

Lecture 13

# CS 131: COMPILERS

# Announcements

- Midterm: November 19<sup>th</sup>
  - In class
  - One-page, letter-sized, double-sided “cheat sheet” of notes permitted
  - Coverage: interpreters, x86, LLVMlite, lexing, parsing
  - See examples of previous exam on Blackboard
- HW3: Due tomorrow.
- HW4: Compiling Oat v.1
  - releasing soon

	number	+	(	)	\$ (EOF)
T	$\mapsto S\$$		$\mapsto S\$$		
S	$\mapsto E S'$		$\mapsto E S'$		
S'		$\mapsto + S$		$\mapsto \epsilon$	$\mapsto \epsilon$
E	$\mapsto \text{num.}$		$\mapsto ( S )$		

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

# DEMO: HANDWRITTEN.ML

# LL(1) Summary

- Top-down parsing that finds the leftmost derivation.
- Language Grammar  $\Rightarrow$  LL(1) grammar  $\Rightarrow$  prediction table  $\Rightarrow$  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?



# 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

# Shift/Reduce Parsing

- 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.

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

$$L \mapsto S \mid L , S$$

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

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

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

# 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

$$\begin{array}{l} S \mapsto ( L ) \mid id \\ L \mapsto S \mid L , S \end{array}$$

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



# Constructing the DFA: Start state & Closure

- First step: Add a new production  $S' \mapsto S\$$  to the grammar
- Start state of the DFA = empty stack, so it contains the item:  
 $S' \mapsto .S\$$
- 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:  $\text{CLOSURE}(\{S' \mapsto .S\$\}) = \{S' \mapsto .S\$, S \mapsto .(L), S \mapsto .id\}$
- Resulting “closed state” contains the set of all possible productions that might be reduced next.

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

# Example: Constructing the DFA

$S' \mapsto .S\$$

$S' \mapsto S\$$

$S \mapsto ( L ) \mid id$

$L \mapsto S \mid L , S$

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

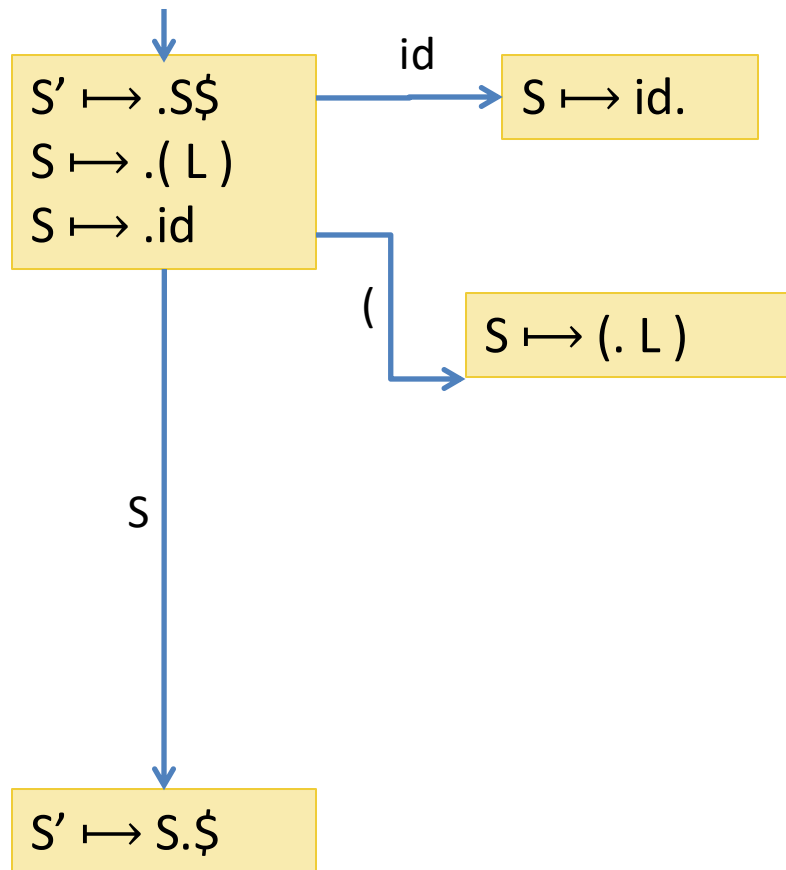
# Example: Constructing the DFA

↓  
 $S' \mapsto .S\$$   
 $S \mapsto .( L )$   
 $S \mapsto .id$

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

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



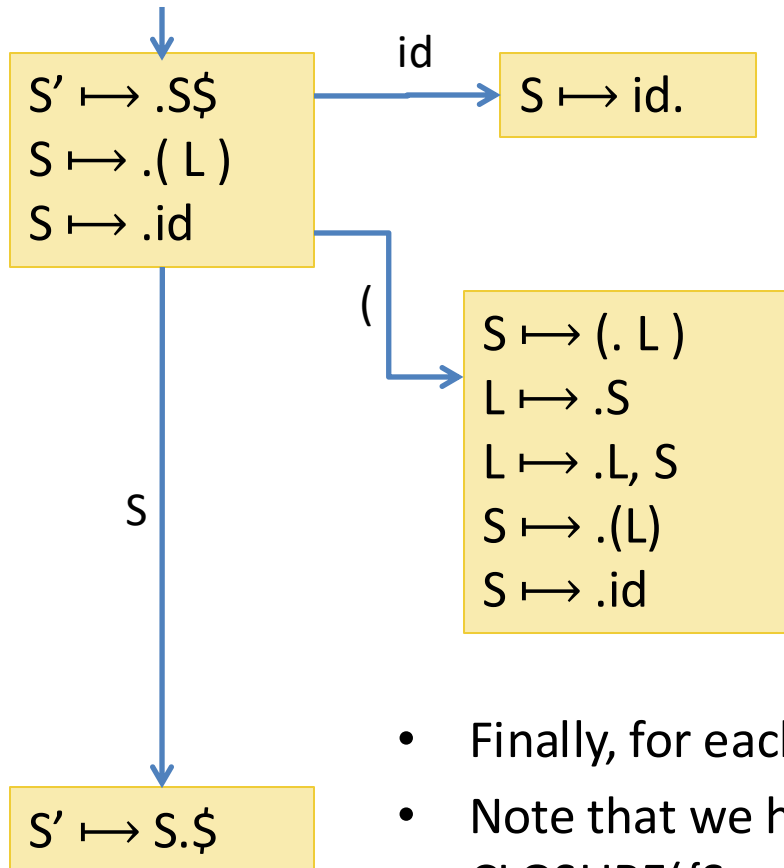
$S' \mapsto S\$$

$S \mapsto (L) \mid id$

$L \mapsto S \mid L, S$

- 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)

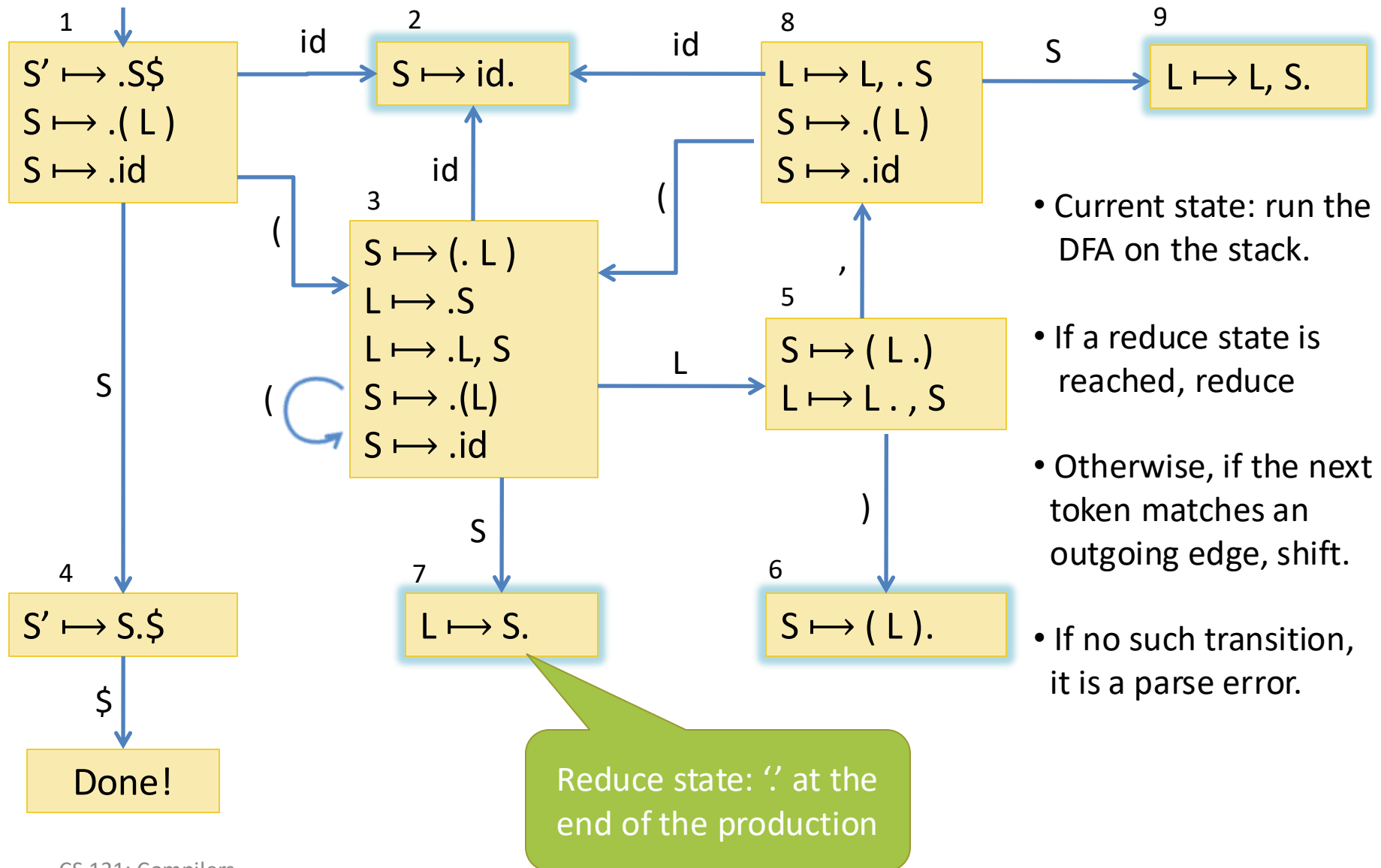
# Example: Constructing the DFA



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

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

# 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

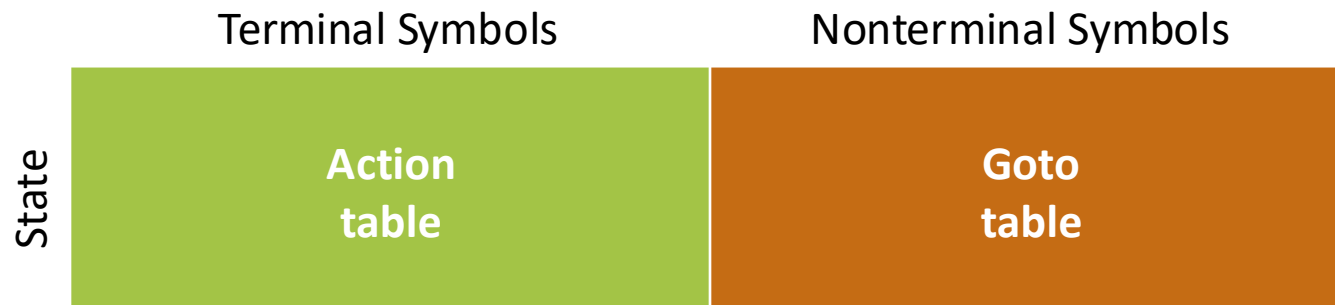
- Run the parser stack through the DFA.
- The resulting state tells us which productions might be reduced next.
  - If not in a reduce state, then shift the next symbol and transition according to DFA.
  - If in a reduce state,  $X \mapsto \gamma$  with stack  $\alpha\gamma$ , pop  $\gamma$  and push  $X$ .
- 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  $X \mapsto \gamma$ , pop stack to reveal the state too:  
e.g. From stack  $_1(_3(_3L_5)_6$  reduce  $S \mapsto ( L )$  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  $X \mapsto \gamma$ 
    - First pop  $\gamma$  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	$S \mapsto id$	$S \mapsto id$	$S \mapsto id$	$S \mapsto id$	$S \mapsto id$		
3	s3		s2			g7	g5
4					DONE		
5		s6		s8			
6	$S \mapsto (L)$	$S \mapsto (L)$	$S \mapsto (L)$	$S \mapsto (L)$	$S \mapsto (L)$		
7	$L \mapsto S$	$L \mapsto S$	$L \mapsto S$	$L \mapsto S$	$L \mapsto S$		
8	s3		s2			g9	
9	$L \mapsto L,S$	$L \mapsto L,S$	$L \mapsto L,S$	$L \mapsto L,S$	$L \mapsto L,S$		

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)
$\epsilon_1$	$(x, (y, z), w)\$$ s3	
$\epsilon_1(3$	$x, (y, z), w)\$$	s2
$\epsilon_1(3x_2$	$, (y, z), w)\$$	Reduce: $S \mapsto id$
$\epsilon_1(3S$	$, (y, z), w)\$$	g7 (from state 3 follow S)
$\epsilon_1(3S_7$	$, (y, z), w)\$$	Reduce: $L \mapsto S$
$\epsilon_1(3L$	$, (y, z), w)\$$	g5 (from state 3 follow L)
$\epsilon_1(3L_5$	$, (y, z), w)\$$	s8
$\epsilon_1(3L_{5,8}$	$(y, z), w)\$$	s3
$\epsilon_1(3L_{5,8}(3$	$y, z), w)\$$ 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

$S \mapsto ( L ).$

shift/reduce

$S \mapsto ( L ).$   
 $L \mapsto .L , S$

reduce/reduce

$S \mapsto L , S.$   
 $S \mapsto , S.$

- 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

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

right

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

- 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.

# SLR(1): “simple” LR(1) Parsers

- What conflicts are there in LR(0) parsing?
  - reduce/reduce conflict: an LR(0) state has two reduce actions
  - shift/reduce conflict: an LR(0) state mixes reduce and shift actions
- Can we use lookahead to disambiguate?
- SLR(1) – uses the same DFA construction as LR(0)
  - modifies the actions based on lookahead
- Suppose reducing nonterminal A is possible in some state:
  - compute Follow(A) for the given grammar
  - if the lookahead symbol is in Follow(A), then reduce, otherwise shift
  - can disambiguate between reduce/reduce conflicts if the follow sets are disjoint

Note: easiest LR variant to construct “by hand”.

# LR(1) Parsing

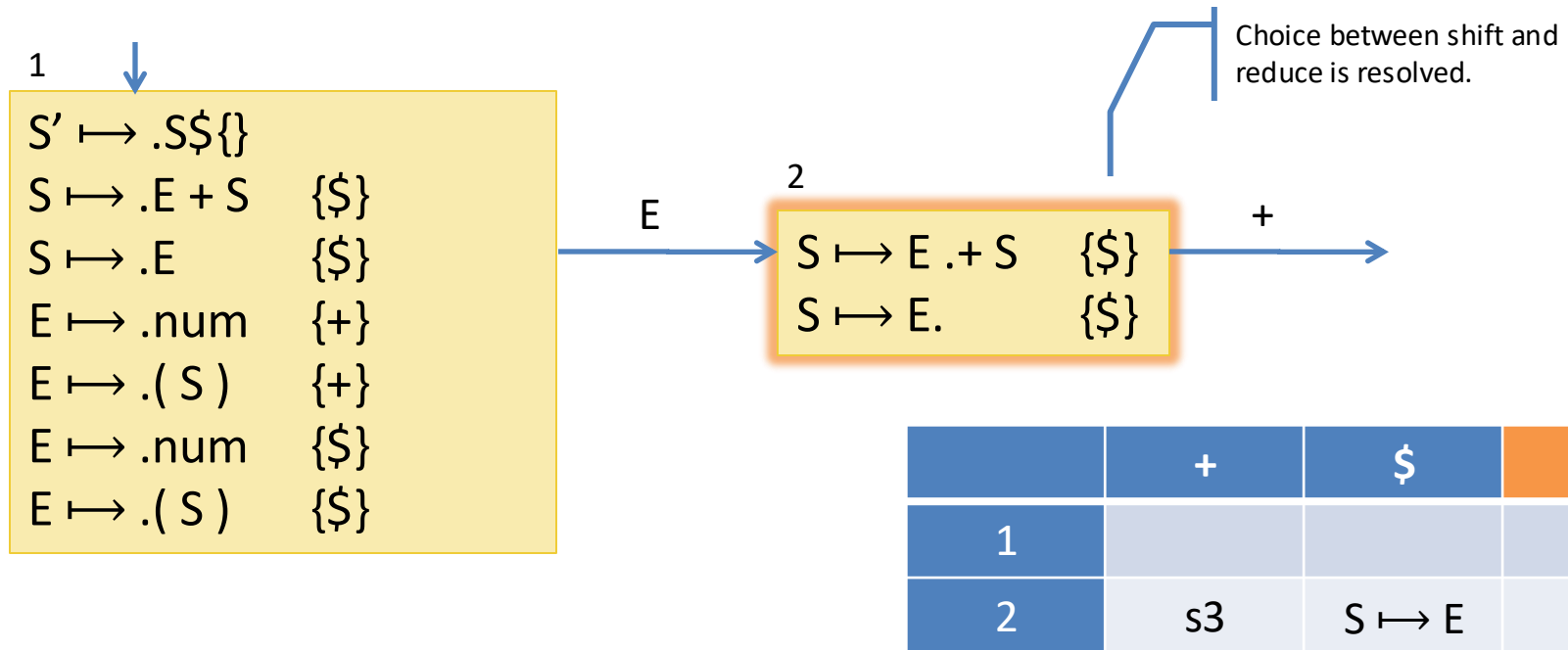
- SLR parsing is a simple refinement of LR(0). We can do more.
- Algorithm is similar to LR(0) DFA construction:
  - LR(1) state = set of LR(1) items
  - An LR(1) item is an LR(0) item + a set of look-ahead symbols  $\mathcal{L}$  :  
 $A \mapsto \alpha.\beta, \mathcal{L}$
- LR(1) closure is a little more complex:
- Form the set of items just as for LR(0) algorithm.
- Whenever a new item  $C \mapsto .\gamma$  is added because  $A \mapsto \beta.C\delta, \mathcal{L}$  is already in the set, we need to compute its look-ahead set  $\mathcal{M}$ :
  1. The look-ahead set  $\mathcal{M}$  includes  $\text{FIRST}(\delta)$   
(the set of terminals that may start strings derived from  $\delta$ )
  2. If  $\delta$  is itself  $\epsilon$  or can derive  $\epsilon$  (i.e., it is nullable), then the look-ahead  $\mathcal{M}$  also contains  $\mathcal{L}$

# Example LR(1) Closure

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

- Start item:  $S' \mapsto .S\$$  ,  $\{\}$
- Since S is to the right of a '.', add:
  - $S \mapsto .E + S$  ,  $\{\$ \}$  Note:  $\{\$ \}$  is  $\text{FIRST}(\$)$
  - $S \mapsto .E$  ,  $\{\$ \}$
- Need to keep closing, since E appears to the right of a '.' in '.E + S':
  - $E \mapsto .\text{number}$  ,  $\{+\}$  Note: + added for reason 1
  - $E \mapsto .( S )$  ,  $\{+\}$   $\text{FIRST}(+ S) = \{+\}$
- Because E also appears to the right of '.' in '.E' we get:
  - $E \mapsto .\text{number}$  ,  $\{\$ \}$  Note: \$ added for reason 2
  - $E \mapsto .( S )$  ,  $\{\$ \}$   $\delta$  is  $\epsilon$
- All items are distinct, so we're done

# Using the DFA



Fragment of the Action & Goto tables

- 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 ' '



# LR variants

- LR(1) gives maximal power out of a 1 look-ahead symbol parsing table
  - DFA + stack is a push-down automaton
- In practice, LR(1) tables are big.
  - Modern implementations (*e.g.*, menhir) directly generate code
- LALR(1) = “Look-ahead LR”
  - Merge any two LR(1) states whose items are identical except for the look-ahead sets:

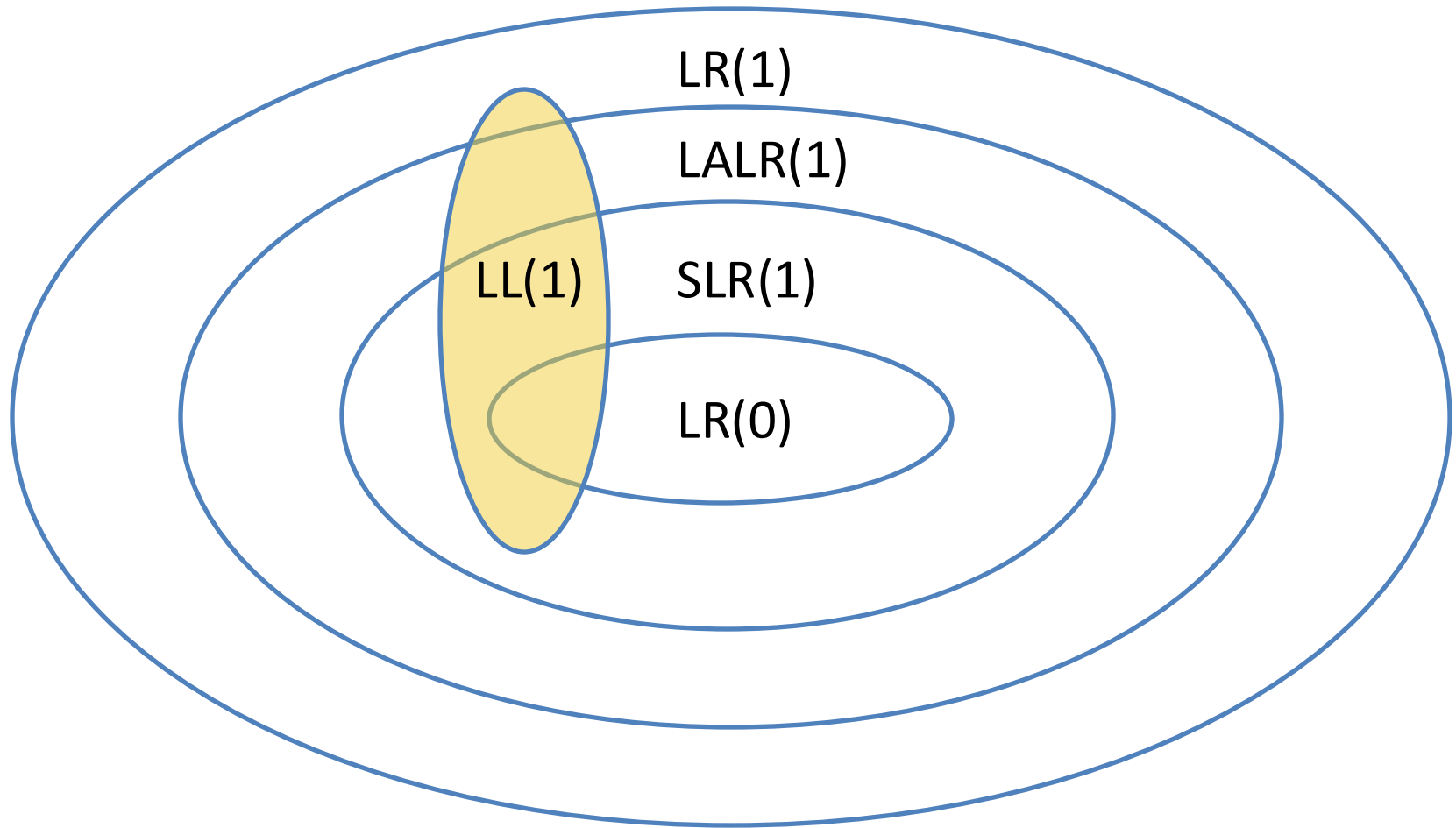
$S' \mapsto .S\$$	$\{\}$
$S \mapsto .E + S$	$\{\$ \}$
$S \mapsto .E \{ \$ \}$	
$E \mapsto .num$	$\{ + \}$
$E \mapsto . ( S )$	$\{ + \}$
$E \mapsto .num$	$\{ \$ \}$
$E \mapsto . ( S )$	$\{ \$ \}$

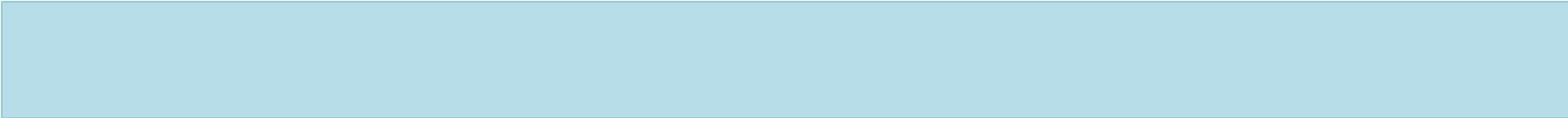


$S' \mapsto .S\$$	$\{\}$
$S \mapsto .E + S$	$\{\$ \}$
$S \mapsto .E \{ \$ \}$	
$E \mapsto .num$	$\{ +, \$ \}$
$E \mapsto . ( S )$	$\{ +, \$ \}$

- 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: yacc, ocaml yacc
- GLR = “Generalized LR” parsing
  - Efficiently compute the set of *all* parses for a given input
  - Later passes should disambiguate based on context

# Classification of Grammars





Debugging parser conflicts.  
Disambiguating grammars.

# MENHIR IN PRACTICE

# Practical Issues

- Dealing with source file location information
  - In the lexer and parser
  - In the abstract syntax
  - See `range.ml`, `ast.ml`
- Lexing comments / strings

# Menhir output

- You can get verbose ocaml yacc debugging information by doing:
  - `menhir --explain ...`
  - or, if using dune, adding this stanza:  
(menhir  
  (modules parser)  
  (flags --explain --dump))
- The result is a `<basename>.conflicts` file that contains a description of the error
  - The parser items of each state use the `'` just as described above
- The flag `--dump` generates a full description of the automaton
- Example: see `start-parser.mly`

# Precedence and Associativity Declarations

- Parser generators, like menhir often support precedence and associativity declarations.
  - Hints to the parser about how to resolve conflicts.
  - See: `good-parser.mly`
- Pros:
  - Avoids having to manually resolve those ambiguities by manually introducing extra nonterminals (as seen in `parser.mly`)
  - Easier to maintain the grammar
- Cons:
  - Can't as easily re-use the same terminal (if associativity differs)
  - 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:

$S \mapsto \text{if } (E) S$

$S \mapsto \text{if } (E) S \text{ else } S$

$S \mapsto X = E$

$E \mapsto \dots$

- Is this grammar OK?

- Consider how to parse:

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

- 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:

if (E<sub>1</sub>) if (E<sub>2</sub>) S<sub>1</sub> else S<sub>2</sub> }

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

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

- See: `else-resolved-parser.mly`



# Alternative: Use { }

- Ambiguity arises because the 'then' branch is not well bracketed:

`if (E1) { if (E2) { S1 } } else S2      // unambiguous`

`if (E1) { if (E2) { S1 } else S2 }    // unambiguous`

- So: could just require brackets
  - But requiring them for the else clause too leads to ugly code for chained if-statements:

```
if (c1) {  
    ...  
} else {  
    if (c2) {  
  
    } else {  
        if (c3) {  
  
        } else {  
  
        }  
    }  
}
```

So, compromise? Allow unbracketed else block only if the body is 'if':

```
if (c1) {  
  
} else if (c2) {  
  
} else if (c3) {  
  
} else {  
  
}
```

Benefits:

- Less ambiguous
- Easy to parse
- Enforces good style