Section 4: CUP & LL

Jack Eggleston, Nate Yazdani & Aaron Johnston

CSE 401/M501 – Compilers

Autumn 2018

Administrivia

- Homework 2 is due tonight!
  - You have late days if you need them
- Parser is due one week from today
- Scanner feedback by next week
  - Be sure to check when debugging parser ☺

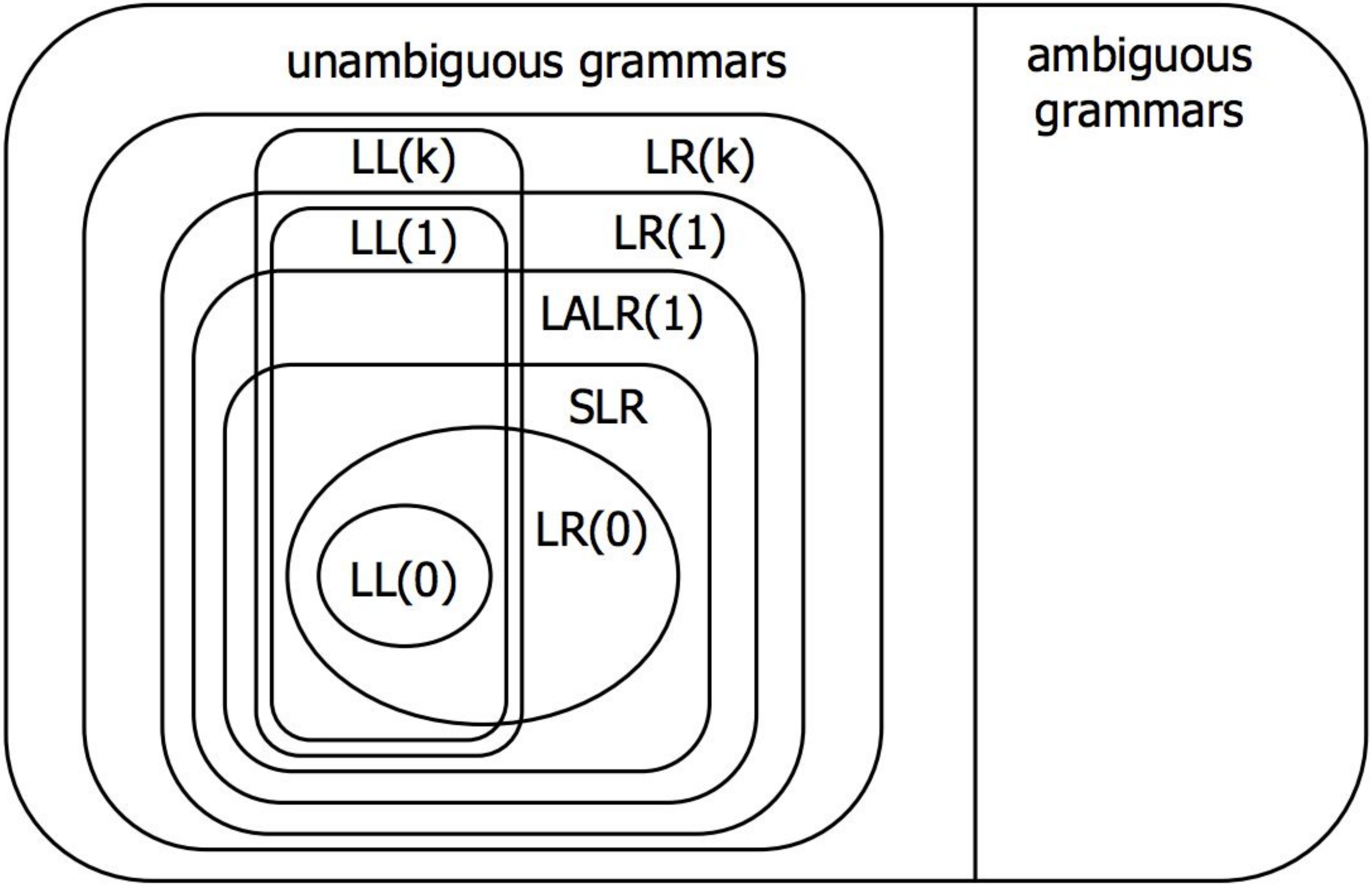
Agenda

- CUP tips, tricks, and demo
- LL parsing
  - See Sec. 3.3 of Cooper & Torczon for more
- A worksheet all about LL

The CUP parser generator

- Uses LALR(1)
  - Weaker but faster variant of LR(1)
- LALR is more sensitive to ambiguity than LR

Language Hierarchies



The CUP parser generator

- Uses LALR(1)
  - Weaker but faster variant of LR(1)
- LALR is more sensitive to ambiguity than LR

- CUP can resolve some ambiguities itself
  - Precedence for reduce/reduce conflicts
  - Associativity for shift/reduce conflicts
- If you use those features, read the docs carefully

The CUP parser generator

Demo: testing and debugging a CUP parser

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 transform a grammar to satisfy them

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 (“trails”) 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 (”trails”) into a new nonterminal, appended to the remaining productions.

$$Sum ::= Sum \text{ “+” } Sum \mid Sum \text{ “-” } Sum \mid Constant$$

$$Constant ::= \text{“1”} \mid \text{“2”} \mid \text{“3”} \mid \ldots$$

$$Sum ::= Constant\ SumTrail$$

$$SumTrail ::= \text{“+” } Sum \mid \text{“-” } Sum \mid \varepsilon$$

$$Constant ::= \text{“1”} \mid \text{“2”} \mid \text{“3”} \mid \ldots$$

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

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)

$$\mathit{Expr} ::= \mathit{Ternary} \mid \mathit{Addition}$$

$$\mathit{Ternary} ::= \mathit{Expr} \text{ “?” } \mathit{Expr} \text{ “:” } \mathit{Stmt}$$

$$\mathit{Addition} ::= \mathit{Expr} \text{ “+” } \mathit{Expr}$$

$$\mathit{Expr} ::= \mathit{Expr} \text{ “?” } \mathit{Expr} \text{ “:” } \mathit{Stmt} \mid \mathit{Expr} \text{ “+” } \mathit{Expr}$$

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, 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