#### **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.

$$B ::= ε | (B) | B B$$

#### Remark

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

$$B \rightarrow (B) \rightarrow (BB) \rightarrow ((B)B) \rightarrow ((B)) \rightarrow (())$$

$$B \rightarrow (B) \rightarrow (BB) \rightarrow ((B)B) \rightarrow (()B) \rightarrow (())$$

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.

#### **Towards Solution**

- (Note that we must preserve precisely the set of strings that can be derived)
- This grammar:

B ::= 
$$\epsilon \mid A$$
  
A ::= ( ) | A A | (A)

solves the problem with multiple  $\epsilon$  symbols generating different trees, but it is still ambiguous: string ( ) ( ) ( ) has two different parse trees

#### Solution

Proposed solution:

$$B := \varepsilon \mid B(B)$$

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

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

$$B ::= \varepsilon \mid C B$$
$$C ::= (B)$$

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

$$B := \varepsilon \mid B(B)$$

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

# Exercise 2: Dangling Else

The dangling-else problem happens when the conditional statements are parsed using the following grammar.

S ::= S ; S

S ::= id **:=** E

S ::= **if** E **then** S

S ::= if E then S else S

Find an unambiguous grammar that accepts the same conditional statements and matches the else statement with the nearest unmatched if.

# Discussion of Dangling Else

```
if (x > 0) then

if (y > 0) then

z = x + y

else x = -x
```

- This is a real problem languages like C, Java
  - resolved by saying else binds to innermost if
- Can we design grammar that allows all programs as before, but only allows parse trees where else binds to innermost if?

# Sources of Ambiguity in this Example

- Ambiguity arises in this grammar here due to:
  - dangling else
  - binary rule for sequence (;) as for parentheses
  - priority between if-then-else and semicolon (;)

```
if (x > 0)

if (y > 0)

z = x + y;

u = z + 1 // last assignment is not inside if
```

Wrong parse tree -> wrong generated code

#### How we Solved It

We identified a wrong tree and tried to refine the grammar to prevent it, by making a copy of the rules. Also, we changed some rules to disallow sequences inside if-then-else and make sequence rule non-ambiguous. The end result is something like this:

At some point we had a useless rule, so we deleted it.

We also looked at what a practical grammar would have to allow sequences inside if-then-else. It would add a case for blocks, like this:

We could factor out some common definitions (e.g. define A in terms of A'), but that is not important for this problem.

# **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 syntax 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.

#### **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

$$A := q A$$

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

Answer: yes, if A is reachable from the top symbol and productive can produce a sequence of tokens

# Making Grammars Unambiguous - some 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

#### different order gives different results:

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

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

# **Ambiguous Expression Grammar**

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

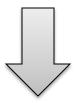
foo + 42 / bar + arg

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

Show that the input above has two parse trees!

# 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) Building trees: left-associative "-"

#### **LEFT-associative** operator

```
x-y-z \rightarrow (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)
```

# 3) Building trees: right-associative "^"

```
RIGHT-associative operator – using recursion
                         (or also loop and then reverse a list)
x \wedge y \wedge z \rightarrow x \wedge (y \wedge 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

## Grammar Rules as Logic Programs

Consider grammar G: S::= a | b S

L(\_) - language of non-terminal

L(G) = L(S) where S is the start non-terminal

$$L(S) = L(G) = \{ b^n a \mid n >= 0 \}$$

From meaning of grammars:

$$w \in L(S) \Leftrightarrow w=a \bigvee w \in L(b S)$$

To check left hand side, we need to check right hand side. Which of the two sides?

- restrict grammar, use current symbol to decide LL(1)
- use dynamic programming (CYK) for any grammar

# Recursive Descent - LL(1)

#### See wiki for

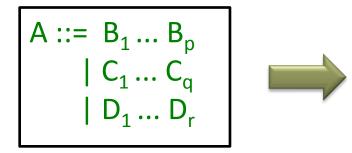
- computing first, nullable, follow for non-terminals of the grammar
- construction of parse table using this information
- LL(1) as an interpreter for the parse table

#### Grammar vs Recursive Descent Parser

```
expr ::= term termList
termList ::= + term termList
           term termList
term ::= factor factorList
factorList ::= * factor factorList
            / factor factorList
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 )")
```

# Rough General Idea



```
def A =
  if (token ∈ T1) {
    B<sub>1</sub> ... B<sub>p</sub>
  else if (token ∈ T2) {
    C<sub>1</sub> ... C<sub>q</sub>
  } else if (token ∈ T3) {
    D<sub>1</sub> ... D<sub>r</sub>
  } else error("expected T1,T2,T3")
```

#### where:

```
T1 = \mathbf{first}(\mathsf{B}_1 \dots \mathsf{B}_p)
T2 = \mathbf{first}(\mathsf{C}_1 \dots \mathsf{C}_q)
T3 = \mathbf{first}(\mathsf{D}_1 \dots \mathsf{D}_r)
\mathbf{first}(\mathsf{B}_1 \dots \mathsf{B}_p) = \{ a \in \Sigma \mid \mathsf{B}_1 \dots \mathsf{B}_p \implies aw \}
T1, T2, T3 \text{ should be } \mathbf{disjoint} \text{ sets of tokens.}
```

# Computing first in the example

```
expr ::= term termList
termList ::= + term termList
            term termList
term ::= factor factorList
factorList ::= * factor factorList
            | / factor factorList
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**

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

$$A ::= B_1 ... B_p$$
 $| C_1 ... C_q$ 
 $| D_1 ... D_r$ 



$$A ::= B_1 ... B_p$$
  
 $| C_1 ... C_q$   
 $| D_1 ... D_r$   
first(A) = first(B\_1 ... B\_p)  
U first(C\_1 ... C\_q)  
U first(D\_1 ... D\_r)

#### Sequences

$$first(B_1...B_p) = first(B_1)$$

if not nullable(B₁)

$$first(B_1...B_p) = first(B_1) \cup ... \cup first(B_k)$$

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

term ::= factor factorList
factorList ::= * factor factorList
            | / factor factorList
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



terminals: a,b

non-terminals: S, X, Y, Z

reachable (from S):

productive:

nullable:

First sets of terminals:

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

## **Example to Generate Constraints**

$$X ::= \mathbf{b} \mid S Y$$

$$Z := \varepsilon \mid \mathbf{a}$$



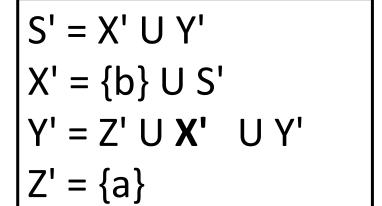
terminals: a,b

non-terminals: S, X, Y, Z

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

productive: X, Z, S, Y

nullable: 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?

#### Iterative Solution of **first** Constraints

- 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', Z' ∈ {0,1}

0 - not nullable

1 - nullable

| - disjunction

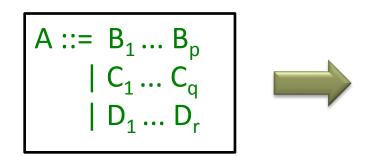
& - conjunction
```

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

# Rough General Idea



```
def A =

if (token ∈ T1) {

B_1 \dots B_p

else if (token ∈ T2) {

C_1 \dots C_q
} else if (token ∈ T3) {

D_1 \dots D_r
} else error("expected T1,T2,T3")
```

#### where:

```
T1 = first(B_1 ... B_p)
T2 = first(C_1 ... C_q)
T3 = first(D_1 ... D_r)
```

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

#### Exercise 1

```
A ::= B EOF
B ::= \epsilon | B B | (B)
```

- Tokens: **EOF**, (, )
- Generate constraints and compute nullable and first for this grammar.
- Check whether first sets for different alternatives are disjoint.

#### Exercise 2

```
S ::= B EOF
B ::= \varepsilon | B (B)
```

- Tokens: **EOF**, (, )
- Generate constraints and compute nullable and first for this grammar.
- Check whether first sets for different alternatives are disjoint.

#### Exercise 3

#### Compute nullable, first for this grammar:

```
stmtList ::= ɛ | 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
```

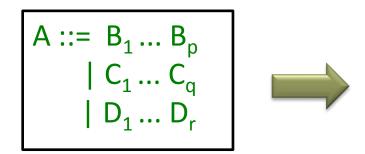
#### **Problem Identified**

```
stmtList ::= & | 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

# General Idea for nullable(A)



```
def A =
  if (token ∈ T1) {
    B<sub>1</sub> ... B<sub>p</sub>
  else if (token ∈ (T2 U T<sub>F</sub>)) {
    C<sub>1</sub> ... C<sub>q</sub>
  } else if (token ∈ T3) {
    D<sub>1</sub> ... D<sub>r</sub>
  } // no else error, just return
```

#### where:

```
T1 = \mathbf{first}(B_1 \dots B_p)
T2 = \mathbf{first}(C_1 \dots C_q)
T3 = \mathbf{first}(D_1 \dots D_r)
T_F = \mathbf{follow}(A)
```

Only one of the alternatives can be nullable (e.g. second) T1, T2, T3,  $T_F$  should be pairwise **disjoint** sets of tokens.

# 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)
- 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

$$first(B_1 ... B_p) = \{a \in \Sigma \mid B_1 ... B_p \implies ... \implies aw \}$$
$$follow(X) = \{a \in \Sigma \mid S \implies ... Xa... \}$$

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  $first(Z) \subseteq follow(Y)$   
and  $follow(X) \subseteq follow(Z)$   
now take care of nullable ones as well:

For each rule 
$$X := Y_1 ... Y_p ... Y_q ... Y_r$$
  
**follow**( $Y_p$ ) should contain:

- first( $Y_{p+1}Y_{p+2}...Y_r$ )
- also follow(X) if nullable(Y<sub>p+1</sub>Y<sub>p+2</sub>Y<sub>r</sub>)

## Compute nullable, first, follow

```
stmtList ::= ɛ | 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

$$B := \varepsilon \mid B(B)$$
(1) (2)

nullable: B

$$follow(S) = \{\}$$

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

#### Parsing table:

	EOF	(	)
S	{1}	{1}	
В	{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 ::= (1) B_1 ... B_p

| (2) C_1 ... C_q

| (3) D_1 ... D_r
```

```
\begin{split} &\text{if} \quad t \in \text{first}(C_1 \dots C_q) \quad \text{add 2} \\ &\quad \text{to choice}(A,t) \\ &\text{if} \quad t \in \text{follow}(A) \text{ add K to choice}(A,t) \\ &\text{where K is nullable alternative} \end{split}
```

For example, when parsing A and seeing token t choice(A,t) =  $\{2\}$  means: parse alternative 2 ( $C_1$  ...  $C_q$ ) choice(A,t) =  $\{1\}$  means: parse alternative 3 ( $D_1$  ...  $D_r$ ) choice(A,t) =  $\{\}$  means: report syntax error choice(A,t) =  $\{2,3\}$ : not LL(1) grammar

# Transform Grammar for LL(1)

S ::= B **EOF**
B ::= 
$$\varepsilon \mid B(B)$$
(1) (2)

Transform the grammar so that parsing table has no conflicts.

S ::= B **EOF**
B ::= 
$$\varepsilon \mid (B) B$$
(1) (2)

Left recursion is bad for LL(1)

#### Old parsing table:

	EOF	(	)
S	{1}	{1}	{}
В	{1}	{1,2}	{1}

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))

	EOF	(	)
S			
В			

choice(A,t)

#### Parse Table is Code for Generic Parser

```
var stack : Stack[GrammarSymbol] // terminal or non-terminal
stack.push(EOF);
stack.push(StartNonterminal);
var lex = new Lexer(inputFile)
while (true) {
X = stack.pop
t = lex.curent
 if (isTerminal(X))
  if (t==X) if (X==EOF) return success
            else lex.next // eat token t
  else parseError("Expected " + X)
 else { // non-terminal
  cs = choice(X)(t) // look up parsing table
  cs match { // result is a set
  case {i} => { // exactly one choice
   rhs = p(X,i) // choose correct right-hand side
   stack.push(reverse(rhs)) }
  case {} => parseError("Parser expected an element of " + unionOfAll(choice(X)))
  case _ => crash("parse table with conflicts - grammar was not LL(1)")
```

# What if we cannot transform the grammar into LL(1)?

1) Redesign your language

2) Use a more powerful parsing technique