Syntactic Analysis

Building a SLR Parser
Building a LR(1) Parser
Building a LALR(1) Parser
Building LR(k) Parsers

Outline

- Limitations in LR(0) languages
- Building a SLR(1) parser engine
- Limitations in SLR(1) languages
- Building a LR(1) parser engine
- Building a LALR(1) parse engine

Building a LR(0) parser engine

- Add the special production **S' → S** $
- Find the items of the CFG
- Create the DFA
  - using **closure** and **goto** functions
- Build the parse table

Example

- String of one more more left parentheses followed by the same number of right parentheses
  <S> → <X> $
  <X> → (<X>)
  <X> → ( )
- String of zero or more more left parentheses followed by the same number of right parentheses

Example

- String of one more more left parentheses followed by the same number of right parentheses
  <S> → <X> $
  <X> → (<X>)
  <X> → ( )

- String of zero or more more left parentheses followed by the same number of right parentheses
  <S> → <X> $
  <X> → (<X>)
  <X> → ε

Example

The grammar

<S> → <X> $
<X> → (<X>)
<X> → ε

Items

<S> → • <X> $
<S> → <X> • $
<X> → • (<X>)
<X> → (• <X>)
<X> → (<X> •)
<X> → (<X>) •
<X> → ????

Example

The grammar

<S> → <X> $
<X> → (<X>)
<X> → ε

Items

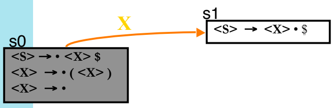
<S> → • <X> $
<S> → <X> • $
<X> → • (<X>)
<X> → (• <X>)
<X> → (<X> •)
<X> → (<X>) •
<X> → •

Building the DFA for the Example

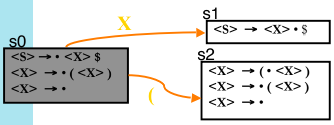
s0

<S> → • <X> $
<X> → • (<X>)
<X> → •

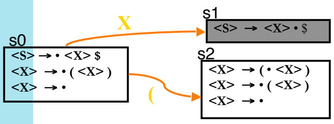
Building the DFA for the Example



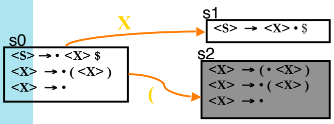
Building the DFA for the Example



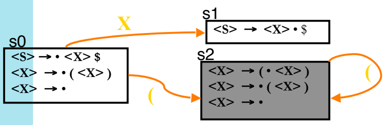
Building the DFA for the Example



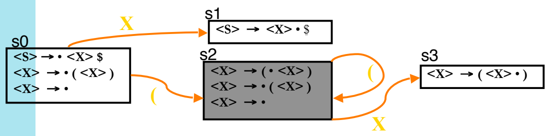
Building the DFA for the Example



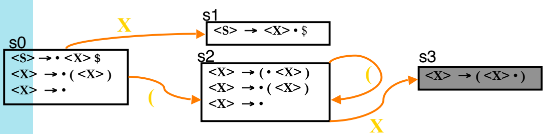
Building the DFA for the Example



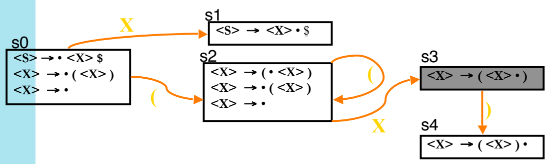
Building the DFA for the Example



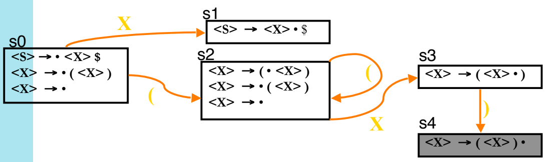
Building the DFA for the Example



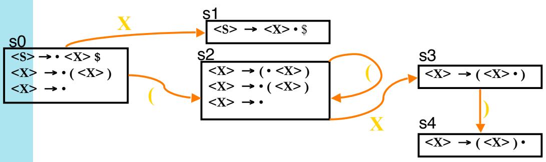
Building the DFA for the Example



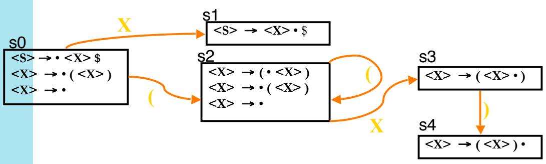
Building the DFA for the Example



Building the DFA for the Example

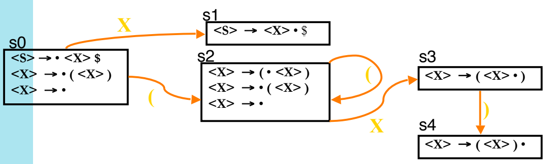


Building the Parse Table



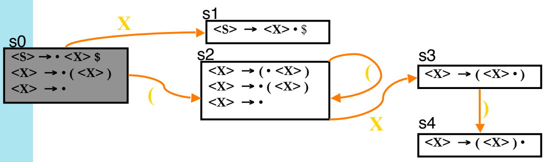
Building the Parse Table

	ACTION			Goto
State	(	)	$	X
s0				
s1				
s2				
s3				
s4				



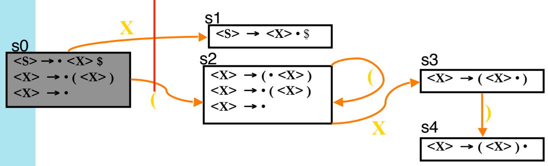
Building the Parse Table

	ACTION			Goto
State	(	)	$	X
s0				
s1				
s2				
s3				
s4				



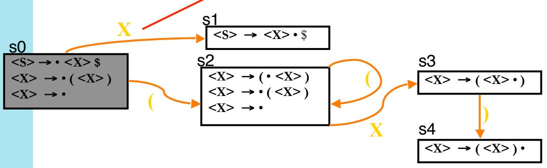
Building the Parse Table

	ACTION			Goto
State	(	)	$	X
s0	shift to s2			
s1				
s2				
s3				
s4				



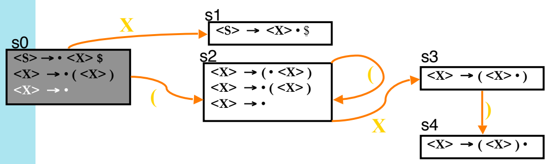
Building the Parse Table

	ACTION			Goto
State	(	)	$	X
s0	shift to s2			goto s1
s1				
s2				
s3				
s4				



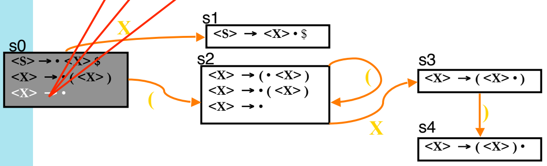
Building the Parse Table

	ACTION			Goto
State	(	)	$	X
s0	shift to s2			goto s1
s1				
s2				
s3				
s4				



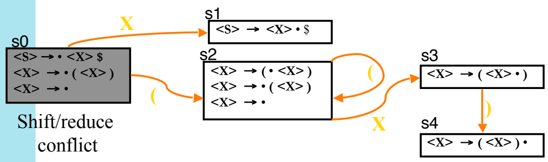
Building the Parse Table

	ACTION			Goto
State	(	)	$	X
s0	shift to s2 / reduce (3)	reduce (3)	reduce (3)	goto s1
s1				
s2				
s3				
s4				



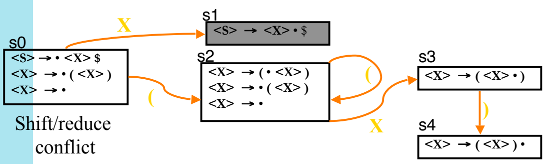
Building the Parse Table

	ACTION			Goto
State	(	)	$	X
s0	shift to s2 / reduce (3)	reduce (3)	reduce (3)	goto s1
s1				
s2				
s3				
s4				



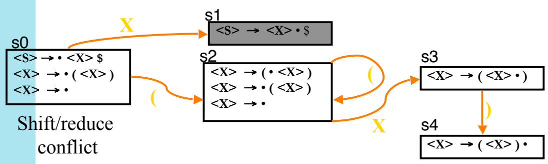
Building the Parse Table

	ACTION			Goto
State	(	)	$	X
s0	shift to s2 / reduce (3)	reduce (3)	reduce (3)	goto s1
s1				
s2				
s3				
s4				



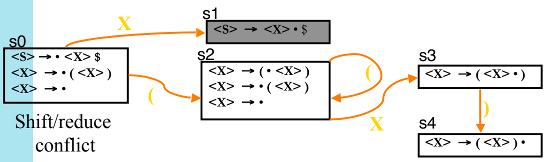
Building the Parse Table

	ACTION			Goto
State	(	)	$	X
s0	shift to s2 / reduce (3)	reduce (3)	reduce (3)	goto s1
s1			accept	
s2				
s3				
s4				



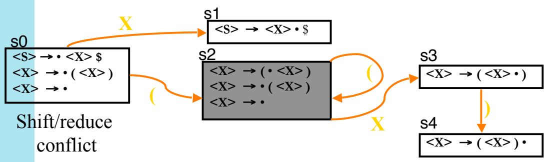
Building the Parse Table

State	ACTION (	)	$	Goto X
s0	shift to s2 / reduce (3)	reduce (3)	reduce (3)	goto s1
s1	error	error	accept	
s2				
s3				
s4				



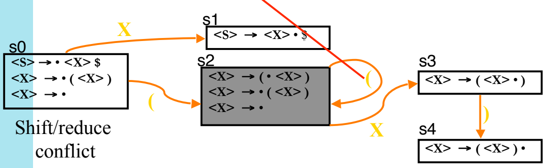
Building the Parse Table

State	ACTION (	)	$	Goto X
s0	shift to s2 / reduce (3)	reduce (3)	reduce (3)	goto s1
s1	error	error	accept	
s2				
s3				
s4				



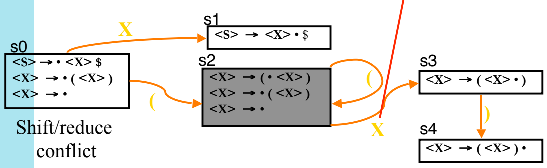
Building the Parse Table

State	ACTION (	)	$	Goto X
s0	shift to s2 / reduce (3)	reduce (3)	reduce (3)	goto s1
s1	error	error	accept	
s2	shift to s2			
s3				
s4				



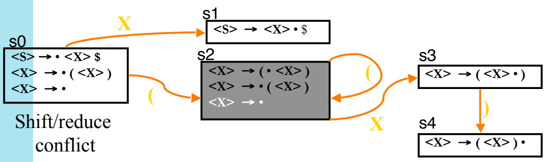
Building the Parse Table

State	ACTION (	)	$	Goto X
s0	shift to s2 / reduce (3)	reduce (3)	reduce (3)	goto s1
s1	error	error	accept	
s2	shift to s2			goto s3
s3				
s4				



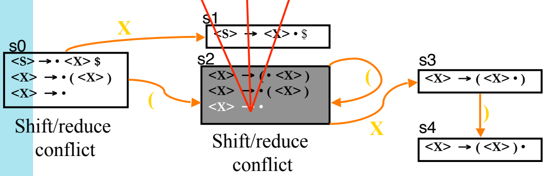
Building the Parse Table

State	ACTION (	ACTION)	ACTION $	Goto X
s0	shift to s2 / reduce (3)	reduce (3)	reduce (3)	goto s1
s1	error	error	accept	
s2	shift to s2			goto s3
s3				
s4				



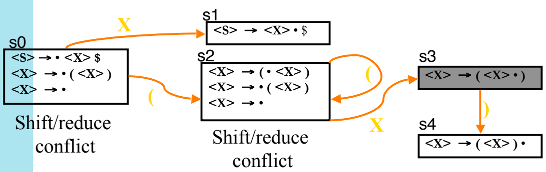
Building the Parse Table

State	ACTION (	ACTION)	ACTION $	Goto X
s0	shift to s2 / reduce (3)	reduce (3)	reduce (3)	goto s1
s1	error	error	accept	
s2	shift to s2 / reduce (3)	reduce (3)	reduce (3)	goto s3
s3				
s4				



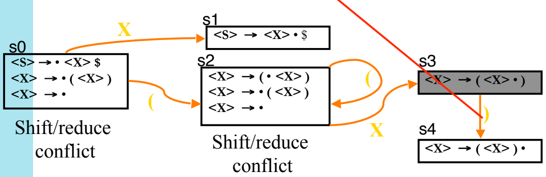
Building the Parse Table

State	ACTION (	ACTION)	ACTION $	Goto X
s0	shift to s2 / reduce (3)	reduce (3)	reduce (3)	goto s1
s1	error	error	accept	
s2	shift to s2 / reduce (3)	reduce (3)	reduce (3)	goto s3
s3				
s4				



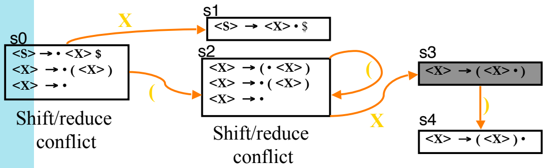
Building the Parse Table

State	ACTION (	ACTION)	ACTION $	Goto X
s0	shift to s2 / reduce (3)	reduce (3)	reduce (3)	goto s1
s1	error	error	accept	
s2	shift to s2 / reduce (3)	reduce (3)	reduce (3)	goto s3
s3		shift to s4		
s4				



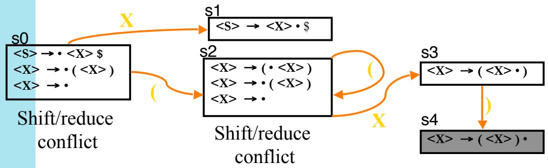
Building the Parse Table

State	ACTION			Goto
	(	)	$	X
s0	shift to s2 / reduce (3)	reduce (3)	reduce (3)	goto s1
s1	error	error	accept	
s2	shift to s2 / reduce (3)	reduce (3)	reduce (3)	goto s3
s3	error	shift to s4	error	
s4				



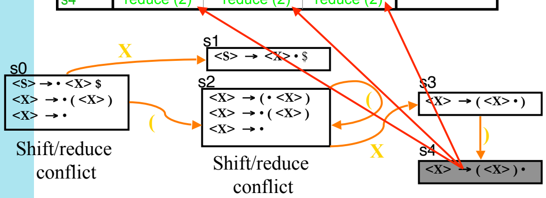
Building the Parse Table

State	ACTION			Goto
	(	)	$	X
s0	shift to s2 / reduce (3)	reduce (3)	reduce (3)	goto s1
s1	error	error	accept	
s2	shift to s2 / reduce (3)	reduce (3)	reduce (3)	goto s3
s3	error	shift to s4	error	
s4				



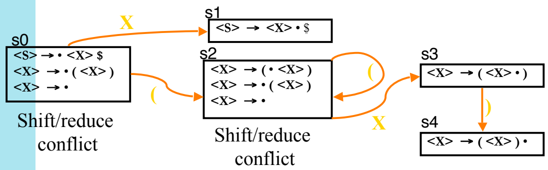
Building the Parse Table

State	ACTION			Goto
	(	)	$	X
s0	shift to s2 / reduce (3)	reduce (3)	reduce (3)	goto s1
s1	error	error	accept	
s2	shift to s2 / reduce (3)	reduce (3)	reduce (3)	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	



Building the Parse Table

State	ACTION			Goto
	(	)	$	X
s0	shift to s2 / reduce (3)	reduce (3)	reduce (3)	goto s1
s1	error	error	accept	
s2	shift to s2 / reduce (3)	reduce (3)	reduce (3)	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	



Building the Parse Table

State	ACTION (	ACTION)	ACTION $	Goto X
s0	shift to s2 / reduce (3)	reduce (3)	reduce (3)	goto s1
s1	error	error	accept	
s2	shift to s2 / reduce (3)	reduce (3)	reduce (3)	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	

How do we get rid of these shift/reduce conflicts?

Limitations of LR(0) grammars

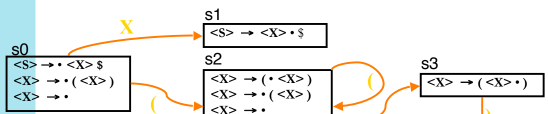
- Many Shift/Reduce Conflicts
- Reason:
  - An item X → α • in the current state identifies a reduction
  - But does not select when to reduce
  - Thus, have to perform the reduction on all input symbols
- Solution:
  - Use FOLLOW set to guide Action in the table!
  - Clearly we should only reduce when the input is showing a terminal in the FOLLOW of that non-terminal

SLR Parsing Table

- Algorithm
  - Construct $C = \{I_0, I_1, \ldots, I_n\}$ the collection of LR(0) items for G'
  - State i is constructed from I_i with parsing actions as follows:
    - If [A → α•aβ] is in I_i and and goto(I_i,a) = I_j, with *a* as terminal then action[i,a] is "shift j"
    - If [A → α•] is in I_i then set action[i,a] to "reduce A → α" for all *a* in FOLLOW(A) where A may not be S'
    - If [S' → S•] is in I_i then set action[i,$] to "accept".
- Difference to LR(0) parsing algorithm?
  - Selectively set reduce only on some terminals

Example with SLR

State	ACTION (	ACTION)	ACTION $	Goto X
s0	shift to s2 / reduce (3)	reduce (3)	reduce (3)	goto s1
s1	error	error	accept	
s2	shift to s2 / reduce (3)	reduce (3)	reduce (3)	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	

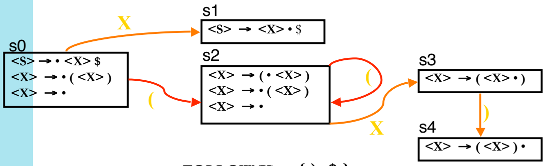


Shift/reduce conflict

FOLLOW(X) = {), $ }

Example with SLR

State	ACTION			Goto
	(	)	$	X
s0	shift to s2	reduce (3)	reduce (3)	goto s1
s1	error	error	accept	
s2	shift to s2	reduce (3)	reduce (3)	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	



FOLLOW(X) = {), $ }

Building a SLR parser engine

- Add the special production **S’** → **S** $
- Find the items of the CFG
- Create the DFA
  - using **closure** and **goto** functions
- Build the parse table
  - using FOLLOW Sets

LR(k) Items

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

An LR(k) item is a pair [P, δ], where

P is a production A→β with a • at some position in the *rhs*

δ is a lookahead string of length ≤ k (words or **EOF**)

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

[A→•βγ,<u>a</u>] means that the input seen so far is consistent with the use of A →βγ immediately after the symbol on top of the stack

[A →β•γ,<u>a</u>] means that the input sees so far is consistent with the use of A →βγ at this point in the parse, ***and*** that the parser has already recognized β.

[A →βγ•,<u>a</u>] means that the parser has seen βγ, ***and*** that a lookahead symbol of <u>a</u> is consistent with reducing to A.

LR(1) Items

The production A→β, where β = $B_1B_2B_3$ with lookahead <u>a</u>, can give rise to 4 items

[A→•$B_1B_2B_3$,<u>a</u>], [A→B_1•B_2B_3,<u>a</u>], [A→B_1B_2•B_3,<u>a</u>], & [A→$B_1B_2B_3$•,<u>a</u>]

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 • at right end
  - Has no direct use in [A→β•γ,<u>a</u>]
  - In [A→β•,<u>a</u>], a lookahead of <u>a</u> implies a reduction by A →β
  - For { [A→β•,<u>a</u>],[B→γ•δ,<u>b</u>] }, <u>a</u> ⇒ ***reduce*** to A; FIRST(δ) ⇒ ***shift***

⇒ 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'* →•*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*→β•*B*δ,a] implies [*B*→•τ,*x*] for each production with *B* on the *lhs*, and each *x* ∈ FIRST(δa)
- Since β*B*δ is valid, any way to derive β*B*δ 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* ⊂ ITEMS
- ➢ Worklist version is faster

  Closure "fills out" a state

Filling in the ACTION & GOTO Tables

The algorithm

```
∀ set sx ∈ S
  ∀ item i ∈ sx
    if i is [A→β •ad,b] and goto(sx,a) = sk , 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(sx ,n) = sk
      then GOTO[x,n] ← k
```

x is the state number

Many items generate no table entry

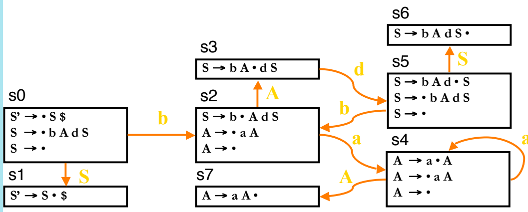
– *Closure( )* instantiates FIRST(*X*) directly for [*A*→β•*X*δ,a]

Example with LR(0) Construction

(1) S' → S $
(2) S → b A d S
(3) S → ε
(4) A → a A
(5) A → ε

Example with LR(0) Construction

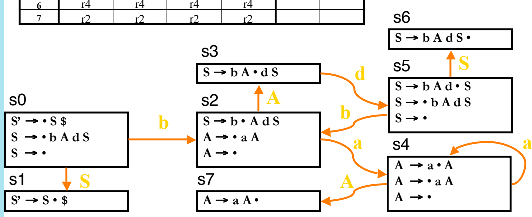
(1) S' → S $
(2) S → b A d S
(3) S → ε
(4) A → a A
(5) A → ε



Example with LR(0) Construction

	Action				Goto	
	a	b	d	$	S	A
0	r1	s2 / r1	r1	r1	1	
1				acc		
2	s4 / r5	r5	r5	r5		3
3			s5			
4	s4 / r5	r5	r5	r5		6
5	r3	s2 / r3	r3	r3		7
6	r4	r4	r4	r4		
7	r2	r2	r2	r2		

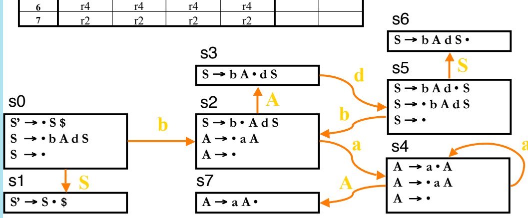
(1) S' → S $
(2) S → b A d S
(3) S → ε
(4) A → a A
(5) A → ε



Example with LR(0) Construction

	Action				Goto	
	a	b	d	$	S	A
0	r1	s2 / r1	r1	r1	1	
1				acc		
2	s4 / r5	r5	r5	r5		3
3			s5			
4	s4 / r5	r5	r5	r5		6
5	r3	s2 / r3	r3	r3		7
6	r4	r4	r4	r4		
7	r2	r2	r2	r2		

(1) S' → S $
(2) S → b A d S
(3) S → ε
(4) A → a A
(5) A → ε

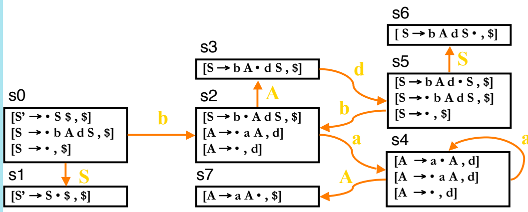


Example with LR(1) Construction

(1) S' → S $
(2) S → b A d S
(3) S → ε
(4) A → a A
(5) A → ε

Example with LR(1) Construction

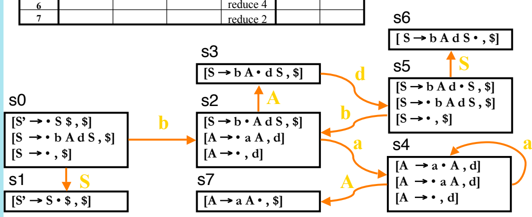
(1) S’ → S $
(2) S → b A d S
(3) S → ε
(4) A → a A
(5) A → ε



Example with LR(1) Construction

	Action				Goto	
	a	b	d	$	S	A
0		shift 2		reduce 3	goto 1	
1				accept		
2	shift 4		reduce 5			goto 3
3			shift 5			
4	shift 4		reduce 5			goto 6
5		shift 2		reduce 3		goto 7
6				reduce 4		
7				reduce 2		

(1) S’ → S $
(2) S → b A d S
(3) S → ε
(4) A → a A
(5) A → ε



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 (...))*

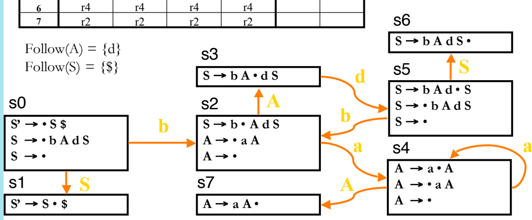
In either case, the grammar is not LR(1)

Example with SLR Construction

	Action				Goto	
	a	b	d	$	S	A
0	r1	s2 / r1	r1	r1	1	
1				acc		
2	s4 / r5	r5	r5	r5		3
3			s5			
4	s4 / r5	r5	r5	r5		6
5	r3	s2 / r3	r3	r3		7
6	r4	r4	r4	r4		
7	r2	r2	r2	r2		

(1) S’ → S $
(2) S → b A d S
(3) S → ε
(4) A → a A
(5) A → ε

Follow(A) = {d}
Follow(S) = {$}

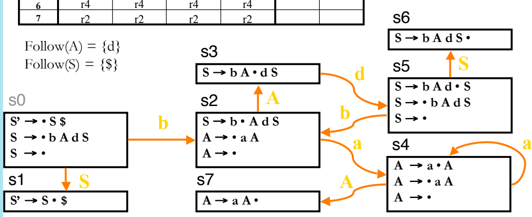


Example with SLR Construction

	Action				Goto	
	a	b	d	$	S	A
0		s2		r1	1	
1				acc		
2	s4 / r5	r5	r5	r5		3
3			s5			
4	s4 / r5	r5	r5	r5		6
5	r3	s2 / r3	r3	r3		7
6	r4	r4	r4	r4		
7	r2	r2	r2	r2		

(1) S' → S $
(2) S → b A d S
(3) S → ε
(4) A → a A
(5) A → ε

Follow(A) = {d}
Follow(S) = {$}

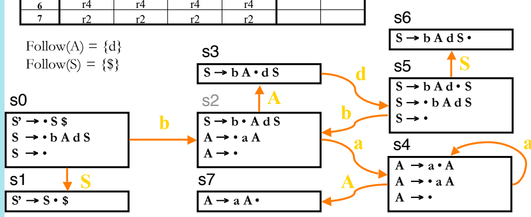


Example with SLR Construction

	Action				Goto	
	a	b	d	$	S	A
0		s2		r1	1	
1				acc		
2	s4		r5			3
3			s5			
4	s4 / r5	r5	r5	r5		6
5	r3	s2/r3	r3	r3		7
6	r4	r4	r4	r4		
7	r2	r2	r2	r2		

(1) S' → S $
(2) S → b A d S
(3) S → ε
(4) A → a A
(5) A → ε

Follow(A) = {d}
Follow(S) = {$}

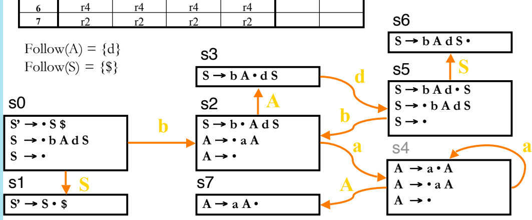


Example with SLR Construction

	Action				Goto	
	a	b	d	$	S	A
0		s2		r1	1	
1				acc		
2	s4		r5			3
3			s5			
4	s4		r5			6
5	r3	s2/r3	r3	r3		7
6	r4	r4	r4	r4		
7	r2	r2	r2	r2		

(1) S' → S $
(2) S → b A d S
(3) S → ε
(4) A → a A
(5) A → ε

Follow(A) = {d}
Follow(S) = {$}

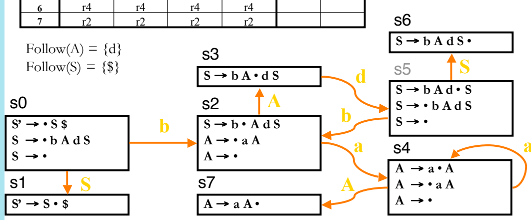


Example with SLR Construction

	Action				Goto	
	a	b	d	$	S	A
0		s2		r1	1	
1				acc		
2	s4		r5			3
3			s5			
4	s4		r5			6
5		s2		r3		7
6	r4	r4	r4	r4		
7	r2	r2	r2	r2		

(1) S' → S $
(2) S → b A d S
(3) S → ε
(4) A → a A
(5) A → ε

Follow(A) = {d}
Follow(S) = {$}

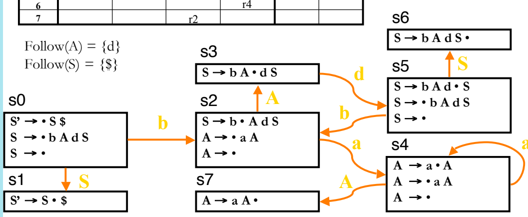


Example with SLR Construction

	Action				Goto	
	a	b	d	$	S	A
0		s2		r1	1	
1				acc		
2	s4		r5			3
3			s5			
4	s4		r5			6
5		s2		r3		7
6				r4		
7			r2			

(1) S’ → S $
(2) S → b A d S
(3) S → ε
(4) A → a A
(5) A → ε

Follow(A) = {d}
Follow(S) = {$}



Shrinking the Tables

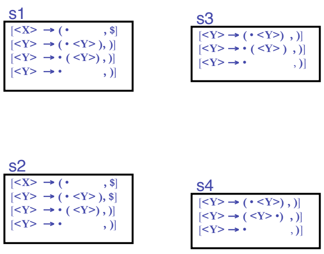
Three options:

- Combine terminals such as <u>number</u> & <u>identifier</u>, <u>+</u> & <u>-</u>, <u>*</u> & <u>/</u>
  - 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

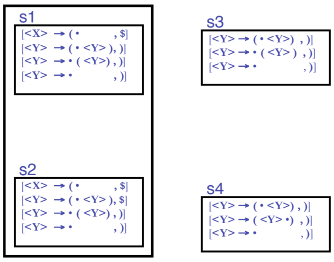
LALR(1) Parser

- Motivation
  - LR(1) Parse Engine has Large Number of States
  - Simple Method to Eliminate States
- If two States are Identical except for the look ahead Symbol of the Items
  - ➨ Merge the States and the corresponding lines

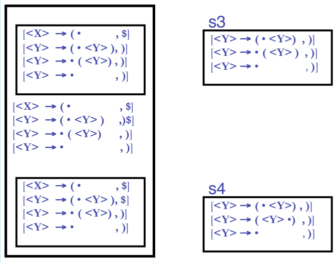
Example of LALR(1)



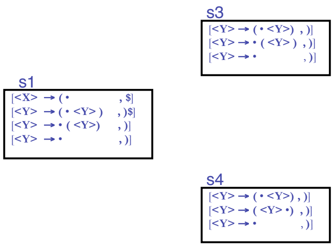
Example of LALR(1)



Example of LALR(1)



Example of LALR(1)



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

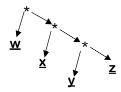
"… *in practice, programming languages do not actually seem to fall in the gap between LL(1) languages and deterministic languages"* *J.J. Horning, "LR Grammars and Analysers", in Compiler Construction, An Advanced Course, Springer-Verlag, 1976*

Parsers in Perspective

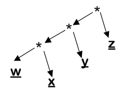
	Advantages	*Disadvantages*
Top-down recursive descent	Fast Good locality Simplicity Good error detection	Hand-coded High maintenance Right associativity
LR(1)	Fast Deterministic langs. Automatable Left associativity	Large working sets Poor error messages Large table sizes

Left Recursion versus Right Recursion

- Right recursion
  - Required for termination in top-down parsers
  - Uses (on average) more stack space
  - Produces right-associative operators
- Left recursion
  - Works fine in bottom-up parsers
  - Limits required stack space
  - Produces left-associative operators
- Rule of thumb
  - Left recursion for bottom-up parsers
  - Right recursion for top-down parsers



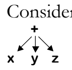
w * (x * (y * z))



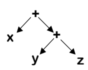
((w * x) * y) * z

Associativity

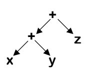
- What difference does it make?
- Can change answers in floating-point arithmetic
- Exposes a different set of common subexpressions
- Consider x+y+z



Ideal operator



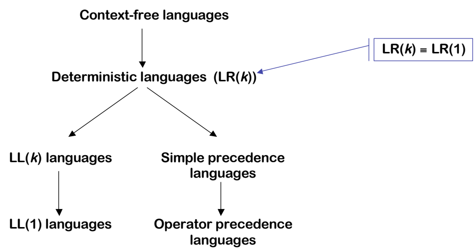
Left association



Right association

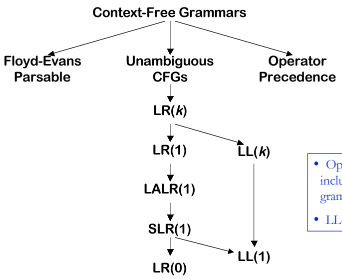
- What if y+z occurs elsewhere? Or x+y? or x+z?
- What if x = 2 & z = 17 ? Neither left nor right exposes 19.
- Best choice is function of surrounding context

Hierarchy of Context-Free Languages



The inclusion hierarchy for context-free languages

Hierarchy of Context-Free Grammars



- Operator precedence includes some ambiguous grammars
- LL(1) is a subset of SLR(1)

The inclusion hierarchy for context-free grammars