Lecture 12

CIS 341: COMPILERS

Announcements

- Midterm: March 3rd
  - In class
  - 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
- HW4: Compiling Oat v.1
  - released soon(ish)
  - due March 23rd

LL(1) GRAMMARS

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 ⟼ S$
S ⟼ ES′
S′ ⟼ ε
S′ ⟼ + S
E ⟼ number | (S)

	number	+	(	)	$ (EOF)
T	⟼ S$		⟼S$		
S	⟼ E S′		⟼E S′		
S′		⟼ + S		⟼ ε	⟼ ε
E	⟼ num.		⟼ (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 → γ
- Construct the set of all input tokens that may appear *first* in strings that can be derived from γ
  - Add the production → γ 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 → γ 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 ⟼ S$
S ⟼ ES′
S′ ⟼ ε
S′ ⟼ + S
E ⟼ number | (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	⟼ S$		⟼S$		
S	⟼ E S′		⟼E S′		
S′		⟼ + S		⟼ ε	⟼ ε
E	⟼ num.		⟼ (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	$\mapsto$ S$		$\mapsto$S$		
S	$\mapsto$ E S′		$\mapsto$E S′		
S′		$\mapsto$ + S		$\mapsto$ ε	$\mapsto$ ε
E	$\mapsto$ num.		$\mapsto$ (S)		

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

DEMO: HANDPARSER.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?

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

Top-down vs. Bottom up

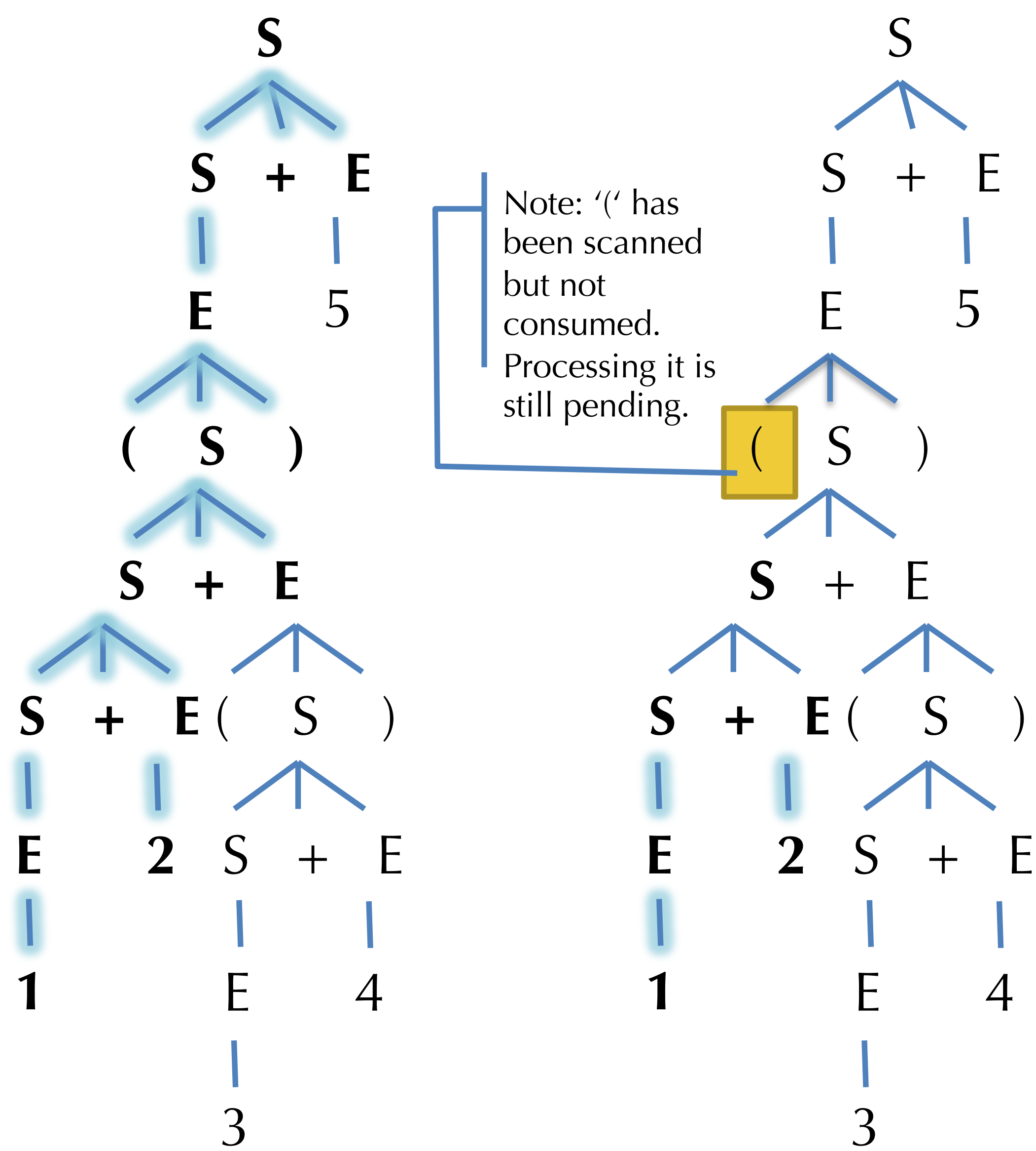
- Consider the left-recursive grammar:

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

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

- What part of the tree must we know after scanning just "(1 + 2" ?

- In top-down, must be able to guess which productions to use…



Top-down | Bottom-up

Progress of Bottom-up Parsing

Rightmost derivation ↑

Reductions	Scanned	Input Remaining
(1 + 2 + (3 + 4)) + 5 ⟻		(1 + 2 + (3 + 4)) + 5
(**E** + 2 + (3 + 4)) + 5 ⟻	(	1 + 2 + (3 + 4)) + 5
(**S** + 2 + (3 + 4)) + 5 ⟻	(1	+ 2 + (3 + 4)) + 5
(S + **E** + (3 + 4)) + 5 ⟻	(1 + 2	+ (3 + 4)) + 5
(**S** + (3 + 4)) + 5 ⟻	(1 + 2	+ (3 + 4)) + 5
(S + (**E** + 4)) + 5 ⟻	(1 + 2 + (3	+ 4)) + 5
(S + (**S** + 4)) + 5 ⟻	(1 + 2 + (3	+ 4)) + 5
(S + (S + **E**)) + 5 ⟻	(1 + 2 + (3 + 4	)) + 5
(S + (**S**)) + 5 ⟻	(1 + 2 + (3 + 4	)) + 5
(S + **E**) + 5 ⟻	(1 + 2 + (3 + 4)	) + 5
(**S**) + 5 ⟻	(1 + 2 + (3 + 4)	) + 5
E + 5 ⟻	(1 + 2 + (3 + 4))	+ 5
S + 5 ⟻	(1 + 2 + (3 + 4))	+ 5
S + **E** ⟻	(1 + 2 + (3 + 4)) + 5	
S		

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

Shift/Reduce Parsing

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

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

Stack	Input	Action
	(1 + 2 + (3 + 4)) + 5	shift (
(	1 + 2 + (3 + 4)) + 5	shift 1
(1	+ 2 + (3 + 4)) + 5	reduce: $E \mapsto$ number
(E	+ 2 + (3 + 4)) + 5	reduce: $S \mapsto E$
(S	+ 2 + (3 + 4)) + 5	shift +
(S +	2 + (3 + 4)) + 5	shift 2
(S + 2	+ (3 + 4)) + 5	reduce: $E \mapsto$ number

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.

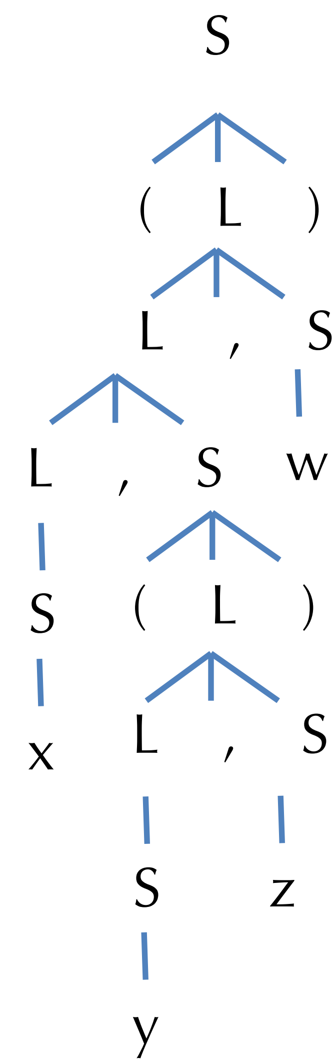
Example LR(0) Grammar: Tuples

- Example grammar for non-empty tuples and identifiers:

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

- 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

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

Stack	Input	Action
	(x, (y, z), w)	shift (
(	x, (y, z), w)	shift x
(x	, (y, z), w)	reduce S ⟼ id
(S	, (y, z), w)	reduce L ⟼ S
(L	, (y, z), w)	shift ,
(L,	(y, z), w)	shift (
(L, (	y, z), w)	shift y
(L, (y	, z), w)	reduce S ⟼ id
(L, (S	, z), w)	reduce L ⟼ S
(L, (L	, z), w)	shift ,
(L, (L,	z), w)	shift z
(L, (L, z	), w)	reduce S ⟼ id
(L, (L, S	), w)	reduce L ⟼ L, S
(L, (L	), w)	shift)
(L, (L)	, w)	reduce S ⟼ (L)
(L, S	, w)	reduce L ⟼ L, S
(L	, w)	shift ,

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 α is different for different possible reductions since in productions $X \longmapsto \gamma$ and $Y \longmapsto \beta$, γ and β 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

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

- 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

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

- 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.

Example: Constructing the DFA

$S' \longmapsto .S\$$

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

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

Example: Constructing the DFA

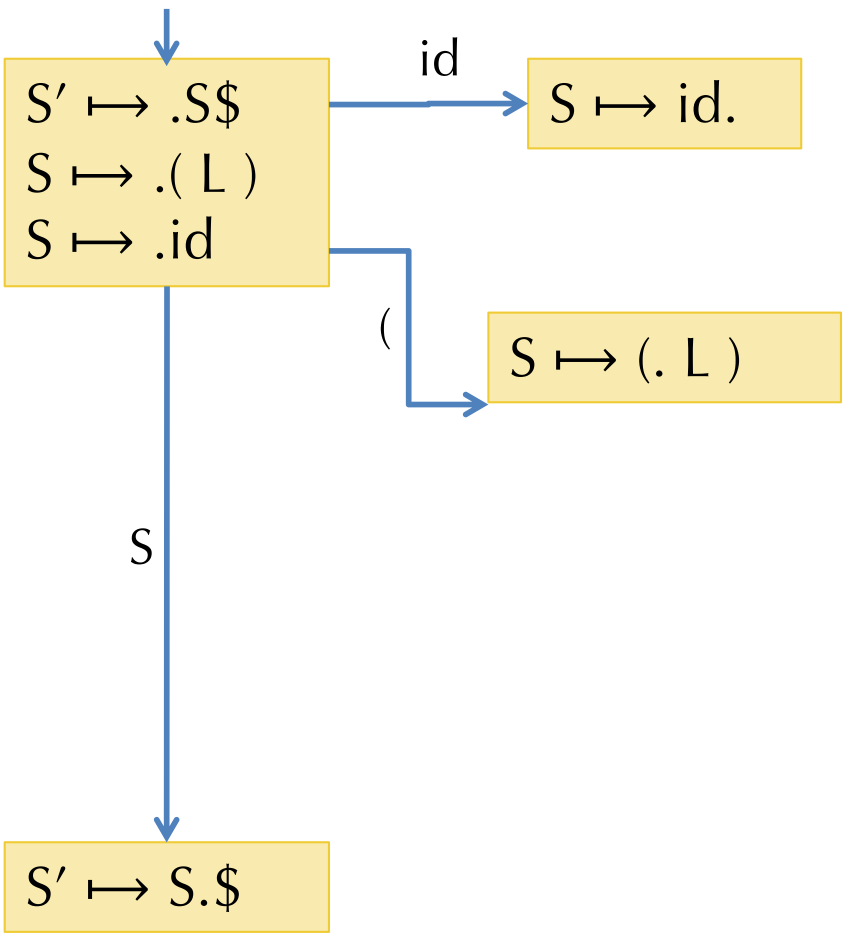
$S' \longmapsto .S\$$
$S \longmapsto .(\ L\ )$
$S \longmapsto .id$

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

- Next, we take the closure of that state:
  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



$S' \longmapsto S\$$

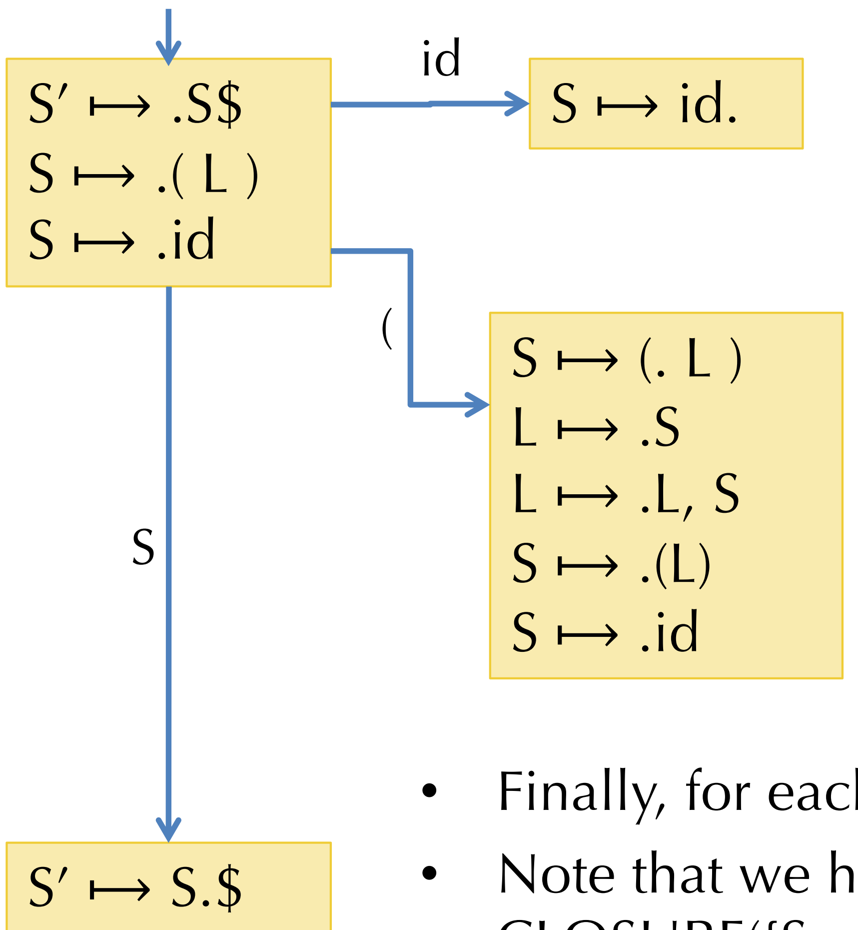
$S \longmapsto (\ L\ )\ \ |\ \ id$

$L \longmapsto S\ \ |\ \ L\ ,\ S$

- 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)

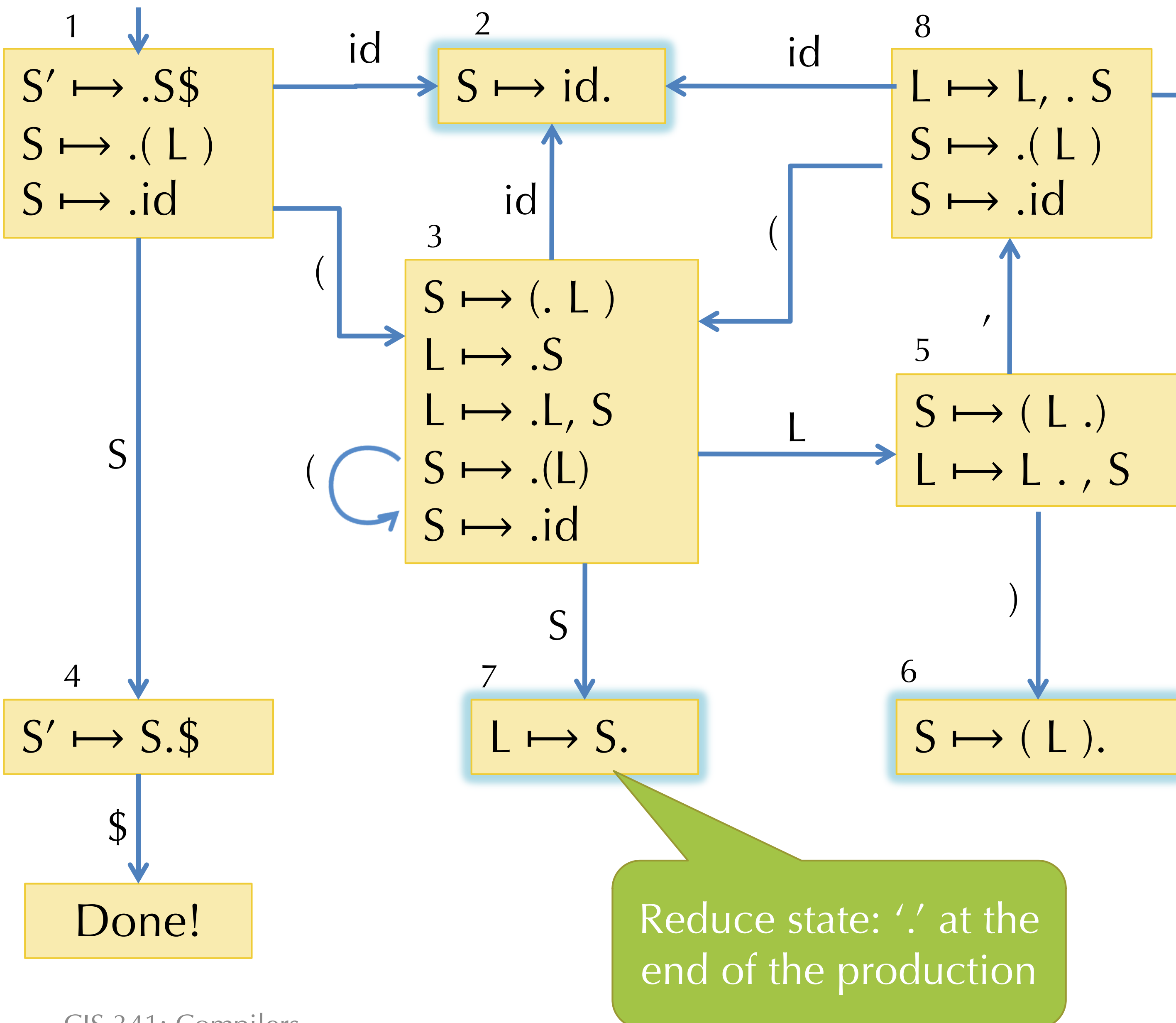
Example: Constructing the DFA

S′ ⟼ S$
S ⟼ (L) | id
L ⟼ 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



- 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.

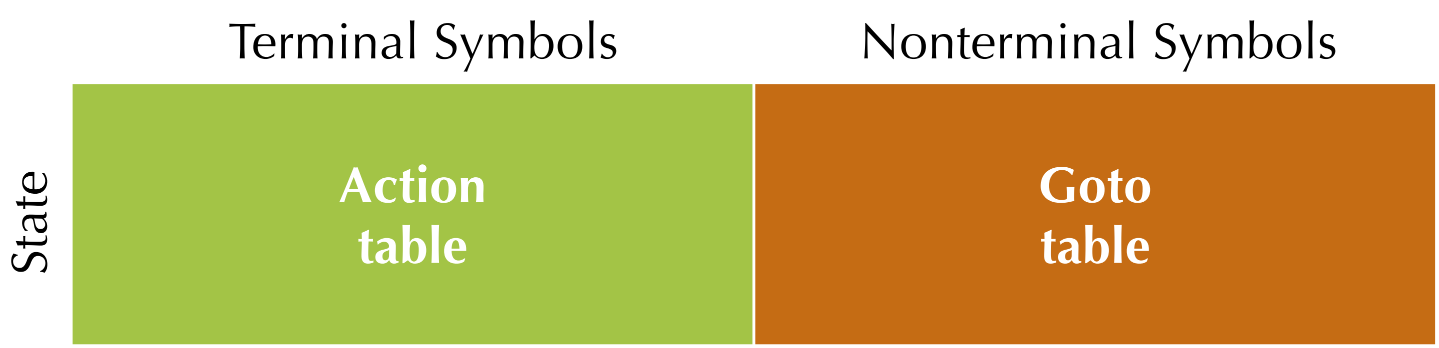
- Optimization: No need to re-run the DFA from beginning every step
  - Store the state with each symbol on the stack: e.g. $_1(_3(_3L_5)_6$
  - On a reduction $X \longmapsto \gamma$, pop stack to reveal the state too:
    e.g. From stack $_1(_3(_3L_5)_6$ reduce $S \longmapsto (L)$ to reach stack $_1(_3$
  - 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⟼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 ⟼ (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 ⟼ S + E | E E ⟼ number | (S)	S ⟼ E + S | E E ⟼ number | (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:

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

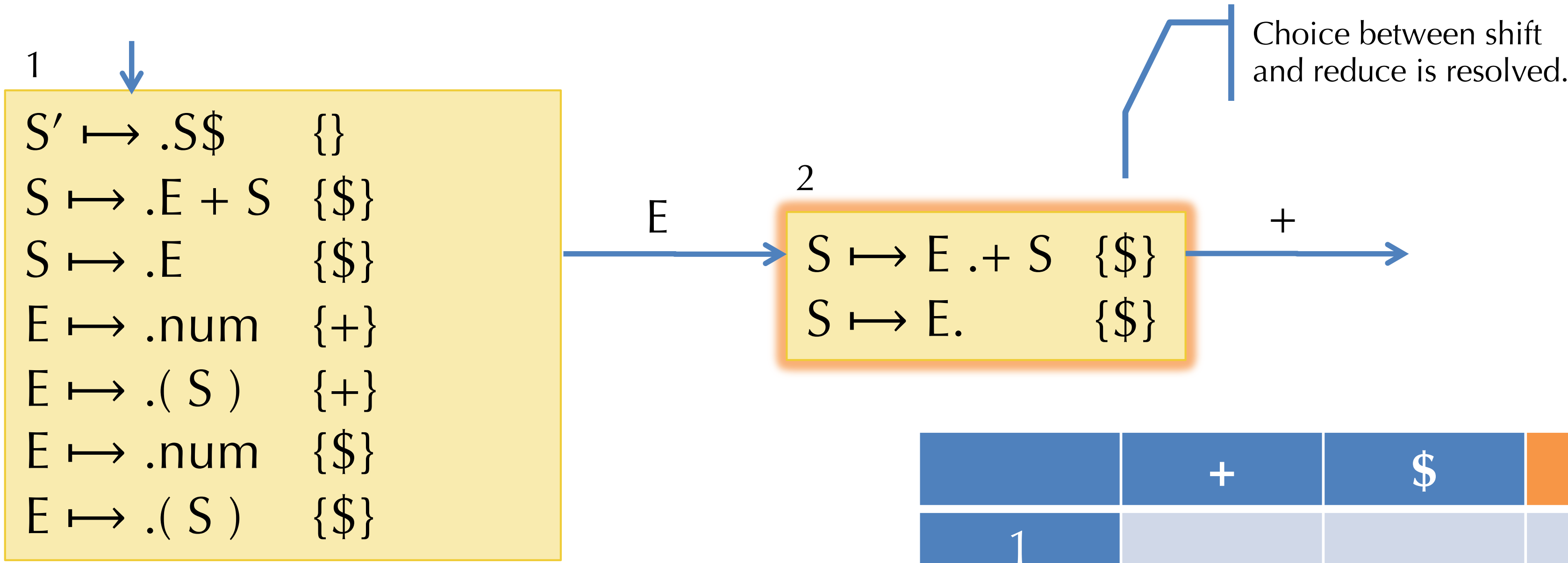
- LR(1) closure is a little more complex:
- Form the set of items just as for LR(0) algorithm.
- Whenever a new item $C \longmapsto .\gamma$ is added because $A \longmapsto \beta.C\delta \ , \ \mathcal{L}$ is already in the set, we need to compute its look-ahead set $\mathcal{M}$:
  1. The look-ahead 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 Closure

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

- Start item: $S' \longmapsto .S\$$, {}
- Since S is to the right of a '.', add:
  - $S \longmapsto .E + S$, {\$} — Note: {\$} is FIRST(\$)
  - $S \longmapsto .E$, {\$}
- Need to keep closing, since E appears to the right of a '.' in '.E + S':
  - $E \longmapsto .\text{number}$, {+} — Note: + added for reason 1
  - $E \longmapsto .(S)$, {+} — FIRST(+ S) = {+}
- Because E also appears to the right of '.' in '.E' we get:
  - $E \longmapsto .\text{number}$, {\$} — Note: \$ added for reason 2
  - $E \longmapsto .(S)$, {\$} — 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 (recall CIS 262)
- 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:

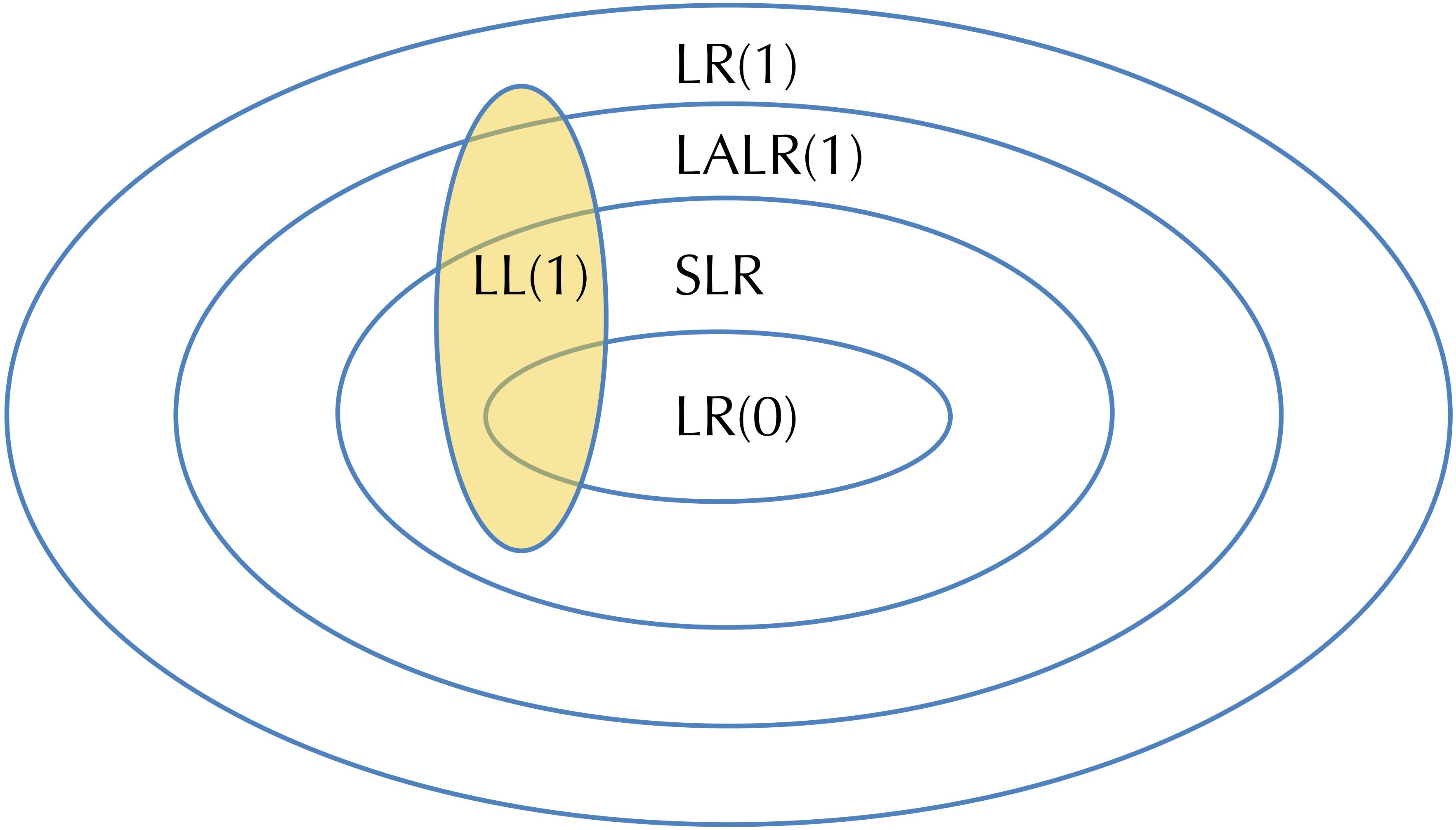
S′ ⟼ .S$	{}
S ⟼ .E + S	{$}
S ⟼ .E	{$}
E ⟼ .num	{+}
E ⟼ .(S)	{+}
E ⟼ .num	{$}
E ⟼ .(S)	{$}

⟹

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 other 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 --explain))
```

- 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:
  $S \mapsto$ `if (E) S`
  $S \mapsto$ `if (E)` S `else` S
  $S \mapsto X = E$
  $E \mapsto \ldots$

- 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:

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

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

$S \mapsto M \mid U$	// 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:

  $\texttt{if (E}_1\texttt{) \{ if (E}_2\texttt{) \{ S}_1\texttt{ \} \} else S}_2$ // 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