CS 516: COMPILERS

Lecture 12

Topics

- Parsing (finding derivations in a grammar)
  - LR Grammars
  - Shift/Reduce parsing
  - LR(0) Grammars
  - Menhir

LR GRAMMARS

Bottom-Up Parsing (LR Parsers)

Bottom-Up Parsing (LR Parsers)

- LR(k) parser:
  - Left-to-right scanning
  - Rightmost derivation
  - k lookahead symbols

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)

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
  - Preferred — Used by many parser generators (e.g. yacc, CUP, ocamlyacc, merlin, etc.)
  - Better error detection/recovery

Bottom-Up Parsing (LR Parsers)

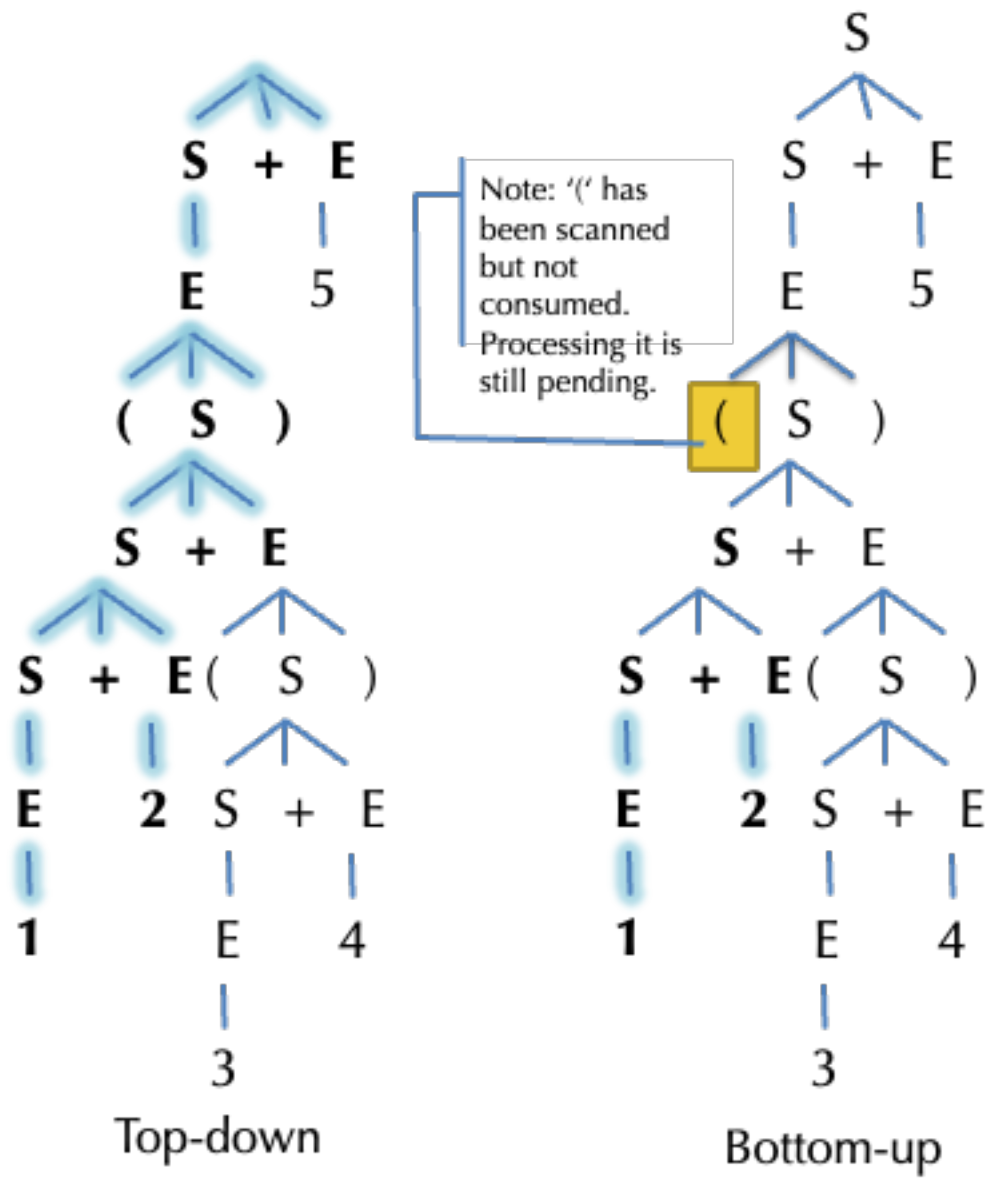
- Consider the left-recursive grammar:

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

- (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…



Bottom-Up Parsing (LR Parsers)

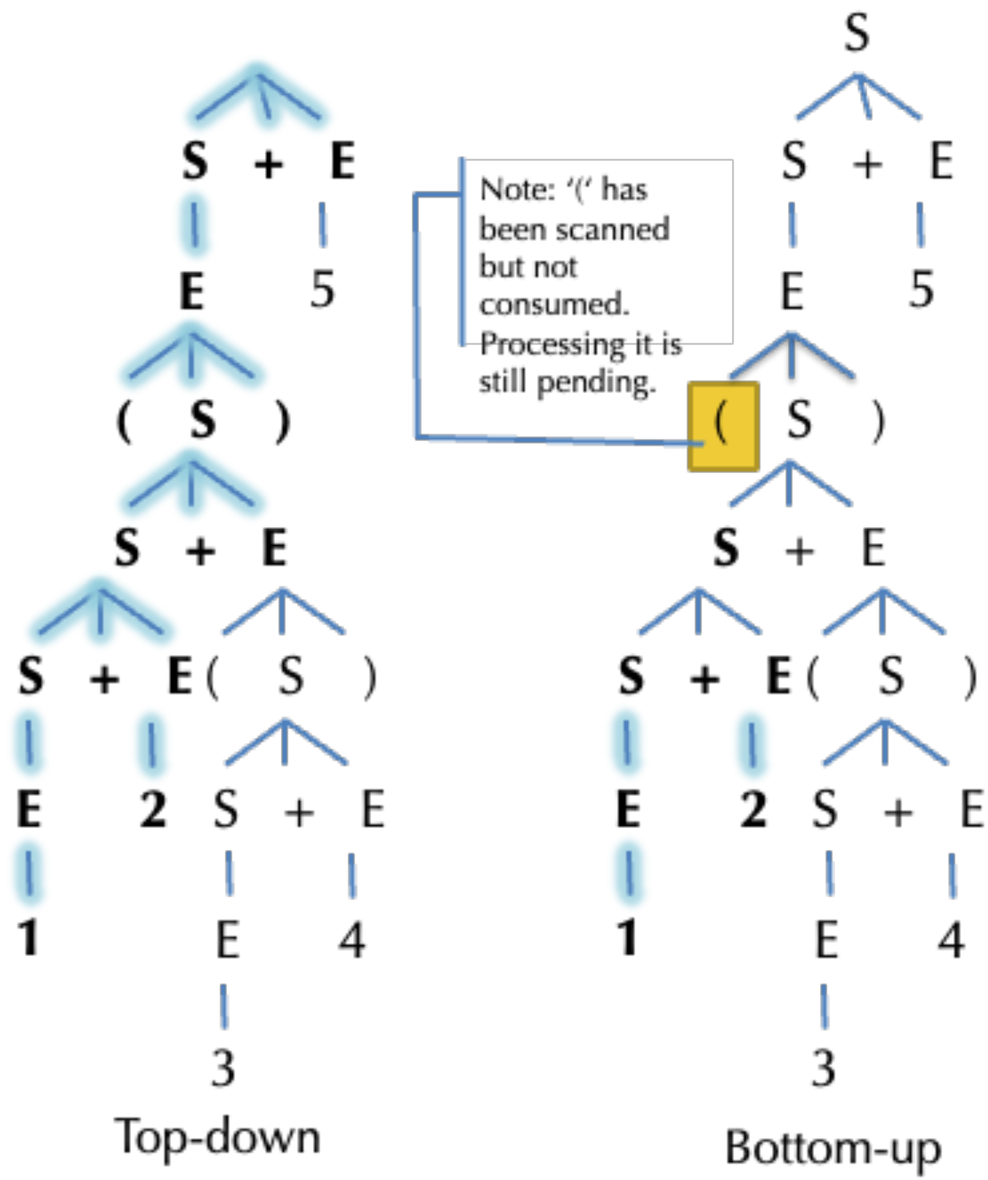
- Consider the left-recursive grammar:

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

- (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…



Bottom-Up Parsing

- **Reduces** a string to the start symbol by **inverting** productions.

Token Stream	Production
`int * int + int`	$T \longmapsto$ `int`
`int * T + int`	$T \longmapsto$ `int * T`
`T + int`	$T \longmapsto$ `int`
`T + T`	$E \longmapsto T$
`T + E`	$E \longmapsto T + E$
`E`	

$E \longmapsto \quad T + E \mid T$
$T \longmapsto \quad$ `int * T` $\mid$ `int` $\mid$ `(E)`

Bottom-Up Parsing

- **Reduces** a string to the start symbol by **inverting** productions.

input string:

Token Stream	Production
`int * int + int`	$T \longmapsto$ `int`
`int * T + int`	$T \longmapsto$ `int * T`
`T + int`	$T \longmapsto$ `int`
`T + T`	$E \longmapsto T$
`T + E`	$E \longmapsto T + E$
`E`	

$E \longmapsto T + E \mid T$
$T \longmapsto$ `int * T` $\mid$ `int` $\mid (E)$

Bottom-Up Parsing

- **Reduces** a string to the start symbol by **inverting** productions.

	Token Stream	Production
input string:	`int * int + int`	$T \longmapsto$ `int`
	`int * T + int`	$T \longmapsto$ `int * T`
	`T + int`	$T \longmapsto$ `int`
	`T + T`	$E \longmapsto T$
	`T + E`	$E \longmapsto T + E$
start symbol:	`E`	

$E \longmapsto T + E \mid T$
$T \longmapsto$ `int * T` $\mid$ `int` $\mid (E)$

Bottom-Up Parsing

- **Reduces** a string to the start symbol by **inverting** productions.

Ran the production backwards

	Token Stream	on
input string:	`int * int + int`	T ⟼ `int`
	`int * T + int`	T ⟼ `int * T`
	`T + int`	T ⟼ `int`
	`T + T`	E ⟼ T
	`T + E`	E ⟼ T + E
start symbol:	`E`	

E ⟼ T + E | T
T ⟼ `int * T` | `int` | (E)

Bottom-Up Parsing

- **Reduces** a string to the start symbol by **inverting** productions.

Ran the production backwards

	Token Stream	on
input string:	`int * int + int`	$T \longmapsto$ `int`
	`int * T + int`	$T \longmapsto$ `int * T`
	`T + int`	$T \longmapsto$ `int`
	`T + T`	$E \longmapsto T$
	`T + E`	$E \longmapsto T + E$
start symbol:	`E`	

$E \longmapsto T + E \mid T$

$T \longmapsto$ `int * T` $\mid$ `int` $\mid$ `(E)`

Bottom-Up Parsing

- **Reduces** a string to the start symbol by **inverting** productions.

Ran the production backwards

	Token Stream	Production
input string:	`int * int + int`	T ⟼ `int`
	`int * T + int`	T ⟼ `int * T`
	`T + int`	T ⟼ `int`
	`T + T`	E ⟼ T
	`T + E`	E ⟼ T + E
start symbol:	`E`	

Highlighting what will be replaced.

E ⟼ T + E | T
T ⟼ `int * T` | `int` | (E)

Bottom-Up Parsing

- **Reduces** a string to the start symbol by **inverting** productions.

	Token Stream	Production
input string:	`int * int + int`	$T \longmapsto$ `int`
	$\boxed{\texttt{int * T}}$ `+ int`	$T \longmapsto$ `int * T`
	`T +` $\boxed{\texttt{int}}$	$T \longmapsto$ `int`
	`T +` $\boxed{\texttt{T}}$	$E \longmapsto T$
	$\boxed{\texttt{T + E}}$	$E \longmapsto T + E$
start symbol:	`E`	

Highlighting what will be replaced.

$E \longmapsto\ T + E \mid T$

$T \longmapsto$ `int * T` | `int` | `(E)`

Bottom-Up Parsing

- **Reduces** a string to the start symbol by **inverting** productions.

	Token Stream	Production
input string:	`int * int + int`	T ⟼ `int`
	`int * T + int`	T ⟼ `int * T`
	`T + int`	T ⟼ `int`
	`T + T`	E ⟼ T
	`T + E`	E ⟼ T + E
start symbol:	`E`	

reductions

Highlighting what will be replaced.

E ⟼ T + E | T
T ⟼ `int * T` | `int` | `(E)`

Bottom-Up Parsing

- **Reduces** a string to the start symbol by **inverting** productions.

	Token Stream	Production
input string:	int * int + int	T ⟼ int
	int * T + int	T ⟼ int * T
	T + int	T ⟼ int
	T + T	E ⟼ T
	T + E	E ⟼ T + E
start symbol:	E	

productions

reductions

Highlighting what will be replaced.

E ⟼ T + E | T
T ⟼ int * T | int | (E)

Bottom-Up Parsing

- The productions, read backwards, trace a **rightmost derivation**

Token Stream	Production
`int * int + int`	$T \longmapsto$ `int`
`int * T + int`	$T \longmapsto$ `int * T`
`T + int`	$T \longmapsto$ `int`
`T + T`	$E \longmapsto T$
`T + E`	$E \longmapsto T + E$
`E`	

Bottom-Up Parsing

- The productions, read backwards, trace a **rightmost derivation**

Token Stream	Production
`int * int + int`	T $\longmapsto$ `int`
`int * T + int`	T $\longmapsto$ `int * T`
`T + int`	T $\longmapsto$ `int`
`T + T`	E $\longmapsto$ T
`T + E`	E $\longmapsto$ T + E
`E`	

parsing

Bottom-Up Parsing

- The productions, read backwards, trace a **rightmost derivation**

Token Stream	Production
`int * int + int`	T ⟼ `int`
`int *` $\boxed{\text{T}}$ `+ int`	T ⟼ `int * T`
$\boxed{\text{T}}$ `+ int`	T ⟼ `int`
T + $\boxed{\text{T}}$	E ⟼ T
T + $\boxed{\text{E}}$	E ⟼ T + E
$\boxed{\text{E}}$	

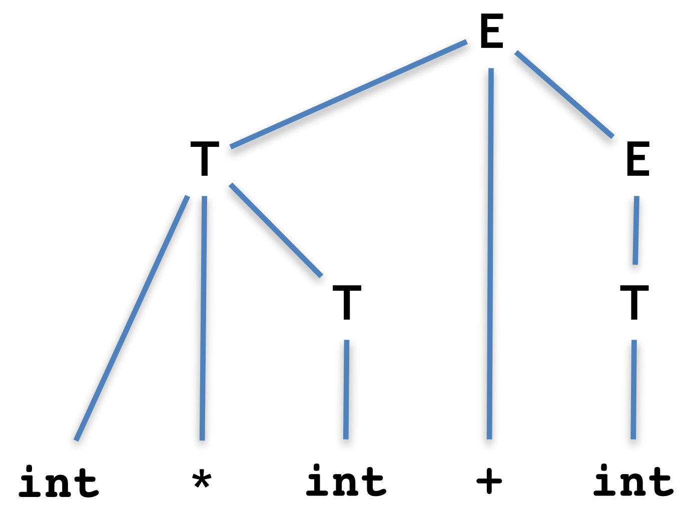
parsing

Rightmost nonterminal is expanded

Bottom-Up Parsing

- Important Fact #1:
  - A bottom-up parser traces a **rightmost derivation** in **reverse**.

`int * int + int`
`int * T + int`
`T + int`
`T + T`
`T + E`
`E`



Bottom-Up Parsing

- Important Fact #1:
  - A bottom-up parser traces a **rightmost derivation** in **reverse**.

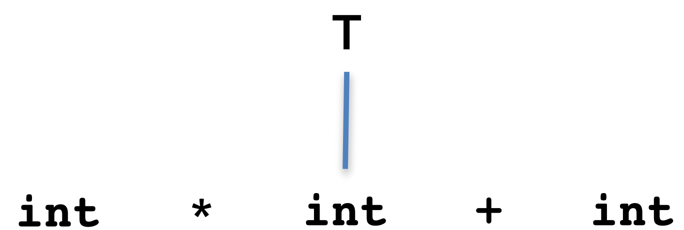
```
int * int + int
```

int   *****   **int**   **+**   **int**

Bottom-Up Parsing

- Important Fact #1:
  - A bottom-up parser traces a **rightmost derivation** in **reverse**.

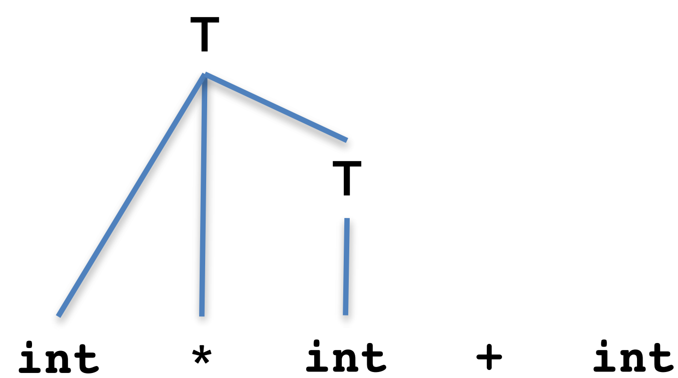
```
int * int + int
int * T   + int
```



Bottom-Up Parsing

- Important Fact #1:
  - A bottom-up parser traces a **rightmost derivation** in **reverse**.

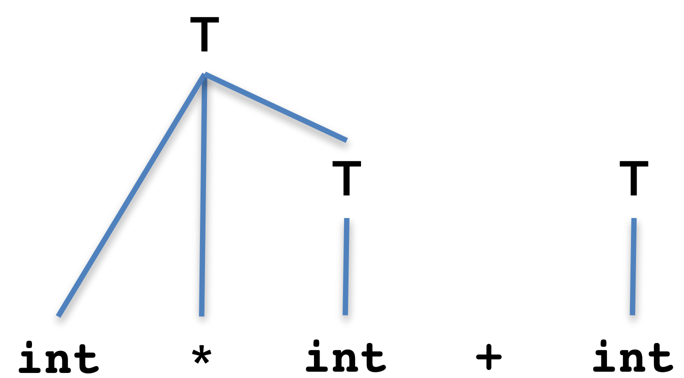
`int * int + int`
`int * T + int`
`T + int`



Bottom-Up Parsing

- Important Fact #1:
  - A bottom-up parser traces a **rightmost derivation** in **reverse**.

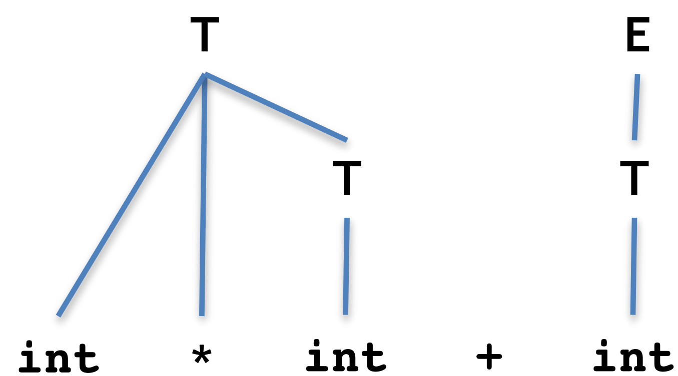
`int * int + int`
`int * T + int`
`T + int`
`T + T`



Bottom-Up Parsing

- Important Fact #1:
  – A bottom-up parser traces a **rightmost derivation** in **reverse**.

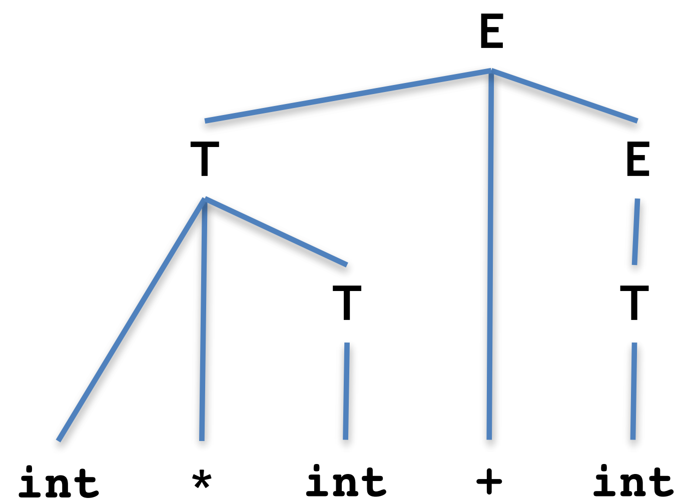
```
int * int + int
int * T   + int
T + int
T + T
T + E
```



Bottom-Up Parsing

- Important Fact #1:
  - A bottom-up parser traces a **rightmost derivation** in **reverse**.

`int * int + int`
`int * T + int`
`T + int`
`T + T`
`T + E`
`E`



SHIFT REDUCE PARSING

Shift/Reduce Parsing

- Important Fact #1:
  - A bottom-up parser traces a **rightmost derivation** in **reverse**.

- Consequences:
  - Let $\alpha\beta\omega$ be a step of a bottom-up parse
  - Assume the next reduction is by $X \longmapsto \beta$
  - Then ω is a string of terminals

- Why?
  - Because $\alpha X\omega \longmapsto \alpha\beta\omega$ is a step in a right-most derivation

Shift/Reduce Parsing

$\alpha X | \omega$

terminals & nonterminals

rightmost non-terminal

terminals only

unexamined input

- Idea: split string into two substrings:
  - Right substring is as yet unexamined by parsing
  - Left substring has terminals and non-terminals
  - The dividing point is marked by a "|"

Shift/Reduce Parsing

- Bottom-up parsing uses only two kinds of actions:

- ***Shift***: Move | one place to the right
  - Shifts a terminal to the left string

$$ABC \mid xyz \Rightarrow ABCx \mid yz$$

- ***Reduce***: Apply an inverse production at the right end of the left string
  - If $A \mapsto xy$ is a production, then

$$Cbxy \mid ijk \Rightarrow CbA \mid ijk$$

Shift/Reduce Parsing

`int * int	+ int`	**reduce** T $\longmapsto$ `int`
`int * T	+ int`	**reduce** T $\longmapsto$ `int * T`
`T + int |`	**reduce** T $\longmapsto$ `int`
`T + T |`	**reduce** E $\longmapsto$ T
`T + E |`	**reduce** E $\longmapsto$ T + E
`E |`	

Shift/Reduce Parsing

| `int * int + int`	shift
`int` | `* int + int`	shift
`int *` | `int + int`	shift
`int * int` | `+ int`	reduce T $\longmapsto$ `int`
`int *` T | `+ int`	reduce T $\longmapsto$ `int * T`
T | `+ int`	shift
T + | `int`	shift
T + `int` |	reduce T $\longmapsto$ `int`
T + T |	reduce E $\longmapsto$ T
T + E |	reduce E $\longmapsto$ T + E
E |	

Shift/Reduce Parsing

```
| int * int + int
```

int * **int** + **int**

Shift/Reduce Parsing

```
| int * int + int
```

int * **int** + **int**

Shift/Reduce Parsing

`| int * int + int`
`int | * int + int`
`int * | int + int`
`int * int | + int`

`int * int + int`

Shift/Reduce Parsing

| int * int + int
int | * int + int
int * | int + int
int * int | + int

int * int + int

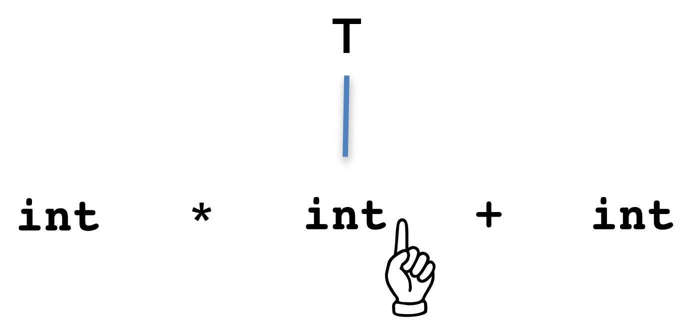
Shift/Reduce Parsing

```
| int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int
```

```
int * int + int
```

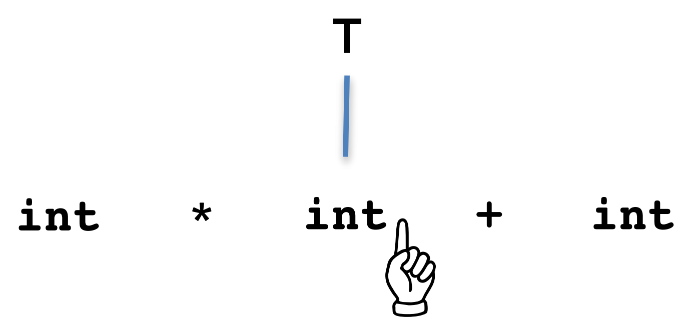
Shift/Reduce Parsing

```
| int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int
```



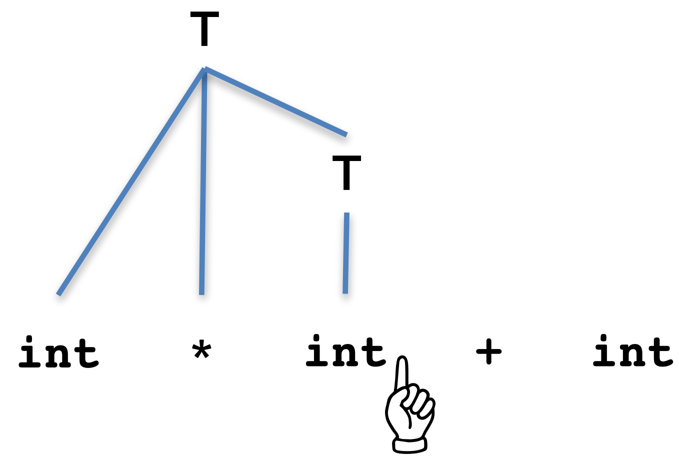
Shift/Reduce Parsing

```
| int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int
T | + int
```



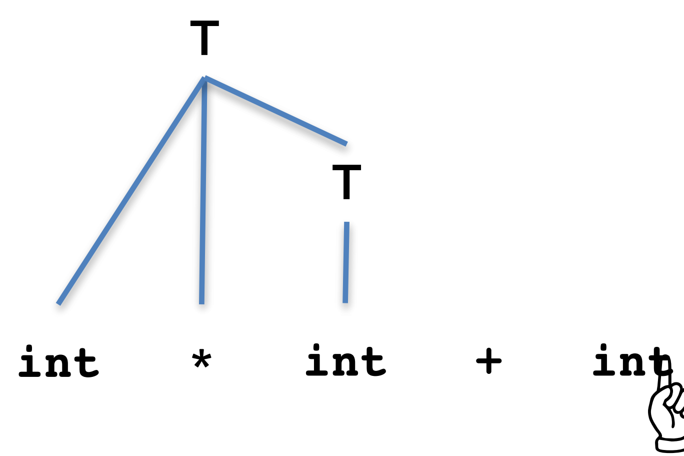
Shift/Reduce Parsing

```
| int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int
T | + int
```



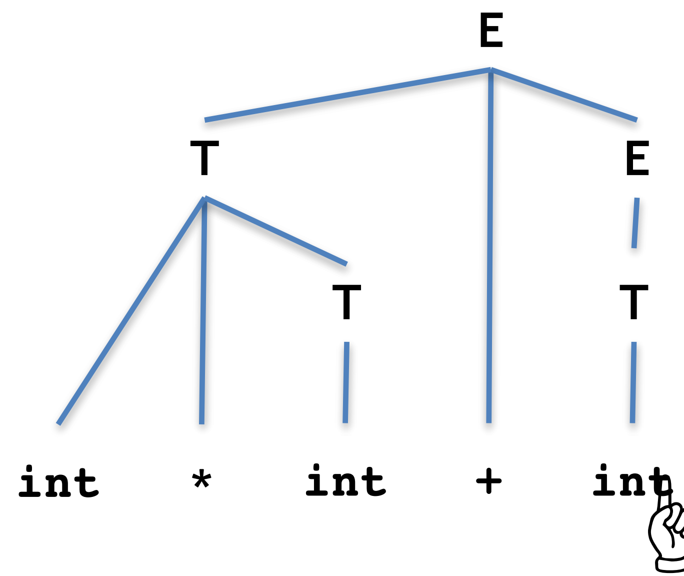
Shift/Reduce Parsing

```
| int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int
T | + int
T + | int
T + int |
```



Shift/Reduce Parsing

| int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int
T | + int
T + | int
T + int |
T + T |
T + E |
E |



IMPLEMENTING SHIFT/REDUCE

Shift/Reduce Parsing

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

- 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 $X \longmapsto \gamma$ 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: $E \longmapsto$ number
(E	+ 2 + (3 + 4)) + 5	reduce: $S \longmapsto E$
(S	+ 2 + (3 + 4)) + 5	shift +
(S +	2 + (3 + 4)) + 5	shift 2
(S + 2	+ (3 + 4)) + 5	reduce: $E \longmapsto$ 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.

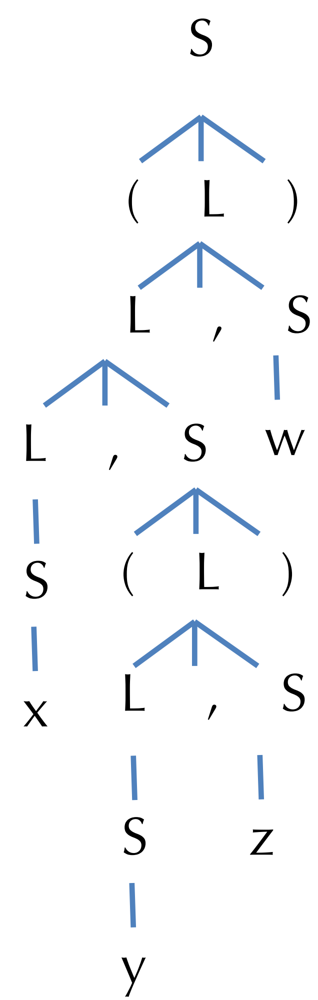
Example LR(0) Grammar: Tuples

- Example grammar for non-empty tuples and identifiers:

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

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

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



Example: Shift/Reduce Parsing of Tuples

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

- 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 $S \mapsto id$
(S	, (y, z), w)	reduce $L \mapsto S$
(L	, (y, z), w)	shift ,
(L,	(y, z), w)	shift (
(L, (	y, z), w)	shift y
(L, (y	, z), w)	reduce $S \mapsto id$
(L, (S	, z), w)	reduce $L \mapsto S$
(L, (L	, z), w)	shift ,
(L, (L,	z), w)	shift z
(L, (L, z	), w)	reduce $S \mapsto id$
(L, (L, S	), w)	reduce $L \mapsto L, S$
(L, (L	), w)	shift)
(L, (L)	, w)	reduce $S \mapsto (L)$
(L, S	, w)	reduce $L \mapsto L, S$
(L	, w)	shift ,
(L,	w)	shift w
(L, w	)	reduce $S \mapsto id$
(L, S	)	reduce $L \mapsto L, S$
(L	)	shift)
(L)		reduce $S \mapsto (L)$

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

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

- Example *items*: $S \longmapsto \bullet(\ L\ )$ or $S \longmapsto (\bullet\ L)$ or $L \longmapsto S\bullet$
- 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:
  $S' \longmapsto \bullet S\$$
- Closure of a state:
  - Adds items for all productions whose LHS nonterminal occurs in an item in the state just after the '$\bullet$'
  - The added items have the '$\bullet$' 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($\{S' \longmapsto \bullet S\$\}$) = $\{S' \longmapsto \bullet S\$,\ S \longmapsto \bullet(L),\ S \longmapsto \bullet id\}$

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

Example: Constructing the DFA

$S' \longmapsto .S\$$

$$\begin{aligned} S' &\longmapsto S\$ \\ S &\longmapsto (\ L\ )\ \mid\ \mathrm{id} \\ L &\longmapsto S\ \mid\ L\ ,\ S \end{aligned}$$

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

Example: Constructing the DFA

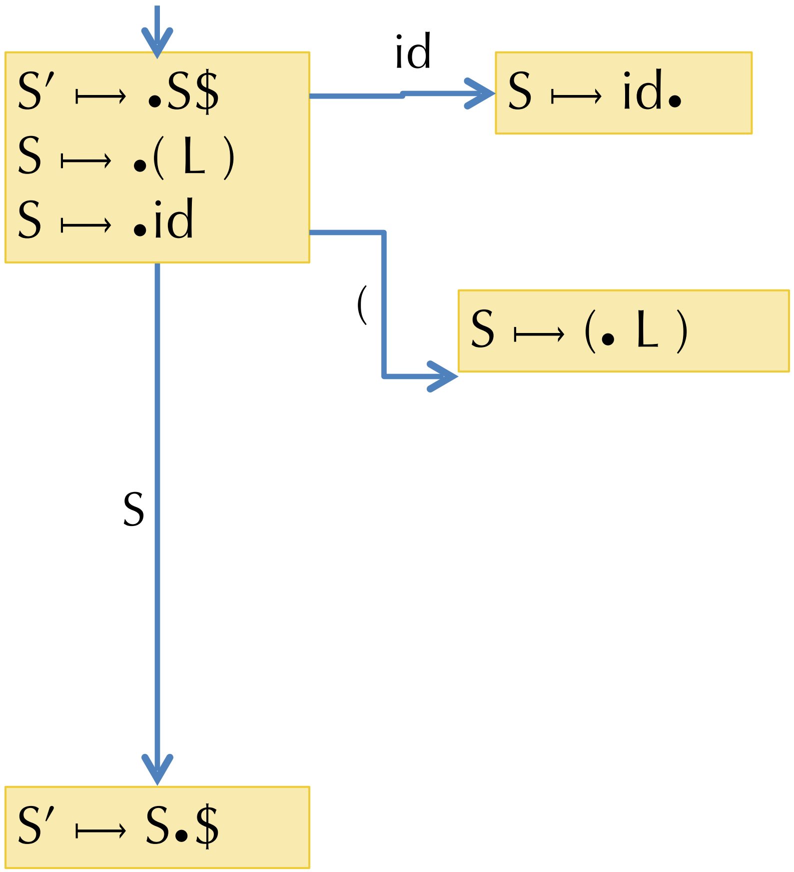
$$S' \longmapsto \bullet S\$$$
$$S \longmapsto \bullet(\ L\ )$$
$$S \longmapsto \bullet \mathrm{id}$$

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

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

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

Example: Constructing the DFA

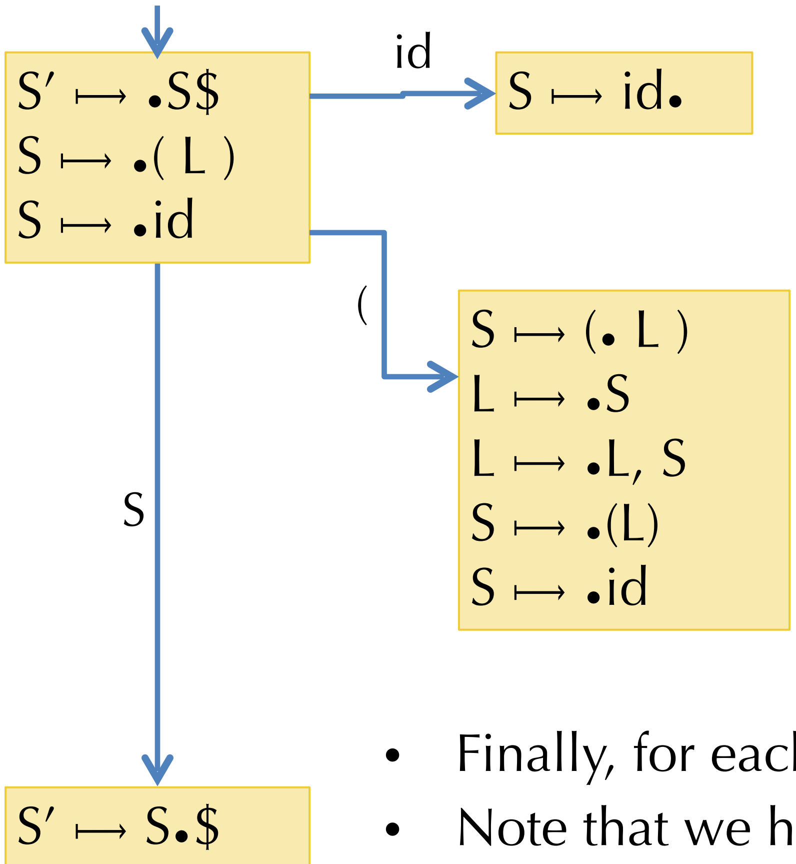


S′ ⟼ S$
S ⟼ (L) | id
L ⟼ S | L , S

- 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)

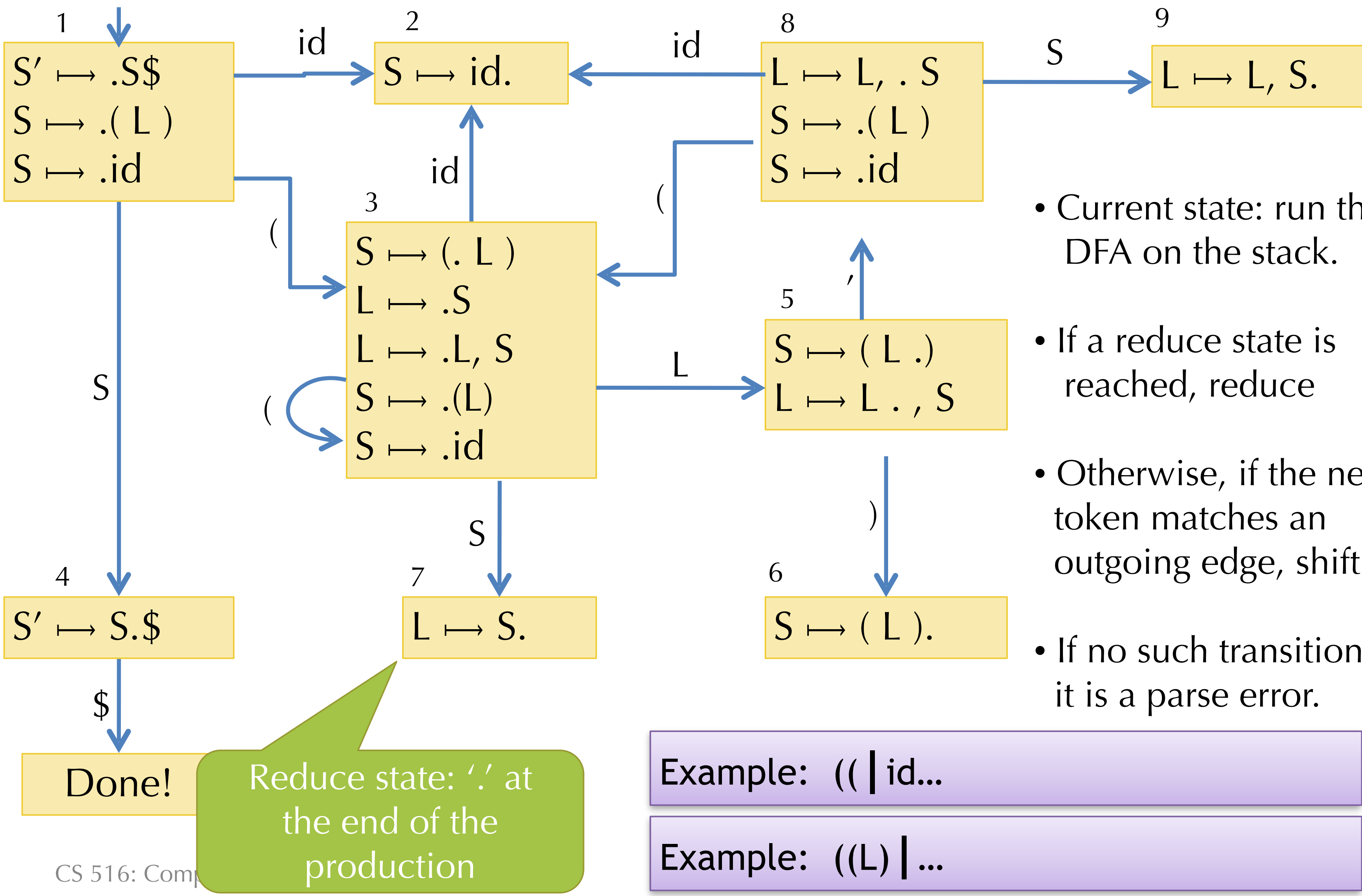
Example: Constructing the DFA

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



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

Full DFA for the Example

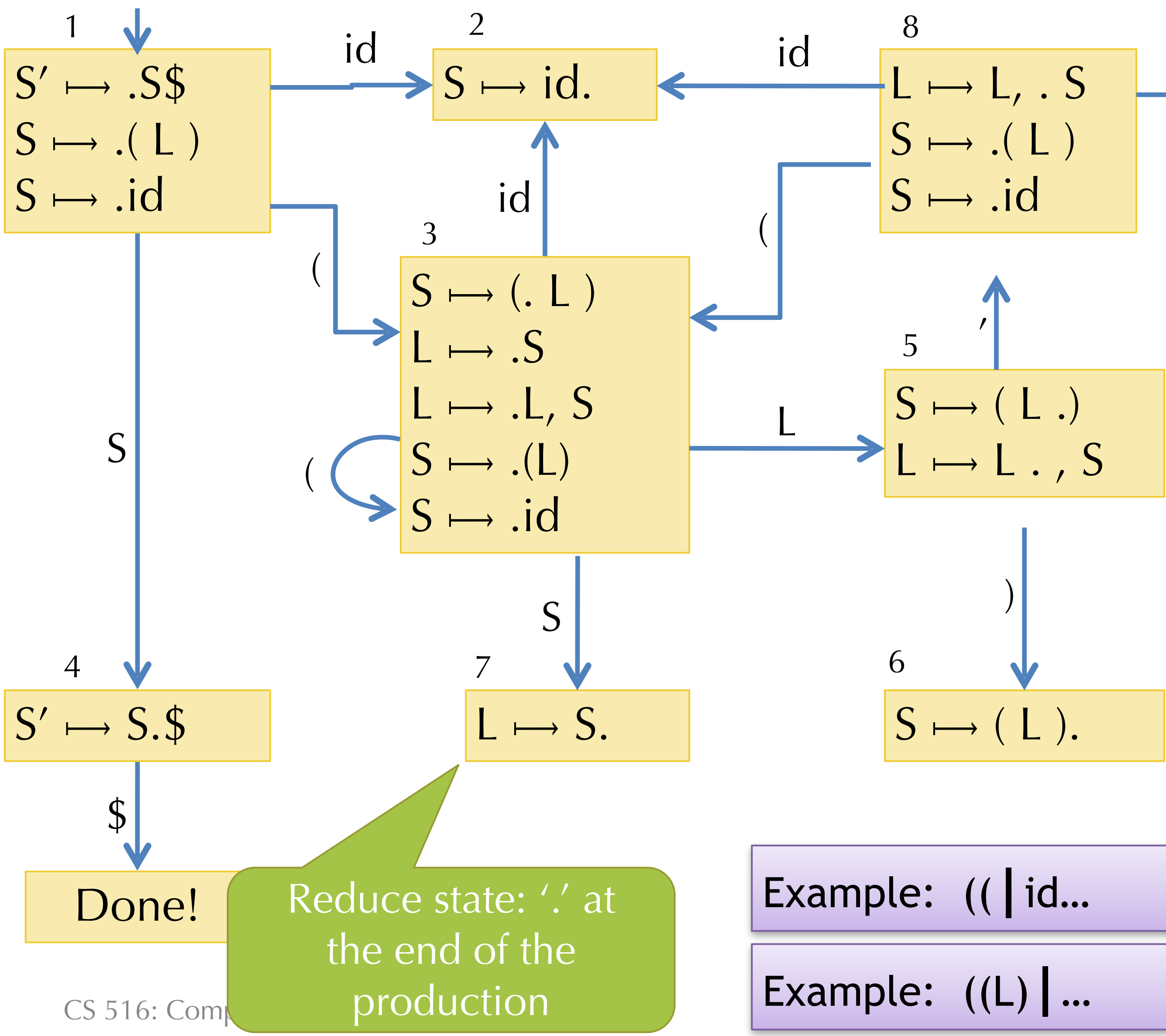


- Current state: run the DFA on the stack.
- If a reduce state is reached, reduce
- Otherwise, if the next token matches an outgoing edge, shift.
- If no such transition, it is a parse error.

Example: ((| id...

Example: ((L) | ...

Full DFA for the Example

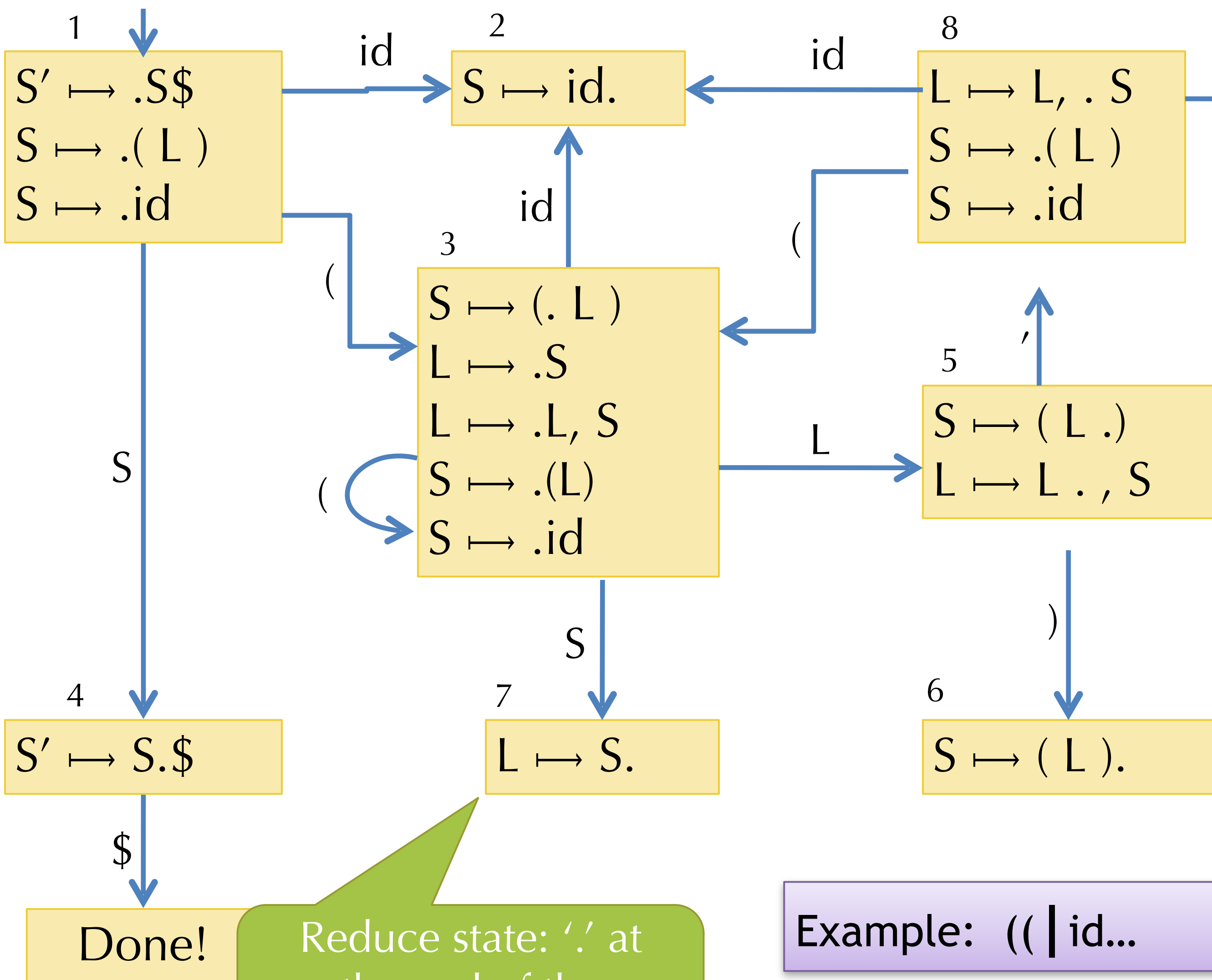


- Current state: run the DFA on the stack.
- If a reduce state is reached, reduce
- Otherwise, if the next token matches an outgoing edge, shift.
- If no such transition, it is a parse error.

Example: ((| id... **shift to:** ((id | ...

Example: ((L) | ...

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.

Example: ((| id... **shift to:** ((id | ...

Example: ((L) | ... **reduce to:** (S | ...

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.

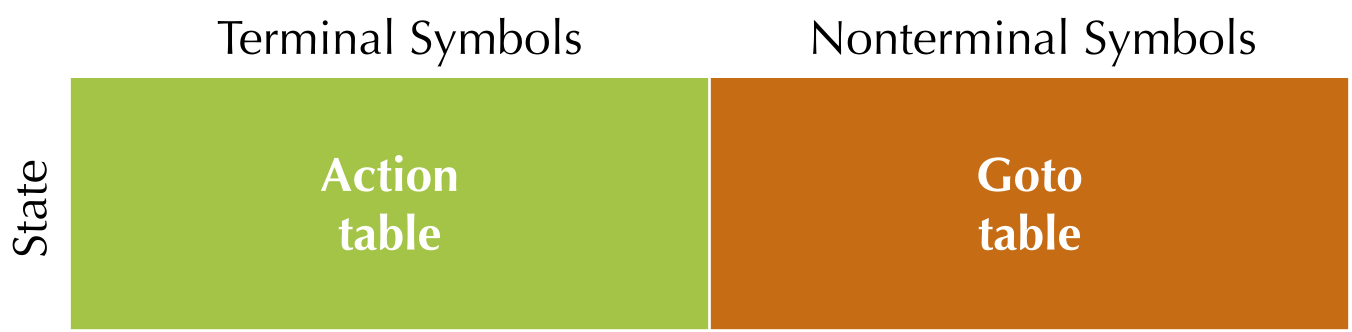
- **Optimization**: No need to re-run the DFA from beginning every step
  - Store the state with each symbol on the stack: e.g. ${}_1({}_3({}_3L_5)_6$
  - On a reduction $X \longmapsto \gamma$, pop stack to reveal the state too:
    e.g. From stack ${}_1({}_3({}_3L_5)_6$ reduce $S \longmapsto$ (L) to 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: $S \longmapsto id$
$\varepsilon_1(_3S$	, (y, z), w)$	g7 (from state 3 follow S)
$\varepsilon_1(_3S_7$	, (y, z), w)$	Reduce: $L \longmapsto S$
$\varepsilon_1(_3L$	, (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

	(	)	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		

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

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

right

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

- 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.

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

SLR(1): Simple LR(1) Parsers

- 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:

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

- 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. (*Propagate*) If δ is or can derive ε (i.e. it is nullable), then the look-ahead $\mathcal{M}$ also contains $\mathcal{L}$

Example Closure

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

Example Closure

$$S' \longmapsto S\$$$
$$S \longmapsto E + S \;|\; E$$
$$E \longmapsto \text{number} \;|\; (\,S\,)$$

- Start item: $S' \longmapsto .S\$ \quad , \quad \{\}$

Example 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: {$} is FIRST($)
  $S \longmapsto .E \quad , \quad \{\$\}$

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:

  $S \longmapsto .E + S$, {\$}
  $S \longmapsto .E$, {\$}

Note: {\$} is FIRST(\$)

Example 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
  - $E \longmapsto .(S) \quad , \quad \{+\}$ — $\text{FIRST}(+\ S) = \{+\}$

Example 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) = {+}

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

Example 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':

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

  $E \longmapsto .(S) \quad , \quad \{+\}$ FIRST(+ S) = {+}
- Because E also appears to the right of '.' in '.E' we get:

  $E \longmapsto .\text{number} \ , \quad \{\$\}$ Note: \$ added for reason 2

  $E \longmapsto .(S) \quad , \quad \{\$\}$ δ is ε

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

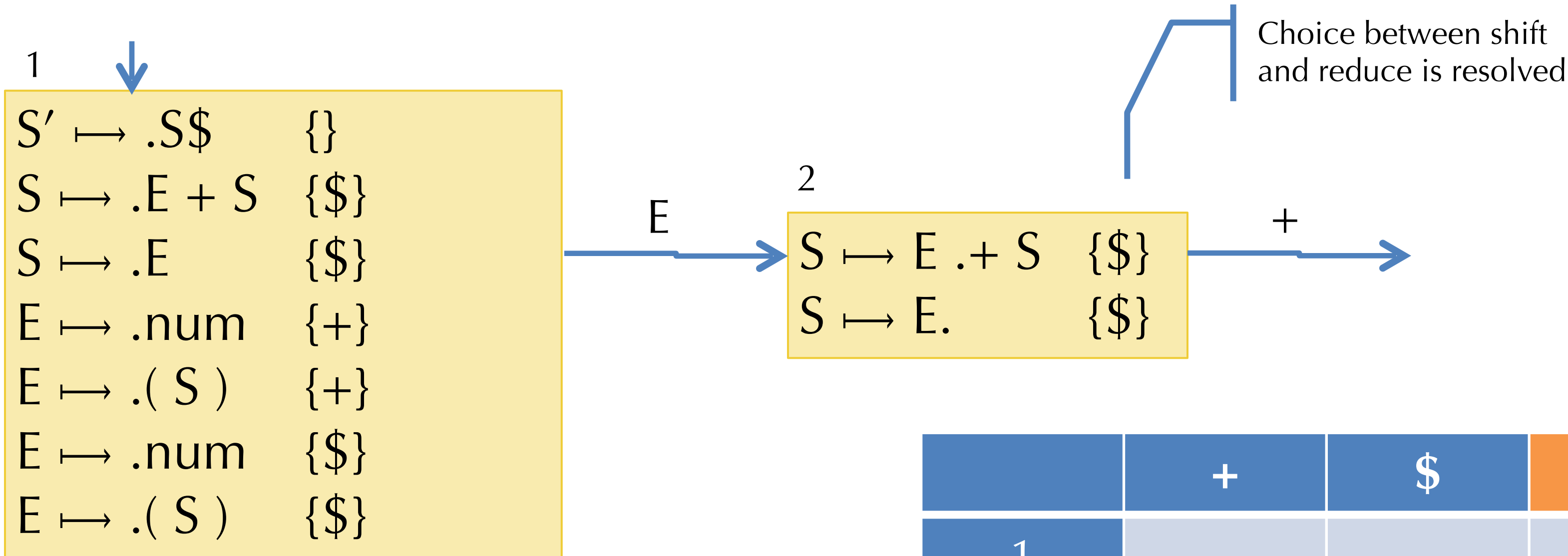
Example Closure

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

- 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

Propagate

Using the DFA



	+	$	E
1			g2
2	s3	$S \longmapsto E$	

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

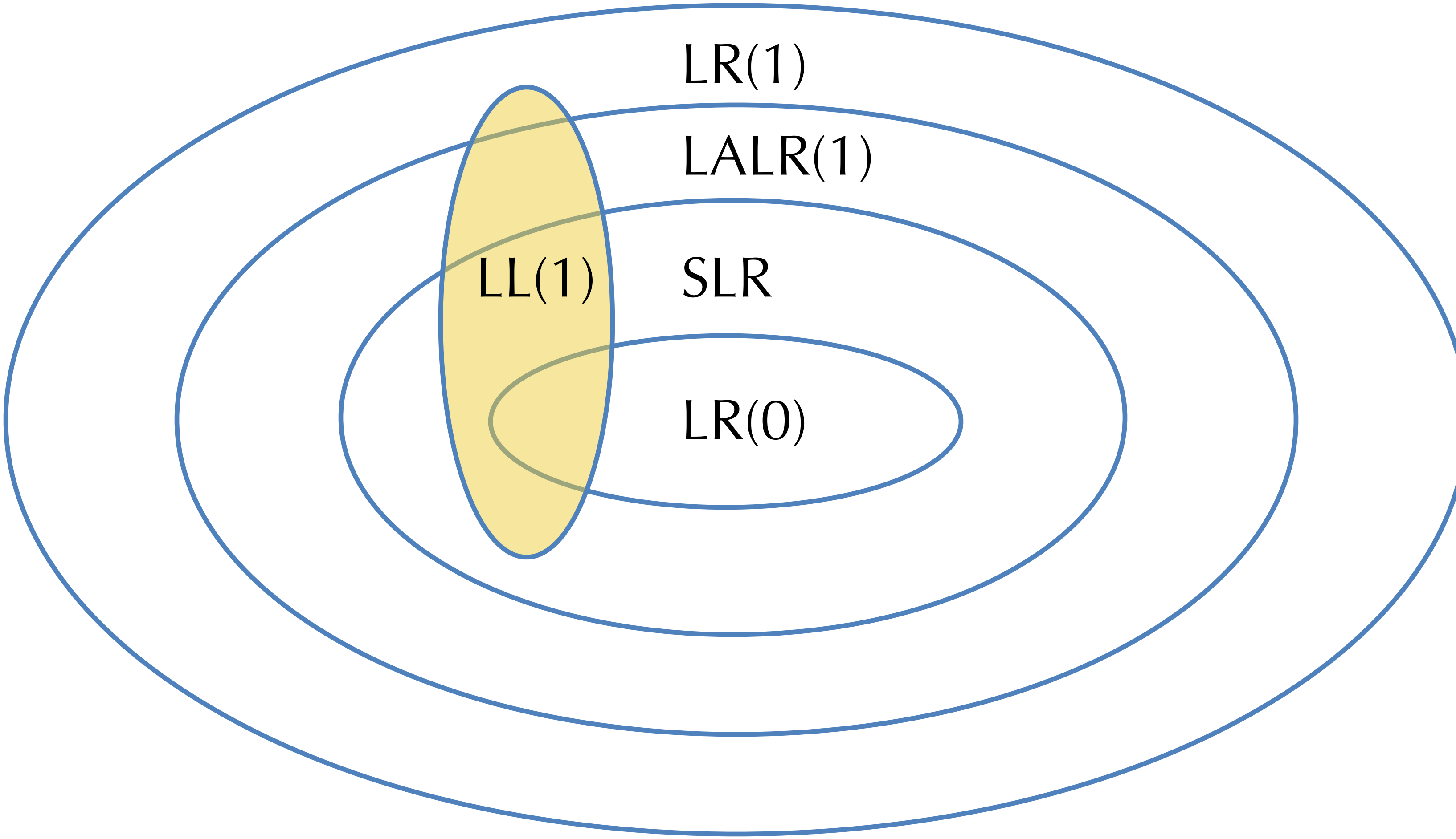
- LR(1) gives maximal power out of a 1 look-ahead symbol parsing table
  - DFA + stack is a push-down automaton
- In practice, LR(1) tables are big.
  - Modern implementations (e.g. menhir) directly generate code
- LALR(1) = "Look-ahead LR"
  - Merge any two LR(1) states whose items are identical except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

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

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

  - 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:
  - `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 hand-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 $\longmapsto$ `if (E)` S
  S $\longmapsto$ `if (E)` S `else` S
  S $\longmapsto$ X = E
  E $\longmapsto$ …

- Is this grammar OK?

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:

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

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

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

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.

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?

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?

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:

$$\texttt{if (E}_1\texttt{)} \left[\texttt{if (E}_2\texttt{) S}_1 \right] \texttt{else S}_2$$

- 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:

  `if (`E_1`) { if (`E_2`) { `S_1` } } else `S_2 // unambiguous

  `if (`E_1`) { if (`E_2`) { `S_1` } else `S_2` }` // 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