Syntax Analysis – Part IV
Bottom-Up Parsing

EECS 483 – Lecture 7
University of Michigan
Wednesday, September 27, 2006

Announcements: Turning in Project 1

- Anonymous ftp to www.eecs.umich.edu
  - login: anonymous
  - pw: your email addr
- cd groups/eecs483
- put uniquename.l
- put uniquename.y
- Note, you won't be able to "get" or "rm" any files in the directory – try if you wish
- If you make a mistake, then put uniquename2.l and send Simon mail (chenxu@umich.edu)
- Grading signup sheet available next wk

Grammars

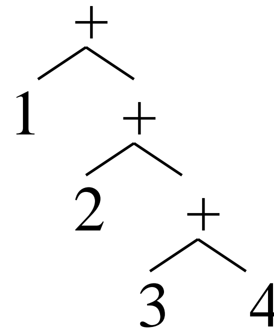
- Have been using grammar for language "sums with parentheses" (1+2+(3+4))+5
- Started with simple, right-associative grammar
  - » S → E + S | E
  - » E → num | (S)
- Transformed it to an LL(1) by left factoring:
  - » S → ES'
  - » S' → ε | +S
  - » E → num (S)
- What if we start with a left-associative grammar?
  - » S → S + E | E
  - » E → num | (S)

Reminder: Left vs Right Associativity

Consider a simpler string on a simpler grammar: "1 + 2 + 3 + 4"

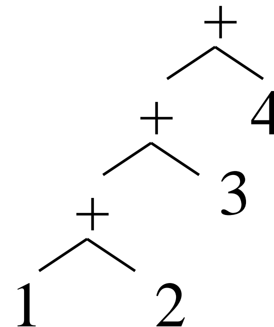
Right recursion : right associative

S → E + S
S → E
E → num



Left recursion : left associative

S → S + E
S → E
E → num



Left Recursion

S → S + E
S → E
E → num

"1 + 2 + 3 + 4"

derived string	lookahead	read/unread
S	1	1+2+3+4
S+E	1	1+2+3+4
S+E+E	1	1+2+3+4
S+E+E+E	1	1+2+3+4
E+E+E+E	1	1+2+3+4
1+E+E+E	2	1+2+3+4
1+2+E+E	3	1+2+3+4
1+2+3+E	4	1+2+3+4
1+2+3+4	$	1+2+3+4

Is this right? If not, what's the problem?

Left-Recursive Grammars

- Left-recursive grammars don't work with top-down parsers: we don't know when to stop the recursion
- Left-recursive grammars are NOT LL(1)!
  - » $S \rightarrow S\alpha$
  - » $S \rightarrow \beta$
- In parse table
  - » Both productions will appear in the predictive table at row S in all the columns corresponding to FIRST(β)

Eliminate Left Recursion

- Replace
  - $X \rightarrow X\alpha 1 \mid \ldots \mid X\alpha m$
  - $X \rightarrow \beta 1 \mid \ldots \mid \beta n$
- With
  - $X \rightarrow \beta 1 X' \mid \ldots \mid \beta n X'$
  - $X' \rightarrow \alpha 1 X' \mid \ldots \mid \alpha m X' \mid \varepsilon$
- See complete algorithm in Dragon book

Class Problem

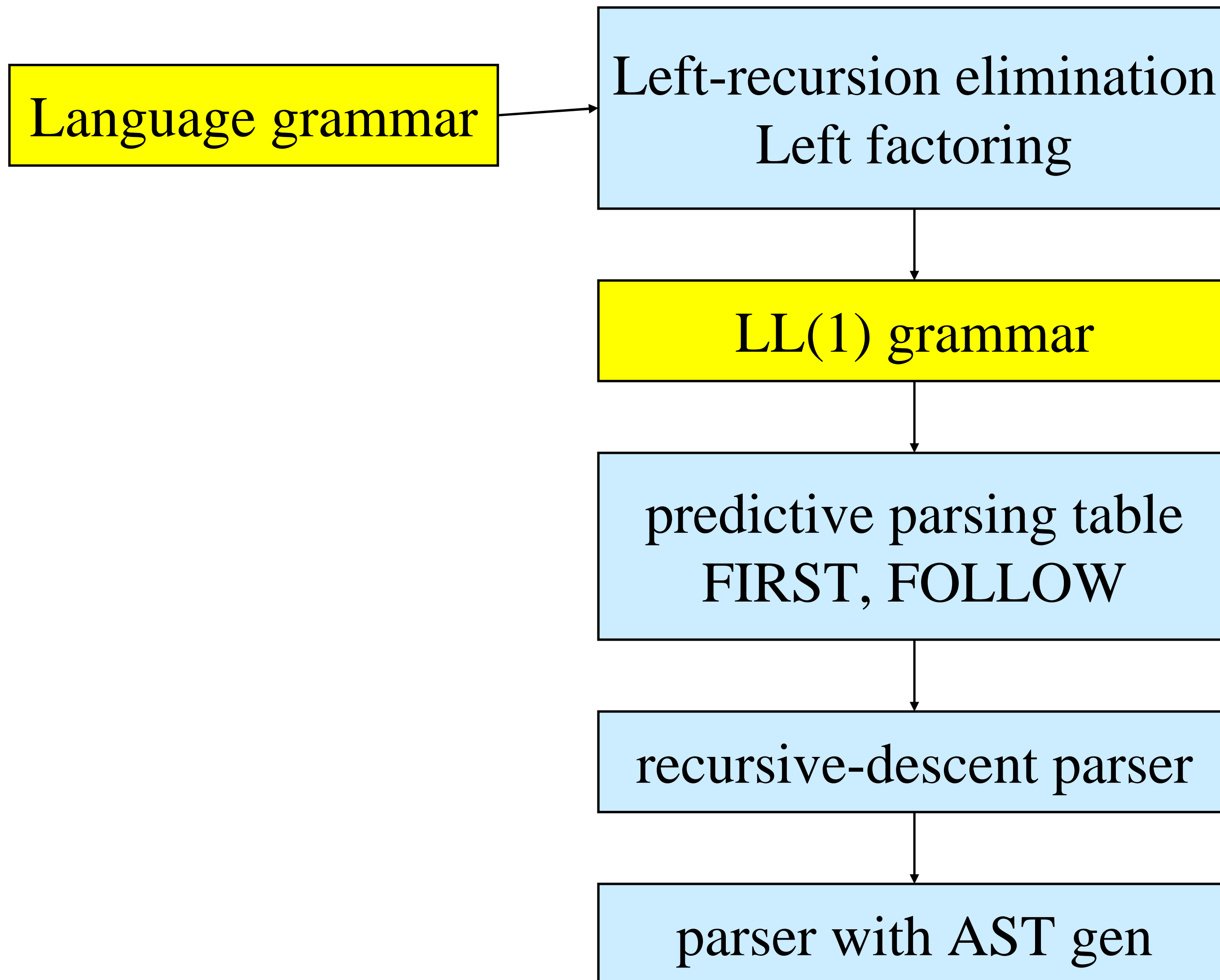
Transform the following grammar to eliminate left recursion:

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid num$

Creating an LL(1) Grammar

- Start with a left-recursive grammar
  - S → S + E
  - S → E
  - and apply left-recursion elimination algorithm
    - S → ES'
    - S' → +ES' | ε
- Start with a right-recursive grammar
  - S → E + S
  - S → E
  - and apply left-factoring to eliminate common prefixes
    - S → ES'
    - S' → +S | ε

Top-Down Parsing Summary



New Topic: Bottom-Up Parsing

- A more power parsing technology
- LR grammars – more expressive than LL
  - » Construct right-most derivation of program
  - » Left-recursive grammars, virtually all programming languages are left-recursive
  - » Easier to express syntax
- Shift-reduce parsers
  - » Parsers for LR grammars
  - » Automatic parser generators (yacc, bison)

Bottom-Up Parsing (2)

- Right-most derivation – Backward
  - Start with the tokens
  - End with the start symbol
  - Match substring on RHS of production, replace by LHS

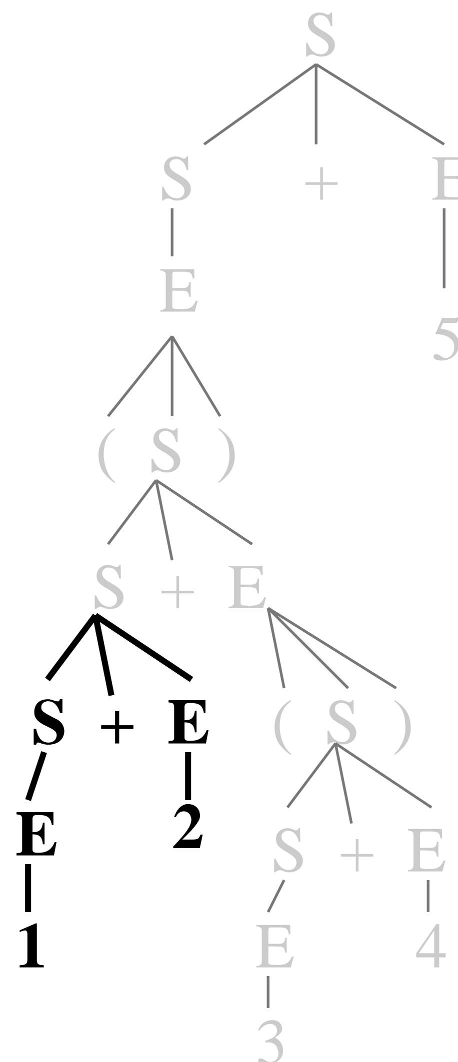
S → S + E | E
E → num | (S)

(1+2+(3+4))+5 ← (E+2+(3+4))+5
← (S+2+(3+4))+5 ← (S+E+(3+4))+5
← (S+(3+4))+5 ← (S+(E+4))+5 ← (S+(S+4))+5
← (S+(S+E))+5 ← (S+(S))+5 ← (S+E)+5 ← (S)+5
← E+5 ← S+E ← S

Bottom-Up Parsing (3)

S → S + E | E
E → num | (S)

(1+2+(3+4))+5
← (E+2+(3+4))+5
← (S+2+(3+4))+5
← (S+E+(3+4))+5

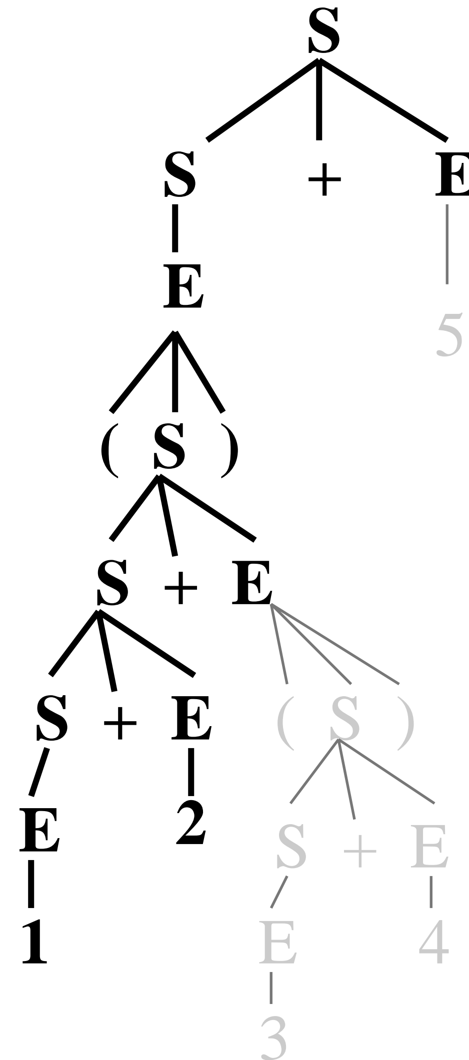


Advantage of bottom-up parsing: can postpone the selection of productions until more of the input is scanned

Top-Down Parsing

S → S + E | E
E → num | (S)

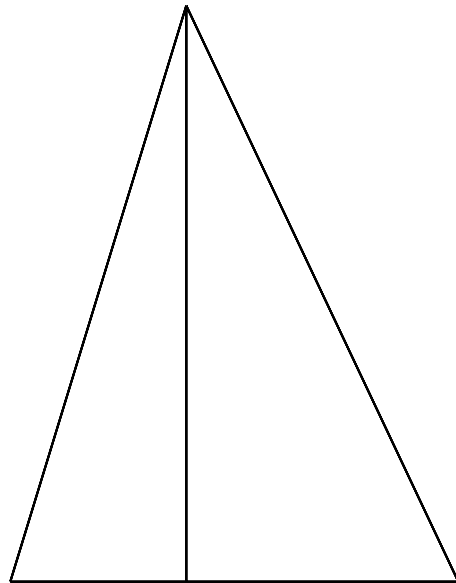
S → S+E → E+E → (S)+E → (S+E)+E
→ (S+E+E)+E → (E+E+E)+E
→ (1+E+E)+E → (1+2+E)+E ...



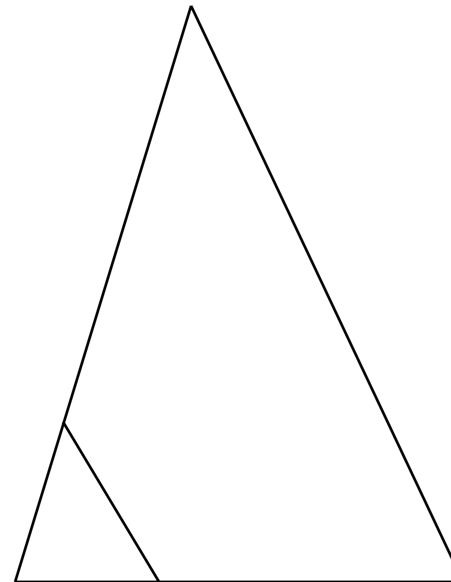
In left-most derivation, entire tree above token (2) has been expanded when encountered

Top-Down vs Bottom-Up

Bottom-up: Don't need to figure out as much of he parse tree for a given amount of input → More time to decide what rules to apply



Top-down



Bottom-up

Terminology: LL vs LR

- LL(k)
  - Left-to-right scan of input
  - Left-most derivation
  - k symbol lookahead
  - [Top-down or predictive] parsing or LL parser
  - Performs pre-order traversal of parse tree
- LR(k)
  - Left-to-right scan of input
  - Right-most derivation
  - k symbol lookahead
  - [Bottom-up or shift-reduce] parsing or LR parser
  - Performs post-order traversal of parse tree

Shift-Reduce Parsing

- Parsing actions: A sequence of shift and reduce operations
- Parser state: A stack of terminals and non-terminals (grows to the right)
- Current derivation step = stack + input

Derivation step	stack	Unconsumed input
(1+2+(3+4))+5 ←		(1+2+(3+4))+5
(E+2+(3+4))+5 ←	(E	+2+(3+4))+5
(S+2+(3+4))+5 ←	(S	+2+(3+4))+5
(S+E+(3+4))+5 ←	(S+E	+(3+4))+5

...

Shift-Reduce Actions

- Parsing is a sequence of shifts and reduces
- Shift: move look-ahead token to stack

stack	input	action
(	1+2+(3+4))+5	shift 1
(1	+2+(3+4))+5	

- Reduce: Replace symbols β from top of stack with non-terminal symbol X corresponding to the production: X→ β (e.g., pop β, push X)

stack	input	action
(S+E	+(3+4))+5	reduce S → S+ E
(S	+(3+4))+5	

Shift-Reduce Parsing

S → S + E | E
E → num | (S)

derivation	stack	input stream	action
(1+2+(3+4))+5		(1+2+(3+4))+5	shift
(1+2+(3+4))+5	(	1+2+(3+4))+5	shift
(1+2+(3+4))+5	(1	+2+(3+4))+5	reduce E→ num
(E+2+(3+4))+5	(E	+2+(3+4))+5	reduce S→ E
(S+2+(3+4))+5	(S	+2+(3+4))+5	shift
(S+2+(3+4))+5	(S+	2+(3+4))+5	shift
(S+2+(3+4))+5	(S+2	+(3+4))+5	reduce E→ num
(S+E+(3+4))+5	(S+E	+(3+4))+5	reduce S → S+E
(S+(3+4))+5	(S	+(3+4))+5	shift
(S+(3+4))+5	(S+	(3+4))+5	shift
(S+(3+4))+5	(S+(	3+4))+5	shift
(S+(3+4))+5	(S+(3	+4))+5	reduce E→ num

…

Potential Problems

- How do we know which action to take: whether to shift or reduce, and which production to apply
- Issues
  - Sometimes can reduce but should not
  - Sometimes can reduce in different ways

Action Selection Problem

- Given stack β and look-ahead symbol b, should parser:
  - » Shift b onto the stack making it βb ?
  - » Reduce $X \rightarrow \gamma$ assuming that the stack has the form $\beta = \alpha\gamma$ making it αX ?
- If stack has the form $\alpha\gamma$, should apply reduction $X \rightarrow \gamma$ (or shift) depending on stack prefix α ?
  - » α is different for different possible reductions since γ's have different lengths

LR Parsing Engine

- Basic mechanism
  - Use a set of parser states
  - Use stack with alternating symbols and states
    - E.g., 1 (6 S 10 + 5 (blue = state numbers)
  - Use parsing table to:
    - Determine what action to apply (shift/reduce)
    - Determine next state
- The parser actions can be precisely determined from the table

LR Parsing Table

	Terminals	Non-terminals
State	Next action and next state	Next state
	Action table	Goto table

- Algorithm: look at entry for current state S and input terminal C
  - » If Table[S,C] = s(S') then shift:
    - push(C), push(S')
  - » If Table[S,C] = $X \rightarrow \alpha$ then reduce:
    - pop(2*|α|), S'= top(), push(X), push(Table[S',X])

LR Parsing Table Example

We want to derive this in an algorithmic fashion

Input terminal / Non-terminals

State	(	)	id	,	$	S	L
1	s3		s2			g4	
2	S→id	S→id	S→id	S→id	S→id		
3	s3		s2			g7	g5
4					accept		
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(k) Grammars

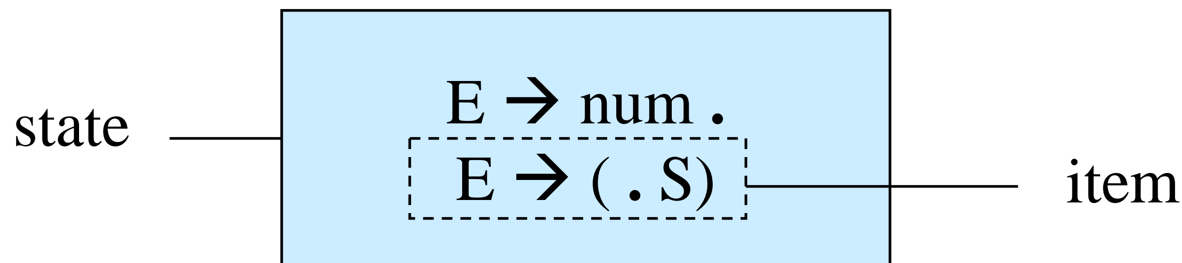
- LR(k) = Left-to-right scanning, right-most derivation, k lookahead chars
- Main cases
  - LR(0), LR(1)
  - Some variations SLR and LALR(1)
- Parsers for LR(0) Grammars:
  - Determine the actions <u>without any lookahead</u>
  - Will help us understand shift-reduce parsing

Building LR(0) Parsing Tables

- To build the parsing table:
  - Define states of the parser
  - Build a DFA to describe transitions between states
  - Use the DFA to build the parsing table
- Each LR(0) state is a set of LR(0) items
  - An LR(0) item: $X \rightarrow \alpha \,.\, \beta$ where $X \rightarrow \alpha\beta$ is a production in the grammar
  - The LR(0) items keep track of the progress on all of the possible upcoming productions
  - The item $X \rightarrow \alpha \,.\, \beta$ abstracts the fact that the parser already matched the string α at the top of the stack

Example LR(0) State

- An LR(0) item is a production from the language with a separator "." somewhere in the RHS of the production



- Sub-string before "." is already on the stack (beginnings of possible γ's to be reduced)
- Sub-string after ".": what we might see next

Class Problem

For the production,

$E \rightarrow num \mid (S)$

Two items are:

$E \rightarrow num\ .$

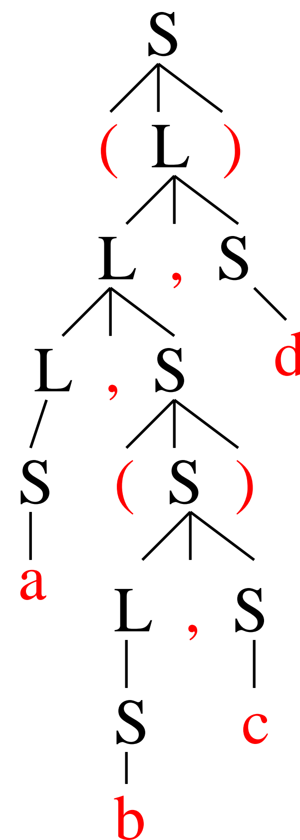
$E \rightarrow (\ .\ S\ )$

Are there any others? If so, what are they? If not, why?

LR(0) Grammar

- Nested lists
  - » S → (L) | id
  - » L → S | L,S
- Examples
  - » (a,b,c)
  - » ((a,b), (c,d), (e,f))
  - » (a, (b,c,d), ((f,g)))

Parse tree for
(a, (b,c), d)



Start State and Closure

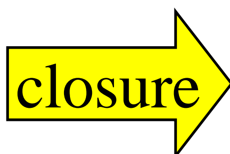
- Start state
  - » Augment grammar with production: S' $\rightarrow$ S \$
  - » Start state of DFA has empty stack: S' $\rightarrow$. S \$
- Closure of a parser state:
  - » Start with Closure(S) = S
  - » Then for each item in S:
    - $X \rightarrow \alpha \,.\, Y \beta$
    - Add items for all the productions $Y \rightarrow \gamma$ to the closure of S: $Y \rightarrow .\, \gamma$

Closure Example

S → (L) | id
L → S | L,S

DFA start state

S' → . S $

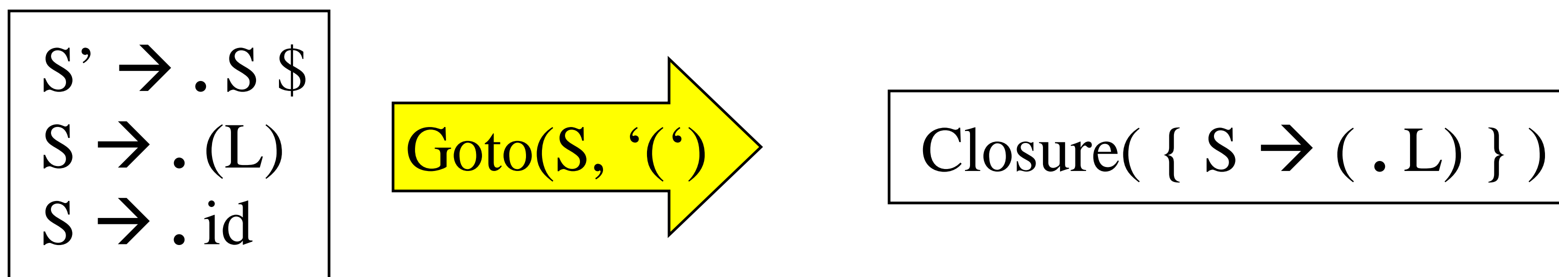


S' → . S $
S → . (L)
S → . id

- Set of possible productions to be reduced next
- Added items have the "." located at the beginning: no symbols for these items on the stack yet

The Goto Operation

- Goto operation = describes transitions between parser states, which are sets of items
- Algorithm: for state S and a symbol Y
  - If the item [X → α . Y β] is in I, then
  - Goto(I, Y) = Closure([X → α Y . β])



Class Problem

E' → E
E → E + T | T
T → T * F | F
F → (E) | id

1. If I = { [E' → . E]}, then Closure(I) = ??

2. If I = { [E' → E .], [E → E . + T] }, then Goto(I,+) = ??