COMP 412
FALL 2009

Parsing Wrap Up + Lab 2 Overview

Comp 412

Copyright 2009, Keith D. Cooper & Linda Torczon, all rights reserved.

Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.

Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

LR(k) versus LL(k)

Finding Reductions

LR(k) ⇒ 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) ⇒ Parser must select the reduction based on

- The complete left context
- The next k terminals

Thus, LR(k) examines more context

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

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

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 '(' 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 will resolve this problem*

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

More formally, 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*.

LR(1) versus LL(1)

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

LR(1) versus LL(1)

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

LR(1) versus LL(1)

	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				

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

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

LR(k) versus LL(k)

Finding Reductions

LR(k) ⇒ 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) ⇒ Parser must select the reduction based on

- The complete left context
- The next k terminals

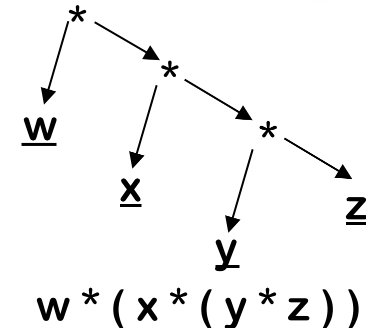
generalizations of LR(1) and LL(1) to longer lookaheads

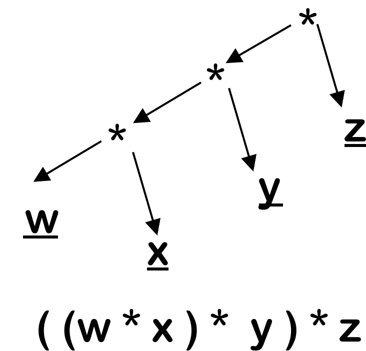
Thus, LR(k) examines more context

"… *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
  - 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

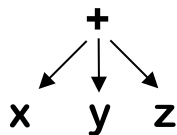




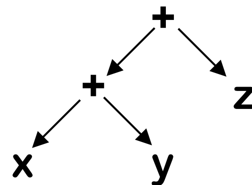
Associativity

What difference does it make?

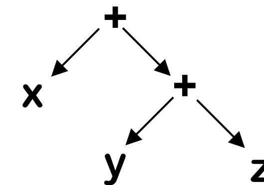
- Can change answers in floating-point arithmetic
- Can change opportunities for optimization
- Consider x+y+z



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

Hierarchy of Context-Free Languages

Context-free languages

Deterministic languages (LR(*k*))

LR(*k*) ≡ LR(1)

LL(*k*) languages

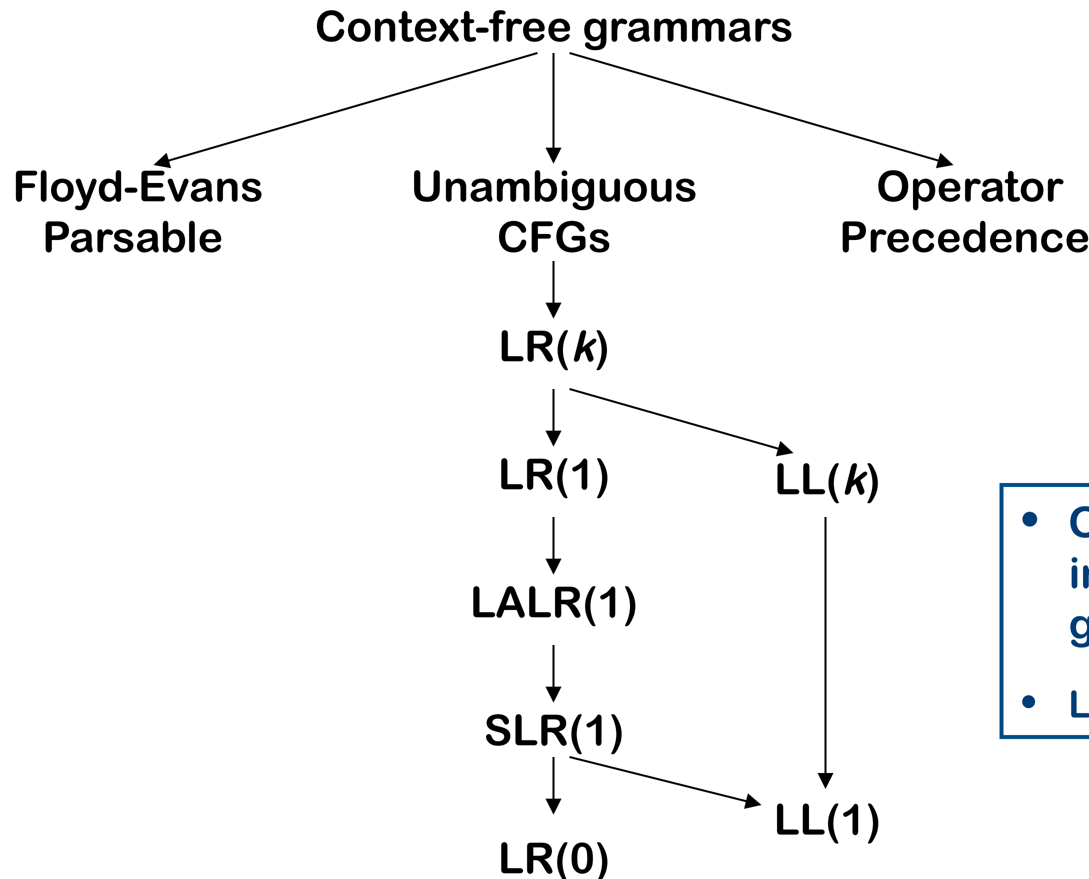
Simple precedence languages

LL(1) languages

Operator precedence languages

The inclusion hierarchy for context-free languages

Hierarchy of Context-Free Grammars



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

The inclusion hierarchy for context-free grammars

Lab 2 Overview: An LL(1) Parser Generator

You will build a program to generate LL(1) parse tables

- Input is a modified form of Backus-Naur Form (MBNF)
- Output is a parse table and some auxiliary data structures
  - Both a pretty (human-readable) version and the C declarations
- You will write:
  - A hand-coded scanner for MBNF
  - A hand-coded, recursive descent parser for MBNF
  - A table generator (First, Follow, $First^+$ sets)
- Interim deadlines to keep you on track
  - Want you to make progress at a steady rate
- Documents will be online by Wednesday
- Skeleton parser ready in about a week, with its own docs

Beyond Syntax

There is a level of correctness that is deeper than grammar

```
fie(a,b,c,d)
   int a, b, c, d;
{ … }

fee() {
   int f[3],g[0],
      h, i, j, k;
  char *p;
   fie(h,i,“ab”,j, k);
   k = f * i + j;
   h = g[17];
   printf(“<%s,%s>.\n”,
      p,q);
   p = 10;
}
```

What is wrong with this C program?

(let me count the ways …)

To generate code, we need to understand its meaning !

There is a level of correctness that is deeper than grammar

```
fie(a,b,c,d)
  int a, b, c, d;
{ … }
fee() {
  int f[3],g[0],
    h, i, j, k;
 char *p;
  fie(h,i,"ab",j, k);
  k = f * i + j;
  h = g[17];
  printf("<%s,%s>.\n",
    p,q);
  p = 10;
}
```

What is wrong with this C program?

(let me count the ways …)

- declared g[0], used g[17]
- wrong number of args to fie()
- "ab" is not an <u>int</u>
- wrong dimension on use of f
- undeclared variable q
- 10 is not a character string

All of these are "deeper than syntax"

Beyond Syntax

To generate code, the compiler needs to answer many questions

- Is "x" a scalar, an array, or a function? Is "x" declared?
- Are there names that are not declared? Declared but not used?
- Which declaration of "x" does each use reference?
- Is the expression "x * y + z" type-consistent?
- In "*a*[i,j,k]", does *a* have three dimensions?
- Where can "z" be stored? *(register, local, global, heap, static)*
- In "f ← 15", how should 15 be represented?
- How many arguments does "fie()" take? What about "printf ()" ?
- Does "*p" reference the result of a "malloc()" ?
- Do "p" & "q" refer to the same memory location?
- Is "x" defined before it is used?

These cannot be expressed in a CFG