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?
- 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 No

  ((()) ()) - balanced

  ( )) (( ) - not balanced
- 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

$q_0, q_1, q_2, \dots, q_K$

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

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

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: $\{ a^nb^n \mid n >= 0 \}$

Finite State Automata are Limited

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

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

S ::= ε - first rule of this grammar

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

Example of a derivation (DEMO)

S => aSb => a aSb b => aa aSb bb => 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 ∈ $(A \cup N)^*$ - sequence of terminals and non-terminals

A derivation in G starts from the starting symbol S

- 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

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 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

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

$w \in L(G) \quad \Leftrightarrow \quad \exists t.\ w=\text{yield}(t) \wedge \text{isParseTree}(G,t)$

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 | 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 tree for aaaabb

Show at least two derivations that correspond to that tree.

Balanced Parentheses Grammar

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

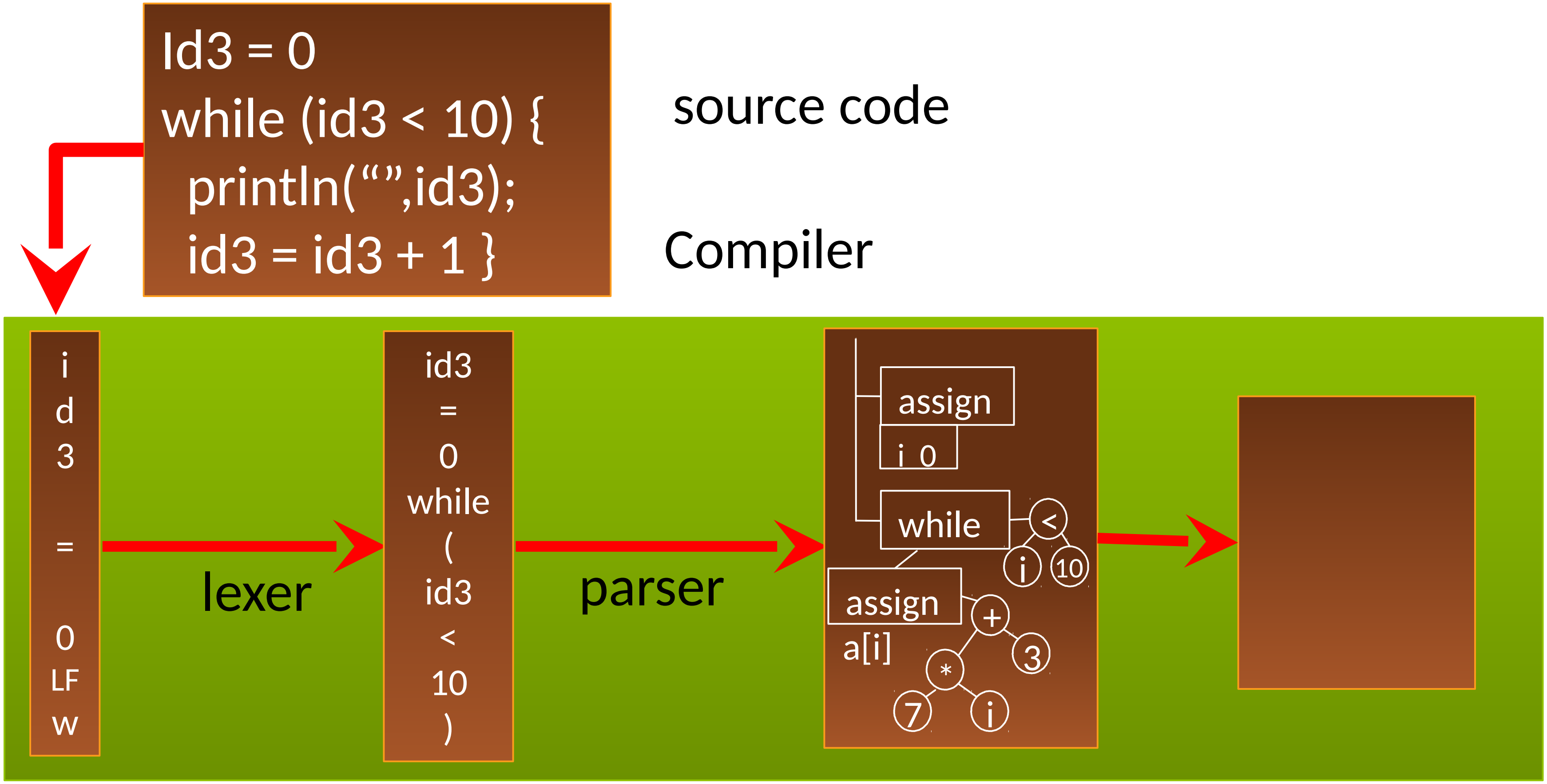
Exercise: give the grammar and example derivation for the first string.

Balanced Parentheses Grammar

G_1	$S ::= \varepsilon \mid S(S)S$
G_2	$S ::= \varepsilon \mid (S)S$
G_3	$S ::= \varepsilon \mid S(S)$
G_4	$S ::= \varepsilon \mid S\ S \mid (S)$

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

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

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

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).

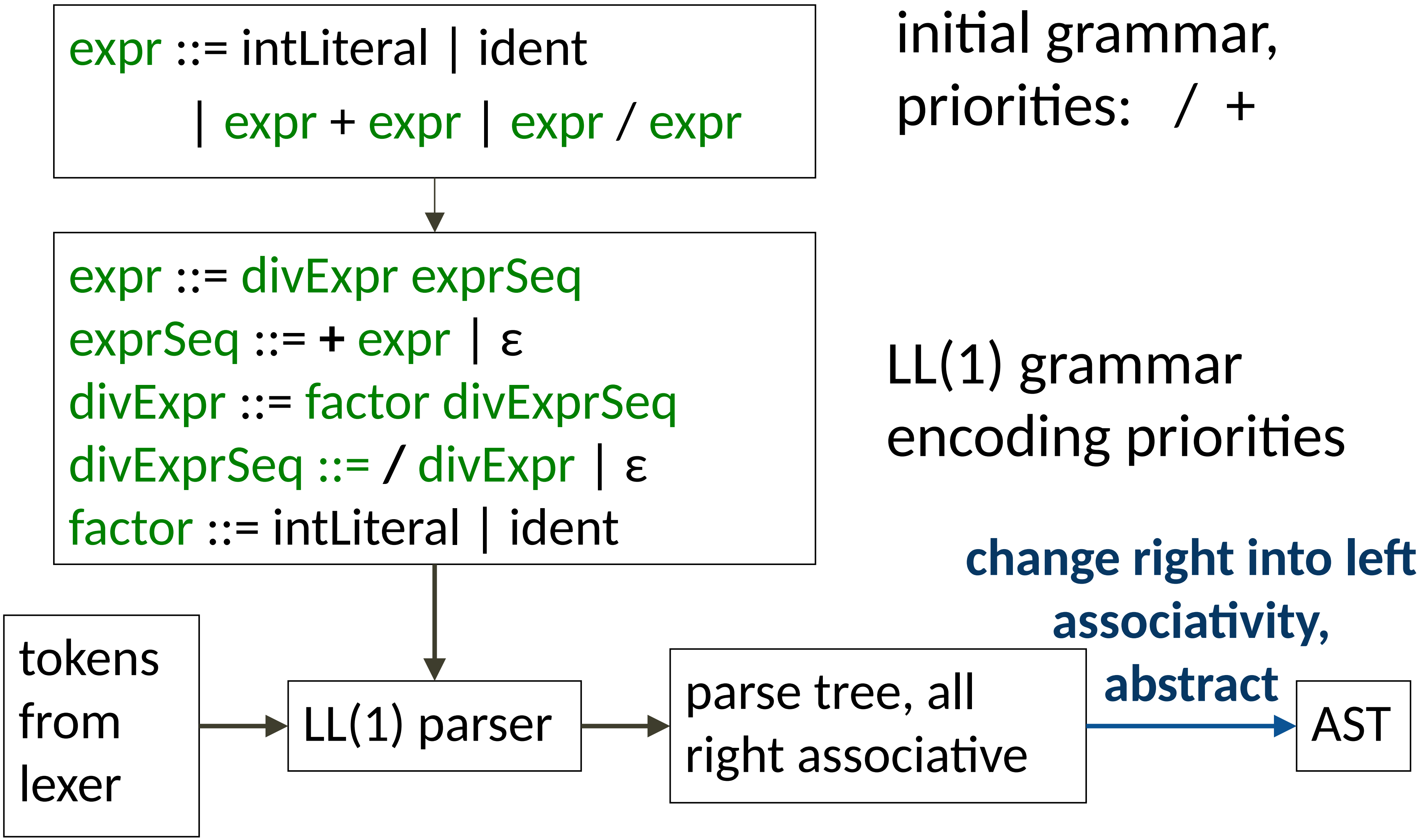
```
expr ::= divExpr + expr | divExpr
divExpr ::= factor | factor / divExpr
factor ::= intLiteral | ident
```

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



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

initial grammar,
priorities: / +

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

LL(1) grammar
encoding priorities

tokens from lexer

LL(1) parser

parse tree, all right associative

change right into left associativity, abstract

AST

Approach on an Example

expr ::= divExpr exprSeq

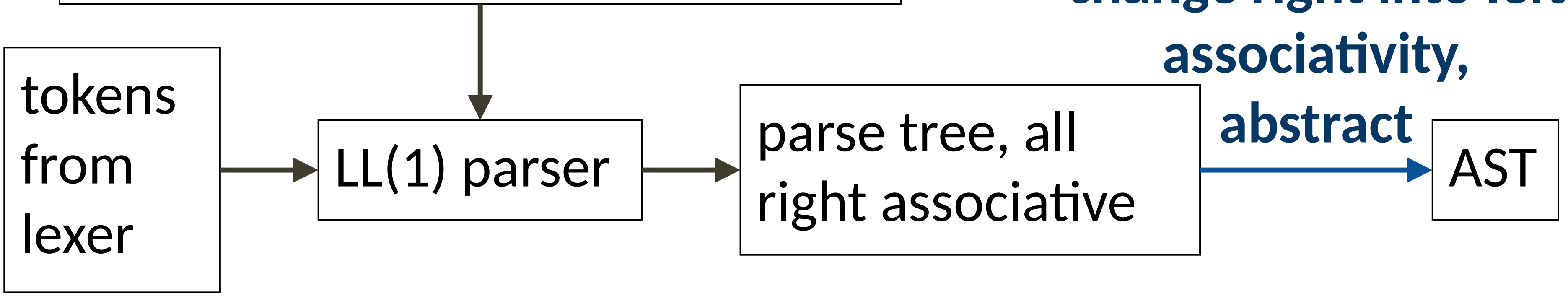
exprSeq ::= **+** expr | ε

divExpr ::= factor divExprSeq

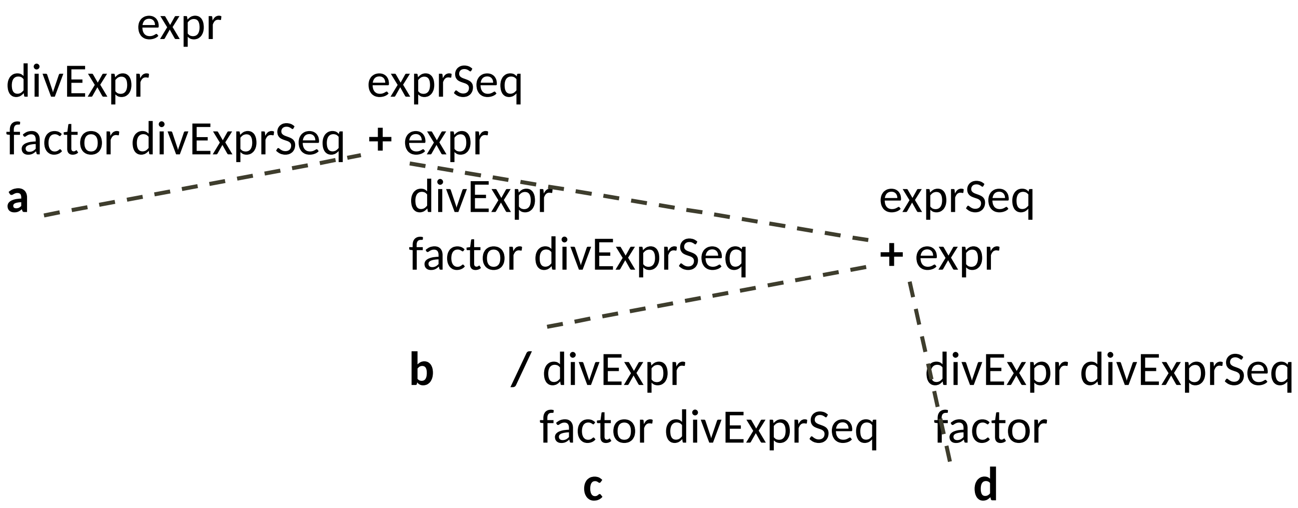
divExprSeq ::= **/** divExpr | ε

factor ::= **a** | **b** | **c** | **d**

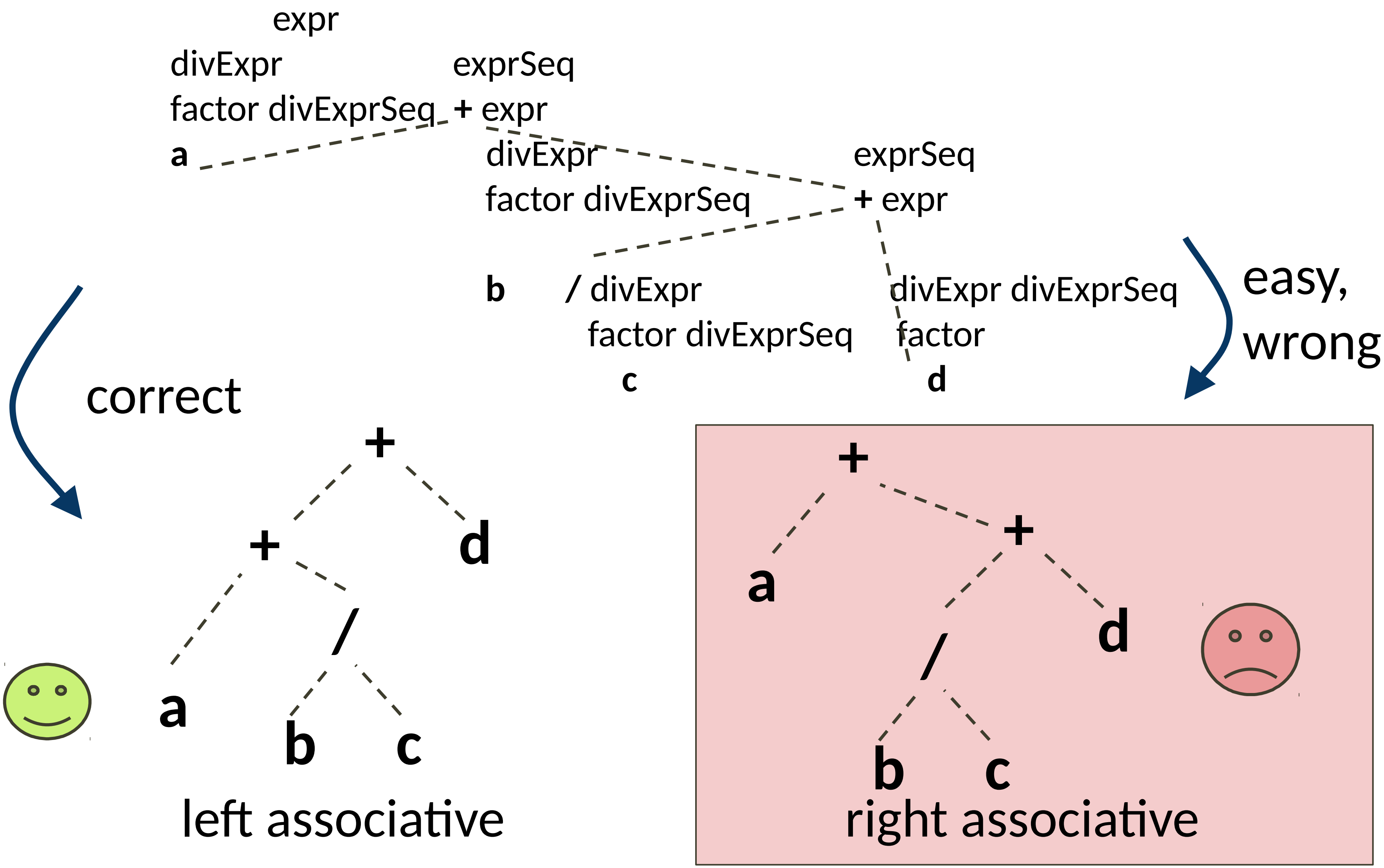
LL(1) grammar
encoding priorities


tokens from lexer
LL(1) parser
parse tree, all right associative
change right into left associativity, abstract
AST

a + b / c + d



Right Associative Parse Trees into Left Associative Abstract Syntax Tree



expr

divExpr exprSeq

factor divExprSeq + expr

a divExpr exprSeq

factor divExprSeq + expr

b / divExpr divExpr divExprSeq

factor divExprSeq factor

c d

correct

easy, wrong

left associative

right associative

Exercise: Unary Minus

1) Show that the grammar

A ::= − A
A ::= A − id
A ::= 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)*

Recursive Descent
LL(1) Parsing

- useful parsing technique
- to make it work, we might need to transform the grammar

Recursive Descent is Decent

Recursive *descent* is a *decent* parsing technique

- can be easily implemented manually based on the grammar (which may require transformation)
- efficient (linear) in the size of the token sequence

Correspondence between grammar and code

- concatenation → ;
- alternative (|) → if
- repetition (*) → while
- nonterminal → recursive procedure

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)
def statmt = {
  if (lexer.token == Println) { lexer.next;
    skip(openParen); skip(stringConst); skip(comma);
    skip(identifier); skip(closedParen)
  } else if (lexer.token == Ident) { lexer.next;
    skip(equality); expr
  } 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 to construct **if** conditions?

statmt ::= println (stringConst , ident)
| if (expr) statmt (else statmt)?
| while (expr) 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 expressions
- 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** }   - we will give an algorithm

Formalizing and Automating Recursive Descent: LL(1) Parsers

Task: Rewrite Grammar to make it suitable for recursive descent parser

- Assume the priorities of operators as in Java

```
expr ::= expr (+|-|*|/) expr
       | name | `(’ expr `)’
name ::= ident
```

Grammar vs Recursive Descent Parser

```
expr ::= term termList
termList ::= + term termList
          |  - term termList
          | ε
term ::= factor factorList
factorList ::= * factor factorList
             | / factor factorList
             | ε
factor ::= name | ( expr )
name ::= ident
```

```
def expr = { term; termList }
def termList =
  if (token==PLUS) {
    skip(PLUS); term; termList
  } else if (token==MINUS)
    skip(MINUS); term; termList
  }
def term = { factor; factorList }
...
def factor =
  if (token==IDENT) name
  else if (token==OPAR) {
    skip(OPAR); expr; skip(CPAR)
  } else error("expected ident or )")
```

Note that the abstract trees we would create in this example do not strictly follow parse trees.

Rough General Idea

$A ::= B_1 \ldots B_p$
$\quad\mid C_1 \ldots C_q$
$\quad\mid D_1 \ldots D_r$

```
def A =
  if (token $\in$ T1) {
    $B_1 \ldots B_p$
  else if (token $\in$ T2) {
    $C_1 \ldots C_q$
  } else if (token $\in$ T3) {
    $D_1 \ldots D_r$
  } else error("expected T1,T2,T3")
```

where:

T1 = **first**($B_1 \ldots B_p$)

T2 = **first**($C_1 \ldots C_q$)

T3 = **first**($D_1 \ldots D_r$)

first($B_1 \ldots B_p$) = $\{a \in \Sigma \mid B_1 \ldots B_p \Rightarrow \ldots \Rightarrow aw\}$

T1, T2, T3 should be **disjoint** sets of tokens.

Computing **first** in the example

expr ::= term termList

termList ::= **+** term termList

| **-** term termList

| ε

term ::= factor factorList

factorList ::= * factor factorList

| / factor factorList

| ε

factor ::= name | **(** expr **)**

name ::= **ident**

first(name) = {**ident**}

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

first(factor) = first(name)

U first(**(** expr **)**)

= {**ident**} U{ **(** }

= {**ident**, **(** }

first(* factor factorList) = { * }

first(/ factor factorList) = { / }

first(factorList) = { *, / }

first(term) = first(factor) = {**ident**, **(** }

first(termList) = { **+** , **-** }

first(expr) = first(term) = {**ident**, **(** }

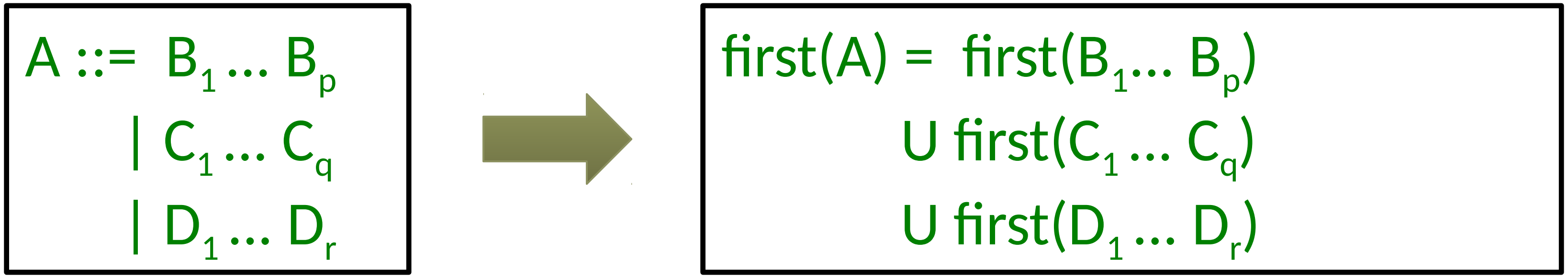
Algorithm for **first**: Goal

Given an arbitrary context-free grammar with a set of rules of the form X ::= Y_1 … Y_n compute first for each right-hand side and for each symbol.

How to handle

- alternatives for one non-terminal
- sequences of symbols
- nullable non-terminals
- recursion

Rules with Multiple Alternatives



$$\begin{aligned} A ::= {} & B_1 \ldots B_p \\ \mid {} & C_1 \ldots C_q \\ \mid {} & D_1 \ldots D_r \end{aligned}$$

$$\begin{aligned} \text{first}(A) = {} & \text{first}(B_1 \ldots B_p) \\ & \cup \text{ first}(C_1 \ldots C_q) \\ & \cup \text{ first}(D_1 \ldots D_r) \end{aligned}$$

Sequences

$\boxed{\text{first}(B_1 \ldots B_p) = \text{first}(B_1)}$ if not nullable(B_1)

$\boxed{\text{first}(B_1 \ldots B_p) = \text{first}(B_1) \cup \ldots \cup \text{first}(B_k)}$

if nullable(B_1), ..., nullable(B_{k-1}) and

not nullable(B_k) or k=p

Abstracting into Constraints

recursive grammar: constraints over finite sets: expr' is first(expr)

```
expr ::= term termList
termList ::= + term termList
         |  - term termList
         | ε
term ::= factor factorList
factorList ::= * factor factorList
                 | / factor factorList
                 | ε
factor ::= name | ( expr )
name ::= ident
```

```
expr' = term'
termList' =  {+}
         U {-}

term' = factor'
factorList' = {*}
               U { / }

factor' = name' U { ( }
name' = { ident }
```

nullable: termList, factorList

For this nice grammar, there is no recursion in constraints. Solve by substitution.

Example to Generate Constraints

S ::= X | Y
X ::= **b** | S Y
Y ::= Z X **b** | Y **b**
Z ::= ε | **a**

S' = X' U Y'
X' =

terminals: **a**,**b**
non-terminals: S, X, Y, Z

reachable (from S):
productive:
nullable:

First sets of terminals:
S', X', Y', Z' ⊆ {a,b}

Example to Generate Constraints

S ::= X | Y
X ::= **b** | S Y
Y ::= Z X **b** | Y **b**
Z ::= ε | **a**

S' = X' U Y'
X' = {b} U S'
Y' = Z' U **X'** U Y'
Z' = {a}

terminals: **a**,**b**
non-terminals: S, X, Y, Z

These constraints are recursive.
How to solve them?

$S', X', Y', Z' \subseteq \{a,b\}$

How many candidate solutions

- in this case?
- for k tokens, n nonterminals?

reachable (from S): S, X, Y, Z
productive: X, Z, S, Y
nullable: Z

Iterative Solution of **first** Constraints

	S'	X'	Y'	Z'
1.	{}	{}	{}	{}
2.	{}	{b}	{b}	{a}
3.	{b}	{b}	{a,b}	{a}
4.	{a,b}	{a,b}	{a,b}	{a}
5.	{a,b}	{a,b}	{a,b}	{a}

S' = X' U Y'

X' = {b} U S'

Y' = Z' U **X'** U Y'

Z' = {a}

- Start from all sets empty.
- Evaluate right-hand side and assign it to left-hand side.
- Repeat until it stabilizes.

Sets grow in each step

- initially they are empty, so they can only grow
- if sets grow, the RHS grows (U is monotonic), and so does LHS
- they cannot grow forever: in the worst case contain all tokens

Constraints for Computing Nullable

- Non-terminal is nullable if it can derive ε

S ::= X | Y
X ::= **b** | S Y
Y ::= Z X **b** | Y **b**
Z ::= ε | **a**

S' = X' | Y'
X' = 0 | (S' & Y')
Y' = (Z' & X' & 0) | (Y' & 0)
Z' = 1 | 0

S', X', Y', Z' ∈ {0,1}
0 - not nullable
1 - nullable
| - disjunction
& - conjunction

	S'	X'	Y'	Z'
1.	0	0	0	0
2.	0	0	0	1
3.	0	0	0	1

again monotonically growing

Computing first and nullable

- Given any grammar we can compute
  - for each non-terminal X whether nullable(X)
  - using this, the set first(X) for each non-terminal X
- General approach:
  - generate constraints over finite domains, following the structure of each rule
  - solve the constraints iteratively
    - start from least elements
    - keep evaluating RHS and re-assigning the value to LHS
    - stop when there is no more change

Summary: Algorithm for **nullable**

```
nullable = {}
changed = true
while (changed) {
  changed = false
  for each non-terminal X
    if ((X is not nullable) and
          (grammar contains rule     X ::= ε | ...     )
              or   (grammar contains rule     X ::= Y1 ... Yn | ...
           where {Y1,...,Yn} ⊆ nullable)
    then {
        nullable = nullable U {X}
        changed = true
     }
}
```

Summary: Algorithm for **first**

```
for each nonterminal X:  first(X)={}
for each terminal t:  first(t)={t}
repeat
  for each grammar rule X ::= Y(1) … Y(k)
  for i = 1 to k
      if i=1 or {Y(1),…,Y(i-1)} ⊆ nullable then
        first(X) = first(X) U first(Y(i))
until none of first(…) changed in last iteration
```

Follow sets. LL(1) Parsing Table

Exercise Introducing Follow Sets

Compute nullable, first for this grammar:

stmtList ::= ε | stmt stmtList

stmt ::= assign | block

assign ::= **ID = ID ;**

block ::= **beginof ID** stmtList **ID ends**

Describe a parser for this grammar and explain how it behaves on this input:

beginof myPrettyCode

x = u;

y = v;

myPrettyCode **ends**

How does a recursive descent parser look like?

```
def stmtList =
  if (???) {}                what should the condition be?
  else { stmt; stmtList }
def stmt =
  if (lex.token == ID) assign
  else if (lex.token == beginof) block
  else error(“Syntax error: expected ID or beginonf”)
…
def block =
  { skip(beginof); skip(ID); stmtList; skip(ID); skip(ends) }
```

Problem Identified

stmtList ::= ε | stmt stmtList

stmt ::= assign | block

assign ::= **ID = ID ;**

block ::= **beginof ID** stmtList **ID ends**

Problem parsing stmtList:

- **ID** could start alternative stmt stmtList
- **ID** could **follow** stmt, so we may wish to parse **ε** that is, do nothing and return

- For nullable non-terminals, we must also compute what **follows** them

LL(1) Grammar - good for building recursive descent parsers

- Grammar is LL(1) if for each nonterminal X
  - first sets of different alternatives of X are disjoint
  - if nullable(X), first(X) must be disjoint from follow(X) and only one alternative of X may be nullable
- For each LL(1) grammar we can build recursive-descent parser
- Each LL(1) grammar is unambiguous
- If a grammar is not LL(1), we can sometimes transform it into equivalent LL(1) grammar

Computing if a token can **follow**

first$(B_1 \ldots B_p) = \{a \in \Sigma \mid B_1 \ldots B_p \Rightarrow \ldots \Rightarrow aw\}$

follow$(X) = \{a \in \Sigma \mid S \Rightarrow \ldots \Rightarrow \ldots Xa \ldots\}$

There exists a derivation from the start symbol that produces a sequence of terminals and nonterminals of the form ...Xa...
(the token a follows the non-terminal X)

Rule for Computing Follow

Given $X ::= YZ$ (for reachable X)

then **first**(Z) ⊆ **follow**(Y)

and **follow**(X) ⊆ **follow**(Z)

now take care of nullable ones as well:

For each rule $X ::= Y_1 \ldots Y_p \ldots Y_q \ldots Y_r$

follow(Y_p) should contain:

- **first**($Y_{p+1}Y_{p+2}\ldots Y_r$)
- also **follow**(X) if **nullable**($Y_{p+1}Y_{p+2}Y_r$)

Compute nullable, first, follow

stmtList ::= ε | stmt stmtList

stmt ::= assign | block

assign ::= **ID = ID ;**

block ::= **beginof ID** stmtList **ID ends**

Is this grammar LL(1)?

Conclusion of the Solution

The grammar is not LL(1) because we have

- nullable(stmtList)
- first(stmt) ∩ follow(stmtList) = {**ID**}

- If a recursive-descent parser sees **ID**, it does not know if it should
  - finish parsing stmtList or
  - parse another stmt

Table for LL(1) Parser: Example

S ::= B **EOF**
(1)

B ::= ε | B **(**B**)**
(1) **(2)**

nullable: B

first(S) = { **(, EOF** }

follow(S) = {}

first(B) = { **(** }

follow(B) = { **)**, **(**, **EOF** }

empty entry:
when parsing S,
if we see) ,
report error

Parsing table:

	EOF	(	)
S	{1}	{1}	{}
B	{1}	{1,2}	{1}

parse conflict - choice ambiguity:
grammar not LL(1)

1 is in entry because **(** is in follow(B)
2 is in entry because **(** is in first(B(B))

Table for LL(1) Parsing

Tells which alternative to take, given current token:

choice : Nonterminal x Token -> Set[Int]

$A ::= \textbf{(1)}\ B_1 \ldots B_p$
$\quad | \ \textbf{(2)}\ C_1 \ldots C_q$
$\quad | \ \textbf{(3)}\ D_1 \ldots D_r$

if $t \in \mathrm{first}(C_1 \ldots C_q)$ add 2 to choice(A,t)

if $t \in \mathrm{follow}(A)$ add K to choice(A,t) where K is nullable

For example, when parsing A and seeing token t

choice(A,t) = {2} means: parse alternative 2 $(C_1 \ldots C_q)$

choice(A,t) = {3} means: parse alternative 3 $(D_1 \ldots D_r)$

choice(A,t) = {} means: report syntax error

choice(A,t) = {2,3} : not LL(1) grammar

General Idea when parsing nullable(A)

$A ::= B_1 \dots B_p$
$\quad\;\; | \; C_1 \dots C_q$
$\quad\;\; | \; D_1 \dots D_r$

```
def A =
  if (token $\in$ T1) {
    $B_1 \dots B_p$
  else if (token $\in$ (T2 $\cup$ $T_F$)) {
    $C_1 \dots C_q$
  } else if (token $\in$ T3) {
    $D_1 \dots D_r$
  } // no else error, just return
```

where:

$T1 =$ **first**$(B_1 \dots B_p)$

$T2 =$ **first**$(C_1 \dots C_q)$

$T3 =$ **first**$(D_1 \dots D_r)$

$T_F =$ **follow**(A)

Only one of the alternatives can be nullable (here: 2nd)

T1, T2, T3, T_F should be pairwise **disjoint** sets of tokens.