

# LR(1) Parsers

Copyright 2010, Keith D. Cooper & Linda Torczon, all rights reserved.

Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

# Building LR(1) Tables

---

How do we build the parse tables for an LR(1) grammar?

- Encode actions & transitions into the ACTION & GOTO tables
- If construction succeeds, the grammar is LR(1)
  - “Succeeds” means defines each table entry uniquely

## The Big Picture

- Model the state of the parser with “LR(1) items”
- The states will be set of LR(1) items
- Use two functions  $\text{goto}(s, X)$  and  $\text{closure}(s)$ 
  - $\text{goto}()$  tells which state you reach
  - $\text{closure}()$  adds information to round out a state
- Build up the states (sets of LR(1) items) and transitions
- Use this information to fill in the ACTION and GOTO tables

$s$  is a state  
 $X$  is T or NT

fixed-point algorithm

## LR(1) Items

---

We represent a valid configuration of an LR(1) parser with a data structure called an LR(1) item

An LR(1) item is a pair  $[P, \delta]$ , where

$P$  is a production  $A \rightarrow \beta$  with a  $\cdot$  at some position in the rhs

$\delta$  is a lookahead string of length  $\leq 1$  (word or EOF)

The  $\cdot$  in an item indicates which portion of the righthandside of the production we have on the top of the stack

## Meaning of an LR(1) Item

---

$[A \rightarrow \cdot \beta \gamma, \underline{a}]$  means that the input seen so far is consistent with the use of

$A \rightarrow \beta \gamma$  immediately after the symbol on top of the stack

"possibility"

$[A \rightarrow \beta \cdot \gamma, \underline{a}]$  means that the input seen so far is consistent with the use of

$A \rightarrow \beta \gamma$  at this point in the parse, and that the parser has already recognized  $\beta$  (that is,  $\beta$  is on top of the stack)

"partially complete"

$[A \rightarrow \beta \gamma \cdot, \underline{a}]$  means that the parser has seen  $\beta \gamma$ , and that a lookahead symbol of  $\underline{a}$  is consistent with reducing to  $A$

"complete"

## LR(1) Items

---

The production  $A \rightarrow \beta$ , where  $\beta = B_1 B_2 B_3$  with lookahead  $\underline{a}$ , can give rise to 4 items

$[A \rightarrow \cdot B_1 B_2 B_3, \underline{a}]$ ,  $[A \rightarrow B_1 \cdot B_2 B_3, \underline{a}]$ ,  $[A \rightarrow B_1 B_2 \cdot B_3, \underline{a}]$ , &  $[A \rightarrow B_1 B_2 B_3 \cdot, \underline{a}]$

The set of LR(1) items for a grammar is **finite**

What's the point of all these **lookahead** symbols?

- Carry them along to help choose the correct reduction
- Lookaheads are bookkeeping, unless item has  $\cdot$  at right end
  - Has no direct use in  $[A \rightarrow \beta \cdot \gamma, \underline{a}]$
  - In  $[A \rightarrow \beta \cdot, \underline{a}]$ , a lookahead of  $\underline{a}$  implies a reduction by  $A \rightarrow \beta$
  - For a parser state modeled with items  $\{ [A \rightarrow \beta \cdot, \underline{a}], [B \rightarrow \gamma \cdot \delta, \underline{b}] \}$ , lookahead of  $\underline{a} \Rightarrow$  reduce to  $A$ ; lookahead in  $\text{FIRST}(\delta) \Rightarrow$  shift

$\Rightarrow$  Limited right context is enough to pick the actions

# LR(1) Table Construction

For convenience, we will require that the grammar have an obvious & unique initial symbol — one that does not appear on the rhs of any production.

## High-level overview

- 1 Build the canonical collection of sets of LR(1) Items
    - a Start with an appropriate initial state  $s_0$ 
      - ◆  $[S' \rightarrow \cdot S \text{ EOF}]$ , along with any equivalent items
      - ◆ Derive equivalent items as  $\text{closure}(s_0)$
    - b Repeatedly compute, for each  $s_k$ , and each symbol  $X$ ,  $\text{goto}(s_k, X)$ 
      - ◆ If the set is not already in the collection, add it
      - ◆ Record all the transitions created by  $\text{goto}()$
- This eventually reaches a fixed point

# Computing Closures

Closure( $s$ ) adds all the items implied by the items already in  $s$

- Item  $[A \rightarrow \beta \cdot C \delta, \underline{a}]$  in  $s$  implies  $[C \rightarrow \cdot \tau, \underline{x}]$  for each production with  $C$  on the lhs, and each  $\underline{x} \in \text{FIRST}(\delta \underline{a})$
- Since  $\beta C \delta$  is a valid rewriting, any way to derive  $\beta C \delta$  is a valid rewriting, too

The algorithm

```
Closure( s )
while ( s is still changing )
   $\forall$  items  $[A \rightarrow \beta \cdot C \delta, \underline{a}] \in s$ 
     $\forall$  productions  $C \rightarrow \tau \in P$ 
       $\forall \underline{x} \in \text{FIRST}(\delta \underline{a})$  //  $\delta$  might be  $\epsilon$ 
        if  $[C \rightarrow \cdot \tau, \underline{x}] \notin s$ 
          then  $s \leftarrow s \cup \{ [C \rightarrow \cdot \tau, \underline{x}] \}$ 
```

- Classic fixed-point method
- Halts because  $s \subset \text{ITEMS}$
- Closure "fills out" a state

Lookaheads are generated here

# Example From SheepNoise

0	<i>Goal</i>	→	<i>SheepNoise</i>
1	<i>SheepNoise</i>	→	<i>SheepNoise</i> <u>baa</u>
2			<u>baa</u>

Initial step builds the item  $[Goal \rightarrow \bullet SheepNoise, EOF]$   
and takes its closure( )

Closure(  $[Goal \rightarrow \bullet SheepNoise, EOF]$  )

#	Item	Derived from ...
1	$[Goal \rightarrow \bullet SheepNoise, EOF]$	Original item
2	$[SheepNoise \rightarrow \bullet SheepNoise \underline{baa}, EOF]$	1, $\delta_a$ is <u>EOF</u>
3	$[SheepNoise \rightarrow \bullet \underline{baa}, EOF]$	1, $\delta_a$ is <u>EOF</u>
4	$[SheepNoise \rightarrow \bullet SheepNoise \underline{baa}, \underline{baa}]$	2, $\delta_a$ is <u>baa</u>
5	$[SheepNoise \rightarrow \bullet \underline{baa}, \underline{baa}]$	2, $\delta_a$ is <u>baa</u>

stop!

4  $\delta_a$  is baa baa

$S_0$  (the first state) is

{  $[Goal \rightarrow \bullet SheepNoise, EOF]$ ,  $[SheepNoise \rightarrow \bullet SheepNoise \underline{baa}, EOF]$ ,  
 $[SheepNoise \rightarrow \bullet \underline{baa}, EOF]$ ,  $[SheepNoise \rightarrow \bullet SheepNoise \underline{baa}, \underline{baa}]$ ,  
 $[SheepNoise \rightarrow \bullet \underline{baa}, \underline{baa}]$  }



# Computing Gotos

---

$Goto(s, x)$  computes the state that the parser would reach if it recognized an  $x$  while in state  $s$

- $Goto(\{ [A \rightarrow \beta \cdot X \delta, \underline{a}] \}, X)$  produces  $[A \rightarrow \beta X \cdot \delta, \underline{a}]$  (obviously)
- It finds all such items & uses  $closure()$  to fill out the state

## The algorithm

```
Goto( s, X )  
  new  $\leftarrow \emptyset$   
   $\forall$  items  $[A \rightarrow \beta \cdot X \delta, \underline{a}] \in s$   
    new  $\leftarrow$  new  $\cup \{ [A \rightarrow \beta X \cdot \delta, \underline{a}] \}$   
  return closure(new)
```

- Not a fixed-point method!
- Straightforward computation
- Uses  $closure()$

# Example from SheepNoise

0	Goal	→	SheepNoise
1	SheepNoise	→	SheepNoise <u>baa</u>
2			<u>baa</u>

$S_0$  is { [Goal→ • SheepNoise, EOF], [SheepNoise→ • SheepNoise baa, EOF],  
[SheepNoise→ • baa, EOF], [SheepNoise→ • SheepNoise baa, baa],  
[SheepNoise→ • baa, baa] }

Goto(  $S_0$  , baa )

- Loop produces

Item	Source
[SheepNoise → <u>baa</u> •, <u>EOF</u> ]	Item 3 in $s_0$
[SheepNoise → <u>baa</u> •, <u>baa</u> ]	Item 5 in $s_0$

- Closure adds nothing since • is at end of rhs in each item

# Building the Canonical Collection : The algorithm

```
s0 ← closure( [S' → ·S, EOF] )
S ← { s0 }
k ← 1
while ( S is still changing )
  ∀ sj ∈ S and ∀ x ∈ ( T ∪ NT )
    † ← goto( sj, x )
    if † ∉ S then
      name closure( † ) as sk
      S ← S ∪ { sk }
      record sj → sk on x
      k ← k + 1
    else
      † is sm ∈ S
      record sj → sm on x
```

Start from  $s_0 = \text{closure}( [S' \rightarrow \cdot S, \underline{\text{EOF}}] )$

Repeatedly construct new states, until all are found

- Fixed-point computation
- Loop adds to  $S$
- $S \subseteq 2^{\text{ITEMS}}$ , so  $S$  is finite

# Example from SheepNoise

0	Goal	→	SheepNoise
1	SheepNoise	→	SheepNoise <u>baa</u>
2			<u>baa</u>

Starts with  $S_0$

$S_0 : \{ [Goal \rightarrow \cdot SheepNoise, \underline{EOF}], [SheepNoise \rightarrow \cdot SheepNoise \underline{baa}, \underline{EOF}], [SheepNoise \rightarrow \cdot \underline{baa}, \underline{EOF}], [SheepNoise \rightarrow \cdot SheepNoise \underline{baa}, \underline{baa}], [SheepNoise \rightarrow \cdot \underline{baa}, \underline{baa}] \}$

Iteration 1 computes

$S_1 = Goto(S_0, SheepNoise) =$

$\{ [Goal \rightarrow SheepNoise \cdot, \underline{EOF}], [SheepNoise \rightarrow SheepNoise \cdot \underline{baa}, \underline{EOF}], [SheepNoise \rightarrow SheepNoise \cdot \underline{baa}, \underline{baa}] \}$

No more for closure!

$S_2 = Goto(S_0, \underline{baa}) = \{ [SheepNoise \rightarrow \underline{baa} \cdot, \underline{EOF}], [SheepNoise \rightarrow \underline{baa} \cdot, \underline{baa}] \}$

No more for closure!

Iteration 2 computes

$S_3 = Goto(S_1, \underline{baa}) = \{ [SheepNoise \rightarrow SheepNoise \underline{baa} \cdot, \underline{EOF}], [SheepNoise \rightarrow SheepNoise \underline{baa} \cdot, \underline{baa}] \}$

No more for closure!

# Example from SheepNoise

0	Goal	→	SheepNoise
1	SheepNoise	→	SheepNoise <u>baa</u>
2			<u>baa</u>

$S_0 : \{ [Goal \rightarrow \cdot SheepNoise, \underline{EOF}], [SheepNoise \rightarrow \cdot SheepNoise \underline{baa}, \underline{EOF}],$   
 $[SheepNoise \rightarrow \cdot \underline{baa}, \underline{EOF}], [SheepNoise \rightarrow \cdot SheepNoise \underline{baa}, \underline{baa}],$   
 $[SheepNoise \rightarrow \cdot \underline{baa}, \underline{baa}] \}$

$S_1 = Goto(S_0, SheepNoise) =$   
 $\{ [Goal \rightarrow SheepNoise \cdot, \underline{EOF}], [SheepNoise \rightarrow SheepNoise \cdot \underline{baa}, \underline{EOF}],$   
 $[SheepNoise \rightarrow SheepNoise \cdot \underline{baa}, \underline{baa}] \}$

$S_2 = Goto(S_0, \underline{baa}) = \{ [SheepNoise \rightarrow \underline{baa} \cdot, \underline{EOF}],$   
 $[SheepNoise \rightarrow \underline{baa} \cdot, \underline{baa}] \}$

$S_3 = Goto(S_1, \underline{baa}) = \{ [SheepNoise \rightarrow SheepNoise \underline{baa} \cdot, \underline{EOF}],$   
 $[SheepNoise \rightarrow SheepNoise \underline{baa} \cdot, \underline{baa}] \}$

# Filling in the ACTION and GOTO Tables

The algorithm

x is the state number

$\forall$  set  $S_x \in S$

$\forall$  item  $i \in S_x$

- case 1 { if  $i$  is  $[A \rightarrow \beta \cdot \underline{a} \delta, \underline{b}]$  and  $\text{goto}(S_x, \underline{a}) = S_k, \underline{a} \in T$  | • before terminal  
then  $\text{ACTION}[x, \underline{a}] \leftarrow$  "shift k"  $\Rightarrow$  shift
- case 2 { else if  $i$  is  $[S' \rightarrow S \cdot \underline{\text{EOF}}]$  | have accept  
then  $\text{ACTION}[x, \underline{\text{EOF}}] \leftarrow$  "accept"
- case 3 { else if  $i$  is  $[A \rightarrow \beta \cdot \underline{a}]$  | • at end  $\Rightarrow$  reduce  
then  $\text{ACTION}[x, \underline{a}] \leftarrow$  "reduce  $A \rightarrow \beta$ "

$\forall n \in NT$

if  $\text{goto}(S_x, n) = S_k$

then  $\text{GOTO}[x, n] \leftarrow k$

# Example from SheepNoise

0	Goal	→	SheepNoise
1	SheepNoise	→	SheepNoise <u>baa</u>
2			<u>baa</u>

$S_0 : \{ [Goal \rightarrow \cdot SheepNoise, EOF], [SheepNoise \rightarrow \cdot SheepNoise \underline{baa}, EOF],$   
 $[SheepNoise \rightarrow \cdot \underline{baa}, EOF], [SheepNoise \rightarrow \cdot SheepNoise \underline{baa}, \underline{baa}],$   
 $[SheepNoise \rightarrow \cdot \underline{baa}, \underline{baa}] \}$

• before terminal  $\Rightarrow$  shift (k)

$S_1 = Goto(S_0, SheepNoise) =$   
 $\{ [Goal \rightarrow SheepNoise \cdot, EOF], [SheepNoise \rightarrow SheepNoise \cdot \underline{baa}, EOF],$   
 $[SheepNoise \rightarrow SheepNoise \cdot \underline{baa}, \underline{baa}] \}$

so, ACTION[ $s_0, \underline{baa}$ ] is "shift  $S_2$ " (case 1)

$S_2 = Goto(S_0, \underline{baa}) =$

$\{ [SheepNoise \rightarrow \underline{baa} \cdot, EOF],$   
 $[SheepNoise \rightarrow \underline{baa} \cdot, \underline{baa}] \}$

$S_3 = Goto(S_1, \underline{baa}) = \{ [SheepNoise \rightarrow SheepNoise \underline{baa} \cdot, EOF],$   
 $[SheepNoise \rightarrow SheepNoise \underline{baa} \cdot, \underline{baa}] \}$

# Example from SheepNoise

0	Goal	→	SheepNoise
1	SheepNoise	→	SheepNoise <u>baa</u>
2			<u>baa</u>

$S_0 : \{ [Goal \rightarrow \cdot SheepNoise, EOF], [SheepNoise \rightarrow \cdot SheepNoise \underline{baa}, EOF],$   
 $[SheepNoise \rightarrow \cdot \underline{baa}, EOF], [SheepNoise \rightarrow \cdot SheepNoise \underline{baa}, \underline{baa}],$   
 $[SheepNoise \rightarrow \cdot \underline{baa}, \underline{baa}] \}$

$S_1 = Goto(S_0, SheepNoise) =$

$\{ [Goal \rightarrow SheepNoise \cdot, EOF], [SheepNoise \rightarrow SheepNoise \cdot \underline{baa}, EOF],$   
 $[SheepNoise \rightarrow SheepNoise \cdot \underline{baa}, \underline{baa}] \}$

$S_2 = Goto(S_0, \underline{baa}) = \{ [SheepNoise \rightarrow \underline{baa} \cdot, EOF],$   
 $[SheepNoise \rightarrow \underline{baa} \cdot, \underline{baa}] \}$

$S_3 = Goto(S_1, \underline{baa}) = \{ [SheepNoise \rightarrow SheepNoise \underline{baa} \cdot, EOF],$   
 $[SheepNoise \rightarrow SheepNoise \underline{baa} \cdot, \underline{baa}] \}$

so, ACTION[ $S_1, \underline{baa}$ ] is  
"shift  $S_3$ " (case 1)



# Example from SheepNoise

0	Goal	→	SheepNoise
1	SheepNoise	→	SheepNoise <u>baa</u>
2			<u>baa</u>

$S_0 : \{ [Goal \rightarrow \cdot SheepNoise, \underline{EOF}], [SheepNoise \rightarrow \cdot SheepNoise \underline{baa}, \underline{EOF}], [SheepNoise \rightarrow \cdot \underline{baa}, \underline{EOF}], [SheepNoise \rightarrow \cdot SheepNoise \underline{baa}, \underline{baa}], [SheepNoise \rightarrow \cdot \underline{baa}, \underline{baa}] \}$

$S_1 = Goto(S_0, SheepNoise) =$

$\{ [Goal \rightarrow SheepNoise \cdot, \underline{EOF}], [SheepNoise \rightarrow SheepNoise \cdot \underline{baa}, \underline{EOF}], [SheepNoise \rightarrow SheepNoise \cdot \underline{baa}, \underline{baa}] \}$

so, ACTION[S<sub>1</sub>,EOF] is "accept" (case 2)

$S_2 = Goto(S_0, \underline{baa}) = \{ [SheepNoise \rightarrow \underline{baa} \cdot, \underline{EOF}], [SheepNoise \rightarrow \underline{baa} \cdot, \underline{baa}] \}$

$S_3 = Goto(S_1, \underline{baa}) = \{ [SheepNoise \rightarrow SheepNoise \underline{baa} \cdot, \underline{EOF}], [SheepNoise \rightarrow SheepNoise \underline{baa} \cdot, \underline{baa}] \}$

# Example from SheepNoise

0	Goal	→	SheepNoise
1	SheepNoise	→	SheepNoise <u>baa</u>
2			<u>baa</u>

$S_0 : \{ [Goal \rightarrow \cdot SheepNoise, EOF], [SheepNoise \rightarrow \cdot SheepNoise \underline{baa}, EOF], [SheepNoise \rightarrow \cdot \underline{baa}, EOF], [SheepNoise \rightarrow \cdot SheepNoise \underline{baa}, \underline{baa}], [SheepNoise \rightarrow \cdot \underline{baa}, \underline{baa}] \}$

$S_1 = Goto(S_0, SheepNoise) =$

$\{ [Goal \rightarrow SheepNoise \cdot, EOF], [SheepNoise \rightarrow SheepNoise \cdot \underline{baa}, EOF], [SheepNoise \rightarrow SheepNoise \cdot \underline{baa}, \underline{baa}] \}$

$S_2 = Goto(S_0, \underline{baa}) = \{ [SheepNoise \rightarrow \underline{baa} \cdot, EOF], [SheepNoise \rightarrow \underline{baa} \cdot, \underline{baa}] \}$

so, ACTION[S<sub>2</sub>,EOF] is "reduce 2" (case 3)

ACTION[S<sub>2</sub>,baa] is "reduce 2" (case 3)

$S_3 = Goto(S_1, \underline{baa}) = \{ [SheepNoise \rightarrow SheepNoise \underline{baa} \cdot, EOF], [SheepNoise \rightarrow SheepNoise \underline{baa} \cdot, \underline{baa}] \}$

# Example from SheepNoise

0	Goal	→	SheepNoise
1	SheepNoise	→	SheepNoise <u>baa</u>
2			<u>baa</u>

$S_0 : \{ [Goal \rightarrow \cdot SheepNoise, EOF], [SheepNoise \rightarrow \cdot SheepNoise \underline{baa}, EOF], [SheepNoise \rightarrow \cdot \underline{baa}, EOF], [SheepNoise \rightarrow \cdot SheepNoise \underline{baa}, \underline{baa}], [SheepNoise \rightarrow \cdot \underline{baa}, \underline{baa}] \}$

$S_1 = Goto(S_0, SheepNoise) =$

ACTION[S<sub>3</sub>,EOF] is "reduce 1" (case 3)

$\{ [SheepNoise \rightarrow \cdot \underline{baa}, EOF], [SheepNoise \rightarrow \cdot \underline{baa}, \underline{baa}] \}$

$S_2 = Goto(S_0, \underline{baa}) = \{ [SheepNoise \rightarrow \underline{baa} \cdot, EOF], [SheepNoise \rightarrow \underline{baa} \cdot, \underline{baa}] \}$

$S_3 = Goto(S_1, \underline{baa}) = \{ [SheepNoise \rightarrow SheepNoise \underline{baa} \cdot, EOF], [SheepNoise \rightarrow SheepNoise \underline{baa} \cdot, \underline{baa}] \}$

ACTION[S<sub>3</sub>,baa] is "reduce 1", as well

0	Goal	→	SheepNoise
1	SheepNoise	→	SheepNoise <u>baa</u>
2			<u>baa</u>

# Example from SheepNoise

The GOTO Table records Goto transitions on NTs

$s_0 : \{ [Goal \rightarrow \cdot SheepNoise, EOF], [SheepNoise \rightarrow \cdot SheepNoise \underline{baa}, EOF], [SheepNoise \rightarrow \cdot \underline{baa}, EOF], [SheepNoise \rightarrow \cdot SheepNoise \underline{baa}, \underline{baa}], [SheepNoise \rightarrow \cdot \underline{baa}, \underline{baa}] \}$

$s_1 = Goto(S_0, SheepNoise) =$    | Puts  $s_1$  in  $GOTO[s_0, SheepNoise]$

$\{ [Goal \rightarrow SheepNoise \cdot, EOF], [SheepNoise \rightarrow SheepNoise \cdot \underline{baa}, EOF], [SheepNoise \rightarrow SheepNoise \cdot \underline{baa}, \underline{baa}] \}$

$s_2 = Goto(S_0, \underline{baa}) = \{ [SheepNoise \rightarrow \underline{baa} \cdot, EOF], [SheepNoise \rightarrow \underline{baa} \cdot, \underline{baa}] \}$  Based on T, not NT and written into the ACTION table

$s_3 = Goto(S_1, \underline{baa}) = \{ [SheepNoise \rightarrow SheepNoise \underline{baa} \cdot, EOF], [SheepNoise \rightarrow SheepNoise \underline{baa} \cdot, \underline{baa}] \}$

Only 1 transition in the entire GOTO table

Remember, we recorded these so we don't need to recompute them.

## ACTION & GOTO Tables

0	<i>Goal</i>	→	<i>SheepNoise</i>
1	<i>SheepNoise</i>	→	<i>SheepNoise</i> <u>baa</u>
2			<u>baa</u>

Here are the tables for the augmented left-recursive SheepNoise grammar

The tables

ACTION TABLE		
State	EOF	<u>baa</u>
0	—	<i>shift 2</i>
1	<i>accept</i>	<i>shift 3</i>
2	<i>reduce 2</i>	<i>reduce 2</i>
3	<i>reduce 1</i>	<i>reduce 1</i>

GOTO TABLE	
State	<i>SheepNoise</i>
0	1
1	0
2	0
3	0

Note that this is the left-recursive SheepNoise; the book shows the right-recursive version.

# What can go wrong?

---

What if set  $s$  contains  $[A \rightarrow \beta \cdot \underline{a} \gamma, \underline{b}]$  and  $[B \rightarrow \beta \cdot \underline{a}]$  ?

- First item generates "shift", second generates "reduce"
- Both define  $ACTION[s, \underline{a}]$  — cannot do both actions
- This is a fundamental ambiguity, called a shift/reduce error
- Modify the grammar to eliminate it (if-then-else)
- Shifting will often resolve it correctly

What if set  $s$  contains  $[A \rightarrow \gamma \cdot, \underline{a}]$  and  $[B \rightarrow \gamma \cdot, \underline{a}]$  ?

- Each generates "reduce", but with a different production
- Both define  $ACTION[s, \underline{a}]$  — cannot do both reductions
- This is a fundamental ambiguity, called a reduce/reduce conflict
- Modify the grammar to eliminate it

In either case, the grammar is not LR(1)

# LR(k) versus LL(k)

---

## Finding Reductions

LR(k)  $\Rightarrow$  Each reduction in the parse is detectable with

- $\rightarrow$  the complete left context,
- $\rightarrow$  the reducible phrase, itself, and
- $\rightarrow$  the  $k$  terminal symbols to its right

generalizations of  
LR(1) and LL(1) to  
longer lookaheads

LL(k)  $\Rightarrow$  Parser must select the reduction based on

- $\rightarrow$  The complete left context
- $\rightarrow$  The next  $k$  terminals

Thus, LR(k) examines more context

# Non-LL Grammars

---

0	$B \rightarrow R$
1	$(B)$
2	$R \rightarrow E = E$
3	$E \rightarrow \underline{a}$
4	$\underline{b}$
5	$(E + E)$

Example from D.E Knuth, "Top-Down Syntactic Analysis," Acta Informatica, 1:2 (1971), pages 79-110

This grammar is actually LR(0)

0	$S \rightarrow \underline{a} A \underline{b}$
1	$\underline{c}$
2	$A \rightarrow \underline{b} S$
3	$B \underline{b}$
4	$B \mid \underline{a} A$
5	$\underline{c}$

Example from Lewis, Rosenkrantz, & Stearns book, "Compiler Design Theory," (1976), Figure 13.1



# Summary

---

	Advantages	Disadvantages
Top-down Recursive descent, LL(1)	Fast Good locality Simplicity Good error detection	Hand-coded High maintenance Right associativity
LR(1)	Fast Deterministic langs. Automatable Left associativity	Large working sets Poor error messages Large table sizes

# Exercise

---

Consider the following grammar:

$$\begin{aligned} \textit{Start} &\rightarrow S \\ S &\rightarrow A a \\ A &\rightarrow B C \\ &\quad | B C f \\ B &\rightarrow b \\ C &\rightarrow c \end{aligned}$$

- a. Construct the canonical collection of sets of LR(1) items for this grammar.
- b. Derive the Action and Goto tables.
- c. Is the grammar LR(1)?

Parse the string *bcfa* and the string *bca*





