Lecture 8

CIS 341: COMPILERS

Announcements

- Project 2: Parsing and Compiling Expressions
  - Due: Tuesday, Feb 12th at 11:59:59pm

LR PARSING CONTINUED

Simple LR parsing with no look ahead.

LR(0) GRAMMARS

LR Parser States

- Goal: know what set of reductions are legal at any given point.
- Idea: Summarize all possible stack prefixes α as a finite parser state.
  - Parser state is computed by a DFA that reads the stack σ.
  - Accept states of the DFA correspond to unique reductions that apply.

- Example: LR(0) parsing
  - **L**eft-to-right scanning, **R**ight-most derivation, **zero** look-ahead tokens
  - Too weak to handle many language grammars (e.g. the "sum" grammar)
  - But, helpful for understanding how the shift-reduce parser works.

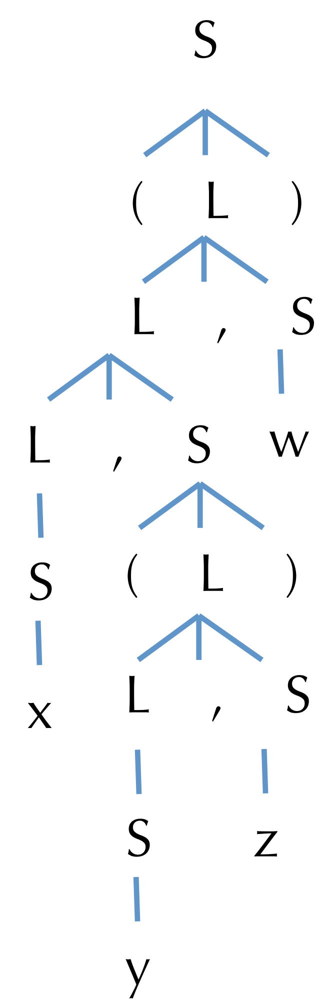
Example LR(0) Grammar: Tuples

- Example grammar for non-empty tuples and identifiers:

$$S \longmapsto (\ L\ )\ \mid\ id$$
$$L \longmapsto S\ \mid\ L\ ,\ S$$

- Example strings:
  - x
  - (x,y)
  - ((((x))))
  - (x, (y, z), w)
  - (x, (y, (z, w)))

Parse tree for:
(x, (y, z), w)



Shift/Reduce Parsing

$S \longmapsto (L) \mid id$
$L \longmapsto S \mid L , S$

- Parser state:
  - Stack of terminals and nonterminals.
  - Unconsumed input is a string of terminals
  - Current derivation step is stack + input
- Parsing is a sequence of *shift* and *reduce* operations:
- Shift: move look-ahead token to the stack: e.g.

Stack	Input	Action
	(x, (y, z), w)	shift (
(	x, (y, z), w)	shift x

- Reduce: Replace symbols γ at top of stack with nonterminal X such that $X \longmapsto \gamma$ is a production. (pop γ, push X): e.g.

Stack	Input	Action
(x	, (y, z), w)	reduce $S \longmapsto id$
(S	, (y, z), w)	reduce $L \longmapsto S$

Example Run

$S \longmapsto (\ L\ )\ \mid\ id$

$L \longmapsto S\ \mid\ L\ ,\ S$

Stack	Input	Action
	(x, (y, z), w)	shift (
(	x, (y, z), w)	shift x
(x	, (y, z), w)	reduce S $\longmapsto$ id
(S	, (y, z), w)	reduce L $\longmapsto$ S
(L	, (y, z), w)	shift ,
(L,	(y, z), w)	shift (
(L, (	y, z), w)	shift y
(L, (y	, z), w)	reduce S $\longmapsto$ id
(L, (S	, z), w)	reduce L $\longmapsto$ S
(L, (L	, z), w)	shift ,
(L, (L,	z), w)	shift z
(L, (L, z	), w)	reduce S $\longmapsto$ id
(L, (L, S	), w)	reduce L $\longmapsto$ L, S
(L, (L	), w)	shift)
(L, (L)	, w)	reduce S $\longmapsto$ (L)
(L, S	, w)	reduce L $\longmapsto$ L, S
(L	, w)	shift ,
(L,	w)	shift w
(L, w	)	reduce S $\longmapsto$ id

Action Selection Problem

- Given a stack σ and a look-ahead symbol b, should the parser:
  - Shift b onto the stack (new stack is σb)
  - Reduce a production $X \longmapsto \gamma$, assuming that $\sigma = \alpha\gamma$ (new stack is αX)?

- Sometimes the parser can reduce but shouldn't
  - For example, $X \longmapsto \varepsilon$ can *always* be reduced
- Sometimes the stack can be reduced in different ways

- Main idea: decide what to do based on a *prefix* α of the stack plus the look-ahead symbol.
  - The prefix a is different for different possible reductions since in productions $X \longmapsto g$ and $Y \longmapsto b$, g and b might have different lengths.

- Main goal: know what set of reductions are legal at any point.
  - How do we keep track?

LR(0) States

- An LR(0) *state* is a *set* of *items* keeping track of progress on possible upcoming reductions.
- An LR(0) *item* is a production from the language with an extra separator "." somewhere in the right-hand-side

$$S \longmapsto (\ L\ )\ \ |\ \ id$$
$$L \longmapsto S\ \ |\ \ L\ ,\ S$$

- Example items: $S \longmapsto .(\ L\ )$ or $S \longmapsto (.\ L)$ or $L \longmapsto S.$
- Intuition:
  - Stuff before the '.' is already on the stack (beginnings of possible γ's to be reduced)
  - Stuff after the '.' is what might be seen next
  - The prefixes α are represented by the state itself

Constructing the DFA: Start state & Closure

- First step: Add a new production $S' \longmapsto S\$$ to the grammar
- Start state of the DFA = empty stack, so it contains the item:
  $S' \longmapsto .S\$$

$$S' \longmapsto S\$$$
$$S \longmapsto (\ L\ )\ |\ \text{id}$$
$$L \longmapsto S\ \ |\ \ L\ ,\ S$$

- Closure of a state:
  - Adds items for all productions whose LHS nonterminal occurs in an item in the state just after the '.'
  - The added items have the '.' located at the beginning (no symbols for those items have been added to the stack yet)
  - Note that newly added items may cause yet more items to be added to the state… keep iterating until a *fixed point* is reached.
- Example: CLOSURE($\{S' \longmapsto .S\$\}$) = $\{S' \longmapsto .S\$,\ S \longmapsto .(L),\ S \longmapsto .\text{id}\}$

- Resulting "closed state" contains the set of all possible productions that might be reduced next.

Example: Constructing the DFA

$S' \longmapsto .S\$$

$S' \longmapsto S\$$
$S \longmapsto (\ L\ )\ \mid\ id$
$L \longmapsto S\ \mid\ L\ ,\ S$

- First, we construct a state with the initial item $S' \longmapsto .S\$$

Example: Constructing the DFA

$S' \longmapsto .S\$$
$S \longmapsto .(L)$
$S \longmapsto .id$

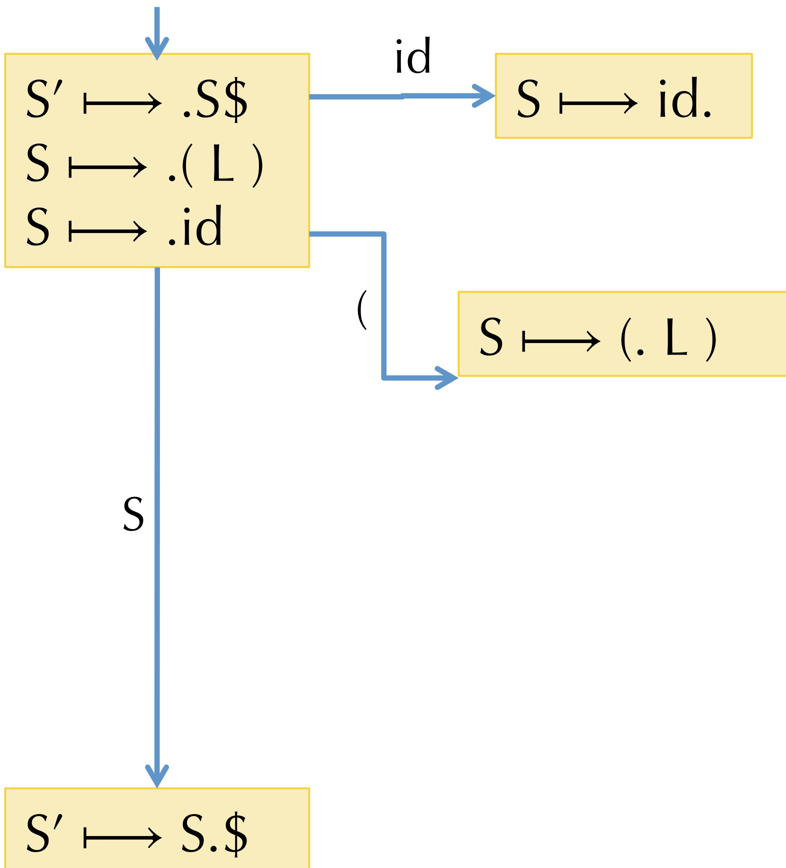
$S' \longmapsto S\$$
$S \longmapsto (L) \mid id$
$L \longmapsto S \mid L , S$

- Next, we take the closure of that state:
  $\text{CLOSURE}(\{S' \longmapsto .S\$\}) = \{S' \longmapsto .S\$, S \longmapsto .(L), S \longmapsto .id\}$

- In the set of items, the nonterminal S appears after the '.'
- So we add items for each S production in the grammar

Example: Constructing the DFA

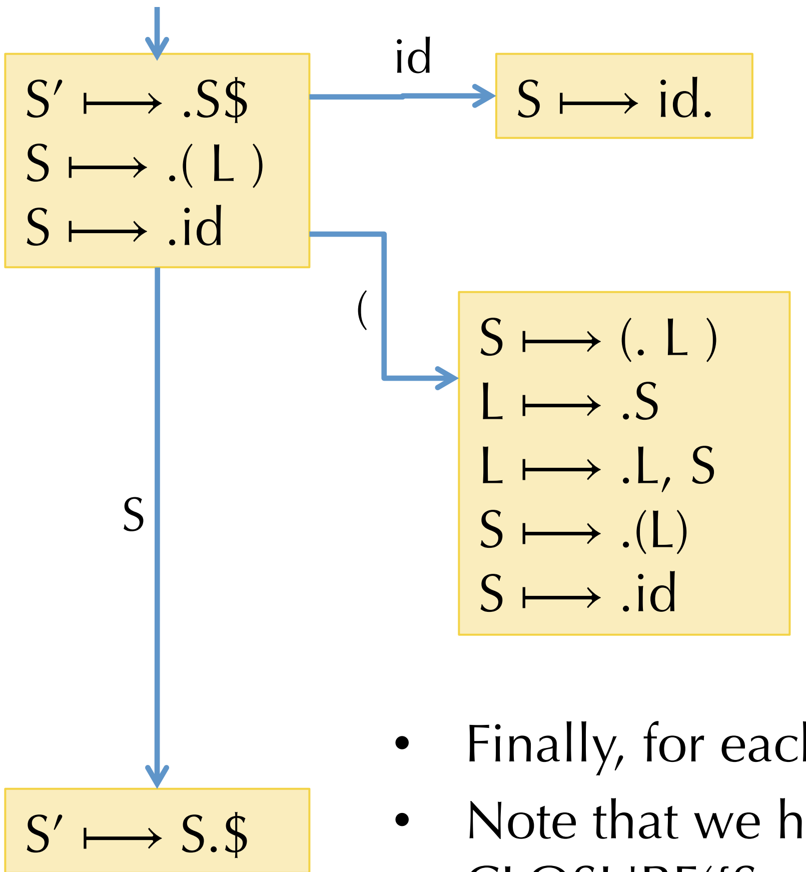
$S' \longmapsto S\$$
$S \longmapsto (\ L\ )\ |\ id$
$L \longmapsto S\ |\ L\ ,\ S$



- Next we add the transitions:
- First, we see what terminals and nonterminals can appear after the '.' in the source state.
  - Outgoing edges have those label.
- The target state (initially) includes all items from the source state that have the edge-label symbol after the '.', but we advance the '.' (to simulate shifting the item onto the stack)

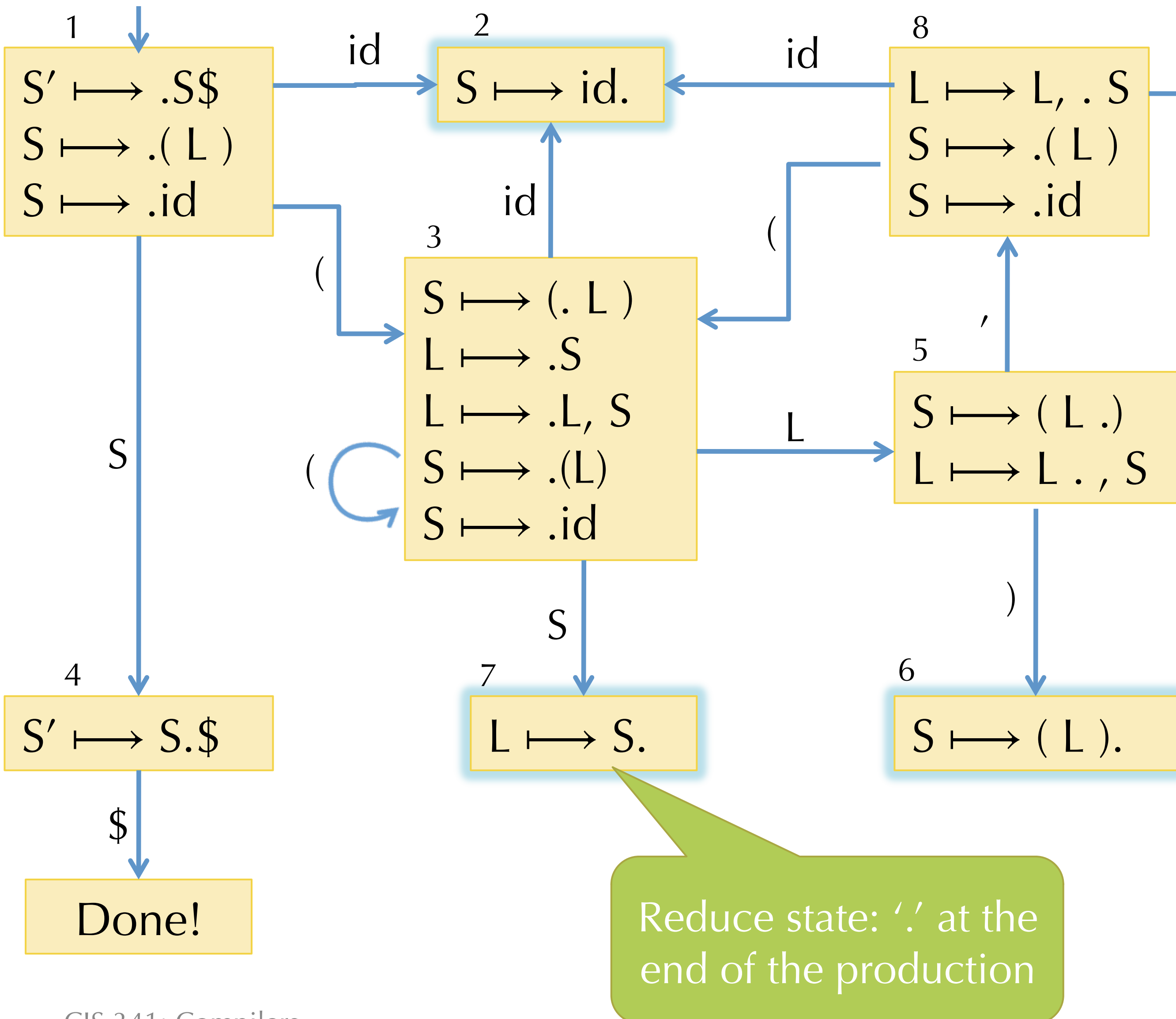
Example: Constructing the DFA

$$S' \longmapsto S\$$$
$$S \longmapsto (\ L\ )\ \mid\ id$$
$$L \longmapsto S\ \mid\ L\ ,\ S$$



- Finally, for each new state, we take the closure.
- Note that we have to perform two iterations to compute CLOSURE({S ⟼ (. L)})
  - First iteration adds L ⟼ .S and L ⟼ .L, S
  - Second iteration adds S ⟼ .(L) and S ⟼ .id

Full DFA for the Example



- Current state: run the DFA on the stack.
- If a reduce state is reached, reduce
- Otherwise, if the next token matches an outgoing edge, shift.
- If no such transition, it is a parse error.

Using the DFA

- Run the parser stack through the DFA.
- The resulting state tells us which productions might be reduced next.
  - If not in a reduce state, then shift the next symbol and transition according to DFA.
  - If in a reduce state, $X \longmapsto \gamma$ with stack $\alpha\gamma$, pop γ and push X.

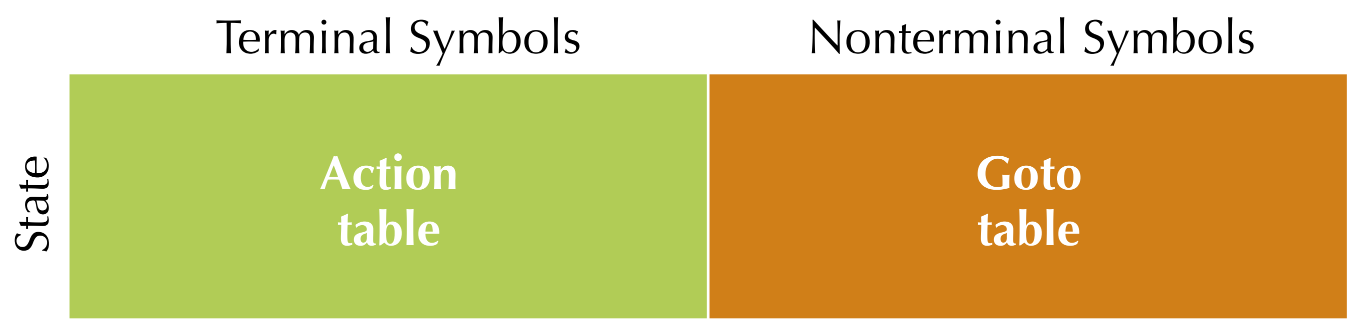
- Optimization: No need to re-run the DFA from beginning every step
  - Store the state with each symbol on the stack: e.g. $_1(_3(_3L_5)_6$
  - On a reduction $X \longmapsto \gamma$, pop stack to reveal the state too:
    e.g. From stack $_1(_3(_3L_5)_6$ reduce $S \longmapsto (L)$ to reach stack $_1(_3$
  - Next, push the reduction symbol: e.g. to reach stack $_1(_3S$
  - Then take just one step in the DFA to find next state: $_1(_3S_7$

Implementing the Parsing Table

Represent the DFA as a table of shape:

state * (terminals + nonterminals)

- Entries for the "action table" specify two kinds of actions:
  - Shift and goto state n
  - Reduce using reduction $X \longmapsto \gamma$
    - First pop γ off the stack to reveal the state
    - Look up X in the "goto table" and goto that state



Example Parse Table

	(	)	id	,	$	S	L
1	s3		s2			g4	
2	$S \longmapsto id$	$S \longmapsto id$	$S \longmapsto id$	$S \longmapsto id$	$S \longmapsto id$		
3	s3		s2			g7	g5
4					DONE		
5		s6		s8			
6	$S \longmapsto (L)$	$S \longmapsto (L)$	$S \longmapsto (L)$	$S \longmapsto (L)$	$S \longmapsto (L)$		
7	$L \longmapsto S$	$L \longmapsto S$	$L \longmapsto S$	$L \longmapsto S$	$L \longmapsto S$		
8	s3		s2			g9	
9	$L \longmapsto L,S$	$L \longmapsto L,S$	$L \longmapsto L,S$	$L \longmapsto L,S$	$L \longmapsto L,S$		

sx = shift and goto state x
gx = goto state x

Example

- Parse the token stream: (x, (y, z), w)$

Stack	Stream	Action (according to table)
ε_1	(x, (y, z), w)$	s3
$\varepsilon_1(_3$	x, (y, z), w)$	s2
$\varepsilon_1(_3x_2$	, (y, z), w)$	Reduce: S⟼id
$\varepsilon_1(_3S$	, (y, z), w)$	g7 (from state 3 follow S)
$\varepsilon_1(_3S_7$	, (y, z), w)$	Reduce: L⟼S
$\varepsilon_1(_3L$	, (y, z), w)$	g5 (from state 3 follow L)
$\varepsilon_1(_3L_5$	, (y, z), w)$	s8
$\varepsilon_1(_3L_5,_8$	(y, z), w)$	s3
$\varepsilon_1(_3L_5,_8(_3$	y, z), w)$	s2

LR(0) Limitations

- An LR(0) machine only works if states with reduce actions have a *single* reduce action.
  - In such states, the machine *always* reduces (ignoring lookahead)
- With more complex grammars, the DFA construction will yield states with shift/reduce and reduce/reduce conflicts:

OK	shift/reduce	reduce/reduce
S ⟼ (L).	S ⟼ (L). L ⟼ .L , S	S ⟼ L ,S. S ⟼ ,S.

- Such conflicts can often be resolved by using a look-ahead symbol: LR(1)

Examples

- Consider the left associative and right associative "sum" grammars:

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

- One is LR(0) the other isn't… which is which and why?
- What kind of conflict do you get? Shift/reduce or Reduce/reduce?

- Ambiguities in associativity/precedence usually lead to shift/reduce conflicts.

LR(1) Parsing

- Algorithm is similar to LR(0) DFA construction:
  - LR(1) state = set of LR(1) items
  - An LR(1) item is an LR(0) item + a set of look-ahead symbols:
    $A \longmapsto \alpha.\beta \ , \ \mathcal{L}$
  - Invariant: no two LR(1) items in the same state have the same LR(0) item. (Just merge the look-ahead sets)

- LR(1) closure is a little more complex:
- Form the set of items just as for LR(0) algorithm.
- Whenever a new item $C \longmapsto .\gamma$ is added because $A \longmapsto \beta.C\delta \ , \ \mathcal{L}$ is already in the set, we need to compute its look-ahead set $\mathcal{M}$:
  1. The look-ahead set $\mathcal{M}$ includes FIRST(δ) (the set of terminals that may start strings derived from δ)
  2. If δ can derive ε (it is nullable), then the look-ahead $\mathcal{M}$ also contains $\mathcal{L}$

Example Closure

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

- Start item: $S' \longmapsto .S \quad , \quad \{\$\}$ Note: $ is treated as initial lookahead
- Since S is to the right of a '.', add:

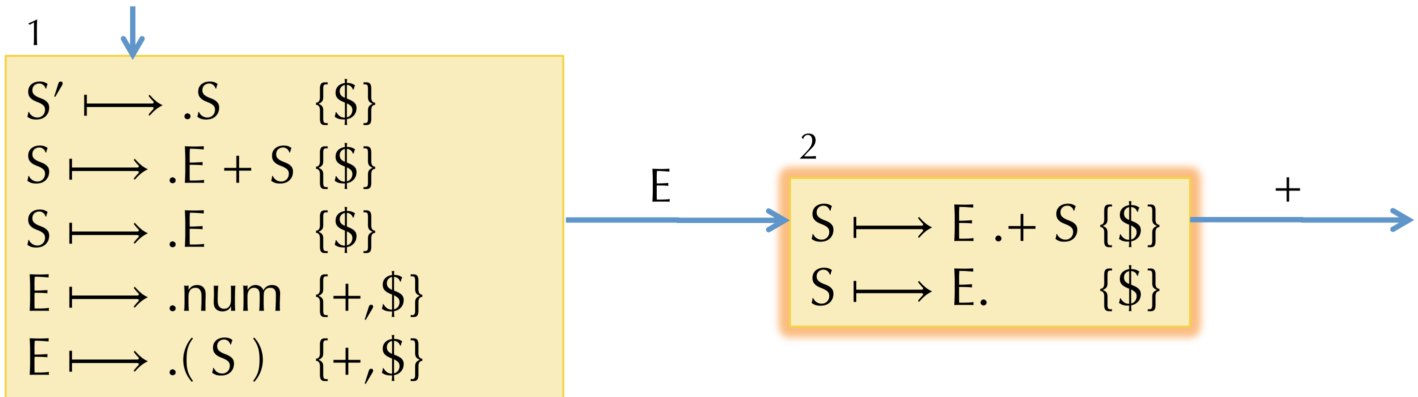
  $S \longmapsto .E + S \quad , \quad \{\$\}$ Note: $ added for reason 2

  $S \longmapsto .E \quad , \quad \{\$\}$
- Need to keep closing, since E appears to the right of a '.':

  $E \longmapsto .\text{number} \quad , \quad \{+,\$\}$ Note: + added for reason 1

  $E \longmapsto .(S) \quad , \quad \{+,\$\}$ Note: $ added for reason 2
- All items are distinct, so we're done

Using the DFA



- The behavior is determined if:
  - There is no overlap among the look-ahead sets for each reduce item, and
  - None of the look-ahead symbols appear to the right of a '.'

	+	$	E
1			g2
2	s3	S ⟼ E	

Fragment of the Action & Goto tables

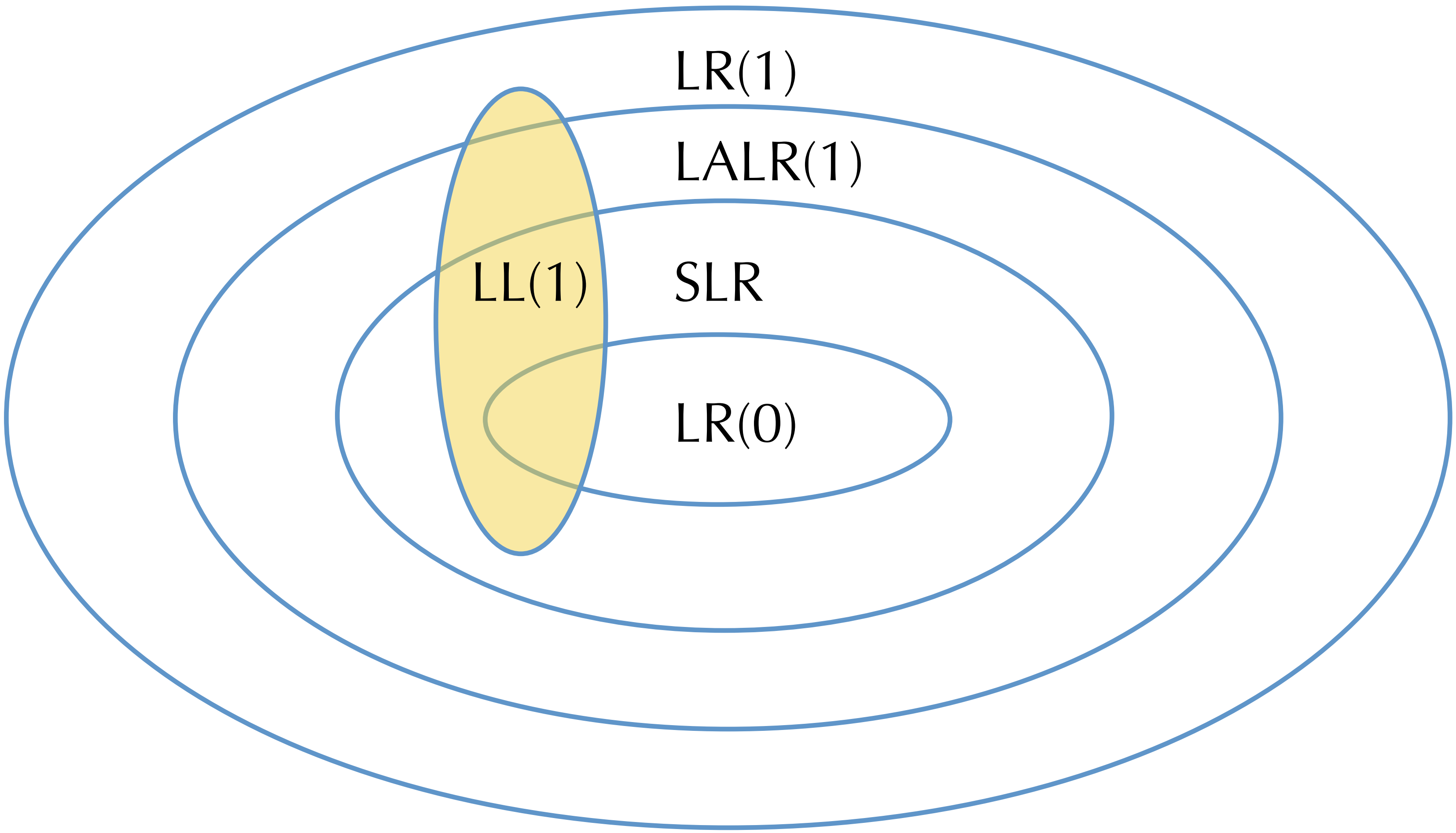
LR variants

- LR(1) gives maximal power out of a 1 look-ahead symbol parsing table
  - DFA + stack is a push-down automaton (recall 262)

- In practice, LR(1) tables are too big.
- LALR(1) = "Look-ahead LR"
  - Merge any two LR(1) states whose items are identical except for the look-ahead sets
  - Such merging can lead to nondeterminism (e.g. reduce/reduce conflicts), but
  - Results in a much smaller parse table and works well in practice
  - This is the usual technology for automatic parser generators

- GLR = "Generalized LR" parsing
  - Efficiently compute the set of all parses for a given input
  - Later passes should disambiguate based on other context

Classification of Grammars



Debugging parser conflicts.
Disambiguating grammars.

OCAMLYACC IN PRACTICE

ocamlyacc output

- You can get verbose ocamlyacc debugging information by doing:
  - `ocamlyacc -v`
  - or, if using ocamlbuild:
    `ocamlbuild -yaccflag –v`
- The result is a parser.output file that contains a pretty-printed version of the DFA.
  - Parser conflicts are marked inline.
  - The parser items of each state use the '.' just as described above
- Example: see parser_ambig.mly

Precedence and Associativity Declarations

- Parser generators, like ocamlyacc often support precedence and associativity declarations.
  - Hints to the parser about how to resolve conflicts.
  - See: parser_hint.mly
- Pros:
  - Avoids having to manually resolve those ambiguities by introducing extra nonterminals (as seen in parser.mly)
  - Easier to maintain the grammar
- Cons:
  - Can't re-use the same terminal in multiple ways
  - Introduces another level of debugging
- Limits:
  - Not always easy to disambiguate the grammar based on just precedence and associativity.

Example Ambiguity in Real Languages

- Consider this grammar:
  $S \longmapsto$ `if (E) S`
  $S \longmapsto$ `if (E)` S `else` S
  $S \longmapsto X = E$
  $E \longmapsto \ldots$

- Is this grammar OK?

- Consider how to parse:

  `if (`E_1`) if (`E_2`)` S_1
  `else` S_2

- This is known as the "dangling else" problem.
- What should the "right answer" be?

- How do we change the grammar?

How to Disambiguate if-then-else

- Want to rule out:

$$\texttt{if } (E_1) \underbrace{\texttt{if } (E_2)\ S_1}\ \texttt{else } S_2$$

- Observation: An un-matched '`if`' should not appear as the '`then`' clause of a containing '`if`'.

$S \longmapsto M \mid U$ // M = "matched", U = "unmatched"
$U \longmapsto$ `if` $(E)\ S$ // Unmatched 'if'
$U \longmapsto$ `if` $(E)\ M$ `else` U // Nested if is matched
$M \longmapsto$ `if` $(E)\ M$ `else` M // Matched 'if'
$M \longmapsto X = E$ // Other statements

- See: ifthen.mly and ifthen_ok.mly