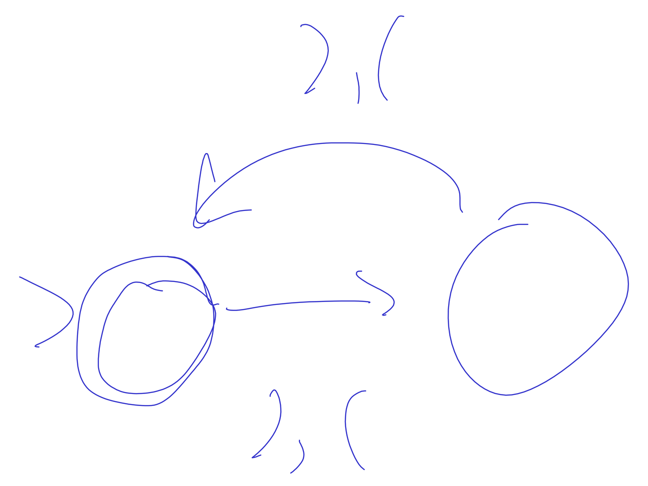
Expressive Power of Automata

For which of the following languages can you find an automaton or regular expression:

- Sequence of open or closed parentheses of even length? E.g. (), ((,)),)()))(, ...



- as many digits before as after decimal point?

1.2
23.35
123.345

- Sequence of balanced parentheses

  ((()) ()) - balanced

  ( )) (( )   - not balanced

- Comments from // until LF
- Nested comments like   /*   ... /*   */ ... */

Expressive Power of Automata

For which of the following languages can you find an automaton or regular expression:

- Sequence of open or closed parentheses of even length? E.g. (), ((,)),)()))(, ... yes
- as many digits before as after decimal point? No
- Sequence of balanced parentheses

  ((()) ()) - balanced

  ( )) (( ) - not balanced

  No
- Comments from // until LF Yes
- Nested comments like /* ... /* */ ... */ No

Automaton that Claims to Recognize $\{ a^n b^n \mid n >= 0 \}$

Make the automaton deterministic

Let the resulting DFA have K states, $|Q|=K$

Feed it a, aa, aaa, …. Let q_i be state after reading a^i

$q_0 , q_1 , q_2 , \ldots , q_K$

This sequence has length K+1 -> a state must repeat

$q_i = q_{i+p}$ $\quad p > 0$

Then the automaton should accept $a^{i+p}b^{i+p}$.

But then it must also accept

$a^i b^{i+p}$

because it is in state after reading a^i as after a^{i+p}.

So it does not accept the given language.

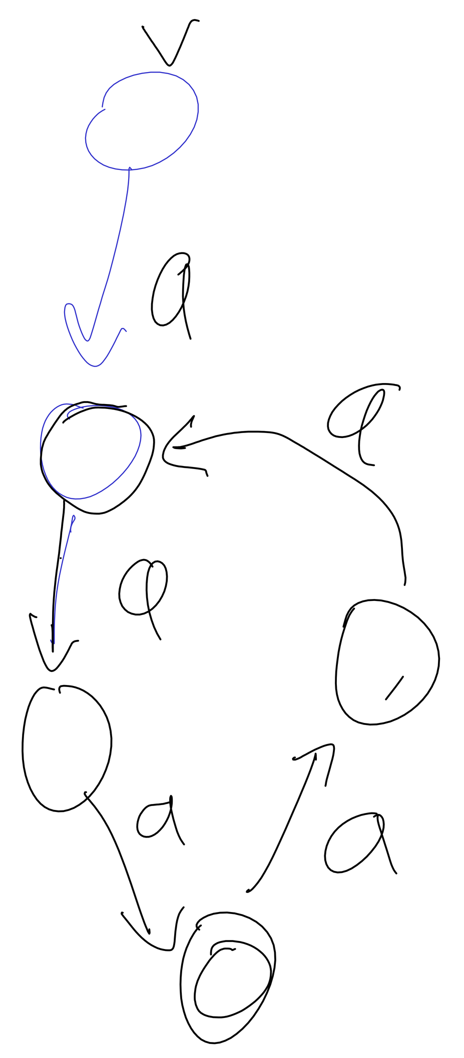
Limitations of Regular Languages

- Every automaton can be made deterministic
- Automaton has finite memory, cannot count
- Deterministic automaton from a given state behaves always the same
- If a string is too long, deterministic automaton will repeat its behavior

Pumping Lemma

If L is a regular language, then there exists a positive integer p (the pumping length) such that every string $s \in L$ for which $|s| \geq p$, can be partitioned into three pieces, $s = x\,y\,z$, such that

- $|y| > 0$
- $|xy| \leq p$
- $\forall i \geq 0.\ xy^iz \in L$



Let's try again: $\{\ a^nb^n\ |\ n >= 0\ \}$

Finite State Automata are Limited

Let us use (context-free) **grammars**!

Context Free Grammar for $a^n b^n$

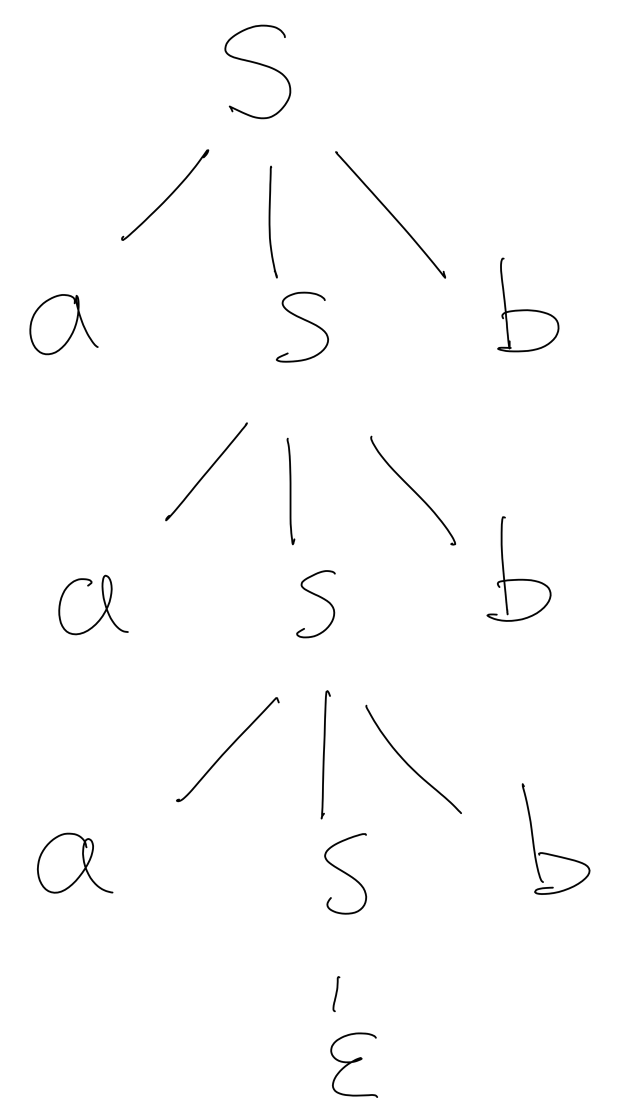
$S ::= \varepsilon$ - first rule of this grammar

$S ::= a\ S\ b$ - second rule of this grammar.

Example of a derivation (DEMO)

$S \Rightarrow aSb \Rightarrow a\ aSb\ b \Rightarrow aa\ aSb\ bb \Rightarrow aaabbb$

Parse tree: leaves give us the result



Context-Free Grammars

G = (A, N, S, R)

- A - terminals (alphabet for generated words w ∈ A*)
- N - non-terminals – symbols with (recursive) definitions
- Grammar rules in R are pairs (n,v), written

  n ::= v   where

  n ∈ N is a non-terminal

  v ∈ (A U N)* - sequence of terminals and non-terminals

$\in A$

$S \to aSb$

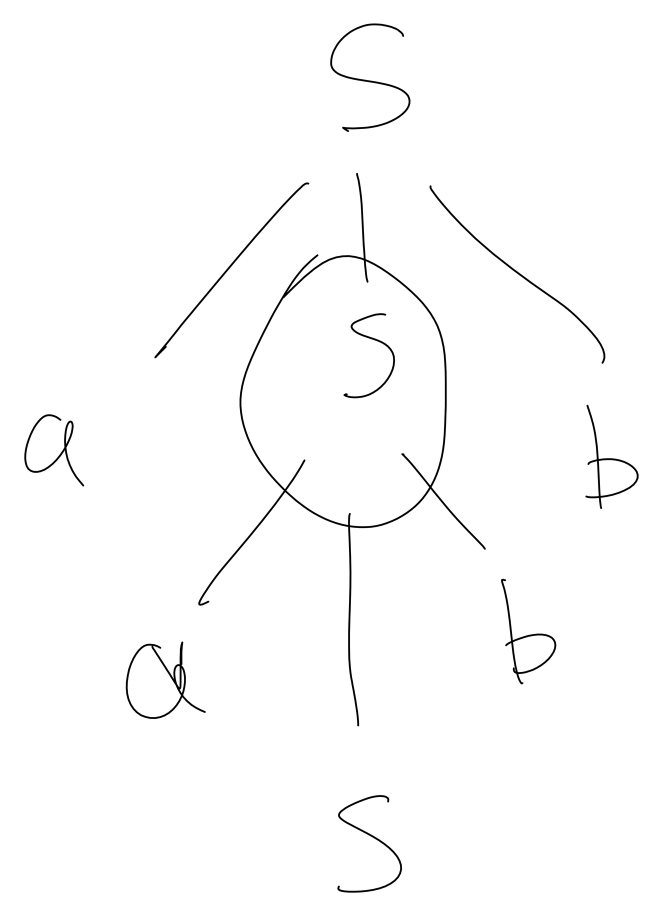
A derivation in G starts from the starting symbol S ∈ N

- Each step replaces a non-terminal with one of its right hand sides

Example from before: G = ({a,b}, {S}, S, {(S,ε), (S,aSb)})

Parse Tree

$S \to aSb$



Given a grammar G = (A, N, S, R), t is a **parse tree** of G
iff t is a node-labelled tree with ordered children that satisfies:

- root is labeled by S
- leaves are labelled by elements of A
- each non-leaf node is labelled by an element of N
- for each non-leaf node labelled by n $\in N$ whose children left to right are labelled by $p_1 \ldots p_n$, we have a rule $(n ::= p_1 \ldots p_n) \in R$

Yield of a parse tree t is the unique word in A^* obtained by reading the leaves of t from left to right

Language of a grammar G = words of all yields of parse trees of G

$L(G) = \{yield(t) \mid isParseTree(G,t)\}$

$w \in L(G) \quad \Leftrightarrow \quad \exists t.\ w = yield(t) \land isParseTree(G,t)$

isParseTree - **easy** to check condition, given t

Harder: know *if for a word there **exists** a parse tree*

Grammar Derivation

A **derivation** for G is any sequence of words $p_i \in (A \cup N)^*$, whose:

- first word is S
- each subsequent word is obtained from the previous one by replacing one of its letters by right-hand side of a rule in R :
  $p_i = unv$, $(n::=q) \in R$,
  $p_{i+1} = uqv$
- Last word has only letters from A

Each parse tree of a grammar has one or more derivations, which result in expanding tree gradually from S

- Different orders of expanding non-terminals may generate the same tree
- Leftmost derivation: always expands leftmost non-terminal
  - Rightmost derivation: always expands rightmost non-terminal

Remark

We abbreviate

$S ::= p$

$S ::= q$

as

$S ::= p \mid q$

Example: Parse Tree vs Derivation

Consider this grammar G = ({a,b}, {S,P,Q}, S, R) where R is:

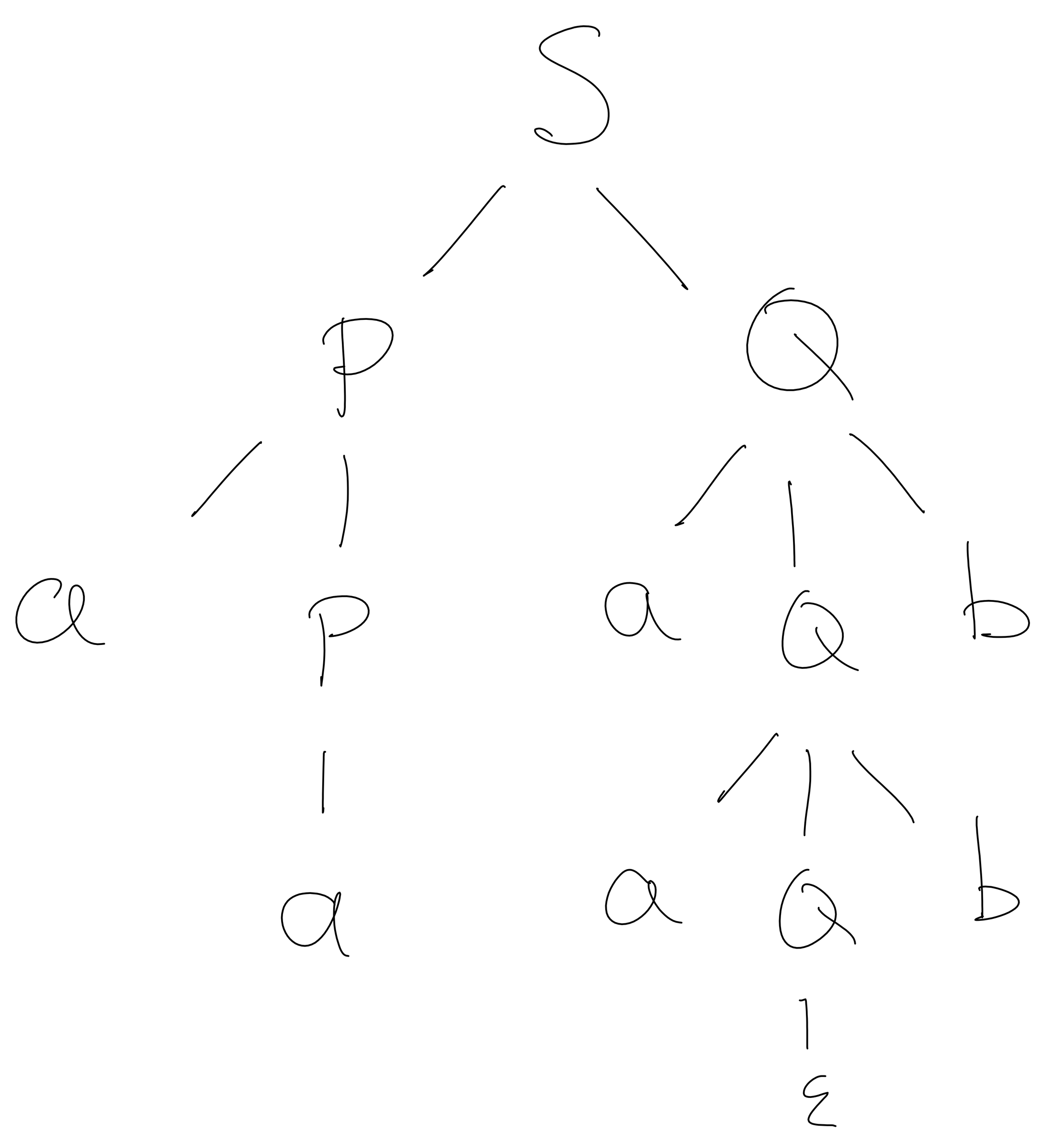
S ::= PQ

P ::= a | aP

Q ::= ε | aQb

Show a ~~derivation~~ parse tree for aaaabb

Show at least two derivations that correspond to that tree.



$S \to PQ \to aPQ \to aaQ \to aaaQb \to aaaaQbb$
$\to aaaabb$

$S \to PQ \to PaQb \to PaaQbb \to Paabb \to$
$\to aPaabb \to aaaabb$

Balanced Parentheses Grammar

$S \to \varepsilon \mid S(S)S$

Consider the language L consisting of precisely those words consisting of parentheses “**(**“ and “**)**” that are balanced (each parenthesis has the matching one)

- Example sequence of parentheses

((()) ()) - balanced, belongs to the language

( )) (( )   - not balanced, does not belong

Exercise: give the grammar and example derivation for the first string.

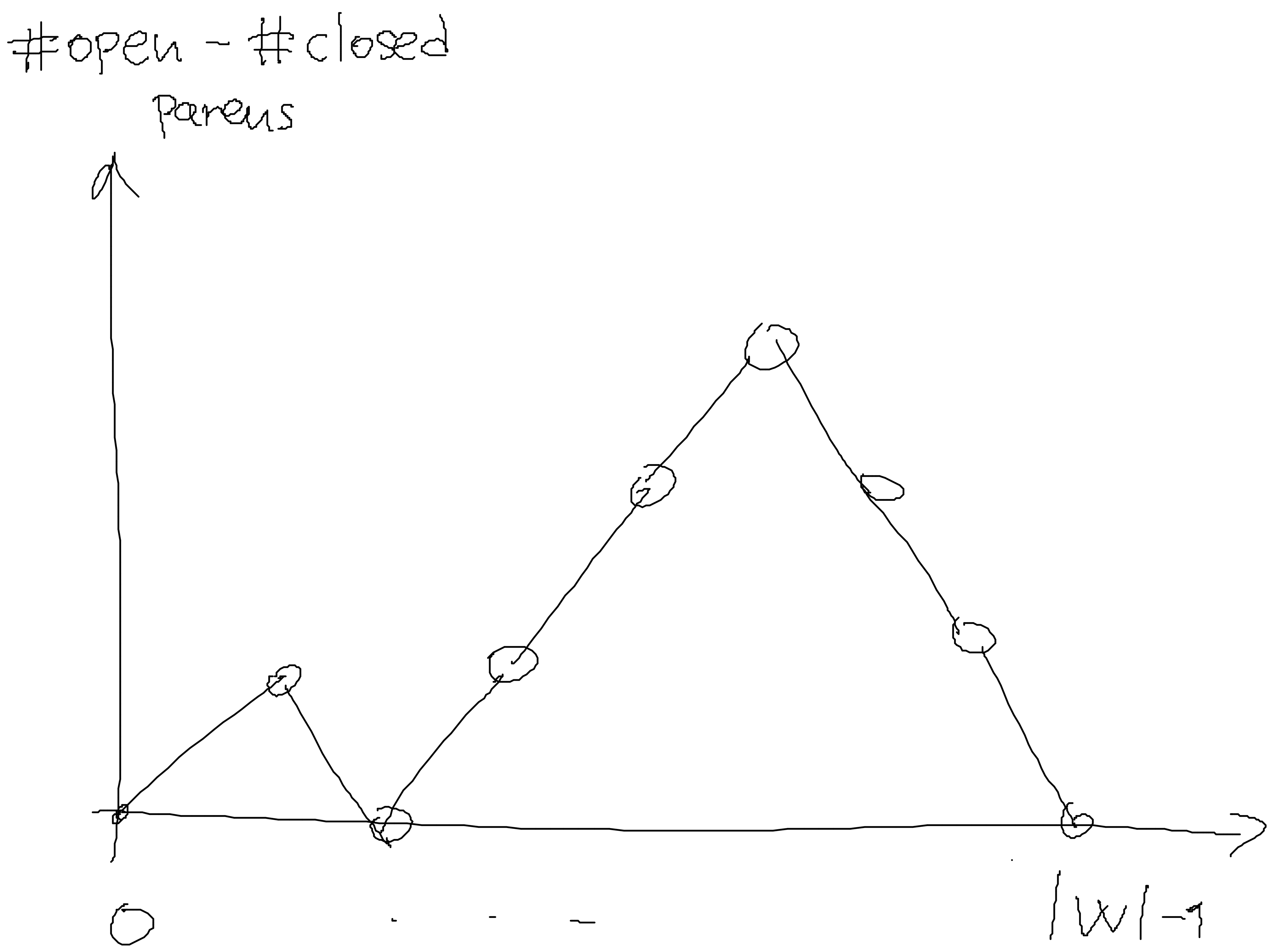
Balanced Parentheses Grammar

G_1 S ::= ε | S(S)S

G_2 S ::= ε | (S)S

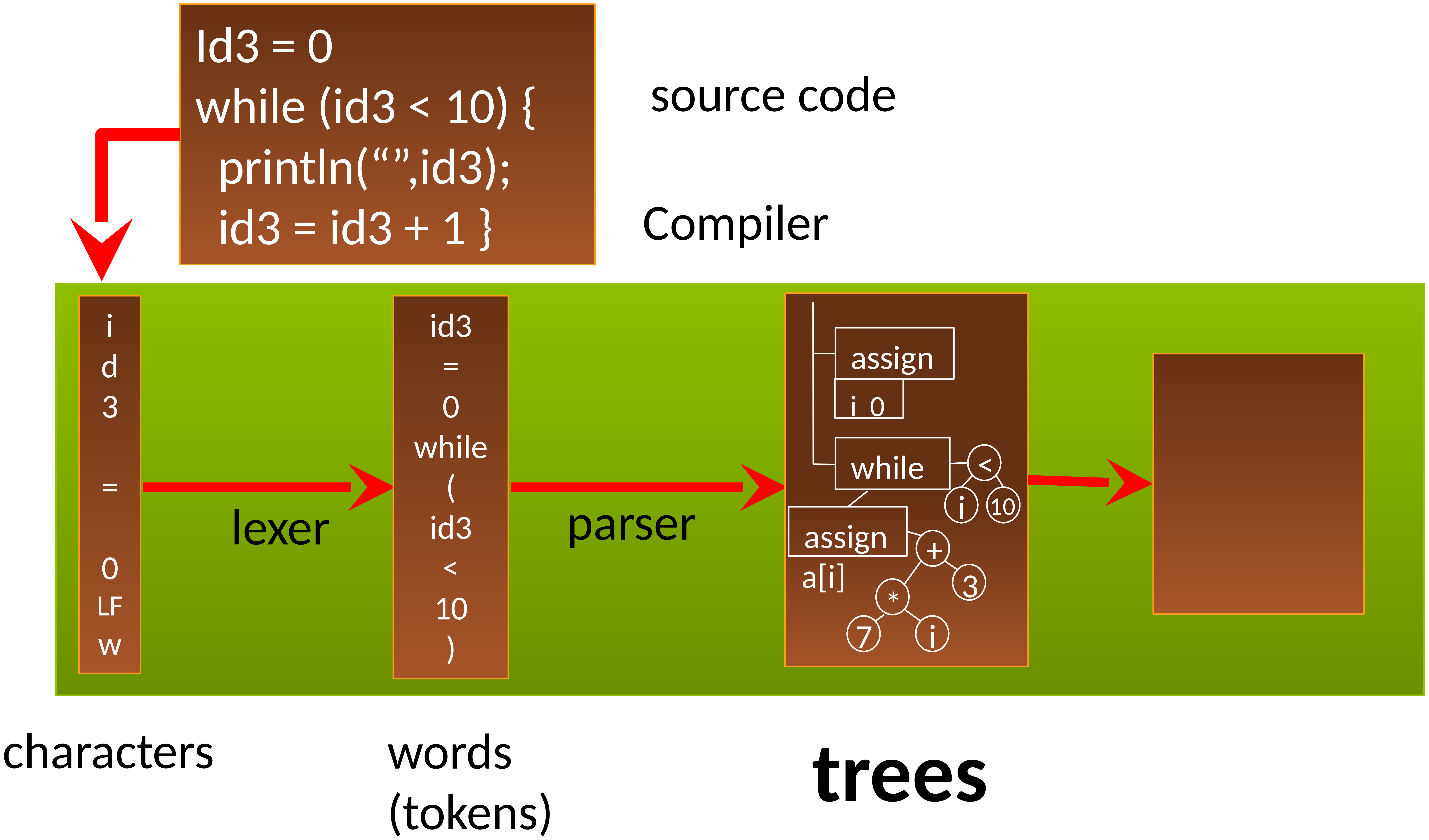
G_3 S ::= ε | S(S)

G_4 S ::= ε | S S | (S)



These all define the same language, the language of balanced parentheses.

Parse Trees and Syntax Trees



Id3 = 0
while (id3 < 10) {
println(“”,id3);
id3 = id3 + 1 }
source code
Compiler
i
d
3
=
0
LF
w
lexer
id3
=
0
while
(
id3
<
10
)
parser
assign
i 0
while
<
i
10
assign
a[i]
+
3
*
7
i
characters
words
(tokens)
trees

While Language Syntax

This syntax is given by a context-free grammar:

program ::= statmt*

statmt ::= println(stringConst , ident)

| ident = expr

| **if** (expr) statmt (else statmt)?

| **while** (expr) statmt

| { statmt* }

expr ::= intLiteral | ident

| expr (&& | < | == | + | - | * | / | %) expr

| ! expr | - expr

N

N → ε | else statmt

statmtSeq ::= ε | statmt | statmtSeq

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, (**but in our labs we will have explicit parse trees**)
- 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 our compiler or interpreter
  - **A compiler often directly builds AST**

Abstract Syntax Trees for Statements

grammar:

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

AST classes:

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
```

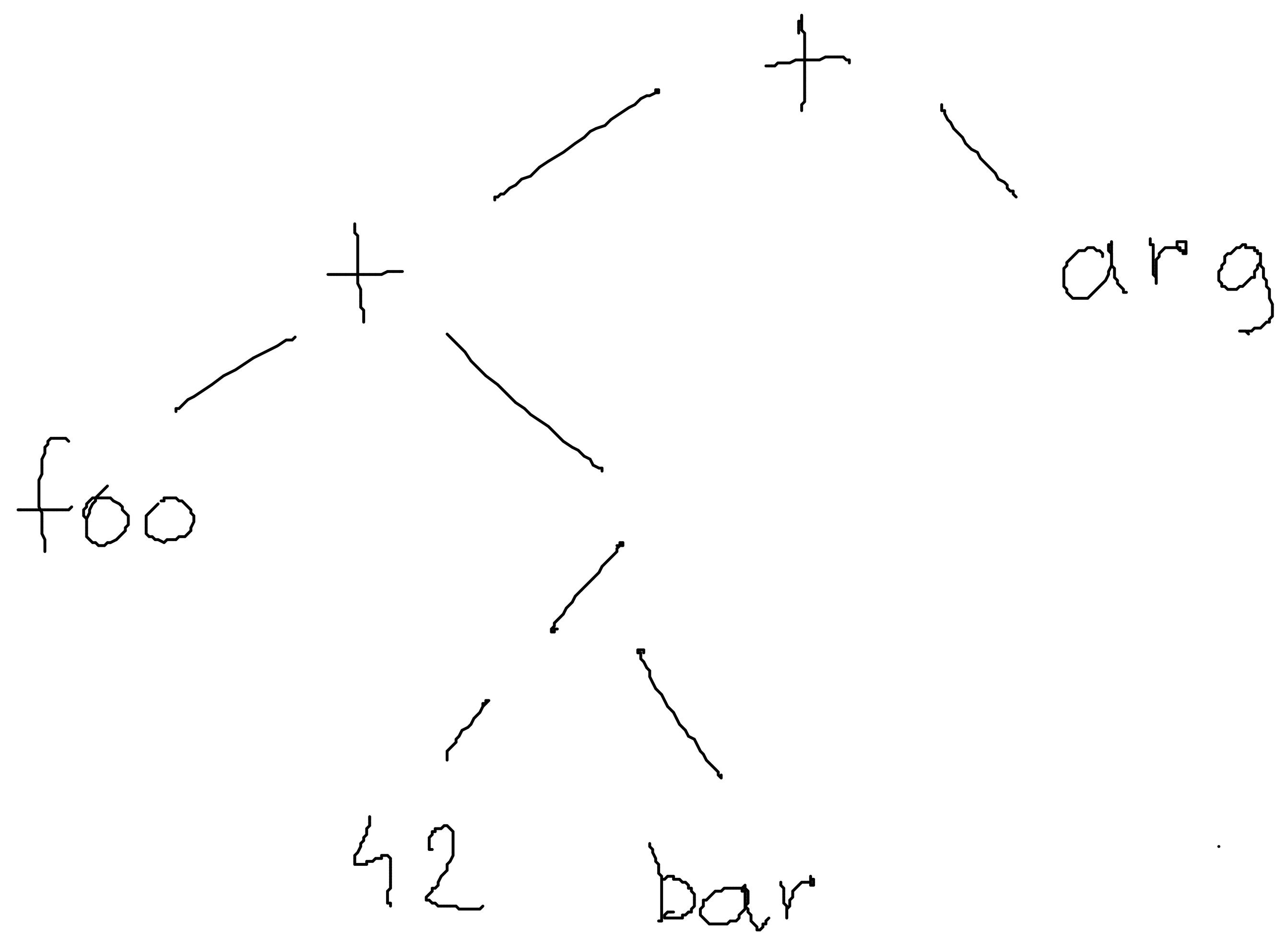
While Language with Simple Expressions

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

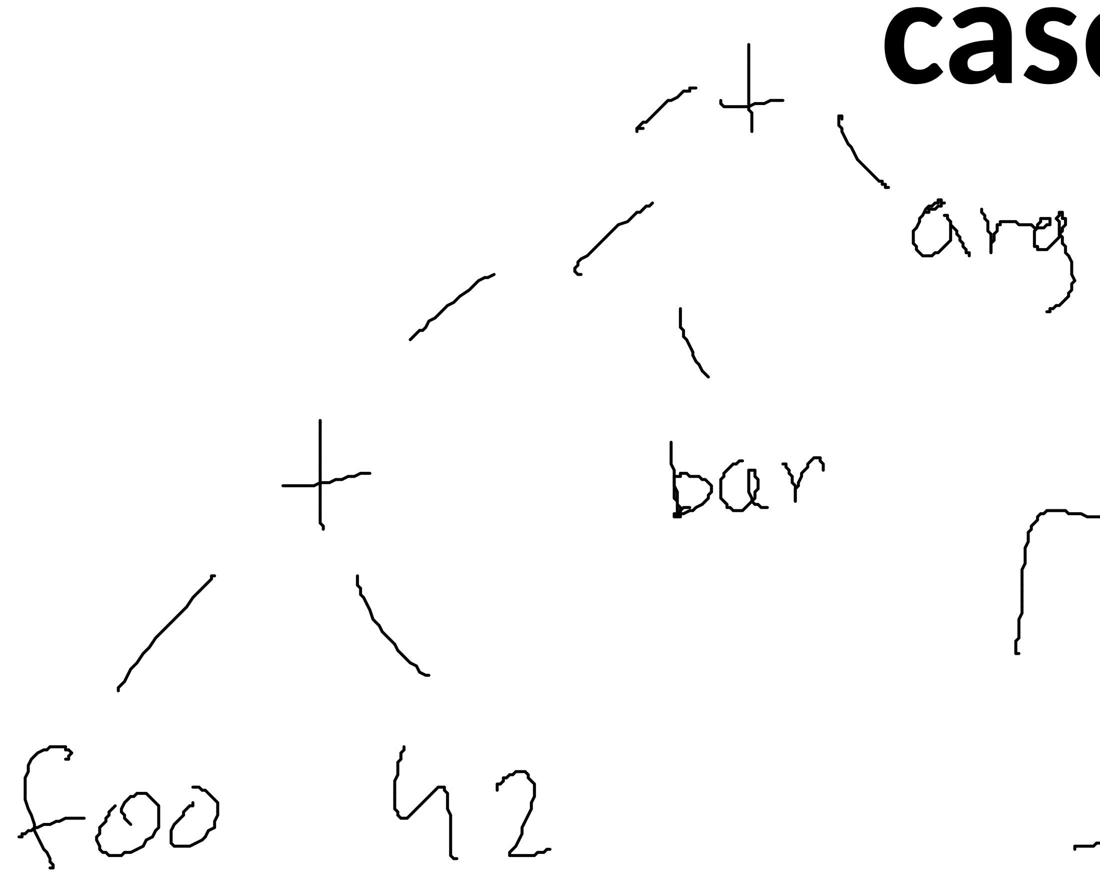
```
expr ::= intLiteral | ident
       | expr ( + | / ) expr
```

Abstract Syntax Trees for Expressions

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



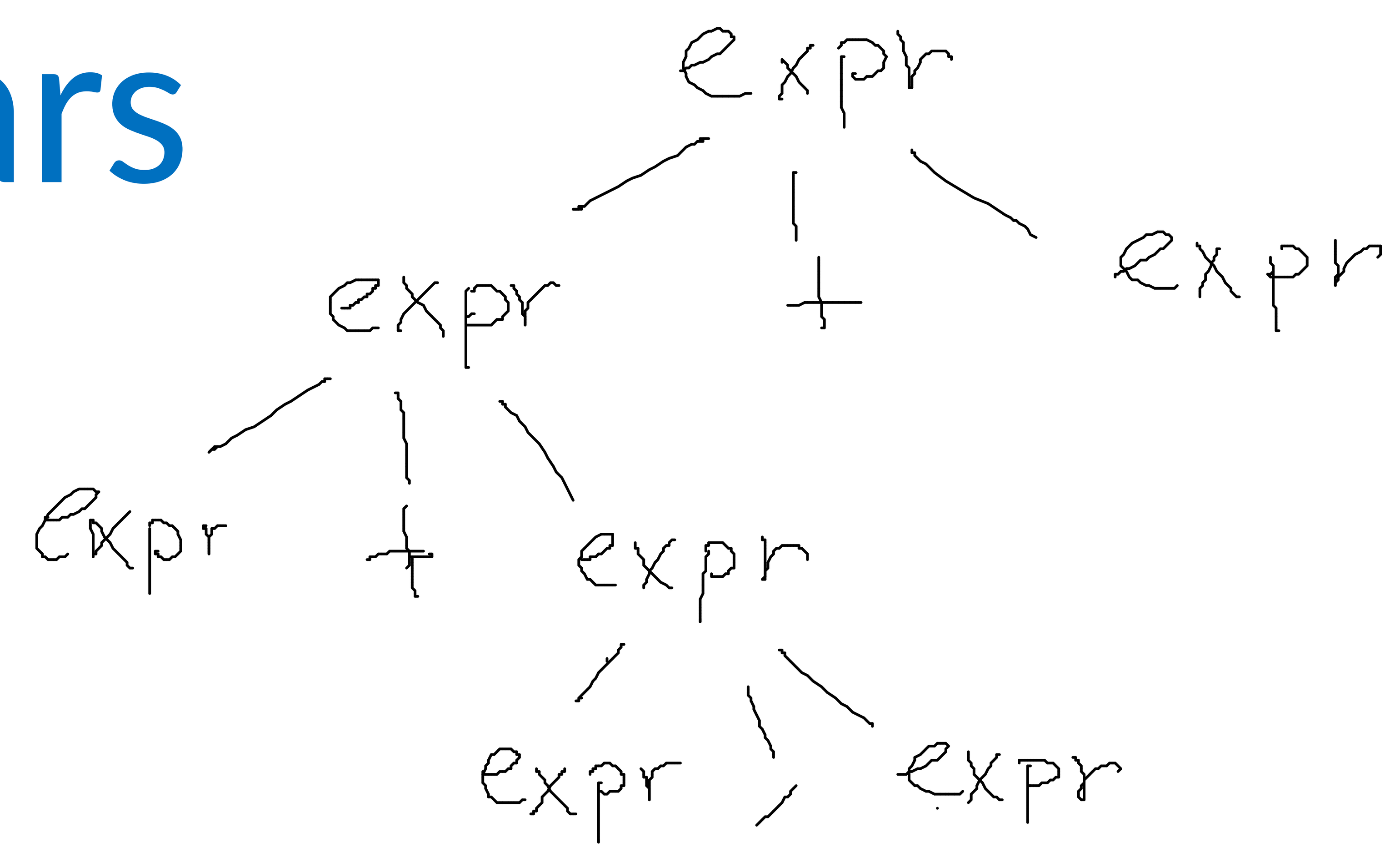

```
abstract class Expr
case class IntLiteral(x : Int) extends Expr
case class Variable(id : Identifier) extends Expr
case class Plus(e1 : Expr, e2 : Expr) extends Expr
case class Divide(e1 : Expr, e2 : Expr) extends Expr
```



foo + 42 / bar + arg

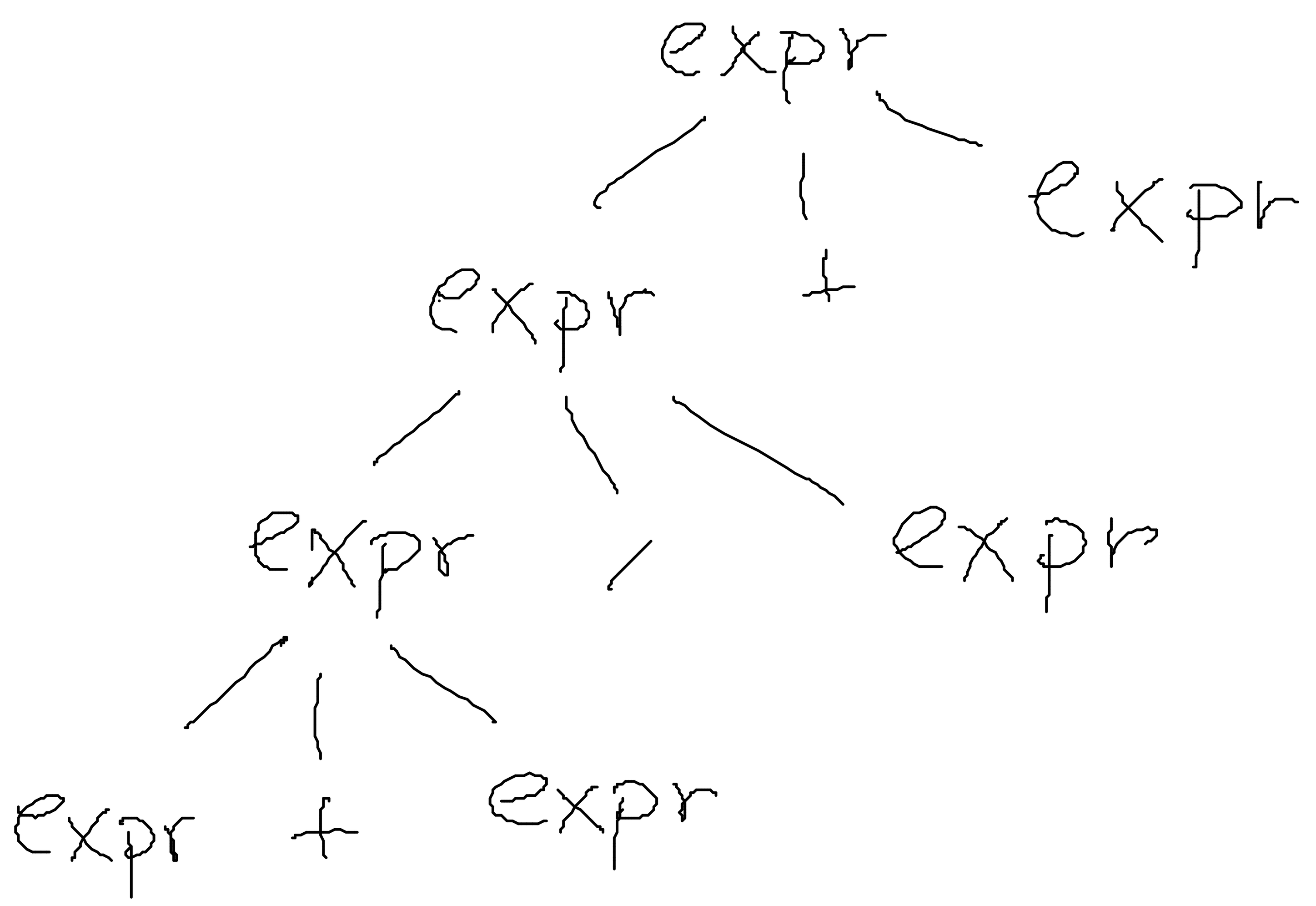
Plus(Plus(Variable("foo"),
Divide(IntLiteral(42), Variable("bar"))
),
Variable("arg"))

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).

Making Grammar Unambiguous and Constructing Correct Trees

Introduction to LL(1) Parsing

Ambiguous Expression Grammar

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

Example input:

ident + intLiteral / ident

has two parse trees, one suggested by

ident + intLiteral / ident

and one by

ident + intLiteral / ident

Suppose Division Binds Stronger

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

Example input:

ident + intLiteral / ident

has two parse trees, one suggested by

ident + intLiteral / ident

and one by **a bad tree**

ident + intLiteral / ident

We do not want arguments of / expanding into expressions with **+** as the top level.

Layering the Grammar by Priorities

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

is transformed into a **new grammar**:

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

The bad tree

ident + intLiteral / ident

cannot be derived in the new grammar.

New grammar: same language, fewer parse trees!

Left Associativity of /

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

Example input:

ident / intLiteral / ident x/9/z

has two parse trees, one suggested by

ident / intLiteral / ident (x/9)/z

and one by **a bad tree**

ident / intLiteral / ident x/(9/z)

We do not want RIGHT argument of / expanding into expression with / as the top level.

Left Associativity - Left Recursion

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

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

No bad / trees
Still bad + trees

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

No bad trees.
Left recursive!

Left vs Right Associativity

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

Left associative
Left recursive,
so not LL(1).

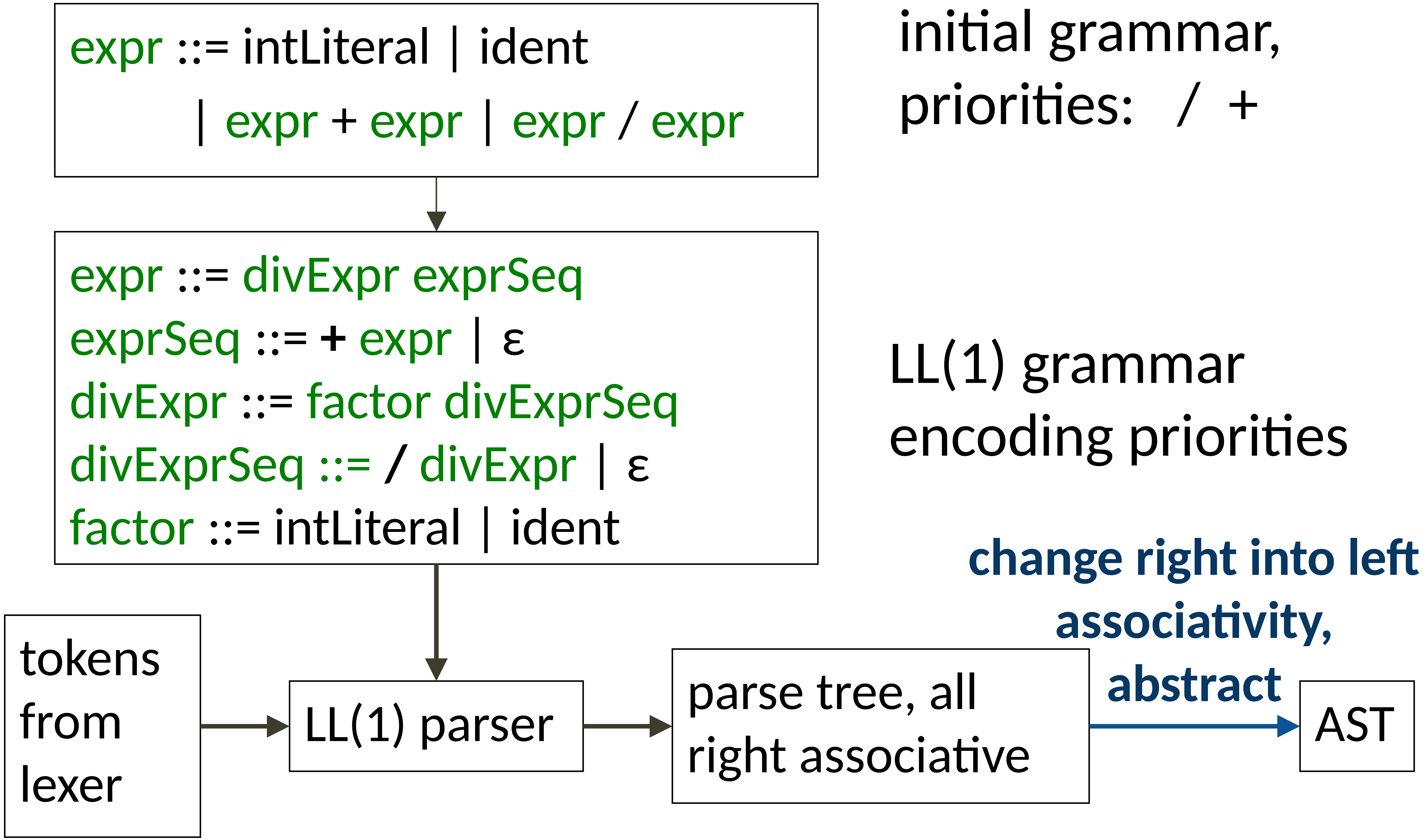
expr ::= divExpr + expr | divExpr
divExpr ::= factor | factor / divExpr
factor ::= intLiteral | ident

Unique trees.
Associativity wrong.
No left recursion.

expr ::= divExpr exprSeq
exprSeq ::= + expr | ε
divExpr ::= factor divExprSeq
divExprSeq ::= / divExpr | ε
factor ::= intLiteral | ident

Unique trees.
Associativity wrong.
LL(1): easy to pick an alternative to use.

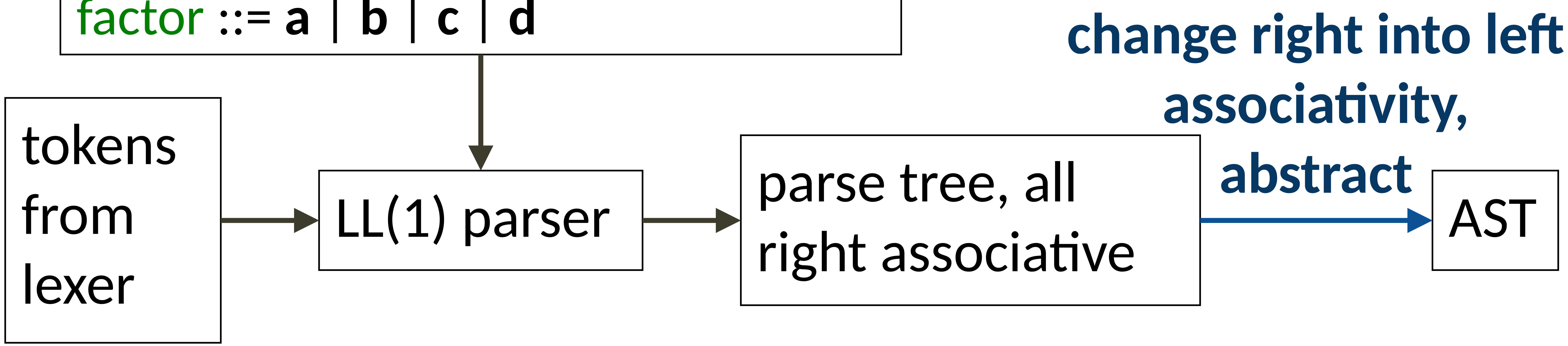
Our Approach



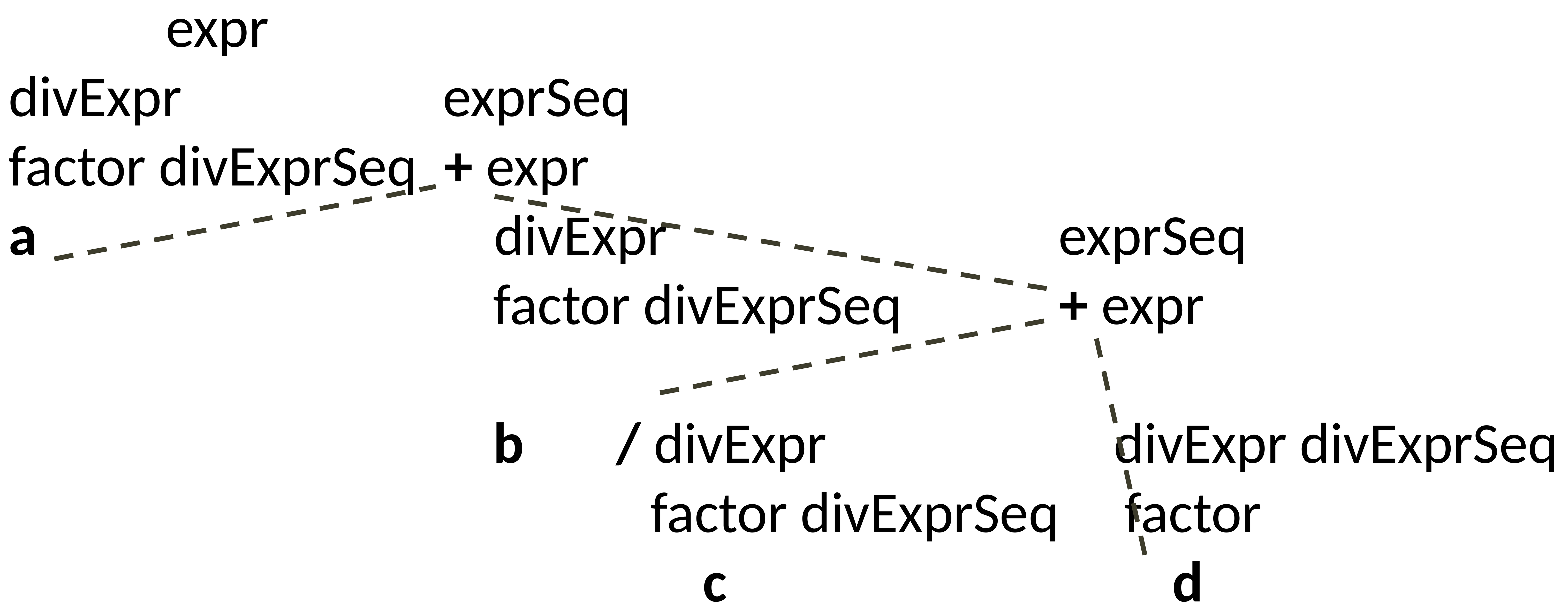
Approach on an Example

expr ::= divExpr exprSeq
exprSeq ::= **+** expr | ε
divExpr ::= factor divExprSeq
divExprSeq ::= **/** divExpr | ε
factor ::= **a** | **b** | **c** | **d**

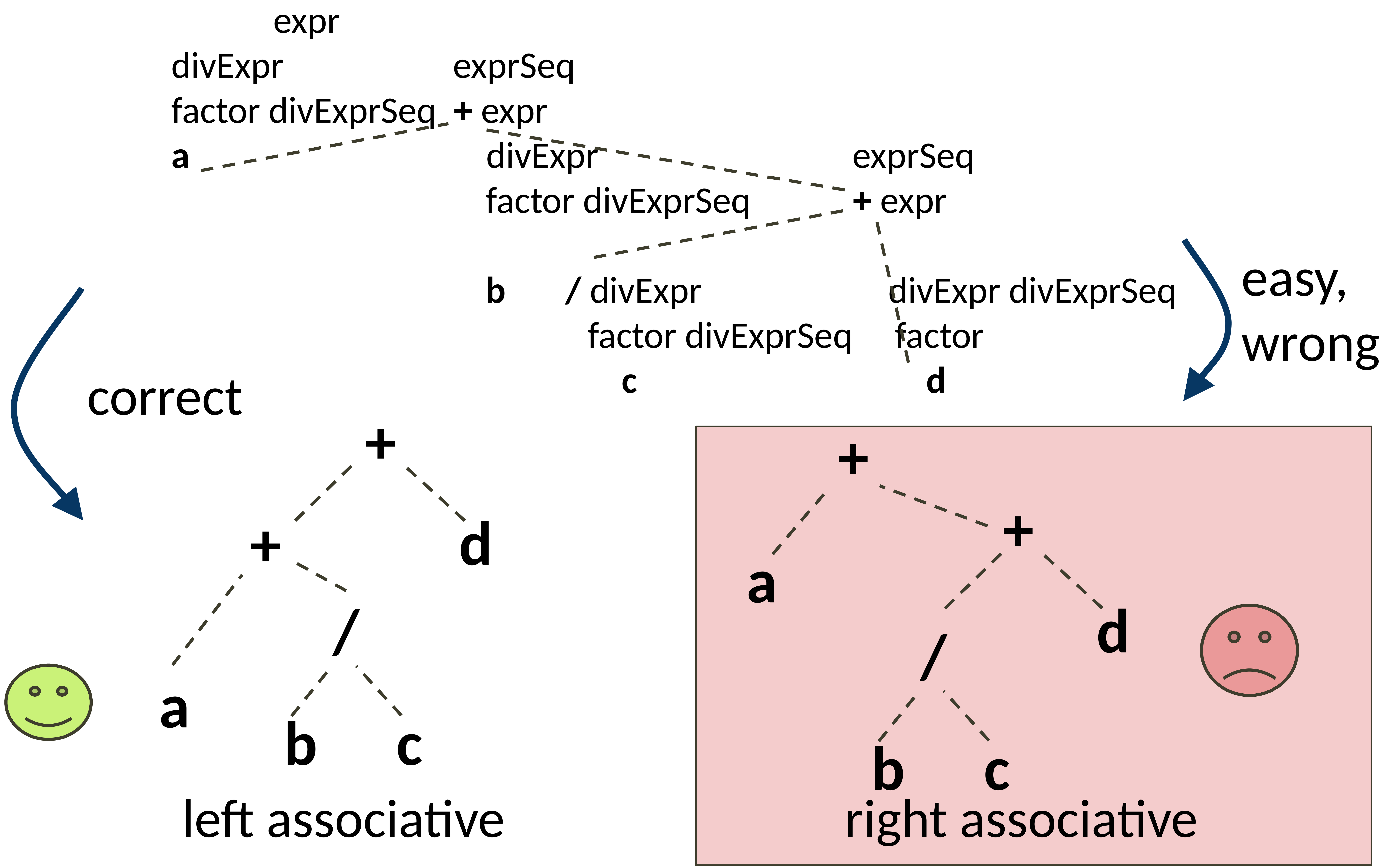
LL(1) grammar
encoding priorities



a + b / c + d



Right Associative Parse Trees into Left Associative Abstract Syntax Tree

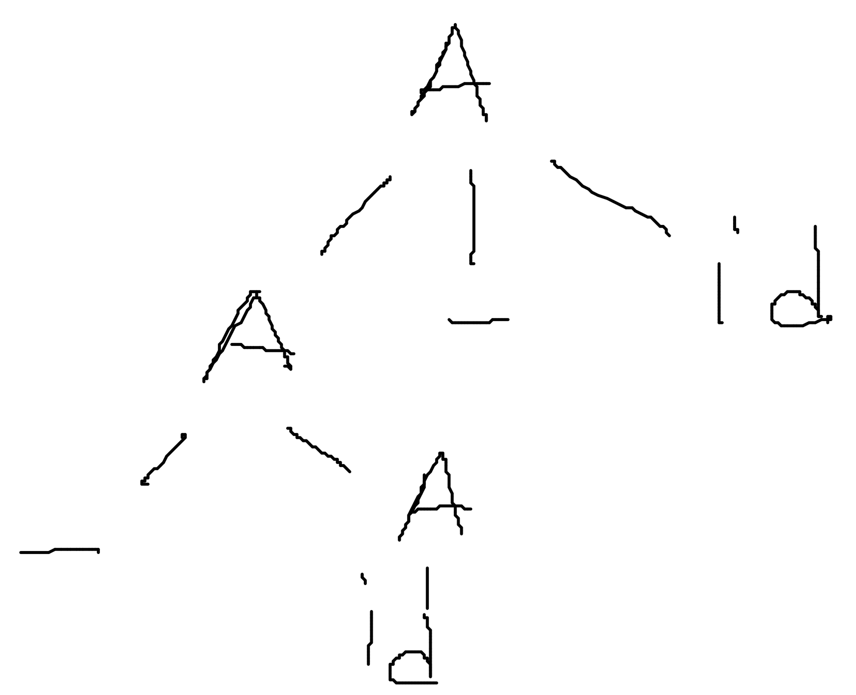


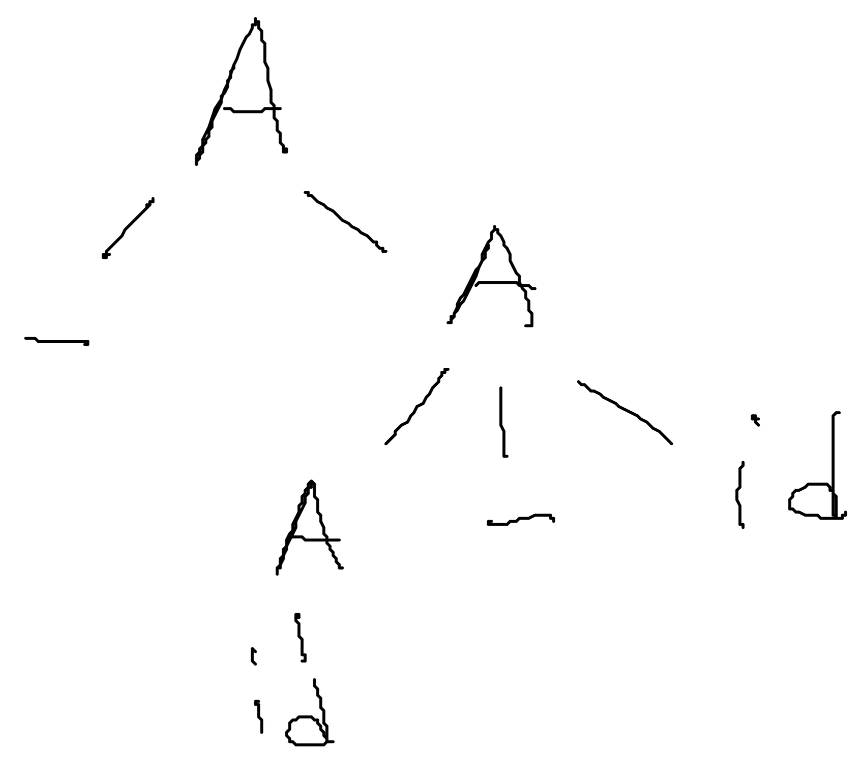
Exercise: Unary Minus

1) Show that the grammar

A ::= – A
A ::= A – id
A ::= id

– id – id





is ambiguous by finding a string that has two different parse trees. Show those parse trees.

2) Make two different unambiguous grammars for the same language:

a) One where prefix minus binds stronger than infix minus.

b) One where infix minus binds stronger than prefix minus.

3) Show the syntax trees using the new grammars for the string you used to prove the original grammar ambiguous.

4) Give a regular expression describing the same language.

Unary Minus Solution Sketch

1) An example of a string with two parse trees is

- id - id

The two parse trees are generated by these imaginary parentheses (shown red): -(id-id) (-id)-id

and can generated by these derivations that give different parse trees

A => -A => - A - id => - id - id

A => A - id => - A - id => - id - id

2) a) prefix minus binds stronger:

A ::= B | A - id B ::= -B | id

b) infix minus binds stronger

A ::= C | -A C ::= id | C - id

3) in two trees that used to be ambiguous instead of some A's we have B's in a) grammar or C's in b) grammar.

4) -*id(-id)*