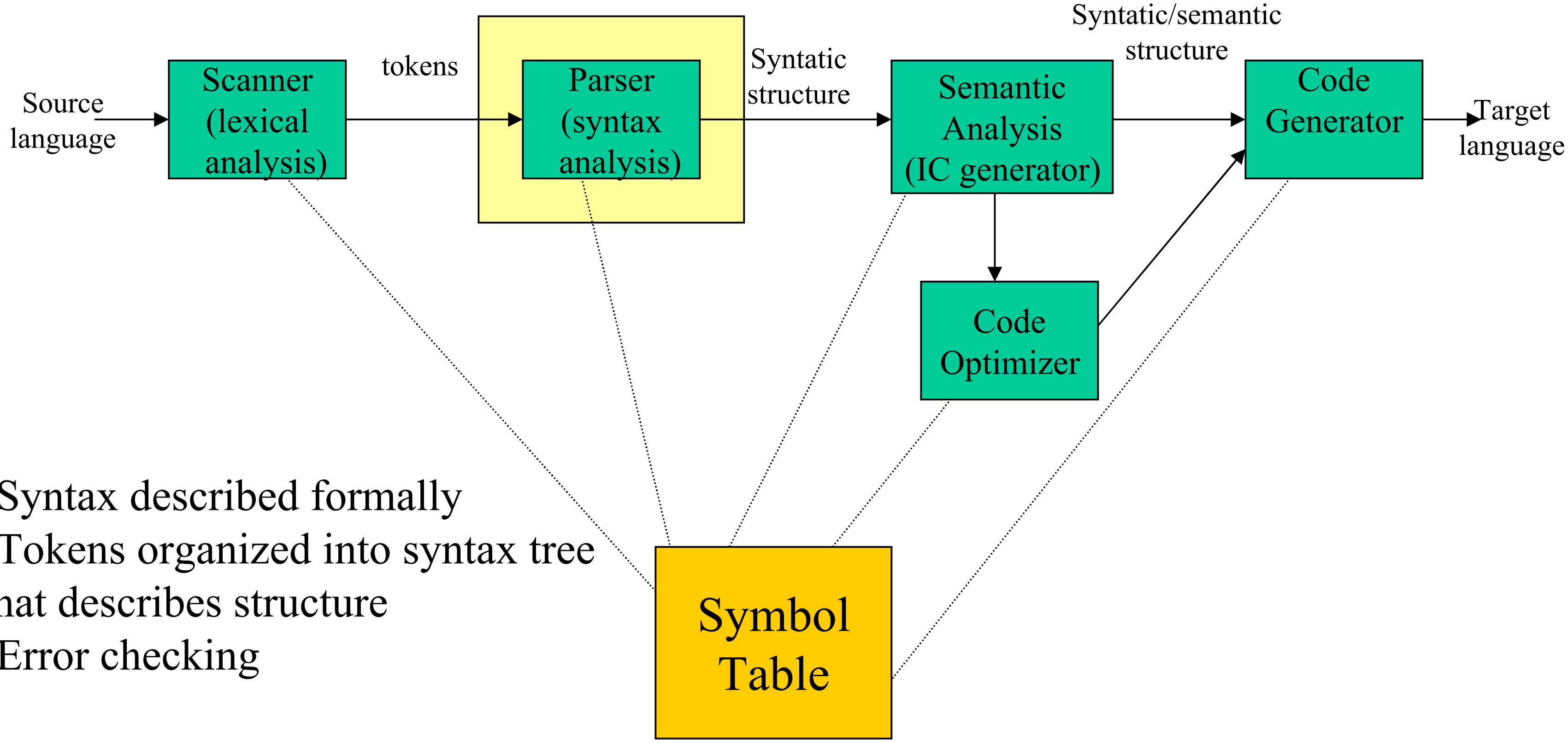
Lecture 5: LR Parsing

CS 540

George Mason University

Static Analysis - Parsing



- Syntax described formally
- Tokens organized into syntax tree that describes structure
- Error checking

LL vs. LR

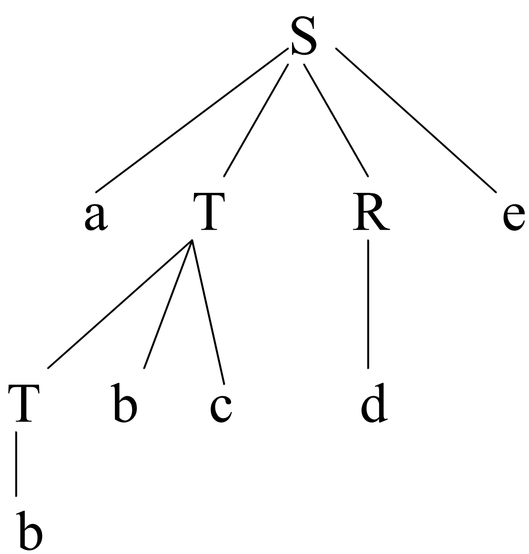
- LR (shift reduce) is more powerful than LL (predictive parsing)
- Can detect a syntactic error as soon as possible.
- LR is difficult to do by hand (unlike LL)

LR(*k*) Parsing – Bottom Up

- Construct parse tree from leaves, ‘reducing’ the string to the start symbol (and a single tree)
- During parse, we have a ‘forest’ of trees
- Shift-reduce parsing
  - **‘Shift’ a new input symbol**
  - **‘Reduce’ a group of symbols to a single non-terminal**
  - Choice is made using the *k* lookaheads
- LR(1)

Example

- S → a T R e
  T → T b c | b
  R → d



- Rightmost derivation:

  S → a T R e
  → a T **d** e
  → a T **b** **c** d e
  → a **b** b c d e

LR parsing corresponds to the rightmost derivation in reverse.

Shift Reduce Parsing

S → a T R e

T → T b c | b

R → d

Remaining input: abbcde

Rightmost derivation:

S → a T R e

→ a T **d** e

→ a **T b c** d e

→ a b b c d e

Shift Reduce Parsing

S → a T R e

T → T b c | b

R → d

Remaining input: bcde

→ Shift a, Shift b

a b

Rightmost derivation:

S → a T R e

→ a T **d** e

→ a **T b c** d e

→ a b b c d e

Shift Reduce Parsing

$S \rightarrow a\ T\ R\ e$

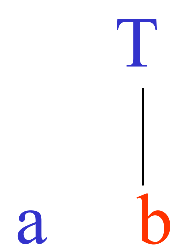
$T \rightarrow T\ b\ c \mid b$

$R \rightarrow d$

Remaining input: bcde

➔ Shift a, Shift b

➔ Reduce $T \rightarrow b$



Rightmost derivation:

$S \rightarrow a\ T\ R\ e$

$\rightarrow a\ T\ \mathbf{d}\ e$

$\rightarrow \underline{a\ T}\ \mathbf{b}\ \mathbf{c}\ d\ e$

$\rightarrow \mathbf{a}\ \mathbf{b}\ b\ c\ d\ e$

Shift Reduce Parsing

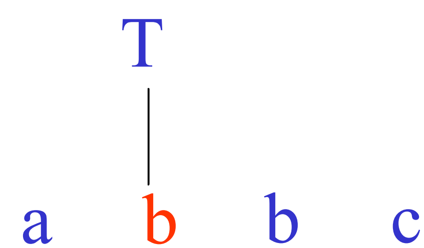
S → a T R e

T → T b c | b

R → d

Remaining input: de

- → Shift a, Shift b
- → Reduce T → b
- → Shift b, Shift c



Rightmost derivation:

S → a T R e

→ a T **d** e

→ **a T b c** d e

→ **a b** b c d e

Shift Reduce Parsing

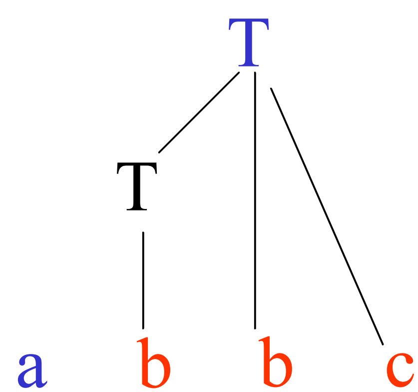
S → a T R e

T → T b c | b

R → d

Remaining input: de

- → Shift a, Shift b
- → Reduce T → b
- → Shift b, Shift c
- → Reduce T → T b c



Rightmost derivation:

S → a T R e

→ a T **d** e

→ a **T b c** d e

→ **a b** b c d e

Shift Reduce Parsing

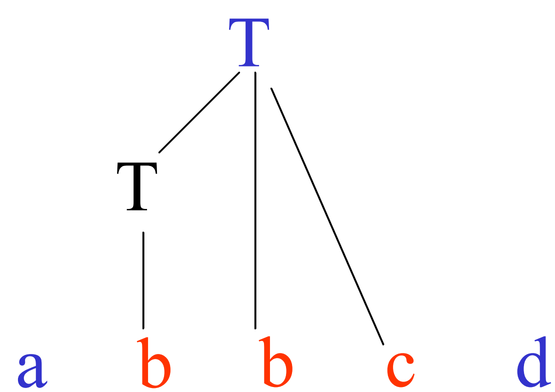
S → a T R e

T → T b c | b

R → d

Remaining input: e

- ➔ Shift a, Shift b
- ➔ Reduce T → b
- ➔ Shift b, Shift c
- ➔ Reduce T → T b c
- ➔ Shift d



Rightmost derivation:

S → a T R e

→ a T **d** e

→ a **T b c** d e

→ **a b** b c d e

Shift Reduce Parsing

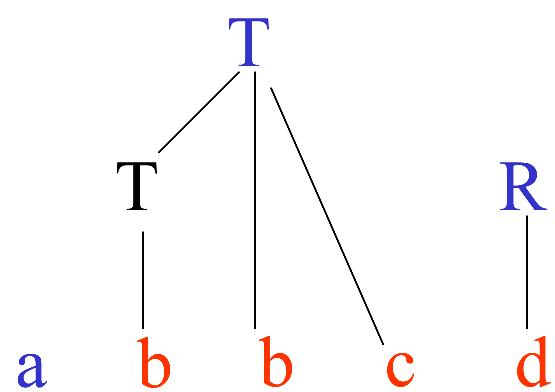
S → a T R e

T → T b c | b

R → d

Remaining input: e

- ➔ Shift a, Shift b
- ➔ Reduce T → b
- ➔ Shift b, Shift c
- ➔ Reduce T → T b c
- ➔ Shift d
- ➔ Reduce R → d



Rightmost derivation:

S → a T R e

→ a T **d** e

→ a **T b c** d e

→ **a b** b c d e

Shift Reduce Parsing

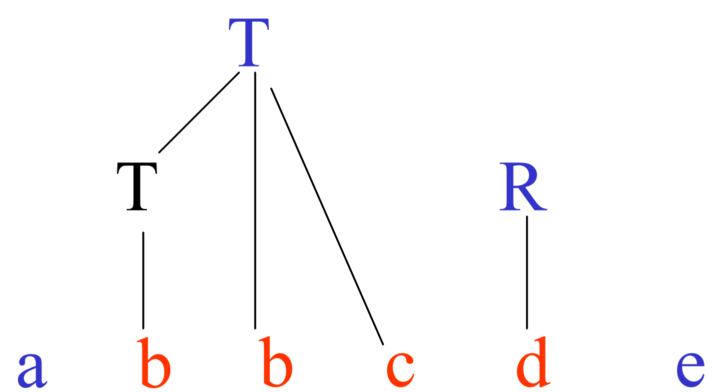
S → a T R e

T → T b c | b

R → d

Remaining input:

➔ Shift a, Shift b
➔ Reduce T → b
➔ Shift b, Shift c
➔ Reduce T → T b c
➔ Shift d
➔ Reduce R → d
➔ Shift e



Rightmost derivation:

S → a T R e

→ a T **d** e

→ a **T b c** d e

→ **a b** b c d e

Shift Reduce Parsing

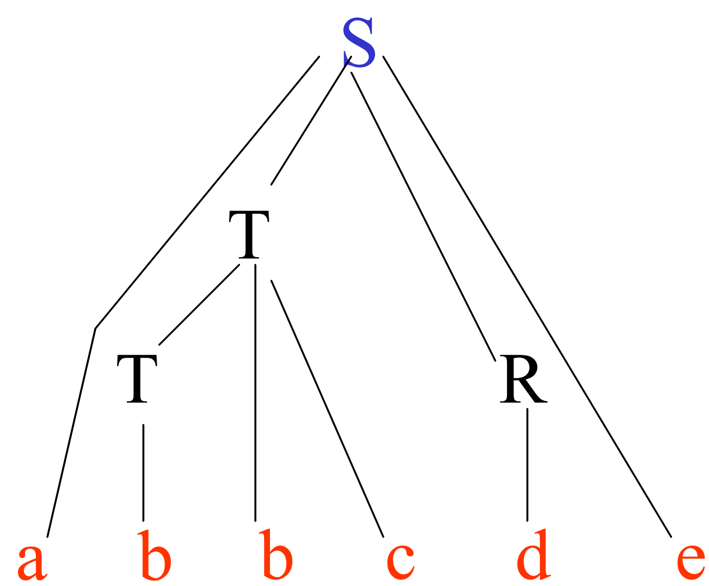
S → a T R e

T → T b c | b

R → d

- ➔ Shift a, Shift b
- ➔ Reduce T → b
- ➔ Shift b, Shift c
- ➔ Reduce T → T b c
- ➔ Shift d
- ➔ Reduce R → d
- ➔ Shift e
- ➔ Reduce S → a T R e

Remaining input:



Rightmost derivation:

S ➔ a T R e

➔ a T **d** e

➔ a **T b c** d e

➔ **a b** b c d e

LR Parsing

- Data Structures:
  - Stack – contains symbol/state pairs. The state on top of stack summarizes the information below.
  - Tables:
    - Action: state x $\Sigma \rightarrow$ reduce/shift/accept/error
    - Goto: state x $V_n \rightarrow$ state

Example LR Table

State	a	b	c	d	e	$	S	T	R
0	s1								
1		s3						2	
2		s5		s6					4
3		r3		r3					
4					s7				
5			s8						
6					r4				
7						acc			
8		r2		r2					

1: S → a T R e
2: T → T b c
3: T → b
4: R → d

Action table
Goto table

s means shift to to some state

r means reduce by some production

Algorithm: LR(1)

```
push($,0);   /* always pushing a symbol/state pair */
lookahead = yylex();
loop
  s = top(); /*always a state */
  if action[s,lookahead] = shift s'
    push(lookahead,s');  lookahead = yylex();
  else if action[s,lookahead] = reduce A → β
    pop size of β pairs
    s' = state on top of stack
    push(A,goto[s',A]);
  else if action[s,lookahead] = accept then return
  else error();
end loop;
```

LR Parsing Example 1

Stack	**Input**	**Action**
$0	a b b c d e $	s1

LR Parsing Example 1

Stack	Input	Action
\$0	a b b c d e \$	s1
\$0,a1	b b c d e \$	s3

LR Parsing Example 1

Stack	Input	Action
$0	a b b c d e $	s1
$0,a1	b b c d e $	s3
$0,a1,b3	b c d e $	r3 (T → b)

LR Parsing Example 1

Stack	Input	Action
$0	a b b c d e $	s1
$0,a1	b b c d e $	s3
$0,a1,b3	b c d e $	r3 (T → b)
$0,a1,T2	b c d e $	s5
$0,a1,T2,b5	c d e $	s8
$0,a1,T2,b5,c8	d e $	r2 (T → T b c)

goto(T,1)=2

LR Parsing Example 1

Stack	Input	Action
$0	a b b c d e $	s1
$0,a1	b b c d e $	s3
$0,a1,b3	b c d e $	r3 (T → b)
$0,a1,T2	b c d e $	s5
$0,a1,T2,b5	c d e $	s8
$0,a1,T2,b5,c8	d e $	r2 (T → T b c)
$0,a1,T2	d e $	s6
$0,a1,T2,d6	e $	r4 (R → d)

goto(T,1)=2

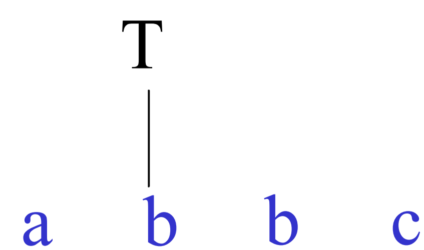
LR Parsing Example 1

Stack	Input	Action
$0	a b b c d e $	s1
$0,a1	b b c d e $	s3
$0,a1,b3	b c d e $	r3 (T → b)
$0,a1,T2	b c d e $	s5
$0,a1,T2,b5	c d e $	s8
$0,a1,T2,b5,c8	d e $	r2 (T → T b c)
$0,a1,T2	d e $	s6
$0,a1,T2,d6	e $	r4 (R → d)
$0,a1,T2,R4	e $	s7
$0,a1,T2,R4,e7	$	accept!

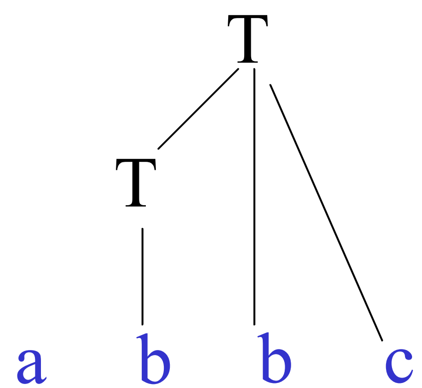
goto(R,2)=4

LR Parse Stack

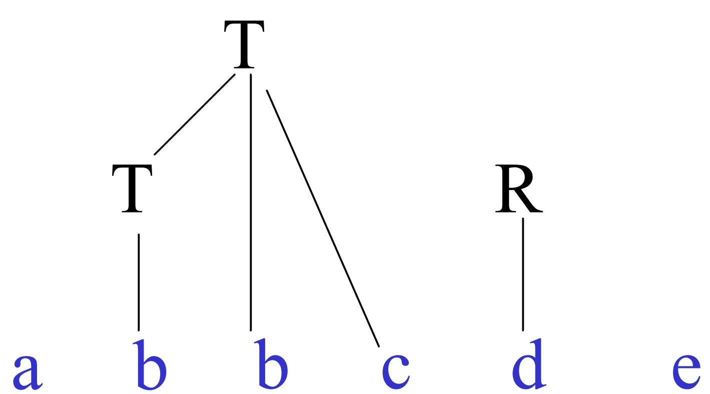
- During LR parsing, there is always a ‘forest’ of trees.
- Parse stack holds root of each of these trees:
  – For example, that stack $0,a1,T2,b5,c8
    represents the corresponding forest



The next stack: $0,a1,T2



Later, we have $0,a1,T2,R6,e7



Where does the table come from?

Handle – "a substring that matches the right side of a production and whose reduction to the non-terminal represents one step along the reverse of a rightmost derivation"

Using the grammar, want to create a DFA to find handles.

SLR parsing

- Simplest LR algorithm
- Provide an understanding of
  - the basic mechanics of shift/reduce parsing
  - source of shift/reduce and reduce/reduce conflicts
- There are better (more powerful) algorithms (LALR, LR) but we won't study them here.

Generating SLR parse tables

- **Augmented grammar: grammar with new start symbol and production S’ → S where S is old start symbol.**
  - Augmentation only required if there is no single production to signal the end.
- **Construct C = {…} the LR(0) items**
- Construct Action table for state *i* of parser:
- All undefined entries are *error*

LR(0) items

- Canonical LR(0) collections are the basis for constructing SLR (simple LR) parsers
- **Defn**: LR(0) item of a grammar G is a production of G with a dot at some point on the right side.
- A → X Y Z   yields *four different* LR(0) items:
  - A → . X Y Z
  - A → X . Y Z
  - A → X Y . Z
  - A → X Y Z .
- A → ε   yields one item
  - A → .

Closure(I) function

Closure(I) where I is a set of LR(0) items =

- Every item in I (*kernel*) and
- If $A \rightarrow \alpha \,.\, B\, \beta$ in closure(I) and $B \rightarrow \gamma$ is a production, add $B \rightarrow .\, \gamma$ to closure(I) (if not already there).
- Keep applying this rule until no more items can be added.

Closure Example

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Closure($\{T \rightarrow T * . F\}$) = $\{T \rightarrow T * . F, F \rightarrow . (E), F \rightarrow . id\}$

Closure($\{E \rightarrow E . + T, F \rightarrow . id\}$) = $\{E \rightarrow E . + T, F \rightarrow . id\}$

Closure Example

E' → E

E → E + T | T

T → T * F | F

F → (E) | id

Closure({F → (. E)}

= {F → (. E), E → . E + T, E → . T}

= {F → (. E), E → . E + T, E → . T, T → . T * F, T → . F}

= {F → (. E), E → . E + T, E → . T, T → . T * F, T → . F,
F → . Id, F → . (E) }

Goto function

Goto(I,X), where I is a set of items and X is a grammar symbol, is the closure($A \rightarrow \alpha\ X\ .\ \beta$) where $A \rightarrow \alpha\ .\ X\ \beta$ is in I.

Ex: Goto($\{E' \rightarrow E\ .,\ E \rightarrow E\ .\ +\ T\}$,+)

$= $ closure($\{E \rightarrow E + .\ T\}$)

$= \{E \rightarrow E + .\ T,\ T \rightarrow .\ T * F,\ T \rightarrow .\ F,\ F \rightarrow .\ id,$
$F \rightarrow .\ (\ E\ )\}$

Goto function

- Goto({T → T * . F, T → . F},F)

  = closure({T → T * F ., T → F . })

  = {T → T * F ., T → F .}

- Goto({E'→E ., E → E + . T},+)

  = closure(∅) = ∅

  since + does not occur before the . symbol

Algorithm: Finding canonical collection $C = \{I_0, I_1, \ldots, I_n\}$ for grammar G

- $C = \{closure(\{S' \rightarrow . S\})\}$ for start symbol S'

  I_0
- Repeat
  - For each I_k in C and grammar symbol X such that $Goto(I_k, X)$ is not empty and not in C
    - Add $Goto(I_k, X)$ to C

Example 1

Grammar: $S \rightarrow a\ T\ R\ e,\ T \rightarrow T\ b\ c \mid b,\ R \rightarrow d$

I_0: $S \rightarrow .\ a\ T\ R\ e$ Goto({$S \rightarrow .\ a\ T\ R\ e$ },a) = I_1

I_1: $S \rightarrow a\ .\ T\ R\ e$ Goto({$S \rightarrow a\ .\ T\ R\ e\ ,\ T \rightarrow .\ T\ b\ c$},T)

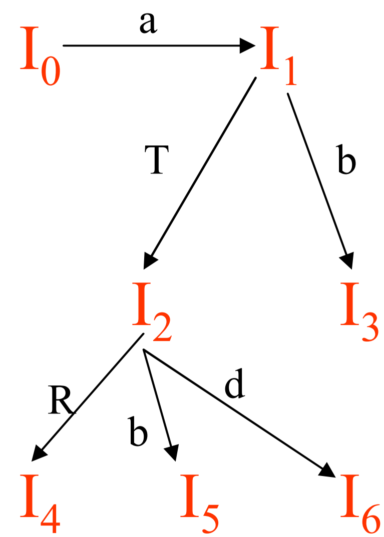
$T \rightarrow .\ T\ b\ c$ = I_2

$T \rightarrow .\ b$ Goto({$T \rightarrow .\ b$ },b) = I_3

I_2: $S \rightarrow a\ T\ .\ R\ e$ goto 4

$T \rightarrow T\ .\ b\ c$ goto 5

$R \rightarrow .\ d$ goto 6



kernel of each item set is in blue

Example 1

Grammar: $S \rightarrow a\ T\ R\ e$, $T \rightarrow T\ b\ c \mid b$, $R \rightarrow d$

I_3: $T \rightarrow b\ .$ reduce

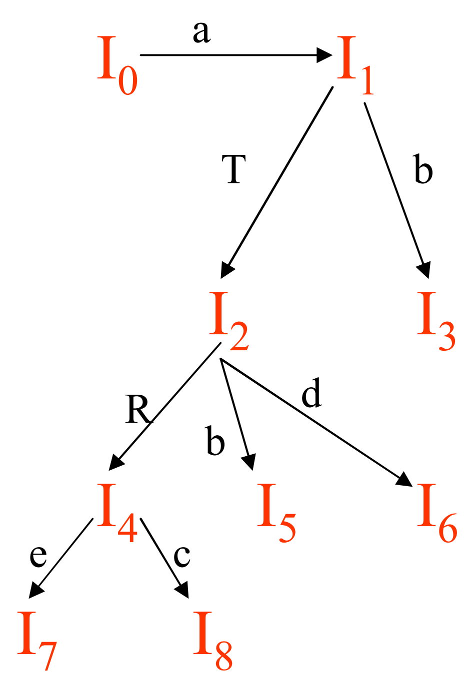
I_4: $S \rightarrow a\ T\ R\ .\ e$ goto state 7

I_5: $T \rightarrow T\ b\ .\ c$ goto state 8

I_6: $R \rightarrow d\ .$ reduce

I_7: $S \rightarrow a\ T\ R\ e\ .$ reduce

I_8: $T \rightarrow T\ b\ c\ .$ reduce



Algorithm: Canonical sets

```
state = 0; max_state = 1;
kernel[0] = [S' → . S]
loop
      c = closure(kernel[state]);
      for t in c, where all productions are form A → α . B β
            if exists k <= state where t = kernel[k] then  goto(state,B) = k;
            else
              kernel[max_state] = goto(state,B) = t;
              max_state++;
      state++;
until state+1 = max_state;
```

Example 2

Grammar: S' $\rightarrow$ S, S $\rightarrow$ A S | b, A $\rightarrow$ S A | c

I_0: S' $\rightarrow$. S

S $\rightarrow$. A S

S $\rightarrow$. b

A $\rightarrow$. S A

A $\rightarrow$. c

I_1: S' $\rightarrow$ S .

A $\rightarrow$ S . A

A $\rightarrow$. S A

A $\rightarrow$. c

S $\rightarrow$. A S

S $\rightarrow$. b

Example 2

Grammar: $S' \rightarrow S$, $S \rightarrow A\ S \mid b$, $A \rightarrow S\ A \mid c$

I_2: $S \rightarrow A\ .\ S$

$S \rightarrow .\ A\ S$

$S \rightarrow .\ b$

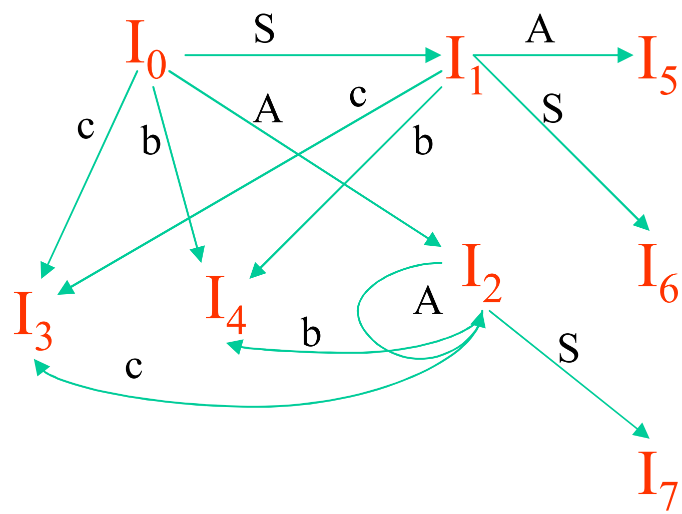
$A \rightarrow .\ S\ A$

$A \rightarrow .\ c$

I_3: $A \rightarrow c\ .$

I_4: $S \rightarrow b\ .$

So far:



Example 2

Grammar: $S' \rightarrow S$, $S \rightarrow A\ S \mid b$, $A \rightarrow S\ A \mid c$

I_5: $S \rightarrow A\ .\ S$
$A \rightarrow S\ A\ .$
$S \rightarrow .\ A\ S$
$S \rightarrow .\ b$
$A \rightarrow .\ S\ A$
$A \rightarrow .\ c$

I_6: $A \rightarrow S\ .\ A$
$A \rightarrow .\ S\ A$
$A \rightarrow .\ c$
$S \rightarrow .\ A\ S$
$S \rightarrow .\ b$

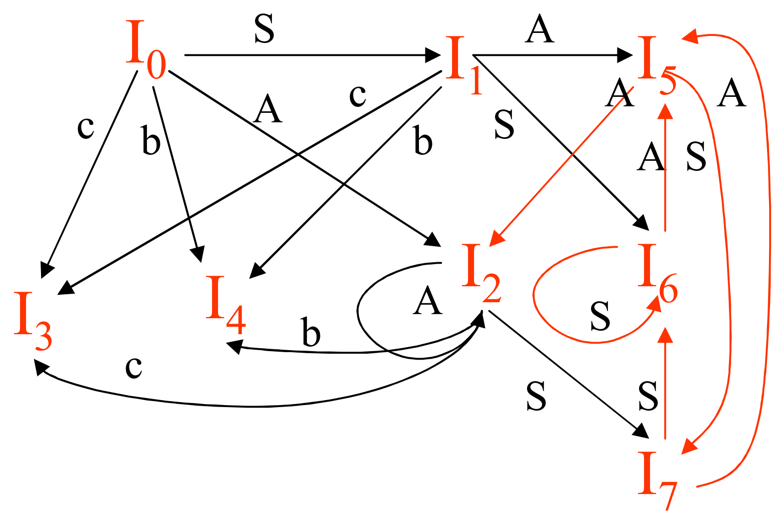
I_7: $S \rightarrow A\ S\ .$
$A \rightarrow S\ .\ A$
$A \rightarrow .\ S\ A$
$A \rightarrow .\ c$
$S \rightarrow .\ A\ S$
$S \rightarrow .\ b$

Example 2

Grammar: S' → S, S → A S | b, A → S A | c

I_0: S' → . S
I_1: S' → S .
  A → S . A
I_2: S → A . S
I_3: A → c .
I_4: S → b .
I_5: S → A . S
  A → S A .
I_6: A → S . A
I_7: S → A S .
  A → S . A

So far:



I5—I7 also have connections to I3 and I4

Generating SLR parse tables

- Construct C = {…} the LR(0) items as in previous slides
- **Action table for state *i* of parser:**
  - **If [A → α . a β] in I_i, goto(I_i,a) = I_j then**
    action[i,a] = *shift j*
  - **If [A → α .,b] in I_i, where A is not S', then**
    action[i,a] = *reduce A* → α for all a in FOLLOW(A)
  - **If [S' → S,$] in I_i, set action[i,$] = *accept***

  All undefined entries are *error*
- **Goto Table for state i of parser:**
  - **If [A → α . B] in I_i and goto(I_i,B) = I_j then**
    goto[i,B] = j

Example 2

Grammar: $S' \rightarrow S$, $S \rightarrow A\ S \mid b$, $A \rightarrow S\ A \mid c$

	First	Follow
S’	cb	$
S	cb	$cb
A	cb	cb

Example 2

Grammar: S' → S, S → A S | b, A → S A | c

I_0: S' → . S goto 1
S → . A S goto 2
S → . b goto 3
A → . S A goto 1
A → . c goto 4

I_1: S' → S . reduce
A → S . A goto 5
A → . S A goto 6
A → . c goto 4
S → . A S goto 5
S → . b goto 3

State	c	b	$	S	A
0	s4	s3		1	2
1	s4	s3	acc	6	5
2					
3					
4					
5					
6					
7					
8					

Example 2

Grammar: $S' \rightarrow S$, $S \rightarrow A\ S \mid b$, $A \rightarrow S\ A \mid c$

I_2: $S \rightarrow A\ .\ S$

$S \rightarrow .\ A\ S$

$S \rightarrow .\ b$

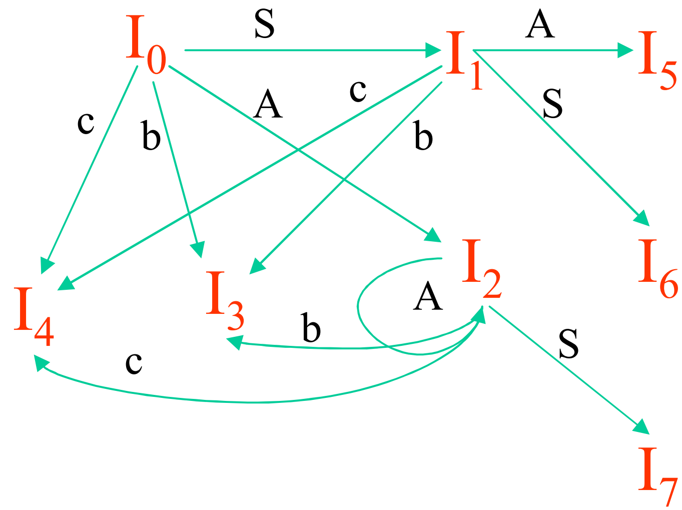
$A \rightarrow .\ S\ A$

$A \rightarrow .\ A$

I_3: $S \rightarrow b\ .$

I_4: $A \rightarrow c\ .$

So far:



LR Table for Example 2

State	c	b	$	S	A
0	s4	s3		1	2
1	s4	s3	acc	6	5
2	s4	s3		7	2
3	r3	r3	r3		
4	r5	r5			
5					
6					
7					
8					

1: S’ → S

2: S → A S

3: S → b

4: A → S A

5: A→ c

Example 2

Grammar: $S' \rightarrow S$, $S \rightarrow A\ S \mid b$, $A \rightarrow S\ A \mid c$

I_5: $S \rightarrow A\ .\ S$
$A \rightarrow S\ A\ .$
$S \rightarrow .\ A\ S$
$S \rightarrow .\ b$
$A \rightarrow .\ S\ A$
$A \rightarrow .\ c$

I_6: $A \rightarrow S\ .\ A$
$A \rightarrow .\ S\ A$
$A \rightarrow .\ c$
$S \rightarrow .\ A\ S$
$S \rightarrow .\ b$

I_7: $S \rightarrow A\ S\ .$
$A \rightarrow S\ .\ A$
$A \rightarrow .\ S\ A$
$A \rightarrow .\ c$
$S \rightarrow .\ A\ S$
$S \rightarrow .\ b$

LR Table for Example 2

State	c	b	$	S	A
0	s4	s3		1	2
1	s4	s3	acc	6	5
2	s4	s3		7	2
3	r3	r3	r3		
4	r5	r5			
5	s4/r4	s3/r4		7	2
6	s4	s3		6	5
7	s4/r2	s3/r2	r2	6	5

1: S’ → S

2: S → A S

3: S → b

4: A → S A

5: A→ c

LR Conflicts

- Shift/reduce
  – When it cannot be determined whether to shift the next symbol or reduce by a production
  – Typically, the default is to shift.
  – Examples: previous grammar, dangling else

if_stmt → if expr then stmt | if expr then stmt else stmt

if ex1 then
  if ex2 then
   stmt;
  else ← which '*if*' owns this *else*??

LR Conflicts

- Reduce/reduce
  - When it cannot be determined which production to reduce by
  - Example:

    stmt → id (expr_list)   ← function call

    expr → id (expr_list)   ← array (as in Ada)
  - Convention: use first production in grammar or use more powerful technique

Error Recovery in LR parsing

Just as with LL, we typically want to discard some part of the input and resume parsing from some ‘known’ point.

– Search back in the stack for some non-terminal A (how to choose A?) then process input until find token in Follow(A)
– Can also decorate the LR table with error recovery routines tailored to the state and token – more complicated to get right.