

# CS536

## Bottom-Up Parsing

# Roadmap

- Last class
  - Name analysis
- Previous-ish last class
  - LL(1)
- Today's class
  - LR Parsing
    - SLR(1)

# Lecture Outline

- Introduce Bottom-Up parsing much like Top-Down
  - Talk about the language class / theory
  - Describe the state that it keeps / intuition
  - Show how it works
  - Show how it is built

# LL(1) Not Powerful Enough for all PL

- Left-recursion
- Not left factored
- Doesn't mean LL(1) is bad
  - Right tool for simple parsing jobs



```
stmtList ::= stmtList stmt
          | /* epsilon */
          ;
```

# We Need a *Little* More Power

- Could increase the lookahead
  - Up until the mid 90s, this was considered impractical
- Could increase the runtime complexity
  - CYK has us covered there
- Could increase the memory complexity
  - i.e. more elaborate parse table

# LR Parsers

- Left-to-right scan of the input file
- Reverse rightmost derivation
- Advantages
  - Can recognize almost any programming language
  - Time and space  $O(n)$  in the input size
  - More powerful than the corresponding LL parser i.e.  $LL(1) < LR(1)$
- Disadvantages
  - More complex parser generation
  - Larger parse tables

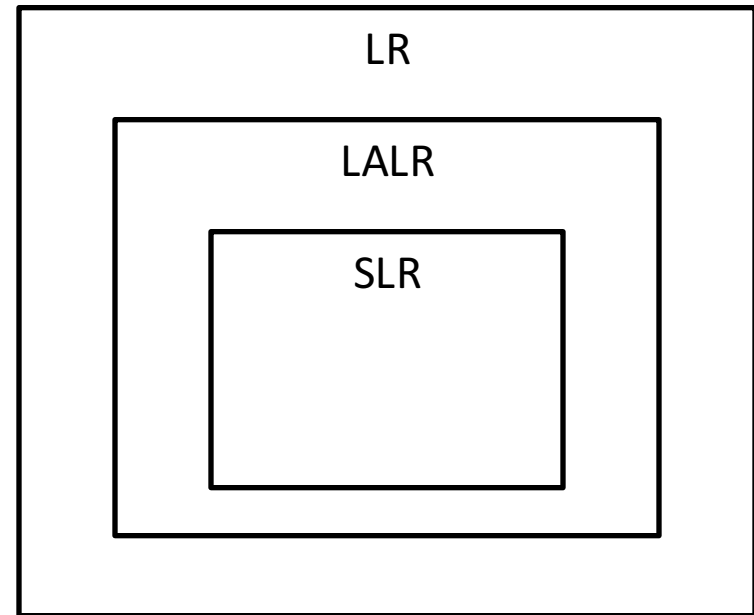
# LR Parser Power

- Let  $S \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \omega$  be a rightmost derivation, where  $\omega$  is a terminal string
- Let  $\alpha A \gamma \Rightarrow \alpha \beta \gamma$  be a step in the derivation
  - So  $A \rightarrow \beta$  must have been a production in the grammar
  - $\alpha \beta \gamma$  must be some  $\alpha_i$  or  $\omega$
  - A grammar is LR(k) if for every derivation step,  $A \rightarrow B$  can be inferred using only a scan of  $\alpha \beta$  and at most k symbols of  $\gamma$
- Much like LL(1), you generally just have to go ahead and try it

# LR Parser types

Recognizable by a  
deterministic PDA

- LR(1)
  - Can recognize any DCFG
  - Can experience blowup in parse table size
- LALR(1)
- SLR(1)
  - Both proposed at the same time to limit parse table size





# Which parser should we use?

- Different variants mostly differ in how they build the parse table, we can still talk about all the family in general terms
  - Today we'll cover SLR
  - Pretty easy to learn LALR from there
- LALR(1)
  - Generally considered a good compromise between parse table size and expressiveness
  - Class for Java CUP, yacc, and bison

# How does Bottom-up Parsing work?

- Already seen 1 such parser: CYK
  - Simultaneously tracked every possible parse tree
    - LR parsers work in a similar same way
- Contrast this to top-down parser
  - We know exactly where we are in the parse
  - Make predictions about what's next

# Parser State

- Top-down parser state

- Current token
- Stack of symbols
  - Represented what we expect in the rest of our descent to the leaves
- Worked down and to the left through tree

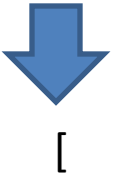
## Grammar

$$\begin{array}{lcl} S & ::= & \epsilon \\ & | & ( S ) \\ & | & [ S ] \end{array}$$

## Stack



## Current



- Bottom-up state

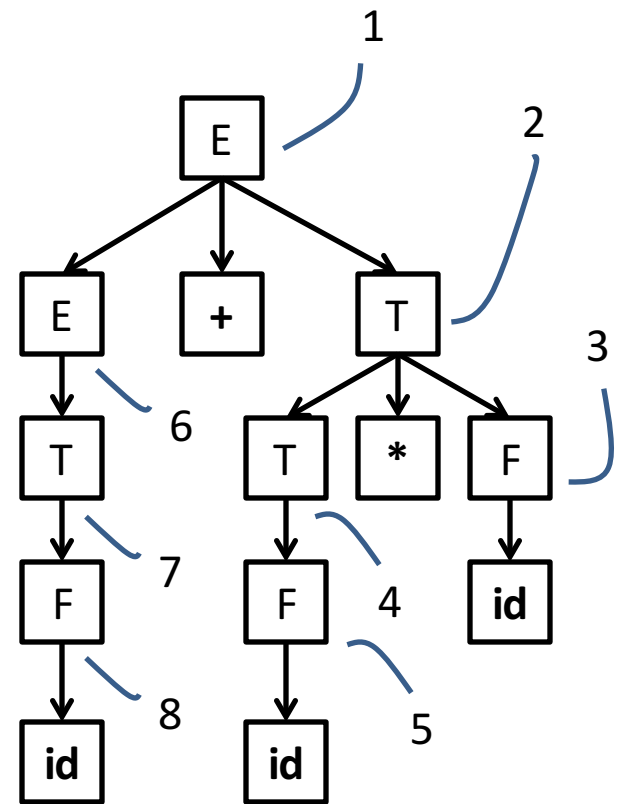
- Also maintains a stack and token
  - Represents summary of input we've seen
- Works upward and to the right through the tree
- Also have an auxiliary state machine to help disambiguate rules

# LR Derivation Order

- Let's remember derivation orders again

Reverse Rightmost derivation

8	1	$E \Rightarrow E + T$
7	2	$\Rightarrow E + T * F$
6	3	$\Rightarrow E + T * id$
5	4	$\Rightarrow E + F * id$
4	5	$\Rightarrow E + id * id$
3	6	$\Rightarrow T + id * id$
2	7	$\Rightarrow F + id * id$
1	8	$\Rightarrow id + id * id$

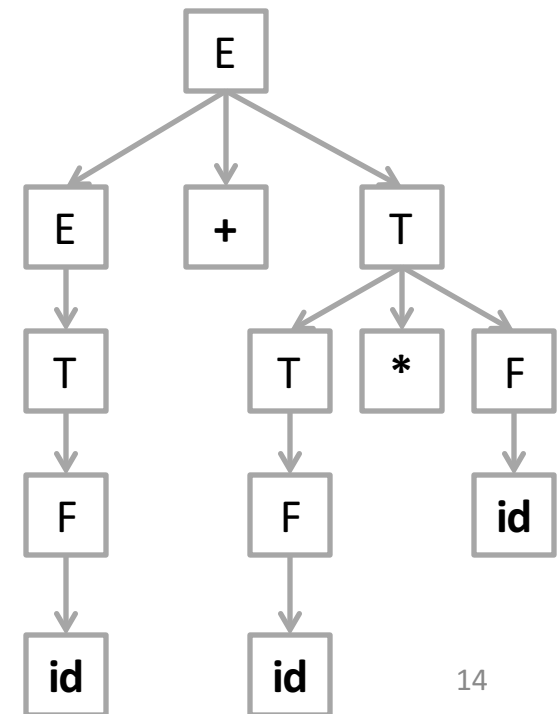


# Parser Operations

- Top-down parser
  - *Scan* the next input token
  - *Push* a bunch of RHS symbols
  - *Pop* a single symbol
- Bottom-up parser
  - *Shift* an input token into a stack item
  - *Reduce* a bunch of stack items into a new parent item (on the stack)

# Parser Actions: Simplified view

<u>Stack</u>	<u>Input</u>	<u>Action</u>
	id + id * id EOF	shift(id)
id	+ id * id EOF	reduce by $F \rightarrow id$
F	+ id * id EOF	reduce by $T \rightarrow F$
T	+ id * id EOF	reduce by $E \rightarrow T$
E	+ id * id EOF	shift +
E +	id * id EOF	shift id
E + id	* id EOF	reduce by $F \rightarrow id$
E + F	* id EOF	reduce by $T \rightarrow F$
E + T	* id EOF	shift *
E + T *	id EOF	shift id
E + T * id	EOF	reduce by $F \rightarrow id$
E + T * F	EOF	reduce by $T \rightarrow T * F$
E + T	EOF	reduce by $E \rightarrow E + T$
E	EOF	accept



# Stack Items

- Note that the previous slide was called “simplified”
- Stack elements are representative of symbols
  - Actually known as items
    - Indicate a production and a position within the production

$$X \rightarrow \alpha . B \beta$$

- Means
  - we are in a production of X
  - We believe we’ve parsed (arbitrary) symbol string  $\alpha$
  - We could handle a production of B
  - After that we’ll have  $\beta$

# Stack Item Examples

- Example 1

$PList \rightarrow ( . IDList )$

- Example 2

$PList \rightarrow ( IDList . )$

- Example 3

$PList \rightarrow ( IDList ) .$

- Example 4

$PList \rightarrow . ( IDList )$



# Stack Item State

- You may not know exactly which item you are parsing
- LR Parsers actually track the set of states that you *could* have been in

## Grammar snippet

$S \rightarrow A$

$A \rightarrow B$

$| C$

$B \rightarrow D \text{ id}$

$C \rightarrow \text{id } E$

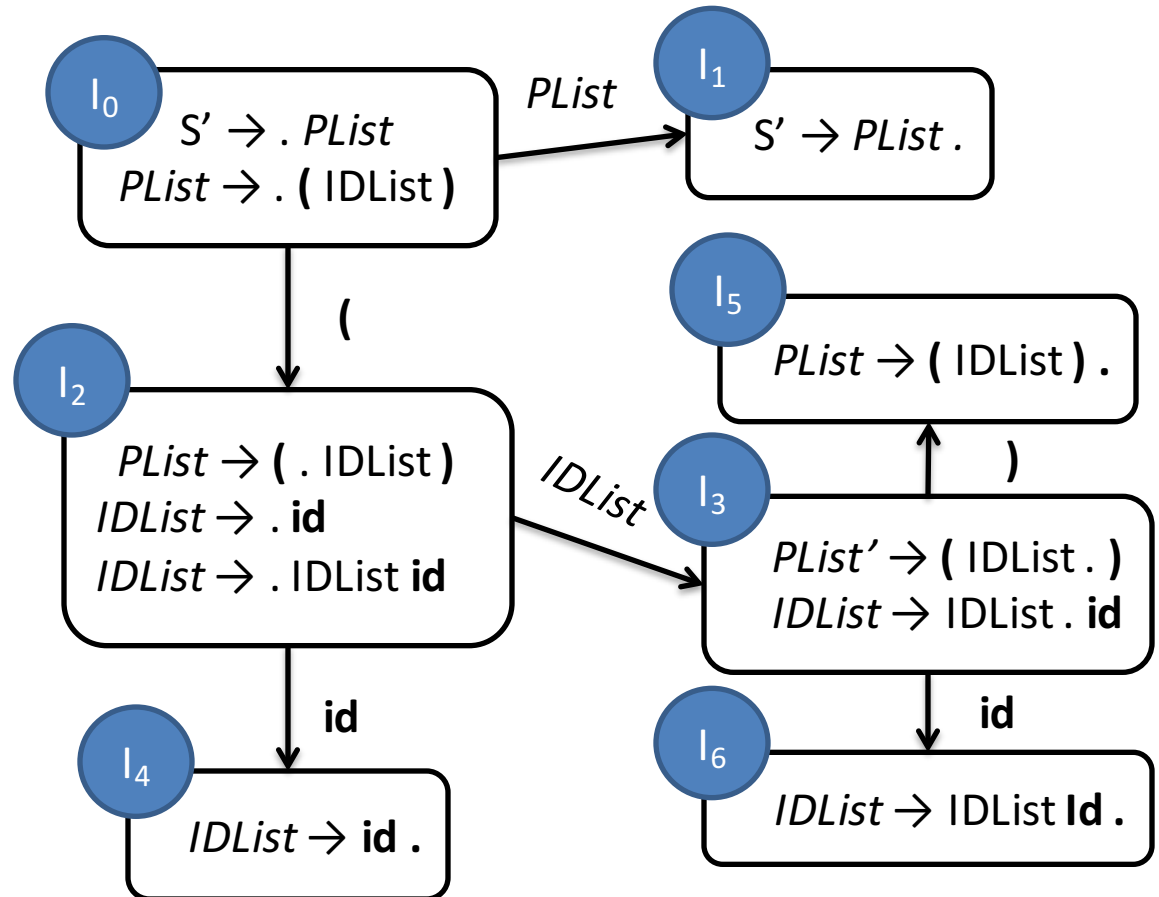
$D \rightarrow \text{id } E$

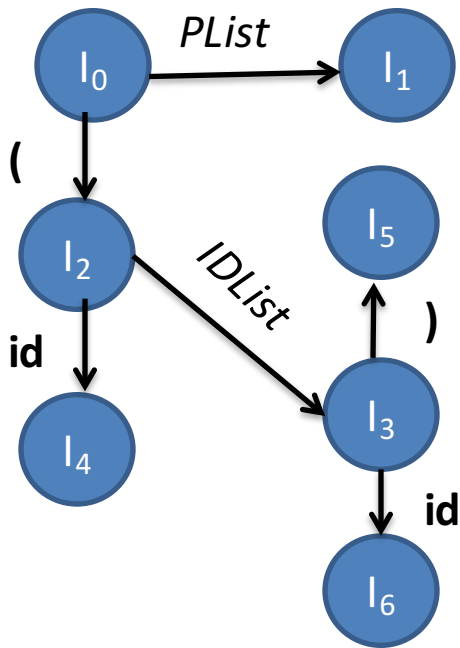
$\{S \rightarrow . A, A \rightarrow . B, A \rightarrow . C, \dots\}$

# LR Parser FSM

## Grammar G

$S' \rightarrow PList$   
 $PList \rightarrow ( IDList )$   
 $IDList \rightarrow id$   
 $IDList \rightarrow IDList id$





# Automaton as a table

- *Shift* corresponds to taking a terminal edge
- *Reduce* corresponds to taking a nonterminal edge

**Action table**

	(	)	id	eof
0	S 2			
1				
2			S 4	
3		S 5	S 6	
4				
5				
6				

**GoTo table**

<i>PList</i>	<i>IDList</i>
1	
	3

# How do we know to reduce?

Action table					GoTo table	
	(	)	id	eof	<i>PList</i>	<i>IDList</i>
0	S 2				1	
1						
2			S 4			3
3		S 5	S 6			
4		R ③	R ③			
5				R ②		
6		R ④	R ④			

## Grammar G

- ①  $S' \rightarrow PList$
- ②  $PList \rightarrow ( IDList )$
- ③  $IDList \rightarrow id$
- ④  $IDList \rightarrow IDList id$

- Only see terminals in the input
- Actually do reduce steps in 2 phases
  - Action table will tell us when to reduce (and how much)
  - GoTo will tell us where to... go to

# How do we know we're done?

Action table					GoTo table	
	(	)	id	eof	<i>PList</i>	<i>IDList</i>
0	S 2				1	
1				☺		
2			S 4			3
3		S 5	S 6			
4		R ③	R ③			
5				R ②		
6		R ④	R ④			

- Add an accept token
- Any other cell is an error

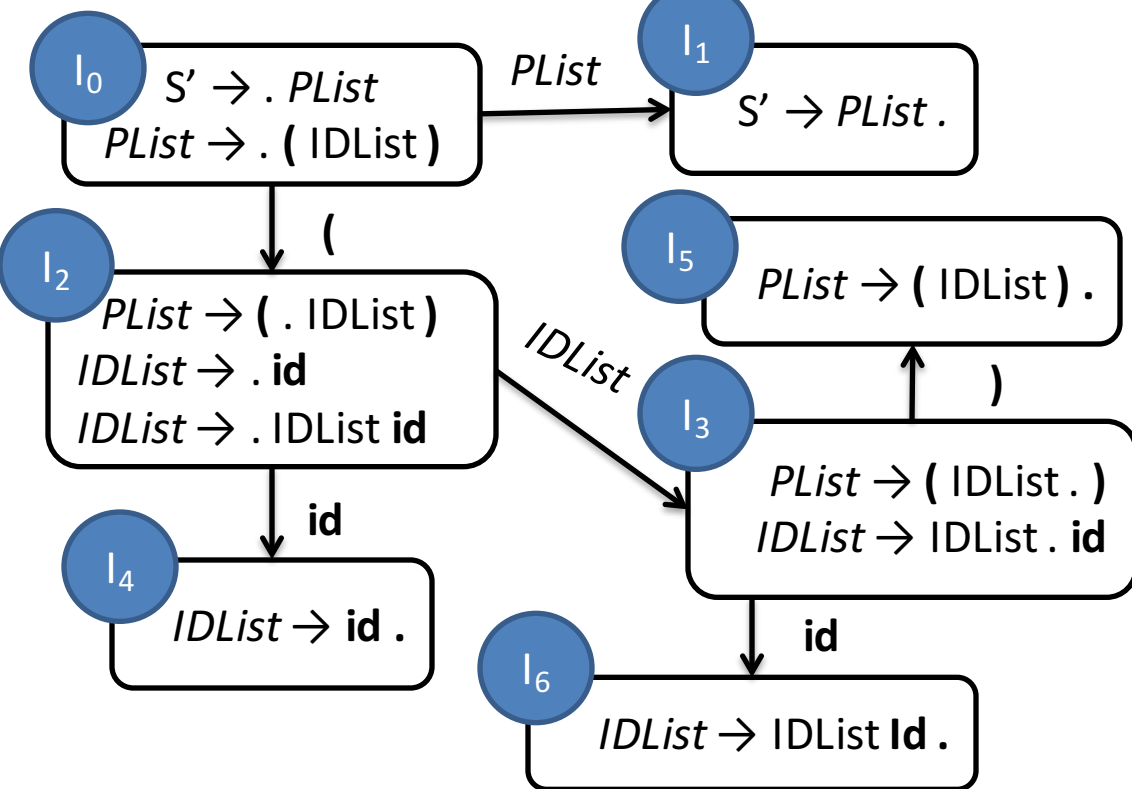
## Grammar G

- ①  $S' \rightarrow PList$
- ②  $PList \rightarrow ( IDList )$
- ③  $IDList \rightarrow id$
- ④  $IDList \rightarrow IDList id$

# Full Parse Table Operation

```
Initialize stack
a = scan()
do forever
    t = top-of-stack (state) symbol
    switch action[t, a] {
        case shift s:
            push(s)
            a = scan()
        case reduce by  $A \rightarrow \alpha$ :
            for i = 1 to length(alpha) do pop() end
            t = top-of-stack symbol
            push(goto[t, A])
        case accept:
            return( SUCCESS )
        case error:
            call the error handler
            return( FAILURE )
    }
end do
```

# Example Time



current item  
 current item  
 current item  
 current item  
 current item  
 ( id id id ) eof

### Grammar G

- 1  $S' \rightarrow PList$
- 2  $PList \rightarrow ( IDList )$
- 3  $IDList \rightarrow id$
- 4  $IDList \rightarrow IDList id$

[ $I_5$ ]  
 [ $I_3$ ]  
 [ $I_1$ ]  
 [ $I_0$ ]

	(	)	id	eof	PList	IDList
0	S 2				1	
1				☺		
2			S 4			3
3		S 5	S 6			
4		R 3	R 3			
5				R 2		
6		R 4	R 4			



Seems that LR Parser works pretty great. What could possibly go wrong?

# LR Parser State Explosion

- Tracking sets of states can cause the size of the FSM to blow up
- The SLR and LALR variants exist to combat this explosion
- Slight modification to item and table form



# Building the SLR Automaton

- Uses 2 sets
  - Closure(I)
    - What is the set of items we could be in?
    - Given I: what is the set of items that could be mistaken for I (reflexive)
  - Goto(I,X)
    - If we are in state I, where might we be after parsing X?
- Vaguely reminiscent of FIRST and FOLLOW

# Closure Sets

Put  $I$  itself into  $\text{Closure}(I)$

While there exists an item in  $\text{Closure}(I)$  of the form

$$X \rightarrow \alpha . B \beta$$

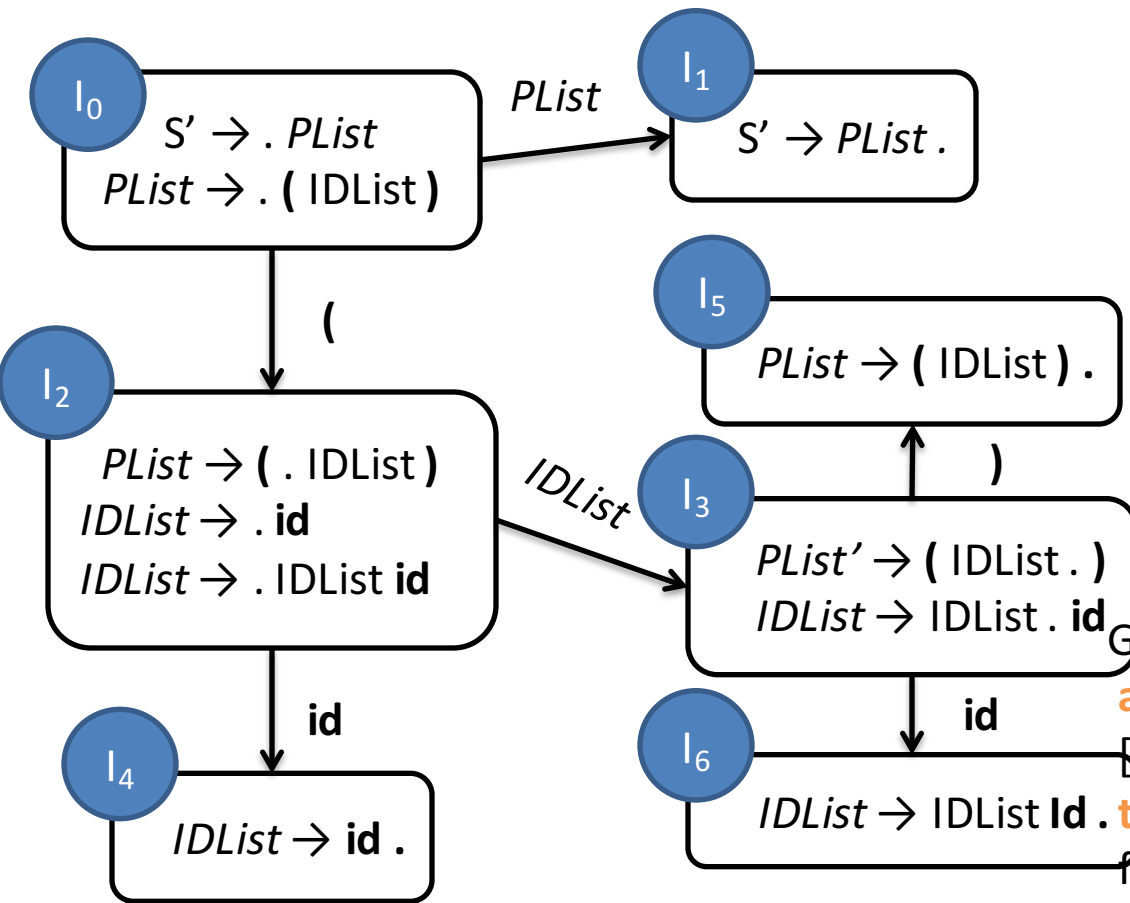
such that there is a production  $B \rightarrow \gamma$ ,

and  $B \rightarrow . \gamma$  is not in  $\text{Closure}(I)$

add  $B \rightarrow . \gamma$  to  $\text{Closure}(I)$

# GoTo Sets

$$\text{Goto}(I, X) = \text{Closure}(\{ A \longrightarrow \alpha X . B \mid A \longrightarrow \alpha . X \beta \text{ is in } I \})$$



## Grammar G

$S' \rightarrow PList$

$PList \rightarrow ( IDList )$

$IDList \rightarrow id$

$IDList \rightarrow IDList id$

## GoTo (LR(0))

For each state  $I$ , compute

Repeat for  $X \in \{ (, id, ) \}$  s.t.  $A \rightarrow \alpha.X\beta \in I$

$X \rightarrow \alpha.B\beta \in \text{Closure}(I)$  s.t.

$\exists B \rightarrow \gamma$ , add  $B \rightarrow \cdot \gamma$  to  $\text{Closure}(I)$

GoTo( $I, X$ ) =  $\text{Closure}(I \rightarrow \alpha.X\beta)$

all items  $A \rightarrow \alpha.X\beta$  s.t.  $A \rightarrow \alpha.X\beta \in I$

all items  $A \rightarrow \alpha.IDList.\beta$

those where  $A \rightarrow \alpha.IDList.\beta \in I_0$

for  $I_1$   $\{ PList \rightarrow ( \cdot IDList ) \}$

for  $I_2$   $\{ IDList \rightarrow \cdot id, IDList \rightarrow \cdot IDList id \}$

for  $I_3$   $\{ PList' \rightarrow ( IDList \cdot ) \}$

for  $I_4$   $\{ IDList \rightarrow id \cdot \}$

for  $I_5$   $\{ PList \rightarrow ( IDList ) \cdot \}$

for  $I_6$   $\{ IDList \rightarrow IDList id \cdot \}$

Items added where  $IDList \rightarrow \gamma \in G$

set to closure is

$\{ IDList \rightarrow \cdot IDList \}$

$\{ PList \rightarrow ( \cdot IDList ) \}$

$\{ IDList \rightarrow IDList \cdot \}$

## Parse Table Construction

1: Add new start  $S'$  and  $S' \rightarrow S$

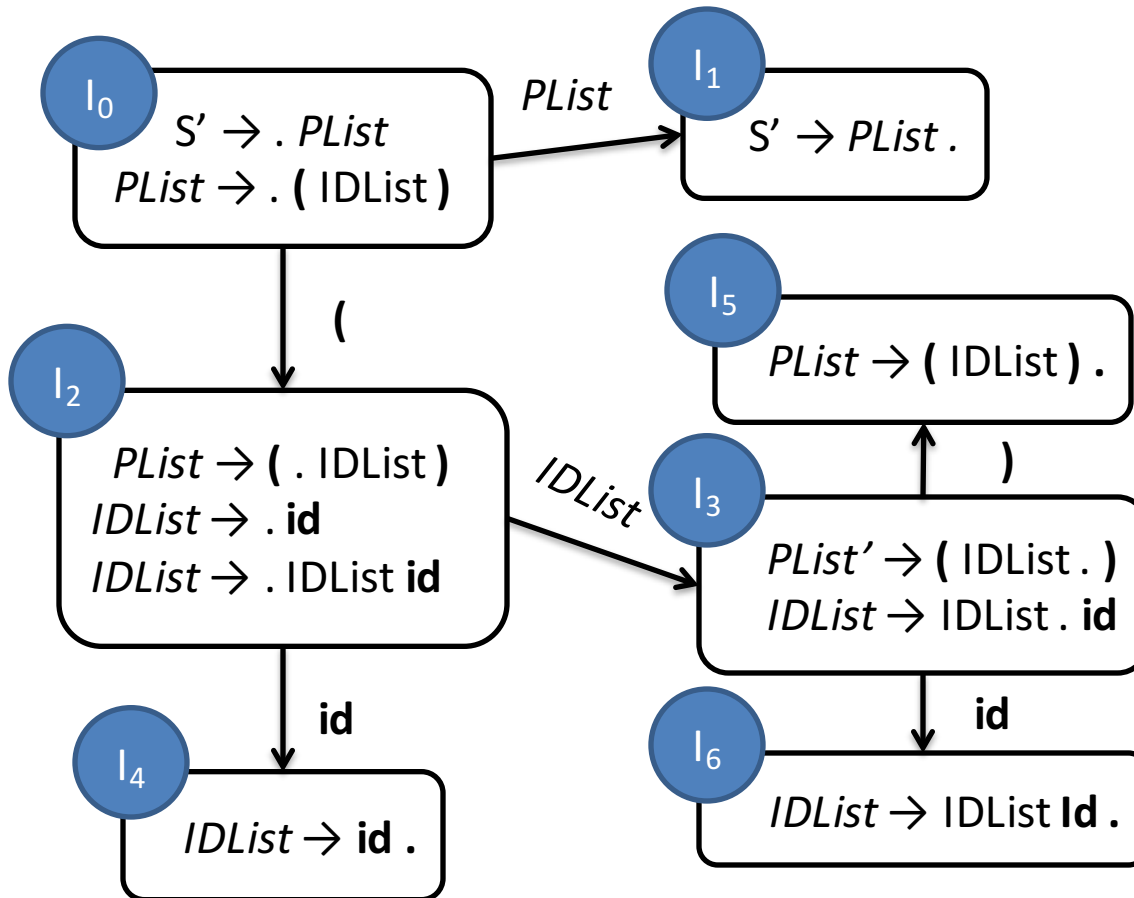
2: Build State  $I_0$  for  $\text{Closure}(\{S' \rightarrow \cdot S\})$

3: Saturate FSM:

for each symbol  $X$  s.t. there is a item in state  $j$  containing  $\cdot X$

add transition from state  $j$  to state for  $\text{GoTo}(j, X)$

# From FSM to parse table(s)

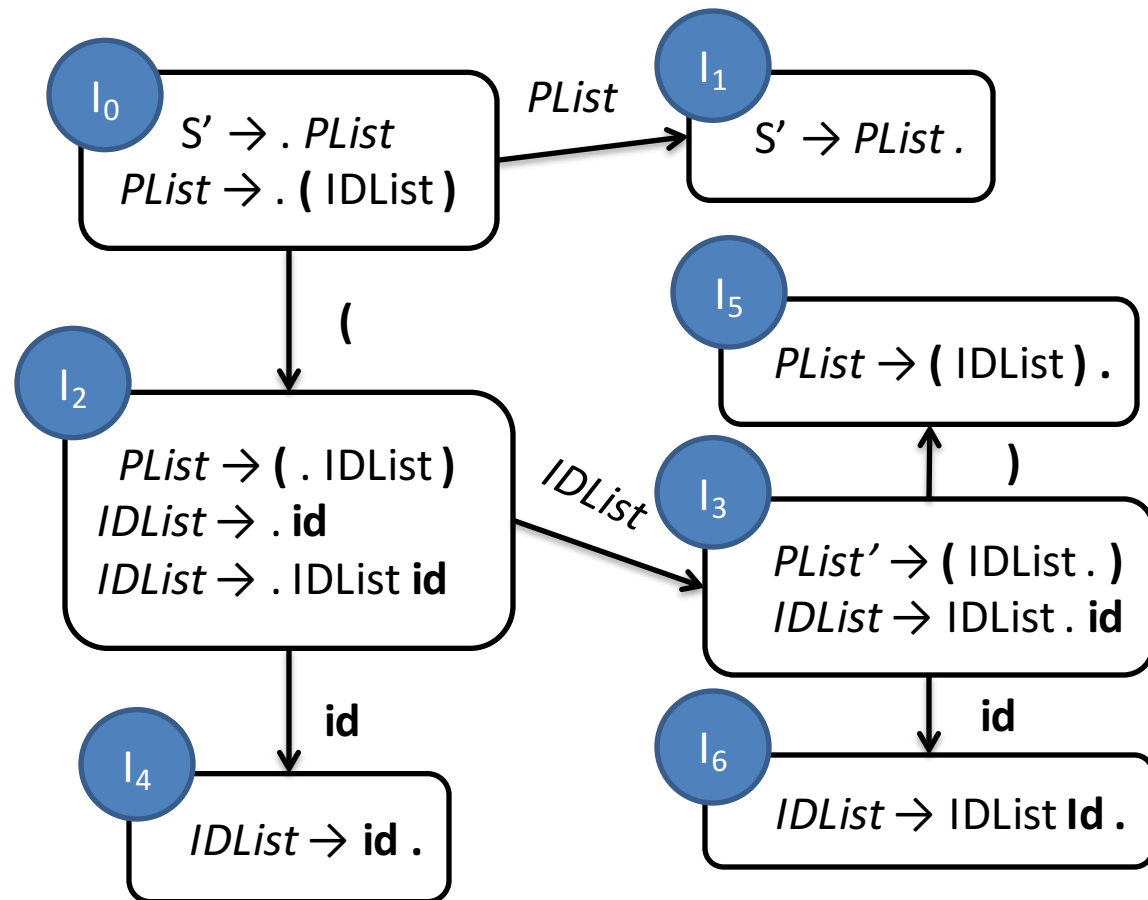


Need to connect the FSM back to the grammar

## Grammar G

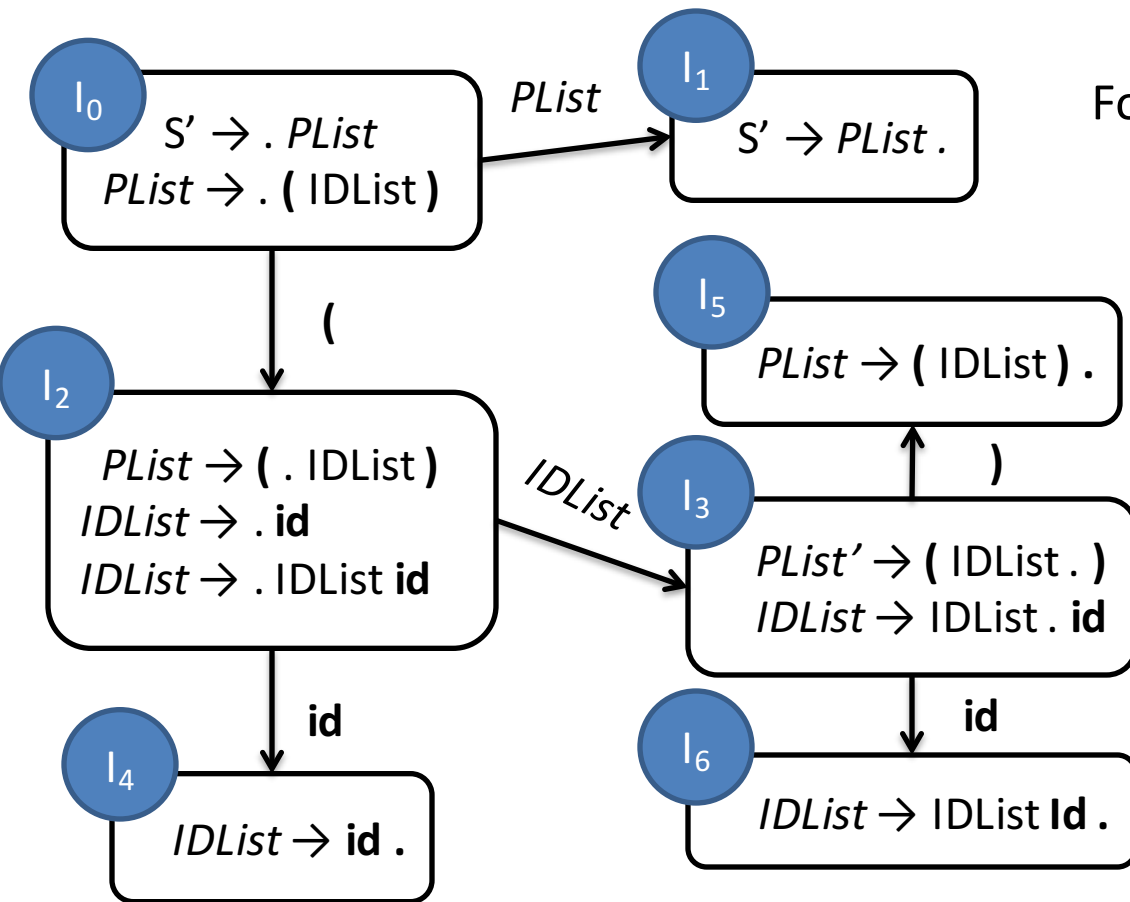
- ①  $S' \rightarrow PList$
- ②  $PList \rightarrow ( IDList )$
- ③  $IDList \rightarrow id$
- ④  $IDList \rightarrow IDList id$

# Can Now Build Action and GoTo Tables





# Building the GoTo Table



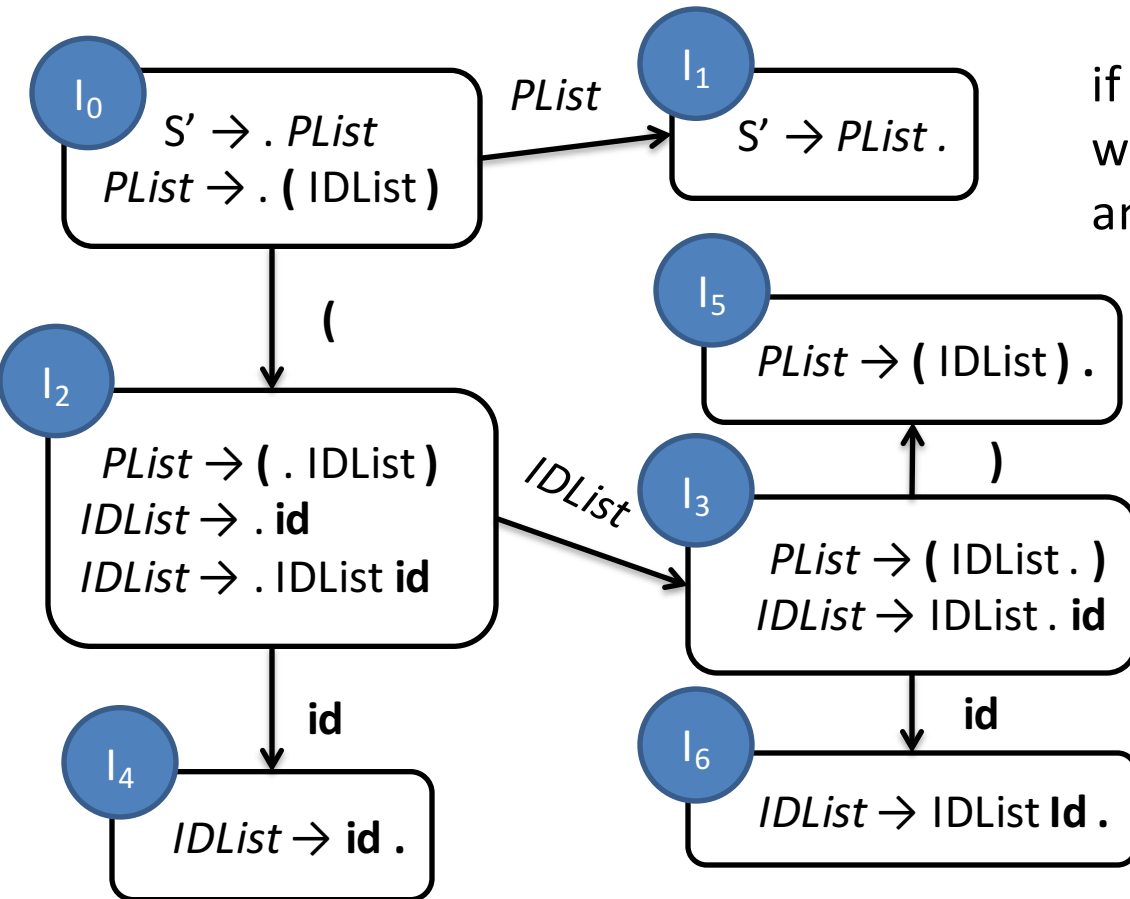
For every nonterminal  $X$   
if there is an  $(i,j)$  edge on  $X$   
set  $GoTo[i,X] = j$

	<i>PList</i>	<i>IDList</i>
0	1	
1		
2		3
3		
4		
5		
6		

# Building the Action Table

- if state  $i$  includes item  $A \rightarrow \alpha . \mathbf{t} \beta$   
where  $\mathbf{t}$  is a terminal  
and there is an  $(i,j)$  transition on  $\mathbf{t}$   
set  $\text{Action}[i,\mathbf{t}] = \text{shift } j$
- If state  $i$  includes item  $A \rightarrow \alpha .$   
where  $A$  is not  $S'$   
for each  $t$  in  $\text{FOLLOW}(A)$ :  
set  $\text{Action}[i,\mathbf{t}] = \text{reduce by } A \rightarrow \alpha$
- if state  $i$  includes item  $S \rightarrow S .$   
set  $\text{Action}[i, \mathbf{eof}] = \text{accept}$
- All other entries are error actions

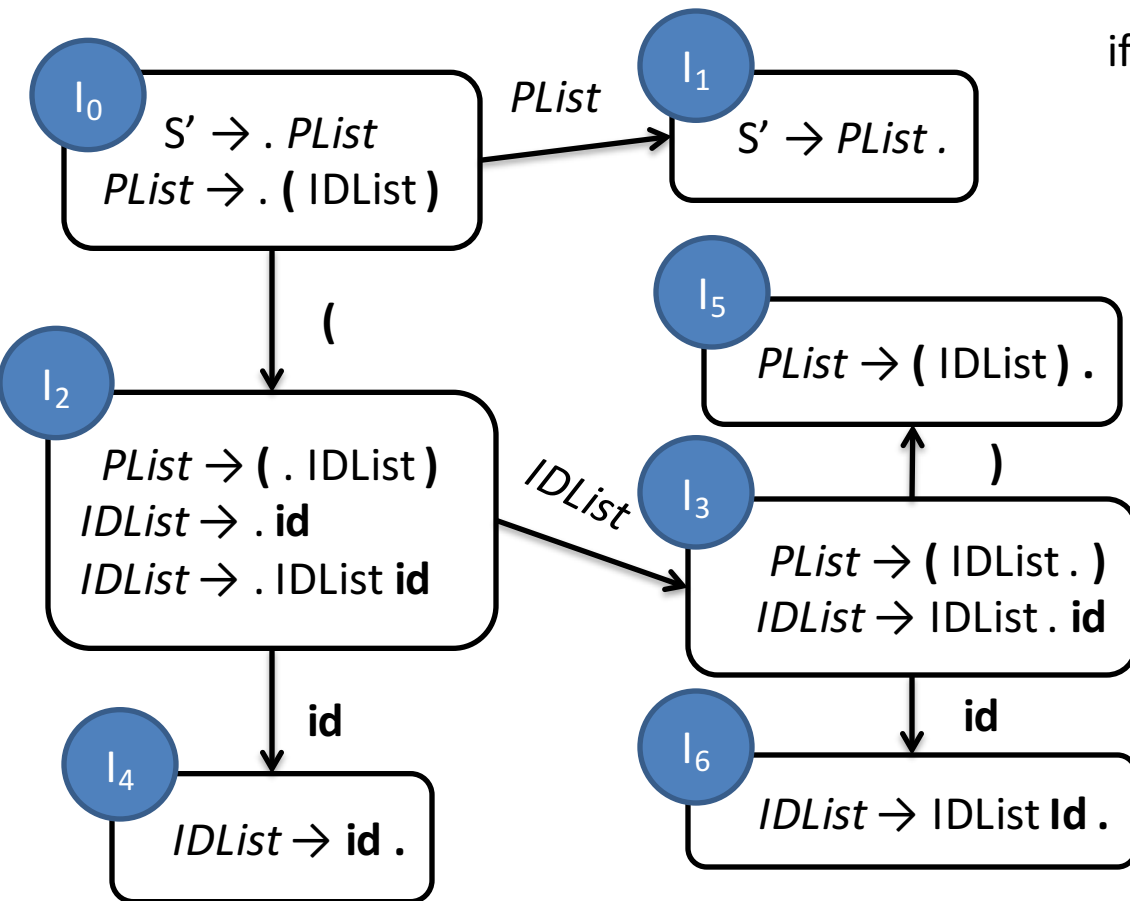
# Action Table: Shift



if state  $i$  includes item  $A \rightarrow \alpha \cdot t \beta$   
 where  $t$  is a terminal  
 and there is an  $(i, j)$  transition on  $t$   
 set  $Action[i, t] = shift\ j$

	(	)	id	eof
0	S 2			
1				
2			S 4	
3		S 5	S 6	
4				
5				
6				

# Action Table: Reduce



if state  $i$  includes item  $A \rightarrow \alpha \cdot$

where  $A$  is not  $S'$

for each  $t$  in  $FOLLOW(A)$ :

set  $Action[i, t] = \text{reduce by } A \rightarrow \alpha$

$FOLLOW(IDList) = \{ ), id \}$

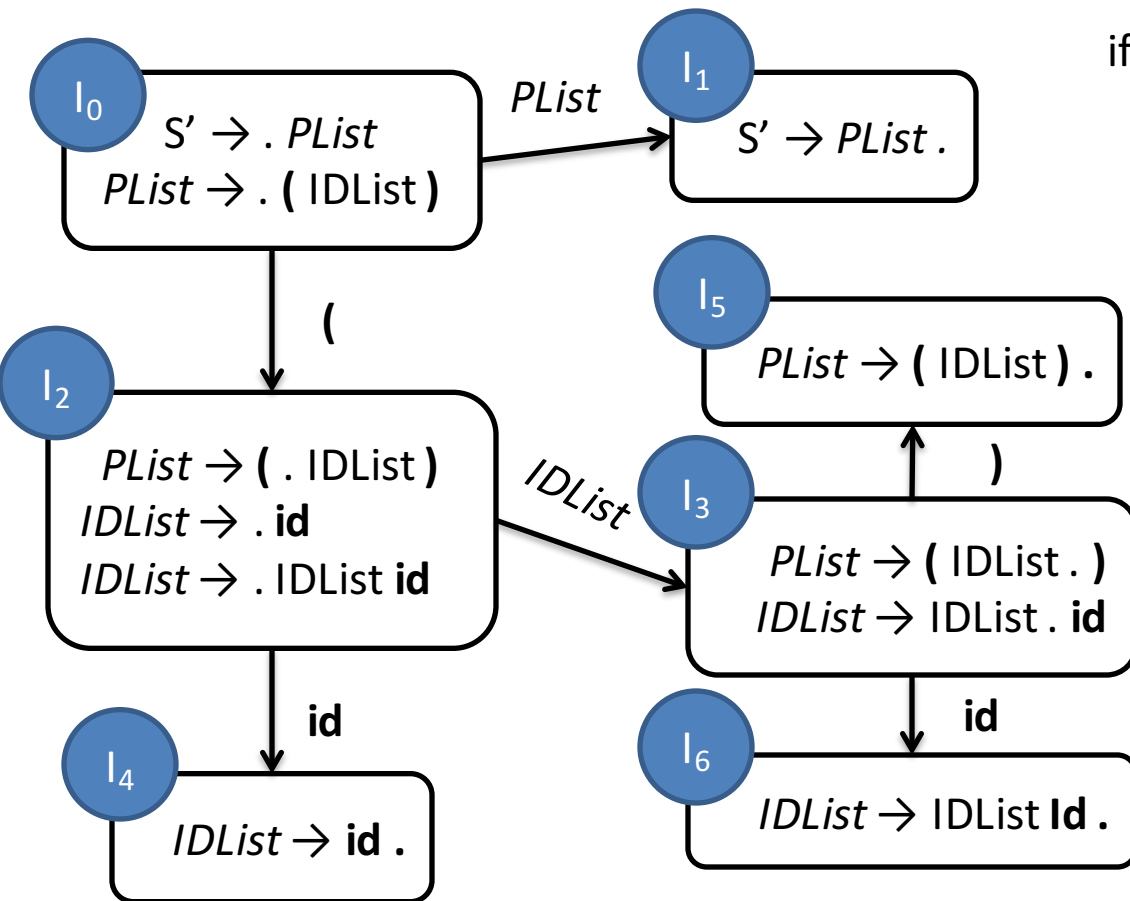
$FOLLOW(PList) = \{ eof \}$

	$($	$)$	$id$	$eof$
0	S 2			
1				
2			S 4	
3		S 5	S 6	
4		R 3	R 3	
5				R 2
6		R 4	R 4	

## Grammar G

- 1  $S' \rightarrow PList$
- 2  $PList \rightarrow ( IDList )$
- 3  $IDList \rightarrow id$
- 4  $IDList \rightarrow IDList id$

# Action Table: Accept



if state  $i$  includes item  $S' \rightarrow S \cdot$   
set  $Action[i, eof] = \text{accept}$

	(	)	id	eof
0	S 2			
1				😊
2			S 4	
3		S 5	S 6	
4		R 3	R 3	
5				R 2
6		R 4	R 4	

## Grammar G

- 1  $S' \rightarrow PList$
- 2  $PList \rightarrow ( IDList )$
- 3  $IDList \rightarrow id$
- 4  $IDList \rightarrow IDList id$

# Some Final Thoughts on LR Parsing

- A bit complicated to build the parse table
  - Fortunately, algorithms exist
- Still not all powerful
  - Shift/reduce: action table cell includes S and R
  - Reduce/reduce: cell include  $> 1$  R rule
- SDT similar to LL(1)
  - Embed SDT action numbers in action table
  - Fire off on reduce rules