Exercise 1

A ::= B **EOF**

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

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 { stmat; 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) }

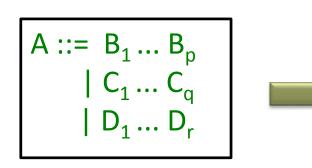
Problem Identified

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

General Idea when parsing nullable(A)



def A = if (token \in T1) { $B_1 \dots B_p$ else if (token \in (T2 U 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.

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

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

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)

$$\Rightarrow$$
 $Xa_{m} =)$
 YZa_{m}

now take care of nullable ones as well:

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

follow(Y_p) should contain:

- **first(** $Y_{p+1}Y_{p+2}...Y_{r}$ **)**
- 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}

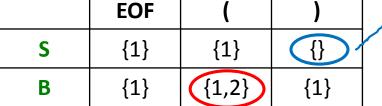
- If a recursive-descent parser sees ID, it does not know if it should
 - finish parsing stmtList or
 - parse another stmt

Table for LL(1) Parser: Example

```
S ::= B EOF
(1)
B ::= ε | B (B)
(1) (2)
```

nullable: B
first(S) = { (}
follow(S) = {}
first(B) = { (}
follow(B) = {), (, EOF }

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



Parsing table:

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)
$$B_1 \dots B_p$$

| (2) $C_1 \dots C_q$
| (3) $D_1 \dots D_r$

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

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

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

Transform Grammar for LL(1)

S ::= B EOF B ::= ε | B (B) (1) (2)

Transform the grammar so that parsing table has no conflicts.

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))

S ::= B EOF B ::= ε | (B) B (1) (2)

Left recursion is bad for LL(1)

	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.pushRev(rhs) } // pushes symbols in rhs so leftmost becomes top of stack **case** {} => parseError("Parser expected an element of " + unionOfAll(choice(X))) **case** $_$ => crash("parse table with conflicts - grammar was not LL(1)") }

Exercise: check if this grammar is LL(1)

- S::= ε | A S
- A ::= id **:=** id
- A ::= if id then A
- A ::= if id then A' else A
- A' ::= id **:=** id
- A' ::= if id then A' else A'

No, because first(**if** id **then** A) and first(**if** id **then** A' **else** A) overlap.

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

1) Redesign your language

2) Use a more powerful parsing technique