Parsing V
LR(1) Parsers

N.B.: This lecture uses a left-recursive version of the SheepNoise grammar. The book uses a right-recursive version.

The derivations (& the tables) are different.

Scheduling Assignments

Mid-term exam

- available Friday 10/7/2005 (before break)
- due Monday 10/17/2005 (two weekends)

Lab 2 — the dreaded parser

- available Wednedsay 10/5/2005
- choose teams by 10/7/2005
- intermediate progress report due 10/20 to 21/2005
  - each team meet with one of the labbies
  - concrete milestones for a portion of the grade
- code due 11/1/2005
- report due 11/2/2005

LR(1) Parsers

- LR(1) parsers are table-driven, shift-reduce parsers that use a limited right context (1 token) for handle recognition
- LR(1) parsers recognize languages that have an LR(1) grammar

Informal definition:

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

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

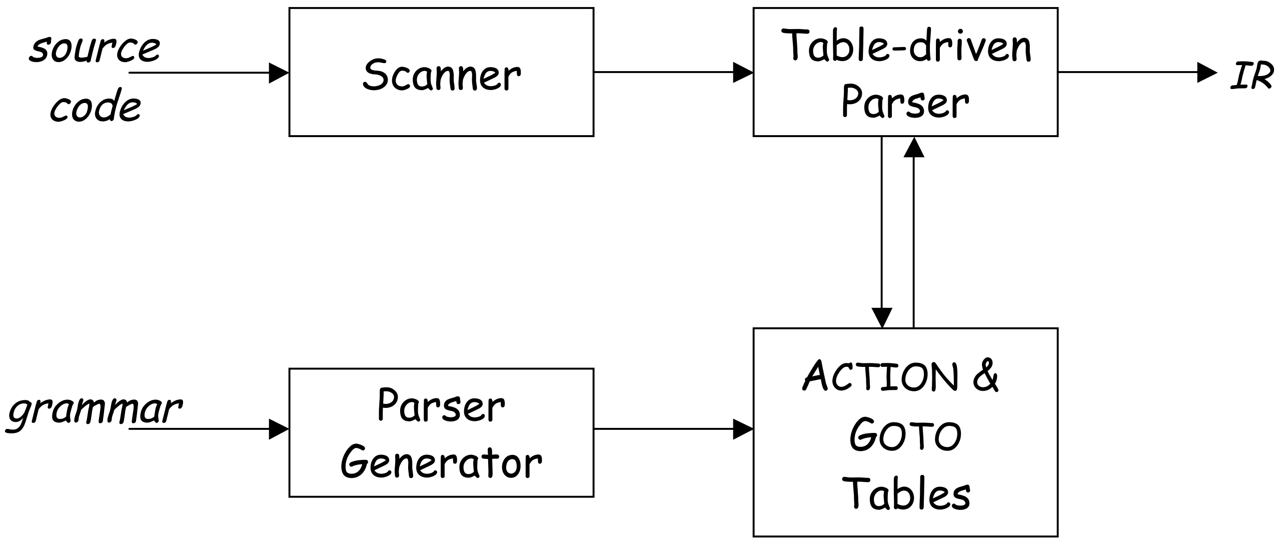
We can

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* symbol beyond the right end of the handle of γ_i

LR(1) Parsers

A table-driven LR(1) parser looks like

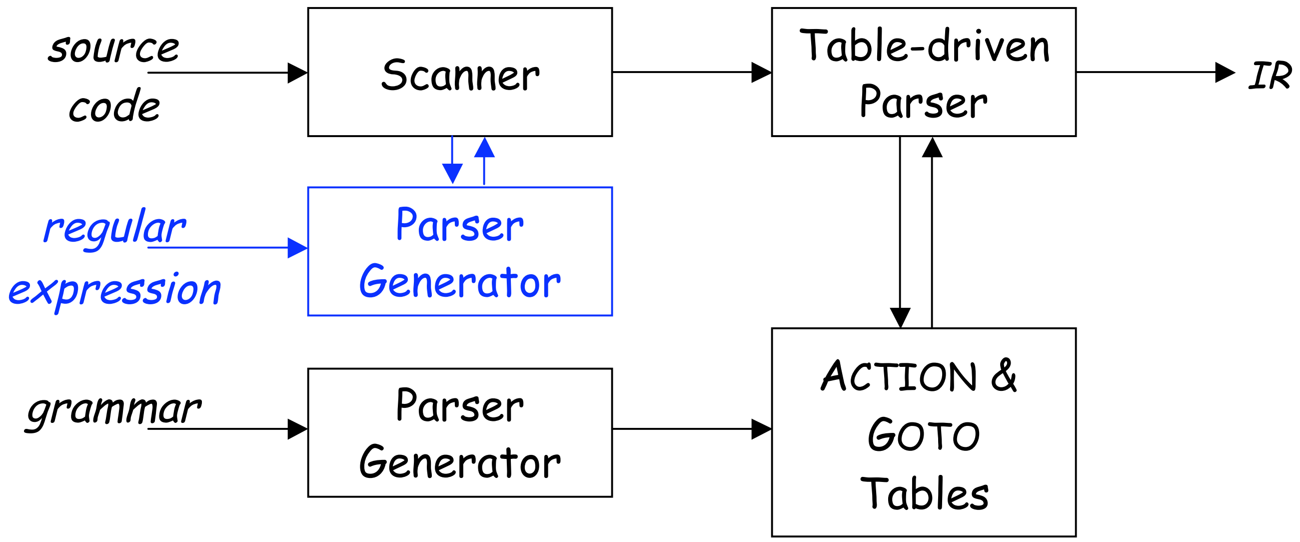


Tables *can* be built by hand

However, this is a perfect task to automate

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

Just like automating construction of scanners …

LR(1) Skeleton Parser

```
stack.push(INVALID);
 stack.push($s_0$);                    // initial state
token = scanner.next_token();
loop forever {
        s = stack.top();
        if ( ACTION[s,token] == "reduce $A \rightarrow \beta$" ) then {
            stack.popnum(2*|$\beta$|);      // pop 2*|$\beta$| symbols
            s = stack.top();
            stack.push($A$);                // push A
            stack.push(GOTO[s,$A$]);        // push next state
        }
        else if ( ACTION[s,token] == "shift $s_i$" ) then {
                stack.push(token); stack.push($s_i$);
                token ← scanner.next_token();
        }
        else if ( ACTION[s,token] == "accept"
                                & token == EOF )
                then break;
        else throw a syntax error;
}
report success;
```

The skeleton parser

- relies on a stack & a scanner
- 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
- follows basic scheme for shift-reduce parsing from last lecture

LR(1) Parsers (parse tables)

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

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

The grammar

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

The tables

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

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

Example Parse 1

The string “baa”

Stack	Input	Action
$ s_0	baa EOF	shift 2

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

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

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

Example Parse 1

The string “baa”

Stack	Input	Action
\$ s_0	baa EOF	shift 2
\$ s_0 baa s_2	EOF	reduce 3

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

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

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

Example Parse 1

The string “baa”

Stack	Input	Action
\$ s_0	baa EOF	shift 2
\$ s_0 baa s_2	EOF	reduce 3
\$ s_0 *SN* s_1	EOF	

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

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

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

Example Parse 1

The string “baa”

Stack	Input	Action
$\$\ s_0$	baa EOF	shift 2
$\$\ s_0$ baa s_2	EOF	reduce 3
$\$\ s_0$ *SN* s_1	EOF	accept

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

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

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

Example Parse 2

The string “baa baa ”

Stack	Input	Action
\$ s_0	baa baa EOF	shift 2
\$ s_0 baa s_2	baa EOF	

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

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

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

Example Parse 2

The string “baa baa ”

Stack	Input	Action
$\$\ s_0$	baa baa EOF	shift 2
$\$\ s_0$ baa s_2	baa EOF	reduce 3
$\$\ s_0$ *SN* s_1	baa EOF	

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

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

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

Example Parse 2

The string “baa baa ”

Stack	Input	Action
$\$\ s_0$	baa baa EOF	shift 2
$\$\ s_0$ baa s_2	baa EOF	reduce 3
$\$\ s_0$ *SN* s_1	baa EOF	shift 3
$\$\ s_0$ *SN* s_1 baa s_3	EOF	

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

ACTION

State	EOF	baa
0	—	shift 2
1	accept	shift 3
2	reduce 3	reduce 3
3	reduce 2	reduce 2

GOTO

State	*SheepNoise*
0	1
1	0
2	0
3	0

Example Parse 2

The string “baa baa ”

Stack	Input	Action
$\$\ s_0$	baa baa EOF	shift 2
$\$\ s_0$ baa s_2	baa EOF	reduce 3
$\$\ s_0$ *SN* s_1	baa EOF	shift 3
$\$\ s_0$ *SN* s_1 baa s_3	EOF	reduce 2
$\$\ s_0$ *SN* s_1	EOF	accept

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

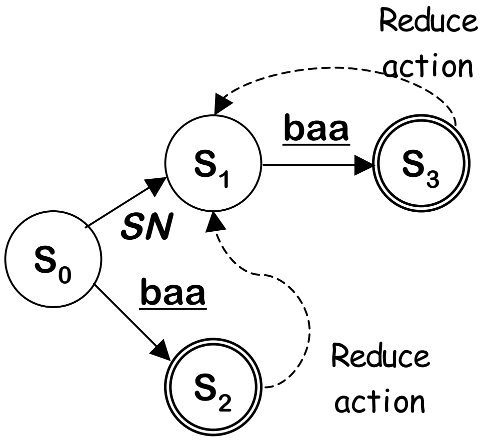
ACTION		
State	EOF	baa
0	—	shift 2
1	accept	shift 3
2	reduce 3	reduce 3
3	reduce 2	reduce 2

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

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
  - ACTION & GOTO tables encode the DFA
- To match subterm, invoke subterm DFA & leave old DFA's state on stack
- Final state in DFA ⇒ a *reduce* action
  - New state is GOTO[state at TOS (after pop), *lhs*]
  - For *SN*, this takes the DFA to s_1



Control DFA for SN

Building LR(1) Parsers

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

- Use the grammar to build a model of the DFA
- Use the model to build *ACTION* & *GOTO* tables
- If construction succeeds, the grammar is LR(1)

Terminal or non-terminal

The Big Picture

- 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

LR(k) Items

The LR(1) table construction algorithm uses LR(1) items to represent valid configurations of an LR(1) parser

An LR(k) item is a pair [P, δ], where

- P is a production $A \rightarrow \beta$ with a $\bullet$ at some position in the *rhs*
- δ is a lookahead string of length $\leq k$ (words or **EOF**)

The $\bullet$ 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 $\beta = B_1 B_1 B_1$ with lookahead $\underline{a}$, can give 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

What's the point of all these lookahead symbols?

- Carry them along to choose the correct reduction, *if there is a choice*
- 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(δ) $\Rightarrow$ ***shift***

⇒ 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 canonical collection completely encodes the transition diagram for the handle-finding ***DFA***

Back to Finding Handles

Revisiting an issue from last class

Parser in a state where the stack (the fringe) was

Expr - *Term*

With lookahead of *

How did it choose to expand *Term* rather than reduce to *Expr?*

- *Lookahead* symbol is the key
- With lookahead of + or -, parser should reduce to *Expr*
- With lookahead of * or /, parser should shift
- Parser uses lookahead to decide
- All this context from the grammar is encoded in the handle recognizing mechanism

Remember this slide from last lecture?

Back to x - 2 * y

Stack	Input	Handle	Action
$	id – num * id	*none*	shift
$ id	– num * id	9,1	red. 9
$ *Factor*	– num * id	7,1	red. 7
$ *Term*	– num * id	4,1	red. 4
$ *Expr*	– num * id	*none*	shift
$ *Expr* –	num * id	*none*	shift
$ *Expr* – num	* id	8,3	red. 8
$ *Expr* – *Factor*	* id	7,3	red. 7
$ *Expr* – *Term*	* id	*none*	shift
$ *Expr* – *Term* *	id	*none*	shift
$ *Expr* – *Term* * id		9,5	red. 9
$ *Expr* – *Term* * *Factor*		5,5	red. 5
$ *Expr* – *Term*		3,3	red. 3
$ *Expr*		1,1	red. 1
$ *Goal*		*none*	accept

shift here

reduce here

1. Shift until TOS is the right end of a handle
2. Find the left end of the handle & reduce

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 $x \in \text{FIRST}(\delta\underline{a})$
- 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 $B \rightarrow \tau \in P$
        $\forall$ $\underline{b} \in \text{FIRST}(\delta\underline{a})$ // $\delta$ might be $\varepsilon$
          if $[B \rightarrow \bullet\,\tau,\underline{b}] \notin s$
            then add $[B \rightarrow \bullet\,\tau,\underline{b}]$ to s
```

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

Example From SheepNoise

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

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

*Closure([Goal→•SheepNoise,*EOF])

Item	***From***
[*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] }

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 also includes *closure(* $[A \rightarrow \beta X \bullet \delta,\underline{a}]$ *)* to fill out the state

The algorithm

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

Goto(S_0 , baa)

- Loop produces

Item	*From*
[*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]

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] }

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

while ( $S$ is still changing )
    $\forall s_j \in S$ and $\forall x \in ( T \cup NT )$
        $s_k \leftarrow goto(s_j, x)$
        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
- $S \subseteq 2^{ITEMS}$, 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] }

Iteration 2 computes

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

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