# LR(1) Parsers

# Building LR(1) Tables

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

- Encode actions & transitions into the ACTION & GOTO tables
- If construction succeeds, the grammar is LR(1)
  - "Succeeds" means defines each table entry uniquely

The Big Picture

- Model the state of the parser with "LR(1) items"
- The states will be set of LR(1) items
- Use two functions goto( s, X ) and closure( s )
  - goto() tells which state you reach
  - closure() adds information to round out a state
- Build up the states (sets of LR(1) items) and transitions
- Use this information to fill in the ACTION and GOTO tables

s is a state
X is T or NT

fixed-point algorithm

# LR(1) Items

We represent a valid configuration of an LR(1) parser with a data structure called 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
- $\delta$ is a lookahead string of length $\leq 1$ (word or EOF )

The • in an item indicates which portion of the righthandside of the production we have on the top of the stack

# Meaning of an LR(1) Item

[$A \rightarrow \bullet\beta\gamma$,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

"possibility"

[$A \rightarrow \beta\bullet\gamma$,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 $\beta$ (that is, $\beta$ is on top of the stack)

"partially complete"

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

"complete"

# 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 parser state modeled with items $\{ [A \rightarrow \beta \bullet, \underline{a}], [B \rightarrow \gamma \bullet \delta, \underline{b}] \}$, lookahead of $\underline{a} \Rightarrow$ reduce to $A$; lookahead in FIRST($\delta$) $\Rightarrow$ shift

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

# LR(1) Table Construction

For convenience, we will require that the grammar have an obvious & unique goal symbol — one that does not appear on the rhs of any production.

High-level overview

1 Build the canonical collection of sets of LR(1) Items

a Start with an appropriate initial 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 symbol 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

# Computing Closures

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

- Item $[A \rightarrow \beta \bullet C\delta, \underline{a}]$ in s implies $[C \rightarrow \bullet \tau, x]$ for each production with C 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, too

The algorithm

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

- Classic fixed-point method
- Halts because s ⊂ ITEMS
- Closure "fills out" a state

Lookaheads are generated here

# Example From SheepNoise

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

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

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 |
| 5 | [*SheepNoise* → • baa, baa] | 2, δa is baa |

stop!

4 $\delta\alpha$ is baa baa

$S_0$ (the first state) 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 finds all such items & uses closure() to fill out the state

The algorithm

```
Goto( s, X )
  new ← Ø
  ∀ items [A→β•X δ,a] ∈ s
    new ← new ∪ { [A→βX •δ,a]}
  return closure(new)
```

- Not a fixed-point method!
- Straightforward computation
- Uses closure ( )

# Example from SheepNoise

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

$S_0$ 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* | *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

# Building the Canonical Collection : 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 $\forall x \in (T \cup NT)$
    $t \leftarrow goto(s_j, x)$
    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

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

Repeatedly construct new states, until all are found

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

# Example from SheepNoise

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

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] }

No more for closure!

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

No more for closure!

Iteration 2 computes

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

No more for closure!

# Example from SheepNoise

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

$S_0$ : { [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] }

# Filling in the ACTION and GOTO Tables

The algorithm

x is the state number

$\forall$ set $S_x \in S$

$\forall$ item $i \in S_x$

case 1 — 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"

• before T $\Rightarrow$ shift

case 2 — else if i is $[S' \rightarrow S \bullet, \underline{EOF}]$

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

have Goal $\Rightarrow$ accept

case 3 — else if i is $[A \rightarrow \beta \bullet, \underline{a}]$

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

• at end $\Rightarrow$ reduce

$\forall$ $n \in NT$

if goto($S_x$ ,n) = $S_k$

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

# Example from SheepNoise

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

$S_0$ : { [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] }

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

$S_2$ = 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

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

$S_0$ : { [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 $S_3$" (case 1)

# Example from SheepNoise

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

$S_0$ : { [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" (case 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

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

$S_0$ : { [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" (case 3)

ACTION[$S_2$,baa] is "reduce 2" (case 3)

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

# Example from SheepNoise

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

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

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

ACTION[$S_3$,EOF] is "reduce 1" (case 3)

EOF], [SheepNoise→ SheepNoise • baa, EOF],
Noise • 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] }

ACTION[$S_3$,baa] is "reduce 1", as well

# Example from SheepNoise

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

## 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] }

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

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

Only 1 transition in the entire GOTO table

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

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

# ACTION & GOTO Tables

Here are the tables for the augmented 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 |

Note that this is the left-recursive SheepNoise; the book shows the right-recursive version.

# What can go wrong?

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

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

- 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

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

# 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

# Non-LL Grammars

| | | | |
|---|---|---|---|
| 0 | *B* | → | *R* |
| 1 | | \| | ( *B* ) |
| 2 | *R* | → | *E* = *E* |
| 3 | *E* | → | a |
| 4 | | \| | b |
| 5 | | \| | ( *E* + *E* ) |

Example from D.E Knuth, "Top-Down Syntactic Analysis," Acta Informatica, 1:2 (1971), pages 79-110

This grammar is actually LR(0)

| | | | |
|---|---|---|---|
| 0 | *S* | → | a *A* b |
| 1 | | \| | c |
| 2 | *A* | → | b *S* |
| 3 | | \| | *B* b |
| 4 | *B* | \| | a *A* |
| 5 | | \| | c |

Example from Lewis, Rosenkrantz, & Stearns book, "Compiler Design Theory," (1976), Figure 13.1

# Summary

| | Advantages | Disadvantages |
|---|---|---|
| Top-down Recursive descent, LL(1) | Fast<br>Good locality<br>Simplicity<br>Good error detection | Hand-coded<br>High maintenance<br>Right associativity |
| LR(1) | Fast<br>Deterministic langs.<br>Automatable<br>Left associativity | Large working sets<br>Poor error messages<br>Large table sizes |

# Exercise

Consider the following grammar:

$$
\begin{array}{lcl}
Start & \rightarrow & S \\
S & \rightarrow & A\ \mathtt{a} \\
A & \rightarrow & B\ C \\
 & | & B\ C\ \mathtt{f} \\
B & \rightarrow & \mathtt{b} \\
C & \rightarrow & \mathtt{c}
\end{array}
$$

**a.** Construct the canonical collection of sets of LR(1) items for this grammar.

**b.** Derive the `Action` and `Goto` tables.

**c.** Is the grammar LR(1)?