Lecture 12

CS 131: COMPILERS

Announcements

- HW3: LLVM lite
  - Available on Blackboard.
  - Due: November 6th at 11:59:59pm

you should have

STARTED EARLY!!

- Midterm: November 19th
  - In class
  - One-page, letter-sized, double-sided “cheat sheet” of notes permitted
  - Coverage: interperters, x86, LLVMlite, lexing, parsing
  - See examples of previous exam on Blackboard

CONTEXT FREE GRAMMARS

Context-free Grammars

- Here is a specification of the language of balanced parens:

$$S \longmapsto (S)S$$

$$S \longmapsto \varepsilon$$

Note: Once again we have to take care to distinguish meta-language elements (e.g. "S" and "$\longmapsto$") from object-language elements (e.g. "(").*

- The definition is *recursive* – S mentions itself.
- Idea: "derive" a string in the language by starting with S and rewriting according to the rules:
  - Example:
    $S \longmapsto (S)S \longmapsto ((S)S)S \longmapsto ((\varepsilon)S)S \longmapsto ((\varepsilon)S)\varepsilon \longmapsto ((\varepsilon)\varepsilon)\varepsilon = (())$
- You can replace the *nonterminal* S by one of its definitions anywhere
- A context-free grammar accepts a string iff there is a derivation from the start symbol

* And, since we're writing this description in English, we are careful distinguish the meta-meta-language (e.g. words) from the meta-language and object-language (e.g. symbols) by using quotes.

CFGs Mathematically

- A Context-free Grammar (CFG) consists of
  - A set of *terminals* (e.g., a lexical 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:

$$S \longmapsto (S)S$$

$$S \longmapsto \varepsilon$$

- How many terminals? How many nonterminals? Productions?

Another Example: Sum Grammar

- A grammar that accepts parenthesized sums of numbers:

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

e.g.: (1 + 2 + (3 + 4)) + 5

- Note the vertical bar '|' is shorthand for multiple productions:

$$
\left.
\begin{array}{l}
S \longmapsto E + S \\
S \longmapsto E \\
E \longmapsto \text{number} \\
E \longmapsto (S)
\end{array}
\right\}
$$

4 productions

2 nonterminals: S, E

4 terminals: (,), +, number

Start symbol: S

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:

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

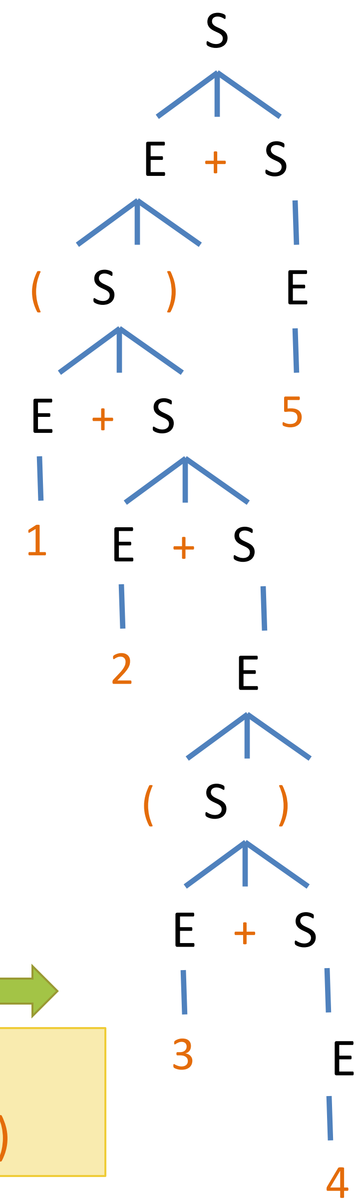
(*substitute* β for an occurrence of A)

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

Note: Underline indicates symbol being expanded.

From Derivations to Parse Trees

- Tree representation of the derivation
- Leaves of the tree are terminals
  - In-order traversal yields the input sequence of tokens
- Internal nodes: nonterminals
- No information about the *order* of the derivation steps



Parse Tree

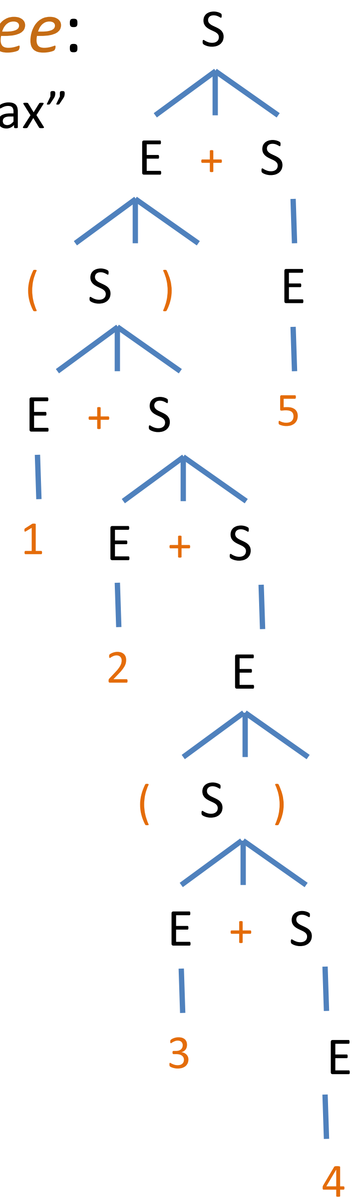
(1 + 2 + (3 + 4)) + 5

S ⟼ E + S | E
E ⟼ number | (S)

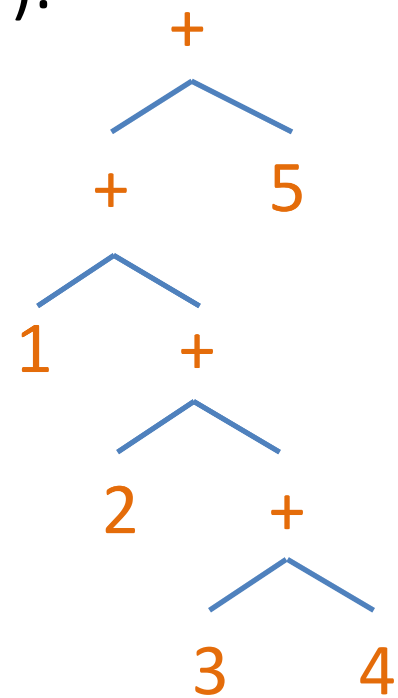
From Parse Trees to Abstract Syntax

- *Parse tree*:

"concrete syntax"



- *Abstract syntax tree* (AST):



- Hides, or *abstracts*, unneeded information.

Derivation Orders

- Productions of the grammar can be applied in any order.
- There are two standard orders:
  - *Leftmost derivation*: Find the left-most nonterminal and apply a production to it.
  - *Rightmost derivation*: Find the right-most nonterminal and apply a production there.

- Note that both strategies (and any other) yield the same parse tree!
  - Parse tree doesn't contain the information about what order the productions were applied.

Example: Left- and rightmost derivations

- Leftmost Derivation
- $\underline{\mathbf{S}} \longmapsto \underline{\mathbf{E}} + S$
  $\longmapsto (\underline{\mathbf{S}}) + S$
  $\longmapsto (\underline{\mathbf{E}} + S) + S$
  $\longmapsto (1 + \underline{\mathbf{S}}) + S$
  $\longmapsto (1 + \underline{\mathbf{E}} + S) + S$
  $\longmapsto (1 + 2 + \underline{\mathbf{S}}) + S$
  $\longmapsto (1 + 2 + \underline{\mathbf{E}}) + S$
  $\longmapsto (1 + 2 + (\underline{\mathbf{S}})) + S$
  $\longmapsto (1 + 2 + (\underline{\mathbf{E}} + S)) + S$
  $\longmapsto (1 + 2 + (3 + \underline{\mathbf{S}})) + S$
  $\longmapsto (1 + 2 + (3 + \underline{\mathbf{E}})) + S$
  $\longmapsto (1 + 2 + (3 + 4)) + \underline{\mathbf{S}}$
  $\longmapsto (1 + 2 + (3 + 4)) + \underline{\mathbf{E}}$
  $\longmapsto (1 + 2 + (3 + 4)) + 5$

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

Loops and Termination

- Some care is needed when defining CFGs
- Consider:

$$S \longmapsto E$$
$$E \longmapsto S$$

  - This grammar has nonterminal definitions that are “nonproductive”. (i.e. they don’t mention any terminal symbols)
  - There is no finite derivation starting from S, so the language is empty.

- Consider:

$$S \longmapsto (\ S\ )$$

  - This grammar is productive, but again there is no finite derivation starting from S, so the language is empty

- It is easy to generalize these examples to a “chain” of many nonterminals, which can be harder to find in a large grammar

- Upshot: be aware of “vacuously empty” CFG grammars.
  - Every nonterminal should eventually rewrite to an alternative that contains only terminal symbols.

Associativity, ambiguity, and precedence.

GRAMMARS FOR PROGRAMMING LANGUAGES

Associativity

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

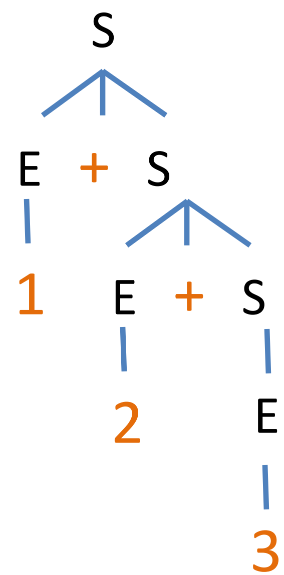
Consider the input: $1 + 2 + 3$

Leftmost derivation:

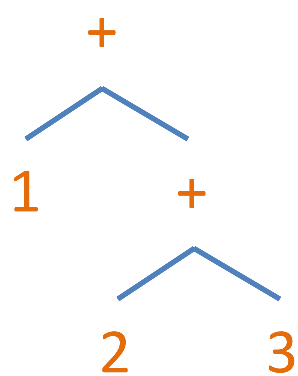
$\underline{\mathbf{S}} \longmapsto \underline{\mathbf{E}} + S$
$\longmapsto 1 + \underline{\mathbf{S}}$
$\longmapsto 1 + \underline{\mathbf{E}} + S$
$\longmapsto 1 + 2 + \underline{\mathbf{S}}$
$\longmapsto 1 + 2 + \underline{\mathbf{E}}$
$\longmapsto 1 + 2 + 3$

Rightmost derivation:

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



Parse Tree



AST

Associativity

- This grammar makes '+' *right associative*…
  – i.e., the abstract syntax tree is the same for both
    1 + 2 + 3 and 1 + (2 + 3)
- Note that the grammar is *right recursive*…

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

S refers to itself on the right of +

- How would you make '+' left associative?
- What are the trees for "1 + 2 + 3"?

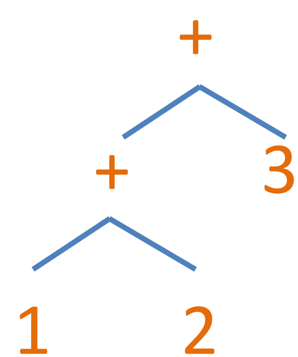
Ambiguity

- Consider this grammar:

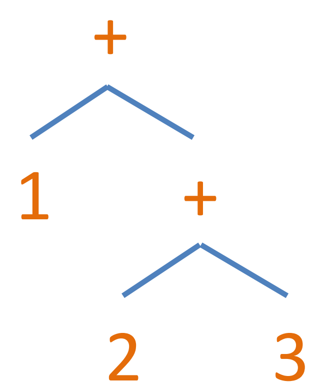
$$S \longmapsto S + S \ | \ (\ S \ ) \ | \ \text{number}$$

- Claim: it accepts the *same* set of strings as the previous one.
- What's the difference?
- Consider these *two* leftmost derivations:
  - $\underline{\mathbf{S}} \longmapsto \underline{\mathbf{S}} + S \longmapsto 1 + \underline{\mathbf{S}} \longmapsto 1 + \underline{\mathbf{S}} + S \longmapsto 1 + 2 + \underline{\mathbf{S}} \longmapsto 1 + 2 + 3$
  - $\underline{\mathbf{S}} \longmapsto \underline{\mathbf{S}} + S \longmapsto \underline{\mathbf{S}} + S + S \longmapsto 1 + \underline{\mathbf{S}} + S \longmapsto 1 + 2 + \underline{\mathbf{S}} \longmapsto 1 + 2 + 3$

- One derivation gives left associativity, the other gives right associativity to '+'
  - Which is which?



AST 1



AST 2

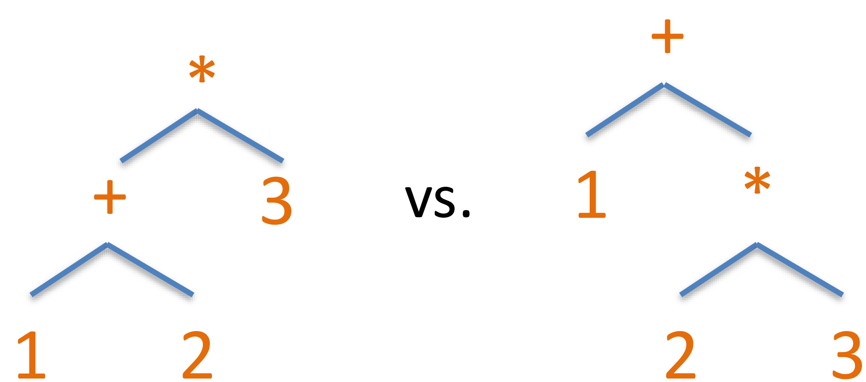
Why do we care about ambiguity?

- The '+' operation is associative, so it doesn't matter which tree we pick. Mathematically, $x + (y + z) = (x + y) + z$
  - But, some binary operations aren't associative. Examples?
  - Some operations are only left (or right) associative. Examples?

- Moreover, if there are multiple operations, ambiguity in the grammar leads to ambiguity in their *precedence*
- Consider:

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

- Input: $1 + 2 * 3$
  - One parse = $(1 + 2) * 3 = 9$
  - The other = $1 + (2 * 3) = 7$



Eliminating Ambiguity

- We can often eliminate ambiguity by adding nonterminals and allowing recursion only on the left (or right) .
- Higher-precedence operators go *farther* from the start symbol.
- Example:

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

- To disambiguate:
  - Decide (following math) to make '*' higher precedence than '+'
  - Make '+' left associative
  - Make '*' right associative
- Note:
  - S_2 corresponds to 'atomic' expressions

$$\begin{aligned} S_0 &\longmapsto S_0 + S_1 \;|\; S_1 \\ S_1 &\longmapsto S_2 * S_1 \;|\; S_2 \\ S_2 &\longmapsto \text{number} \;|\; (S_0) \end{aligned}$$

Context Free Grammars: Summary

- Context-free grammars allow concise specifications of programming languages.
  - An unambiguous CFG specifies how to parse: convert a token stream to a (parse tree)
  - Ambiguity can (often) be removed by encoding precedence and associativity in the grammar.
- Even with an unambiguous CFG, there may be more than one derivation
  - Though all derivations correspond to the same abstract syntax tree.
- Still to come: finding a derivation
  - But first: menhir

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:

$$S \longmapsto (S)S$$

$$S \longmapsto \varepsilon$$

- How many terminals? How many nonterminals? Productions?

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
S	(	(1 + 2 + (3 + 4)) + 5
$\longmapsto$ **E** + S	(	(1 + 2 + (3 + 4)) + 5
$\longmapsto$ (**S**) + S	1	(1 + 2 + (3 + 4)) + 5
$\longmapsto$ (**E** + S) + S	1	(1 + 2 + (3 + 4)) + 5
$\longmapsto$ (1 + **S**) + S	2	(1 + 2 + (3 + 4)) + 5
$\longmapsto$ (1 + **E** + S) + S	2	(1 + 2 + (3 + 4)) + 5
$\longmapsto$ (1 + 2 + **S**) + S	(	(1 + 2 + (3 + 4)) + 5
$\longmapsto$ (1 + 2 + **E**) + S	(	(1 + 2 + (3 + 4)) + 5
$\longmapsto$ (1 + 2 + (**S**)) + S	3	(1 + 2 + (3 + 4)) + 5
$\longmapsto$ (1 + 2 + (**E** + S)) + S	3	(1 + 2 + (3 + 4)) + 5
$\longmapsto$ …		

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.

LL(1) GRAMMARS

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:

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

- 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.

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

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
$\longmapsto (1\ \underline{\mathbf{S'}})\ S'$	+	(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
$\longmapsto (1 + 2 + \underline{\mathbf{S}})\ S'$	(	(1 + 2 + (3 + 4)) + 5
$\longmapsto (1 + 2 + \underline{\mathbf{E}}\ S')\ S'$	(	(1 + 2 + (3 + 4)) + 5
$\longmapsto (1 + 2 + (\underline{\mathbf{S}})S')\ S'$	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, '(' }

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

- Follow(S') = Follow(S)
- Follow(S) = { \$, ')' } ∪ Follow(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!)
- Is there a better way?