COMP261 Parsing 4 of 4

Marcus Frean

When does recursive descent work?

Victoria
UNIVERSITY OF WELLINGTON
Te Whare Wānanga
o te Ūpoko o te Ika a Māui
CAPITAL CITY UNIVERSITY

Today: when does recursive descent work / fail?

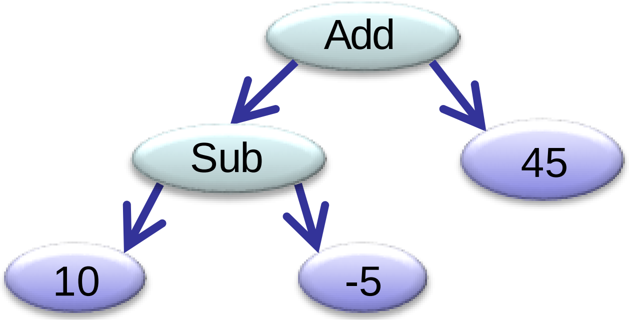
The bullet-point summary of this lecture!

- What can we do with an AST? Evaluate it, print it,…
- the LL(1) condition – failures, and what could help, sometimes.
  - left factoring (when possible) can ensure LL(1) condition
  - but some grammars are *ambiguous*, e.g. list-like example
    - converting to left-recursion: unambiguous ☺, but fails LL(1) ☹
    - right-recursion instead: ☺ ☺, *and yet:*
  - we might also care about operator precedence, e.g. "infix" example
    - neither left- or right-recursion respects operator precedence ☹ (a.k.a. 'BEDMAS')
    - it's tricky : there's lots more to grammars
- **Take-Home Message: it's easy to stray beyond LL(1), and easy to end up with a parse tree that evaluates in the wrong order!**

✓ end of parsing section

What can we do with an AST?

“add(sub(10, -5), 45)”



- **We can "execute" parse trees!**

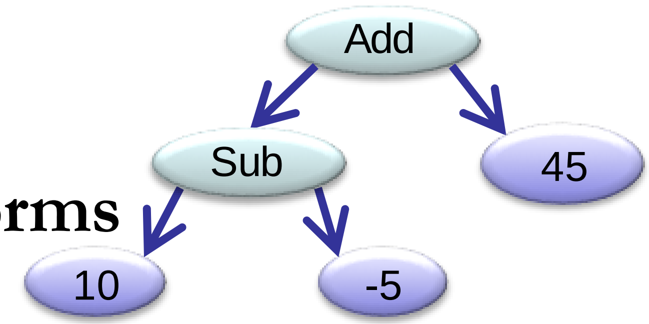
```
interface Node {
   public int evaluate();
}
```

```
class NumberNode implements Node {
   ...
   public int evaluate() { return this.value; }
}
```

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

Recursive DFS evaluation of expression tree "add(sub(10, -5), 45)"

What can we do with an AST?



- **We can print expressions in other forms**

```
class AddNode implements Node {
   private Node left, right;
   public AddNode(Node lt, Node rt) {
      left = lt;
      right = rt;
   }
   public int evaluate() {
      return left.evaluate() + right.evaluate();
   }

   public String toString() {
      return "(" + left + " + " + right + ")";
   }
}
```

eg: print in human-friendly "infix" notation (with brackets)
→ → ((10- -5)+45)

Extending the Language

- **Allow floating point numbers as well as integers**
  - need more complex patterns for numbers.

```
class NumberNode implements Node {
   final double value;
   public NumberNode(double v) {
      value= v;
   }
   public String toString() {
      return String.format("%.5f", value);
   }
   public double evaluate() { return value; }
}
```

Extending the Language: Examples

Expression: **add**(10.5 ,-8)

Print → (10.5 + -8.0)

Value → 2.500

---

add(sub(10.5 ,-8), **mul**(**div**(45, 5), 6.8))

Print → ((10.5 – –8.0) + ((45.0 / 5.0) * 6.8))

Value → 79.700

---

add(14.0, **sub**(**mul**(**div** (1.0, 28), 17), **mul**(3, **div**(5, **sub**(7, 5)))))

Print → (14.0 + (((1.0 / 28.0) * 17.0) – (3.0 * (5.0 / (7.0 – 5.0)))))

Value → 7.107

Exercise: Can you minimize the number or brackets used?

Extending the Language to >2 arguments

Expr ::= Num | Add | Sub | Mul | Div

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

Sub ::= “sub” “(“ Expr [“,” Expr]+ “)”

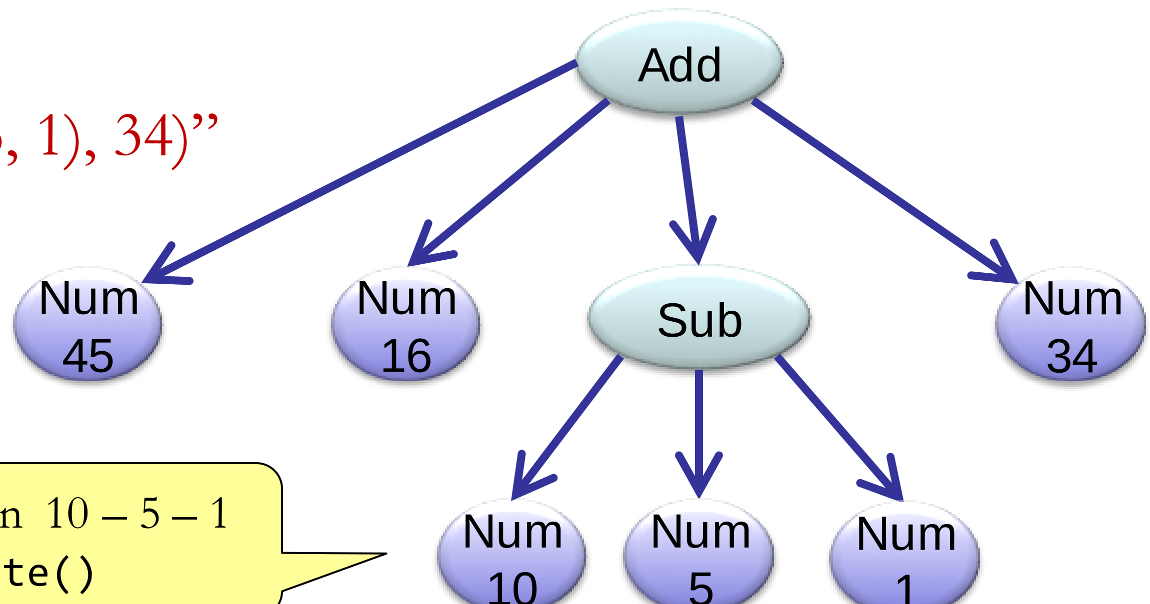
Mul ::= “mul” “(” Expr [“,” Expr]+ “)”

Div ::= “div” “(“ Expr [“,” Expr]+ “)”

note use of regex “[…]+”

‘at least one of’

“add(45, 16, sub(10, 5, 1), 34)”



eg: “sub(10, 5, 1)” *could* mean 10 – 5 – 1
That’s determined by `evaluate()`

Extending Node Classes (to allow >2 args)

```
class AddNode implements Node {
    final List<Node> args;
    public AddNode(List<Node> nds) {
        args = nds;
    }
    public String toString() {
        String ans = "(" + args.get(0);
        for (int i=1;i<args.size(); i++) {
            ans += " + "+ args.get(i);
        }
        return ans + ")";
    }
    public double evaluate() {
        double ans = 0;
        for (nd : args) { ans += nd.evaluate(); }
        return ans;
    }
```

Extending the Parser (to allow >2 args)

Allow add(1,2,3), etc.

```
public Node parseAdd(Scanner s) {
   List<Node> args = new ArrayList<Node>();
   require(addPat, "Expecting add", s);
   require(openPat, "Missing '('", s);
   args.add(parseExpr(s));
   do {
      require (commaPat, "Missing ','", s);
      args.add(parseExpr(s));
    } while (!s.hasNext(closePat));
    require(closePat, "Missing ')'", s);
    return new AddNode(args);
 }
```

(need new version of require, taking a Pattern instead of a String)

Recursive Descent Parsing (ie. "*LL(1)*") - recap

- Method for each nonterminal/Node type
- Peek at next token to determine which branch to follow
- Build and return node
- Throw error (including error message) when parsing breaks
- Use require(…) to wrap up "check, then do stuff or fail"
- Adjust grammar to make it cleaner
- LL(1) = deterministic, left-to-right, top down parsing with one symbol lookahead

When does it work?

If we have a grammar rule involving *choices*:

$$N ::= W_1 \mid W_2 \mid \ldots \mid W_n$$

we must be able to tell which alternative to take, by looking just at the next input token.

<u>LL(1) condition:</u>

For any i and j (where $j \neq i$) there is no symbol that can start **both** an instance of W_i and an instance of W_j.

- Easy to check if W_i and W_j start with terminals.
- What if they start with nonterminals?...

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

- `IfStmt ::= “if” “(“ Cond “)” Stmt   |`
  `“if” “(“ Cond “)” Stmt “else” Stmt`

an IF statement

- `A := B “c” | B “d”`

shared leading NT (has less obvious versions)

- `E ::=` *`num`* `| E “+” E | E “-” E | E “*” E | E “/” E`

“infix” arithmetic expression

- `L ::=` *`id`* `| L “,” L`

list

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

What can we do? - Left-factoring

- Consider this grammar rule:

```
IfStmt ::= "if" "(" Cond ")" Stmt  |
           "if" "(" Cond ")" Stmt "else" Stmt
```

- If we see an "if", we can't tell which branch to take.
- We can fix this by "factoring" out the common part:

```
IfStmt ::= "if" "(" Cond ")" Stmt RestIf
RestIf ::= "" | "else" Stmt
```

Notice the Empty string. (Aside: need to test it *last* out of the options)

We can now parse this ok, with:

```
• public Node parseIfStmt(Scanner s) {
    require(ifPat, “Missing ‘if’”, s);
    require(leftBracPat, “Missing ‘(‘”, s);
    Node c = parseExp(s);
    require(rightBracPat, “Missing ‘(‘”, s);
    Node thenPart = parseStmt(s);
    Node elsePart = parseRestIf(s);
    return new IfNode(c, thenPart, elsePart);
  }
  public Node parseRestIf(Scanner s) {
    if ( s.hasNext(elsePat) ) {
      s.next(); return parseStmt(s);
    } else { return null; }
```

assemble pieces…

make node

Taking the empty branch if no other branch is possible. Using null to represent empty.

More about Left-factoring

on this slide, nonterminals are upper case, and terminals lowercase

- We can apply this idea to lots of grammars.

```
A ::= B c | B d
```

fails LL(1)

```
A ::= B E
E ::= c | d
```

left-factored

```
A ::=  B c |  D e
B ::=  f g |  h i
D ::=  h j |  k l
```

fails LL(1)

left-factored

```
A ::= h M | f g c | k l e
M ::= i c | j e
```

- These can be done using simple algebraic laws – just like simplifying Boolean expressions

When does it work?

- Consider the following grammar for lists of identifiers separated by commas.
- Informally, a list is either an identifier, or two lists separated by a comma.

L ::= *id* | L "," L

- This grammar is *ambiguous* – we can construct more that one parse tree for some strings.
- Ex: Draw all parse trees for "a,b,c".
- Recursive descent doesn't work for ambiguous grammars – must be able to construct a unique parse tree for any text in the language.

ambiguous

`L ::= `*`id`*`  |  L “,” L`

`a,b,c`

- imagine what happens with `a,b,c,d,e,…` !!

Left-recursion

- We can rewrite the grammar as:

  L ::= *id* | L "," *id*

  left recursive, *yet same language!*
- This is unambiguous – draw parse trees as before.
- But … any L starts with an *id*.
- So, if we see an *id* we can' t tell which branch to take!

  fails LL(1)
- In this case, we can't factor out the common parts.
- Ex: try it!

- Generalising: Recursive descent doesn't work for grammars with left-recursive rules (where the nonterminal on the left occurs at the start of some branch on the right)

Right-recursion

- We can also rewrite the grammar as:

  L ::= *id* | *id* "," L

  right recursive, *yet same language!*

  This is also unambiguous – draw parse trees as before.

- And now we can factor out the common parts, to ensure the LL(1) condition

  L ::= *id* R   (or L ::= *id* ["," L])

  R ::= "" | "," L

But what does this do to the parse trees? Does it matter?

Infix expressions

- Consider the following grammar for arithmetic expressions:

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

  This, again, is ambiguous – can get many different parse trees for some expressions.

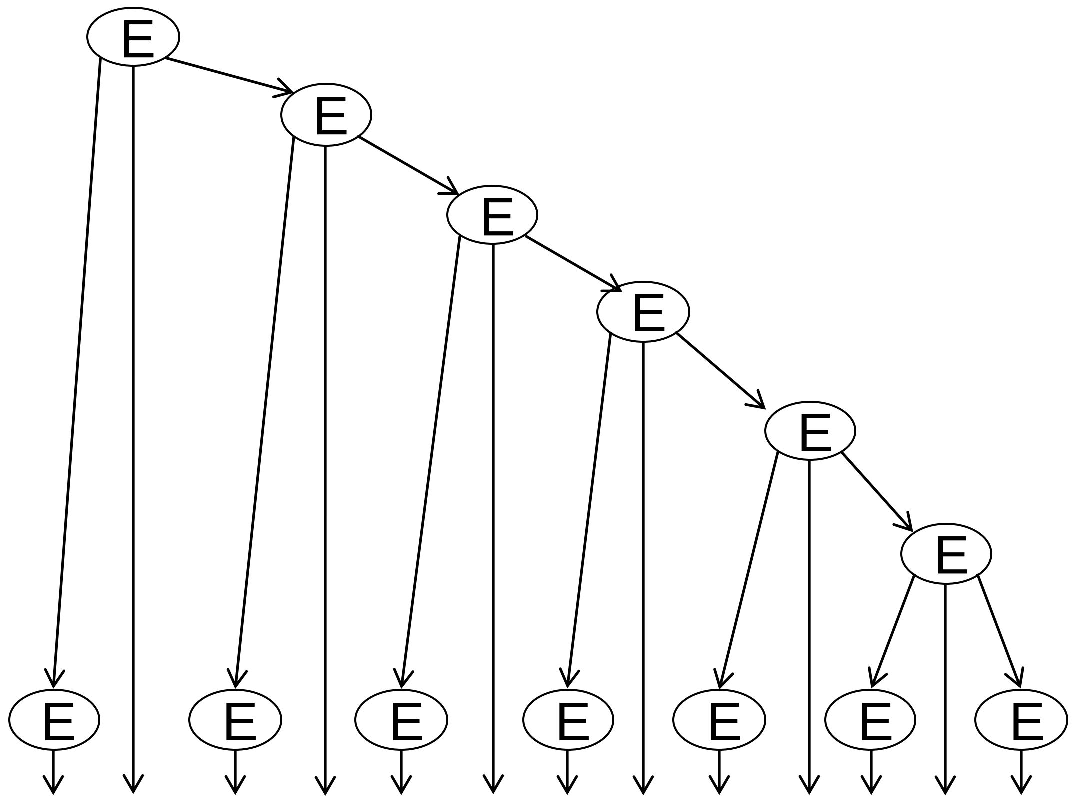
- Does it matter which parse tree we use?

- Think about order of evaluation!

Infix expressions

Grammar:

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

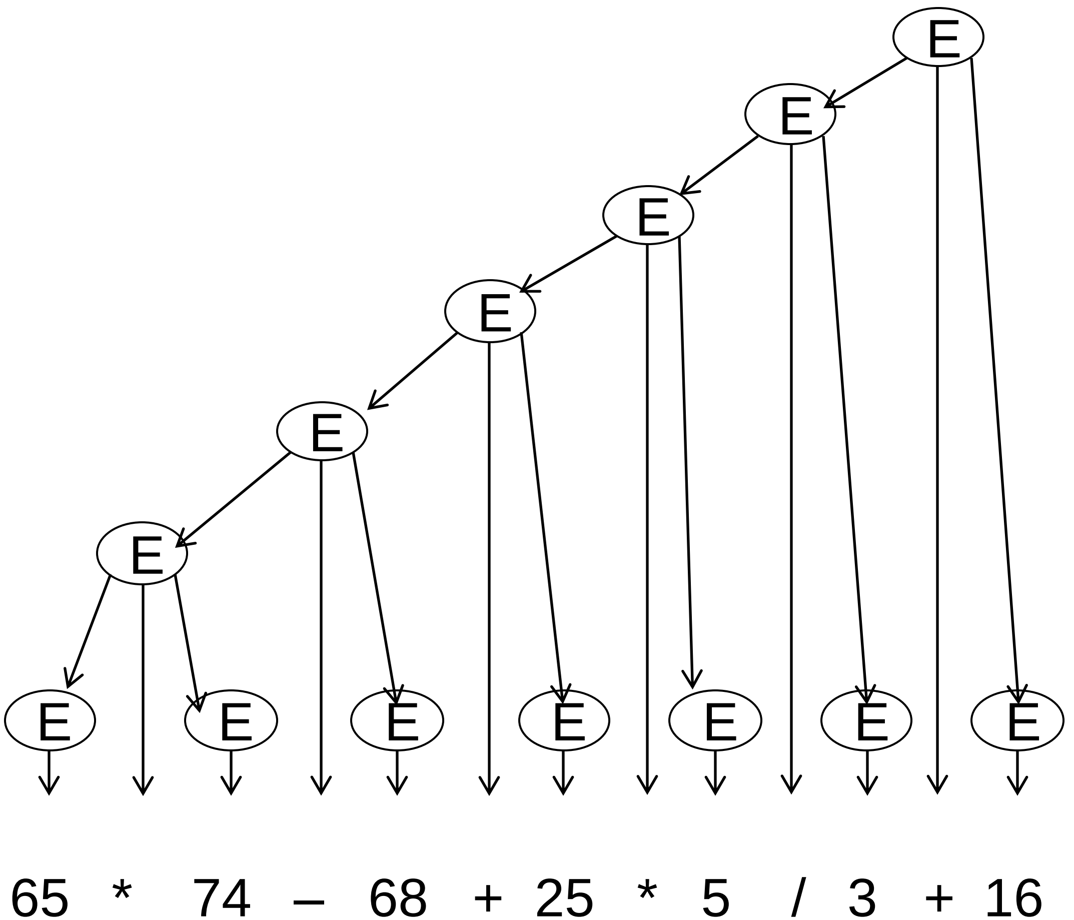


65 * 74 – 68 + 25 * 5 / 3 + 16

Infix expressions

Grammar:

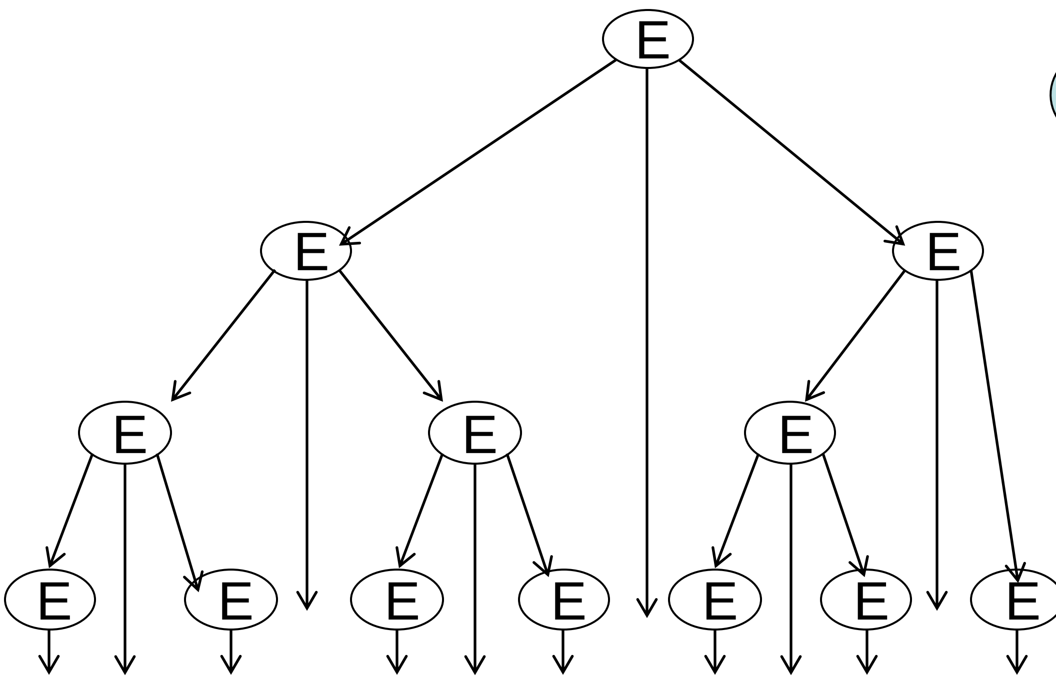
E ::= *number* | E "+" E | E "–" E | E "*" E | E "/" E

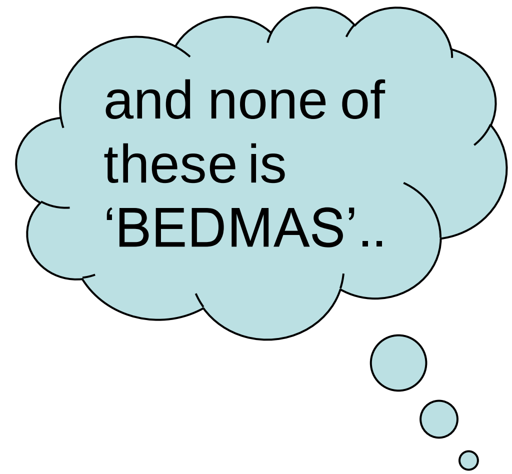


Infix expressions

Grammar:

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





65 * 74 – 68 + 25 * 5 / 3 + 16

Infix Expressions

- We can make the grammar unambiguous by making it right-recursive, as for lists.

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

- And the make it LL(1) by left-factoring:

  EXPR ::= *number* RESTOFEXPR

  RESTOFEXPR ::= "+" EXPR | "–" EXPR | "*" TERM | "/" TERM | ""

- What does this do to the parse tree?
- Is that what we want?

Infix Expressions

- We could handle precedence by introducing an extra nonterminal.

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

  TERM ::= *number* | *number* "*" TERM | *number* "/" TERM | "(" EXPR ")"

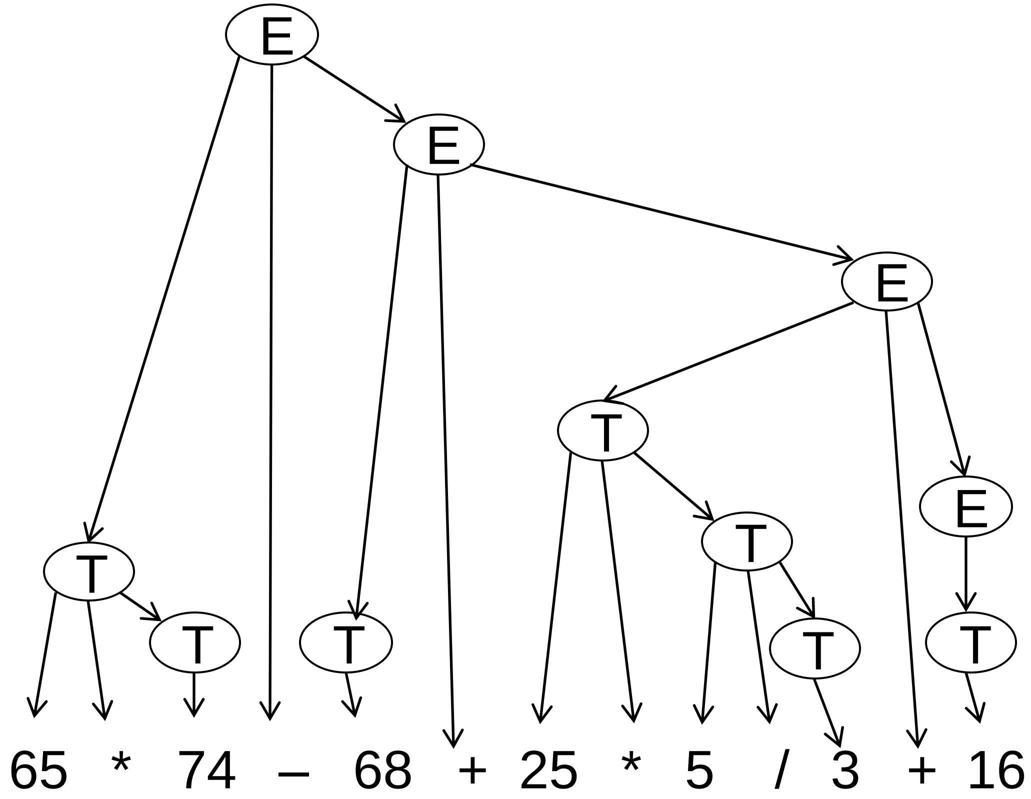
- And then make that LL(1) by left-factoring:

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

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

- this works! (see next slide)

Infix Expressions



(steps toward) a more practical approach

- Instead of

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

- Write:

  E ::= *number* [Op *number*]*

  Op := "+" | "–" | "*" | "/"

- And the parser as:

```
parseNum(s);
while (s.hasNext(opPat)) {
  s.next();
  parseNum(s);
 }
```

A more practical approach (just checking here)

- What about operator precedence: * before +, etc?
- Grammar:

  E ::= T [(“+” | ”-”) T]*   Expression
  T ::= F [(“*” | ”/”) F]*   Term
  F ::= number | “(“ E “)”   Factor

- Parser:

```
public parseE(s) {
   parseT;
   while (s.hasNext(addOpPat)) { // + or -
     s.next();  // grab the operator
     parseT(s);
    }
 }
```

A more practical approach (build parse tree)

- Now extend to build a parse tree

```
public Node parseE(Scanner s) {
   Node t = parseT(Scanner s);
   while (s.hasNext(addOpPat)) { // + or -
      String op = s.next();
      Node r = parseT(s);
      if ( op == "add" )
           t = new AddNode(t, r);
      else t = new SubNode(t, r);
     }
     return t;
  }
```

- Yay, but the take-home msg: that was not trivial.

Today: when does recursive descent work / fail?

- the LL(1) condition – failures, and what could help, sometimes.
  - left factoring (when possible) can ensure LL(1) condition
  - but some grammars are *ambiguous*, e.g. list-like example
    - converting to left-recursion: unambiguous ☺, but fails LL(1) ☹
    - right-recursion instead: ☺ ☺, *and yet:*
  - we might also care about operator precedence, e.g. "infix" example
    - neither left- or right-recursion respects operator precedence ☹ (a.k.a. 'BEDMAS')
    - it's tricky : there's lots more to grammars

- **Take-Home Message: it's easy to stray beyond LL(1), and easy to end up with a parse tree that evaluates in the wrong order!**

✓ end of parsing section