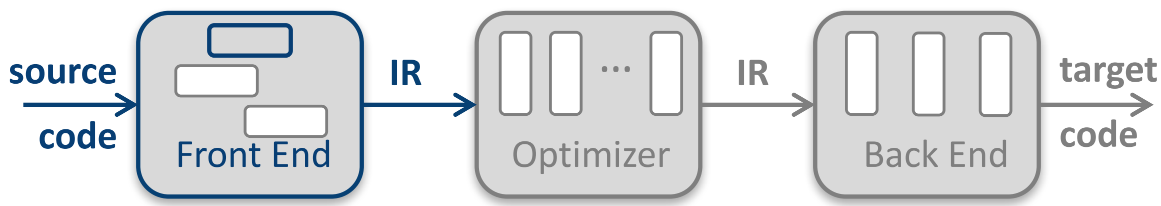
COMP 412
FALL 2017

Syntax Analysis, VI

Examples from LR Parsing

Comp 412



Chapter 3 in EaC2e

Roadmap

Last Class

- Bottom-up parsers, reverse rightmost derivations
- The mystical concept of a handle
  - Easy to understand if we are given an oracle
  - Opaque (at this point) unless we are given an oracle
- Saw a bottom-up, shift-reduce parser at work on x – 2 * y

This Class

- Structure & operation of an **LR(1)** parser
  - Both a skeleton parser & the **LR(1)** tables
- Example from the Parentheses Language
  - Look at how the **LR(1)** parser uses lookahead to determine *shift* vs *reduce*
- Lay the groundwork for the table construction lecture
  - **LR(1)** items, *Closure*(), and *Goto*()

LR(1) Parsers

This week will focus on LR(1) parsers

- **LR(1)** parsers are table-driven, shift-reduce parsers that use a limited right context (1 word) for handle recognition
- The class of grammars that these parsers recognize is called the set of **LR(1)** grammars

Informal definition:

A grammar is **LR(1)** if, given a rightmost derivation

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \textit{sentence}$$

We can

1. ***isolate the handle of each right-sentential form γ_i***, and
2. ***determine the production by which to reduce***,

by scanning γ_i from *left-to-right*, going at most *1* word beyond the right end of the handle of γ_i

I always find this definition to be unsatisfying because it isn't an operational definition.

LR(1) *implies a* ***left-to-right scan*** *of the input, a* ***rightmost derivation*** *(in reverse), and* ***1*** *word of lookahead.*

Bottom-up Parser

Our conceptual *shift-reduce parser* from last lecture

```
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
report success
```

Shift-reduce parsers have four kinds of actions:

Shift: next word is moved from input to stack

***Reduce**:* handle is at **TOS**

pop **RHS** of handle

push **LHS** of handle

Accept: stop & report success

Error: report an error

Shift & *Accept* are **O**(1)

Reduce is **O**(|**RHS**|) (*typically small*)

Key insight: the parser shifts until a handle appears at **TOS**

The **LR(1)** Skeleton Parser

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

The Skeleton LR(1) parser

- follows basic shift-reduce scheme from last slide
- relies on a stack & a scanner
- Stacks <symbol, state> pairs
- handle finder is encoded in two tables: ACTION & GOTO
- shifts |*words*| times
- reduces |*derivation*| times
- accepts at most once
- detects errors by *failure of the handle-finder*, not by exhausting the input

Given tables, we have a parser.

The Parentheses Language

Language of Balanced Parentheses

- Any sentence that consists of an equal number of (‘s and)’s
- Beyond the power of regular expressions
  - Classic justification for context-free grammar

1	*Goal*	$\rightarrow$	*List*
2	*List*	$\rightarrow$	*List Pair*
3		|	*Pair*
4	*Pair*	$\rightarrow$	(*List*)
5		|	( )

Good example to elucidate the role of context in **LR(1)** parsing

This grammar and its tables differ, slightly, from the one in EaC2e.

On Handout

LR(1) Tables for Parenthesis Grammar

1	Goal	→	List
2	List	→	List Pair
3		|	Pair
4	Pair	→	(List)
5		|	( )

ACTION			
State	(	)	EOF
s_0	s 3		
s_1	s 3		acc
s_2	r 3		r 3
s_3	s 7	s 8	
s_4	r 2		r 2
s_5	s 7	s 10	
s_6	r 3	r 3	
s_7	s 7	s 12	
s_8	r 5		r 5
s_9	r 2	r 2	
s_{10}	r 4		r 4
s_{11}	s 7	s 13	
s_{12}	r 5	r 5	
s_{13}	r 4	r 4	

GOTO		
State	List	Pair
s_0	1	2
s_1		4
s_2		
s_3	5	6
s_4		
s_5		9
s_6		
s_7	11	6
s_8		
s_9		
s_{10}		
s_{11}		9
s_{12}		
s_{13}		

"s 23" means shift & goto state 23

"r 18" means reduce by prod'n 18 (& find next state in the GOTO table)

Blank is an error entry

Parsing "()"

The Parentheses Language

State	Lookahead	Stack	Handle	Action
—	(	$ 0	*—none—*	—
0	(	$ 0	*—none—*	shift 3
3	)	$ 0 (3	*—none—*	shift 8
8	**EOF**	$ 0 (3) 8	*Pair* → ( )	reduce 5
2	**EOF**	$ 0 *Pair* 2	*List* → *Pair*	reduce 3
1	**EOF**	$ 0 *List* 1	*Goal* → *List*	accept

The **Lookahead** column shows the contents of *word* in the algorithm

Parsing "(())()"

The Parentheses Language

State	Lookahead	Stack	Handle	Action
—	(	\$ 0	—none—	—
0	(	\$ 0	—none—	shift 3
3	(	\$ 0 (3	—none—	shift 7
7	)	\$ 0 (3 (7	—none—	shift 12
12	)	\$ 0 (3 (7) 12	*Pair* → ( )	reduce 5
6	)	\$ 0 (3 *Pair* 6	*List* → *Pair*	reduce 3
5	)	\$ 0 (3 *List* 5	—none—	shift 10
10	(	\$ 0 (3 *List* 5) 10	*Pair* → (*List*)	reduce 4
2	(	\$ 0 *Pair* 2	*List* → *Pair*	reduce 3
1	(	\$ 0 *List* 1	—none—	shift 3
3	)	\$ 0 *List* 1 (3	—none—	shift 8
8	**EOF**	\$ 0 *List* 1 (3) 8	*Pair* → ( )	reduce 5
4	**EOF**	\$ 0 *List* 1 *Pair* 4	*List* → *List Pair*	reduce 2
1	**EOF**	\$ 0 *List* 1	*Goal* → *List*	accept

Parsing “(())()”

The Parentheses Language

State	Lookahead	Stack	Handle	Action
—	(	$ 0	—none—	—
0	(	$ 0	—none—	shift 3
3	(	$ 0 (3	—none—	shift 7
7	)	$ 0 (3 (7	—none—	shift 12
12	)	$ 0 (3 (7) 12	*Pair* → ( )	reduce 5
6	)	$ 0 (3 *Pair* 6	*List* → *Pair*	reduce 3
5	)	$ 0 (3 *List* 5	—none—	shift 10
10	(	$ 0 (3 *List* 5) 10	*Pair* → (*List*)	reduce 4
2	(	$ 0 *Pair* 2	*List* → *Pair*	reduce 3
1	(	$ 0 *List* 1	—none—	shift 3
3	)	$ 0 *List* 1 (3	—none—	shift 8
8	**EOF**	$ 0 *List* 1 (3) 8	*Pair* → ( )	reduce 5
4	**EOF**	$ 0 *List* 1 *Pair* 4	*List* → *List Pair*	reduce 2
1	**EOF**	$ 0 *List* 1	*Goal* → *List*	accept

Let’s look at how it reduces “()”

We have seen 3 examples

The Parentheses Language

State	Lookahead	Stack	Handle	Action
—	(	$ 0	—*none*—	—
0	(	$ 0	—*none*—	shift 3
3	)	$ 0 (3	—*none*—	shift 8
8	**EOF**	$ 0 **(3) 8**	*Pair* → ( )	reduce 5
2	**EOF**	$ 0 *Pair* 2	*List* → *Pair*	reduce 3
1	**EOF**	$ 0 *List* 1	*Goal* → *List*	accept

In the string “( )”, reducing by production 5 reveals state s_0.

Goto(s_0, *Pair*) is s_2, which leads to chain of productions 3 & 1.

1	Goal	→	List
2	List	→	List Pair
3		|	Pair
4	Pair	→	(List)
5		|	( )

The Parentheses Language

State	Lookahead	Stack	Handle	Action
—	(	$ 0	—*none*—	—
0	(	$ 0	—*none*—	shift 3
3	(	$ 0 (3	—*none*—	shift 7
7	)	$ 0 (3 (7	—*none*—	shift 12
12	)	$ 0 (3 (7) 12	*Pair* → ( )	reduce 5
6	)	$ 0 (3 *Pair* 6	*List* → *Pair*	reduce 3
5	)	$ 0 (3 *List* 5	—*none*—	shift 10
10	(	$ 0 (3 *List* 5) 10	*Pair* → (*List*)	reduce 4
				reduce 3
				shift 3
				r
				r
1	**EOF**	$ 0 *List* 1	*Goal* → *List*	

Here, reducing by 5 reveals state s_3, which represents the left context of an unmatched ‘(‘. There will be one s_3 per unmatched ‘(‘ — they count the remaining ‘(‘s.

Goto(s_3, *Pair*) is s_6, a state in which the parser expects a ‘)’. That state leads to reductions by 3 and then 4.

1	Goal	→	List
2	List	→	List Pair
3		|	Pair
4	Pair	→	(List)
5		|	( )

Parsing "(())()"

The Parentheses Language

1	*Goal*	→	*List*	
2	*List*	→	*List Pair*	
3		|	*Pair*	
4	*Pair*	→	(*List*)	
5		|	( )	

State	Lookahead	Stack	Handle	
—	(	$ 0	—*none*—	
0	(	$ 0	—*none*—	shift 3
				shift 7
				shift 12
				reduce 5
				reduce 3
				shift 10
10	(	$ 0 (3 *List* 5) 10	*Pair* → (*List*)	reduce 4
2	(	$ 0 *Pair* 2	*List* → *Pair*	reduce 3
1	(	$ 0 *List* 1	—*none*—	shift 3
3	)	$ 0 *List* 1 (3	—*none*—	shift 8
8	**EOF**	**$ 0** ***List* 1** **(3) 8**	*Pair* → ( )	**reduce 5**
4	**EOF**	$ 0 *List* 1 *Pair* 4	*List* → *List Pair*	reduce 2
1	**EOF**	$ 0 *List* 1	*Goal* → *List*	accept

Here, reducing by 5 reveals state s_1, which represents the left context of a previously recognized *List*.

Goto(s_1, *Pair*) is s_4, a state in which the parser will reduce *List Pair* to *List* (production 2) on a lookahead of either '(' or **EOF**.

Here, lookahead is **EOF**, which leads to reduction by 2, then by 1.

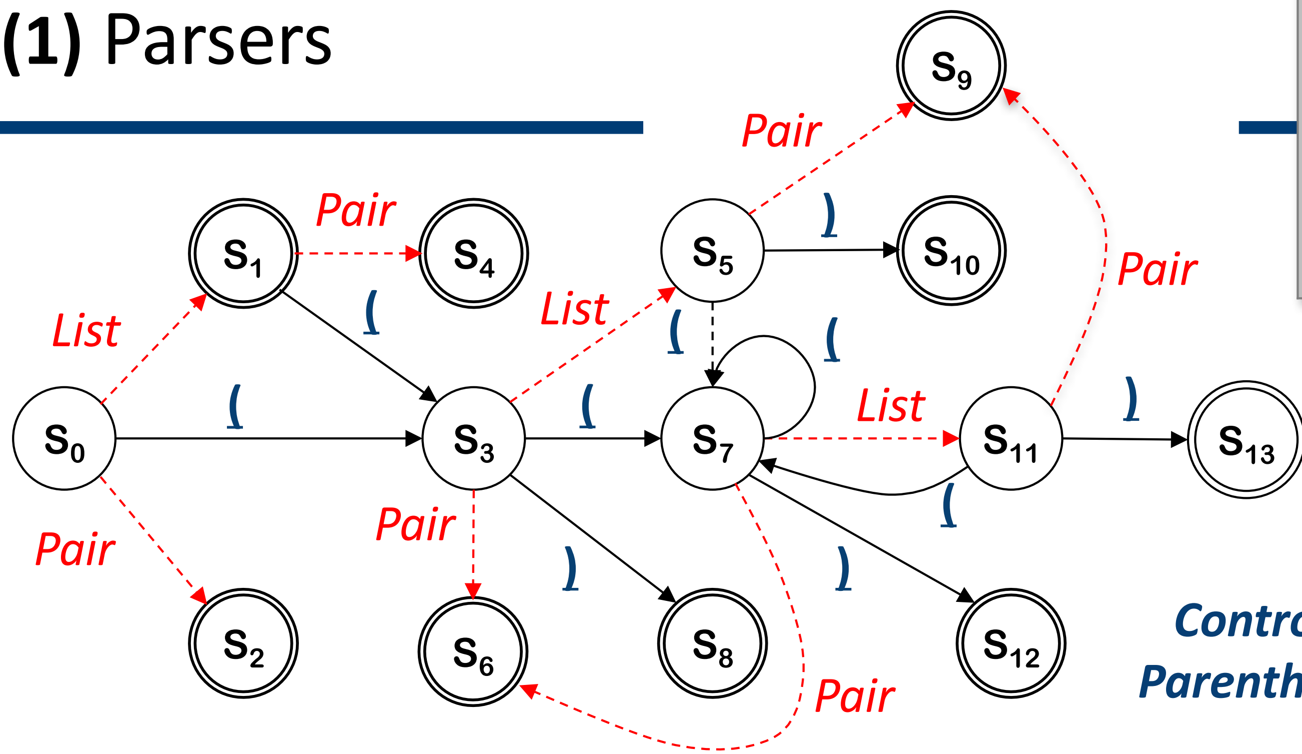
LR(1) Parsers

Recap: How does an LR(1) parser work?

- Unambiguous grammar $\Rightarrow$ 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** $\Rightarrow$ 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

1	Goal	→	List
2	List	→	List Pair
3		|	Pair
4	Pair	→	(List)
5		|	( )



Control DFA for the Parentheses Language

The Control **DFA** for the parentheses language is embedded in the ACTION and GOTO Tables

→ Transitions on **terminals** represent shift actions [ACTION Table]

→ Transitions on **nonterminals** follow reduce actions [GOTO Table]

The table construction derives this **DFA** from the grammar.

This point is not obvious. To see it, compare the **ACTION** & **GOTO** tables for the parenthesis language with the **DFA**.

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*

An operational definition

The Big Picture

grammar symbol, *T* or *NT*

- Model the state of the parser with **LR(1)** items
- Use two functions *goto(s, X)* and *closure(s)*
  - *goto()* is analogous to *move()* in the subset construction
  - Given a partial state, *closure()* adds all the items implied by the partial 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,
similar to the subset construction

LR(1) Table Construction

To understand the algorithms, we need to understand the data structure that they use: LR(1) items

- The **LR(1)** table construction algorithm models the set of possible states that the parser can enter
  - Mildly reminiscent of the subset construction (**NFA→DFA**)
- The construction needs a representation for the parser's state, as a function of the context it has seen and might see

LR(1) Items

- The **LR(1)** table construction algorithm represents each valid configuration of an **LR(1)** parser with an **LR(1)** item
- An **LR(1)** item is a pair $[P, \delta]$, where

  P is a production $A \rightarrow \beta$ with a • at some position in the **RHS**
  δ is a single symbol lookahead (*symbol* ≅ *word* or **EOF**)

LR(1) Items

The *intermediate representation* of the **LR(1)** table construction algorithm

An **LR(1)** 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 single symbol lookahead (*symbol* ≅ *word* or **EOF**)

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

[$A \rightarrow \bullet\beta\gamma$,<u>a</u>] 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. We call an item like this a *<u>possibility</u>*.

[$A \rightarrow \beta\bullet\gamma$,<u>a</u>] 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). We call an item like this a *<u>partially</u>* *<u>complete</u>* item.

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

Might **STOP**

LR(k) parsers rely on items with a lookahead of ≤ k symbols. That leads to **LR(k)** items, with correspondingly longer δ.

LR(1) Items

The production $A \rightarrow \beta$, where $\beta = B_1B_2B_3$ with lookahead $\underline{a}$, can give rise to 4 items

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

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 $[A \rightarrow \beta \bullet \gamma, \underline{a}]$
  - In $[A \rightarrow \beta \bullet, \underline{a}]$, a lookahead of $\underline{a}$ implies a reduction by $A \rightarrow \beta$
  - For $\{ [A \rightarrow \beta \bullet, \underline{a}], [B \rightarrow \gamma \bullet \delta, \underline{b}] \}$, $\underline{a} \Rightarrow$ *reduce* to A; **FIRST**$(\delta) \Rightarrow$ *shift*

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

$\underline{a} \in$ **FIRST**$(\delta) \Longrightarrow$ a conflict, not LR(1)

LR(1) Items: Why should you know this stuff?

Debugging a grammar

- When you build an **LR(1)** parser, it is possible (likely) that the initial grammar is not **LR(1)**
- The tools will provide you with debugging output
- To the right is a sample of **bison**'s output for the **if-then-else** grammar

goal	→	stmt_list
stmt_list	→	stmt_list stmt
	|	stmt
stmt	→	IF EXPR THEN stmt
	|	IF EXPR THEN stmt ELSE stmt
	|	OTHER

state 10

4 stmt : IF EXPR THEN stmt .
5 | IF EXPR THEN stmt . ELSE stmt

ELSE shift, and go to state 11

ELSE   [reduce using rule 4 (stmt)]
$default reduce using rule 4 (stmt)

That period is the •

The state is described by its **LR(1)** items

Likely **STOP**

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
   - [$S' \rightarrow \bullet S$,**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

S is the start symbol. To simplify things, we add $S' \rightarrow S$ to create a unique goal production.

goto(s_i, X) contains the set of LR(1) items that represent possible parser configurations if the parser recognizes an X while in state s_i

2 Fill in the table from the Canonical Collection of Sets of **LR(1)** items

The sets in the canonical collection form the states of the Control **DFA**.

The construction traces the **DFA**'s transitions

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
  - [$S' \rightarrow \bullet S$,<u>**EOF**</u>], 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 Canonical Collection of Sets of **LR(1)** items

Let's build the tables for the left-recursive *SheepNoise* grammar (S' is *Goal*)

0	*Goal*	$\rightarrow$	*SheepNoise*
1	*SheepNoise*	$\rightarrow$	*SheepNoise* <u>baa</u>
2		|	<u>baa</u>

Computing Closures

Closure(_s_) adds all the items implied by items already in _s_

- Any item $[A \rightarrow \beta \bullet B\delta, \underline{a}]$ where $B \in NT$ implies $[B \rightarrow \bullet \tau, x]$ for each production that has 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)
    $\forall$ *items* $[A \rightarrow \beta \bullet B\delta, \underline{a}] \in s$
      $\forall$ *productions* $B \rightarrow \tau \in P$
        $\forall\ \underline{b} \in$ FIRST($\delta\underline{a}$) // δ might be ε
          if $[B \rightarrow \bullet \tau, \underline{b}] \notin s$
            then $s \leftarrow s \cup \{ [B \rightarrow \bullet \tau, \underline{b}] \}$

- Classic fixed-point method
- Halts because $s \subset I$, the set of items
- Worklist version is faster
- Closure *"fills out"* a state *s*

Generate new lookaheads.
See note on p. 128

This is the left-recursive SheepNoise; EaC2e shows the right-recursive version.

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**]	**ITER** 1, **PR** 0, δa is **EOF**
[*SheepNoise* → • baa, **EOF**]	**ITER** 1, **PR** 0, δa is **EOF**
[*SheepNoise* → • *SheepNoise* baa, baa]	**ITER** 2, **PR** 1, δa is baa **EOF**
[*SheepNoise* → • baa, baa]	**ITER** 2, **PR** 1, δa is baa **EOF**

Symbol	*FIRST*
Goal	{ baa }
SheepNoise	{ baa }
baa	{ baa }
EOF	{ **EOF** }

So, 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

Computing Gotos

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

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

The Algorithm

> ***Goto***(s, X)
> $new \leftarrow \emptyset$
> $\forall$ *items* $[A \to \beta \bullet X \delta, \underline{a}] \in s$
> $new \leftarrow new \cup \{ [A \to \beta X \bullet \delta, \underline{a}] \}$
>   *return* ***closure***(*new*)

- Not a fixed-point method!
- Straightforward computation
- Uses *Closure* ( )
- *Goto*( ) models a transition in the automaton

Goto in this construction is analogous to *Move* in the subset construction.

Example from SheepNoise

Assume that S_0 is

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

From earlier slide

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***([$S' \rightarrow S$,<u>**EOF**</u>])

Repeatedly construct new states, until all are found

The Algorithm

$s_0 \leftarrow$ ***Closure*** ([$S' \rightarrow S$,<u>EOF</u>])
$S \leftarrow \{ s_0 \}$
$k \leftarrow 1$

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

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

This membership / equality test requires careful and/or clever implementation.

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] }

Hard STOP

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