Lecture 9

Syntax Analysis IV

Bottom-Up Parsing

Bottom-Up Parsing

- Given a grammar *G, a parse tree for a given string is constructed by* starting at the *leaves (terminals of the string) and working to the root (the start symbol S).*

- They are able to accept a more general class of grammars compared to top-down predictive parsers.

- It builds on the concepts developed in top-down parsing.
- Preferred method for most of the parser generators including *bison*

- They don't need left-factored grammars
  - So, its valid to use the following grammar

$$E \rightarrow T + E \mid T$$

$$T \rightarrow int * T \mid int \mid (E)$$

Bottom-Up Parsing

- A parse for a string generates a sequence of derivations of the form:

$$S \rightarrow \delta_0 \rightarrow \delta_1 \rightarrow \delta_2 \rightarrow \ldots \ldots \rightarrow \delta_{n-1} \rightarrow \textit{sentence}$$

- Bottom-up parsing *reduces a string to the start symbol by* inverting productions

- Let $A \rightarrow b$ *be a production and* δ_{i-1} *and* δ_i *be two* consecutive derivations with sentential forms: $\alpha A\beta$ *and* $\alpha b\beta$
  - δ_{i-1} *is derived from* δ_i *by match the RHS b in* δ_i*, and then replacing b with its corresponding LHS, A. This is called a* ***reduction***

- The **parse tree is the result of the tokens and the reductions.**

Bottom-Up Parsing

- Consider the parse for the input string: *int * int + int*

  E → T + E | T

  T → int * T | int | (E)

Sentential Form	Productions
int * <u>int</u> + int	
<u>int * T</u> + int	T → int
T + <u>int</u>	T → int * T
T + <u>T</u>	T → int
<u>T + E</u>	E → T
E	E → T+E

- When we reduce, we only have terminals to the right.
  - In the reduction: $\alpha A\beta \rightarrow \alpha b\beta$
  - If *b to A is a step of a bottom-up* parsing (i.e. $A \rightarrow b$ is a reduction)
  - Then β is a string of terminals

- In other words, a *bottom-up parser traces a rightmost derivation in reverse*

Shift-Reduce Parsing

- Idea: Split string being parsed into two parts:
  - Two parts are separated by a special character '|'
  - Left part is a string of terminals and non terminals
  - Right part is a string of terminals
    - Still to be examined
- Bottom up parsing has two actions
  - Shift: Move terminal symbol from right string to left string
    - **ABC | xyz → ABCx | yz**
  - Reduce: Apply an inverse production at the right end of the left string
  - If A → xy is a production, then
    - **Cbxy | ijk → CbA | ijk**

Shift-Reduce Example

Sentential Form	Productions
|int * int + int	Shift
int | * int + int	Shift
int * | int + int	Shift
int * int | + int	Reduce T → int
int * T | + int	Reduce T → int * T
T | + int	Shift
T + | int	Shift
T + int |	Reduce T → int
T + T |	Reduce E → T
T + E |	Reduce E → T+E
E |	Accept

To Shift or Reduce

- Symbols on the left of "|" are kept on a stack
  - Top of the stack is at "|"
  - Shift pushes a terminal on the stack
  - Reduce pops symbols (RHS of production) and pushes a non terminal (LHS of production) onto the stack

- The most important issues are:
  - When to shift and when to reduce!
  - Which production to use for reduction?
  - Sometimes parser can reduce but it should not!
  - $X \rightarrow \epsilon$ can always be reduced!
  - Sometimes parser can reduce in different ways!

To Shift or Reduce ? - Conflicts

- In a given state, more than one action (shift or reduce) may lead to a valid parse
  - If it is legal to shift or reduce:
    - ***Shift-reduce conflict***
  - If it is legal to reduce by two different productions:
    - ***Reduce-reduce conflict***

- Reduce action should be taken only if the result can be reduced to the start symbol

Shift-Reduce Parsing - Handles

- A substring that matches the right-side of a production that occurs as one step in the rightmost derivation. This substring is called a *handle*.

- Because d is a right-sentential form, the substring to the right of a handle contains only terminal symbols. Therefore, the parser doesn't need to scan past the handle.

- If a grammar is unambiguous, then every right sentential form has a unique handle

- If we can find those handles, we can build a derivation

Recognizing Handles

- Given the grammar: $E \rightarrow T + E \mid T$
  $T \rightarrow int * T \mid int \mid (E)$
- Consider step : *int | * int + int*
- We could reduce by $T \rightarrow$ int giving T | * int + int
  - But this is incorrect because:
    - No way to reduce to the start symbol E
- So, a handle is a reduction that also allows further reductions back to the start symbol
- In shift-reduce parsing, handles appear only at the top of the stack, never inside

Recognizing Handles

- Handles always appear only at stack top:
  - Immediately after reducing a handle
  - Right-most non-terminal on top of the stack
  - Next handle must be to right of right-most non-terminal, because this is a right-most derivation
  - Sequence of shift moves reaches next handle

- It is not obvious how to detect handles
- At each step the parser sees only the stack, not the entire input; start with that . . .
- α is a *viable prefix if there is a β such that $\alpha|\beta$ is a state of* a shift-reduce parser

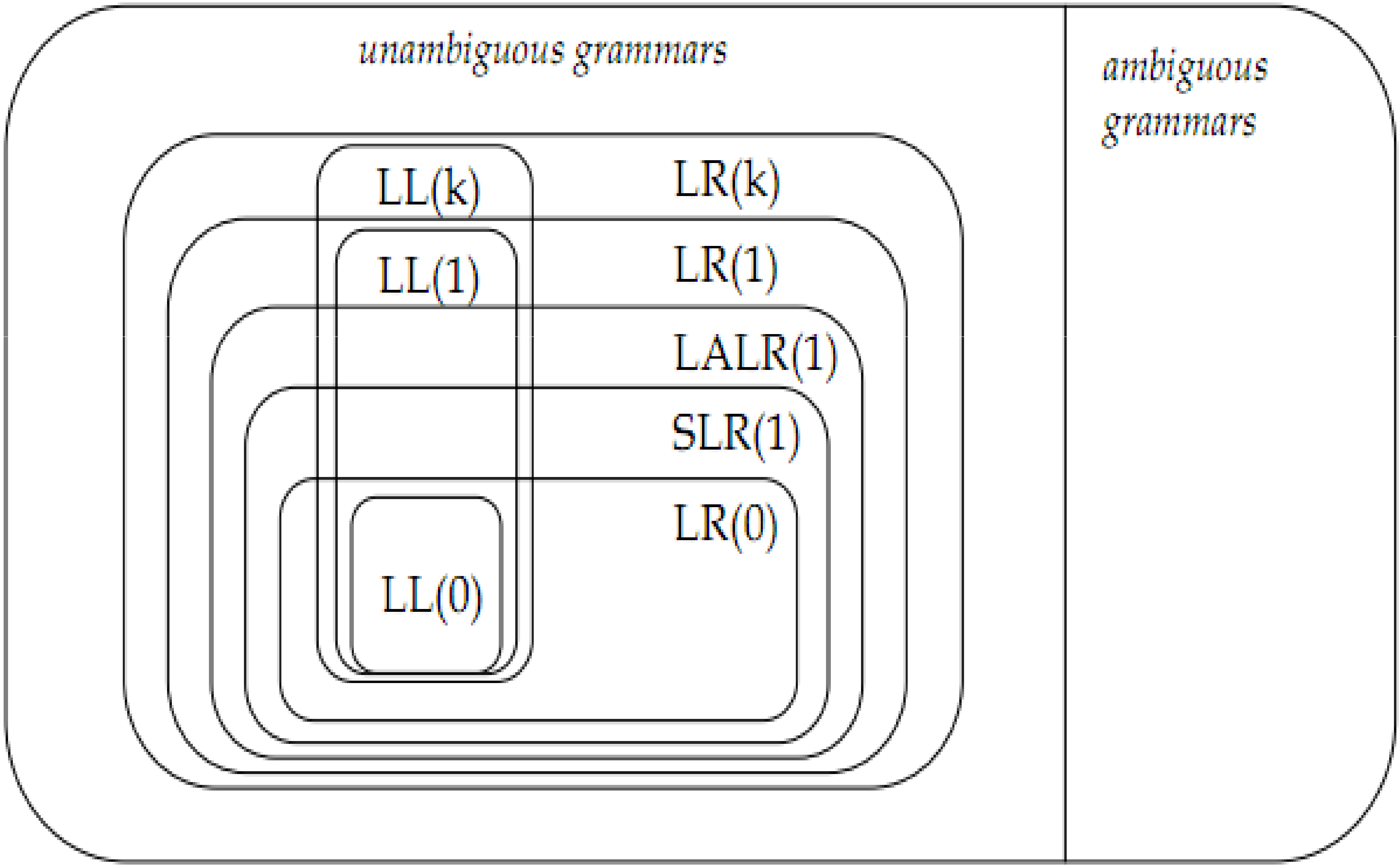
Viable Prefixes

- A viable prefix does not extend past the right end of the handle
- It's a viable prefix because it is a prefix of the handle
- As long as a parser has viable prefixes on the stack no parsing error has been detected

- *For any grammar, the set of viable prefixes is a regular language*

- *So, we can generate an automata to recognize viable prefixes!*

Hierarchy of Grammar Class



to be continued...