Parsing VI
The LR(1) Table Construction

Building the Canonical Collection

Start from s_0 = *closure(* [$S' \rightarrow S$,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
- $S \subseteq 2^{\text{ITEMS}}$, so S is finite

Worklist version is faster

Computing Closures

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

- Any item [$A \rightarrow \beta \bullet B\delta$,a] implies [$B \rightarrow \bullet\tau$,$x$] for each production with B on the *lhs*, and each $x \in$ FIRST(δa)
- 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 $s \subset$ 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( )*

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] }

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

Computing Gotos

Goto(s,x) computes the state that the parser would reach if it recognized an *x* while in state *s*

- *Goto(* { [*A*→β•*X*δ,a] }, *X*) produces [*A*→β*X*•δ,a] *(easy part)*
- Should also includes *closure(* [*A*→β*X*•δ,a]) *(fill out the state)*

The algorithm

```
Goto( s, X )
   new ← Ø
   ∀ items [A→β•Xδ,a] ∈ s
      new ← new ∪ [A→βX•δ,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*→ • *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

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] }

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 ← {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)*

$s_4 \leftarrow goto(s_0$, 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* , EOF], [*Factor* → · ident , EOF],
[*Term* → · *Factor* * *Term* , –],[*Term* → · *Factor* , –],
[*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* ·, EOF],
[*Term* → *Factor* · * *Term* , –], [*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* , EOF],
[*Term* → · *Factor* , EOF], [*Factor* → · ident , EOF],
[*Term* → · *Factor* * *Term* , –],[*Term* → · *Factor* , –],
[*Factor* → · ident , –], [*Factor* → · ident , *] }

Example (*Summary*)

S_6 : { [*Term* → *Factor* * · *Term* , EOF], [*Term* → *Factor* * · *Term* , –],
[*Term* → · *Factor* * *Term* , EOF], [*Term* → · *Factor* * *Term* , –],
[*Term* → · *Factor* , EOF], [*Term* → · *Factor* , –],
[*Factor* → · ident , EOF], [*Factor* → · ident , –], [*Factor* → · ident , *] }

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

```
for each set s_x ∈ S
  for each 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"
  for each 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*→β•aγ,b] and [*B*→β•,a] ?

- First item generates "shift", second generates "reduce"
- Both define ACTION[s,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*→γ•, a] and [*B*→γ•, a] ?

- Each generates "reduce", but with a different production
- Both define ACTION[s,a] — cannot do both reductions
- This fundamental ambiguity is called a *reduce/reduce error*
- Modify the grammar to eliminate it *(PL/I's overloading of (...))*

EaC includes a worked example

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

3. LL(*k*) ⇒ Parser must select the reduction based on
   1. The complete left context
   2. The next *k* terminals
2. 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