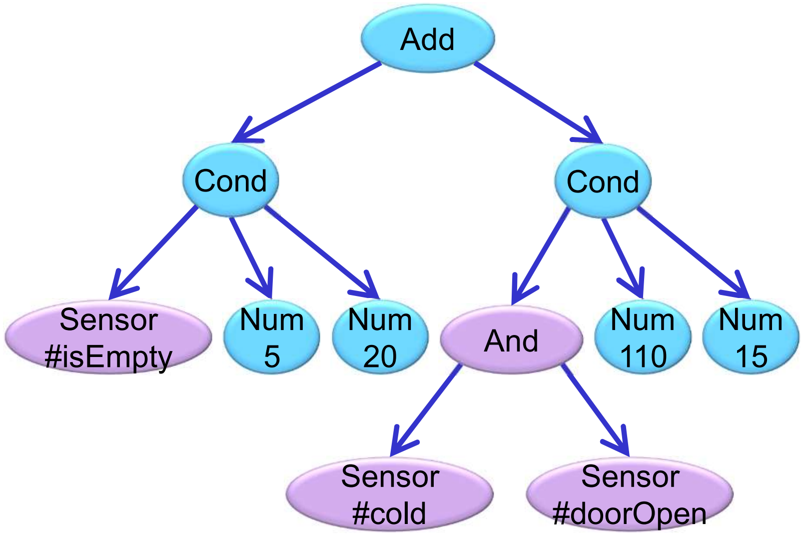
- Suppose the expression language used for customizing what is displayed for a smart home system which includes a number of sensors.
  - The sensors report on the state of the house: #isEmpty, #nighttime, #cold, #windowsOpen, ….
  - The expressions specify what values should be calculated and displayed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

```
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(); }
   }
}
```

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

Extending the language 2: Node classes: SensorNode

```
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);
   }
}
```

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

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

Recursive Descent Parsing

- Recursive Descent Parsing works on LL(1) grammars:
  - top-down (works down from the top of the parse tree, with expectations of what's next)
  - deterministic, (always knows what choice to make next)
  - one-token lookahead (bases choice on the next token only)
  - left-to-right.

- When can the LL(1) conditions fail? Can we fix it?
  - Two options start with the same token
    - May be able to fix the grammar by "left factoring"
  - Some grammars are *ambiguous* - multiple possible parse trees.
    - May be able to force "right-recursion" to make it deterministic
    - May be able to change grammar to respect operator precedence (eg ‘BEDMAS’)
    - it’s tricky : there’s lots more to grammars

4 example grammars (or bits of) that "fail" LL(1)

- CMD := FILE "delete" ";" | FILE "copy" ";"
- IFSTMT ::= "if" "(" COND ")" STMT | "if" "(" CONT ")" STMT "else" STMT
- LIST ::= *id* | LIST "," LIST
- E ::= *number* | E "+" E | E "–" E | E "*" E | E "/" E

All these fail LL(1). The last two are also ambiguous.

Fixing the unclear choices: Factoring

CMD := FILE "delete" ";" | FILE "copy" ";"

IFSTMT ::= "if" "(" COND ")" STMT | "if" "(" COND ")" STMT "else" STMT

- Factor out the common first part:

CMD := FILE OP
OP := "delete" ";"| "copy" ";"

IFSTMT ::= "if" "(" COND ")" STMT
RESTIF ::= "else" STMT | ""

ParseIfStmt code.

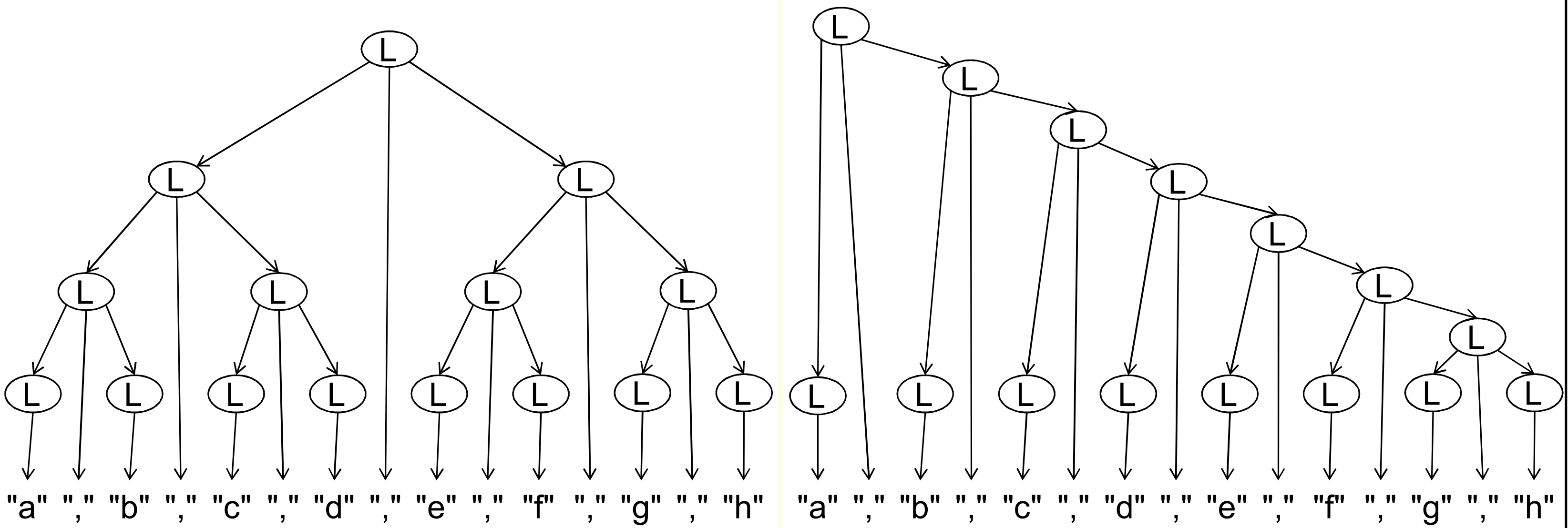
```
public Node parseIfStmt(Scanner s) {
   require(IF_PAT, “Missing ‘if’”, s);
   require(LEFT_PAT, “Missing ‘(‘”, s);
   BooleanNode cond = parseBoolean(s);
   require(RIGHT_PAT, “Missing ‘(‘”, s);
   ProgramNode thenPart = parseStmt(s);
   ProgramNode elsePart = parseRestIf(s);
   return new IfNode(cond, thenPart, elsePart);
}

public Node parseRestIf(Scanner s) {
   if ( s.hasNext(ELSE_PAT) ) { s.next(); return parseStmt(s);}
   else { return null; }
}
```

Ambiguous grammars

- LIST ::= *id* | LIST "," LIST

a, b, c, d, e, f, g, h



Fixing ambiguous grammars (a)

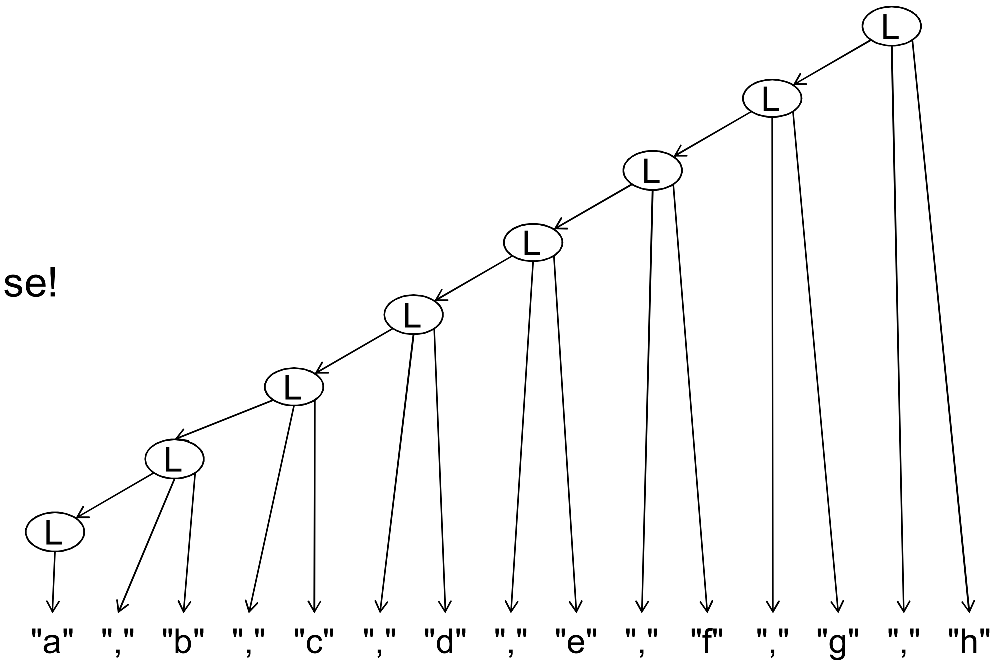
Force left recursion:

LIST ::= *id* | LIST "," *id*

Now it is unambiguous!

BUT

We can't tell which option to use!



Fixing ambiguous grammars (a)

Force right recursion:

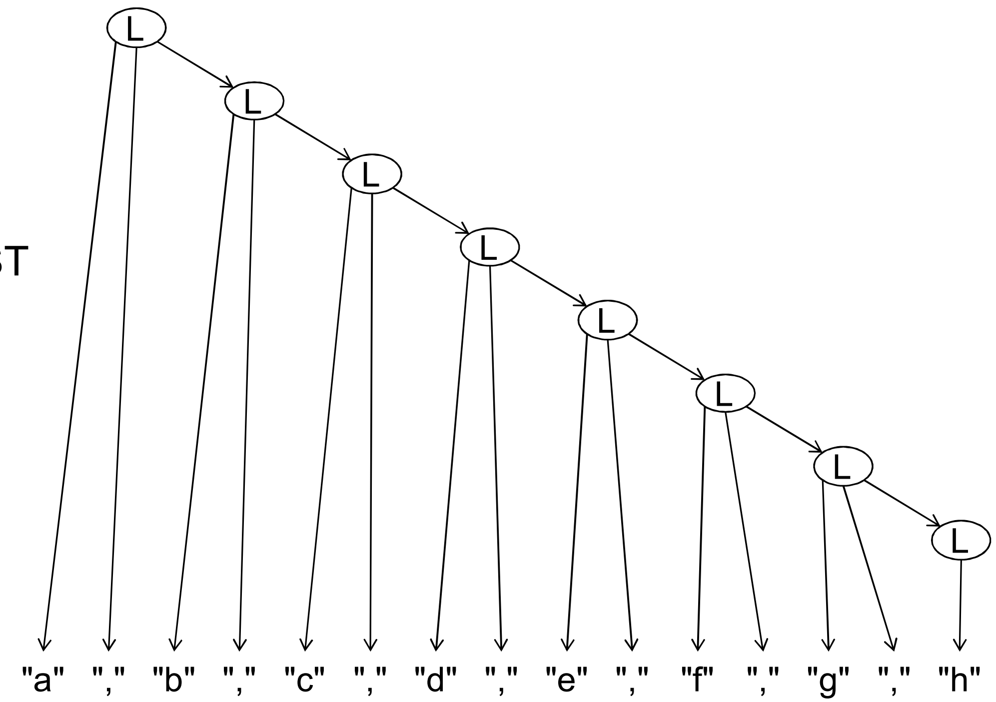
LIST ::= *id* | *id* "," LIST

Factoring solves the choice:

LIST ::= *id* RESTLIST

RESTLIST ::= "," LIST | ""

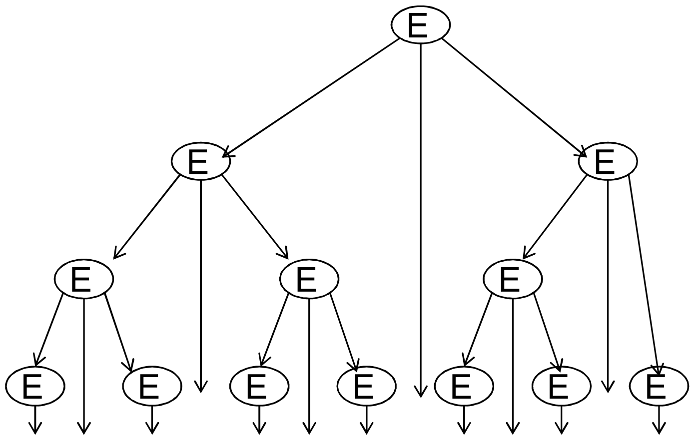
Use same coding trick
to not make RESTLIST nodes



Ambiguous Grammars

Grammar:

E ::= *number* | E "+" E | E "–" E | E "*" E | E "/" E

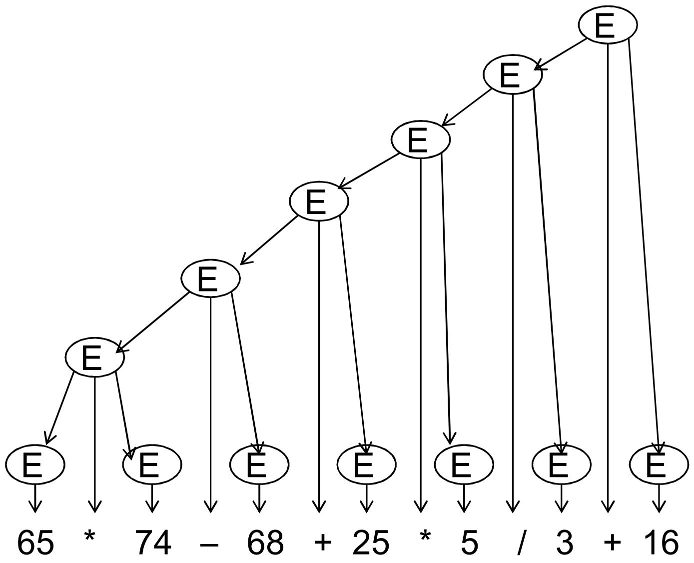


Text: 65 * 74 – 68 + 25 * 5 / 3 + 16

Possible Parses

Grammar:

E ::= *number* | E “+” E | E “–” E | E “*” E | E “/” E



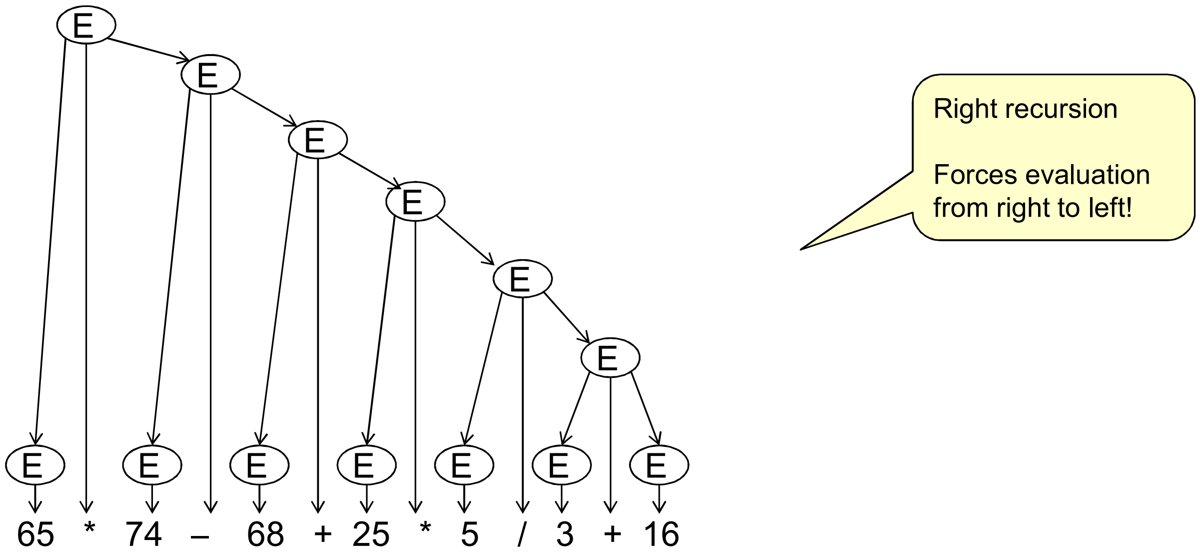
Left recursion

This is what 4-function calculators do!

Possible Parses

Grammar:

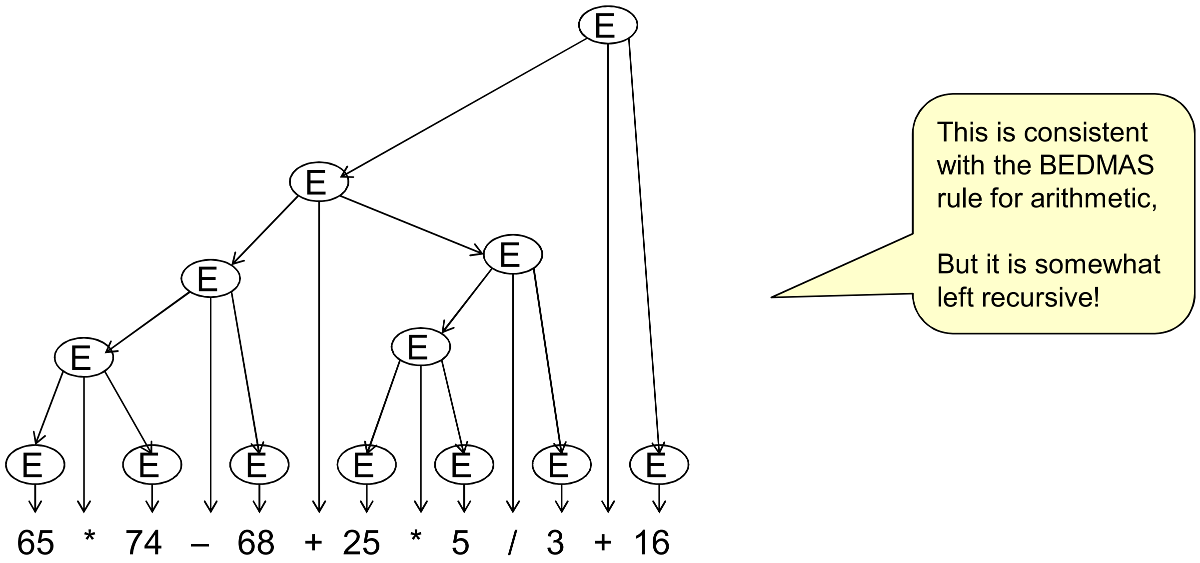
E ::= *number* | E "+" E | E "–" E | E "*" E | E "/" E



Possible Parses

Grammar:

E ::= *number* | E "+" E | E "–" E | E "*" E | E "/" E



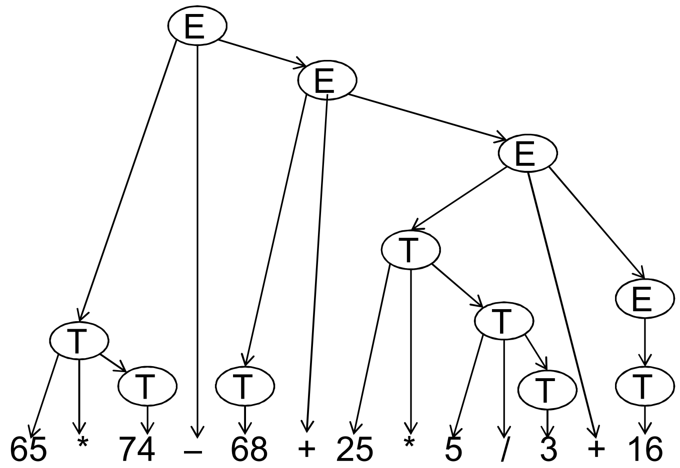
Fixing Ambiguous Grammars (b)

- Use Operator Precedence & Right Recursion to resolve ambiguity.

Right recursion

EXPR ::= TERM | TERM "+" EXPR | TERM "–" EXPR
TERM ::= FACTOR | FACTOR "*" TERM | FACTOR "/" TERM
FACTOR ::= *number* | "(" EXPR ")"

Three "levels" of non-terminals forces BEDMAS operator precedence



Also need to factor the rules

EXPR ::= TERM | TERM "+" EXPR | TERM "–" EXPR

TERM ::= FACTOR | FACTOR "*" TERM | FACTOR "/" TERM

FACTOR ::= *number* | "(" EXPR ")"

Problem: all options start the same way
Which should we choose?

- Factor:

EXPR ::= TERM RESTOFEXPR

RESTOFEXPR ::= "+" EXPR | "–" EXPR | ∊

TERM ::= *number* RESTOFTERM

RESTOFTERM ::= "*" TERM | "/" TERM | ∊

FACTOR ::= *number* | "(" EXPR ")"

∊ means "empty string"

- Transformations such as these can often turn a problematic grammar into a tractable grammar, but not always!!