Lecture 13

CIS 4521/5521: COMPILERS

Announcements

- Midterm: March 6th
  - In class
  - One-page, letter-sized, ***hand-written***, double-sided “cheat sheet” of notes permitted
  - Coverage: interpreters, x86, IRs, LLVM IR, calling conventions, lexing, parsing (up to today)
  - See Ed post for previous exams

- Looking ahead: HW4: Oat compiler Frontend
  - released next week (i.e., before Spring Break)
  - Due: Wednesday, March 26th at 10:00pm

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

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

- 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 $X \mapsto \gamma$ is a production. (pop γ, 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

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

- 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 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:
  $S' \longmapsto .S\$$
- 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($\{S' \longmapsto .S\$\}$) = $\{S' \longmapsto .S\$,\ S \longmapsto .(L),\ S \longmapsto .id\}$

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

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

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

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

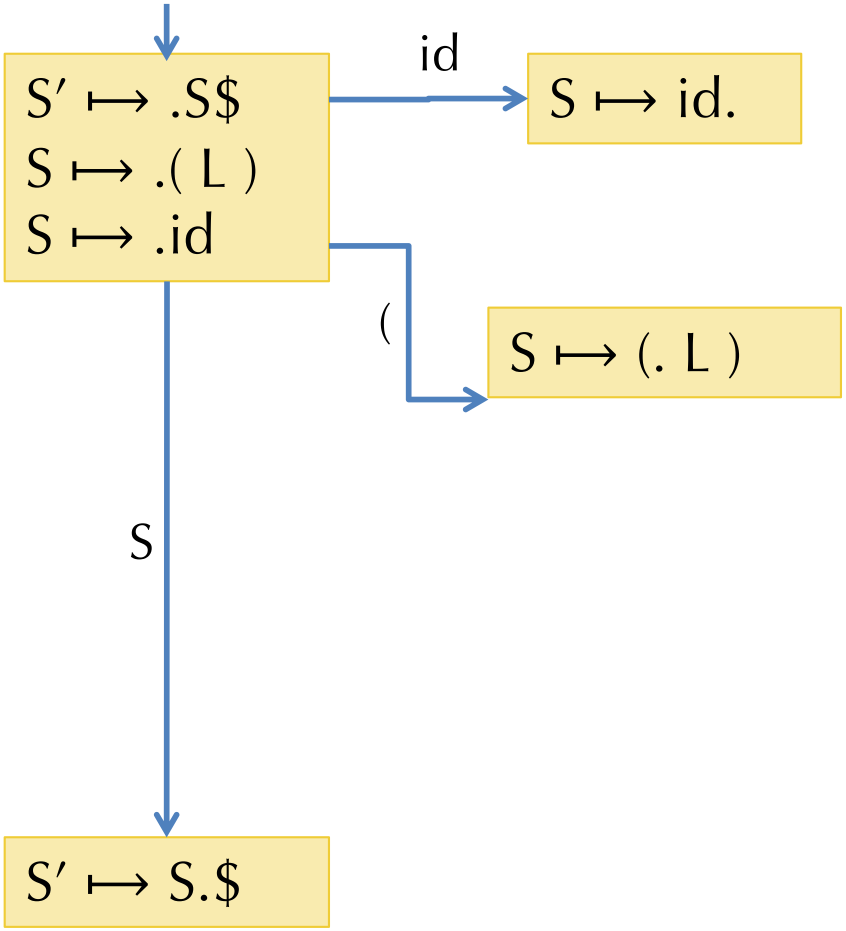
$S' \longmapsto S\$$
$S \longmapsto (\ L\ ) \mid \text{id}$
$L \longmapsto S \mid L\ ,\ S$

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

- 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

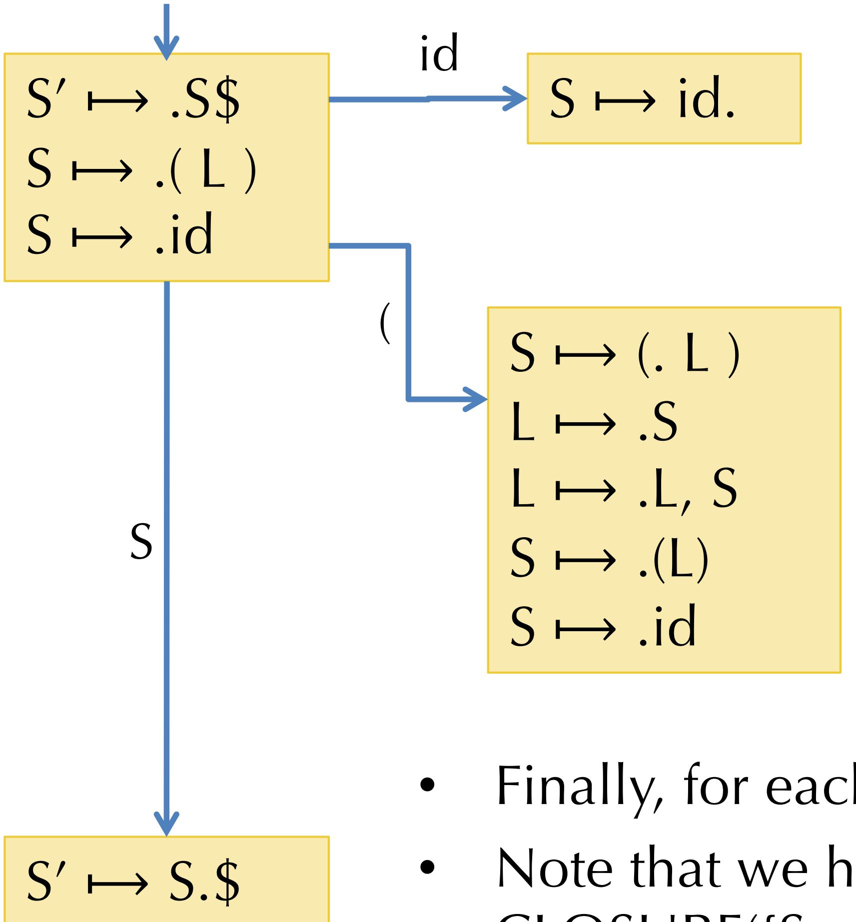
$S' \mapsto S\$$
$S \mapsto (\, L\,) \mid id$
$L \mapsto S \mid L\, ,\, S$



- 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)

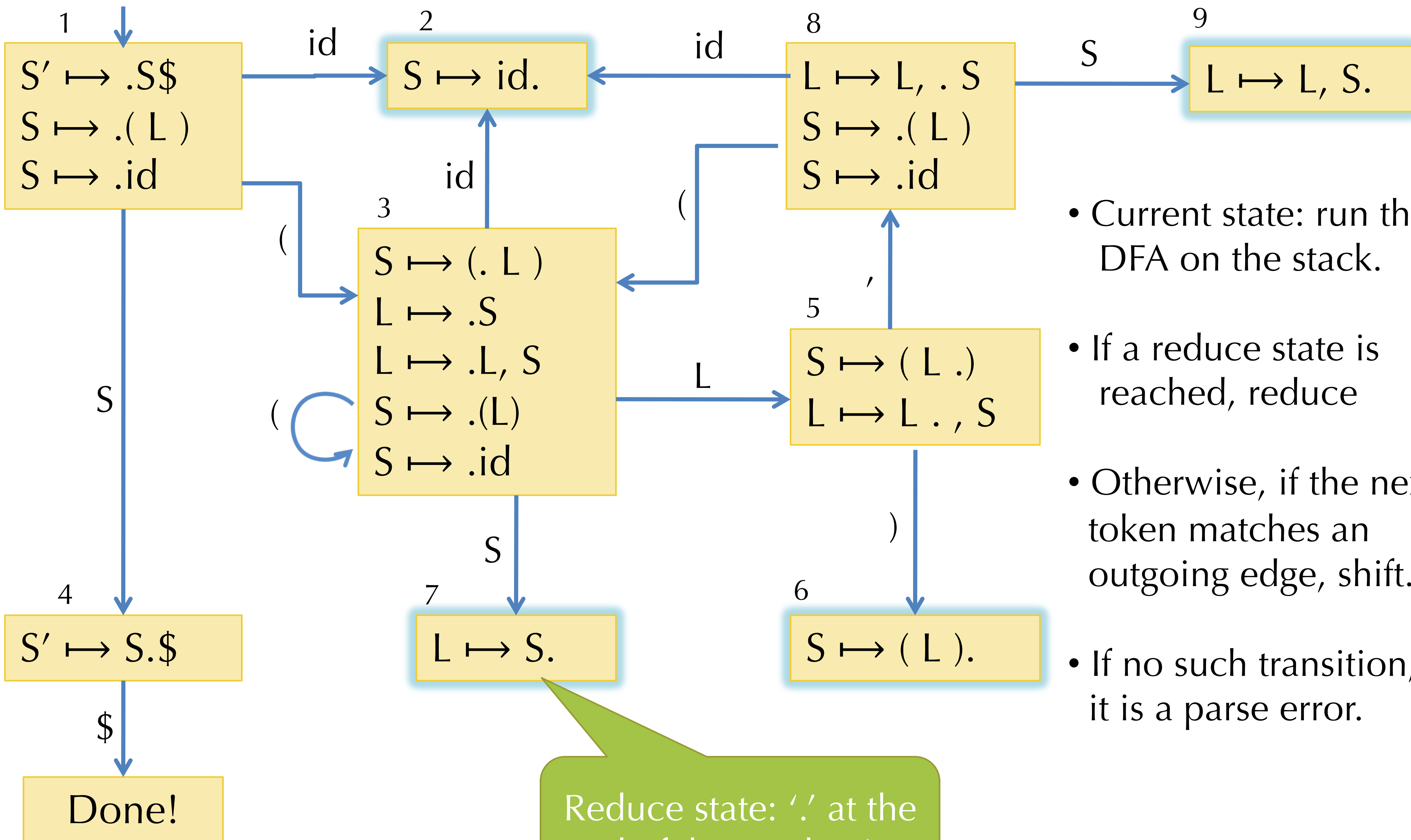
Example: Constructing the DFA

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



- 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



Reduce state: '.' at the end of the production

- 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.

Parsing Using the DFA

1. Run the parser stack through the DFA.
   – If in an accept state: done!
2. Otherwise, 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 no transition, then parse error!)
   – If in a reduce state, $X \mapsto \gamma$ with stack $\alpha\gamma$, pop γ and push X.
3. Go to step 1

Optimization: No need to re-run the DFA from beginning every step

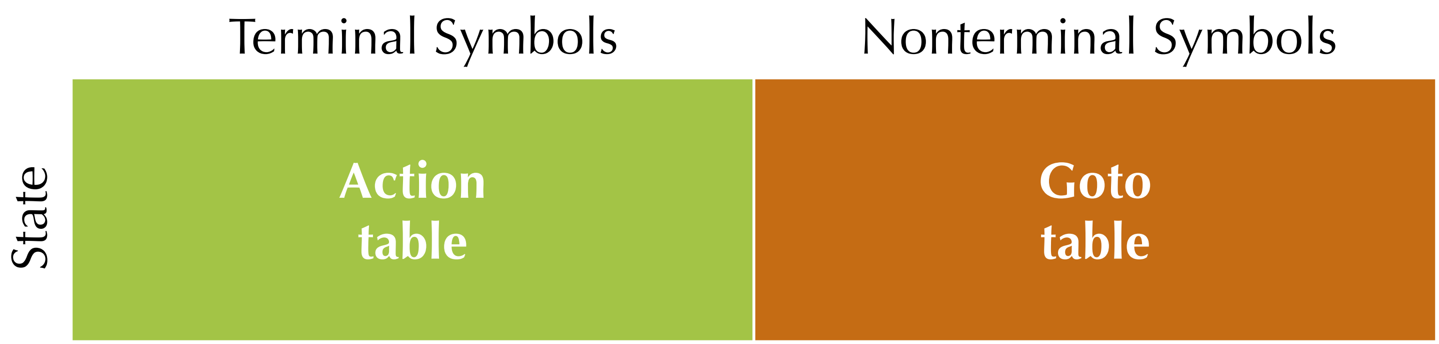
- Store the state with each symbol on the stack: e.g. $_1(_3(_3L_5)_6$
- On a reduction $X \mapsto g$, pop stack to reveal the state too:
  e.g. From stack $_1(_3(_3L_5)_6$ reduce $S \mapsto (L)$ to reach stack $_1(_3$
- Next, push the reduction symbol: e.g. to reach stack $_1(_3S$
- Then take just one 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 (transitions of the DFA)
  - 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⟼id	S⟼id	S⟼id	S⟼id	S⟼id		
3	s3		s2			g7	g5
4					DONE		
5		s6		s8			
6	S ⟼ (L)	S ⟼ (L)	S ⟼ (L)	S ⟼ (L)	S ⟼ (L)		
7	L ⟼ S	L ⟼ S	L ⟼ S	L ⟼ S	L ⟼ S		
8	s3		s2			g9	
9	L ⟼ L,S	L ⟼ L,S	L ⟼ L,S	L ⟼ L,S	L ⟼ 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
$S \longmapsto (\ L\ ).$	$S \longmapsto (\ L\ ).$ $L \longmapsto .L\ ,\ S$	$S \longmapsto L\ ,S.$ $S \longmapsto ,S.$

- 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?

Answer: left grammar is LR(0), right grammar is not. In a state where the stack has E on top, the state will have items
$S \longmapsto E. + S$ and $S \longmapsto E.$
which is a shift/reduce conflict.

- 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 $\mathcal{L}$:

    $A \longmapsto \alpha.\beta \;,\; \mathcal{L}$

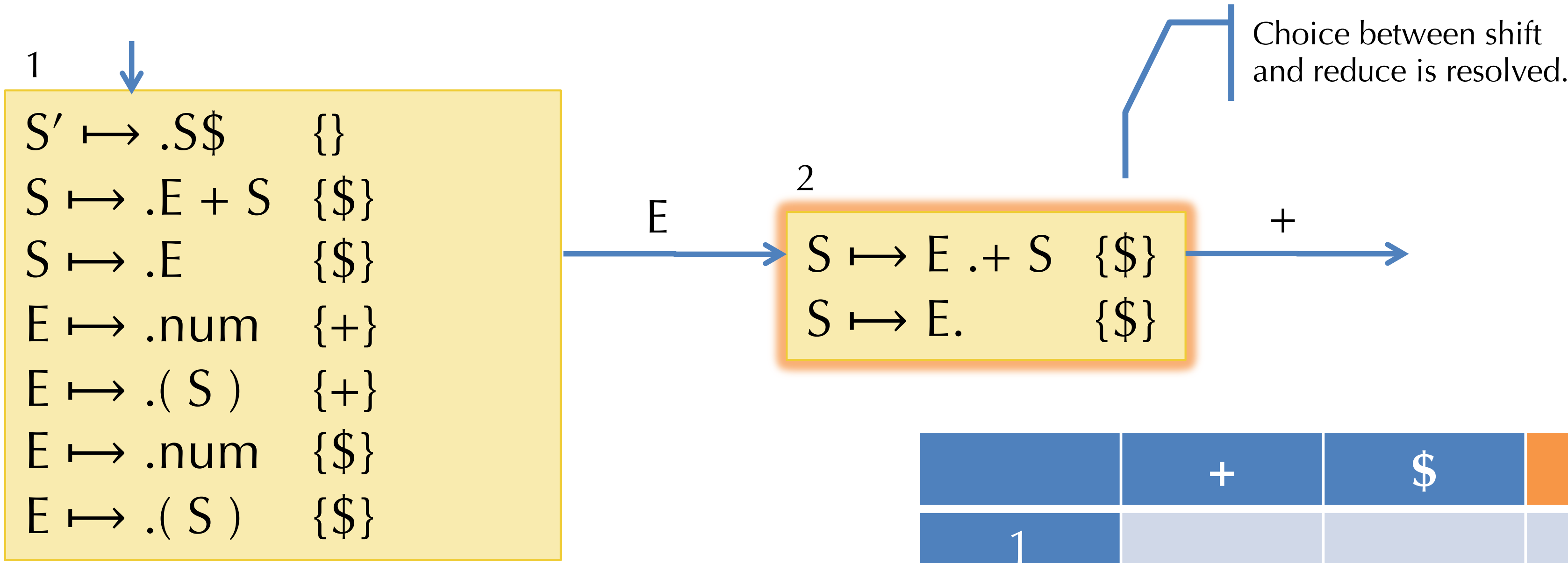
- LR(1) closure is a little more complex:
- Form the set of items just as for LR(0) algorithm.
- Whenever a new item $C \longmapsto .\gamma$ is added because $A \longmapsto \beta.C\delta \;,\; \mathcal{L}$ is already in the set, we need to compute its look-ahead set $\mathcal{M}$:
  1. The look-ahead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

Example LR(1) Closure

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

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

Using the DFA



	+	$	E
1			g2
2	s3	S ⟼ E	

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:

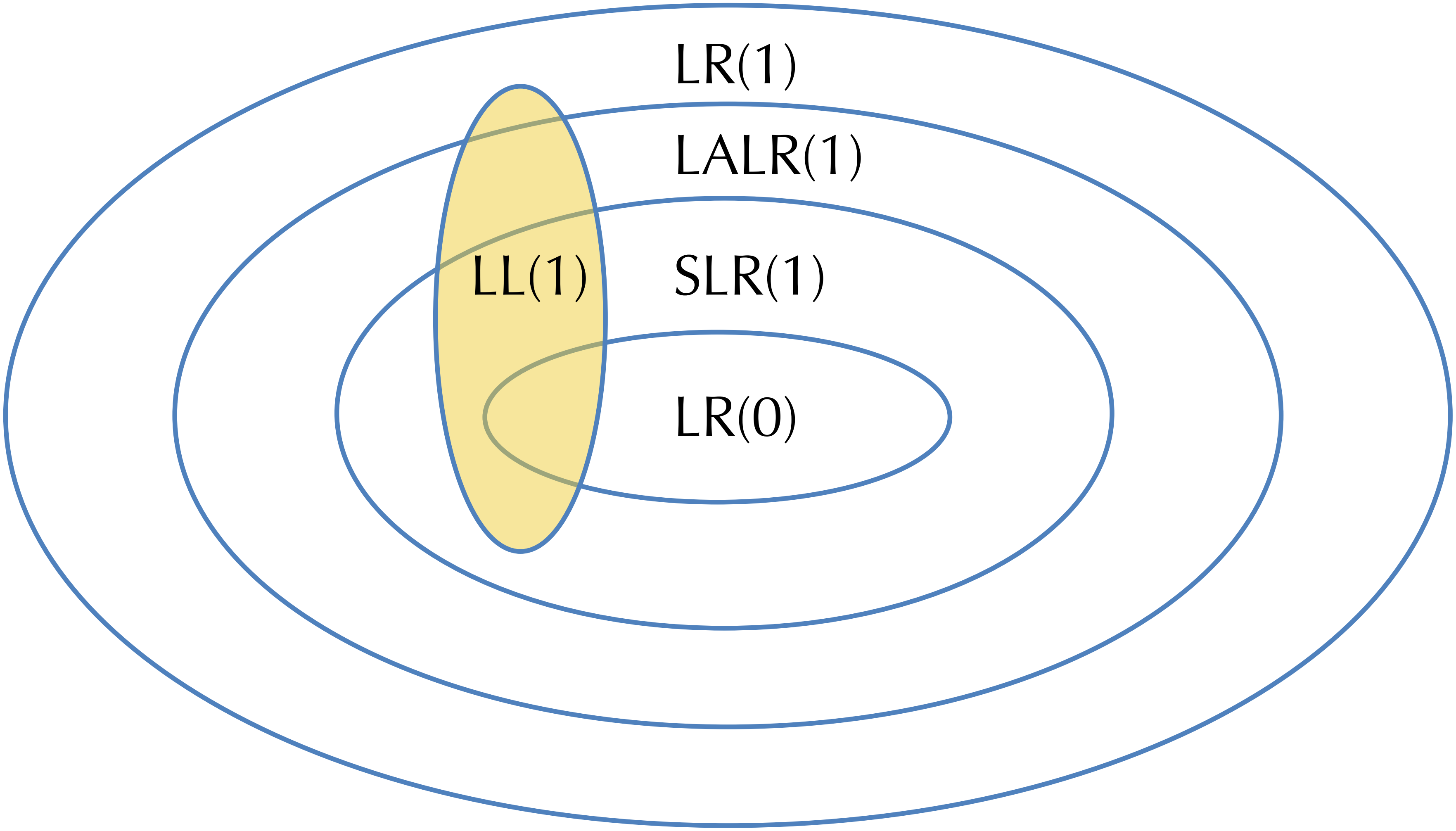
S′ ⟼ .S$	{}
S ⟼ .E + S	{$}
S ⟼ .E	{$}
E ⟼ .num	{+}
E ⟼ .(S)	{+}
E ⟼ .num	{$}
E ⟼ .(S)	{$}

⇒

S′ ⟼ .S$	{}
S ⟼ .E + S	{$}
S ⟼ .E	{$}
E ⟼ .num	{+,$}
E ⟼ .(S)	{+,$}

  – 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, ocamlyacc
- 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 ocamlyacc debugging information by doing:
  - `menhir --explain …`
  - or, if using dune, adding this stanza:

```
(menhir
  (modules parser)
  (flags --dump)
  (explain true))
```

- 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 ⟼ if (E) S
S ⟼ if (E) S else S
S ⟼ X = E
E ⟼ …
```

- 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:

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

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

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

- See: else-resolved-parser.mly

Alternative: Use { }

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

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

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