

# 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 $\rightarrow$ 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  
 $L_G(N) = \{ \text{set of strings that N can derive} \}$

e.g.  $L(\text{statmt})$  – all statements of while language

$\text{first}(N) = \{ a \mid aw \text{ in } L_G(N), a - \text{terminal}, w - \text{string of terminals} \}$

$\text{first}(\text{statmt}) = \{ \text{println}, \text{ident}, \text{if}, \text{while}, \{ \} \}$

$\text{first}(\text{while ( expr ) statmt}) = \{ \text{while} \}$

- 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

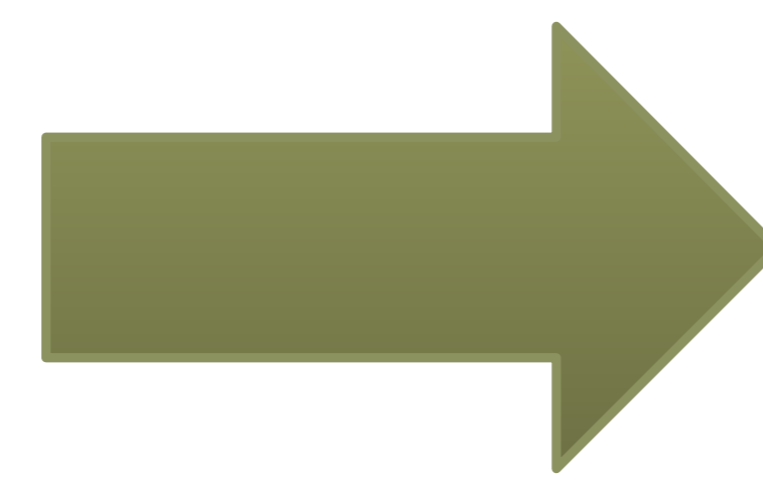
```
expr ::= term termList
termList ::= + term termList
           | - term termList
           | ε
term ::= factor factorList
factorList ::= * factor factorList
            | / factor factorList
            | ε
factor ::= name | ( expr )
name ::= ident
```

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

```
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

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



```
def A =  
  if (token ∈ T1) {  
    B1 ... Bp  
  } else if (token ∈ T2) {  
    C1 ... Cq  
  } else if (token ∈ T3) {  
    D1 ... Dr  
  } else error("expected T1,T2,T3")
```

where:

$T1 = \text{first}(B_1 \dots B_p)$

$T2 = \text{first}(C_1 \dots C_q)$

$T3 = \text{first}(D_1 \dots D_r)$

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

$T1, T2, T3$  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
```

$\text{first}(\text{name}) = \{\mathbf{ident}\}$

$\text{first}(\text{( expr )}) = \{ (\ }$

$\text{first}(\text{factor}) = \text{first}(\text{name})$   
 $\quad \cup \text{first}(\text{( expr )})$   
 $= \{\mathbf{ident}\} \cup \{ (\ }$   
 $= \{\mathbf{ident}, (\ }$

$\text{first}(* \text{ factor factorList}) = \{ * \}$

$\text{first}(/ \text{ factor factorList}) = \{ / \}$

$\text{first}(\text{factorList}) = \{ *, / \}$

$\text{first}(\text{term}) = \text{first}(\text{factor}) = \{\mathbf{ident}, (\ }$

$\text{first}(\text{termList}) = \{ +, - \}$

$\text{first}(\text{expr}) = \text{first}(\text{term}) = \{\mathbf{ident}, (\ }$

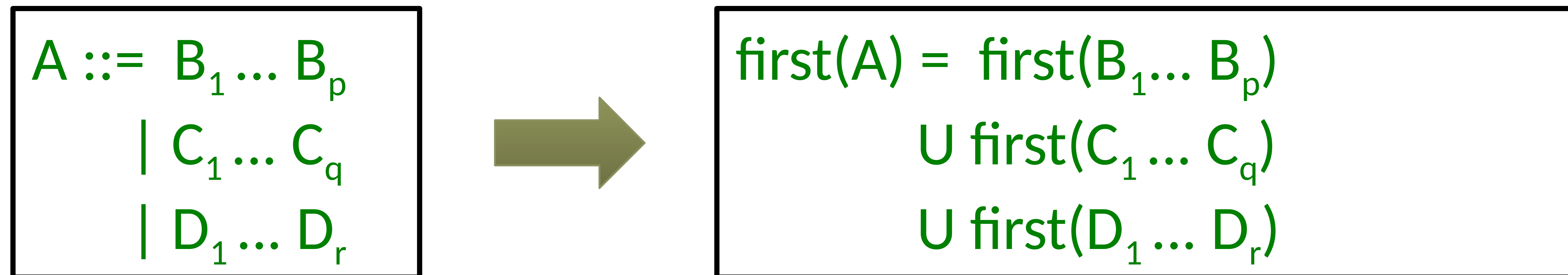
# Algorithm for **first**: Goal

Given an arbitrary context-free grammar with a set of rules of the form  $X ::= Y_1 \dots Y_n$  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



## Sequences

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

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

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:  $\text{expr}'$  is  $\text{first}(\text{expr})$

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

**nullable:**  $\text{termList}$ ,  $\text{factorList}$

```
expr' = term'
termList' = {+}
           U {-}
term' = factor'
factorList' = {*}
           U {/}
factor' = name' U { ( }
name' = { ident }
```

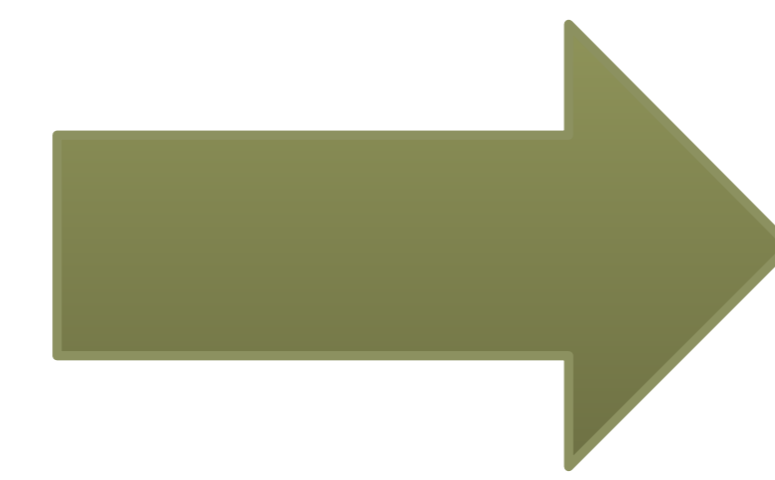
For this nice grammar, there is no recursion in constraints. Solve by substitution.

# Example to Generate Constraints

$$\begin{aligned} S &::= X \mid Y \\ X &::= \mathbf{b} \mid SY \\ Y &::= ZX\mathbf{b} \mid Y\mathbf{b} \\ Z &::= \varepsilon \mid \mathbf{a} \end{aligned}$$

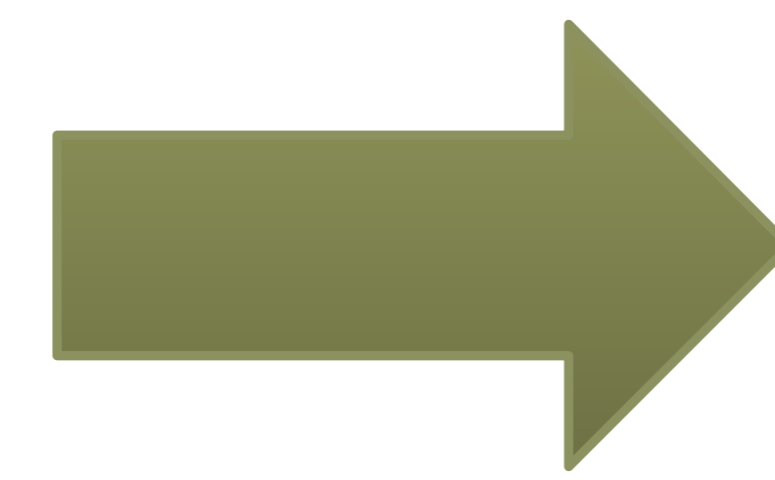
terminals:  $\mathbf{a}, \mathbf{b}$   
non-terminals:  $S, X, Y, Z$

reachable (from  $S$ ):  
productive:  
nullable:


$$\begin{aligned} S' &= X' \cup Y' \\ X' &= \end{aligned}$$

First sets of terminals:  
 $S', X', Y', Z' \subseteq \{a, b\}$

# Example to Generate Constraints

$$\begin{aligned} S &::= X \mid Y \\ X &::= b \mid SY \\ Y &::= ZXb \mid Yb \\ Z &::= \varepsilon \mid a \end{aligned}$$

$$\begin{aligned} S' &= X' \cup Y' \\ X' &= \{b\} \cup S' \\ Y' &= Z' \cup X' \cup Y' \\ Z' &= \{a\} \end{aligned}$$

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

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

$$S' = X' \cup Y'$$

$$X' = \{b\} \cup S'$$

$$Y' = Z' \cup X' \cup Y'$$

$$Z' = \{a\}$$

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

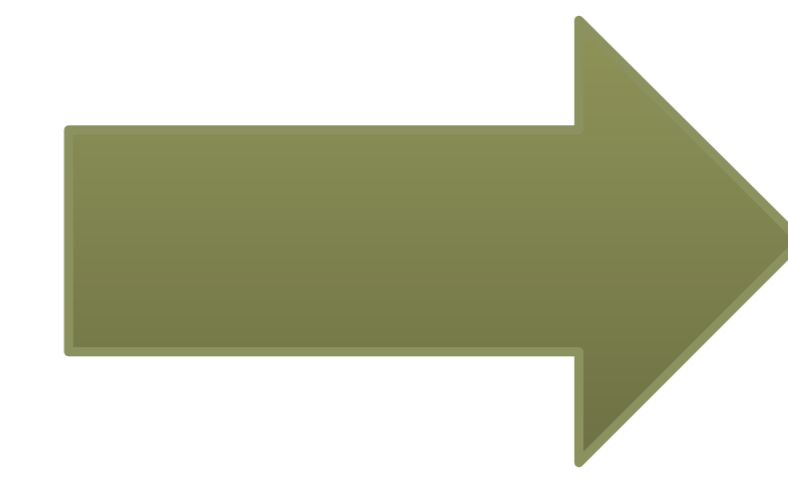
Sets grow in each step

- 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  $\varepsilon$

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



$S' = X' \mid Y'$   
 $X' = 0 \mid (S' \& Y')$   
 $Y' = (Z' \& X' \& 0) \mid (Y' \& 0)$   
 $Z' = 1 \mid 0$

$S', X', Y', Z' \in \{0,1\}$

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  $X$ :  $\text{first}(X) = \{\}$

**for each** terminal  $t$ :  $\text{first}(t) = \{t\}$

**repeat**

**for each** grammar rule  $X ::= Y(1) \dots Y(k)$

**for**  $i = 1$  to  $k$

**if**  $i=1$  or  $\{Y(1), \dots, Y(i-1)\} \subseteq \text{nullable}$  **then**

$\text{first}(X) = \text{first}(X) \cup \text{first}(Y(i))$

**until** none of  $\text{first}(\dots)$  changed in last iteration

# Follow sets. LL(1) Parsing Table

# Exercise Introducing Follow Sets

Compute nullable, first for this grammar:

$\text{stmtList} ::= \varepsilon \mid \text{stmt stmtList}$

$\text{stmt} ::= \text{assign} \mid \text{block}$

$\text{assign} ::= \mathbf{ID = ID ;}$

$\text{block} ::= \mathbf{\text{beginof ID stmtList ID ends}}$

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

$\text{stmtList} ::= \varepsilon \mid \text{stmt stmtList}$

$\text{stmt} ::= \text{assign} \mid \text{block}$

$\text{assign} ::= \mathbf{ID} = \mathbf{ID} ;$

$\text{block} ::= \mathbf{beginof ID stmtList ID ends}$

## Problem parsing $\text{stmtList}$ :

- $\mathbf{ID}$  could start alternative  $\text{stmt stmtList}$
- $\mathbf{ID}$  could **follow**  $\text{stmt}$ , so we may wish to parse  $\varepsilon$  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( $X$ ),  $\text{first}(X)$  must be disjoint from  $\text{follow}(X)$  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**

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

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

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

# Rule for Computing Follow

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

then  $\text{first}(Z) \subseteq \text{follow}(Y)$

and  $\text{follow}(X) \subseteq \text{follow}(Z)$

now take care of nullable ones as well:

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

$\text{follow}(Y_p)$  should contain:

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

## Compute nullable, first, follow

$\text{stmtList} ::= \varepsilon \mid \text{stmt stmtList}$

$\text{stmt} ::= \text{assign} \mid \text{block}$

$\text{assign} ::= \mathbf{ID = ID ;}$

$\text{block} ::= \mathbf{\text{beginof ID stmtList ID ends}}$

Is this grammar LL(1)?

# Conclusion of the Solution

The grammar is not LL(1) because we have

- nullable(stmtList)
- $\text{first}(\text{stmt}) \cap \text{follow}(\text{stmtList}) = \{\mathbf{ID}\}$
- 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

$S ::= B \text{ EOF}$

(1)

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

(1)

(2)

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

nullable: B

$\text{first}(S) = \{ (, \text{EOF} \}$

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

$\text{first}(B) = \{ ( \}$

$\text{follow}(B) = \{ ), (, \text{EOF} \}$

**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  $\rightarrow$  Set[Int]

$A ::=$  (1)  $B_1 \dots B_p$   
| (2)  $C_1 \dots C_q$   
| (3)  $D_1 \dots D_r$

if  $t \in \text{first}(C_1 \dots C_q)$  add 2  
to choice(A,t)  
if  $t \in \text{follow}(A)$  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 ( $C_1 \dots C_q$ )

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

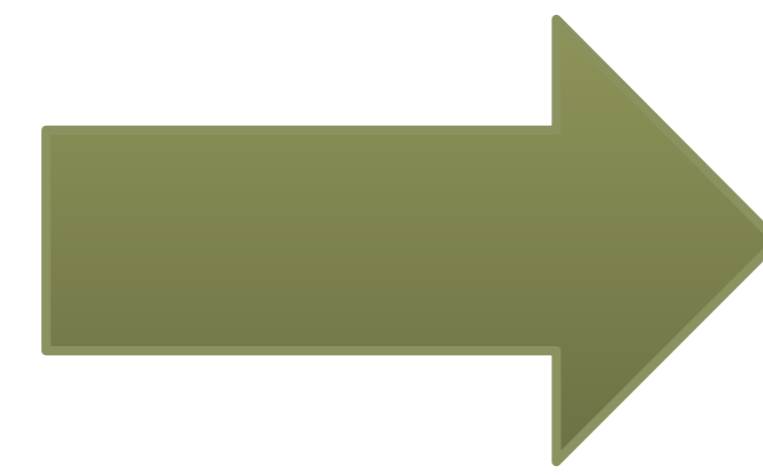
choice(A,t) = {} means: report syntax error

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



# General Idea when parsing nullable(A)

```
A ::= B1 ... Bp
      | C1 ... Cq
      | D1 ... Dr
```



```
def A =
  if (token ∈ T1) {
    B1 ... Bp
  } else if (token ∈ (T2 ∪ TF)) {
    C1 ... Cq
  } else if (token ∈ T3) {
    D1 ... Dr
  } // no else error, just return
```

where:

T1 = **first**(B<sub>1</sub> ... B<sub>p</sub>)

T2 = **first**(C<sub>1</sub> ... C<sub>q</sub>)

T3 = **first**(D<sub>1</sub> ... D<sub>r</sub>)

T<sub>F</sub> = **follow**(A)

Only one of the alternatives can be nullable (here: 2nd)  
T1, T2, T3, T<sub>F</sub> should be pairwise **disjoint** sets of tokens.