RICE

COMP 412
FALL 2010

LR(1) Parsers

Comp 412

LR Parsers — A Roadmap

- Last Lecture
  - Handles, handles, and more handles
- Today
  - Revisit handles (briefly)
  - Skeleton LR(1) parser & its operation
  - Introduce notation & concepts for LR(1) Table Construction
- Monday
  - LR(1) Table construction

Handles

One more time, with less tedium

Consider our example

Sentential Form				
Goal				
Expr				
Expr	–	*Term*		
Expr	–	*Term*	*	*Factor*
Expr	–	*Term*	*	<id,y>
Expr	–	*Factor*	*	<id,y>
Expr	–	<num,2>	*	<id,y>
Term	–	<num,2>	*	<id,y>
Factor	–	<num,2>	*	<id,y>
<id,x>	–	<num,2>	*	<id,y>

Unambiguous grammar implies unique rightmost derivation

- At each step, we have one step that leads to x – 2 * y
- Any other choice leads to another distinct expression

A bottom-up parse reverses the rightmost derivation

- It has a unique reduction at each step
- The key is finding that reduction

Handles

Consider our example

Sentential Form					*Reduction*
Goal					
Expr					*Goal* → *Expr*
Expr	–	*Term*			*Expr* → *Expr* – *Term*
Expr	–	*Term*	*	*Factor*	*Term* → *Term* * *Factor*
Expr	–	*Term*	*	<id,y>	*Factor* → id
Expr	–	*Factor*	*	<id,y>	*Term* → *Factor*
Expr	–	<num,2>	*	<id,y>	*Factor* → num
Term	–	<num,2>	*	<id,y>	*Expr* → *Term*
Factor	–	<num,2>	*	<id,y>	*Term* → *Factor*
<id,x>	–	<num,2>	*	<id,y>	*Factor* → id

Handles

Consider our example

Sentential Form				
Goal				
Expr				
Expr	–	*Term*		
Expr	–	*Term*	*	*Factor*
Expr	–	*Term*	*	<id,y>
Expr	–	*Factor*	*	<id,y>
Expr	–	<num,2>	*	<id,y>
Term	–	<num,2>	*	<id,y>
Factor	–	<num,2>	*	<id,y>
<id,x>	–	<num,2>	*	<id,y>

Now, look at the sentential forms in the example

- They have a specific form
- $NT^* \; (NT \mid T)^* \; T^*$
- Reductions happen in the $(NT \mid T)^*$ portion
  - Track right end of region
  - Search left from there
  - Finite set of rhs strings

We know that each step has a unique reduction

That reduction is the handle

From Handles to Parsers

Consider the sentential forms in the derivation

$$NT^* \quad (NT \mid T)^* \quad T^*$$

- They are the upper fringe of partially complete syntax tree
- The suffix consisting of T^* is, at each step, the unread input

So, our shift-reduce parser operates by:

- Keeping the portion NT^* $(NT \mid T)^*$ on a stack
  - Leftmost symbol at bottom of stack, rightmost at stack top
- Searching for handles from stack top to stack bottom
- If search fails, shift another terminal onto stack

Bottom-up Parser

A conceptual *shift-reduce parser:*

```
push INVALID
word ← NextWord( )
repeat until (top of stack = Goal and word = 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 (word ≠ EOF)
            then // shift
                push word
                word ← NextWord( )
        else      // need to shift, but out of input
            report an error
```

What happens on an error?

- It fails to find a handle
- Thus, it keeps shifting
- Eventually, it consumes all input

This parser reads all input before reporting an error, not a desirable property.

Table-driven LR parsers do much better. The handle finder reports errors as soon as possible.

Actual LR(1) Skeleton Parser

```
stack.push(INVALID);
stack.push(s0);                         // initial state
word = scanner.next_word();
loop forever {
      s = stack.top();
      if ( ACTION[s,word] == "reduce A→β" ) then {
         stack.popnum(2*|β|);        // pop 2*|β| symbols
         s = stack.top();
         stack.push(A);              // push A
         stack.push(GOTO[s,A]);  // push next state
      }
      else if ( ACTION[s,word] == "shift si" ) then {
            stack.push(word); stack.push(si);
            word ← scanner.next_word();
      }
      else if ( ACTION[s,word] == "accept"
                        & word == EOF )
            then break;
      else throw a syntax error;
}
report success;
```

The skeleton parser

- follows basic scheme for shift-reduce parsing from last slide
- relies on a stack & a scanner
- keeps both symbols & states on the stack
- uses two tables, called ACTION & GOTO
- shifts |*words*| times
- reduces |derivation| times
- accepts at most once
- detects errors by failure of the other three cases, which is a failure to find a handle

The Parentheses Language

Language of balanced parentheses

- Beyond power of REs
- Exhibits role of context in LR(1) parsing

0	Goal	$\rightarrow$	List
1	List	$\rightarrow$	List Pair
2		|	Pair
3	Pair	$\rightarrow$	(Pair)
4		|	( )

The Parentheses Language

ACTION TABLE			
State	eof	(	)
0		S 3	
1	*acc*	S 3	
2	R 2	R 2	
3		S 6	S 7
4	R 1	R 1	
5			S 8
6		S 6	S 10
7	R 4	R 4	
8	R 3	R 3	
9			S 11
10			R 4
11			R 3

GOTO TABLE		
State	List	Pair
0	1	2
1		4
2		
3		5
4		
5		
6		9
7		
8		
9		
10		
11		

0	Goal	→	List
1	List	→	List Pair
2		|	Pair
3	Pair	→	(Pair)
4		|	( )

The Parentheses Language

State	Lookahead	Stack	Handle	Action
—	(	$ 0	—none—	—
0	(	$ 0	—none—	*shift 3*
3	)	$ 0 (3	—none—	*shift 7*
7	eof	$ 0 (3) 7	( )	*reduce 4*
2	eof	$ 0 *Pair* 2	*Pair*	*reduce 2*
1	eof	$ 0 *List* 1	*List*	*accept*

Parsing "( )"

0	Goal	→	List
1	List	→	List Pair
2		|	Pair
3	Pair	→	(Pair)
4		|	( )

0	Goal	→	List
1	List	→	List Pair
2		|	Pair
3	Pair	→	(Pair)
4		|	( )

The Parentheses Language

State	L'ahead	Stack	Handle	Action
—	(	$ 0	—none—	—
0	(	$ 0	—none—	*shift 3*
3	(	$ 0 (3	—none—	*shift 6*
6	)	$ 0 (3 (6	—none—	shift 10
10	)	$ 0 (3 (6) 10	( )	*reduce 4*
5	)	$ 0 (3 *Pair* 5	—none—	shift 8
8	(	$ 0 (3 *Pair* 5) 8	(*Pair*)	*reduce 3*
2	(	$ 0 *Pair* 2	*Pair*	*reduce 2*
1	(	$ 0 *List* 1	—none—	*shift 3*
3	)	$ 0 *List* 1 (3	—none—	*shift 7*
7	eof	$ 0 *List* 1 (3) 7	( )	*reduce 4*
4	eof	$ 0 *List* 1 *Pair* 4	*List Pair*	*reduce 1*
1	eof	$ 0 *List* 1	*List*	*accept*

Parsing
"(( )) ( )"

Let's look at how the parser reduces "( )"

The Parentheses Language

State	Lookahead	Stack	Handle	Action
—	(	$ 0	—none—	—
0	(	$ 0	—none—	*shift 3*
3	)	$ 0 (3	—none—	*shift 7*
7	eof	$ $\boxed{0}$ (3) 7	( )	$\boxed{\textit{reduce4}}$
2	eof	$ 0 *Pair* 2	*Pair*	*reduce 2*
1	eof	$ 0 *List* 1	*List*	*accept*

Parsing "( )"

Here, peeling ( ) off the stack reveals s_0.

Goto(s_0,*Pair*) is s_2.

0	Goal	→	List
1	List	→	List Pair
2		|	Pair
3	Pair	→	(Pair)
4		|	( )

0	Goal	→	List	
1	List	→	List Pair	
2		|	Pair	
3	Pair	→	(Pair)	
4		|	( )	

The Parentheses Language

State	L'ahead	Stack	Handle	Action
—	(	$ 0	—none—	—
0	(	$ 0	—none—	*shift 3*
3	(	$ 0 (3	—none—	*shift 6*
6	)	$ 0 (3 (6	—none—	shift 10
10	)	$ 0 (3 (6) 10	( )	*reduce 4*
5	)	$ 0 (3 *Pair* 5	—none—	shift 8
8	(	$ 0 (3 *Pair* 5) 8	(*Pair*)	*reduce 3*
2	(			
1	(			
3	)			
7	eof			
4	eof			
1	eof	$ 0 *List* 1	*List*	*accept*

Parsing
"(( )) ( )"

Here, peeling ( ) off the stack reveals s_3, which represents the left context of an unmatched (.

Goto(s_3,*Pair*) is s_5, a state in which we expect a). That path leads to a reduction by production 3, *Pair* → (*Pair*) .

0	Goal	→	List	
1	List	→	List Pair	
2		|	Pair	
3	Pair	→	(Pair)	
4		|	( )	

The Parentheses Language

State	L'ahead	Stack	Handle	Action
—	(	$ 0	*—none—*	—
0	(	$ 0	*—none—*	*shift 3*
3				*ift 6*
6				*ift 10*
10				*luce 4*
5				*ift 8*
8				*luce 3*
2	(	$ 0 *Pair* 2	*Pair*	*reduce 2*
1	(	$ 0 *List* 1	*—none—*	*shift 3*
3	)	$ 0 *List* 1 (3	*—none—*	*shift 7*
7	eof	$ 0 *List* $\boxed{1}$ (3) 7	( )	$\boxed{\textit{reduce 1}}$
4	eof	$ 0 *List* 1 *Pair* 4	*List Pair*	*reduce 0*
1	eof	$ 0 *List* 1	*List*	*accept*

Here, peeling ( ) off the stack reveals s_1, which represents the left context of a previously recognized List.

Goto(s_1,*Pair*) is s_4, a state in which we can reduce *List Pair* to *List* (on lookahead of either (or eof).

Parsing
"(( )) ( )"

Three copies of "reduce 4" with different context — produce three distinct behaviors

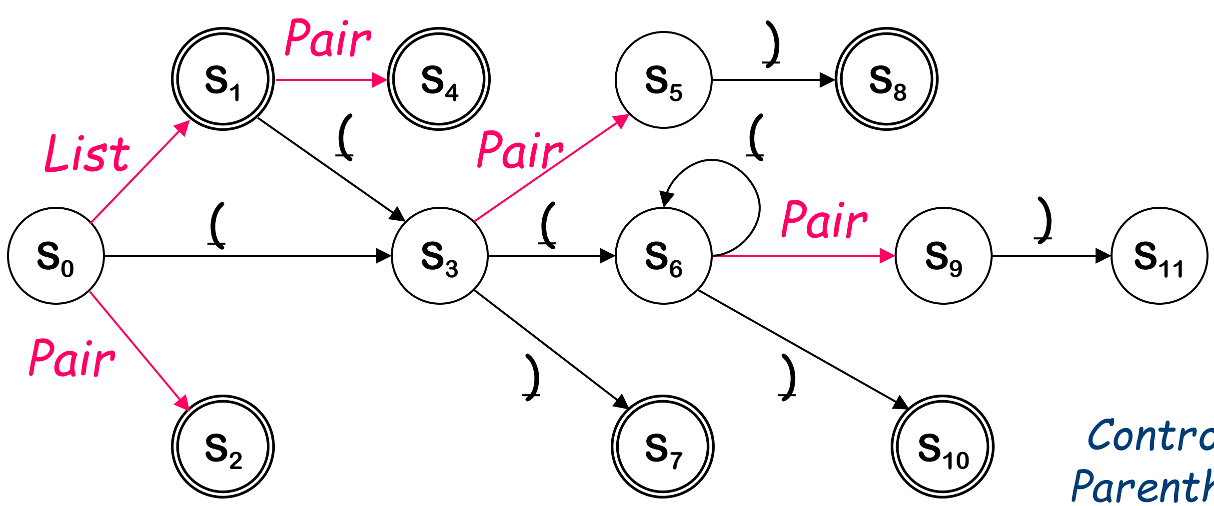
LR(1) Parsers

How does this LR(1) stuff work?

- Unambiguous grammar ⇒ unique rightmost derivation
- Keep upper fringe on a stack
  - All active handles include top of stack (TOS)
  - Shift inputs until TOS is right end of a handle
- Language of handles is regular (finite)
  - Build a handle-recognizing DFA to control the stack-based recognizer
  - ACTION & GOTO tables encode the DFA
- To match a subterm, invoke the DFA recursively
  - leave old DFA's state on stack and go on
- Final state in DFA ⇒ a *reduce* action
  - Pop rhs off the stack to reveal invoking state
    - → "It would be legal to recognize an *x*, and we did …"
  - New state is GOTO[revealed state, *lhs*]
  - Take a DFA transition on the new *NT* — the lhs we just pushed…

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

Building LR(1) Tables

How do we generate the *ACTION* and *GOTO* tables?

- Use the grammar to build a model of the Control DFA
- Encode actions & transitions in *ACTION* & *GOTO* tables
- If construction succeeds, the grammar is LR(1)
  - "Succeeds" means defines each table entry uniquely

The Big Picture

grammar symbol, *T* or *NT*

- Model the state of the parser
- Use two functions *goto(s, X)* and *closure(s)*
  - *goto()* is analogous to *move()* in the subset construction
  - *closure()* adds information to round out a state
- Build up the states and transition functions of the DFA
- Use this information to fill in the *ACTION* and *GOTO* tables

fixed-point algorithm, like subset construction

LR(k) Items

The LR(1) table construction algorithm represents a valid configuration of an LR(1) parser with a data structure called an LR(1) items

An LR(k) item is a pair [P, δ], where

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

δ is a lookahead string of length ≤ k (*words or EOF*)

The • in an item indicates the position of the top of the stack

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

[$A \rightarrow \beta\bullet\gamma$,$\underline{a}$] means that the input sees 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 β (that is, β is on top of the stack).

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

LR(1) Items

The production $A \rightarrow \beta$, where $\beta = B_1B_1B_1$ with lookahead $\underline{a}$, can give rise to 4 items

$[A \rightarrow \cdot B_1B_2B_3, \underline{a}]$, $[A \rightarrow B_1 \cdot B_2B_3, \underline{a}]$, $[A \rightarrow B_1B_2 \cdot B_3, \underline{a}]$, & $[A \rightarrow B_1B_2B_3 \cdot, \underline{a}]$

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 • 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 $\{ [A \rightarrow \beta \cdot, \underline{a}], [B \rightarrow \gamma \cdot \delta, \underline{b}] \}$, $\underline{a} \Rightarrow$ *reduce* to A; FIRST(δ) $\Rightarrow$ *shift*

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

LR(1) Table Construction

High-level overview

1 Build the canonical collection of sets of LR(1) Items, *I*

- a Begin in an appropriate state, s_0
  - [$S' \rightarrow \cdot S$,EOF], along with any equivalent items
  - Derive equivalent items as *closure*(s_0)
- b Repeatedly compute, for each s_k, and each X, *goto*(s_k,X)
  - If the set is not already in the collection, add it
  - Record all the transitions created by *goto*( )

  This eventually reaches a fixed point

2 Fill in the table from the collection of sets of LR(1) items

The states of the 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 items already in *s*

- Any item $[A \rightarrow \beta \bullet B\delta, \underline{a}]$ implies $[B \rightarrow \bullet \tau, x]$ for each production with B on the *lhs*, and each $x \in$ FIRST($\delta\underline{a}$)
- Since $\beta B\delta$ is valid, any way to derive $\beta B\delta$ is valid, too

The algorithm

```
Closure( s )
  while ( s is still changing )
    ∀ items [A → β•Bδ,a] ∈ s
      ∀ productions B → τ ∈ P
        ∀ b ∈ FIRST(δa) // δ might be ε
          if [B→ • τ,b] ∉ s
            then s ← s ∪ { [B→ • τ,b] }
```

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

Lookaheads are generated here

Example From SheepNoise

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

Closure([*Goal*→•*SheepNoise*,EOF])

Item	*Source*
[*Goal* → • *SheepNoise*,EOF]	Original item
[*SheepNoise* → • *SheepNoise* baa, EOF]	1, δa is EOF
[*SheepNoise* → • baa, EOF]	1, δa is EOF
[*SheepNoise* → • *SheepNoise* baa, baa]	2, δa is baa EOF
[*SheepNoise* → • baa, baa]	2, δa is baa EOF

So, S_0 is

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

Remember, this is the left-recursive SheepNoise; EaC shows the right-recursive version.

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

Computing Gotos

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

- *Goto*({ [$A \rightarrow \beta \bullet X\delta,\underline{a}$] }, *X*) produces [$A \rightarrow \beta X \bullet \delta,\underline{a}$] *(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* ( )
- *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] }

From Slide 16

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]

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

Building the Canonical Collection

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

Repeatedly construct new states, until all are found

The algorithm

$s_0 \leftarrow closure(\ [S' \rightarrow S, \underline{EOF}]\ )$
$S \leftarrow \{\ s_0\ \}$
$k \leftarrow 1$
$while\ (\ S\ is\ still\ changing\ )$
$\quad \forall s_j \in S\ and\ \forall x \in (\ T \cup NT\ )$
$\qquad s_k \leftarrow goto(s_j, x)$
$\qquad record\ s_j \rightarrow s_k\ on\ x$
$\quad if\ \ s_k \notin S\ then$
$\qquad S \leftarrow S \cup \{\ s_k\ \}$
$\qquad k \leftarrow k + 1$

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

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] }

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

x is the state number

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

- before $T \Rightarrow$ *shift*

have *Goal* $\Rightarrow$ *accept*

- at end $\Rightarrow$ *reduce*

Many items generate no table entry

→ *Closure( )* instantiates FIRST(X) directly for $[A \to \beta \bullet X\delta, \underline{a}]$

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] }

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] }

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>] }

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

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

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

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 , baa) = { [*SheepNoise*→ *SheepNoise* baa •, EOF],
[*SheepNoise*→ *SheepNoise* baa •, 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 3" (clause 3)

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

Example from SheepNoise

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

ACTION[S_3,EOF] is
"reduce 2" (clause 3)

e) =
•, EOF], [*SheepNoise*→*SheepNoise* • baa, EOF],
[*SheepNoise*→*SheepNoise* • baa, baa] }

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

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] }

ACTION[S_3,baa] is
"reduce 2", as well

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

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] }

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

Based on T, not NT

S_3 = Goto(S_1 , baa) = { [***SheepNoise***→ ***SheepNoise*** baa •, EOF],
[***SheepNoise***→ ***SheepNoise*** baa •, 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 augmented left-recursive *SheepNoise* grammar

Remember, this is the left-recursive SheepNoise; EaC shows the right-recursive version.

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?

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 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 is set s contains $[A \to \gamma \bullet, \underline{a}]$ and $[B \to \gamma \bullet, \underline{a}]$?

- 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 *(PL/I's overloading of (...))*

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

Shrinking the Tables

Three options:

- Combine terminals such as number & identifier, + & -, * & /
  - Directly removes a column, may remove a row
  - For expression grammar, 198 (vs. 384) table entries
- Combine rows or columns
  - Implement identical rows once & remap states
  - Requires extra indirection on each lookup
  - Use separate mapping for ACTION & for GOTO
- Use another construction algorithm
  - Both LALR(1) and SLR(1) produce smaller tables
  - Implementations are readily available

left-recursive expression grammar with precedence, see § 3.7.2 in EAC

classic space-time tradeoff

Is handle recognizing DFA minimal?

LR(k) versus LL(k)

Finding Reductions

LR(k) ⇒ Each reduction in the parse is detectable with

→ the complete left context,

→ the reducible phrase, itself, and

→ the k terminal symbols to its right

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

LL(k) ⇒ Parser must select the reduction based on

→ The complete left context

→ The next k terminals

Thus, LR(k) examines more context

"… in practice, programming languages do not actually seem to fall in the gap between LL(1) languages and deterministic languages"
J.J. Horning, "LR Grammars and Analysers", in Compiler Construction, An Advanced Course, Springer-Verlag, 1976

Summary

	Advantages	*Disadvantages*
Top-down recursive descent	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