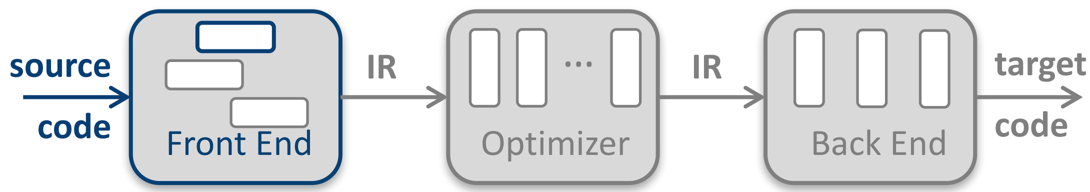
RICE

COMP 412
FALL 2017

Syntax Analysis, VII

The Canonical LR(1) Table Construction

Comp 412



Chapter 3 in EaC2e

Review

LR(1) Items

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

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**)

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.
We call an item like this a <u>*possibility*</u>.

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

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

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

Review

LR(1) Items

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

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

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

What's the point of all these lookahead symbols?

- Carry them along to 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) 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'* →•*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 Canonical Collection of Sets of **LR(1)** items

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

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

Computing Closures

Closure*(*s*) adds all the *possibilities* for the 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* $[A \rightarrow \beta \bullet B\delta,\underline{a}] \in s$
      lookahead $\leftarrow$ FIRST($\delta\underline{a}$) // δ might be ε
      $\forall$ *productions* $B \rightarrow \tau \in P$
        $\forall$ $\underline{b} \in$ *lookahead*
          if $[B \rightarrow \bullet \tau,\underline{b}] \notin s$
            then $s \leftarrow s \cup \{ [B \rightarrow \bullet \tau,\underline{b}] \}$

Generate new lookaheads.
See note on p. 128

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

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, δa is **EOF**
[*SheepNoise* → • baa, **EOF**]	Iter 1, δa is **EOF**
[*SheepNoise* → • *SheepNoise* baa, baa]	Iter 2, δa is baa **EOF**
[*SheepNoise* → • baa, baa]	Iter 2, 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***({ $[A \rightarrow \beta \bullet X\delta,\underline{a}]$ }, 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 ← Ø
  ∀ *items* $[A \rightarrow \beta \bullet X\delta,\underline{a}] \in s$
    new ← *new* ∪ { $[A \rightarrow \beta X \bullet \delta,\underline{a}]$ }
  return ***Closure***(*new*)

- ***Goto***( ) models a transition in the automaton
- Straightforward computation
- ***Goto()*** is <u>not</u> a fixed-point method (but it calls ***Closure()***)

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

Repeatedly construct new states, until all are found

The Algorithm

$s_0 \leftarrow$ ***Closure*** ({ [$S' \rightarrow \bullet S$,<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 (*monotone*)
- $S \subseteq 2^{\text{ITEMS}}$, 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* $\rightarrow$ • *SheepNoise*, EOF], [*SheepNoise* $\rightarrow$ • *SheepNoise* baa, EOF],
[*SheepNoise* $\rightarrow$ • baa, EOF], [*SheepNoise* $\rightarrow$ • *SheepNoise* baa, baa],
[*SheepNoise* $\rightarrow$ • baa, baa] }

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

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

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

State	*SN*	baa
s_0	s_1	s_2
s_1	—	s_3
s_2	—	—
s_3	—	—

Goto Relationships

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

Filling in the ACTION and GOTO Tables

The Table Construction Algorithm

x is the state number

$\forall$ *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*

have *Goal* $\Rightarrow$ *accept*

- at end $\Rightarrow$ *reduce*

Many items generate no table entry

→ Placeholder before a *NT* does not generate an **ACTION** table entry

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

Example from SheepNoise

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

• before $T \Rightarrow$ *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] }

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)

Example from SheepNoise

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

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

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

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

$\boldsymbol{S_3}$ = *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 2*"
(clause 3) (baa, too)

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

ACTION[S_3,EOF] is "*reduce 1*"
(clause 3) (baa, too)

Building the Goto Table

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] }

The Goto table holds just the entries for nonterminal symbols.

(*ignore the column for* baa)

State	*SN*	baa
s_0	s_1	s_2
s_1	—	s_3
s_2	—	—
s_3	—	—

Goto Relationships

ACTION & GOTO Tables

Here are the tables for the left-recursive *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

Remember, this is the left-recursive SheepNoise; EaC2e shows the right-recursive version.

What can go wrong?

The **if-then-else** grammar is worked as an example in EaC2e

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

- 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 if a set *s* contains [$A \rightarrow \gamma \bullet, \underline{a}$] and [$B \rightarrow \gamma \bullet, \underline{a}$] ?

- 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)

~~Implementing the Construction~~

Building the Canonical Collection

Start from s_0 = ***closure***([S'→•S,EOF])

Repeatedly construct new states, unti

The algorithm

$s_0 \leftarrow$ ***closure***([S'→•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*
    $S \leftarrow S \cup \{ s_k \}$
    $k \leftarrow k + 1$

Remember this comment about implementing the equality test at the bottom of the algorithm to build the Canonical Collection of Sets of LR(1) Items?

- Only need to compare core items — the rest will follow
- Represent items as a triple (R,P,L)
  - R is the rule or production
  - P is the position of the placeholder
  - L is the lookahead symbol
- Order items, then
  1. Compare set cardinalities
  2. Compare (in order) by R, P, L

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

STOP