CMPT 379
Compilers

Anoop Sarkar
`http://www.cs.sfu.ca/~anoop`

Parsing - Roadmap

- Parser:
  - decision procedure: builds a parse tree
- Top-down vs. bottom-up
- LL(1) – Deterministic Parsing
  - recursive-descent
  - table-driven
- LR(k) – Deterministic Parsing
  - LR(0), SLR(1), LR(1), LALR(1)
- Parsing arbitrary CFGs – Polynomial time parsing

Top-Down vs. Bottom Up

Grammar: $S \rightarrow A\ B$
$A \rightarrow c \mid \varepsilon$
$B \rightarrow cbB \mid ca$

Input String: ccbca

Top-Down/leftmost		Bottom-Up/rightmost	
$S \Rightarrow AB$	$S \rightarrow AB$	$ccbca \Leftarrow Acbca$	$A \rightarrow c$
$\Rightarrow cB$	$A \rightarrow c$	$\Leftarrow AcbB$	$B \rightarrow ca$
$\Rightarrow ccbB$	$B \rightarrow cbB$	$\Leftarrow AB$	$B \rightarrow cbB$
$\Rightarrow ccbca$	$B \rightarrow ca$	$\Leftarrow S$	$S \rightarrow AB$

Bottom-up parsing overview

- Start from terminal symbols, search for a path to the start symbol
- Apply shift and reduce actions: postpone decisions
- LR parsing:
  - L: left to right parsing
  - R: rightmost derivation (in reverse or bottom-up)
- LR(0) → SLR(1) → LR(1) → LALR(1)
  - 0 or 1 or *k* lookahead symbols

Actions in Shift-Reduce Parsing

- Shift
  - add terminal to parse stack, advance input
- Reduce
  - If αw on stack, and A→ w, and there is a $\beta \in$ T* such that S $\Rightarrow^*_{rm}$ $\alpha A \beta$ $\Rightarrow_{rm}$ $\alpha w \beta$ then we can *prune the handle* w; we reduce αw to αA on the stack
  - αw is a *viable prefix*
- Error
- Accept

Questions

- When to shift/reduce?
  - What are valid handles?
  - Ambiguity: Shift/reduce conflict
- If reducing, using which production?
  - Ambiguity: Reduce/reduce conflict

Rightmost derivation for **id + id * id**

E → E + E	E ⇒ E * E	
E → E * E	⇒ E * **id**	
E → (E)	⇒ E + E * **id**	
E → - E	⇒ E + **id** * **id**	reduce with E → **id**
E → **id**	⇒ **id** + **id** * **id**	shift

E $\Rightarrow^*_{rm}$ E + E * id

LR Parsing

- Table-based parser
  - Creates rightmost derivation (in reverse)
  - For "less massaged" grammars than LL(1)
- Data structures:
  - Stack of states/symbols {s}
  - Action table: **action**[s, a]; a $\in$ **T**
  - Goto table: **goto**[s, X]; X $\in$ **N**

Productions	
1	**T → F**
2	**T → T*F**
3	**F → id**
4	**F → (T)**

Action/Goto Table

	*	(	)	id	$	T	F
0		S5		S8		2	1
1	R1	R1	R1	R1	R1		
2	S3				Acc!		
3		S5		S8			4
4	R2	R2	R2	R2	R2		
5		S5		S8		6	1
6	S3		S7				
7	R4	R4	R4	R4	R4		
8	R3	R3	R3	R3	R3		

Trace "(id)*id"

Stack	Input	Action
0	**(id) * id $**	**Shift S5**
0 5	**id) * id $**	**Shift S8**
0 5 8	**) * id $**	**Reduce 3 F→id, pop 8, goto [5,F]=1**
0 5 1	**) * id $**	**Reduce 1 T→ F, pop 1, goto [5,T]=6**
0 5 6	**) * id $**	**Shift S7**
0 5 6 7	*** id $**	**Reduce 4 F→ (T), pop 7 6 5, goto [0,F]=1**
0 1	*** id $**	**Reduce 1 T → F pop 1, goto [0,T]=2**

Productions	
1	**T → F**
2	**T → T*F**
3	**F → id**
4	**F → (T)**

	*	(	)	id	$	T	F
0		S5		S8		2	1
1	R1	R1	R1	R1	R1		
2	S3				A		
3		S5		S8			4
4	R2	R2	R2	R2	R2		
5		S5		S8		6	1
6	S3		S7				
7	R4	R4	R4	R4	R4		
8	R3	R3	R3	R3	R3		

"(id)*id"

	Input	A
0	**(id) * id $**	**Sh**
0 5	**id) * id $**	**Sh**
0 5 8	**) * id $**	**Re po**
0 5 1	**) * id $**	**Reduce 1 T→ F, pop 1, goto [5,T]=6**
0 5 6	**) * id $**	**Shift S7**
0 5 6 7	*** id $**	**Reduce 4 F→ (T), pop 7 6 5, goto [0,F]=1**
0 1	*** id $**	**Reduce 1 T → F pop 1, goto [0,T]=2**

Trace "(id)*id"

Stack	Input	Action
0 1	*** id $**	**Reduce 1 T→F, pop 1, goto [0,T]=2**
0 2	*** id $**	**Shift S3**
0 2 3	**id $**	**Shift S8**
0 2 3 8	**$**	**Reduce 3 F→id, pop 8, goto [3,F]=4**
0 2 3 4	**$**	**Reduce 2 T→T * F pop 4 3 2, goto [0,T]=2**
0 2	**$**	**Accept**

Productions	
1	T → F
2	T → T*F
3	F → id
4	F → (T)

“(id)*id”

	*	(	)	id	$	T	F
0		S5		S8		2	1
1	R1	R1	R1	R1	R1		
2	S3				A		
3		S5		S8			4
4	R2	R2	R2	R2	R2		
5		S5		S8		6	1
6	S3		S7				
7	R4	R4	R4	R4	R4		
8	R3	R3	R3	R3	R3		

Stack	Input	Actio
0 1	*** id $**	**Reduc pop 1,**
0 2	*** id $**	**Shift S**
0 2 3	**id $**	**Shift S8**
0 2 3 8	**$**	**Reduce 3 F→id, pop 8, goto [3,F]=4**
0 2 3 4	**$**	**Reduce 2 T→T * F pop 4 3 2, goto [0,T]=2**
0 2	**$**	**Accept**

Tracing LR: **action**[*s*, *a*]

- case **shift** *u*:
  - push state *u*
  - read new *a*
- case **reduce** *r*:
  - lookup production *r*: $X \rightarrow Y_1..Y_k$;
  - pop *k* states, find state *u*
  - push **goto**[*u*, *X*]
- case **accept**: done
- no entry in action table: **error**

Configuration set

- Each set is a parser state
- Consider

$$\mathbf{T \rightarrow T * \bullet F}$$
$$\mathbf{F \rightarrow \bullet (T)}$$
$$\mathbf{F \rightarrow \bullet id}$$

- Like NFA-to-DFA conversion

Closure

Closure property:

- If $\mathbf{T \rightarrow X_1 \ldots X_i \bullet X_{i+1} \ldots X_n}$ is in set, and $\mathbf{X_{i+1}}$ is a nonterminal, then $\mathbf{X_{i+1} \rightarrow \bullet Y_1 \ldots Y_m}$ is in the set as well for all productions $\mathbf{X_{i+1} \rightarrow Y_1 \ldots Y_m}$
- Compute as fixed point

Starting Configuration

- Augment Grammar with S'
- Add production S' → S
- Initial configuration set is

  closure(S' → • S)

Example: I = closure(S' → • T)

S' → • T

T → • T * F

T → • F

F → • id

F → • (T)

S' → T
T → F | T * F
F → id | (T)

Successor(I, X)

Informally: "move by symbol X"

1. move dot to the right in all items where dot is before X
2. remove all other items

   (viable prefixes only!)
3. compute closure

Successor Example

I = {S' → • T,
T → • F,
T → • T * F,
F → • id,
F → • (T) }

S' → T
T → F | T * F
F → id | (T)

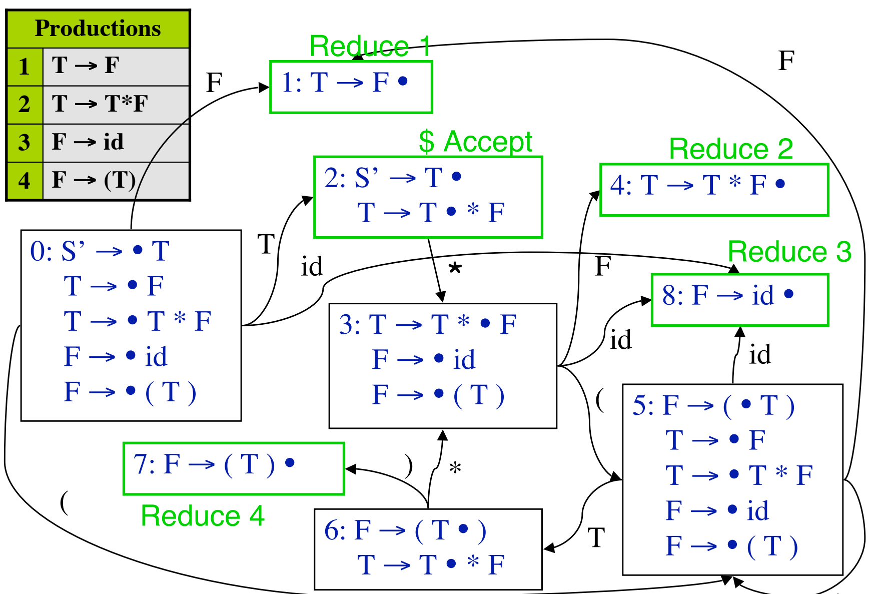
Compute **Successor**(I, "(")

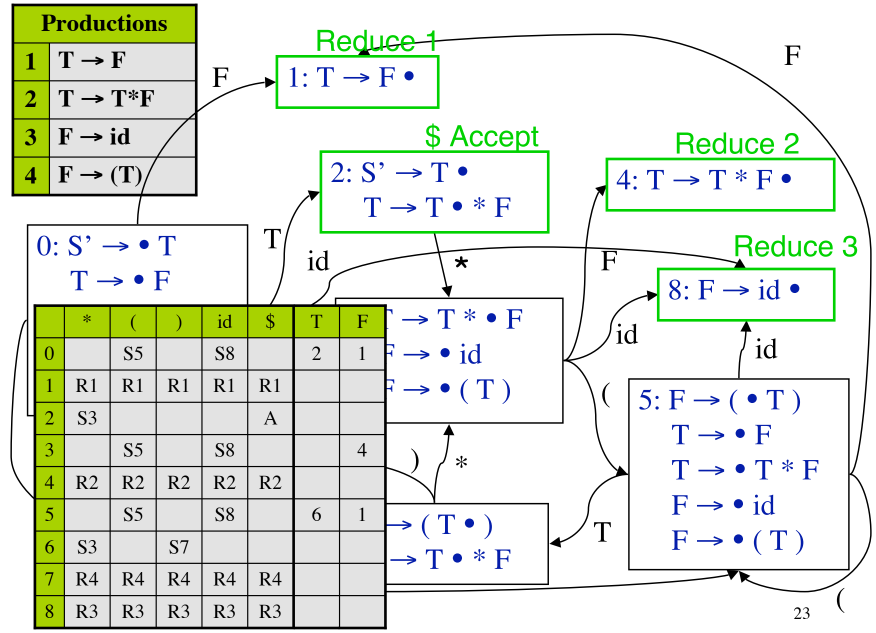
{ F → (• T), T → • F, T → • T * F,
F → • id, F → • (T) }

Sets-of-Items Construction

Family of configuration sets

function items(G’)
    C = { closure({S’ → • S}) };
    do foreach I ∈ C **do**
        foreach X ∈ (N ∪ T) **do**
            C = C ∪ { **Successor**(I, X) };
    while C changes;





	*	(	)	id	$	T	F
0		S5		S8		2	1
1	R1	R1	R1	R1	R1		
2	S3				A		
3		S5		S8			4
4	R2	R2	R2	R2	R2		
5		S5		S8		6	1
6	S3		S7				
7	R4	R4	R4	R4	R4		
8	R3	R3	R3	R3	R3		

LR(0) Construction

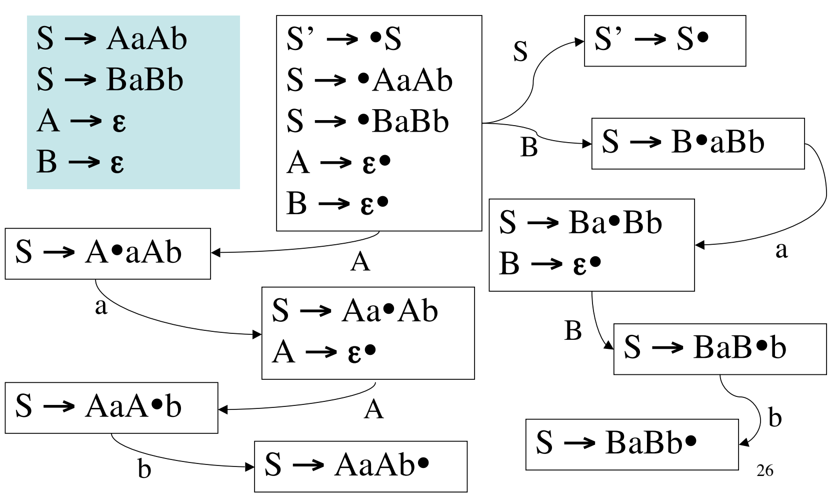
1. Construct F = {I_0, I_1, …I_n}
2. a) if {A → α•} ∈ I_i and A != S
   then action[i, _] := reduce A → α
   b) if {S’ → S•} ∈ I_i
   then action[i,$] := accept
   c) if {A → α•aβ} ∈ I_i and Successor(I_i,a) = I_j
   then action[i,a] := shift j
3. if Successor(I_i,A) = I_j then goto[i,A] := j

LR(0) Construction (cont'd)

4. All entries not defined are errors
5. Make sure I_0 is the initial state

- Note: LR(0) always reduces if $\{A \rightarrow \alpha\bullet\} \in I_i$, no lookahead
- Shift and reduce items can't be in the same configuration set
  - Accepting state doesn't count as reduce item
- At most one reduce item per set

Set-of-items with Epsilon rules



LR(0) conflicts:

S' → F
F → id | (T)
F → id = T ;
T → T * F
T → id

5: F → id •
F → id • = T
Shift/reduce conflict

2: F → id •
T → id •
Reduce/Reduce conflict

Need more lookahead: SLR(1)

SLR(1) : Simple LR(1) Parsing

S' → T
T → F | T * F | C (T)
F → id | id ++ | (T)
C → id

0: S' → • T
T → • F
T → • T * F
T → • C (T)
F → • id
F → • id ++
F → • (T)
C → • id

id

1: F → id •
F → id • ++
C → id •

Follow(F) = { *,), $ }

Follow(C) = { (}

action[1,*]= action[1,)] = action[1,$] = Reduce F → id
action[1,(] = Reduce C → id
action[1,++] = Shift

SLR(1) Construction

1. Construct $F = \{I_0, I_1, \ldots I_n\}$
2. a) if $\{A \rightarrow \alpha\bullet\} \in I_i$ and $A \mathrel{!}= S'$
   then action[i, b] := reduce $A \rightarrow \alpha$
   for all $b \in \mathrm{Follow}(A)$

   b) if $\{S' \rightarrow S\bullet\} \in I_i$
   then action[i, \$] := accept

   c) if $\{A \rightarrow \alpha\bullet a\beta\} \in I_i$ and $\mathrm{Successor}(I_i, a) = I_j$
   then action[i, a] := shift j
3. if $\mathrm{Successor}(I_i, A) = I_j$ then goto[i, A] := j

SLR(1) Construction (cont'd)

4. All entries not defined are errors
5. Make sure I_0 is the initial state

- Note: SLR(1) only reduces $\{A \rightarrow \alpha\bullet\}$ if lookahead in Follow(A)
- Shift and reduce items or more than one reduce item can be in the same configuration set as long as lookaheads are disjoint

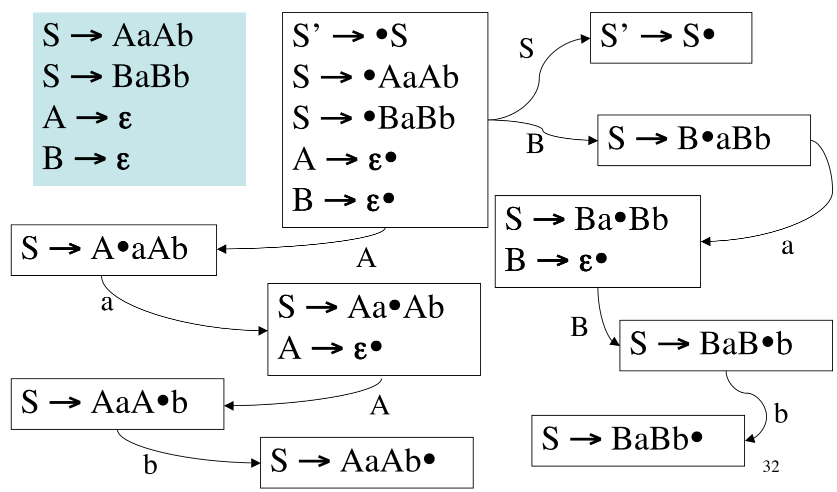
SLR(1) Conditions

- A grammar is SLR(1) if for each configuration set:
  - For any item $\{A \rightarrow \alpha\bullet x\beta: x \in T\}$ there is no $\{B \rightarrow \gamma\bullet: x \in \mathrm{Follow}(B)\}$
  - For any two items $\{A \rightarrow \alpha\bullet\}$ and $\{B \rightarrow \beta\bullet\}$ $\mathrm{Follow}(A) \cap \mathrm{Follow}(B) = \varnothing$

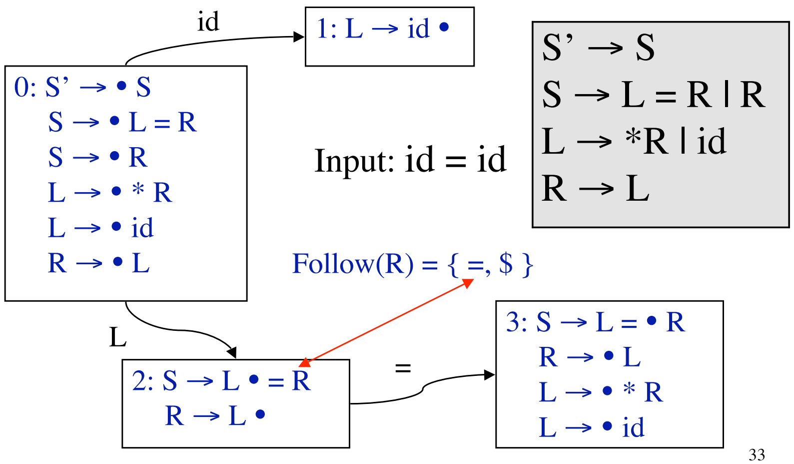
LR(0) Grammars $\subset$ SLR(1) Grammars

Is this grammar SLR(1)?

$S \rightarrow AaAb$
$S \rightarrow BaBb$
$A \rightarrow \varepsilon$
$B \rightarrow \varepsilon$



SLR limitation: lack of context



Solution: Canonical LR(1)

- Extend definition of configuration
  - Remember lookahead
- New closure method
- Extend definition of Successor

LR(1) Configurations

- [A → α•β, a] for a ∈ T is valid for a viable prefix δα if there is a rightmost derivation
  $S \Rightarrow^* \delta A \eta \Rightarrow^* \delta\alpha\beta\eta$ and
  ($\eta = a\gamma$) or ($\eta = \varepsilon$ and $a = \$$)
- Notation: [A → α•β, a/b/c]
  - if [A → α•β, a], [A → α•β, b], [A → α•β, c] are valid configurations

LR(1) Configurations

S → B B
B → a B | b

- $S \Rightarrow^*_{rm} aaBab \Rightarrow_{rm} aaaBab$
- Item [B → a • B, a] is valid for viable prefix *aaa*
- $S \Rightarrow^*_{rm} BaB \Rightarrow_{rm} BaaB$
- Also, item [B → a • B, $] is valid for viable prefix *Baa*

LR(1) Closure

Closure property:

- If $[A \rightarrow \alpha \bullet B\beta, a]$ is in set, then $[B \rightarrow \bullet \gamma, b]$ is in set if $b \in First(\beta a)$
- Compute as fixed point
- Only include contextually valid lookaheads to guide reducing to B

Starting Configuration

- Augment Grammar with S' just like for LR(0), SLR(1)
- Initial configuration set is

$$I = closure([S' \rightarrow \bullet S, \$])$$

Example: closure([S' → • S, $])

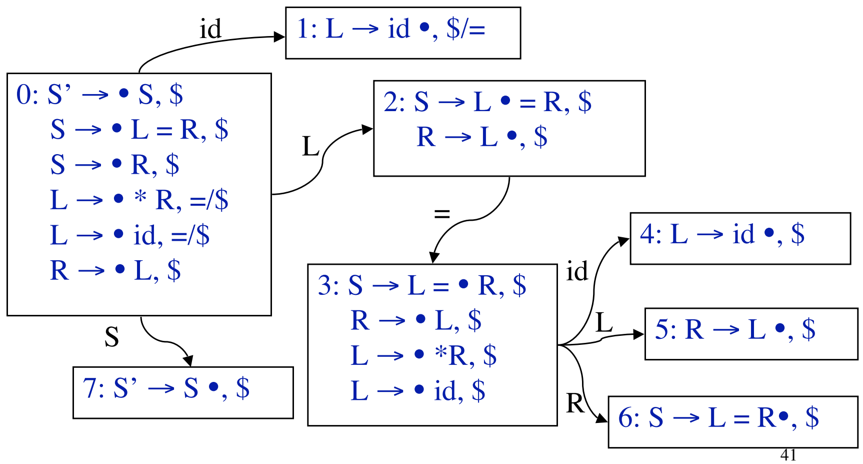
$[S' \rightarrow \bullet S, \$]$
$[S \rightarrow \bullet L = R, \$]$
$[S \rightarrow \bullet R, \$]$
$[L \rightarrow \bullet * R, =]$
$[L \rightarrow \bullet id, =]$
$[R \rightarrow \bullet L, \$]$
$[L \rightarrow \bullet *R, \$]$
$[L \rightarrow \bullet id, \$]$

$S' \rightarrow S$
$S \rightarrow L = R \mid R$
$L \rightarrow *R \mid id$
$R \rightarrow L$

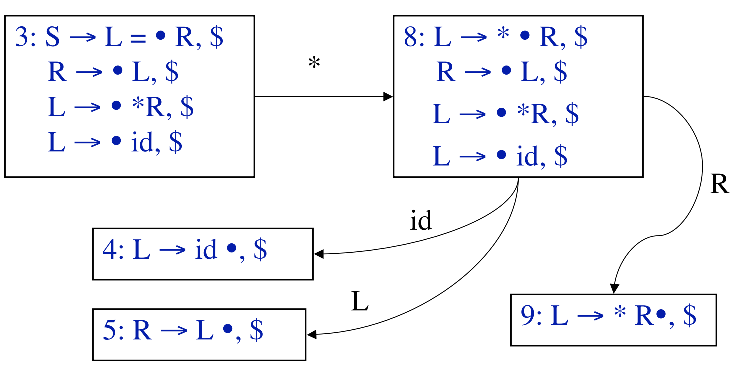
LR(1) Successor(C, X)

- Let $I = [A \rightarrow \alpha \bullet B\beta, a]$ **or** $[A \rightarrow \alpha \bullet b\beta, a]$
- Successor(I, B)
  $= closure([A \rightarrow \alpha B \bullet \beta, a])$
- Successor(I, b)
  $= closure([A \rightarrow \alpha b \bullet \beta, a])$

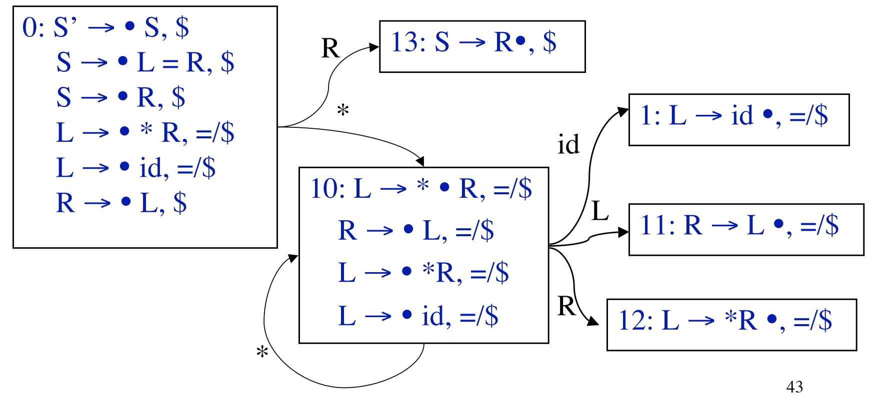
LR(1) Example



LR(1) Example (contd)



LR(1) Example (contd)



Productions	
1	**S → L = R**
2	**S → R**
3	**L → * R**
4	**L → id**
5	**R → L**

	id	=	*	$	S	L	R
0	S1		S10		7	2	13
1		R4		R4			
2		S3		R5			
3	S4		S8			5	6
4				R4			
5				R5			
6				R1			
7				Acc			
8	S4					5	9
9				R3			
10	S1		S10			11	12
11		R5		R5			
12		R3		R3			
13				R2			

LR(1) Construction

1. Construct $F = \{I_0, I_1, \ldots I_n\}$
2. a) if $[A \rightarrow \alpha\bullet, a] \in I_i$ and $A \mathrel{!}= S'$
   then action[i, a] := reduce $A \rightarrow \alpha$
   b) if $[S' \rightarrow S\bullet, \$] \in I_i$
   then action[i, \$] := accept
   c) if $[A \rightarrow \alpha\bullet a\beta, b] \in I_i$ and $\text{Successor}(I_i, a)=I_j$
   then action[i, a] := shift j
3. if $\text{Successor}(I_i, A) = I_j$ then goto[i, A] := j

LR(1) Construction (cont'd)

4. All entries not defined are errors
5. Make sure I_0 is the initial state

- Note: LR(1) only reduces using $A \rightarrow \alpha$ for $[A \rightarrow \alpha\bullet, a]$ if a follows
- LR(1) states remember context by virtue of lookahead
- Possibly many states!
  – LALR(1) combines some states

LR(1) Conditions

- A grammar is LR(1) if for each configuration set holds:
  – For any item $[A \rightarrow \alpha\bullet x\beta, a]$ with $x \in T$ there is no $[B \rightarrow \gamma\bullet, x]$
  – For any two complete items $[A \rightarrow \gamma\bullet, a]$ and $[B \rightarrow \beta\bullet, b]$ it follows a and a != b.
- Grammars:
  – $LR(0) \subset SLR(1) \subset LR(1) \subset LR(k)$
- Languages expressible by grammars:
  – $LR(0) \subset SLR(1) \subset LR(1) = LR(k)$

Canonical LR(1) Recap

- LR(1) uses left context, current handle and lookahead to decide when to reduce or shift
- Most powerful parser so far
- LALR(1) is practical simplification with fewer states

Merging States in LALR(1)

- S’ → S
  S → XX
  X → aX
  X → b
- Same **Core Set**
- Different lookaheads

3: X → a • X, a/b
X → • a X, a/b
X → • b, a/b

6: X → a • X, $
X → • a X, $
X → • b, $

36: X → a • X, a/b/$
X → • a X, a/b/$
X → • b, a/b/$

R/R conflicts when merging

- B → d
  B → f X g
  X → …
- If R/R conflicts are introduced, grammar is not LALR(1)!

2: B → d •, c
B → f X g •, e

4: B → d •, g
B → f X g •, c

24: B → d •, c/g
B → f X g •, c/e

LALR(1)

- LALR(1) Condition:
  - Merging in this way does not introduce reduce/reduce conflicts
  - Shift/reduce can’t be introduced
- Merging brute force or step-by-step
- More compact than canonical LR, like SLR(1)
- More powerful than SLR(1)
  - Not always merge to full Follow Set

S/R & ambiguous grammars

- Lx(k) Grammar vs. Language
  - Grammar is Lx(k) if it can be parsed by Lx(k) method – according to criteria that is specific to the method.
  - A Lx(k) grammar may or may not exist for a language.
- Even if a given grammar is not LR(k), shift/reduce parser can *sometimes* handle them by accounting for ambiguities
  - Example: ‘dangling’ else
    - Preferring shift to reduce means matching inner ‘if’

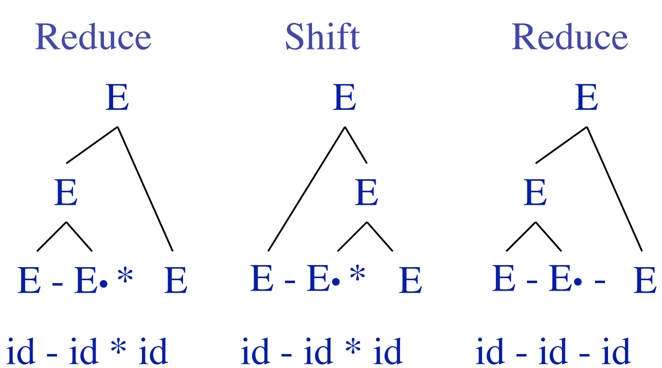
Dangling 'else'

1. S → if E then S
2. S → if E then S else S

- Viable prefix "if E then if E then S"
  - Then read else
- Shift "else" (means go for 2)
- Reduce (reduce using production #1)
- NB: dangling else as written above is ambiguous
  - NB: Ambiguity can be resolved, but there's still no LR(k) grammar

Precedence & Associativity

- Consider $E \rightarrow E - E \mid E * E \mid id$



Precedence Relations

- Let $A \rightarrow w$ be a rule in the grammar
- And b is a terminal
- In some state q of the LR(1) parser there is a shift-reduce conflict:
  - either reduce with $A \rightarrow w$ or shift on b
- Write down a rule, either:

  $A \rightarrow w, < b$ or $A \rightarrow w, > b$

Precedence Relations

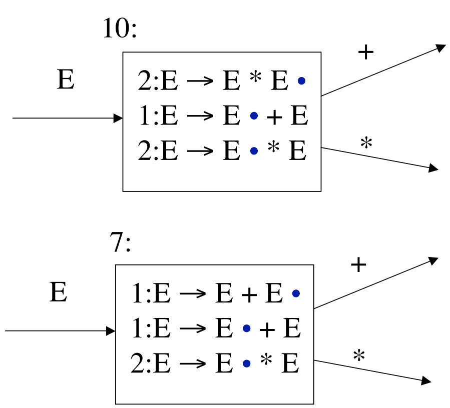
- $A \rightarrow w, < b$ means rule has less precedence and so we shift if we see b in the lookahead
- $A \rightarrow w, > b$ means rule has higher precedence and so we reduce if we see b in the lookahead
- If there are multiple terminals with shift-reduce conflicts, then we list them all:

  $A \rightarrow w, > b, < c, > d$

Precedence Relations

- Consider the grammar
  E → E + E | E * E | (E) | a
- Assume left-association so that E+E+E is interpreted as (E+E)+E
- Assume multiplication has higher precedence than addition
- Then we can write precedence rules/relns:
  E → E + E, > +, < *
  E → E * E, > +, > *

Precedence & Associativity



E → E + E, > +, < *
E → E * E, > +, > *

	+	*
7	R1	Shift
10	R2	R2

Handling S/R & R/R Conflicts

- Have a conflict?
  - No? – Done, grammar is compliant.
- Already using most powerful parser available?
  - No? – Upgrade and goto 1
- Can the grammar be rearranged so that the conflict disappears?
  - While preserving the language!

Conflicts revisited (cont'd)

- Can the grammar be rearranged so that the conflict disappears?
  - No?
    - Is the conflict S/R and does shift-to-reduce preference yield desired result?
      - Yes: Done. (Example: dangling else)
    - Else: Bad luck
  - Yes: Is it worth it?
    - Yes, resolve conflict.
    - No: live with default or specified conflict resolution (precedence, associativity)

Compiler (parser) compilers

- Rather than build a parser for a particular grammar (e.g. recursive descent), write down a grammar as a text file
- Run through a compiler compiler which produces a parser for that grammar
- The parser is a program that can be compiled and accepts input strings and produces user-defined output

Compiler (parser) compilers

- For LR parsing, all it needs to do is produce action/goto table
  - Yacc (yet another compiler compiler) was distributed with Unix, the most popular tool. Uses LALR(1).
  - Many variants of yacc exist for many languages
- As we will see later, translation of the parse tree into machine code (or anything else) can also be written down with the grammar
- Handling errors and interaction with the lexical analyzer have to be precisely defined

Parsing CFGs

- Consider the problem of parsing with arbitrary CFGs
- For any input string, the parser has to produce a parse tree
- The simpler problem: print **yes** if the input string is generated by the grammar, print **no** otherwise
- This problem is called *recognition*

Parsing - Summary

- Top-down vs. bottom-up
- Lookahead: FIRST and FOLLOW sets
- LL(1) – Parsing: $O(n)$ time complexity
  - recursive-descent and table-driven predictive parsing
- LR(k) – Parsing : $O(n)$ time complexity
  - LR(0), SLR(1), LR(1), LALR(1)
- Resolving shift/reduce conflicts
  - using precedence, associativity