Mapping Priorities to Trees in Recursive Descent Parsers

- summary and exercise -

Expressions with Two Operators

expr ::= ident | expr - expr | expr ^ expr | (expr)

where:

- “-” is left-associative
- “^” is right-associative
- “^” has higher priority than “-”

Draw parentheses and a tree for token sequence: **a – b – c ^ d ^ e – f**

((a – b) – (c ^ (d ^ e))) – f

Goal: Build Expressions

```
abstract class Expr
case class Variable(id : Identifier) extends Expr
case class Minus(e1 : Expr, e2 : Expr) extends Expr
case class Exp(e1 : Expr, e2 : Expr) extends Expr
```

1) Layer the grammar by priorities

expr ::= ident | expr - expr | expr ^ expr | (expr)

⇓

expr ::= term (- term)*

term ::= factor (^ factor)*

factor ::= id | (expr)

lower priority binds weaker,
so it goes outside

2) Build trees in the right way

LEFT-associative operator

x – y – z ➔ (x – y) – z

Minus(Minus(Var(“x”),Var(“y”)), Var(“z”))

```
def expr : Expr = {
  var e = term
  while (lexer.token == MinusToken) {
    lexer.next
    e = Minus(e, term)
  }
  e
}
```

2) Build trees in the right way

RIGHT-associative operator – using a loop

x ^ y ^ z ➔ x ^ (y ^ z)

Exp(Var(“x”), Exp(Var(“y”), Var(“z”)))

```
def expr : Expr = {
  val e = factor
  if (lexer.token == ExpToken) {
    lexer.next
    Exp(e, expr)
  } else e
}
```