Parsing VI
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.

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

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)

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

Terminal or non-terminal

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

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

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

What's the point of all these lookahead symbols?

- Carry them along to choose the correct reduction, *if there is a choice*
- Lookaheads are bookkeeping, unless item has $\cdot$ 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 canonical collection completely encodes the transition diagram for the handle-finding ***DFA***

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)
    $\forall$ *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 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*

Pay close attention to lookahead generation

Example From SheepNoise

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

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**

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

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 $\leftarrow \emptyset$
  $\forall$ *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**] }

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]

Building the Canonical Collection

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

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
- $S \subseteq 2^{\text{ITEMS}}$, 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] }

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 .

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] }

Filling in the ACTION and GOTO Tables

The algorithm

x is the state number

$\forall$ *set* $S_x \in S$

$\quad\forall$ *item* $i \in S_x$

$\quad\quad$ *if* i *is* $[A \rightarrow \beta \bullet \underline{a}\delta, \underline{b}]$ *and* $goto(S_x, \underline{a}) = S_k$, $\underline{a} \in T$

$\quad\quad\quad$ *then* ACTION[x,$\underline{a}$] $\leftarrow$ "*shift k*"

$\quad\quad$ *else if* i *is* $[S' \rightarrow S \bullet, \text{EOF}]$

$\quad\quad\quad$ *then* ACTION[x ,$\underline{a}$] $\leftarrow$ "*accept*"

$\quad\quad$ *else if* i *is* $[A \rightarrow \beta \bullet, \underline{a}]$

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

$\quad\forall$ $n \in NT$

$\quad\quad$ *if* $goto(S_x, n) = S_k$

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

- before $T \Rightarrow$ *shift*

have *Goal* $\Rightarrow$ *accept*

- at end $\Rightarrow$ *reduce*

Many items generate no table entry

$\rightarrow$ *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* ⇒ *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)

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

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] }

S_1 = *Goto*(S_0 , ***SheepNoise***) =
{ OF], [***SheepNoise***→ ***SheepNoise*** • baa, EOF],
oise • baa, baa] }

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

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

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

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

ACTION[S_2,EOF] is "*reduce 3*" (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] }

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		
State	EOF	<u>baa</u>
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

The grammar

1	*Goal*	→	SheepNoise
2	*SheepNoise*	→	SheepNoise <u>baa</u>
3		|	<u>baa</u>

What can go wrong?

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

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

- 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 *(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

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

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

- The complete left context
- The next k terminals

Thus, LR(k) examines more context

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

"*… 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