A Rule of While Language Syntax

// Where things work very nicely for recursive descent!

```
statmt ::=
        println ( stringConst , ident )
      | ident = expr
      | if ( expr ) statmt (else statmt)?
      | while ( expr ) statmt
      | { statmt* }
```

Parser for the statmt (rule -> code)

```
def skip(t : Token) = if (lexer.token == t) lexer.next
  else error("Expected"+ t)
// statmt ::=
def statmt = {
  // println ( stringConst , ident )
  if (lexer.token == Println) { lexer.next;
    skip(openParen); skip(stringConst); skip(comma);
    skip(identifier); skip(closedParen)
  // | ident = expr
  } else if (lexer.token == Ident) { lexer.next;
    skip(equality); expr
  // | if ( expr ) statmt (else statmt)?
  } else if (lexer.token == ifKeyword) { lexer.next;
    skip(openParen); expr; skip(closedParen); statmt;
    if (lexer.token == elseKeyword) { lexer.next; statmt }
  // | while ( expr ) statmt
```

Continuing Parser for the Rule

// | while (expr) statmt

} **else if** (lexer.token == whileKeyword) { lexer.next;

skip(openParen); expr; skip(closedParen); statmt

// | { statmt }*

} **else if** (lexer.token == openBrace) { lexer.next;

while (**isFirstOfStatmt**) { statmt }

skip(closedBrace)

} **else** { error("Unknown statement, found token " + lexer.token) }

How the parser decides which alternative to follow?

statmt ::= println (stringConst , ident)
| ident = expr
| if (expr) statmt (else statmt)$^?$
| while (expr) statmt
| { statmt* }

- Look what each alternative starts with to decide what to parse
- Here: we have terminals at the beginning of each alternative!
- More generally, we have 'first'computation, as for regular expresssions
- Consider a grammar G and non-terminal N

$L_G(N)$ = { set of strings that N can derive }

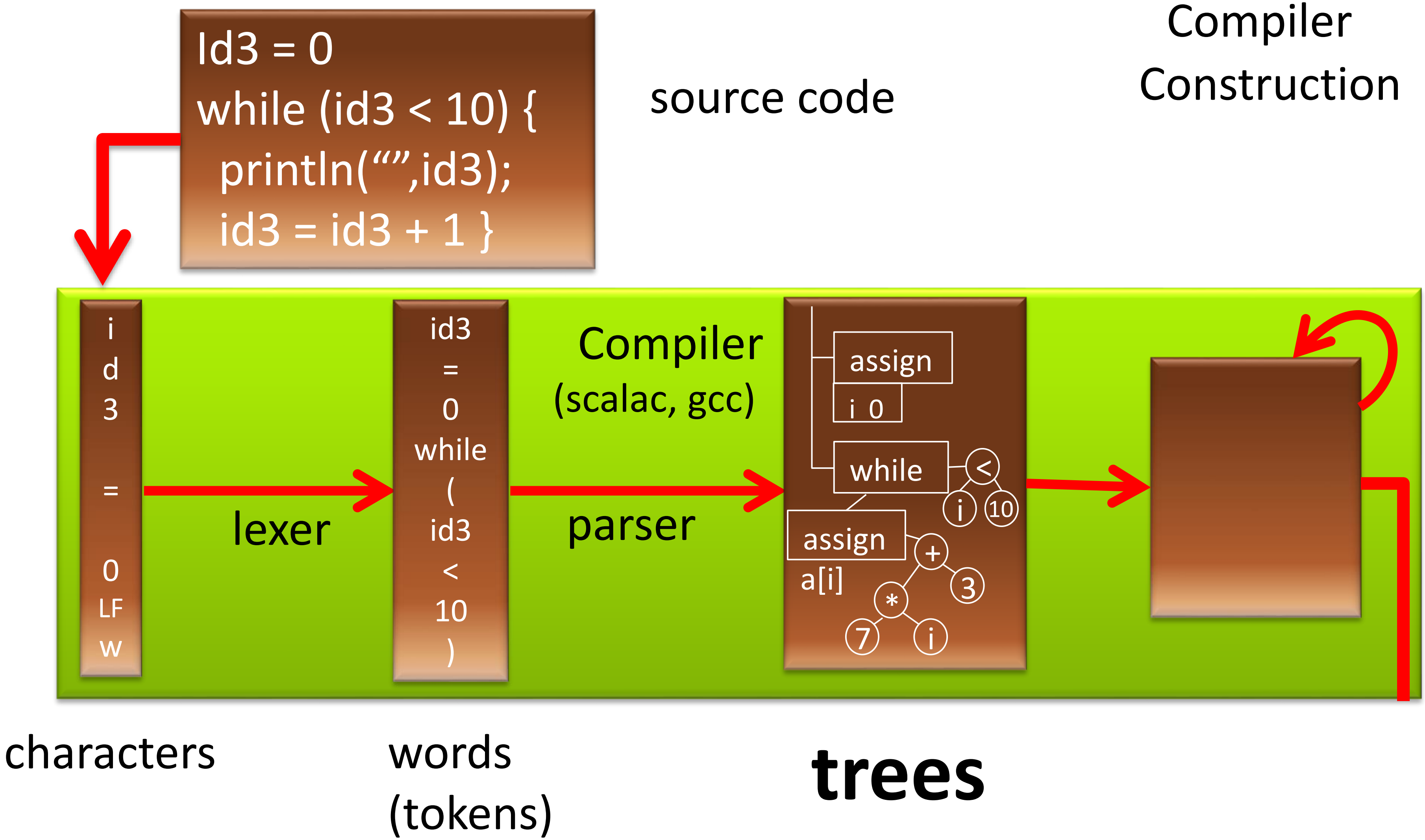
e.g. L(statmt) – all statements of while language

first(N) = { a | aw in $L_G(N)$, a – terminal, w – string of terminals}

first(statmt) = { **println**, **ident**, **if**, **while**, **{** }

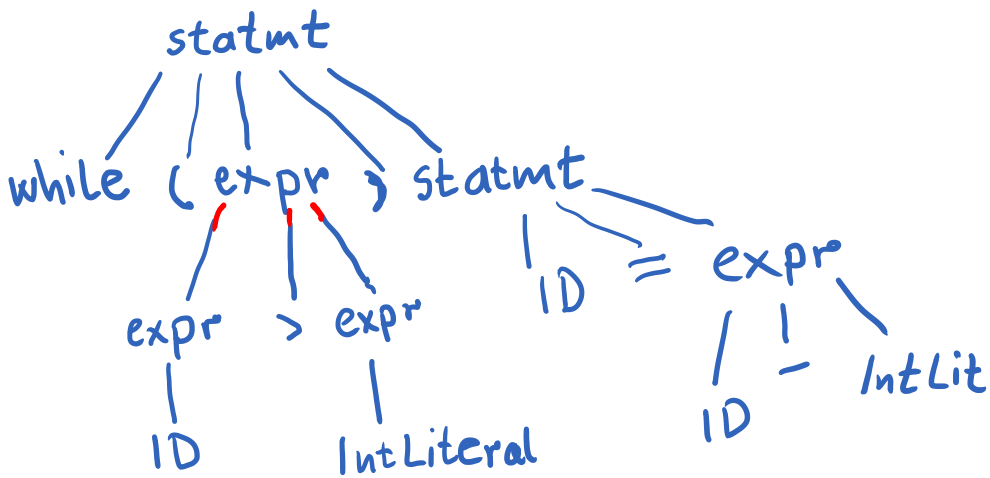
first(**while** (expr) statmt) = { **while** }

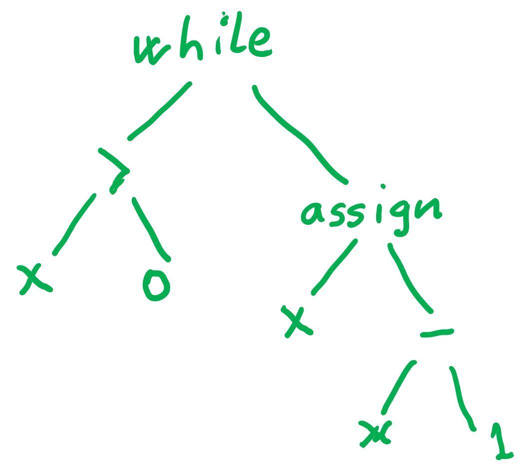
Compiler Construction



Parse Tree vs Abstract Syntax Tree (AST)

while (x > 0) x = x - 1





Pretty printer: takes abstract syntax tree (AST) and outputs the leaves of one possible (concrete) parse tree.

parse(prettyPrint(ast)) ≈ ast

Parse Tree vs Abstract Syntax Tree (AST)

- Each node in **parse tree** has children corresponding **precisely to right-hand side of grammar rules**. The definition of parse trees is fixed given the grammar
  - **Often compiler never actually builds parse trees in memory**

- Nodes in **abstract syntax tree (AST)** contain only useful information and usually omit the punctuation signs. We can choose our own syntax trees, to make it convenient for both construction in parsing and for later stages of compiler or interpreter
  - **A compiler typically directly builds AST**

Abstract Syntax Trees for Statements

```
statmt ::= println ( stringConst , ident )
         | ident = expr
         | if ( expr ) statmt (else statmt)?
         | while ( expr ) statmt
         | { statmt* }
```

```
abstract class Statmt
case class PrintlnS(msg : String, var : Identifier) extends Statmt
case class Assignment(left : Identifier, right : Expr) extends Statmt
case class If(cond : Expr, trueBr : Statmt,
                     falseBr : Option[Statmt]) extends Statmt
case class While(cond : Expr, body : Expr) extends Statmt
case class Block(sts : List[Statmt]) extends Statmt
```

Abstract Syntax Trees for Statements

```
statmt ::= println ( stringConst , ident )
         | ident = expr
         | if ( expr ) statmt (else statmt)?
         | while ( expr ) statmt
         | { statmt* }
```

```
abstract class Statmt
case class PrintlnS(msg : String, var : Identifier) extends Statmt
case class Assignment(left : Identifier, right : Expr) extends Statmt
case class If(cond : Expr, trueBr : Statmt,
              falseBr : Option[Statmt]) extends Statmt
case class While(cond : Expr, body : Statmt) extends Statmt
case class Block(sts : List[Statmt]) extends Statmt
```

Our Parser Produced Nothing ☹

```
def skip(t : Token) : unit = if (lexer.token == t) lexer.next
  else error("Expected"+ t)
// statmt ::=
def statmt : Unit = {
  // println ( stringConst , ident )
  if (lexer.token == Println) { lexer.next;
    skip(openParen); skip(stringConst); skip(comma);
    skip(identifier); skip(closedParen)
  // | ident = expr
  } else if (lexer.token == Ident) { lexer.next;
    skip(equality); expr
```

New Parser: Returning an AST ☺

```
def expect(t : Token) : Token = if (lexer.token == t) { lexer.next;t}
  else error("Expected"+ t)
// statmt ::=
def statmt : Statmt = {
  // println ( stringConst , ident )
  if (lexer.token == Println) { lexer.next;
    skip(openParen); val s = getString(expect(stringConst));
    skip(comma);
    val id = getIdent(expect(identifier)); skip(closedParen)
    PrintlnS(s, id)
 // | ident = expr
  } else if (lexer.token.class == Ident) { val lhs = getIdent(lexer.token)
    lexer.next;
    skip(equality); val e = expr
    Assignment(lhs, e)
```

Constructing Tree for ‘if’

```
def expr : Expr = { … }
// statmt ::=
def statmt : Statmt = {
  …
// if ( expr ) statmt (else statmt)?
// case class If(cond : Expr, trueBr: Statmt, falseBr: Option[Statmt])
  } else if (lexer.token == ifKeyword) { lexer.next;
    skip(openParen); val c = expr; skip(closedParen);
    val trueBr = statmt
    val elseBr = if (lexer.token == elseKeyword) {
      lexer.next; Some(statmt) } else None
    If(c, trueBr, elseBr)   // made a tree node ☺
  }
```

Task: Constructing AST for 'while'

```
def expr : Expr = { ... }
// statmt ::=
def statmt : Statmt = {
  ...
// while ( expr ) statmt
// case class While(cond : Expr, body : Expr) extends Statmt
} else if (lexer.token == WhileKeyword) { lexer.next
      skip(openParen)
   val e = expr
      skip(closedParen)
   val s = statmt
   While(e, s)

} else
```

Here each alternative started with different token

```
statmt ::=
        println ( stringConst , ident )
      | ident = expr
      | if ( expr ) statmt (else statmt)?
      | while ( expr ) statmt
      | { statmt* }
```

What if this is not the case?

Left Factoring Example: Function Calls

statmt ::=

println (stringConst , ident)
| ident = expr
| if (expr) statmt (else statmt)$^?$
| while (expr) statmt
| { statmt* }
| ident (expr (, expr)*)

foo = 42 + x
foo (u , v)

code to parse the grammar:

```
} else if (lexer.token.class == Ident) {
    ???
}
```

Left Factoring Example: Function Calls

$x \cdot y + x \cdot z = x(y+z)$

statmt ::=

println (stringConst , ident)

| ident assignmentOrCall

| if (expr) statmt (else statmt)$^{?}$

| while (expr) statmt

| { statmt* }

assignmentOrCall ::= “=“ expr | (expr (, expr)*)

code to parse the grammar:

```
} else if (lexer.token.class == Ident) {
    val id = getIdentifier(lexer.token); lexer.next
    assignmentOrCall(id)
}              // Factoring pulls common parts from alternatives
```

Beyond Statements: Parsing Expressions

While Language with Simple Expressions

statmt ::=

println (stringConst , ident)

| ident = expr

| if (expr) statmt (else statmt)$^?$

| while (expr) statmt

| { statmt* }

```
expr ::= intLiteral | ident
       | expr ( + | / ) expr
```

| expr + expr

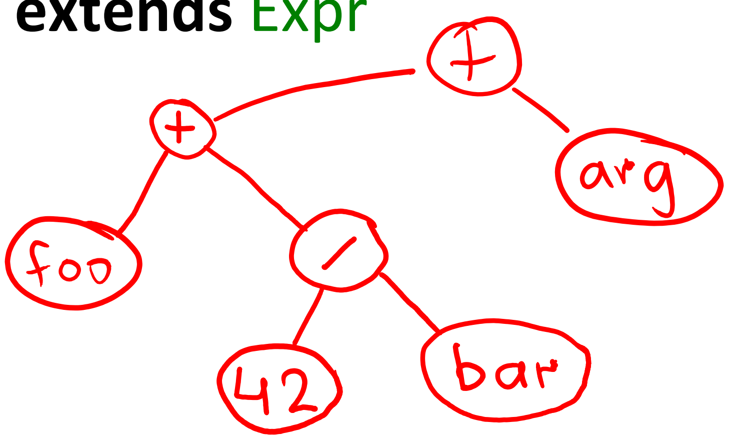
| expr / expr

Abstract Syntax Trees for Expressions

```
expr ::= intLiteral | ident
       | expr + expr | expr / expr
```

abstract class Expr
case class IntLiteral(x : Int) **extends** Expr
case class Variable(id : Identifier) **extends** Expr
case class Plus(e1 : Expr, e2 : Expr) **extends** Expr
case class Divide(e1 : Expr, e2 : Expr) **extends** Expr

foo + 42 / bar + arg



Parser That Follows the Grammar?

```
expr ::= intLiteral | ident
       | expr + expr | expr / expr
```

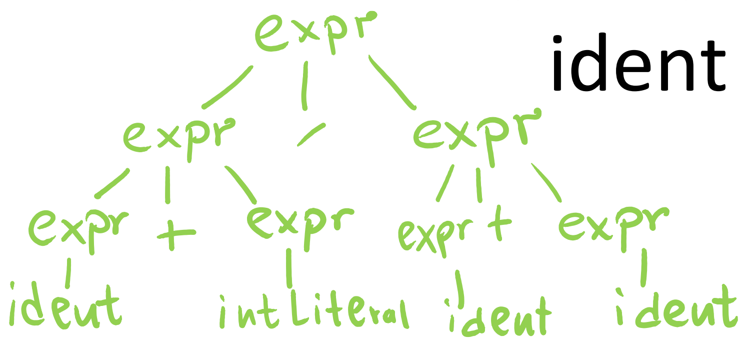
input:
foo + 42 / bar + arg

```
def expr : Expr = {
  if (??) IntLiteral(getInt(lexer.token))
  else if (??) Variable(getIdent(lexer.token))
  else if (??) {
    val e1 = expr; val op = lexer.token; val e2 = expr
    op match Plus {
     case PlusToken => Plus(e1, e2)
     case DividesToken => Divides(e1, e2)
    } }
```

When should parser enter the recursive case?!

Ambiguous Grammars

expr ::= intLiteral | ident
| expr + expr | expr / expr



ident + intLiteral / ident + ident

Each node in parse tree is given by one grammar alternative.

Ambiguous grammar: if some token sequence has multiple parse trees (then it is has multiple abstract trees).

Ambiguous grammar: if some token sequence has multiple parse trees

(then it is usually has multiple abstract trees)

Two parse trees, each following the grammar, their leaves both give the same token sequence.

Ambiguous Expression Grammar

```
expr ::= intLiteral | ident
       | expr + expr | expr / expr
```

ident + intLiteral / ident + ident

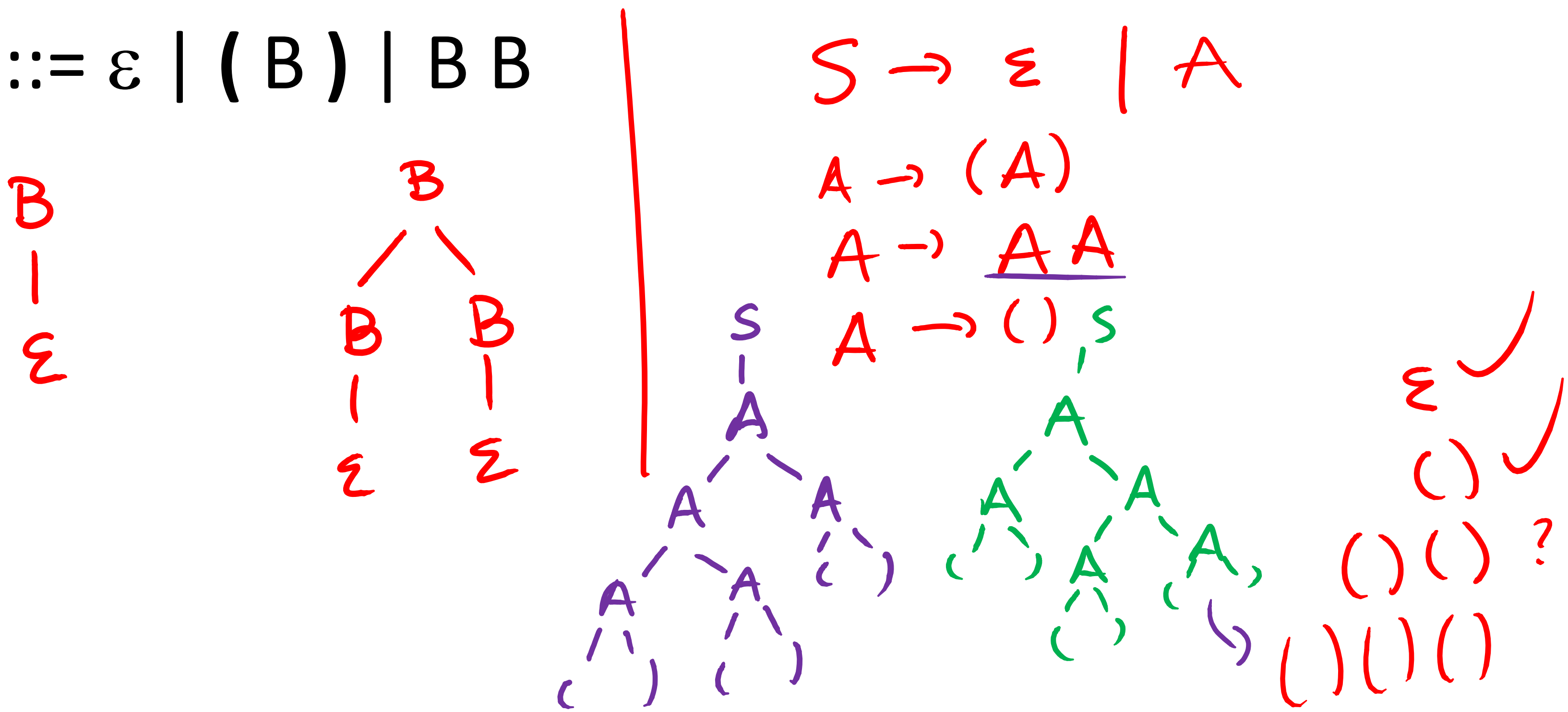
Each node in parse tree is given by one grammar alternative.

Show that the input above has two parse trees!

Exercise: Balanced Parentheses I

Show that the following balanced parentheses grammar is ambiguous (by finding two parse trees for some input sequence).

B ::= ε | (B) | B B



Remark

$B \to \varepsilon \mid (B)B$

- The same parse tree can be derived using two different derivations, e.g.

  B -> (B) -> (BB) -> ((B)B) -> ((B)) -> (())

  B -> (B) -> (BB) -> ((B)B) -> (()B) -> (())

this correspond to different orders in which nodes in the tree are expanded.

- Ambiguity refers to the fact that there are actually multiple *parse trees*, not just multiple derivations.

Exercise: Balanced Parentheses

Show that the following balanced parentheses grammar is ambiguous (by finding two parse trees for some input sequence) **and find unambiguous grammar for the same language.**

B ::= ε | **(** B **)** | B B

Not Quite Solution

- This grammar:

  B ::= ε | A

  A ::= **( )** | A A | (A)

solves the problem with multiple ε symbols generating different trees.

Does string **( ) ( ) ( )** have a unique parse tree?

Solution for Unambiguous Parenthesis Grammar

B ::= ε | BB | (B)

- Proposed solution:

  B ::= ε | B **(**B**)**

- How to come up with it?
- Clearly, rule B::= B B generates any sequence of B's. We can also encode it like this:

  B ::= C*

  C ::= (B)

- Now we express sequence using recursive rule that does not create ambiguity:

  B ::= ε | C B

  C ::= (B)

- but now, look, we "inline" C back into the rules for so we get exactly the rule

  B ::= ε | B **(**B**)**

This grammar is not ambiguous and is the solution. We did not prove unambiguity (we only tried to find ambiguous trees but did not find any).

Exercise:
Left Recursive and Right Recursive

We call a production rule “left recursive” if it is of the form

A ::= A p

for some sequence of symbols p. Similarly, a "right-recursive" rule is of a form

A ::= q A

S ::= A | ε
A ::= A B
A ::= c A

Is every context free grammar that contains both left and right recursive rule for a some nonterminal A ambiguous?

S ::= b
A ::= A c
A ::= bA
A ::= ε

$A \Rightarrow Ac \Rightarrow Acc \Rightarrow bAcc \Rightarrow bcc$

G = (.., S, ...)
L(G) = {b}

An attempt to rewrite the grammar

```
expr ::= simpleExpr (( + | / ) simpleExpr)*
simpleExpr ::= intLiteral | ident
```

```
def simpleExpr : Expr = { … }
def expr : Expr = {
  var e = simpleExpr
  while (lexer.token == PlusToken ||
         lexer.token == DividesToken)) {
    val op = lexer.token
    val eNew = simpleExpr
    op match {
      case TokenPlus => { e = Plus(e, eNew) }
      case TokenDiv => { e = Divide(e, eNew) }
    }
  }
  e }
```

foo + 42 / bar + arg

expr ::= mulExpr
  | mulExpr (+ expr)

mulExpr ::= simpleExpr
  | simpleExpr (/ mulExpr)

Not ambiguous, but gives wrong tree.

Making Grammars Unambiguous
- some useful recipes -

Ensure that there is always only one parse tree

Construct the correct abstract syntax tree

Goal: Build Expression Trees

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

2 ^ 3 ^ 5 ≡ 2^(3^5)   e1 - e2 - e3

$2^{3^5} = 2^{32}$

different parse trees give ASTs:

Minus(e1, Minus(e2,e3))	e1 - (e2 - e3)
Minus(Minus(e1,e2),e3)	(e1 - e2) - e3

1) Layer the grammar by priorities

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

ident - ident ^ ident - ident

expr ::= term (- term)*

term ::= factor (^ factor)*

factor ::= id | (expr)

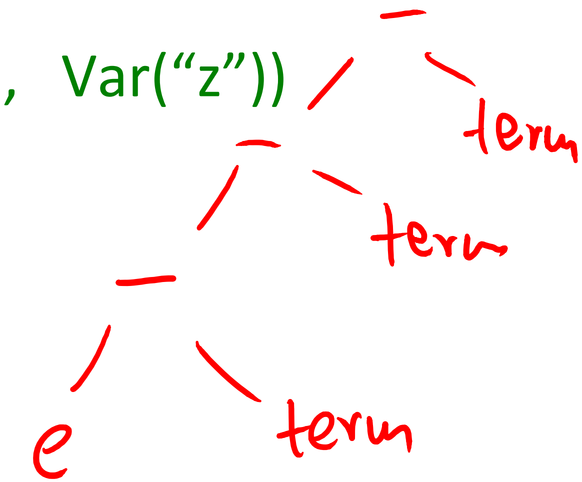
lower priority binds weaker, so it goes outside

2) Building trees: left-associative "-"

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
}
```

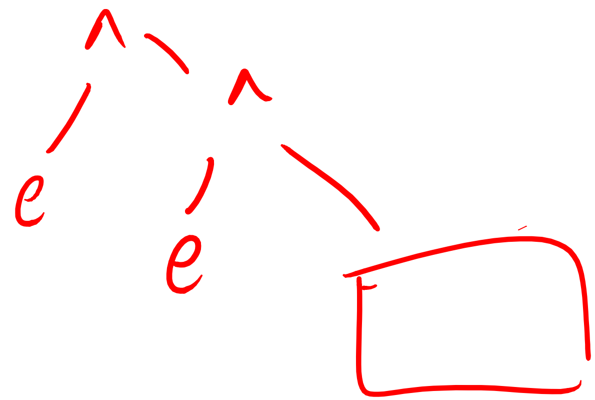
3) Building trees: right-associative "^"

RIGHT-associative operator – using recursion
(or also loop and then reverse a list)

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
}
```



Manual Construction of Parsers

- Typically one applies previous transformations to get a nice grammar
- Then we write recursive descent parser as set of mutually recursive procedures that check if input is well formed
- Then enhance such procedures to construct trees, paying attention to the associativity and priority of operators