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

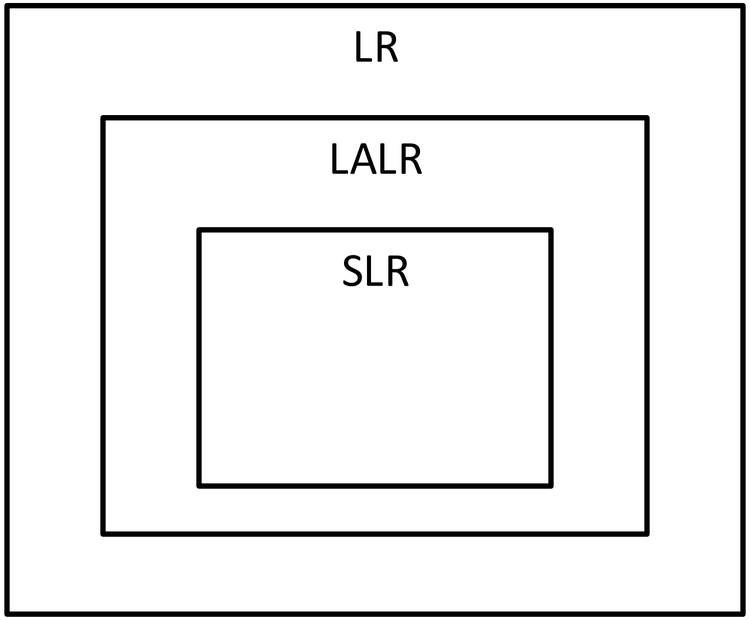
LR Parser Power

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

LR Parser types

- 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

Recognizable by a deterministic PDA



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?

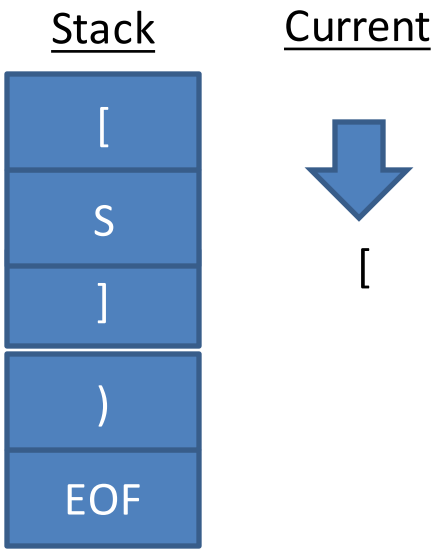
- Already seen 1 such parser: CYK
  - Simultaneously tracked every possible parse tree
    - LR parsers work in a similar same way
- Contrast this to top-down parser
  - We know exactly where we are in the parse
  - Make predictions about what’s next

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
- 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 澄清

<u>Grammar</u>

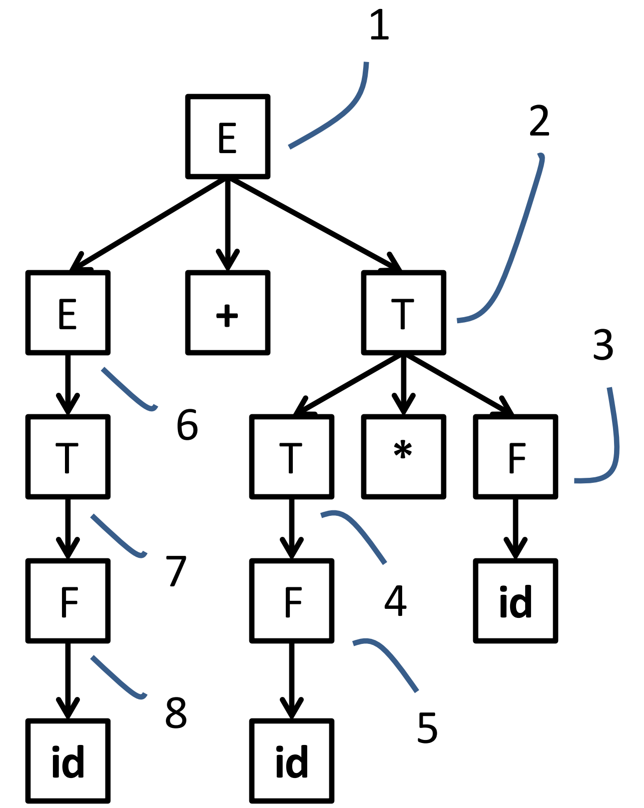
$$
\begin{aligned}
S \; ::= \; & \varepsilon \\
\mid \; & (\; S \;) \\
\mid \; & [\; S \;]
\end{aligned}
$$



LR Derivation Order

- Let's remember derivation orders again

Reverse	Rightmost	derivation
8	1	$E \Longrightarrow E + T$
7	2	$\Longrightarrow E + T * F$
6	3	$\Longrightarrow E + T * id$
5	4	$\Longrightarrow E + F * id$
4	5	$\Longrightarrow E + id * id$
3	6	$\Longrightarrow T + id * id$
2	7	$\Longrightarrow F + id * id$
1	8	$\Longrightarrow id + id * id$

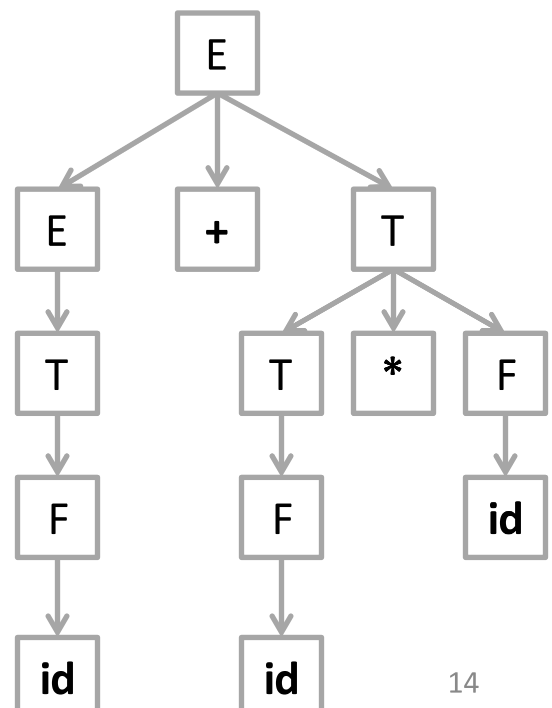


Parser Operations

- 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

Stack	Input	Action
	id + id * id EOF	shift(id)
id	+ id * id EOF	reduce by F ⟶ id
F	+ id * id EOF	reduce by T ⟶ F
T	+ id * id EOF	reduce by E ⟶ T
E	+ id * id EOF	shift +
E +	id * id EOF	shift id
E + id	* id EOF	reduce by F ⟶ id
E + F	* id EOF	reduce by T ⟶ F
E + T	* id EOF	shift *
E + T *	id EOF	shift id
E + T * id	EOF	reduce by F ⟶ id
E + T * F	EOF	reduce by T ⟶ T * F
E + T	EOF	reduce by E ⟶ 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

$$X \longrightarrow \alpha \;.\; B \;\beta$$

    - Means
      - we are in a production of X
      - We believe we've parsed (arbitrary) symbol string α
      - We could handle a production of B
      - After that we'll have β

Stack Item Examples

- Example 1

  PList → **(** . IDList **)**

- Example 2

  PList → **(** IDList . **)**

- Example 3

  PList → **(** IDList **)** **.**

- Example 4

  PList → **.** **(** 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

<u>Grammar snippet</u>

$S \rightarrow A$
$A \rightarrow B$
$\quad | \; C$
$B \rightarrow D\ \textbf{id}$
$C \rightarrow \textbf{id}\ E$
$D \rightarrow \textbf{id}\ E$

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

LR Parser FSM

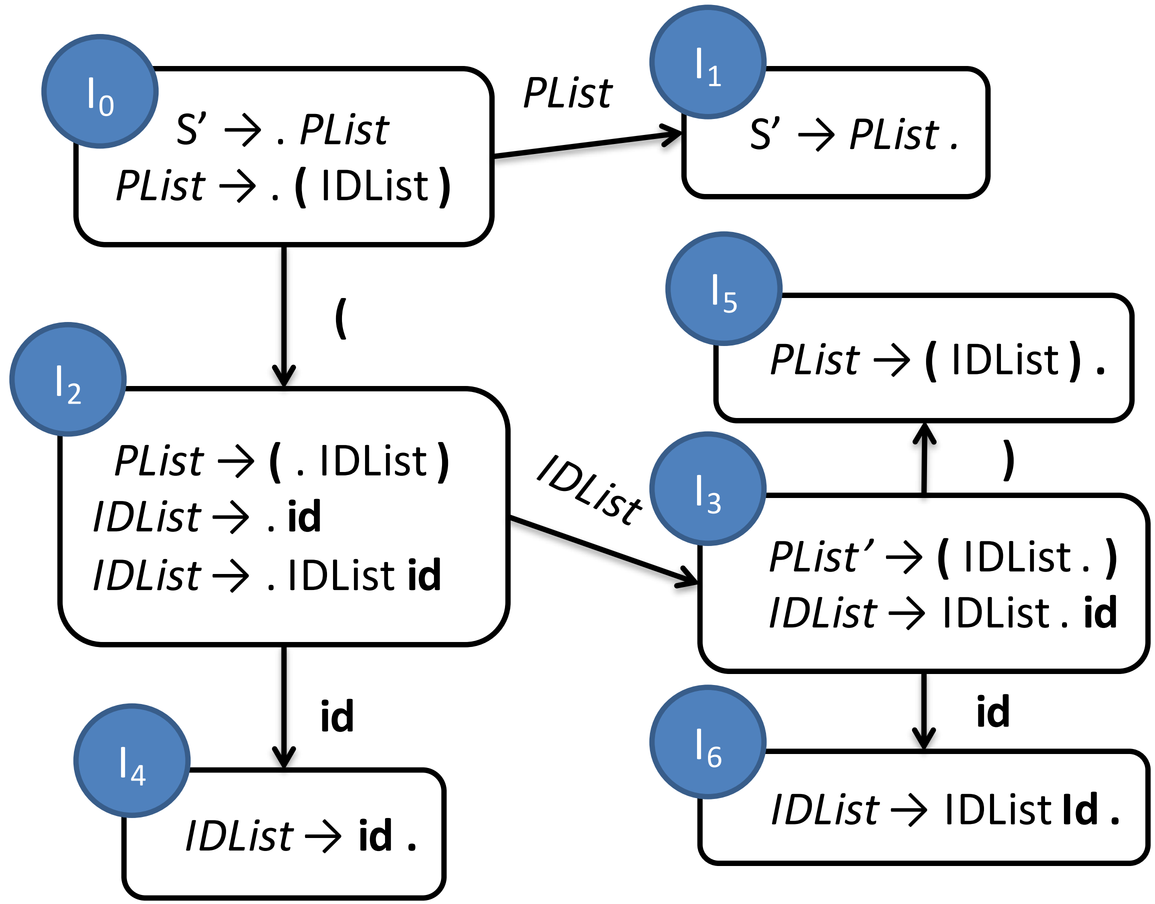
Grammar G

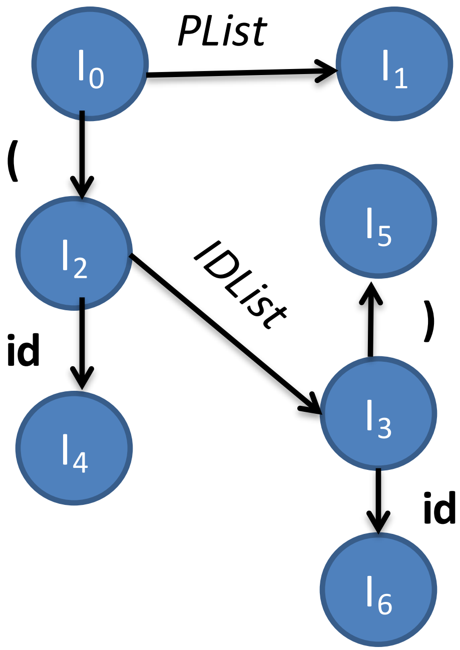
S' → *PList*

PList → (*IDList*)

IDList → **id**

IDList → *IDList* **id**





Automaton as a table

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

Action table / **GoTo table**

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

How do we know to reduce?

Action table

	(	)	id	eof
0	S 2			
1				
2			S 4	
3		S 5	S 6	
4		R ❸	R ❸	
5				R ❷
6		R ❹	R ❹	

GoTo table

	PList	*IDList*
0	1	
1		
2		3
3		
4		
5		
6		

Grammar G

❶ *S'* → *PList*
❷ *PList* → (*IDList*)
❸ *IDList* → **id**
❹ *IDList* → *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

	(	)	id	eof
0	S 2			
1				☺
2			S 4	
3		S 5	S 6	
4		R ❸	R ❸	
5				R ❷
6		R ❹	R ❹	

GoTo table

PList	*IDList*
1	
	3

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

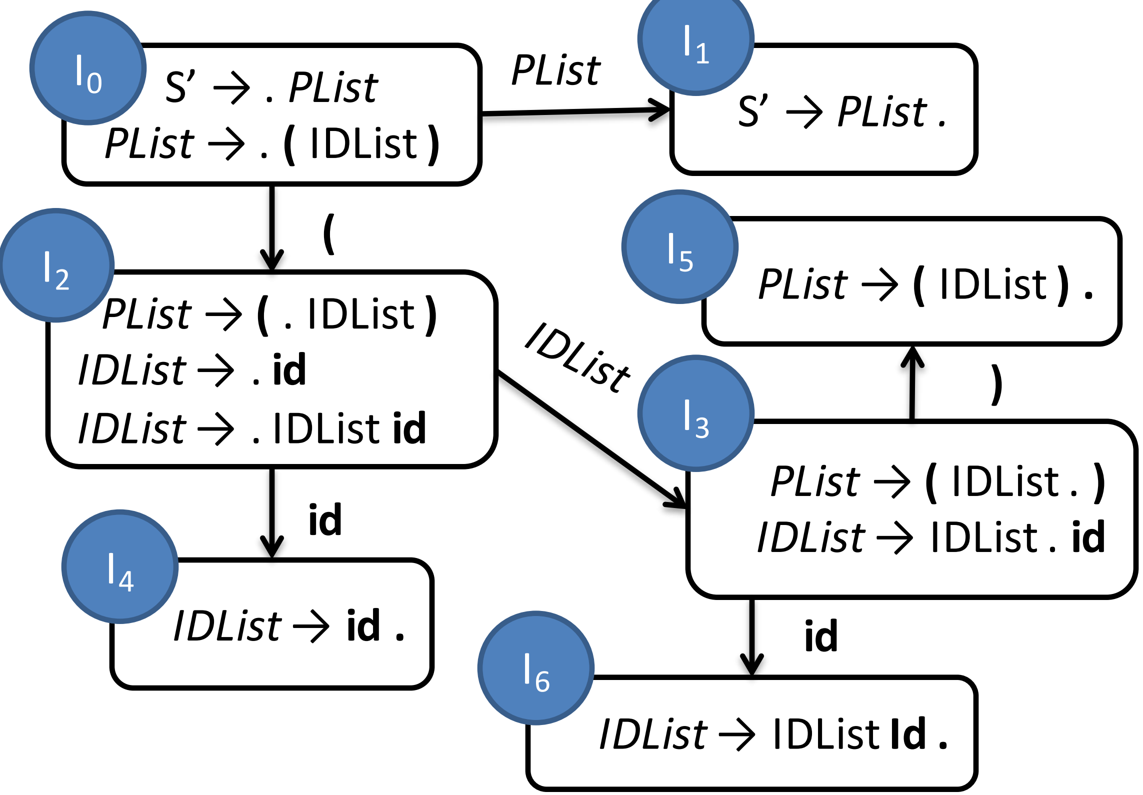
<u>Grammar G</u>

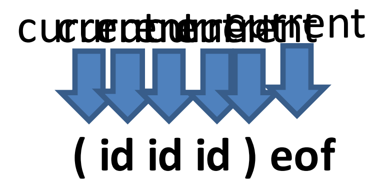
❶ *S'* → *PList*
❷ *PList* → (*IDList*)
❸ *IDList* → **id**
❹ *IDList* → *IDList* **id**

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 → 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
```

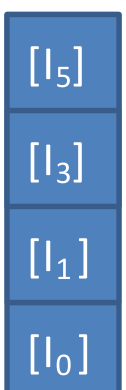
Example Time





Grammar G

1. S' → *PList*
2. *PList* → (*IDList*)
3. *IDList* → **id**
4. *IDList* → *IDList* **id**



	(	)	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 possible 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 Closure(I)

While there exists an item in Closure(I) of the form

$X \longrightarrow \alpha \, . \, B \, \beta$

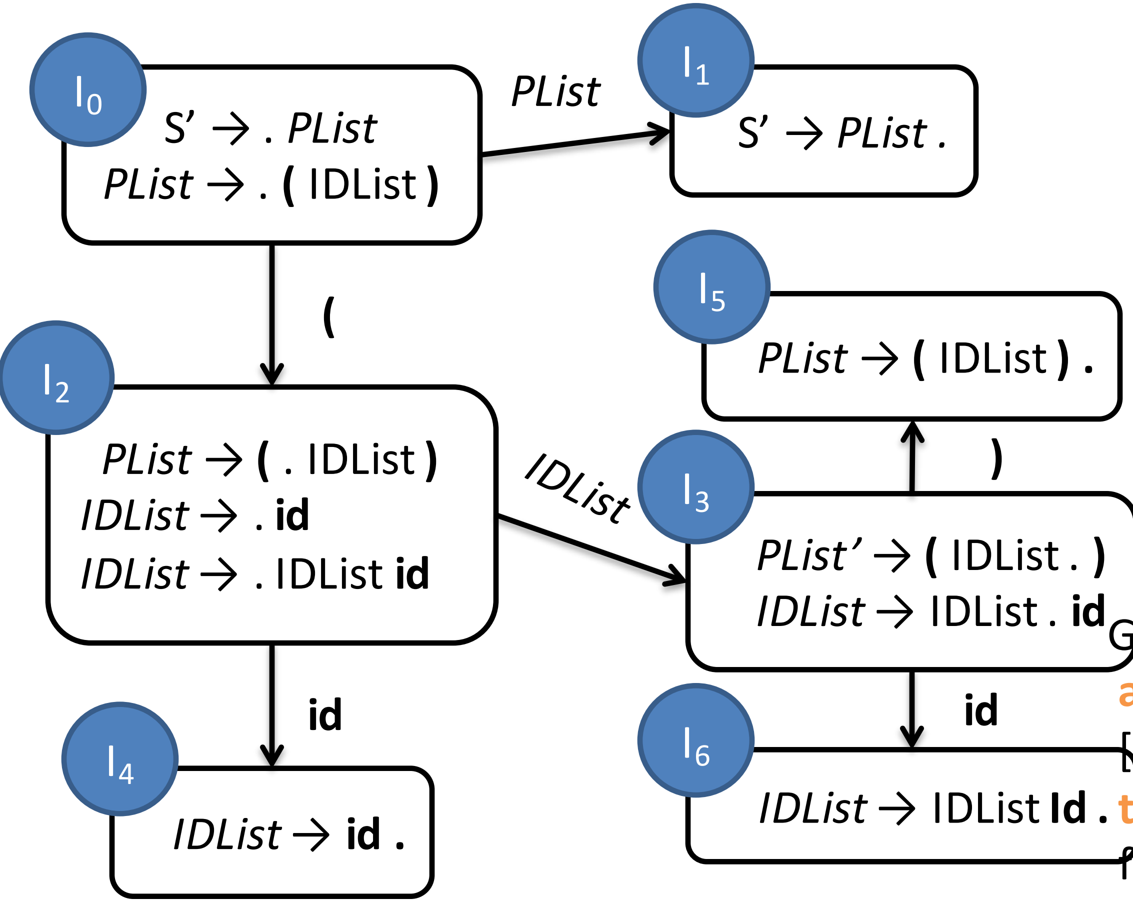
such that there is a production $B \longrightarrow \gamma$,

and $B \longrightarrow . \, \gamma$ is not in Closure(I)

add $B \longrightarrow . \, \gamma$ to 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' → *PList*

PList → (*IDList*)

IDList → **id**

IDList → *IDList* **id**

GoTo(I, X) / **Closure(I)**

Put I in Closure(I) / Closure of all items

Repeat for A → α.X.β s.t. A → α.Xβ ϵ I

X → α.Bβ ϵ Closure(I) s.t.

∃B → γ, add B → .γ to Closure(I)

GoTo(I_0, *PList*)

GoTo(I_2, *IDList*)

Done with closure, and GoTo

Parse Table Construction

1: Add new start S' and S' → S

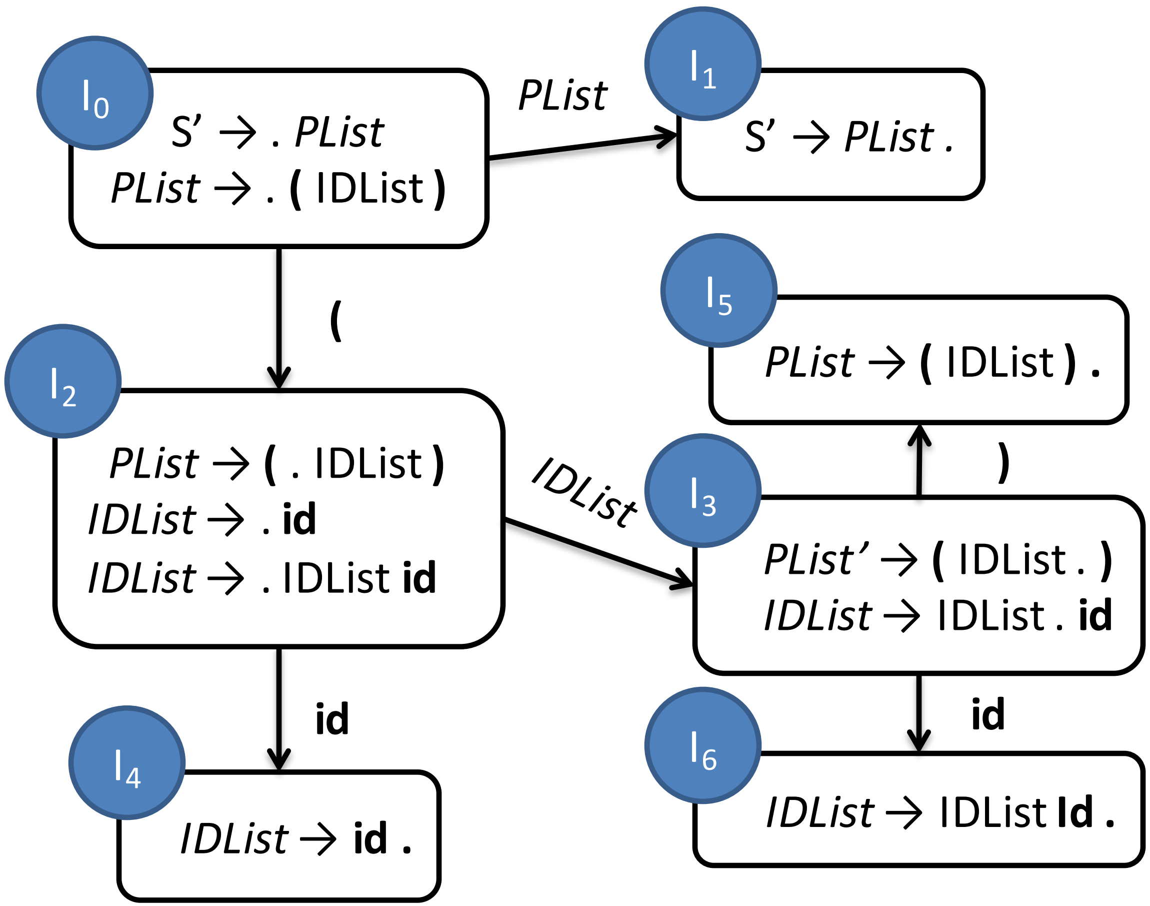
2: Build State I_0 for Closure({S' → . S })

3: Saturate FSM:

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

add transition from state j to state for GoTo(j, X)

From FSM to parse table(s)

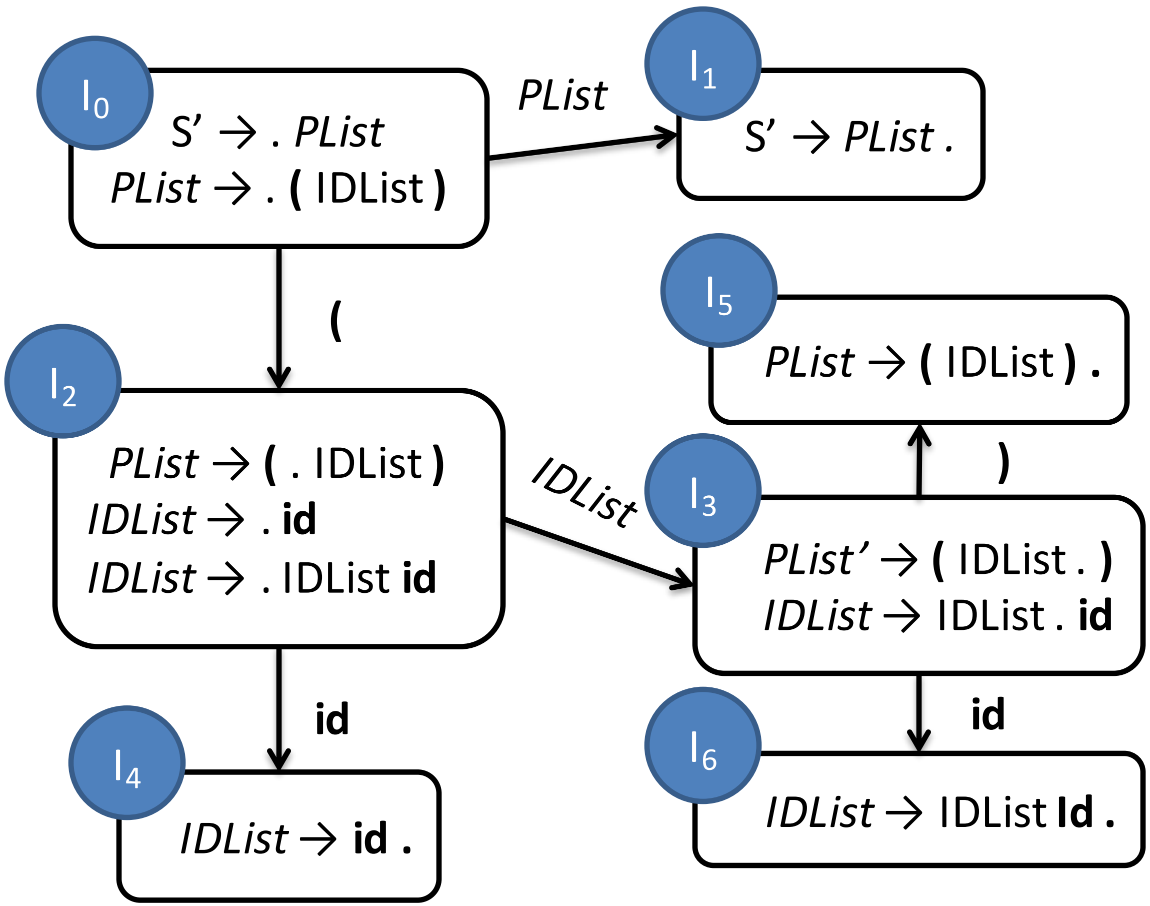


Need to connect the FSM back to the grammar

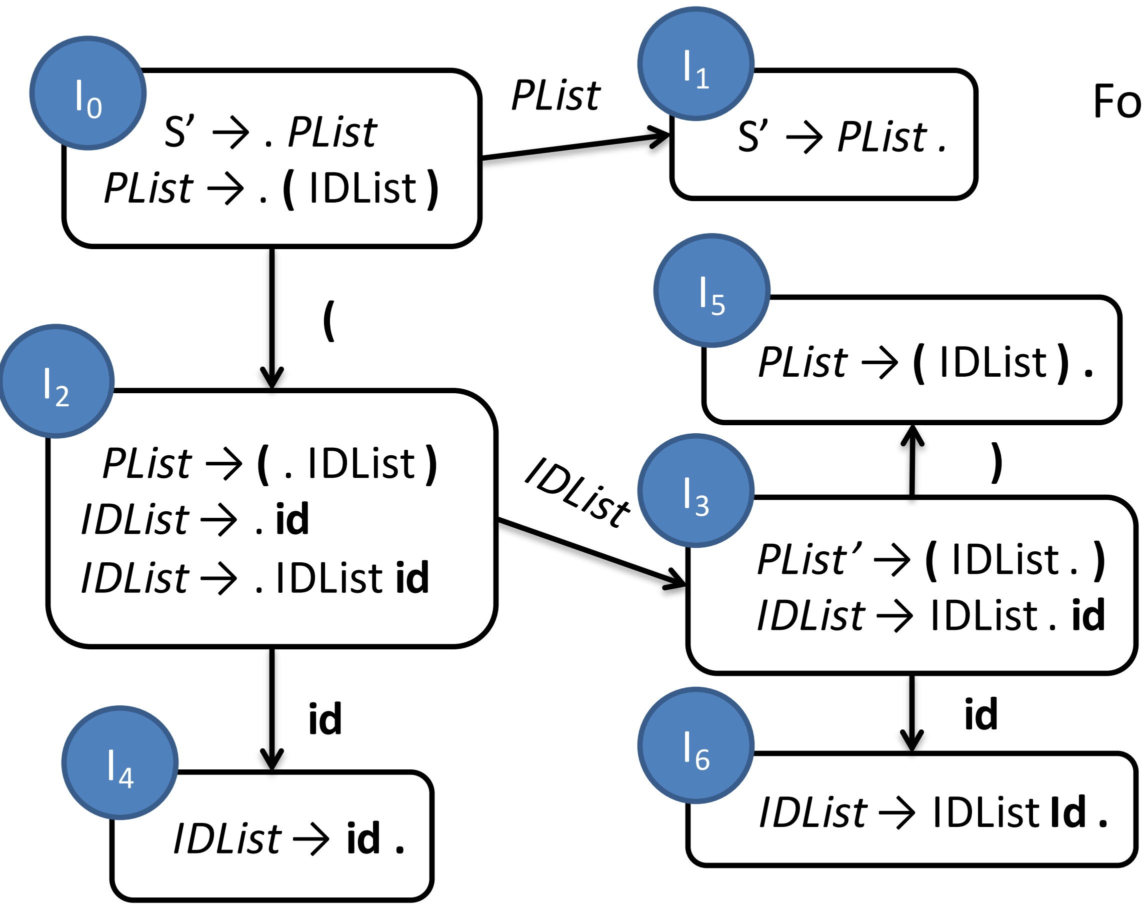
Grammar G

1. S' → *PList*
2. *PList* → (*IDList*)
3. *IDList* → **id**
4. *IDList* → *IDList* **id**

Can Now Build Action and GoTo Tables



Building the GoTo Table



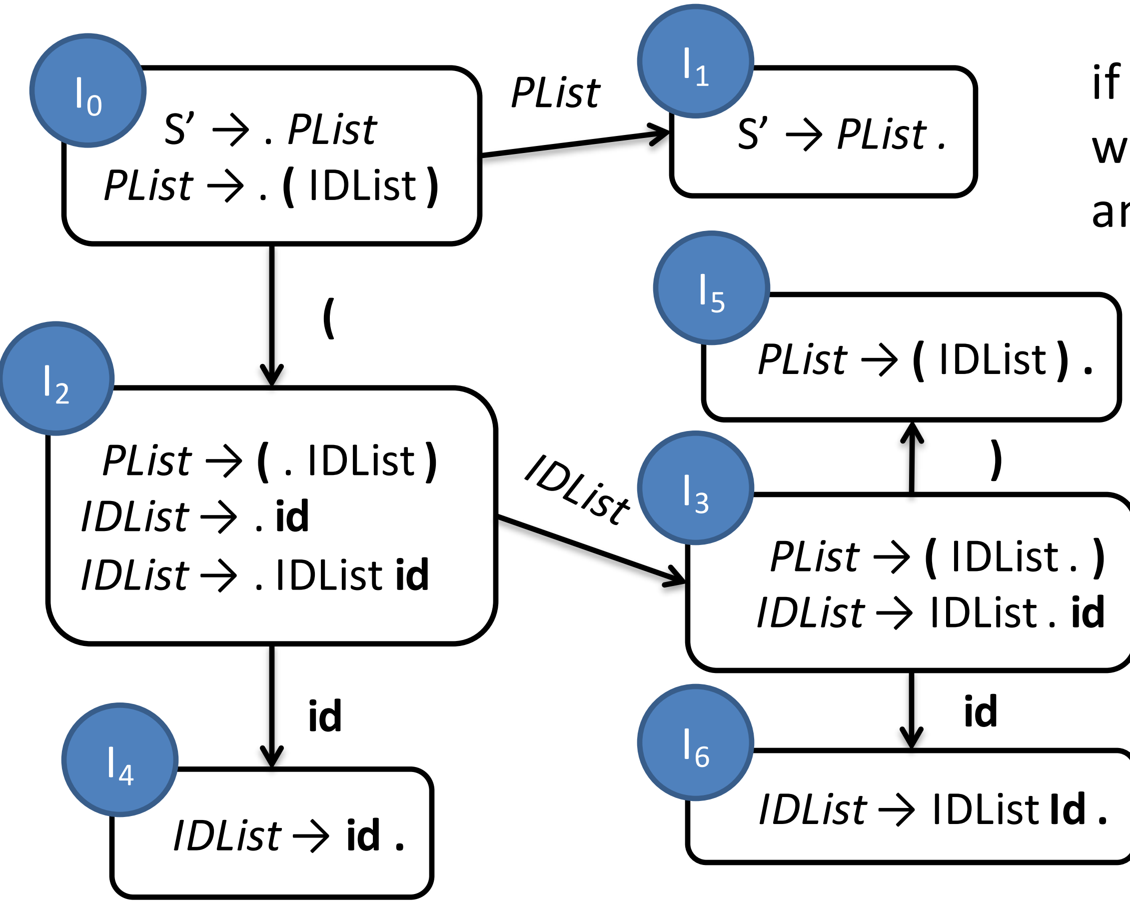
For every nonterminal X
if there is an (i,j) edge on X
set GoTo[i,X] = j

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

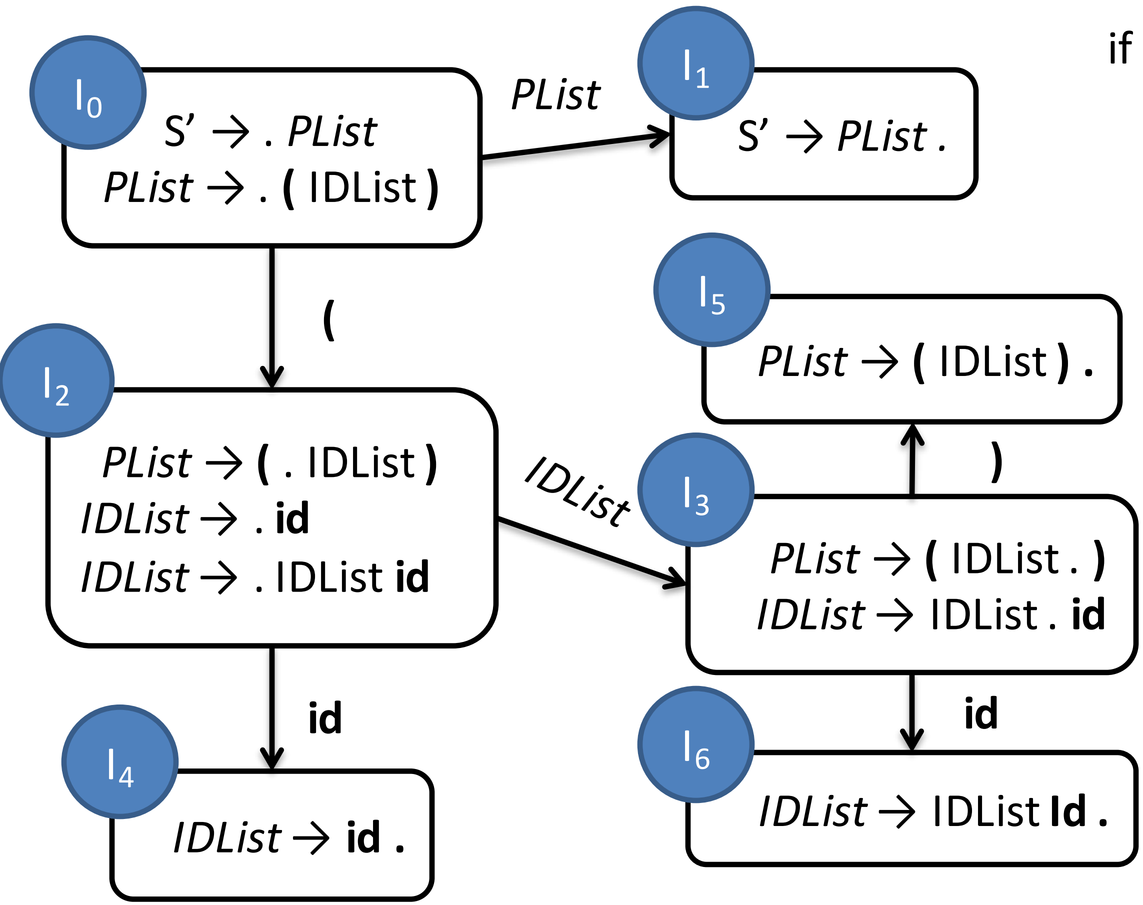
Action Table: Shift



if state i includes item A → α . **t** β
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 → α .
  where A is not *S'*
    for each t in FOLLOW(A):
      set Action[i,**t**] = reduce by A → α

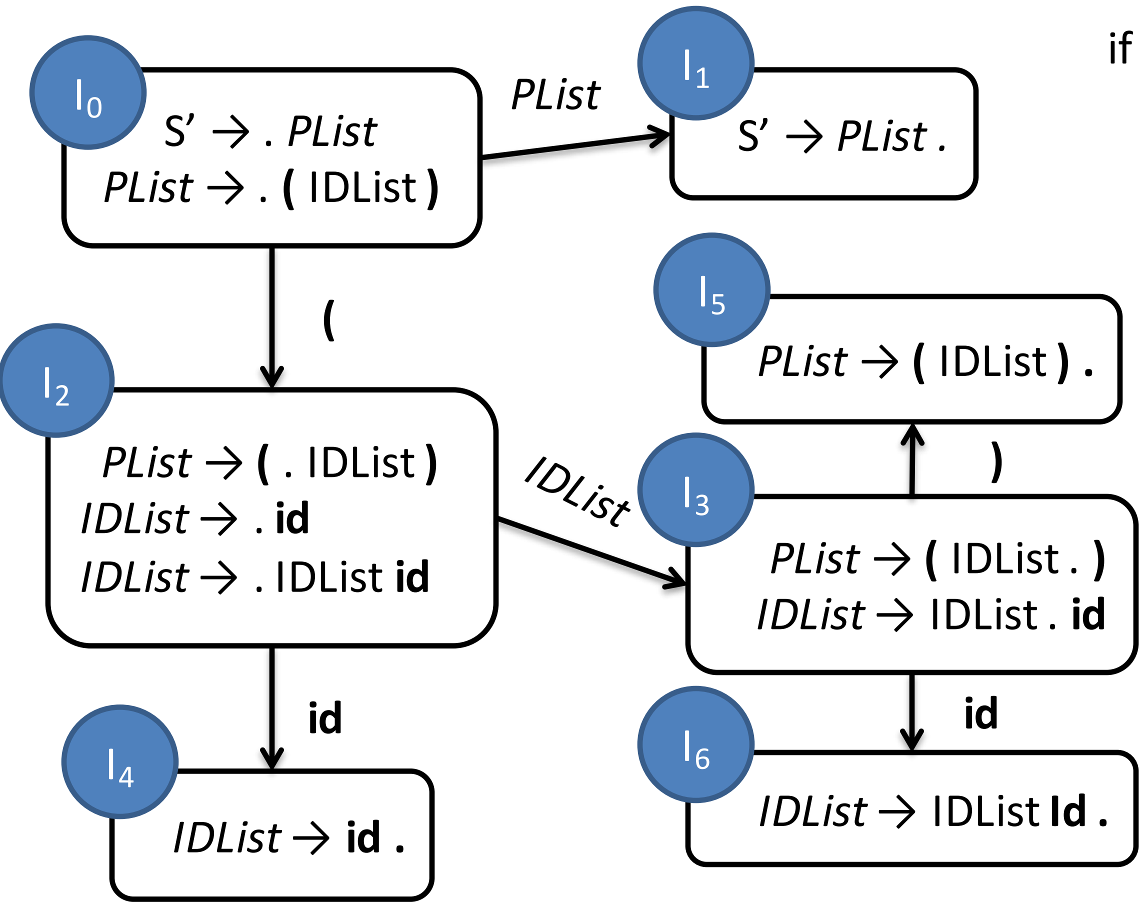
FOLLOW(*IDList*) = { **)**, **id** }
FOLLOW(*PList*) = { **eof** }

	(	**)**	**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'* → *PList*
2. *PList* → (*IDList*)
3. *IDList* → **id**
4. *IDList* → *IDList* **id**

Action Table: Accept



if state i includes item S' → S .
set Action[i,**eof**] = 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' → *PList*
2. *PList* → (*IDList*)
3. *IDList* → **id**
4. *IDList* → *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