EECS 483: Compiler Construction

Lecture 23:
LL(1) Grammars, Top-Down Parsing

Month Day
Winter Semester 2025

Announcements

- Assignment 5 Autograder released

  Starter code updated with additional public test cases, small bugfix to the cli.

  Get started! Different structure from previous assignments

Searching for derivations.

LL & LR PARSING

CFGs Mathematically

- A Context-free Grammar (CFG) consists of
  - A set of *terminals* (e.g., a token or ε)
  - A set of *nonterminals* (e.g., S and other syntactic variables)
  - A designated nonterminal called the *start symbol*
  - A set of productions: LHS $\longmapsto$ RHS
    - LHS is a nonterminal
    - RHS is a *string* of terminals and nonterminals

- Example: The balanced parentheses language:

$$S \longmapsto (S)S$$

$$S \longmapsto \varepsilon$$

Derivations in CFGs

- Example: derive (1 + 2 + (3 + 4)) + 5
- $\underline{\mathbf{S}} \longmapsto \underline{\mathbf{E}} + S$

$\longmapsto (\underline{\mathbf{S}}) + S$

$\longmapsto (\underline{\mathbf{E}} + S) + S$

$\longmapsto (1 + \underline{\mathbf{S}}) + S$

$\longmapsto (1 + \underline{\mathbf{E}} + S) + S$

$\longmapsto (1 + 2 + \underline{\mathbf{S}}) + S$

$\longmapsto (1 + 2 + \underline{\mathbf{E}}) + S$

$\longmapsto (1 + 2 + (\underline{\mathbf{S}})) + S$

$\longmapsto (1 + 2 + (\underline{\mathbf{E}} + S)) + S$

$\longmapsto (1 + 2 + (3 + \underline{\mathbf{S}})) + S$

$\longmapsto (1 + 2 + (3 + \underline{\mathbf{E}})) + S$

$\longmapsto (1 + 2 + (3 + 4)) + \underline{\mathbf{S}}$

$\longmapsto (1 + 2 + (3 + 4)) + \underline{\mathbf{E}}$

$\longmapsto (1 + 2 + (3 + 4)) + 5$

$S \longmapsto E + S \mid E$

$E \longmapsto \text{number} \mid (S)$

For arbitrary strings α, β, γ and production rule $A \longmapsto \beta$
a single step of the derivation is:

$$\alpha A \gamma \longmapsto \alpha \beta \gamma$$

(*substitute* β for an occurrence of A)

In general, there are many possible derivations for a given string

Note: Underline indicates symbol being expanded.

Example: Left- and rightmost derivations

S ⟼ E + S | E
E ⟼ number | (S)

- Leftmost derivation:
- **S** ⟼ **E** + S

⟼ (**S**) + S
⟼ (**E** + S) + S
⟼ (1 + **S**) + S
⟼ (1 + **E** + S) + S
⟼ (1 + 2 + **S**) + S
⟼ (1 + 2 + **E**) + S
⟼ (1 + 2 + (**S**)) + S
⟼ (1 + 2 + (**E** + S)) + S
⟼ (1 + 2 + (3 + **S**)) + S
⟼ (1 + 2 + (3 + **E**)) + S
⟼ (1 + 2 + (3 + 4)) + **S**
⟼ (1 + 2 + (3 + 4)) + **E**
⟼ (1 + 2 + (3 + 4)) + 5

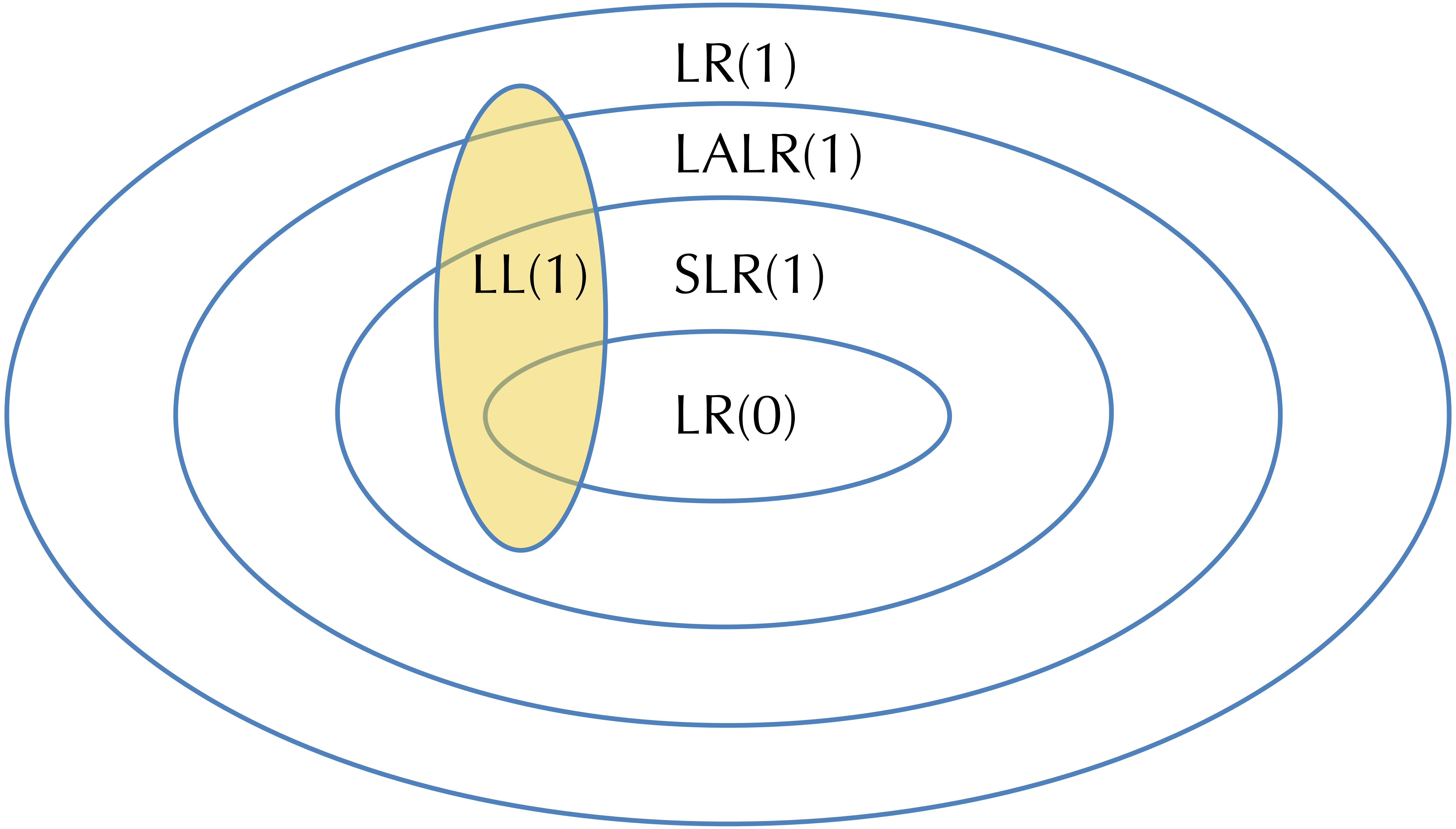
Rightmost derivation:

S ⟼ E + **S**
⟼ E + **E**
⟼ **E** + 5
⟼ (**S**) + 5
⟼ (E + **S**) + 5
⟼ (E + E + **S**) + 5
⟼ (E + E + **E**) + 5
⟼ (E + E + (**S**)) + 5
⟼ (E + E + (E + **S**)) + 5
⟼ (E + E + (E + **E**)) + 5
⟼ (E + E + (**E** + 4)) + 5
⟼ (E + **E** + (3 + 4)) + 5
⟼ (**E** + 2 + (3 + 4)) + 5
⟼ (1 + 2 + (3 + 4)) + 5

CFGs In Practice

- Context-free Grammars are elegant, *declarative* specifications, generalizing regular expressions
- A parser for a CFG amounts to a *search procedure* for derivations
- Unlike regular expressions, which are easily compiled to linear time recognizers, practical algorithms for parsing *general* CFGs are O(n^3) in input string length
  - Compromise: add restrictions to the CFGs
    - Benefit: Linear time
    - Drawback: have to rewrite the grammar to make it fit the restrictions

Classification of Grammars



LL(1) GRAMMARS

Consider finding left-most derivations

- Look at only one input symbol at a time.

$S \longmapsto E + S \;|\; E$
$E \longmapsto \text{number} \;|\; (\,S\,)$

Partly-derived String	Look-ahead	Parsed/Unparsed Input
S	(	(1 + 2 + (3 + 4)) + 5
⟼ **E** + S	(	(1 + 2 + (3 + 4)) + 5
⟼ (**S**) + S	1	(1 + 2 + (3 + 4)) + 5
⟼ (**E** + S) + S	1	(1 + 2 + (3 + 4)) + 5
⟼ (1 + **S**) + S	2	(1 + 2 + (3 + 4)) + 5
⟼ (1 + **E** + S) + S	2	(1 + 2 + (3 + 4)) + 5
⟼ (1 + 2 + **S**) + S	(	(1 + 2 + (3 + 4)) + 5
⟼ (1 + 2 + **E**) + S	(	(1 + 2 + (3 + 4)) + 5
⟼ (1 + 2 + (**S**)) + S	3	(1 + 2 + (3 + 4)) + 5
⟼ (1 + 2 + (**E** + S)) + S	3	(1 + 2 + (3 + 4)) + 5
⟼ …		

There is a problem

- We want to decide which production to apply based on the look-ahead symbol.
- But, there is a choice:

$S \longmapsto E + S \mid E$
$E \longmapsto \text{number} \mid (S)$

(1) $S \longmapsto E \longmapsto (S) \longmapsto (E) \longmapsto (1)$

vs.

(1) + 2 $S \longmapsto E + S \longmapsto (S) + S \longmapsto (E) + S \longmapsto (1) + S \longmapsto (1) + E \longmapsto (1) + 2$

- Given the look-ahead symbol: '(' it isn't clear whether to pick $S \longmapsto E$ or $S \longmapsto E + S$ first.

Grammar is the problem

- Not all grammars can be parsed "top-down" with only a single lookahead symbol.
- *Top-down*: starting from the start symbol (root of the parse tree) and going down

- LL(1) means
  - **L**eft-to-right scanning
  - **L**eft-most derivation,
  - **1** lookahead symbol

- This language isn't "LL(1)"
- Is it LL(k) for some k?

$$S \longmapsto E + S \mid E$$
$$E \longmapsto \text{number} \mid (S)$$

- What can we do?

Making a grammar LL(1)

- *Problem:* We can't decide which S production to apply until we see the symbol after the first expression.
- *Solution:* "Left-factor" the grammar. There is a common S prefix for each choice, so add a new non-terminal S′ at the decision point:

$$S \longmapsto E + S \mid E$$
$$E \longmapsto \text{number} \mid (S)$$

→

$$S \longmapsto ES'$$
$$S' \longmapsto \varepsilon$$
$$S' \longmapsto + S$$
$$E \longmapsto \text{number} \mid (S)$$

- Also need to eliminate left-recursion somehow. Why?
- Consider:

$$S \longmapsto S + E \mid E$$
$$E \longmapsto \text{number} \mid (S)$$

LL(1) Parse of the input string

- Look at only one input symbol at a time.

$S \longmapsto ES'$
$S' \longmapsto \varepsilon$
$S' \longmapsto + S$
$E \longmapsto \text{number} \mid (S)$

Partly-derived String	Look-ahead	Parsed/Unparsed Input
S	(	(1 + 2 + (3 + 4)) + 5
⟼ **E** S′	(	(1 + 2 + (3 + 4)) + 5
⟼ (**S**) S′	1	(1 + 2 + (3 + 4)) + 5
⟼ (**E** S′) S′	1	(1 + 2 + (3 + 4)) + 5
⟼ (1 **S′**) S′	+	(1 + 2 + (3 + 4)) + 5
⟼ (1 + **S**) S′	2	(1 + 2 + (3 + 4)) + 5
⟼ (1 + **E** S′) S′	2	(1 + 2 + (3 + 4)) + 5
⟼ (1 + 2 **S′**) S′	+	(1 + 2 + (3 + 4)) + 5
⟼ (1 + 2 + **S**) S′	(	(1 + 2 + (3 + 4)) + 5
⟼ (1 + 2 + **E** S′) S′	(	(1 + 2 + (3 + 4)) + 5
⟼ (1 + 2 + (**S**)S′) S′	3	(1 + 2 + (3 + 4)) + 5

Predictive Parsing

- Given an LL(1) grammar:
  - For a given nonterminal, the lookahead symbol uniquely determines the production to apply.
  - Top-down parsing = predictive parsing
  - Driven by a predictive parsing table:
    nonterminal * input token → production

T ⟼ S$
S ⟼ ES′
S′ ⟼ ε
S′ ⟼ + S
E ⟼ number | (S)

	number	+	(	)	$ (EOF)
T	⟼ S$		⟼S$		
S	⟼ E S′		⟼E S′		
S′		⟼ + S		⟼ ε	⟼ ε
E	⟼ num		⟼ (S)		

- Note: it is convenient to add a special *end-of-file* token $ and a start symbol T (top-level) that requires $.

How do we construct the parse table?

- Consider a given production: A → γ
- Construct the set of all input tokens that may appear *first* in strings that can be derived from γ
  - Add the production → γ to the entry (A,token) for each such token.
- If γ can derive ε (the empty string), then we construct the set of all input tokens that may *follow* the nonterminal A in the grammar.
  - Add the production → γ to the entry (A, token) for each such token.

- Note: The grammar is LL(1) *if and only if* all entries have at most one production

Example

- First(T) = First(S)
- First(S) = First(E)
- First(S′) = { + }
- First(E) = { number, ‘(‘ }

- Follow(S′) = Follow(S)
- Follow(S) = { $, ‘)’ } ∪ Follow(S′)

Note: we want the *least* solution to this system of set equations… a *fixpoint* computation. Just like in program analysis!

T ⟼ S$
S ⟼ ES′
S′ ⟼ ε
S′ ⟼ + S
E ⟼ number | (S)

	number	+	(	)	$ (EOF)
T	⟼ S$		⟼S$		
S	⟼ E S′		⟼E S′		
S′		⟼ + S		⟼ ε	⟼ ε
E	⟼ num.		⟼ (S)		

Converting the table to code

- Define n mutually recursive functions
  - one for each nonterminal A: parse_A
  - The type of parse_A is `() -> ast` if A is *not* an auxiliary nonterminal
  - Parse functions for auxiliary nonterminals (e.g., S′) take extra ast's as inputs, one for each nonterminal in the "factored" prefix.

- Each function "peeks" at the lookahead token and then follows the production rule in the corresponding entry.
  - Consume terminal tokens from the input stream
  - Call parse_X to create sub-tree for nonterminal X
  - If the rule ends in an auxiliary nonterminal, call it with appropriate ast's. (The auxiliary rule is responsible for creating the ast after looking at more input.)
  - Otherwise, this function builds the ast tree itself and returns it.

	number	+	(	)	$ (EOF)
T	⟼ S$		⟼S$		
S	⟼ E S′		⟼E S′		
S′		⟼ + S		⟼ ε	⟼ ε
E	⟼ num.		⟼ (S)		

Hand-generated LL(1) code for the table above.

DEMO: HANDPARSER.RS

LL(1) Summary

- Top-down parsing that finds the leftmost derivation.
- Language Grammar
  ⇒ LL(1) grammar (manual rewrite)

  ⇒ prediction table (intermediate representation)

  ⇒ recursive-descent parser (code generation)

- Problems:
  - Grammar must be LL(1)
  - Can extend to LL(k) (it just makes the table bigger)
  - Grammar cannot be left recursive (parser functions will loop!)

- Is there a better way?

Next time

LR GRAMMARS