RICE

COMP 412
FALL 2010

Parsing Wrap Up

Comp 412

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		|	a
5	*B*	→	(*B* >
6		|	b

- It requires an arbitrary lookahead to choose between *A* & *B*
- An LR(1) parser can carry the left context (the '(' s) until it sees a or b
- 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 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*.

Building LR(1) Tables for Waite's Language

The Canonical Collection of Sets of LR(1) Items

cc_0	[Goal→•S, EOF], [S→•A, EOF], [S→•B, EOF] [A→•(A), EOF], [A→•a, EOF], [B→•(B >, EOF], [B→•b, EOF],
cc_1	[Goal→S •, EOF]
cc_2	[S→A•,EOF]
cc_3	[S→B•,EOF]
cc_4	[A→ (• A),)], [A→ (•A), EOF], [A→•a,)], [B→• (B >, >], [B→ (• B >, EOF], [B→•b, >]
cc_5	[A→a•, EOF]
cc_6	[B→b•, EOF]
cc_7	[A→ (A •), EOF]
cc_8	[B→ (B • >, EOF]

cc_4 is goto(cc_0, ( )

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

Building LR(1) Tables for Waite's Language

And, the rest of it …

cc_9	[A→ (• A),)], [A→ (•A),)], [A→•a,)], [B→• (B >, >], [B→ (• B >, >], [B→•b, >]
cc_{10}	[A→a•,)]
cc_{11}	[B→b•, >]
cc_{12}	[A→(A)•, EOF]
cc_{13}	[B→ (B >•, EOF]
cc_{14}	[A→ (A •),)]
cc_{15}	[B→ (B • >, >]
cc_{16}	[A→ (A)•,)]
cc_{17}	[B→ (B >•, >]

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

Building LR(1) Tables for Waite's Language

	EOF	(	)	a	>	b	S	A	B
s_0		s 4		s 5		s 6	1	2	3
s_1	acc								
s_2	r 2								
s_3	r 3								
s_4		s 9		s 10		s 11		7	8
s_5	r 5								
s_6	r 7								
s_7			s 12						
s_8					s 13				
s_9				s 10		s 11		14	15
s_{10}			r 5						
s_{11}					r 7				
s_{12}	r 4								
s_{13}	r 6								
s_{14}			s 16						
s_{15}					s 17				
s_{16}			r 4						
s_{17}					r 6				

- Notice how sparse the table is.
  - Goto has 7 of 54
  - Action has 23 of 108
- Notice rows & columns that might be combined.
- Notice |CC| > |P|

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

LR(k) versus LL(k)

Other Non-LL Grammars

0	B	$\rightarrow$	R
1		|	<u>(</u> B <u>)</u>
2	R	$\rightarrow$	E <u>=</u> E
3	E	$\rightarrow$	<u>a</u>
4		|	<u>b</u>
5		|	<u>(</u> E <u>+</u> E <u>)</u>

0	S	$\rightarrow$	<u>a</u> A <u>b</u>
1		|	<u>c</u>
2	A	$\rightarrow$	<u>b</u> S
3		|	B <u>b</u>
4	B	|	<u>a</u> A
5		|	<u>c</u>

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

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

This grammar is actually LR(0)

LR(k) versus LL(k)

Finding the next step in a derivation

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

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

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

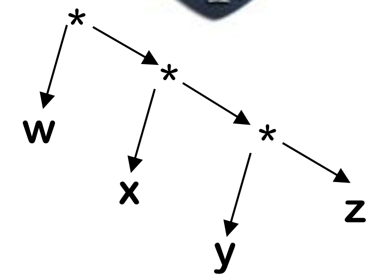
- → The complete left context
- → The next k terminals

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

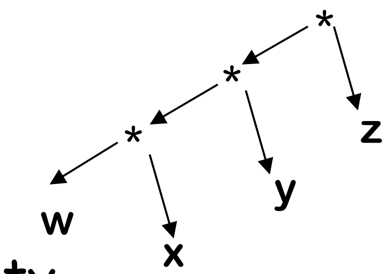
Left Recursion versus Right Recursion

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



w * (x * (y * z))

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



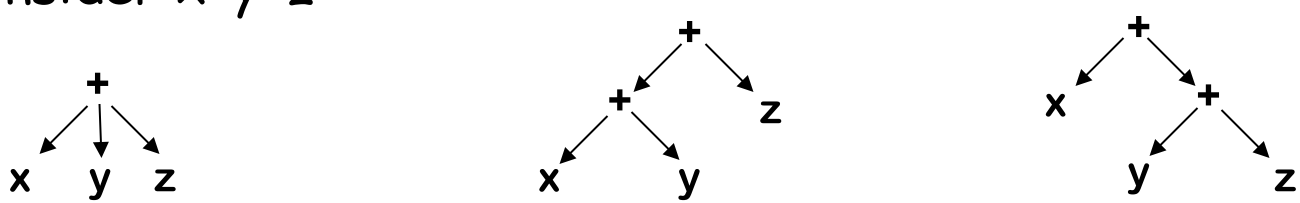
((w * x) * y) * z

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

Associativity

What difference does it make?

- Can change answers in floating-point arithmetic
- Can change opportunities for optimization
- Consider x+y+z



Ideal operator *Left association* *Right association*

What if y+z occurs elsewhere? Or x+y? or x+z?

The compiler may want to change the "shape" of expressions

- What if x = 2 & z = 17 ? Neither left nor right exposes 19.
- Best shape is function of surrounding context

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

Shrinking the ACTION and GOTO 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
    - → 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 § 3.7.2 in EAC

classic space-time tradeoff

Fewer grammars, same languages

Hierarchy of Context-Free Languages

Context-free languages

Deterministic languages (LR(k))

$LR(k) \equiv LR(1)$

LL(k) languages

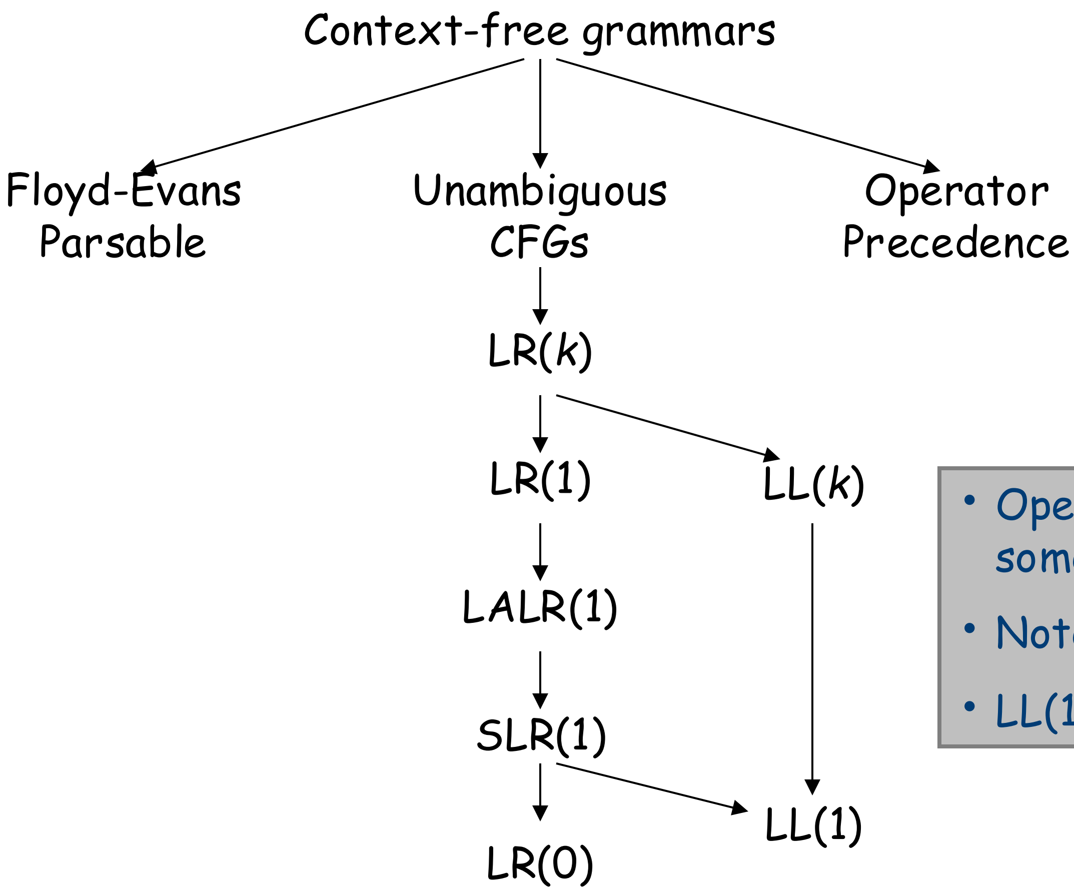
Simple precedence languages

LL(1) 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

Summary

	Advantages	*Disadvantages*
Top-down Recursive descent, LL(1)	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