Parsing Wrap-up

Filling in the ACTION and GOTO Tables

The algorithm

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

x is the state number

Many items generate no table entry

> *Closure( )* instantiates FIRST(γ) directly for $[\alpha \rightarrow \beta \bullet \gamma\delta, \underline{a}]$

What can go wrong?

What if set *s* contains $[\alpha \rightarrow \beta \bullet \underline{a}\gamma, \underline{b}]$ and $[\alpha \rightarrow \beta \bullet, \underline{a}]$?

- First item generates "shift", second generates "reduce"
- Both define ACTION[s,$\underline{a}$] — 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

What if set *s* contains $[\alpha \rightarrow \gamma, \underline{a}]$ and $[\beta \rightarrow \gamma \bullet, \underline{a}]$?

- Each generates "reduce", but with a different production
- Both define ACTION[s,$\underline{a}$] — cannot do both reductions
- This is a fundamental ambiguity, called a *reduce/reduce conflict*
- Modify the grammar to eliminate it *(PL/I's and FORTRAN's overloading*
- *of (...))*

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
  - 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

Direct Encoding

Rather than using a table-driven interpreter …

- Generate spaghetti code that implements the logic
- Each state becomes a small case statement or if-then-else
- Analogous to direct coding a scanner

Advantages

- No table lookups and address calculations
-
- No outer loop —it is implicit in the code for the states

This produces a faster parser with more code but no table

LR(k) versus LL(k) *(Top-down Recursive Descent)*

Finding Reductions

LR(*k*) ⇒ Each reduction in the parse is detectable with

1. the complete left context,
2. the reducible phrase, itself, and
3. the *k* terminal symbols to its right

LL(*k*) ⇒ Parser must select the reduction based on

1. The complete left context
2. The next *k* terminals

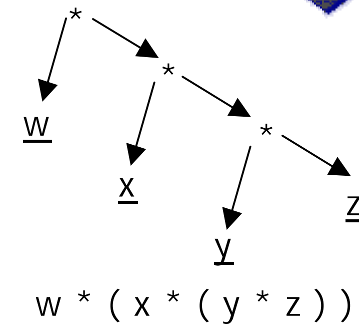
Thus, LR(*k*) examines more context

"… 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

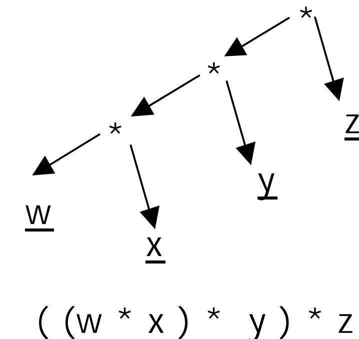
Right recursion

- Required for termination in top-down parsers
- Uses (on average) more stack space
- Produces right-associative operators



Left recursion

- Works fine in bottom-up parsers
- Limits required stack space
- Produces left-associative operators



Rule of thumb

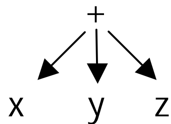
- Left recursion for bottom-up parsers
- Right recursion for top-down parsers

Associativity

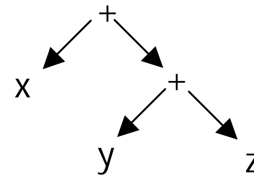
What difference does it make?

- Can change answers in floating-point arithmetic
- Exposes a different set of common subexpressions

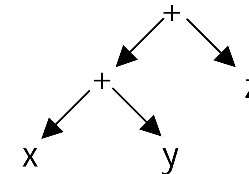
Consider x+y+z



Ideal operator



Left association



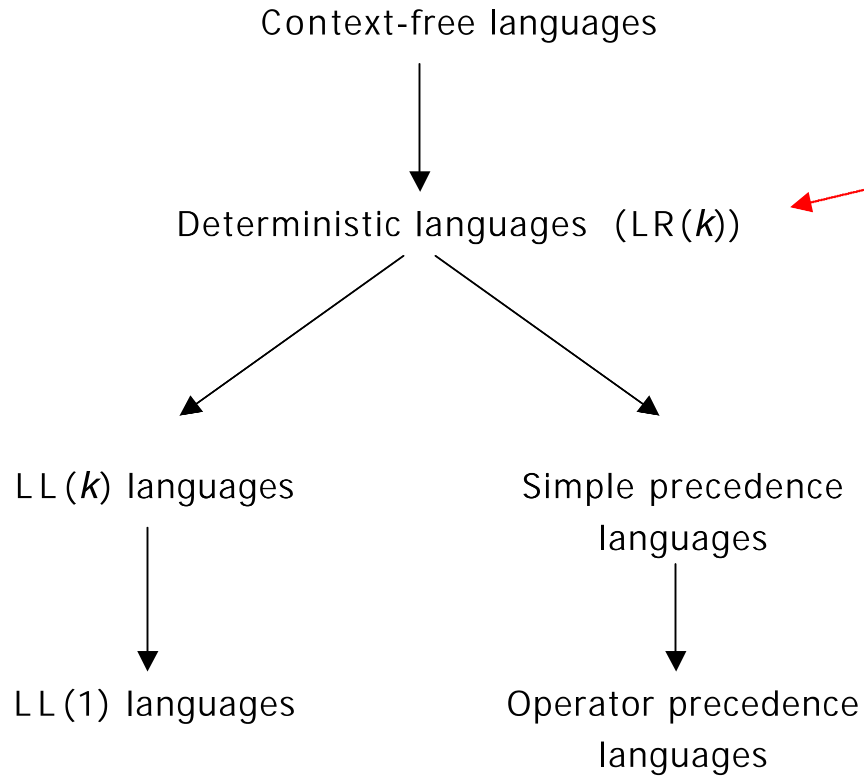
Right association

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

What if x = 2 & z = 17 ? Neither left nor right exposes 19.

Best choice is function of surrounding context

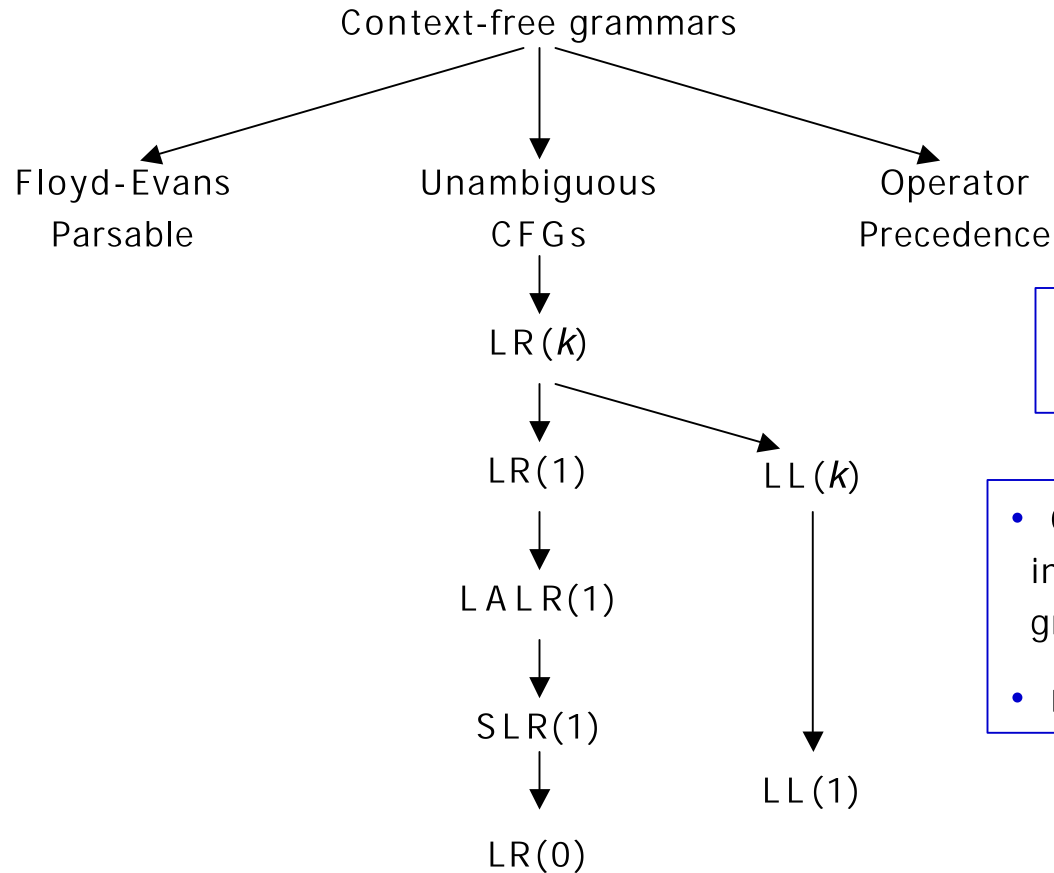
Hierarchy of Context-Free Languages



LR(*k*) ≡ LR(1)

The inclusion hierarchy for context-free languages

Hierarchy of Context-Free Grammars



Inclusion hierarchy for context-free grammars

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

Summary

	Advantages	*Disadvantages*
Top-down recursive descent	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