Parsing V
The LR(1) Table Construction

LR(*1*) items

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

An LR(*1*) item is a pair [*P, a*], where

P is a production $A \rightarrow \beta$ with a • at some position in the *rhs and* ***a*** is a lookahead word (or EOF)

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

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

[$A \rightarrow \beta\bullet\gamma$,a] means that the input seen so far is consistent with $A \rightarrow \beta\gamma$ at this point in the parse, *and* that the parser has already recognized β

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

LR(1) Table Construction

High-level overview

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

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, *goto(*S_k,*X)*, *where X is all NT and T*

- 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***

The SheepNoise Grammar (*revisited*)

We will use this grammar extensively in today's lecture

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

Computing FIRST Sets

Define FIRST as

- If $\alpha \Rightarrow^* \underline{a}\beta$, $\underline{a} \in T$, $\beta \in (T \cup NT)^*$, then $\underline{a} \in$ FIRST(α)
- If $\alpha \Rightarrow^* \varepsilon$, then $\varepsilon \in$ FIRST(α)

Note: if $\alpha = X\beta$, FIRST(α) = FIRST(X)

To compute FIRST

- Use a fixed-point method
- FIRST(A) $\in 2^{(T \cup \varepsilon)}$
- Loop is monotonic

$\Rightarrow$ Algorithm halts

Computing FIRST Sets

for each $x \in T$*, FIRST(x)* $\leftarrow \{x\}$
for each $A \in NT$*, FIRST(A)* $\leftarrow \emptyset$

while (FIRST sets are still changing)
    for each $p \in P$*, of the form* $A \rightarrow \beta$*,*
        if β *is* $B_1B_2 \ldots B_k$ *where* $B_i \in T \cup NT$ *then begin*
            $FIRST(A) \leftarrow FIRST(A) \cup (FIRST(B_1) - \{ \varepsilon \})$
            for $i \leftarrow 1$ *to* $k\text{-}1$ *by 1 while* $\varepsilon \in FIRST(B_i)$
                $FIRST(A) \leftarrow FIRST(A) \cup (FIRST(B_{i+1}) - \{ \varepsilon \})$
        if $i = k$ *and* $\varepsilon \in FIRST(B_k)$
            then $FIRST(A) \leftarrow FIRST(A) \cup \{ \varepsilon \}$

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 )
    ∀ items [A → β • Bδ,a] ∈ s
      ∀ productions B → τ ∈ P
        ∀ b ∈ FIRST(δa) // δ might be ε
          if [B→ • τ,b] ∉ s
            then add [B→ • τ,b] to s
```

- Classic fixed-point method
- Halts because $s \subset$ LR ITEMS
- *Closure* "*fills out*" *state s*

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* →• baa *SheepNoise* ,EOF]	**1, δa is EOF**
[*SheepNoise* → • baa,EOF]	**1, δa is EOF**

So, S_0 is

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

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 [$A \rightarrow \beta X \bullet \delta, \underline{a}$] *(easy part)*
- Also computes *closure(* [$A \rightarrow \beta X \bullet \delta, \underline{a}$] *)* *(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 forward

Example from SheepNoise

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

Goto(S_0 , baa)

- Loop produces

Item	*From*
[*SheepNoise* →baa• SheepNoice, EOF]	Item 2 in s_0
[*SheepNoise* →baa•, EOF]	Item 3 in s_0
[*SheepNoise* →•baa, EOF]	Item 1 in s_1
[*SheepNoise* →•baa SheepNoise , EOF]	Item 1 in s_1

- Closure adds two items since • is before SheepNoise in first

Example from SheepNoise

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

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

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

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

Building the Canonical Collection

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

Repeatedly construct new states, until all are found

The algorithm

```
s0 ← closure( [S'→S,EOF])
S ← { s0 }
k ← 1
while ( S is still changing)
   ∀ sj ∈ S and ∀ x ∈ ( T ∪ NT)
        sk ← goto(sj,x)
        record sj → sk on x
   if sk ∉ S then
        S ← S ∪ sk
        k ← k + 1
```

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

Example from SheepNoise

Starts with S_0

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

Example from SheepNoise

Starts with S_0

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

Iteration 1 computes

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

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

Example from SheepNoise

Starts with S_0

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

Iteration 1 computes

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

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

Iteration 2 computes

Goto(S_2,baa) creates S_2

S_3 = *Goto*(S_2,SheepNoise) = {[*SheepNoise*→ baa SheepNoise •, EOF]}

Example from SheepNoise

Starts with S_0

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

Iteration 1 computes

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

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

Iteration 2 computes

Goto(S_2,**baa**) creates S_2

S_3 = *Goto*(S_2,**SheepNoise**) = {[***SheepNoise***→ **baa** **SheepNoise** •, **EOF**]}

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

Example (*grammar & sets*)

Simplified, right recursive expression grammar

Goal → *Expr*
Expr → *Term* – *Expr*
Expr → *Term*
Term → *Factor* * *Term*
Term → *Factor*
Factor → ident

Symbol	FIRST
Goal	{ ident }
Expr	{ ident }
Term	{ ident }
Factor	{ ident }
–	{ – }
*	{ * }
ident	{ ident }

Example (*building the collection*)

Initialization Step

$s_0 \leftarrow$ *closure*({ [*Goal* → •*Expr* , EOF] })

{ [*Goal* → • *Expr* , EOF], [*Expr* → • *Term* – *Expr* , EOF],
[*Expr* → • *Term* , EOF], [*Term* → • *Factor* * Term , EOF],
[*Term* → • *Factor* * *Term* , –], [*Term* → • *Factor* , EOF],
[*Term* → • *Factor* , –], [*Factor* → • ident , EOF],
[*Factor* → • ident , –], [*Factor* → • ident , *] }

$S \leftarrow \{s_0\}$

Example (*building the collection*)

Iteration 1

$s_1 \leftarrow goto(s_0, Expr)$

$s_2 \leftarrow goto(s_0, Term)$

$s_3 \leftarrow goto(s_0, Factor)$

$s_4 \leftarrow goto(s_0, \underline{\text{ident}})$

Iteration 2

$s_5 \leftarrow goto(s_2, -)$

$s_6 \leftarrow goto(s_3, *)$

Iteration 3

$s_7 \leftarrow goto(s_5, Expr)$

$s_8 \leftarrow goto(s_6, Term)$

Example (*Summary*)

S_0 : { [*Goal* → • *Expr* , EOF], [*Expr* → • *Term* – *Expr* , EOF],
[*Expr* → • *Term* , EOF], [*Term* → • *Factor* * *Term* , EOF],
[*Term* → • *Factor* * *Term* , –], [*Term* → • *Factor* , EOF],
[*Term* → • *Factor* , –], [*Factor* → • ident , EOF],
[*Factor* → • ident , –], [*Factor* → • ident, *] }

S_1 : { [*Goal* → *Expr* •, EOF] }

S_2 : { [*Expr* → *Term* • – *Expr* , EOF], [*Expr* → *Term* •, EOF] }

S_3 : { [*Term* → *Factor* • * *Term* , EOF],[*Term* → *Factor* • * *Term* , –],
[*Term* → *Factor* •, EOF], [*Term* → *Factor* •, –] }

S_4 : { [*Factor* → ident •, EOF],[*Factor* → ident •, –], [*Factor* → ident •, *] }

S_5 : { [*Expr* → *Term* – • *Expr* , EOF], [*Expr* → • *Term* – *Expr* , EOF],
[*Expr* → • *Term* , EOF], [*Term* → • *Factor* * *Term* , –],
[*Term* → • *Factor* , –], [*Term* → • *Factor* * *Term* , EOF],
[*Term* → • *Factor* , EOF], [*Factor* → • ident , *],
[*Factor* → • ident , –], [*Factor* → • ident , EOF] }

Example (*Summary*)

S_6 : { [*Term* → *Factor* * • *Term* , EOF], [*Term* → *Factor* * • *Term* , –],
[*Term* → • *Factor* * *Term* , EOF], [*Term* → • *Factor* * *Term* , –],
[*Term* → • *Factor* , EOF], [*Term* → • *Factor* , –],
[*Factor* → • <u>ident</u> , EOF], [*Factor* → • <u>ident</u> , –], [*Factor* → • <u>ident</u> , *] }

S_7: { [*Expr* → *Term* – *Expr* •, EOF] }

S_8 : { [*Term* → *Factor* * *Term* •, EOF], [*Term* → *Factor* * *Term* •, –] }

Example (*Summary*)

The Goto Relationship *(from the construction)*

State	Expr	Term	Factor	-	*	Ident
0	1	2	3			4
1						
2				5		
3					6	
4						
5	7	2	3			4
6		8	3			4
7						
8						

Filling in the ACTION and GOTO Tables

The algorithm

```
∀ set s_x ∈ S
    ∀ item i ∈ s_x
        if  i is [A→β •ad,b] and goto(s_x,a) = s_k , a ∈ T
            then ACTION[x,a] ← "shift k"
        else if  i is [S'→S •,EOF]
            then ACTION[x ,a] ← "accept"
        else if  i is [A→β •,a]
             then ACTION[x,a] ← "reduce A→β"
    ∀ n ∈ NT
        if  goto(s_x ,n) = s_k
            then GOTO[x,n] ← k
```

***x* is the state number**

Many items generate no table entry

- *Closure( )* instantiates FIRST(*X*) directly for [*A*→β•*X*δ,a]

Example (*Filling in the tables*)

The algorithm produces the following table

	ACTION				GOTO		
	Ident	-	*	EOF	*Expr*	*Term*	*Factor*
0	s 4				1	2	3
1				acc			
2		s 5		r 3			
3		r 5	s 6	r 5			
4		r 6	r 6	r 6			
5	s 4				7	2	3
6	s 4					8	3
7				r 2			
8		r 4		r 4			

Plugs into the skeleton LR(1) parser

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

EaC includes a worked example

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 fundamental ambiguity is called a *reduce/reduce error*
- Modify the grammar to eliminate it

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

LR(k) versus LL(k) (*Top-down Recursive Descent*)

Finding Reductions

LR(k) ⇒ Each reduction in the parse is detectable with

1. the complete left context,
2. the reducible phrase, itself, and
3. the k terminal symbols to its right

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

1. The complete left context
2. The next k terminals

Thus, LR(k) examines more context

"... *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