Section 4: CUP & LL

Kris, Michael, Gavin, Anand, Eunia

CSE 401/M501 – Compilers

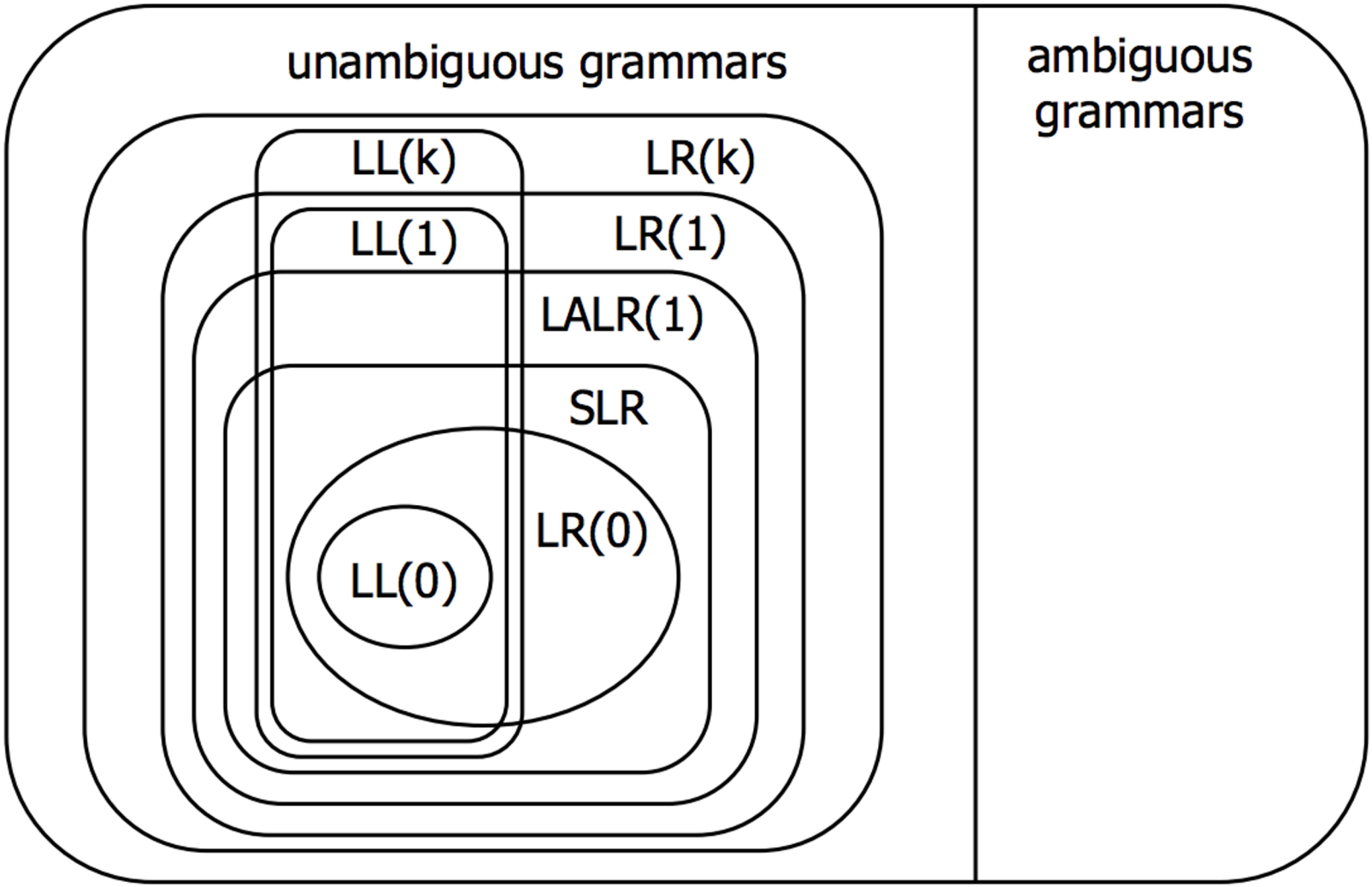
Administrivia

- Homework 2 is due tonight!
  - You have late days if you need them
- Parser is due one week from today
  - Be sure to check your Scanner feedback
- Watch demonstration videos!
  - CUP
  - AST Hierarchies

The CUP parser generator

- Uses LALR(1)
  - A little weaker (less selective), but many fewer states than LR(1) parsers
  - Handles most realistic programming language grammars
  - More selective than SLR (or LR(0)) about when to do reductions, so works for more languages

Language Hierarchies



The CUP parser generator

- Based on LALR(1)

- CUP can resolve some ambiguities itself
  - Precedence for reduce/reduce conflicts
  - Associativity for shift/reduce conflicts
  - Useful for our project for things like arithmetic expressions (exp+exp, exp*exp for fewer non-terminals, then add precedence and associativity declarations). Read the docs

LL(k) parsing

- LL(*k*) scans left-to-right, builds leftmost derivation, and looks ahead *k* symbols
- Typically *k* = 1, just like LR

- The LL condition enable the parser to choose productions correctly with 1 symbol of look-ahead
- We can often transform a grammar to satisfy this if needed

LL Condition

For each nonterminal in the grammar:

- Its *productions* must have disjoint FIRST sets

✗
```
A ::= x | B
B ::= x
```

✓
```
A ::= x | B
B ::= y
```

- If it is *nullable*, the FIRST sets of its productions must be disjoint from its FOLLOW set

✗
```
S ::= A x
A ::= ε | x
```

✓
```
S ::= A y
A ::= ε | x
```

Factoring out common prefixes

When multiple productions of a nonterminal share a common prefix, turn the different suffixes into a new nonterminal.

Greeting ::= “hello, ” “world” | “hello, ” “friend”
| “hello, ” *Name*
Name ::= “Sarah” | “John” | …

Greeting ::= “hello, ” *Address*
Address ::= “world” | “friend” | *Name*
Name ::= “Sarah” | “John” | …

Removing direct left recursion

When a nonterminal has left-recursive productions, turn the different suffixes into a new nonterminal, appended to the remaining productions.

Sum ::= *Sum* “+” *Sum* | *Sum* “-” *Sum* | *Constant*
Constant ::= “1” | “2” | “3” | …

Sum ::= *Constant SumTail*
SumTail ::= “+” *Constant SumTail* | “-” *Constant SumTail* | ε
Constant ::= “1” | “2” | “3” | …

Removing indirect left recursion

- Pseudocode from Cooper & Torczon:

```
impose an order on the nonterminals, A1, A2, ..., An

for i ← 1 to n do;
    for j ← 1 to i - 1 do;
        if ∃ a production Ai → Aj γ
            then replace Ai → Aj γ with one or more
                 productions that expand Aj
    end;
    rewrite the productions to eliminate
         any direct left recursion on Ai
end;
```

■ **FIGURE 3.6** Removal of Indirect Left Recursion.

- Rather conservative: no need to push A_j into A_i if you know that $A_j \not\Rightarrow \alpha A_i \beta$ for any α, β
- Be sure to order non-terminals and use that order

Removing indirect left recursion

When a nonterminal has another nonterminal (B) on the left of a production, rewrite that production to use all possible expansions of B. Repeat until the left side of every production is a terminal or direct left recursion. (Must choose an order to process nonterminals)

Expr ::= *Ternary* | *Addition*
Ternary ::= *Expr* “?” *Expr* “:” *Expr*
Addition ::= *Expr* “+” *Expr*

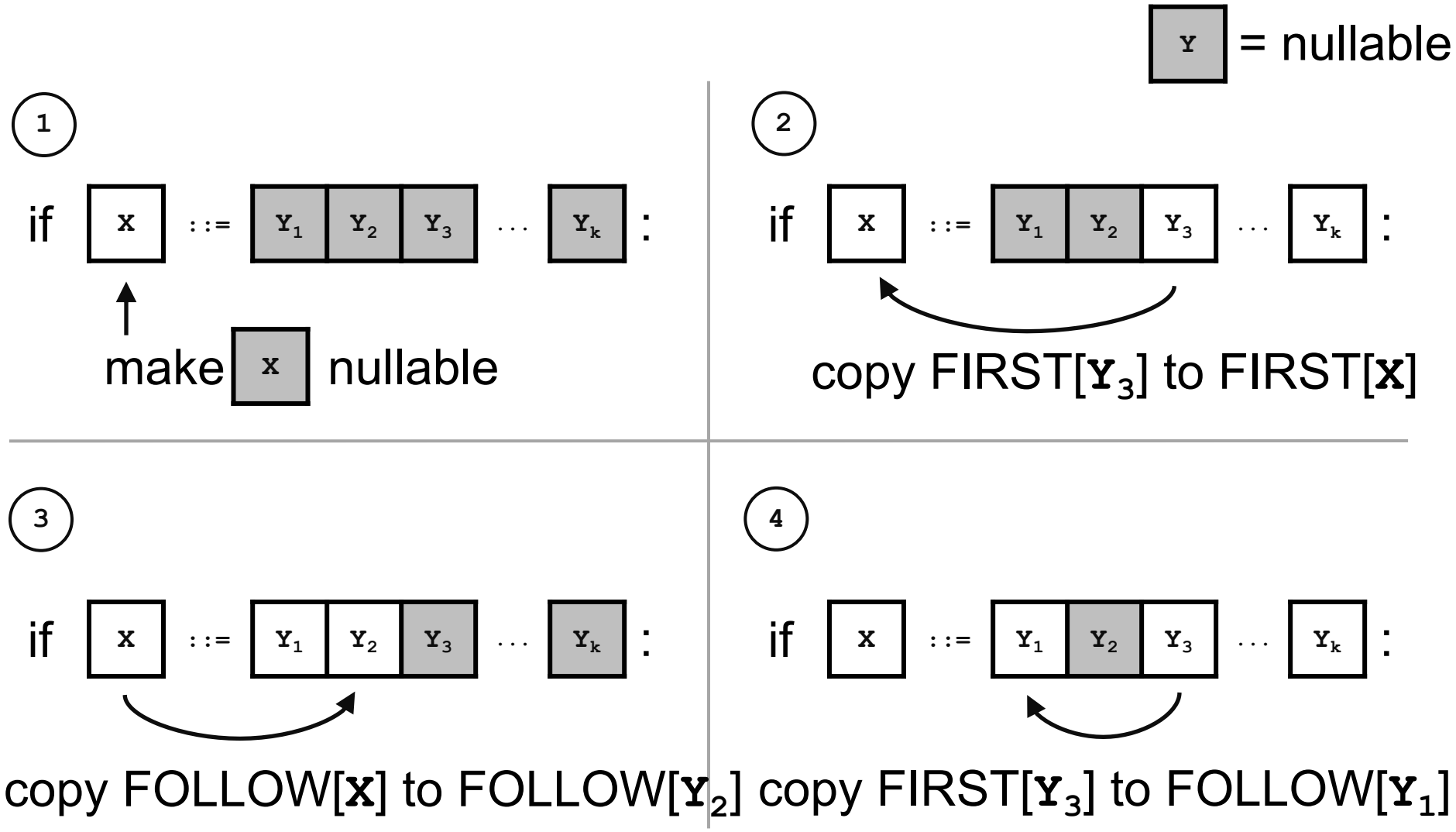
Expr ::= *Expr* “?” *Expr* “:” *Expr* | *Expr* “+” *Expr*

Worksheet

- Discuss and work in small groups!

- Reminders:
  - FIRST(α) is the set of terminal symbols that can begin a string derived from α
  - FOLLOW(*A*) is the set of terminal symbols that may immediately follow *A* in a derived string
  - nullable(*A*) is whether *A* can derive ε

Computing FIRST, FOLLOW, & nullable (3)



Computing FIRST, FOLLOW, and nullable

repeat
  for each production $X := Y_1\ Y_2 \ldots Y_k$
    if $Y_1 \ldots Y_k$ are all nullable (or if $k = 0$)
      set nullable[X] = true
    for each i from 1 to k and each j from i +1 to k
      if $Y_1 \ldots Y_{i-1}$ are all nullable (or if $i = 1$)
        add FIRST[Y_i] to FIRST[X]
      if $Y_{i+1} \ldots Y_k$ are all nullable (or if $i = k$)
        add FOLLOW[X] to FOLLOW[Y_i]
      if $Y_{i+1} \ldots Y_{j-1}$ are all nullable (or if i+1=j)
        add FIRST[Y_j] to FOLLOW[Y_i]
Until FIRST, FOLLOW, and nullable do not change