Lecture 12

EECS 483:
COMPILER CONSTRUCTION

Announcements

- Midterm: Tuesday, March 12th
  - 7-9pm, DOW 1013 and 1014 (seat assignments will be announced later)
  - One-page, letter-sized, double-sided "cheat sheet" of notes permitted
  - Coverage: interpreters / program transformers / x86 / calling conventions / IRs / LLVM / Lexing / Parsing
  - See examples of previous exams on the web pages
  - March 11 class: review/office hours, no lecture

- HW4: Compiling Oat v.1
  - Lexing + Parsing + translate to LLVMlite
  - released after Spring Break
  - due March 26th

parser.mly, lexer.mll, range.ml, ast.ml, main.ml

DEMO: BOOLEAN LOGIC

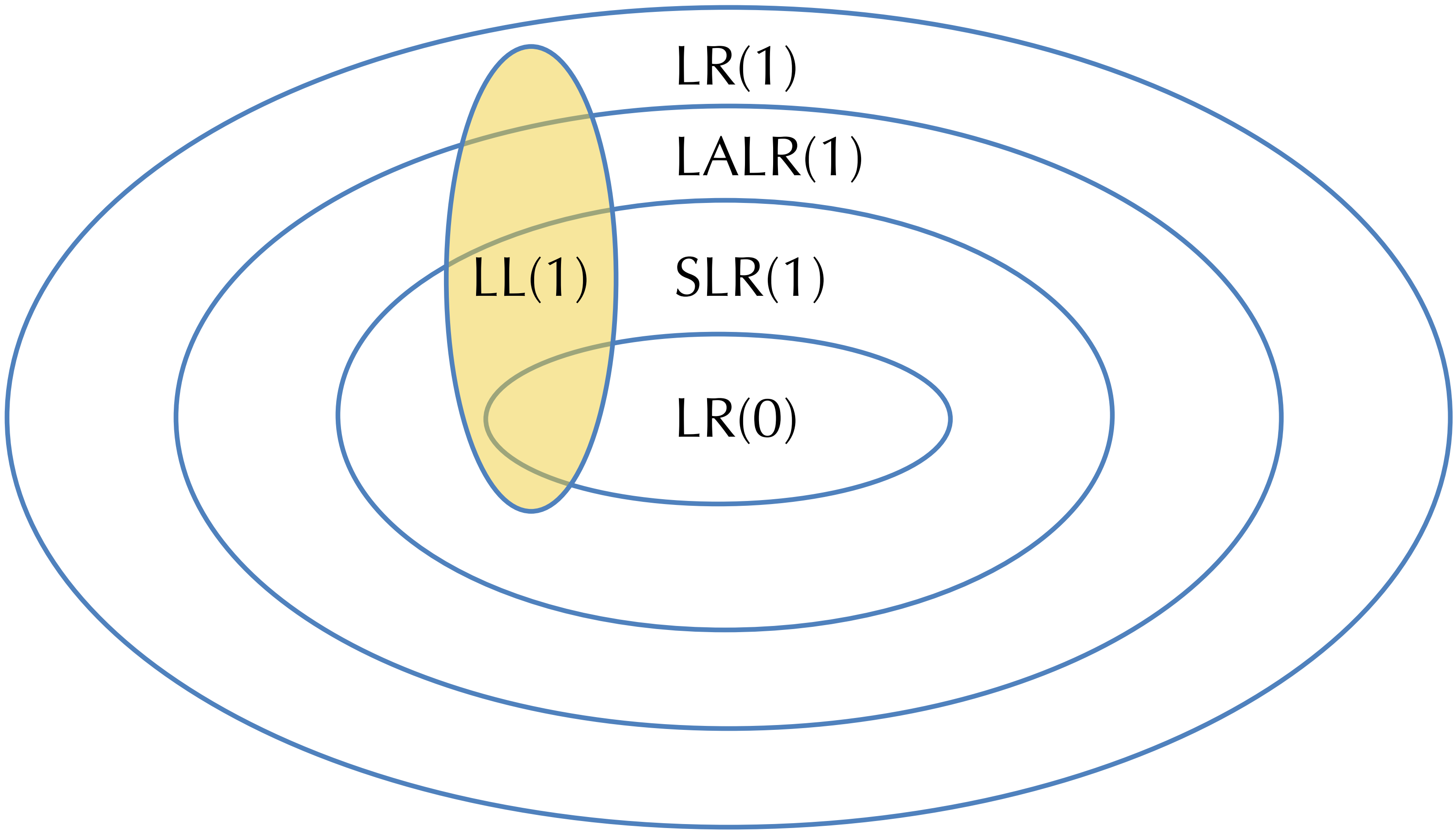
Searching for derivations.

LL & LR PARSING

The Parsing Problem

- The Parsing Problem:
  - Input: a context-free grammar G
  - Output: a **parser** that takes in a string and outputs a parse tree of that string in G or raises an exception if there is no parse tree.
  - Notice that an ambiguous grammar may be parsed in multiple ways
- In practice: fuse the generation of the parse tree with *semantic actions* that construct the abstract syntax tree
  - The parse tree is usually never "materialized" in memory
- Another "mini-compiler" for a DSL
- Bad news: best algorithms are O(n^3)
  - CYK, Earley, GLR algorithms
- Compromise: find restrictions on CFGs that allow for O(n) parsing
  - Intuition: parsing is a ***search problem***, find restrictions that limit the amount of backtracking needed.
  - Cost: more burden on the programmer (i.e., ***you***) to adapt their grammar to fit the restriction

Classification of Grammars



Top-down vs. Bottom up

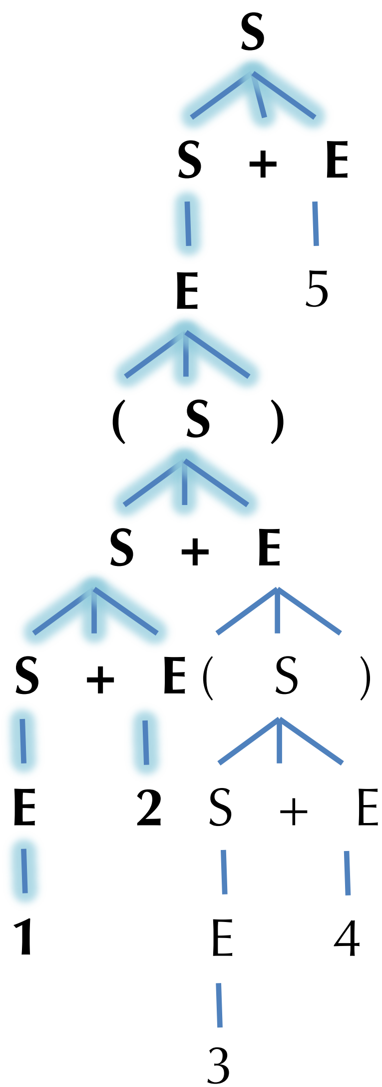
- Consider the left-recursive grammar:

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

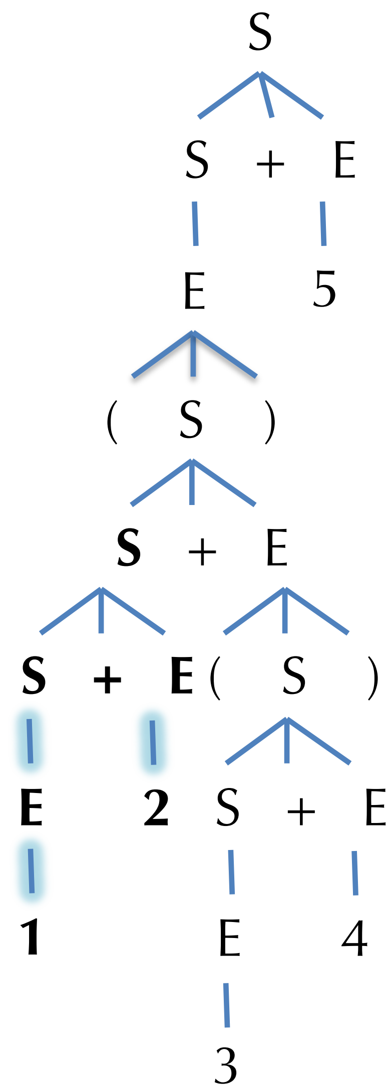
- (1 + 2 + (3 + 4)) + 5

- We want to parse by doing a linear scan, left-to-right

- Top-down: construct a partial tree from the root
- Bottom-up: construct partial tree from the leaves



Top-down



Bottom-up

LL(1) GRAMMARS AND TOP-DOWN PREDICTIVE 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:

$$S \longmapsto (S)S$$

$$S \longmapsto \varepsilon$$

Consider finding left-most derivations

- Look at only one input symbol at a time.

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

Partly-derived String	Look-ahead	Parsed/Unparsed Input
$\underline{\mathbf{S}}$	(	(1 + 2 + (3 + 4)) + 5
$\longmapsto \underline{\mathbf{E}} + S$	(	(1 + 2 + (3 + 4)) + 5
$\longmapsto (\underline{\mathbf{S}}) + S$	1	(1 + 2 + (3 + 4)) + 5
$\longmapsto (\underline{\mathbf{E}} + S) + S$	1	(1 + 2 + (3 + 4)) + 5
$\longmapsto (1 + \underline{\mathbf{S}}) + S$	2	(1 + 2 + (3 + 4)) + 5
$\longmapsto (1 + \underline{\mathbf{E}} + S) + S$	2	(1 + 2 + (3 + 4)) + 5
$\longmapsto (1 + 2 + \underline{\mathbf{S}}) + S$	(	(1 + 2 + (3 + 4)) + 5
$\longmapsto (1 + 2 + \underline{\mathbf{E}}) + S$	(	(1 + 2 + (3 + 4)) + 5
$\longmapsto (1 + 2 + (\underline{\mathbf{S}})) + S$	3	(1 + 2 + (3 + 4)) + 5
$\longmapsto (1 + 2 + (\underline{\mathbf{E}} + S)) + S$	3	(1 + 2 + (3 + 4)) + 5
$\longmapsto \ldots$		

There is a problem

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

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

(1) $\boxed{S \longmapsto E} \longmapsto (S) \longmapsto (E) \longmapsto (1)$

vs.

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

- 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:

$$
\begin{aligned}
S &\longmapsto E + S \mid E \\
E &\longmapsto \text{number} \mid (S)
\end{aligned}
\quad\Longrightarrow\quad
\begin{aligned}
S &\longmapsto E S' \\
S' &\longmapsto \varepsilon \\
S' &\longmapsto + S \\
E &\longmapsto \text{number} \mid (S)
\end{aligned}
$$

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

$$
\begin{aligned}
S &\longmapsto S + E \mid E \\
E &\longmapsto \text{number} \mid (S)
\end{aligned}
$$

LL(1) Parse of the input string

- Look at only one input symbol at a time.

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

Partly-derived String	Look-ahead	Parsed/Unparsed Input
$\underline{\mathbf{S}}$	(	(1 + 2 + (3 + 4)) + 5
$\mapsto \underline{\mathbf{E}}\ S'$	(	(1 + 2 + (3 + 4)) + 5
$\mapsto (\underline{\mathbf{S}})\ S'$	1	**(** 1 + 2 + (3 + 4)) + 5
$\mapsto (\underline{\mathbf{E}}\ S')\ S'$	1	**(** 1 + 2 + (3 + 4)) + 5
$\mapsto (1\ \underline{\mathbf{S'}})\ S'$	+	**(1** + 2 + (3 + 4)) + 5
$\mapsto (1 + \underline{\mathbf{S}})\ S'$	2	**(1 +** 2 + (3 + 4)) + 5
$\mapsto (1 + \underline{\mathbf{E}}\ S')\ S'$	2	**(1 +** 2 + (3 + 4)) + 5
$\mapsto (1 + 2\ \underline{\mathbf{S'}})\ S'$	+	**(1 + 2** + (3 + 4)) + 5
$\mapsto (1 + 2 + \underline{\mathbf{S}})\ S'$	(	**(1 + 2 +** (3 + 4)) + 5
$\mapsto (1 + 2 + \underline{\mathbf{E}}\ S')\ S'$	(	**(1 + 2 +** (3 + 4)) + 5
$\mapsto (1 + 2 + (\underline{\mathbf{S}})S')\ S'$	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

$T \longmapsto S\$$
$S \longmapsto ES'$
$S' \longmapsto \varepsilon$
$S' \longmapsto + S$
$E \longmapsto \text{number} \mid (S)$

	number	+	(	)	$ (EOF)
T	$\longmapsto S\$$		$\longmapsto S\$$		
S	$\longmapsto E\ S'$		$\longmapsto E\ S'$		
S′		$\longmapsto + S$		$\longmapsto \varepsilon$	$\longmapsto \varepsilon$
E	$\longmapsto$ num.		$\longmapsto (S)$		

- 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: $A \rightarrow \gamma$
- Construct the set of all input tokens that may appear *first* in strings that can be derived from γ
  - Add the production $\rightarrow \gamma$ 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 $\rightarrow \gamma$ to the entry (A, token) for each such token.

- Note: if there are two different productions for a given entry, the grammar is not LL(1)

Example

- First(T) = First(S)
- First(S) = First(E)
- First(S′) = { + }
- First(E) = { number, '(' }

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

$T \longmapsto S\$$
$S \longmapsto ES'$
$S' \longmapsto \varepsilon$
$S' \longmapsto + S$
$E \longmapsto \text{number} \mid (S)$

Note: we want the *least* solution to this system of set equations… a *fixpoint* computation. More on these later in the course.

	number	+	(	)	$ (EOF)
T	$\longmapsto S\$$		$\longmapsto S\$$		
S	$\longmapsto E\ S'$		$\longmapsto E\ S'$		
S′		$\longmapsto + S$		$\longmapsto \varepsilon$	$\longmapsto \varepsilon$
E	$\longmapsto$ num.		$\longmapsto (S)$		

Converting the table to code

- Define n mutually recursive functions
  - one for each nonterminal A: parse_A
  - The type of parse_A is `unit -> 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	$\longmapsto$ S$		$\longmapsto$S$		
S	$\longmapsto$ E S′		$\longmapsto$E S′		
S′		$\longmapsto$ + S		$\longmapsto \varepsilon$	$\longmapsto \varepsilon$
E	$\longmapsto$ num.		$\longmapsto$ (S)		

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

DEMO: PARSER.ML

LL(1) Summary

- Top-down parsing that finds the leftmost derivation.
- Language Grammar ⇒ LL(1) grammar ⇒ prediction table ⇒ recursive-descent parser

- 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!)

- Advantage:
  - Relatively easy to understand, write by hand.

- Is there a better way?

LR GRAMMARS

Bottom-up Parsing (LR Parsers)

- LR(k) parser:
  - Left-to-right scanning
  - Rightmost derivation
  - k lookahead symbols

- LR grammars are more expressive than LL
  - Can handle left-recursive (and right recursive) grammars; virtually all programming languages
  - Easier to express programming language syntax (no left factoring)

- Technique: "Shift-Reduce" parsers
  - Work bottom up instead of top down
  - Construct right-most derivation of a program in the grammar
  - Used by many parser generators (e.g. yacc, CUP, ocamlyacc, merlin, etc.)
  - Better error detection/recovery