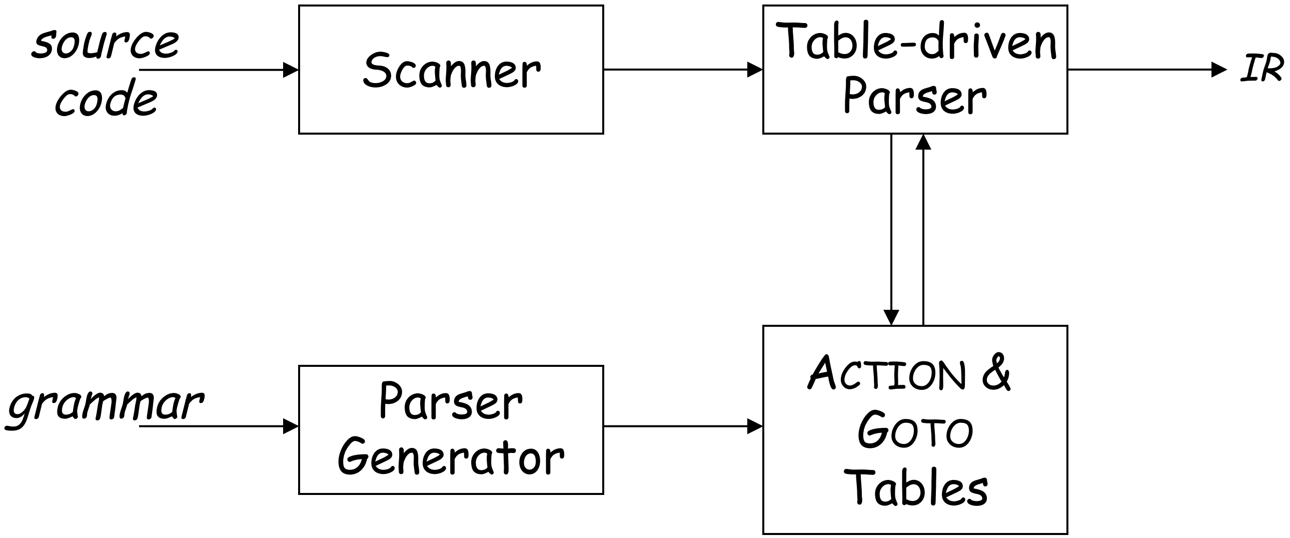
LR(1) Parsers
Part III
Last Parsing Lecture

LR(1) Parsers

A table-driven LR(1) parser looks like



Tables *can* be built by hand

However, this is a perfect task to automate

Bottom-up Parser

A simple *shift-reduce parser:*

```
push INVALID
token ← next_token( )
repeat until (top of stack = Goal and token = EOF)
    if the top of the stack is a handle A→β
        then    // reduce β to A
            pop |β| symbols off the stack
            push A onto the stack
        else if (token ≠ EOF)
            then // shift
                push token
                token ← next_token( )
        else    // need to shift, but out of input
        report an error
```

LR(1) Parsers (*parse tables*)

To make a parser for *L(G)*, need a set of tables

The grammar

1	*Goal*	→	*SheepNoise*
2	*SheepNoise*	→	*SheepNoise* baa
3		|	baa

The tables

ACTION Table		
State	EOF	baa
0	—	*shift 2*
1	*accept*	*shift 3*
2	*reduce 3*	*reduce 3*
3	*reduce 2*	*reduce 2*

GOTO Table	
State	*SheepNoise*
0	1
1	0
2	0
3	0

LR(1) Parsers (*parse tables*)

To make a parser for *L(G)*, need a set of tables

The grammar

1	*Goal*	→	*SheepNoise*
2	*SheepNoise*	→	*SheepNoise* baa
3		|	baa

The tables

ACTION Table		
State	EOF	baa
0	—	*shift 2*
1	*accept*	*shift 3*
2	*reduce 3*	*reduce 3*
3	*reduce 2*	*reduce 2*

GOTO Table	
State	*SheepNoise*
0	1
1	0
2	0
3	0

Correspond to state

LR(1) Parsers (*parse tables*)

To make a parser for *L(G)*, need a set of tables

The grammar

1	*Goal*	→	*SheepNoise*
2	*SheepNoise*	→	*SheepNoise* baa
3		|	baa

The tables

ACTION Table		
State	EOF	baa
0	–	*shift 2*
1	*accept*	*shift 3*
2	*reduce 3*	*reduce 3*
3	*reduce 2*	*reduce 2*

GOTO Table	
State	*SheepNoise*
0	1
1	0
2	0
3	0

Correspond to production rule

Building LR(1) Tables : ACTION and GOTO

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

- Use grammar to build model of Control DFA
- ACTION table
  - Provides actions to perform
- GOTO table
  - Tells us state to goto next

- If table construction succeeds, the grammar is LR(1)

Building LR(1) Tables: The Big Picture

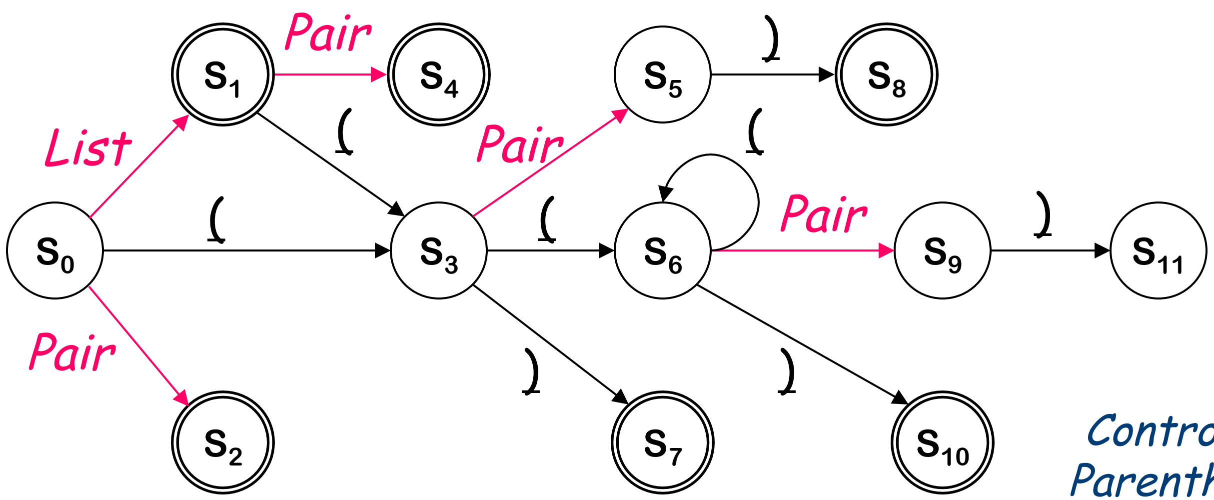
- Model the state of the parser with "LR(1) items"
- Use two functions:
  - *goto(s, X)*
  - *closure(s)*
- Build up states and transition functions of the DFA

Parenthesis Grammar

1	$Goal \rightarrow List$
2	$List \rightarrow List\ Pair$
3	$\mid\ Pair$
4	$Pair \rightarrow \underline{(}\ Pair\ \underline{)}$
5	$\mid\ \underline{(}\ \underline{)}$

LR(1) Parsers

The Control DFA for the Parentheses Language



Control DFA for the Parentheses Language

Transitions on terminals represent shift actions [ACTION]

Transitions on nonterminals represent reduce actions [GOTO]

The table construction derives this DFA from the grammar

LR(1) Items

LR(1) items represent set of valid states

An LR(1) item is a pair [*P*, δ], where

- *P* is a production *A*→β with a • at some position in the *rhs*
- δ is a lookahead string (*word or EOF*)

The • ("placeholder") in item indicates TOS position

LR(1) Items

[$A \rightarrow \bullet\beta\gamma$,<u>a</u>] means that input seen so far is consistent with use of $A \rightarrow \beta\gamma$ immediately after the symbol on TOS

"possibility"

[$A \rightarrow \beta\bullet\gamma$,<u>a</u>] means that input seen so far is consistent with use of $A \rightarrow \beta\gamma$ at this point in the parse, <u>*and*</u> that the parser has already recognized β (that is, β is on TOS)

"partially complete"

[$A \rightarrow \beta\gamma\bullet$,<u>a</u>] means that parser has seen $\beta\gamma$, <u>*and*</u> that a lookahead symbol of <u>a</u> is consistent with reducing to *A*.

"complete"

LR(1) Items

Production $A \rightarrow \beta$, $\beta = B_1 B_2 B_3$ and lookahead $\underline{a}$, gives rise to 4 items

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

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

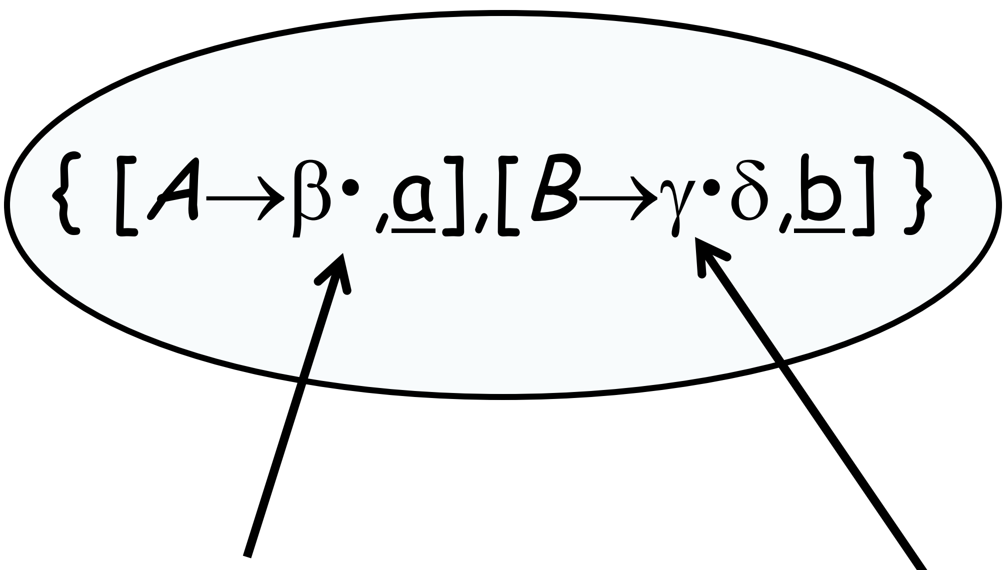
$[A \rightarrow B_1 B_2 \bullet B_3, \underline{a}]$

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

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

Lookahead symbols?

- Helps to choose the correct reduction



lookahead of a ⇒ *reduce* to *A*;

lookahead in FIRST(δ) ⇒ *shift*

LR(1) Table Construction : Overview

Build Canonical Collection (CC) of sets of LR(1) Items, I

Step 1: Start with initial state, s_0

- [$S' \rightarrow \cdot S$,EOF], along with any equivalent items
- Derive equivalent items as *closure*(s_0)

Grammar has an unique goal symbol

LR(1) Table Construction : Overview

Step 2: For each s_k, and each symbol *X*, compute *goto*(s_k,*X*)

- If the set is not already in CC, add it
- Record all the transitions created by *goto( )*

This eventually reaches a fixed point

LR(1) Table Construction : Overview

Step 3: Fill in the table from the collection of sets of LR(1) items

The states of canonical collection are precisely the states of the Control DFA

The construction traces the DFA's transitions

Computing Closures

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

s

$[A \rightarrow \beta \bullet C\delta, \underline{a}]$

Closure($[A \rightarrow \beta \bullet C\delta, \underline{a}]$) adds $[C \rightarrow \bullet \tau, x]$

where *C* is on the *lhs* and each $x \in$ FIRST($\delta \underline{a}$)

Since $\beta C \delta$ is valid, any way to derive $\beta C \delta$ is valid

Closure algorithm

Closure(s)
while (s is still changing)
$\forall$ *items* $[A \rightarrow \beta \bullet C\delta, \underline{a}] \in s$
$\forall$ *productions* $C \rightarrow \tau \in P$
$\forall\ \underline{x} \in \textsc{First}(\delta\underline{a})$ // δ might be ε
if $[C \rightarrow \bullet\ \tau, \underline{x}] \notin s$
then $s \leftarrow s \cup \{ [C \rightarrow \bullet\ \tau, \underline{x}] \}$

- Classic fixed-point method
- Halts because $s \subset$ Items
- *Closure "fills out" a state*

Closure algorithm

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

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

Closure algorithm

Closure(s)

  while (s is still changing)

    $\forall$ *items* $[A \rightarrow \beta \cdot C\delta, \underline{a}] \in s$

      $\forall$ *productions* $C \rightarrow \tau \in P$

        $\forall\ \underline{x} \in$ FIRST($\delta\underline{a}$)   // δ might be ε

          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 $s \subset$ ITEMS
- *Closure "fills out" a state*

Closure algorithm

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

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

Closure algorithm

Closure(s)

  while (s *is still changing*)

    $\forall$ *items* $[A \rightarrow \beta \bullet C\delta, \underline{a}] \in s$

      $\forall$ *productions* $C \rightarrow \tau \in P$

        $\forall\ \underline{x} \in \text{FIRST}(\delta\underline{a})$ // δ might be ε

          if $[C \rightarrow \bullet\ \tau, \underline{x}] \notin s$

            then $s \leftarrow s \cup \{ [C \rightarrow \bullet\ \tau, \underline{x}] \}$

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

Example From SheepNoise

Initial step builds the item [*Goal*→•*SheepNoise*,EOF] and takes its *closure( )*

0	*Goal*	→	*SheepNoise*
1	*SheepNoise*	→	*SheepNoise* baa
2		|	baa

*Closure([Goal→•SheepNoise,*EOF*])*

Example From SheepNoise

Initial step builds the item [*Goal*→•*SheepNoise*,EOF] and takes its *closure( )*

0	*Goal*	→	*SheepNoise*
1	*SheepNoise*	→	*SheepNoise* baa
2		|	baa

*Closure([Goal→•SheepNoise,*EOF*])*

#	Item	Derived from …
1	[*Goal* → • *SheepNoise*,EOF]	Original item
2	[*SheepNoise* → • *SheepNoise* baa, EOF]	1, δa is EOF
3	[*SheepNoise* → • baa, EOF]	1, δa is EOF
4	[*SheepNoise* → • *SheepNoise* baa, baa]	2, δa is baa baa
5	[*SheepNoise* → • baa, baa]	2, δa is baa baa

So, S_0 is

{ [*Goal*→ • *SheepNoise*,EOF], [*SheepNoise*→ • *SheepNoise* baa,EOF], [*SheepNoise*→• baa,EOF], [*SheepNoise*→ • *SheepNoise* baa,baa], [*SheepNoise*→ • baa,baa] }

Computing Gotos

Goto(s,x) computes state parser would reach if it recognized *x* while in state *s*

$$Goto(\{ [A \to \beta \bullet X \delta, \underline{a}] \}, X)$$

Produces

$$[A \to \beta X \bullet \delta, \underline{a}]$$

- Creates new LR(1) item & uses *closure()* to fill out the state

Goto Algorithm

Goto(s, X)
  new ← Ø
  ∀ *items* $[A \rightarrow \beta \cdot X \delta, \underline{a}] \in s$
    new ← *new* $\cup \{[A \rightarrow \beta X \cdot \delta, \underline{a}]\}$

  return closure(new)

- Not a fixed-point method!
- Uses *closure( )*
- *Goto() moves us forward*

Example from SheepNoise

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

0	*Goal*	→	*SheepNoise*
1	*SheepNoise*	→	*SheepNoise* baa
2		|	baa

Goto(S_0 , baa)

Example from SheepNoise

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

0	*Goal*	→	*SheepNoise*
1	*SheepNoise*	→	*SheepNoise* baa
2		|	baa

Goto(S_0 , baa)

- Loop produces

Item	*Source*
[*SheepNoise* → baa •, EOF]	Item 3 in s_0
[*SheepNoise* → baa •, baa]	Item 5 in s_0

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

In the construction, this produces s_2

{ [*SheepNoise*→baa •, {EOF,baa}] }

New, but *obvious*, notation for two distinct items
[*SheepNoise*→baa •, EOF] &
[*SheepNoise*→baa •, baa]

Canonical Collection Algorithm

$s_0 \leftarrow closure([S' \rightarrow \cdot\, S, \underline{EOF}])$
$S \leftarrow \{ s_0 \}$
$k \leftarrow 1$

while (S *is still changing*)
  $\forall s_j \in S$ *and* $\forall x \in (T \cup NT)$
    $t \leftarrow goto(s_j, x)$
    if $t \notin S$ *then*
      name t *as* s_k
      $S \leftarrow S \cup \{ s_k \}$
      record $s_j \rightarrow s_k$ *on* x
      $k \leftarrow k + 1$
    else
      t *is* $s_m \in S$
        record $s_j \rightarrow s_m$ *on* x

Add initial state; fill out state with closure

Canonical Collection Algorithm

$s_0 \leftarrow$ *closure*([$S' \rightarrow \cdot\ S$,EOF])
$S \leftarrow \{ s_0 \}$
$k \leftarrow 1$

while (*S is still changing*)
  $\forall s_j \in S$ *and* $\forall x \in (T \cup NT)$
    $t \leftarrow goto(s_j, x)$
    if $t \notin S$ *then*
      name t as s_k
      $S \leftarrow S \cup \{ s_k \}$
      record $s_j \rightarrow s_k$ *on x*
      $k \leftarrow k + 1$
    else
      t is $s_m \in S$
        record $s_j \rightarrow s_m$ *on x*

- Fixed-point computation
- Loop adds to *S*

Canonical Collection Algorithm

$s_0 \leftarrow closure(\,[S' \rightarrow \cdot\, S, \underline{EOF}]\,)$
$S \leftarrow \{\, s_0 \,\}$
$k \leftarrow 1$

$while\ (\,S\ is\ still\ changing\,)$
$\quad \forall\, s_j \in S\ and\ \forall\, x \in (\,T \cup NT\,)$
$\qquad t \leftarrow goto(s_j, x)$
$\qquad if\ t \notin S\ then$
$\qquad\quad name\ t\ as\ s_k$
$\qquad\quad S \leftarrow S \cup \{\, s_k \}$
$\qquad\quad record\ s_j \rightarrow s_k\ on\ x$
$\qquad\quad k \leftarrow k + 1$
$\qquad else$
$\qquad\quad t\ is\ s_m \in S$
$\qquad\qquad record\ s_j \rightarrow s_m\ on\ x$

- Iterate through all items in state and all symbols

Canonical Collection Algorithm

$s_0 \leftarrow$ *closure*([$S' \rightarrow \cdot\ S$,EOF])
$S \leftarrow \{ s_0 \}$
$k \leftarrow 1$

while (S *is still changing*)
  $\forall s_j \in S$ *and* $\forall x \in (T \cup NT)$
    $t \leftarrow goto(s_j,x)$
    if $t \notin S$ *then*
      name t *as* s_k
      $S \leftarrow S \cup \{ s_k \}$
      record $s_j \rightarrow s_k$ *on* x
      $k \leftarrow k + 1$
    else
      t *is* $s_m \in S$
        record $s_j \rightarrow s_m$ *on* x

- Call goto function to get transition from s_j to new state t

Canonical Collection Algorithm

$s_0 \leftarrow$ *closure*([$S' \rightarrow \cdot\ S$,EOF])
$S \leftarrow \{ s_0 \}$
$k \leftarrow 1$

while (S *is still changing*)
  $\forall s_j \in S$ *and* $\forall x \in (T \cup NT)$
    $t \leftarrow goto(s_j, x)$
    if $t \notin S$ *then*
      name t *as* s_k
      $S \leftarrow S \cup \{ s_k \}$
      record $s_j \rightarrow s_k$ *on* x
      $k \leftarrow k + 1$
    else
      t *is* $s_m \in S$
      record $s_j \rightarrow s_m$ *on* x

- Add t to CC and add transition in DFA

Canonical Collection Algorithm

$s_0 \leftarrow closure([S' \rightarrow \cdot\ S, \underline{EOF}])$
$S \leftarrow \{ s_0 \}$
$k \leftarrow 1$

while (S *is still changing*)
  $\forall s_j \in S$ *and* $\forall x \in (T \cup NT)$
    $t \leftarrow goto(s_j, x)$
    if $t \notin S$ *then*
      name t *as* s_k
      $S \leftarrow S \cup \{ s_k \}$
      record $s_j \rightarrow s_k$ *on* x
      $k \leftarrow k + 1$
    else
      t *is* $s_m \in S$
      record $s_j \rightarrow s_m$ *on* x

- t is already in CC; it is some state s_m add transition to DFA

Example from SheepNoise

Starts with S_0

S_0 : { [*Goal*→ • *SheepNoise*, EOF], [*SheepNoise*→ • *SheepNoise* baa, EOF], [*SheepNoise*→ • baa, EOF], [*SheepNoise*→ • *SheepNoise* baa, baa], [*SheepNoise*→ • baa, baa] }

$s_0 \leftarrow$ *closure*([*S'*→• *S*,EOF])
$S \leftarrow \{ s_0 \}$
$k \leftarrow 1$
...

0	*Goal*	→	*SheepNoise*
1	*SheepNoise*	→	*SheepNoise* baa
2		|	baa

Example from SheepNoise

Starts with S_0

S_0 : { [*Goal*→ • *SheepNoise*, EOF], [*SheepNoise*→ • *SheepNoise* baa, EOF], [*SheepNoise*→ • baa, EOF], [*SheepNoise*→ • *SheepNoise* baa, baa], [*SheepNoise*→ • baa, baa] }

Iteration 1 computes

S_1 = *Goto*(S_0 , *SheepNoise*) =
{ [*Goal*→ *SheepNoise* •, EOF], [*SheepNoise*→ *SheepNoise* • baa, EOF], [*SheepNoise*→ *SheepNoise* • baa, baa] }

...
while (*S is still changing*)
  ∀ s_j ∈ *S and* ∀ *x* ∈ (*T* ∪ *NT*)
    t ← *goto*(s_j,*x*)
...

0	*Goal*	→	*SheepNoise*
1	*SheepNoise*	→	*SheepNoise* baa
2		|	baa

Example from SheepNoise

Starts with S_0

S_0 : { [*Goal*→ • *SheepNoise*, EOF], [*SheepNoise*→ • *SheepNoise* baa, EOF],
[*SheepNoise*→ • baa, EOF], [*SheepNoise*→ • *SheepNoise* baa, baa],
[*SheepNoise*→ • baa, baa] }

Iteration 1 computes

S_1 = *Goto*(S_0 , *SheepNoise*) =
{ [*Goal*→ *SheepNoise* •, EOF], [*SheepNoise*→ *SheepNoise* • baa, EOF],
[*SheepNoise*→ *SheepNoise* • baa, baa] }

S_2 = *Goto*(S_0 , baa) = { [*SheepNoise*→ baa •, EOF],
[*SheepNoise*→ baa •, baa] }

0	*Goal*	→	*SheepNoise*
1	*SheepNoise*	→	*SheepNoise* baa
2		|	baa

Example from SheepNoise

S_1 = *Goto*(S_0 , *SheepNoise*) =

{ [*Goal*→ *SheepNoise* •, EOF], [*SheepNoise*→ *SheepNoise* • baa, EOF],
[*SheepNoise*→ *SheepNoise* • baa, baa] }

Nothing more to compute, since • is at the end of every item in S_3 .

Iteration 2 computes

S_3 = *Goto*(S_1 , baa) = { [*SheepNoise*→ *SheepNoise* baa •, EOF],
[*SheepNoise*→ *SheepNoise* baa •, baa] }

0	*Goal*	→	*SheepNoise*
1	*SheepNoise*	→	*SheepNoise* baa
2		|	baa

Example from SheepNoise

S_0 : { [*Goal*→ • *SheepNoise*, EOF], [*SheepNoise*→ • *SheepNoise* baa, EOF],
[*SheepNoise*→ • baa, EOF], [*SheepNoise*→ • *SheepNoise* baa, baa],
[*SheepNoise*→ • baa, baa] }

S_1 = *Goto*(S_0 , *SheepNoise*) =
{ [*Goal*→ *SheepNoise* •, EOF], [*SheepNoise*→ *SheepNoise* • baa, EOF],
[*SheepNoise*→ *SheepNoise* • baa, baa] }

S_2 = *Goto*(S_0 , baa) = { [*SheepNoise*→ baa •, EOF],
[*SheepNoise*→ baa •, baa] }

S_3 = *Goto*(S_1 , baa) = { [*SheepNoise*→ *SheepNoise* baa •, EOF],
[*SheepNoise*→ *SheepNoise* baa •, baa] }

0	*Goal*	→	*SheepNoise*
1	*SheepNoise*	→	*SheepNoise* baa
2		|	baa

Filling in the ACTION and GOTO Tables

The algorithm

$\forall$ *set* $S_x \in S$
  $\forall$ *item* $i \in S_x$
    if i *is* $[A \rightarrow \beta \bullet \underline{a}\delta, \underline{b}]$ *and* $goto(S_x, \underline{a}) = S_k$, $\underline{a} \in T$
      then ACTION[x,$\underline{a}$] $\leftarrow$ "*shift k*"
    else if i *is* $[S' \rightarrow S \bullet, \underline{EOF}]$
      then ACTION[x ,$\underline{EOF}$] $\leftarrow$ "*accept*"
    else if i *is* $[A \rightarrow \beta \bullet, \underline{a}]$
      then ACTION[x,$\underline{a}$] $\leftarrow$ "reduce $A \rightarrow \beta$"
  $\forall$ $n \in NT$
    if $goto(S_x, n) = S_k$
      then GOTO[x,n] $\leftarrow$ k

Fill ACTION table

Filling in the ACTION and GOTO Tables

The algorithm

x is the state number

$\forall$ *set* $S_x \in S$

$\quad \forall$ *item* $i \in S_x$

$\quad\quad$ *if* i *is* $[A \rightarrow \beta \bullet \underline{a}\delta, \underline{b}]$ *and* $goto(S_x, \underline{a}) = S_k$, $\underline{a} \in T$

$\quad\quad\quad$ *then* ACTION[x,$\underline{a}$] $\leftarrow$ "*shift k*"

$\quad\quad$ *else if* i *is* $[S' \rightarrow S \bullet, \underline{\text{EOF}}]$

$\quad\quad\quad$ *then* ACTION[x ,$\underline{\text{EOF}}$] $\leftarrow$ "*accept*"

$\quad\quad$ *else if* i *is* $[A \rightarrow \beta \bullet, \underline{a}]$

$\quad\quad\quad$ *then* ACTION[x,$\underline{a}$] $\leftarrow$ "reduce $A \rightarrow \beta$"

$\quad \forall$ $n \in NT$

$\quad\quad$ *if* $goto(S_x, n) = S_k$

$\quad\quad\quad$ *then* GOTO[x,n] $\leftarrow$ k

Filling in the ACTION and GOTO Tables

The algorithm

$\forall$ *set* $S_x \in S$

$\forall$ *item* $i \in S_x$

if i *is* $[A \rightarrow \beta \bullet \underline{a}\delta, \underline{b}]$ *and* $goto(S_x, \underline{a}) = S_k$, $\underline{a} \in T$

then ACTION[x,$\underline{a}$] $\leftarrow$ "*shift k*"

else if i *is* $[S' \rightarrow S \bullet, \underline{EOF}]$

then ACTION[x ,$\underline{EOF}$] $\leftarrow$ "*accept*"

else if i *is* $[A \rightarrow \beta \bullet, \underline{a}]$

then ACTION[x,$\underline{a}$] $\leftarrow$ "reduce $A \rightarrow \beta$"

$\forall$ $n \in NT$

if $goto(S_x, n) = S_k$

then GOTO[x,n] $\leftarrow$ k

• before $T \Rightarrow$ *shift*

Filling in the ACTION and GOTO Tables

The algorithm

$\forall$ *set* $S_x \in S$

$\quad\forall$ *item* $i \in S_x$

$\quad\quad$ *if* i *is* $[A \rightarrow \beta \bullet \underline{a}\delta, \underline{b}]$ *and* $goto(S_x, \underline{a}) = S_k$, $\underline{a} \in T$

$\quad\quad\quad$ *then* ACTION$[x, \underline{a}] \leftarrow$ "*shift k*"

$\quad\quad$ *else if* i *is* $[S' \rightarrow S \bullet, \underline{EOF}]$ ← have *Goal* ⇒ *accept*

$\quad\quad\quad$ *then* ACTION$[x, \underline{EOF}] \leftarrow$ "*accept*"

$\quad\quad$ *else if* i *is* $[A \rightarrow \beta \bullet, \underline{a}]$

$\quad\quad\quad$ *then* ACTION$[x, \underline{a}] \leftarrow$ "reduce $A \rightarrow \beta$"

$\quad\forall$ $n \in NT$

$\quad\quad$ *if* $goto(S_x, n) = S_k$

$\quad\quad\quad$ *then* GOTO$[x, n] \leftarrow k$

Filling in the ACTION and GOTO Tables

The algorithm

$\forall$ *set* $S_x \in S$

$\forall$ *item* $i \in S_x$

if i *is* $[A \rightarrow \beta \bullet \underline{a}\delta, \underline{b}]$ *and* $goto(S_x, \underline{a}) = S_k$, $\underline{a} \in T$

then ACTION[x,$\underline{a}$] $\leftarrow$ "*shift k*"

else if i *is* $[S' \rightarrow S \bullet, \underline{EOF}]$

then ACTION[x ,$\underline{EOF}$] $\leftarrow$ "*accept*"

else if i *is* $[A \rightarrow \beta \bullet, \underline{a}]$

then ACTION[x,$\underline{a}$] $\leftarrow$ "reduce $A \rightarrow \beta$"

$\forall$ $n \in NT$

if $goto(S_x, n) = S_k$

then GOTO[x,n] $\leftarrow$ k

- at end $\Rightarrow$ *reduce*

Filling in the ACTION and GOTO Tables

The algorithm

$\forall$ *set* $S_x \in S$

$\quad\forall$ *item* $i \in S_x$

$\quad\quad$*if* i *is* $[A \rightarrow \beta \bullet \underline{a}\delta, \underline{b}]$ *and* $goto(S_x, \underline{a}) = S_k$, $\underline{a} \in T$

$\quad\quad\quad$*then* ACTION$[x, \underline{a}] \leftarrow$ "*shift k*"

$\quad\quad$*else if* i *is* $[S' \rightarrow S \bullet, \underline{EOF}]$

$\quad\quad\quad$*then* ACTION$[x, \underline{EOF}] \leftarrow$ "*accept*"

$\quad\quad$*else if* i *is* $[A \rightarrow \beta \bullet, \underline{a}]$

$\quad\quad\quad$*then* ACTION$[x, \underline{a}] \leftarrow$ "reduce $A \rightarrow \beta$"

$\quad\forall$ $n \in NT$

$\quad\quad$*if* $goto(S_x, n) = S_k$

$\quad\quad\quad$*then* GOTO$[x, n] \leftarrow k$

Fill GOTO table

Example from SheepNoise

S_0 : { [*Goal*→ • *SheepNoise*, EOF], [*SheepNoise*→ • *SheepNoise* baa, EOF],
[*SheepNoise*→ • baa, EOF], [*SheepNoise*→ • *SheepNoise* baa, baa],
[*SheepNoise*→ • baa, baa] }

• before *T* ⇒ *shift k*

S_1 = *Goto*(S_0 , *SheepNoise*) =
{ [*Goal*→ *SheepNoise* •, EOF], [*SheepNoise*→ *SheepNoise* • baa, EOF],
[*SheepNoise*→ *SheepNoise* • baa, baa] }

S_2 = *Goto*(S_0 , baa) = { [*SheepNoise*→ baa •, EOF],
[*SheepNoise*→ baa •, baa] }

S_3 = *Goto*(S_1 , bac

…

if i is [$A \rightarrow \beta \bullet \underline{a}\delta, \underline{b}$] *and goto*($S_x$,a) = S_k , a ∈ *T*
then ACTION[*x*,a] ← "*shift k*"

…

0	*Goal*	→	*SheepNoise*
1	*SheepNoise*	→	*SheepNoise* baa
2		|	baa

Example from SheepNoise

S_0 : { [*Goal*→ • *SheepNoise*, EOF], [*SheepNoise*→ • *SheepNoise* baa, EOF],
[*SheepNoise*→ • baa, EOF], [*SheepNoise*→ • *SheepNoise* baa, baa],
[*SheepNoise*→ • baa, baa] }

• before $T \Rightarrow$ *shift* *k*

S_1 = *Goto*(S_0 , *SheepNoise*) =
{ [*Goal*→ *SheepNoise* •, EOF], [*SheepNoise*→ *SheepNoise* • baa, EOF],
[*SheepNoise*→ *SheepNoise* • baa, baa] }

S_2 = *Goto*(S_0 , baa) = { [*SheepNoise*→ baa •, EOF],
[*SheepNoise*→ baa •, baa] }

so, ACTION[s_0,baa] is "*shift* S_2" (clause 1)
(items define same entry)

S_3 = *Goto*(S_1 , baa) = { [*SheepNoise*→ *SheepNoise* baa •, EOF],
[*SheepNoise*→ *SheepNoise* baa •, baa] }

0	*Goal*	→	*SheepNoise*
1	*SheepNoise*	→	*SheepNoise* baa
2		|	baa

Example from SheepNoise

S_0 : { [*Goal*→ • ***SheepNoise***, EOF], [***SheepNoise***→ • ***SheepNoise*** baa, EOF],
[***SheepNoise***→ • baa, EOF], [***SheepNoise***→ • ***SheepNoise*** baa, baa],
[***SheepNoise***→ • baa, baa] }

S_1 = *Goto*(S_0 , ***SheepNoise***) =
{ [*Goal*→ ***SheepNoise*** •, EOF], [***SheepNoise***→ ***SheepNoise*** • baa, EOF],
[***SheepNoise***→ ***SheepNoise*** • baa, baa] }

S_2 = *Goto*(S_0 , baa) = { [***SheepNoise***→ baa •, EOF],
[***SheepNoise***→ baa •, baa] }

S_3 = *Goto*(S_1 , baa) = { [***SheepNoise***→ ***SheepNoise*** baa •, EOF],
[***SheepNoise***→ ***SheepNoise*** baa •, baa] }

so, ACTION[S_1,baa] is "*shift* S_3" (clause 1)

0	*Goal*	→	
1	*SheepNoise*	→	
2		|	

...

if i is [$A \rightarrow \beta \bullet \underline{a}\delta, \underline{b}$] *and goto*($S_x$,a) = S_k , a ∈ *T*
then ACTION[*x*,a] ← "*shift k*"

...

Example from SheepNoise

S_0 : { [*Goal*→ • *SheepNoise*, EOF], [*SheepNoise*→ • *SheepNoise* baa, EOF],
[*SheepNoise*→ • baa, EOF], [*SheepNoise*→ • *SheepNoise* baa, baa],
[*SheepNoise*→ • baa, baa] }

S_1 = *Goto*(S_0 , *SheepNoise*) =
{ [*Goal*→ *SheepNoise* •, EOF], [*SheepNoise*→ *SheepNoise* • baa, EOF],
[*SheepNoise*→ *SheepNoise* • baa, baa] }

so, ACTION[S_1,EOF] is "*accept*" (clause 2)

S_2 = *Goto*(S_0 , baa) = { [*SheepNoise*→ baa •, EOF],
[*SheepNoise*→ baa •, baa] }

S_3 = *Goto*(S_1 , ba

…

else if *i is* [*S'*→*S* •,EOF]
then ACTION[*x* ,EOF] ← "*accept*"

…

0	*Goal*	→	*SheepNoise*
1	*SheepNoise*	→	*SheepNoise* baa
2		|	baa

Example from SheepNoise

S_0 : { [*Goal*→ • *SheepNoise*, EOF], [*SheepNoise*→ • *SheepNoise* baa, EOF],
[*SheepNoise*→ • baa, EOF], [*SheepNoise*→ • *SheepNoise* baa, baa],
[*SheepNoise*→ • baa, baa] }

S_1 = *Goto*(S_0 , *SheepNoise*) =
{ [*Goal*→ *SheepNoise* •, EOF], [*SheepNoise*→ *SheepNoise* • baa, EOF],
[*SheepNoise*→ *SheepNoise* • baa, baa] }

S_2 = *Goto*(S_0 , baa) = { [*SheepNoise*→ baa •, EOF],
[*SheepNoise*→ baa •, baa] }

so, ACTION[S_2,EOF] is "*reduce 2*" (clause 3)

ACTION[S_2,baa] is "*reduce 2*" (clause 3)

S_3 = *Goto*(S_1 , baa) = { [*SheepNoise*→ *SheepNoise* baa •, EOF],
[*SheepNoise*→ *SheepNoise* baa •, baa] }

0	*Goal*	→
1	*SheepNoise*	→
2		|

…

else if i is [$A \rightarrow \beta \bullet$,a]
then ACTION[*x*,a] ← "reduce $A \rightarrow \beta$"

…

Example from SheepNoise

S_0 : { [*Goal*→ • *SheepNoise*, <u>EOF</u>], [*SheepNoise*→ • *SheepNoise* <u>baa</u>, <u>EOF</u>],
[*SheepNoise*→ • <u>baa</u>, <u>EOF</u>], [*SheepNoise*→ • *SheepNoise* <u>baa</u>, <u>baa</u>],
[*SheepNoise*→ • <u>baa</u>, <u>baa</u>] }

ACTION[S_3,<u>EOF</u>] is "*reduce 1*" (clause 3)

...e) =
•, <u>EOF</u>], [*SheepNoise*→ *SheepNoise* • <u>baa</u>, <u>EOF</u>],
[*SheepNoise*→ *SheepNoise* • <u>baa</u>, <u>baa</u>] }

S_2 = *Goto*(S_0 , <u>baa</u>) = { [*SheepNoise*→ <u>baa</u> •, <u>EOF</u>],
[*SheepNoise*→ <u>baa</u> •, <u>baa</u>] }

S_3 = *Goto*(S_1 , <u>baa</u>) = { [*SheepNoise*→ *SheepNoise* <u>baa</u> •, <u>EOF</u>],
[*SheepNoise*→ *SheepNoise* <u>baa</u> •, <u>baa</u>] }

ACTION[S_3,<u>baa</u>] is "*reduce 1*", as well

0	*Goal*	→
1	*SheepNoise*	→
2		|

...

else if i is [$A \rightarrow \beta \bullet$,<u>a</u>]
then ACTION[*x*,<u>a</u>] ← "reduce $A \rightarrow \beta$"

...

Example from SheepNoise

The GOTO Table records Goto transitions on NTs

s_0 : { [*Goal*→ • *SheepNoise*, EOF], [*SheepNoise*→ • *SheepNoise* baa, EOF],
[*SheepNoise*→ • baa, EOF], [*SheepNoise*→ • *SheepNoise* baa, baa],
[*SheepNoise*→ • baa, baa] }

s_1 = *Goto*(S_0 , *SheepNoise*) =
{ [*Goal*→ *SheepNoise* •, EOF], [*SheepNoise*→ *SheepNoise* • baa, EOF],
[*SheepNoise*→ *SheepNoise* • baa, baa] }

Puts s_1 in GOTO[s_0,*SheepNoise*]

s_2 = *Goto*(S_0 , baa) = { [*SheepNoise*→ baa •, EOF],
[*SheepNoise*→ baa •, baa] }

Based on *T*, not *NT* and written into the ACTION table

s_3 = *Goto*(S_1 , baa) = { [*SheepNoise*→ *SheepNoise* baa •, EOF],
[*SheepNoise*→ *SheepNoise* baa •, baa] }

0	*Goal*	→	*SheepNoise*
1	*SheepNoise*	→	*SheepNoise* baa
2		|	baa

Only 1 transition in the entire GOTO table

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

ACTION & GOTO Tables

Here are the tables for the *SheepNoise* grammar

The tables

ACTION TABLE		
State	EOF	baa
0	—	*shift 2*
1	*accept*	*shift 3*
2	*reduce 2*	*reduce 2*
3	*reduce 1*	*reduce 1*

GOTO TABLE	
State	*SheepNoise*
0	1
1	0
2	0
3	0

The grammar

0	*Goal*	$\rightarrow$	*SheepNoise*
1	*SheepNoise*	$\rightarrow$	*SheepNoise* baa
2		|	baa

What can go wrong? Shift/reduce error

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

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

What can go wrong? Reduce/reduce conflict

What is set *s* contains [$A \rightarrow \gamma \bullet$, a] and [$B \rightarrow \gamma \bullet$, a] ?

- Each generates “reduce”, but with a different production
- Both set ACTION[s,a] — cannot do both reductions
- This ambiguity is called *reduce/reduce conflict*
- Modify the grammar to eliminate it
  (PL/I’s overloading of (...))

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

Summary

- LR(1) items
- Creating ACTION and GOTO table
- What can go wrong?