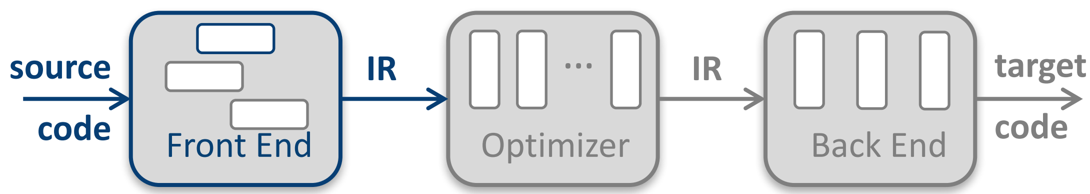
RICE

COMP 412
FALL 2017

Syntax Analysis, VII

One more LR(1) example, plus some more stuff

Comp 412



Chapter 3 in EaC2e

Computing Closures

Review

Closure(s) adds all the ***possibilities*** for the items already in s

- Any item $[A \rightarrow \beta \bullet B\delta, \underline{a}]$ where $B \in NT$ implies $[B \rightarrow \bullet \tau, x]$ for each production that has 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*)
    $\forall$ *items* $[A \rightarrow \beta \bullet B\delta, \underline{a}] \in s$
      lookahead $\leftarrow$ FIRST($\delta\underline{a}$) // δ might be ε
      $\forall$ *productions* $B \rightarrow \tau \in P$
        $\forall$ $\underline{b} \in$ *lookahead*
          if $[B \rightarrow \bullet \tau, \underline{b}] \notin s$
            then $s \leftarrow s \cup \{ [B \rightarrow \bullet \tau, \underline{b}] \}$

- Classic fixed-point method
- Halts because $s \subset I$, the set of all items (*finite*)
- Worklist version is faster
- ***Closure*** *"fills out"* a state s

Generate new lookaheads.
See note on p. 128

Computing Gotos

Review

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

- ***Goto***({ [$A \rightarrow \beta \bullet X\delta$,a] }, X) produces { [$A \rightarrow \beta X \bullet \delta$,a] } ***(obviously)***
- It finds all such items & uses *Closure*() to fill out the state

The Algorithm

Goto(*s*, *X*)
  new ← Ø
  ∀ *items* [$A \rightarrow \beta \bullet X\delta$,a] ∈ *s*
    new ← *new* ∪ { [$A \rightarrow \beta X \bullet \delta$,a] }
  return ***Closure***(*new*)

- ***Goto***( ) models a transition in the automaton
- Straightforward computation
- ***Goto()*** is not a fixed-point method (but it calls ***Closure()***)

***Goto* in this construction is analogous to *Move* in the subset construction.**

Building the Canonical Collection

Review

Start from s_0 = ***Closure***($[S' \rightarrow \bullet S, \underline{\textbf{EOF}}]$)

Repeatedly construct new states, until all are found

The Algorithm

$s_0 \leftarrow$ ***Closure*** ($\{ [S' \rightarrow \bullet S, \underline{\textbf{EOF}}] \}$)
$S \leftarrow \{ s_0 \}$
$k \leftarrow 1$

while (*S is still changing*)

  $\forall s_j \in S$ *and* $\forall x \in (T \cup NT)$
    $s_k \leftarrow$ ***Goto***(s_j, x)
    record $s_j \rightarrow s_k$ *on* x

  if $s_k \notin S$ *then*
    $S \leftarrow S \cup \{ s_k \}$
    $k \leftarrow k + 1$

- Fixed-point computation
- Loop adds to S (*monotone*)
- $S \subseteq 2^{\textbf{ITEMS}}$, so S is finite
- *Worklist version is faster because it avoids duplicated effort*

This membership / equality test requires careful and/or clever implementation.

Filling in the ACTION and GOTO Tables **Review**

The Table Construction Algorithm

x is the state number

$\forall$ *set* $S_x \in S$
  $\forall$ *item* $i \in S_x$
    if i *is* $[A \rightarrow \beta \bullet \underline{a}\delta, \underline{b}]$ *and* $goto(S_x, \underline{a}) = S_k$, $\underline{a} \in T$
      then ACTION[x,$\underline{a}$] $\leftarrow$ "*shift k*"
    else if i *is* $[S' \rightarrow S \bullet, \underline{\text{EOF}}]$
      then ACTION[x, $\underline{\text{EOF}}$] $\leftarrow$ "*accept*"
    else if i *is* $[A \rightarrow \beta \bullet, \underline{a}]$
      then ACTION[x,$\underline{a}$] $\leftarrow$ "*reduce* $A \rightarrow \beta$"
  $\forall$ $n \in NT$
    if $goto(S_x, n) = S_k$
      then GOTO[x,n] $\leftarrow$ k

- before $T \Rightarrow$ *shift*

have *Goal* $\Rightarrow$ *accept*

- at end $\Rightarrow$ *reduce*

Many items generate no table entry

→ Placeholder before a *NT* does not generate an **ACTION** table entry

→ ***Closure***( ) instantiates FIRST(X) directly for $[A \rightarrow \beta \bullet X\delta, \underline{a}]$

Another Example (*grammar & sets*)

Simplified, right recursive expression grammar

0	*Goal*	→	*Expr*
1	*Expr*	→	*Term* - *Expr*
2		|	*Term*
3	*Term*	→	*Factor* * *Term*
4		|	*Factor*
5	*Factor*	→	id

SYMBOL	FIRST
Goal	{ id }
Expr	{ id }
Term	{ id }
Factor	{ id }
–	{ – }
*	{ * }
id	{ id }

Simplified Expression Grammar

Initialization Step

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

{ [*Goal* $\rightarrow$ • *Expr* , EOF],

[*Expr* $\rightarrow$ • *Term* – *Expr* , EOF], [*Expr* $\rightarrow$ • *Term* , EOF],

[*Term* $\rightarrow$ • *Factor* * Term , EOF], [*Term* $\rightarrow$ • *Factor* * *Term* , –],

[*Term* $\rightarrow$ • *Factor* , EOF], [*Term* $\rightarrow$ • *Factor* , –],

[*Factor* $\rightarrow$ • id , EOF], [*Factor* $\rightarrow$ • id , –], [*Factor* $\rightarrow$ • id , *] }

$S \leftarrow \{s_0\}$

Item in ***black*** is the initial item.
Items in *gray* are added by *closure*().

Simplified Expression Grammar

Iteration 1

$s_1 \leftarrow \textbf{goto}(s_0, Expr)$

$s_2 \leftarrow \textbf{goto}(s_0, Term)$

$s_3 \leftarrow \textbf{goto}(s_0, Factor)$

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

Goal, * , & - generate empty sets

Iteration 2

$s_5 \leftarrow \textbf{goto}(s_2, -)$

$s_6 \leftarrow \textbf{goto}(s_3, *)$

Goal, Expr, Term, Factor, & id generate empty sets

Iteration 3

$s_7 \leftarrow \textbf{goto}(s_5, Expr)$

$s_8 \leftarrow \textbf{goto}(s_6, Term)$

Goal, *, & - generate empty sets. Term, Factor, & id start to re-create existing sets.

Simplified Expression Grammar

The Details

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

{ [*Goal* $\rightarrow$ • *Expr* , EOF],

[*Expr* $\rightarrow$ • *Term* – *Expr* , EOF], [*Expr* $\rightarrow$ • *Term* , EOF],

[*Term* $\rightarrow$ • *Factor* * *Term* , EOF], [*Term* $\rightarrow$ • *Factor* * *Term* , –],

[*Term* $\rightarrow$ • *Factor* , EOF], [*Term* $\rightarrow$ • *Factor* , –],

[*Factor* $\rightarrow$ • <u>id</u> , EOF], [*Factor* $\rightarrow$ • <u>id</u> , –], [*Factor* $\rightarrow$ • <u>id</u>, *] }

$s_1 \leftarrow$ ***goto***(s_0 , *Expr*)

{ [*Goal* $\rightarrow$ *Expr* •, EOF] }

$s_2 \leftarrow$ ***goto***(s_0 , *Term*)

{ [*Expr* $\rightarrow$ *Term* • – *Expr* , EOF], [*Expr* $\rightarrow$ *Term* •, EOF] }

$s_3 \leftarrow$ ***goto***(s_0 , *Factor*)

{ [*Term* $\rightarrow$ *Factor* • * *Term* , EOF],[*Term* $\rightarrow$ *Factor* • * *Term* , –],

[*Term* $\rightarrow$ *Factor* •, EOF], [*Term* $\rightarrow$ *Factor* •, –] }

Items in *black* are core items, generated by moving the placeholder.
Items in *gray* are added by *closure*().

Simplified Expression Grammar

The Details

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

$\{\ [Factor \rightarrow \underline{\text{id}}\ \bullet, \text{EOF}], [Factor \rightarrow \underline{\text{id}}\ \bullet, -], [Factor \rightarrow \underline{\text{id}}\ \bullet, *]\ \}$

$s_5 \leftarrow \textbf{goto}(s_2, -)$

$\{\ [Expr \rightarrow Term - \bullet\ Expr, \text{EOF}], [Expr \rightarrow \bullet\ Term - Expr, \text{EOF}],$
$[Expr \rightarrow \bullet\ Term, \text{EOF}],$
$[Term \rightarrow \bullet\ Factor * Term, -], [Term \rightarrow \bullet\ Factor, -],$
$[Term \rightarrow \bullet\ Factor * Term, \text{EOF}], [Term \rightarrow \bullet\ Factor, \text{EOF}],$
$[Factor \rightarrow \bullet\ \underline{\text{id}}, *], [Factor \rightarrow \bullet\ \underline{\text{id}}, -], [Factor \rightarrow \bullet\ \underline{\text{id}}, \text{EOF}]\ \}$

$s_6 \leftarrow \textbf{goto}(s_3, *)$

$\{\ [Term \rightarrow Factor * \bullet\ Term, \text{EOF}], [Term \rightarrow Factor * \bullet\ Term, -],$
$[Term \rightarrow \bullet\ Factor * Term, \text{EOF}], [Term \rightarrow \bullet\ Factor * Term, -],$
$[Term \rightarrow \bullet\ Factor, \text{EOF}], [Term \rightarrow \bullet\ Factor, -],$
$[Factor \rightarrow \bullet\ \underline{\text{id}}, \text{EOF}], [Factor \rightarrow \bullet\ \underline{\text{id}}, -], [Factor \rightarrow \bullet\ \underline{\text{id}}, *]\ \}$

Items in *black* are core items, generated by moving the placeholder.
Items in *gray* are added by *closure*().

Simplified Expression Grammar

The Details

$s_7 \leftarrow$ ***goto***$(s_5, Expr)$

$\{ [Expr \rightarrow Term - Expr \bullet, \text{EOF}] \}$

goto$(s_5, Term)$ recreates s_2

goto$(s_5, Factor)$ recreates s_3

goto$(s_5, \underline{\text{id}})$ recreates s_4

$s_8 \leftarrow$ ***goto***$(s_6, Term)$

$\{ [Term \rightarrow Factor * Term \bullet, \text{EOF}], [Term \rightarrow Factor * Term \bullet, -] \}$

goto$(s_6, Term)$ recreates s_3

goto$(s_6, \underline{\text{id}})$ recreates s_4

The next iteration creates no new sets.

Items in *black* are core items, generated by moving the placeholder.
Items in *gray* are added by *closure*().

Simplified Expression Grammar

Recorded Transitions

The Goto Relationship *(recorded during the construction)*

State	Expr	Term	Factor	-	*	id
s_0	1	2	3			4
s_1						
s_2				5		
s_3					6	
s_4						
s_5	7	2	3			4
s_6		8	3			4
s_7						
s_8						

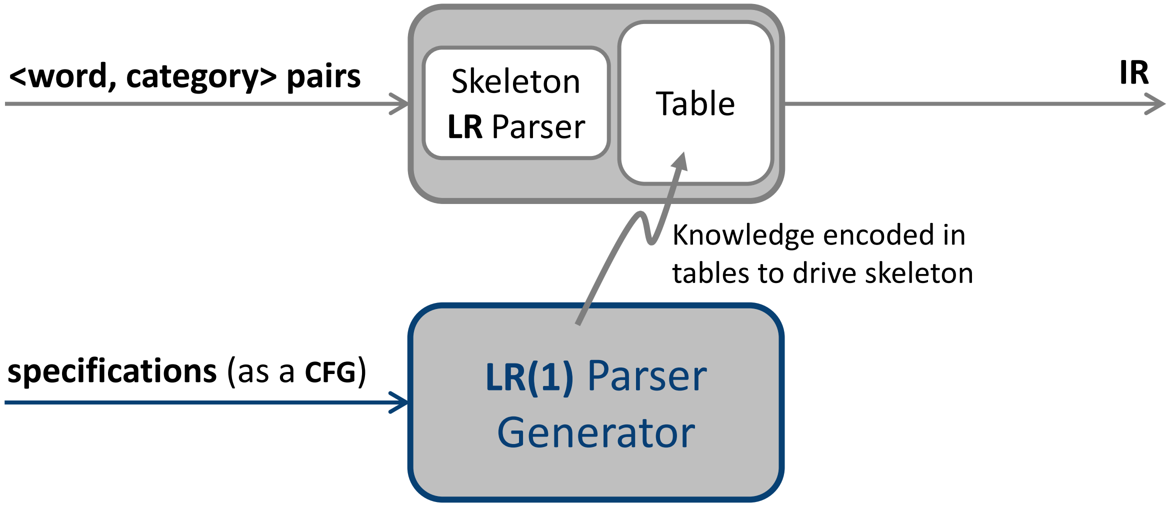
Simplified Expression Grammar

Filling in the Tables

The algorithm produces the following tables

State	ACTION				GOTO		
	id	-	*	EOF	*Expr*	*Term*	*Factor*
s_0	s 4				1	2	3
s_1				acc			
s_2		s 5		r 3			
s_3		r 5	s 6	r 5			
s_4		r 6	r 6	r 6			
s_5	s 4				7	2	3
s_6	s 4					8	3
s_7				r 2			
s_8		r 4		r 4			

Brief Commercial: Why Are We Doing This?



The goal of this exercise is to automate construction of parsers

- Compiler writer provides a **CFG** written in modified **BNF**
- Tools provide an efficient and correct parser
  - *One that works well with an automatically generated scanner*
- **LR** parser generators accept the largest class of grammars that are deterministically parsable, and they are highly efficient
  - *Generated parsers are preferable to hand-coded ones for large grammars*

§ 3.6.2 in EaC2e

Shrinking the ACTION and GOTO Tables

Three classic 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 mappings for ACTION & GOTO
- Use another construction algorithm
  - Both **LALR(1)** and **SLR(1)** produce smaller tables
    - → ***LALR(1)*** *represents each state with its "core" items*
    - → ***SLR(1)*** *uses* ***LR(0)*** *items and the* ***FOLLOW*** *set*
  - Implementations are readily available

Shrinking the Grammar

The Classic Expression Grammar

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* * *Factor*
5		|	*Term* / *Factor*
6		|	*Factor*
7	*Factor*	→	(*Expr*)
8		|	number
9		|	id

State	Action Table								
	eof	+	–	×	÷	(	)	num	name
0						s 4		s 5	s 6
1	acc	s 7	s 8						
2	r 4	r 4	r 4	s 9	s 10				
3	r 7	r 7	r 7	r 7	r 7				
4						s 14		s 15	s 16
5	r 9	r 9	r 9	r 9	r 9				
6	r 10	r 10	r 10	r 10	r 10				
7						s 4		s 5	s 6
8						s 4		s 5	s 6
9						s 4		s 5	s 6
10						s 4		s 5	s 6
11		s 21	s 22				s 23		
12		r 4	r 4	s 24	s 25		r 4		
13		r 7	r 7	r 7	r 7		r 7		
14						s 14		s 15	s 16
15		r 9	r 9	r 9	r 9		r 9		
16		r 10	r 10	r 10	r 10		r 10		
17	r 2	r 2	r 2	s 9	s 10				
18	r 3	r 3	r 3	s 9	s 10				
19	r 5	r 5	r 5	r 5	r 5				
20	r 6	r 6	r 6	r 6	r 6				
21						s 14		s 15	s 16
22						s 14		s 15	s 16
23	r 8	r 8	r 8	r 8	r 8				
24						s 14		s 15	s 16
25						s 14		s 15	s 16
26		s 21	s 22				s 31		
27		r 2	r 2	s 24	s 25		r 2		
28		r 3	r 3	s 24	s 25		r 3		
29		r 5	r 5	r 5	r 5		r 5		
30		r 6	r 6	r 6	r 6		r 6		
31		r 8	r 8	r 8	r 8		r 8		

■ **FIGURE 3.31** Action Table for the Classic Expression Grammar.

Canonical construction produces 32 states

- 32 x (9 + 3) = 384 ACTION/GOTO entries
- Large table, but still just 1.5kb

Shrinking the Grammar

We can combine some of the syntactically equivalent symbols

- Combine **+** and **–** into AddSub
- Combine ***** and **/** into MulDiv
- Combine identifier and number into Val

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* AddSub *Term*
2		|	*Term*
3	*Term*	→	*Term* MulDiv *Factor*
4		|	*Factor*
5	*Factor*	→	(*Expr*)
6		|	Val

The "Reduced" Expression Grammar

This grammar has

- Fewer terminals
- Fewer productions

Which leads to

- Fewer columns in ACTION
- Fewer states, which leads to fewer rows in both tables

Pages 151-153 in EaC2e

Shrinking the Grammar

The Resulting Tables

0	*Goal*	→	*Expr*		
1	*Expr*	→	*Expr*	AddSub	*Term*
2		|	*Term*		
3	*Term*	→	*Term*	MulDiv	*Factor*
4		|	*Factor*		
5	*Factor*	→	(	*Expr*	)
6		|	Val		

	Action Table						Goto Table		
	eof	addsub	muldiv	(	)	val	*Expr*	*Term*	*Factor*
0				s 4		s 5	1	2	3
1	acc	s 6							
2	r 3	r 3	s 7						
3	r 5	r 5	r 5						
4				s 11		s 12	8	9	10
5	r 7	r 7	r 7						
6				s 4		s 5		13	3
7				s 4		s 5			14
8		s 15			s 16				
9		r 3	s 17		r 3				
10		r 5	r 5		r 5				
11				s 11		s 12	18	9	10
12		r 7	r 7		r 7				
13	r 2	r 2	s 7						
14	r 4	r 4	r 4						
15				s 11		s 12		19	10
16	r 6	r 6	r 6						
17				s 11		s 12			20
18		s 15			s 21				
19		r 2	s 17		r 2				
20		r 4	r 4		r 4				
21		r 6	r 6		r 6				

(b) Action and Goto Tables for the Reduced Expression Grammar

■ **FIGURE 3.33** The Reduced Expression Grammar and its Tables.

- 22 states
- 22 * (6 + 3) = 198 ACTION/GOTO entries
- 48.4% reduction (384 - 198) / 384
- Builds (essentially) the same parse tree

Shrinking the ACTION and GOTO Tables

Three classic 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 mappings for ACTION & GOTO
- Use another construction algorithm
  - Both **LALR(1)** and **SLR(1)** produce smaller tables
    - → ***LALR(1)*** *represents each state with its "core" items*
    - → ***SLR(1)*** *uses* ***LR(0)*** *items and the* ***FOLLOW*** *set*
  - Implementations are readily available

left-recursive expression grammar with precedence, see § 3.6.2 in EAC

classic space-time tradeoff

Fewer grammars, same languages

LR(k) versus LL(k)

Finding the next step in a derivation

LR(k) $\Rightarrow$ Each reduction in the parse is detectable with

$\rightarrow$ the complete left context,

$\rightarrow$ the reducible phrase, itself, and

$\rightarrow$ the k terminal symbols to its right

generalizations of LR(1) and LL(1) to longer lookaheads

LL(k) $\Rightarrow$ Parser must select the expansion based on

$\rightarrow$ The complete left context

$\rightarrow$ The next k terminals

Thus, **LR(k)** examines more context

The question is, do languages fall in the gap between LR(k) and LL(k)?

Example due to Bill Waite, UC Boulder

LR(1) versus LL(1)

The following **LR(1)** grammar has no **LL(1)** counterpart

- The Canonical Collection has 18 sets of LR(1) Items
  - It is not a simple grammar
  - It is, however, LR(1)

0	*Goal*	→	*S*
1	*S*	→	*A*
2		|	*B*
3	*A*	→	(*A*)
4		|	<u>a</u>
5	*B*	→	(*B* >
6		|	<u>b</u>

- It requires an arbitrary lookahead to choose between *A* & *B*
- An **LR(1)** parser can carry the left context (the '<u>(</u>' s) until it sees <u>a</u> or <u>b</u>
- The table construction will handle it
- In contrast, an **LL(1)** parser cannot decide whether to expand *Goal* by *A* or *B*
  - → *No amount of massaging the grammar and no amount of lookahead will resolve this problem*

More precisely, the language described by this **LR(1)** grammar cannot be described with an **LL(1)** grammar. In fact, the language has no **LL(*k*)** grammar, for finite *k*.

ACTION & GOTO Tables for Waite's Example

	EOF	(	)	a	}
s_0		s 4		s 5	
s_1	acc				
s_2	r 2				
s_3	r 3				
s_4		s 8		s 9	
s_5	r 5				
s_6			s 10		
s_7					s 11
s_8		s 8		s 9	
s_9			r 5		r 7
s_{10}	r 4				
s_{11}	r 6				
s_{12}			s 14		
s_{13}					s 15
s_{14}			r 4		
s_{15}					r 6

	S	*A*	*B*
s_0	1	2	3
s_1			
s_2			
s_3			
s_4		6	7
s_5			
s_6			
s_7			
s_8		12	13
s_9			
s_{10}			
s_{11}			
s_{12}			
s_{13}			
s_{14}			
s_{15}			

0	*Start*	→	A
1		|	B
2	*A*	→	(*A*)
3		|	a
4	*B*	→	(*B* }
5		|	a

LR(*k*) versus LL(*k*)

Other Non-LL Grammars

0	B	$\rightarrow$	R
1		$\mid$	$\underline{(}\ B\ \underline{)}$
2	R	$\rightarrow$	$E = E$
3	E	$\rightarrow$	$\underline{a}$
4		$\mid$	$\underline{b}$
5		$\mid$	$\underline{(}\ E + E\ \underline{)}$

Example from D.E Knuth, “Top-Down Syntactic Analysis,” *Acta Informatica, 1:2 (1971), pages 79-110*

This grammar is actually LR(0)

0	S	$\rightarrow$	$\underline{a}\ A\ \underline{b}$
1		$\mid$	$\underline{c}$
2	A	$\rightarrow$	$\underline{b}\ S$
3		$\mid$	$B\ \underline{b}$
4	B	$\rightarrow$	$\underline{a}\ A$
5		$\mid$	$\underline{c}$

Example from Lewis, Rosenkrantz, & Stearns book, “Compiler Design Theory,” (1976), Figure 13.1

LR(k) versus LL(k)

Finding the next step in a derivation

LR(k) $\Rightarrow$ Each reduction in the parse is detectable with

$\rightarrow$ the complete left context,

$\rightarrow$ the reducible phrase, itself, and

$\rightarrow$ the k terminal symbols to its right

LL(k) $\Rightarrow$ Parser must select the expansion based on

$\rightarrow$ The complete left context

$\rightarrow$ The next k terminals

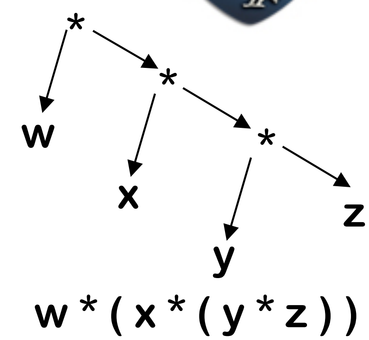
Thus, LR(k) examines more context

*"… in practice, **programming languages** do not actually seem to fall in the gap between LL(1) languages and deterministic languages"*

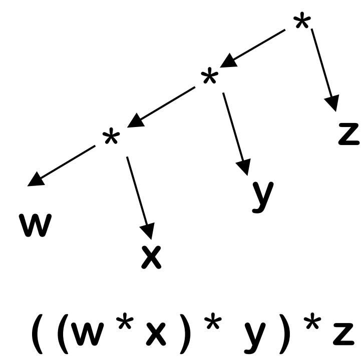
J.J. Horning, "LR Grammars and Analysers",
in Compiler Construction, An Advanced Course, Springer-Verlag, 1976

Left Recursion versus Right Recursion

- Right recursion
  - Required for termination in top-down parsers
  - Uses (on average) more stack space
  - Naïve right recursion produces right-associativity



- Left recursion
  - Works fine in bottom-up parsers
  - Limits required stack space
  - Naïve left recursion produces left-associativity



- Rule of thumb
  - Left recursion for bottom-up parsers
  - Right recursion for top-down parsers

Left Recursion versus Right Recursion

A real example, from the lab 1 ILOC simulator's front end

The simulator was built by two of my successful Ph.D.s

- It is actually a more complex piece of software than you might guess
- The front end is an LR(1) parser, generated by Bison
- The grammar contained the following productions:

instruction_list	:	*instruction*
	|	*label_def instruction*
	|	*instruction instruction_list*
	|	*label_def instruction instruction_list*

When my colleague first ran the timing blocks through the simulator, it exploded with the error message "memory exhausted".

$\Rightarrow$ What happened?

Left Recursion versus Right Recursion

A real example, from the lab 1 simulator's front end

instruction_list	:	*instruction*
	|	*label_def instruction*
	|	*instruction* ***instruction_list***
	|	*label_def instruction* ***instruction_list***

right recursion

- The parse stack overflowed as it tried to instantiate the instruction_list

Left Recursion versus Right Recursion

A real example, from the lab 1 simulator's front end

instruction_list	:	*instruction*
	|	*label_def instruction*
	|	*instruction* ***instruction_list***
	|	*label_def instruction* ***instruction_list***

right recursion

- The parse stack overflowed as it tried to instantiate the instruction_list
- The fix was easy

instruction_list	:	*instruction*
	|	*label_def instruction*
	|	***instruction_list*** *instruction*
	|	***instruction_list*** *label_def instruction*

left recursion

- This grammar has (small) bounded stack space & (thus) scales well

See pp. 145-146 in EaC2e

Error Detection and Recovery

Error Detection

- Recursive descent
  - Parser takes the last else clause in a routine
  - Compiler writer can code almost any arbitrary action
- Table-driven **LL(1)**
  - In state s_i facing word x, entry is an error
  - Report the error, valid entries in row for s_i encode possibilities
- Table-driven **LR(1)**
  - In state s_i facing word x, entry is an error
  - Report the error, shift states in row encode possibilities
  - Can precompute better messages from **LR(1)** items

Error Detection and Recovery

Error Recovery

- Table-driven **LL(1)**
  - Treat as missing token, e.g. ')', ⇒ expand by desired symbol
  - Treat as extra token, e.g., 'x-+y', ⇒ pop stack and move ahead
- Table-driven **LR(1)**
  - Treat as missing token, e.g. ')', ⇒ shift the token
  - Treat as extra token, e.g., 'x-+y', ⇒ don't shift the token

Can pre-compute sets of states with appropriate entries…

Error Detection and Recovery

One common strategy is "hard token" recovery

Skip ahead in input until we find some "hard" token, e.g. ';'

- ';' separates statements, makes a logical break in the parse
- Resynchronize state, stack, and input to point after hard token
  - → LL(1): pop stack until we find a row with entry for ';'
  - → LR(1): pop stack until we find a state with a reduction on ';'
- Does not correct the input, rather it allows parse to proceed

```
NT ← pop()
repeat until Table[NT,';'] ≠ error
    NT ← pop()
token ← NextToken()
repeat until token = ';'
    token ← NextToken()
```

*Resynchronizing an **LL(1)** parser*

```
repeat until token = ';'
    shift token
    shift s_e
    token ← NextToken()
reduce by error production
    // pops all that state off stack
```

*Resynchronizing an **LR(1)** parser*

Hierarchy of Context-Free Languages

Context-free languages

↓

Deterministic languages (**LR(*k*)**)

LR(*k*) ≡ **LR(1)**

↙ ↘

LL(*k*) languages

↓

LL(1) languages

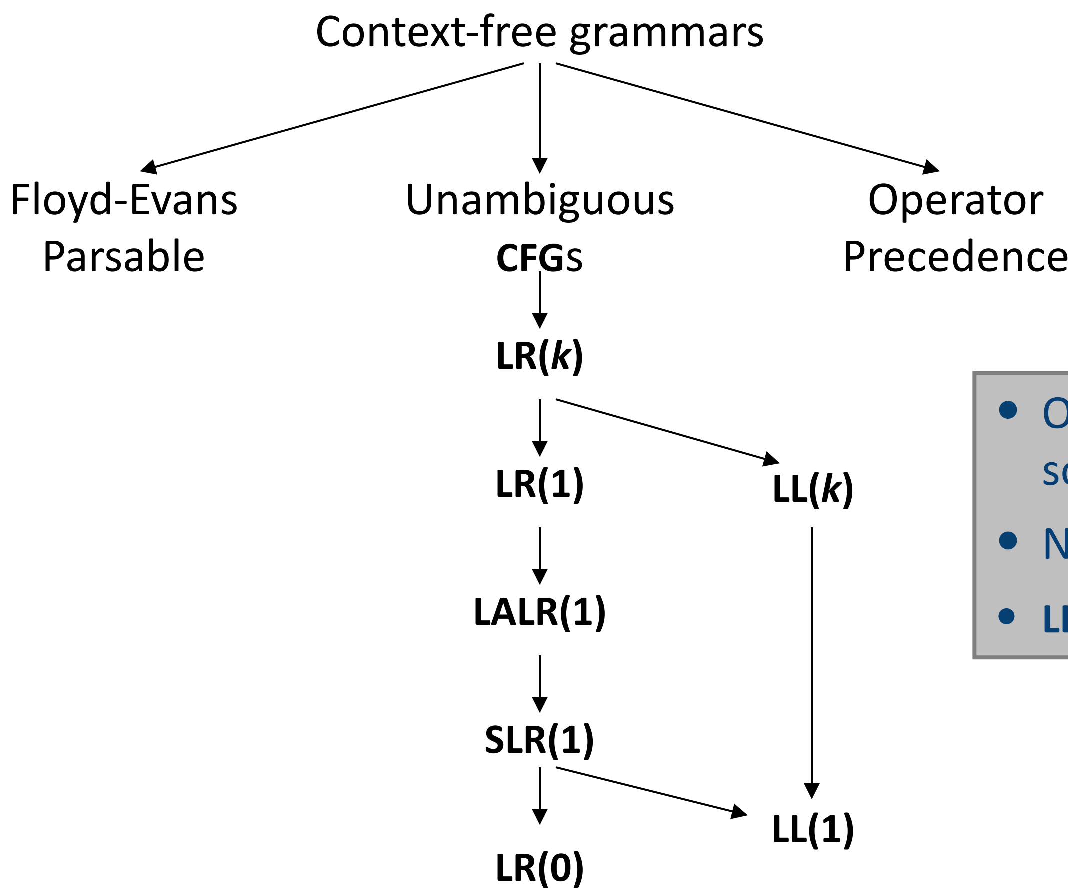
Simple precedence languages

↓

Operator precedence languages

The inclusion hierarchy for context-free ***languages***

Hierarchy of Context-Free Grammars



- Operator precedence includes some ambiguous grammars
- Note sub-categories of **LR(1)**
- **LL(1)** is a subset of **SLR(1)**

*The inclusion hierarchy for context-free **grammars***