RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

CS415 Compilers
Syntax Analysis
Bottom-up Parsing

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University

Review - 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 $A \rightarrow \beta$ 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

LR(1):

[$A \rightarrow \bullet \beta\gamma$, 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 the use of $A \rightarrow \beta\gamma$ 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 βγ, *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, *I*
   a. Begin in an appropriate state, s_0
      - Assume: $S' \rightarrow S$, and S' is unique start symbol that does not occur on any RHS of a production (extended CFG - ECFG)
      - $[S' \rightarrow \bullet S, \underline{EOF}]$, 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 $x \in \text{FIRST}(\delta\underline{a})$

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

  Closure "fills out" a state

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)*
- Should also includes *closure(* [$A \rightarrow \beta X \bullet \delta, \underline{a}$] *)* *(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

Building the Canonical Collection

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

Repeatedly construct new states, until all are found

The algorithm

$cc_0 \leftarrow$ *closure*([S' → •S, EOF])
$CC \leftarrow \{ cc_0 \}$
while (*new sets are still being added to CC*)
  for each unmarked set $cc_j \in CC$
    mark cc_j *as processed*
    for each x following a • *in an item in* cc_j
      $temp \leftarrow goto(cc_j, x)$
      if $temp \notin CC$
        then $CC \leftarrow CC \cup \{ temp \}$
      record transitions from cc_j *to temp on x*

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

Another Example (*grammar & sets*)

Simplified, right recursive expression grammar

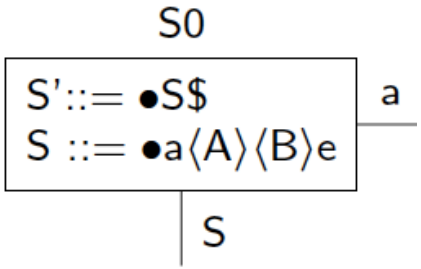
1: *Goal* → *Expr*
2: *Expr* → *Term* - *Expr*
3: *Expr* → *Term*
4: *Term* → *Factor* * *Term*
5: *Term* → *Factor*
6: *Factor* → *ident*

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

One Example

Construct LR(0) States

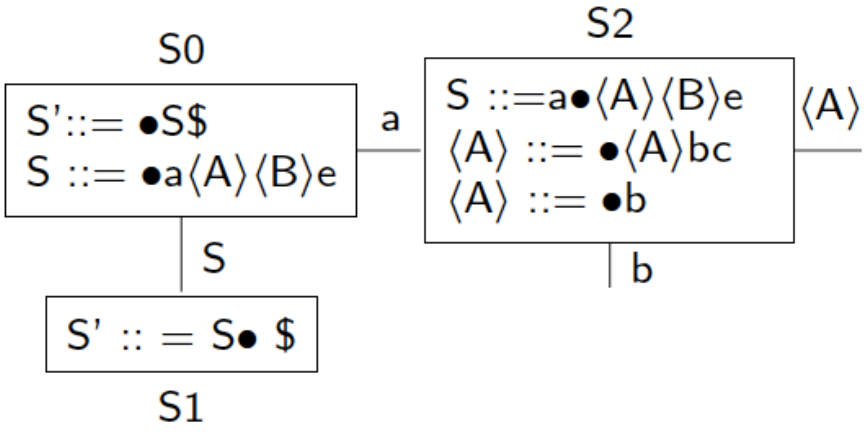
1	⟨S⟩::=a ⟨A⟩ ⟨B⟩ e
2	⟨A⟩::=⟨A⟩ b c
3	⟨A⟩::=b
4	⟨B⟩::=d



One Example

Construct LR(0) States

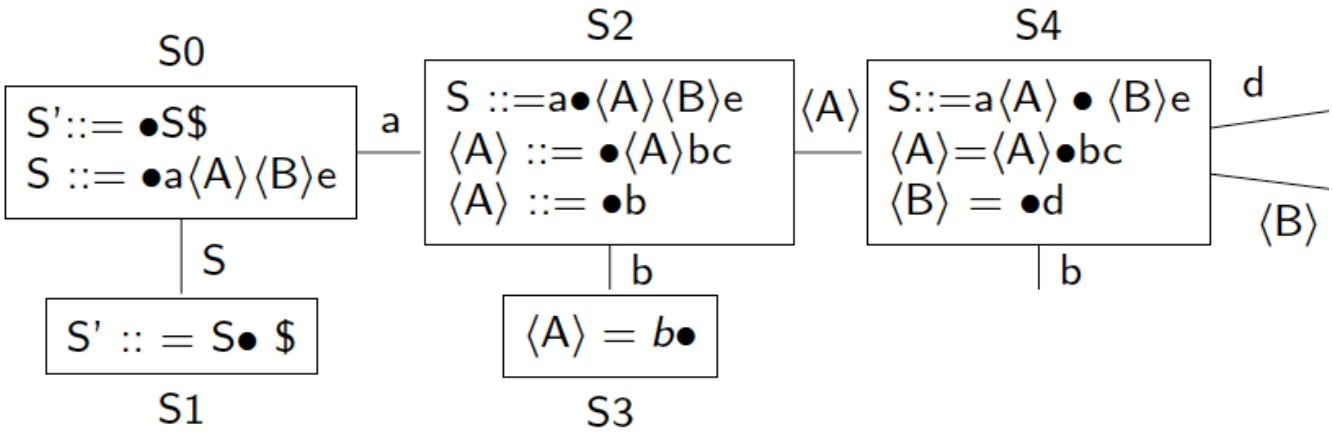
1	⟨S⟩::=a ⟨A⟩ ⟨B⟩ e
2	⟨A⟩::=⟨A⟩ b c
3	⟨A⟩::=b
4	⟨B⟩::=d



One Example

Construct LR(0) States

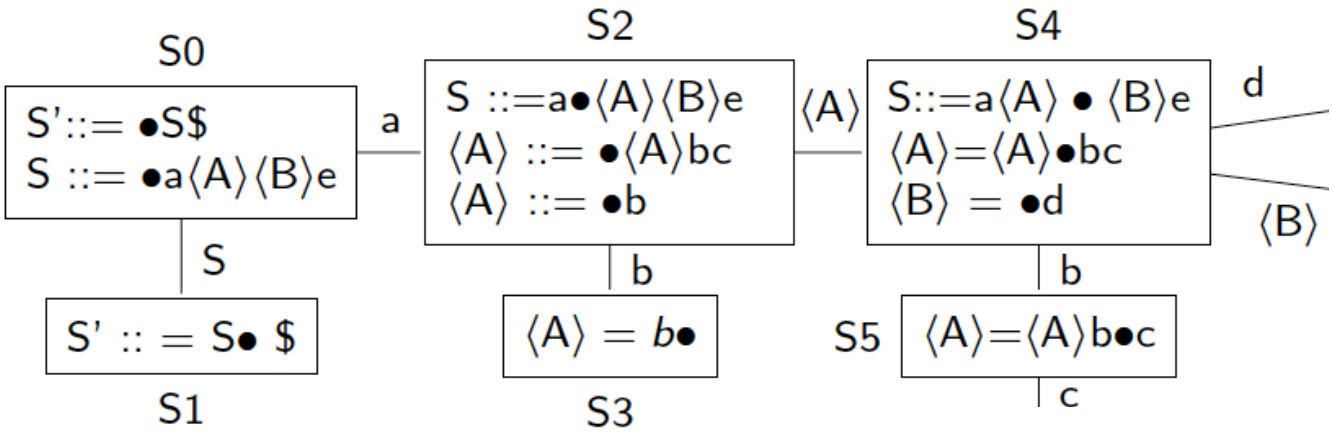
1	⟨S⟩::=a ⟨A⟩ ⟨B⟩ e
2	⟨A⟩::=⟨A⟩ b c
3	⟨A⟩::=b
4	⟨B⟩::=d



One Example

Construct LR(0) States

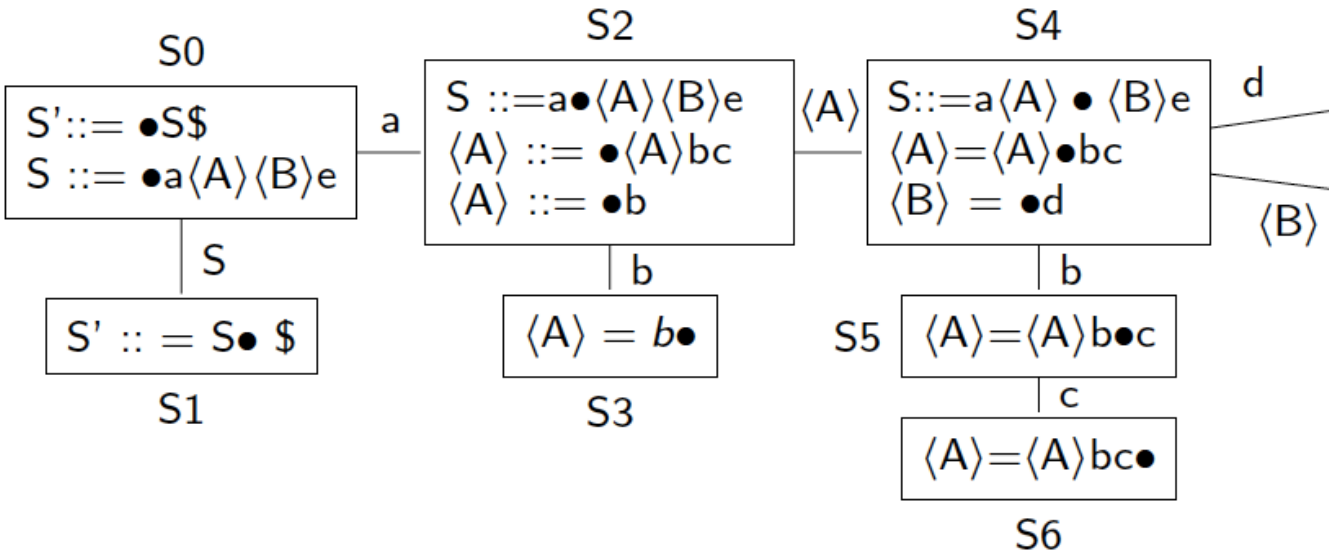
1	⟨S⟩::=a ⟨A⟩ ⟨B⟩ e
2	⟨A⟩::=⟨A⟩ b c
3	⟨A⟩::=b
4	⟨B⟩::=d



One Example

Construct LR(0) States

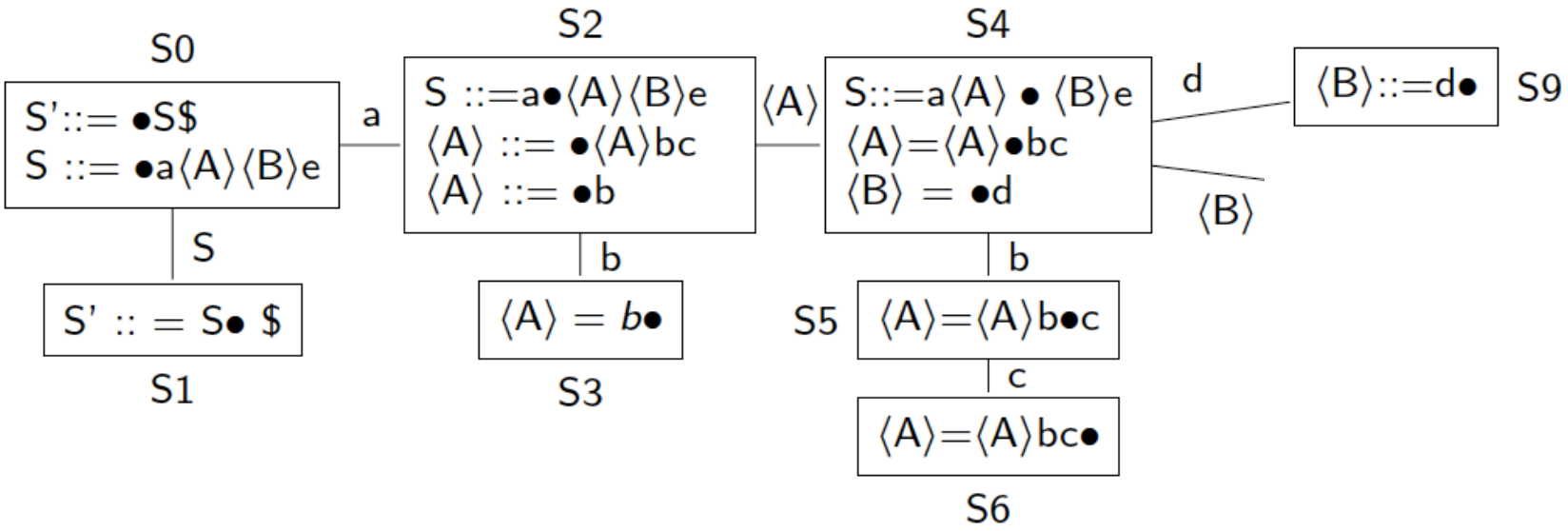
1	⟨S⟩::=a ⟨A⟩ ⟨B⟩ e
2	⟨A⟩::=⟨A⟩ b c
3	⟨A⟩::=b
4	⟨B⟩::=d



One Example

Construct LR(0) States

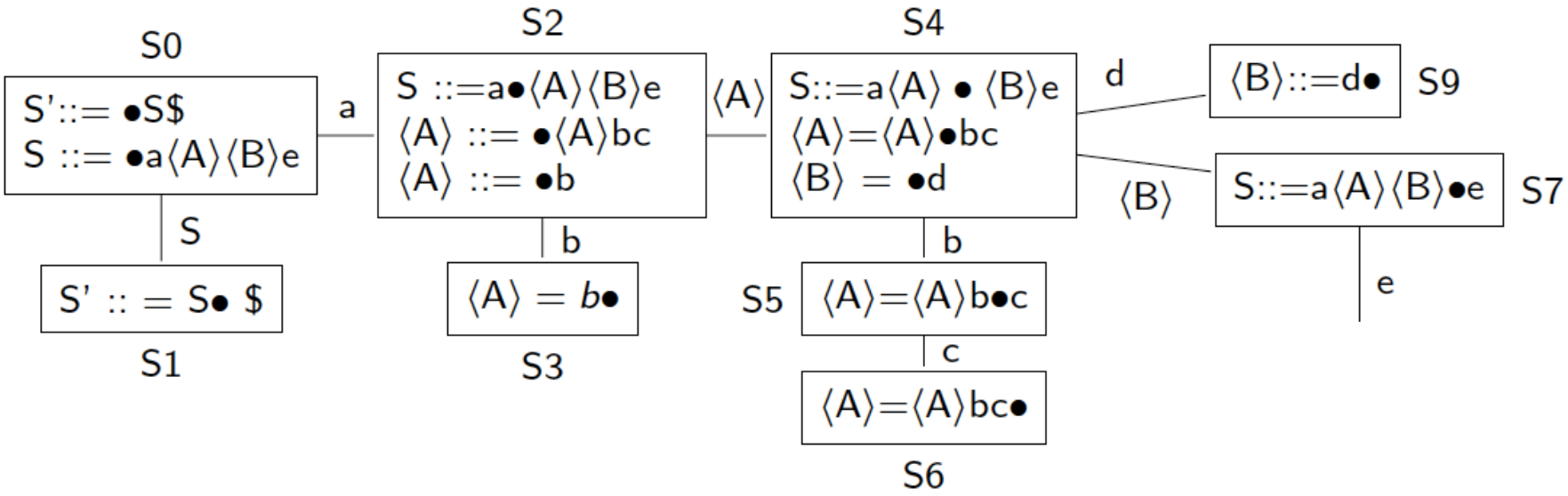
1	⟨S⟩::=a ⟨A⟩ ⟨B⟩ e
2	⟨A⟩::=⟨A⟩ b c
3	⟨A⟩::=b
4	⟨B⟩::=d



One Example

Construct LR(0) States

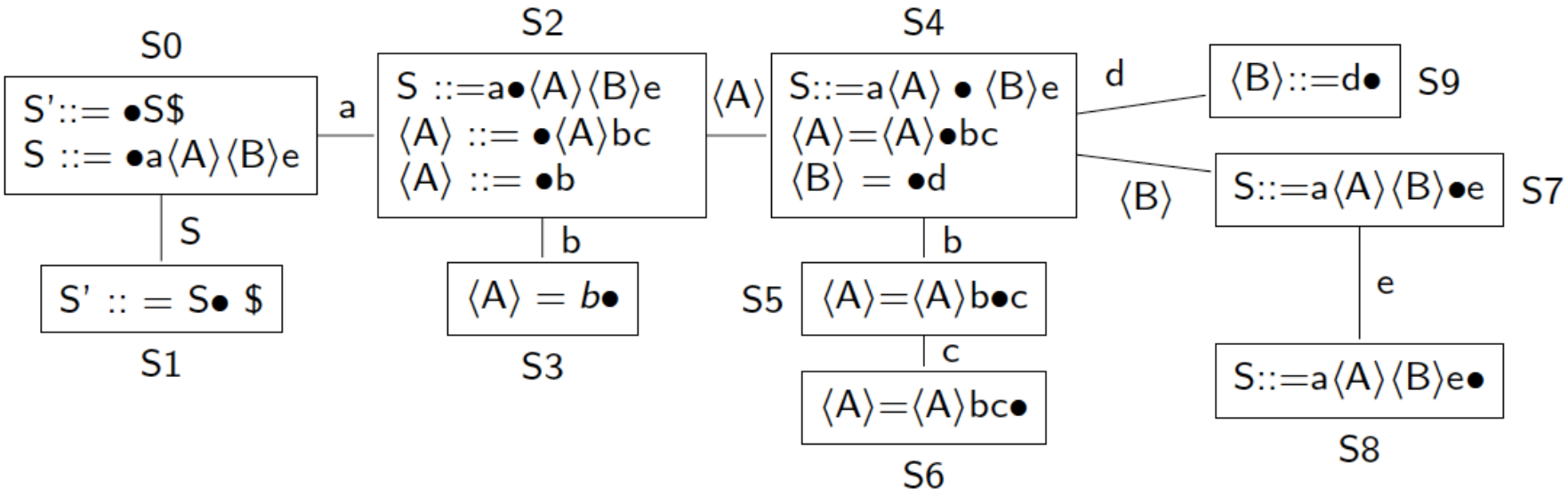
1	⟨S⟩::=a ⟨A⟩ ⟨B⟩ e
2	⟨A⟩::=⟨A⟩ b c
3	⟨A⟩::=b
4	⟨B⟩::=d



One Example – LR(0) states

Construct LR(0) States

1	⟨S⟩::=a ⟨A⟩ ⟨B⟩ e
2	⟨A⟩::=⟨A⟩ b c
3	⟨A⟩::=b
4	⟨B⟩::=d



Another Example (*building the collection*)

1: *Goal* → *Expr*
2: *Expr* → *Term* – *Expr*
3: *Expr* → *Term*
4: *Term* → *Factor* * *Term*
5: *Term* → *Factor*
6: *Factor* → *ident*

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

Initialization Step

$s_0 \leftarrow$ *closure*({ [*Goal* → • *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]}

S ← { S_0 }

Example (*building the collection*)

$s_0 \leftarrow$ *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 , *] }

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)

Example (*building the collection*)

$s_0 \leftarrow closure(\{ [Goal \rightarrow \cdot Expr\ ,\ EOF] \})$

$\{ [Goal \rightarrow \cdot Expr\ ,\ EOF],\ [Expr \rightarrow \cdot Term - Expr\ ,\ EOF],$
$[Expr \rightarrow \cdot Term\ ,\ EOF],\ [Term \rightarrow \cdot Factor * Term\ ,\ EOF],$
$[Term \rightarrow \cdot Factor * Term\ ,\ -],\ [Term \rightarrow \cdot Factor\ ,\ EOF],$
$[Term \rightarrow \cdot Factor\ ,\ -],\ [Factor \rightarrow \cdot \underline{ident}\ ,\ EOF],$
$[Factor \rightarrow \cdot \underline{ident}\ ,\ -],\ [Factor \rightarrow \cdot \underline{ident}\ ,\ *] \}$

Iteration 1

$s_1 \leftarrow goto(s_0\ ,\ Expr) = \{ [Goal \rightarrow Expr \cdot,\ EOF] \}$

$s_2 \leftarrow goto(s_0\ ,\ Term) = \{ [Expr \rightarrow Term \cdot - Expr\ ,\ EOF],\ [Expr \rightarrow Term \cdot,\ EOF] \}$

$s_3 \leftarrow goto(s_0\ ,\ Factor) = \{ [Term \rightarrow Factor \cdot * Term\ ,\ EOF],\ [Term \rightarrow Factor \cdot * Term\ ,\ -],\ [Term \rightarrow Factor \cdot,\ EOF],\ [Term \rightarrow Factor \cdot,\ -] \}$

$s_4 \leftarrow goto(s_0\ ,\ \underline{ident}\ ) = \{ [Factor \rightarrow \underline{ident} \cdot,\ EOF],\ [Factor \rightarrow \underline{ident} \cdot,\ -],\ [Factor \rightarrow \underline{ident} \cdot,\ *] \}$

Example (*building the collection*)

Iteration 1

$s_1 \leftarrow goto(s_0, Expr) = \{ [Goal \rightarrow Expr \bullet, EOF] \}$

$s_2 \leftarrow goto(s_0, Term) = \{ [Expr \rightarrow Term \bullet - Expr, EOF], [Expr \rightarrow Term \bullet, EOF] \}$

$s_3 \leftarrow goto(s_0, Factor) = \{ [Term \rightarrow Factor \bullet * Term, EOF], [Term \rightarrow Factor \bullet * Term, -], [Term \rightarrow Factor \bullet, EOF], [Term \rightarrow Factor \bullet, -] \}$

$s_4 \leftarrow goto(s_0, \underline{ident}) = \{ [Factor \rightarrow \underline{ident} \bullet, EOF], [Factor \rightarrow \underline{ident} \bullet, -], [Factor \rightarrow \underline{ident} \bullet, *] \}$

Iteration 2

$s_5 \leftarrow goto(s_2, -)$

$s_6 \leftarrow goto(s_3, *)$

Example (*building the collection*)

Iteration 1

$s_1 \leftarrow goto(s_0, Expr) = \{ [Goal \rightarrow Expr \bullet, EOF] \}$

$s_2 \leftarrow goto(s_0, Term) = \{ [Expr \rightarrow Term \bullet - Expr, EOF], [Expr \rightarrow Term \bullet, EOF] \}$

$s_3 \leftarrow goto(s_0, Factor) = \{ [Term \rightarrow Factor \bullet * Term, EOF], [Term \rightarrow Factor \bullet * Term, -], [Term \rightarrow Factor \bullet, EOF], [Term \rightarrow Factor \bullet, -] \}$

$s_4 \leftarrow goto(s_0, \underline{ident}) = \{ [Factor \rightarrow \underline{ident} \bullet, EOF], [Factor \rightarrow \underline{ident} \bullet, -], [Factor \rightarrow \underline{ident} \bullet, *] \}$

Iteration 2

$s_5 \leftarrow goto(s_2, -) = \{ [Expr \rightarrow Term - \bullet Expr, EOF], [Expr \rightarrow \bullet Term - Expr, EOF], [Expr \rightarrow \bullet Term, EOF], [Term \rightarrow \bullet Factor * Term, -], [Term \rightarrow \bullet Factor, -], [Term \rightarrow \bullet Factor * Term, EOF], [Term \rightarrow \bullet Factor, EOF], [Factor \rightarrow \bullet \underline{ident}, *], [Factor \rightarrow \bullet \underline{ident}, -], [Factor \rightarrow \bullet \underline{ident}, EOF] \}$

$s_6 \leftarrow goto(s_3, *) =$ *... see next page*

Example (*building the collection*)

Iteration 2

$s_5 \leftarrow goto(s_2, -) =$ { [*Expr* → *Term* - • *Expr*, EOF], [*Expr* → • *Term* - *Expr*, EOF], [*Expr* → • *Term*, EOF], [*Term* → • *Factor* * *Term*, -], [*Term* → • *Factor* * *Term*, EOF], [*Term* → • *Factor*, -], [*Term* → • *Factor*, EOF], [*Factor* → • ident, *], [*Factor* → • ident, -], [*Factor* → • ident, EOF] }

$s_6 \leftarrow goto(s_3, *) =$ { [*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, *] }

Iteration 3

$s_7 \leftarrow goto(s_5, Expr) =$ { [*Expr* → *Term* - *Expr* •, EOF] }

$s_8 \leftarrow goto(s_6, Term) =$ { [*Term* → *Factor* * *Term* •, EOF], [*Term* → *Factor* * *Term* •, -] }

$goto(s_5, Term) = S_2$, $goto(s_5, factor) = s_3$, $goto(S_5, ident) = s_4$

$goto(s_6, Factor) = s_3$, $goto(S_6, ident) = s_4$

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] }

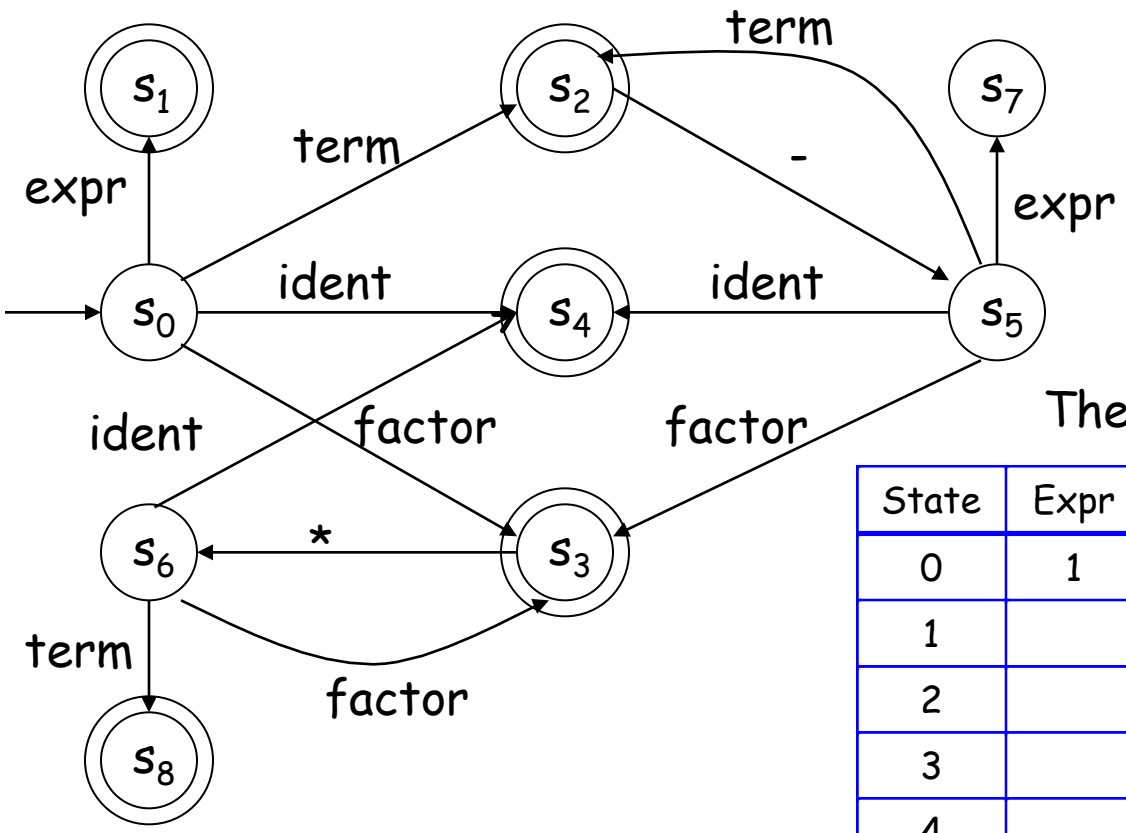
Example (*Summary*)

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

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

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

Example (DFA)



The State Transition Table

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* $s_x \in S$
  ∀ *item* $i \in s_x$
    if i *is* $[A \rightarrow \beta \cdot \underline{a}d, \underline{b}]$ *and* $goto(s_x, \underline{a}) = s_k$, $\underline{a} \in T$
      then ACTION$[x, \underline{a}] \leftarrow$ "*shift k*"
    else if i *is* $[S' \rightarrow S \cdot, EOF]$
      then ACTION$[x, EOF] \leftarrow$ "*accept*"
    else if i *is* $[A \rightarrow \beta \cdot, \underline{a}]$
      then ACTION$[x, \underline{a}] \leftarrow$ "*reduce* $A \rightarrow \beta$"
  ∀ $n \in NT$
    if $goto(s_x, n) = s_k$
      then GOTO$[x, n] \leftarrow k$

Many items generate no table entry

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•a̲γ,b̲] and [*B*→β•,a̲] ?

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

- Each generates "reduce", but with a different production
- Both define ACTION[s,a̲] — 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 (*table compression*)
  - → 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)

Finding Reductions

LR(k) $\Rightarrow$ 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) $\Rightarrow$ Parser must select the next rule based on
- The complete left context
- The next k terminals

Thus, LR(k) examines more context

Summary

	Advantages	*Disadvantages*
Top-down recursive descent	Easy to implement Good locality (fast) Simplicity Easy to embed actions (code access)	Right associativity
LR(1)	Fast Deterministic langs. Automatable (tool support) Left associativity	Large working sets Poor error messages Large table sizes

Left Recursion versus Right Recursion

Right recursion

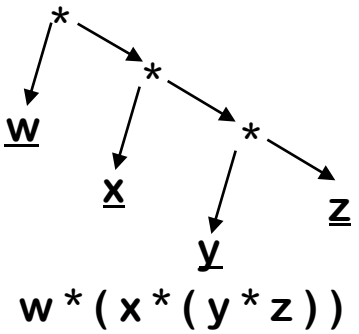
- Required for termination in top-down parsers
- Produces right-associative operators

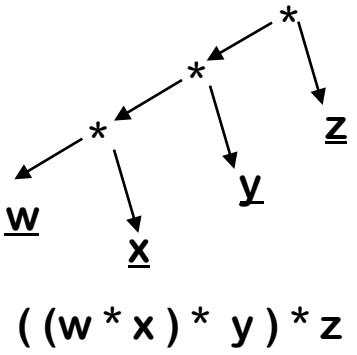
Left recursion

- 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





LR(0) versus SLR(1) versus LR(1)

Example:

$S' \rightarrow S$
$S \rightarrow S\ ;\ a \mid a$

LR(0) ?

LR(1) ?

SLR(1) ?

SLR(1): add FOLLOW(A) to each LR(0) item $[A \rightarrow \gamma \bullet]$ as its second component: $[A \rightarrow \gamma \bullet, \underline{a}]$, $\forall a \in$FOLLOW(A)

LALR(1) versus LR(1)

Example:

S' → S
S → aAd | bBd | aBe | bAe
A → c
B → c

LR(0) ?

LR(1) ?

LALR(1) ?

LALR(1): Merge two sets of LR(1) items (states), if they have the same core.

core of set of LR(1) items: set of LR(0) items derived by ignoring the lookahead symbols

LALR(1) versus LR(1)

Example:

S' → S
S → aAd | bBd | aBe | bAe
A → c
B → c

LR(0) ?

LR(1) ?

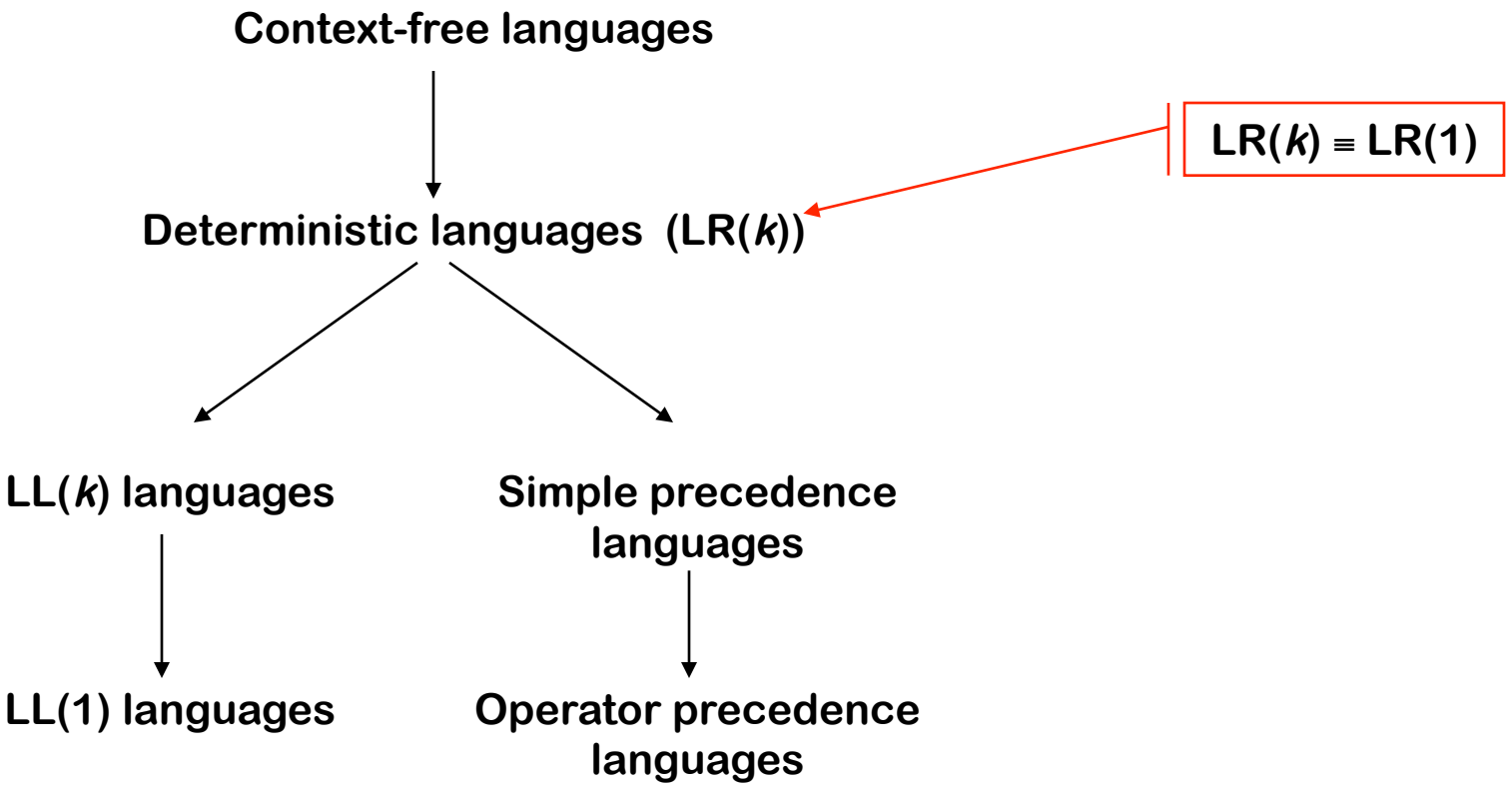
LALR(1) ?

LALR(1): Merge two sets of LR(1) items (states), if they have the same core.

core of set of LR(1) items: set of LR(0) items derived by ignoring the lookahead symbols

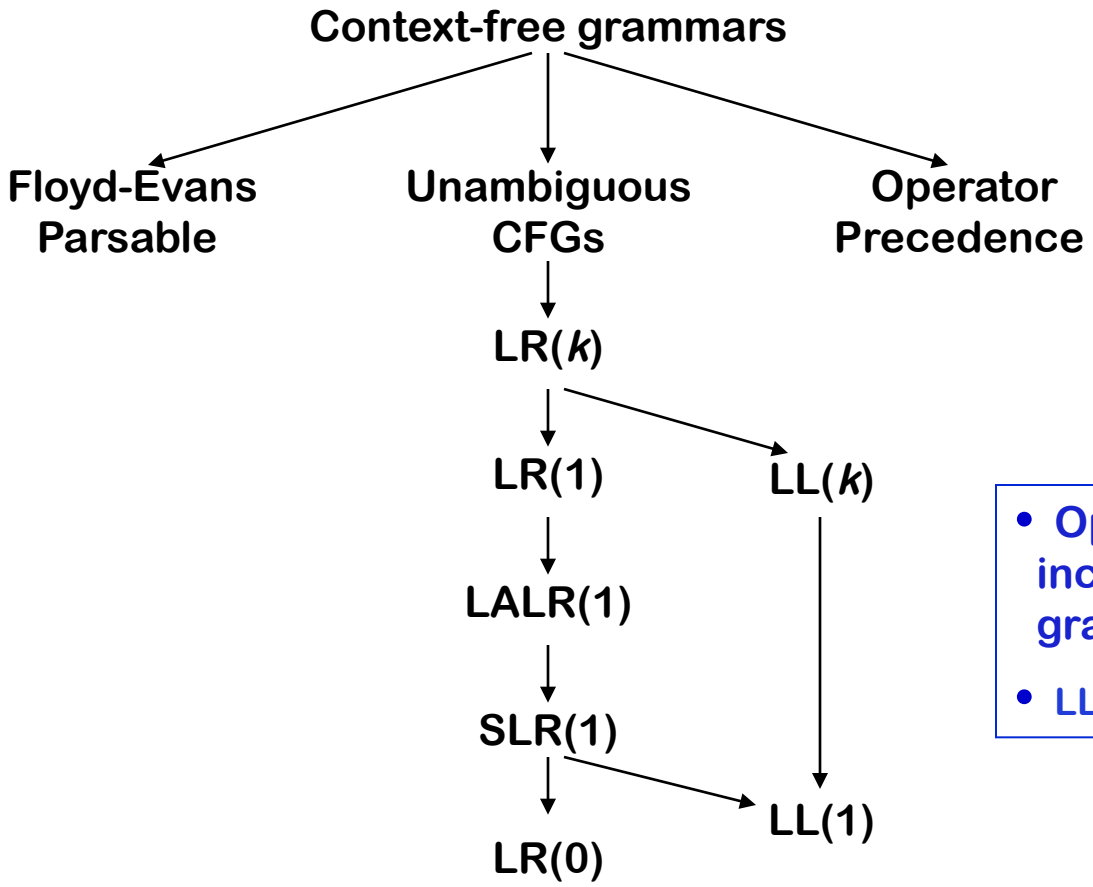
FACT: collapsing LR(1) states into LALR(1) states cannot introcude shift/reduce conflicts

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

Next class

Error Recovery & Context-Sensitive Analysis

Read EaC: Chapters 4.1 – 4.3

Homework 6 will be posted tomorrow.