Parsing IV
Bottom-up Parsing

Parsing Techniques

Top-down parsers *(LL(1), recursive descent)*

- Start at the root of the parse tree and grow toward leaves
- Pick a production & try to match the input
- Bad "pick" ⇒ may need to backtrack
- Some grammars are backtrack-free *(predictive parsing)*

Bottom-up parsers *(LR(1), operator precedence)*

- Start at the leaves and grow toward root
- As input is consumed, encode possibilities in an internal state
- Start in a state valid for legal first tokens
- Bottom-up parsers handle a large class of grammars

Bottom-up Parsing (definitions)

The point of parsing is to construct a ***derivation***

A derivation consists of a series of rewrite steps

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \textit{sentence}$$

- Each γ_i is a sentential form
  - → If γ contains only terminal symbols, γ is a sentence in *L(G)*
  - → If γ contains ≥ 1 non-terminals, γ is a sentential form
- To get γ_i from γ_{i-1}, expand some NT $A \in \gamma_{i-1}$ by using $A \rightarrow \beta$
  - → Replace the occurrence of $A \in \gamma_{i-1}$ with β to get γ_i
  - → In a leftmost derivation, it would be the first NT $A \in \gamma_{i-1}$

A *left-sentential form* occurs in a *leftmost* derivation

A *right-sentential form* occurs in a *rightmost* derivation

Bottom-up Parsing

A bottom-up parser builds a derivation by working from the input sentence back toward the start symbol S

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \textit{sentence}$$

$$\xleftarrow{\hspace{10em}} \text{ bottom-up}$$

To reduce γ_i to γ_{i-1} match some *rhs* β against γ_i then replace β with its corresponding *lhs*, A. *(assuming the production $A \rightarrow \beta$)*

In terms of the parse tree, this is working from leaves to root

- Nodes with no parent in a partial tree form its *upper fringe*
- Since each replacement of β with A shrinks the upper fringe, we call it a *reduction*.

The parse tree need not be built, it can be simulated

$$|\textit{parse tree nodes}| = |\textit{words}| + |\textit{reductions}|$$

Finding Reductions

Consider the simple grammar

1	*Goal*	→	a *A* *B* e
2	*A*	→	*A* b c
3		|	b
4	*B*	→	d

And the input string abbcde

Sentential Form	*Next Reduction*	
	Prod'n	*Pos'n*
abbcde	3	2
a *A* bcde	2	4
a *A* de	4	3
a *A* *B* e	1	4
Goal	—	—

The trick is scanning the input and finding the next reduction

The mechanism for doing this must be efficient

Finding Reductions (Handles)

The parser must find a substring β of the tree's frontier that *matches some production $A \to \beta$ that occurs as one step in the rightmost derivation* *($\Rightarrow \beta \to A$ is in RRD)*

Informally, we call this substring β a *handle*

Formally,

A *handle* of a right-sentential form γ is a pair $\langle A \to \beta, \mathbf{k} \rangle$ where $A \to \beta \in P$ and k is the position in γ of β's rightmost symbol.

If $\langle A \to \beta, \mathbf{k} \rangle$ is a handle, then replacing β at k with A produces the right sentential form from which γ is derived in the rightmost derivation.

Because γ is a right-sentential form, the substring to the right of a handle contains only terminal symbols

$\Rightarrow$ the parser doesn't need to scan past the handle *(very far)*

Finding Reductions (Handles)

Critical Insight *(Theorem?)*

If G is unambiguous, then every right-sentential form has a unique handle.

If we can find those handles, we can build a derivation !

Sketch of Proof:

1 *G* is unambiguous ⇒ rightmost derivation is unique

2 ⇒ a unique production $A \rightarrow \beta$ applied to derive γ_i from γ_{i-1}

3 ⇒ a unique position ***k*** at which $A \rightarrow \beta$ is applied

4 ⇒ a unique handle $\langle A \rightarrow \beta, \boldsymbol{k} \rangle$

This all follows from the definitions

Example

(a very busy slide)

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

The expression grammar

Prod'n.	*Sentential Form*	*Handle*
—	*Goal*	—
1	*Expr*	1,1
3	*Expr* - *Term*	3,3
5	*Expr* - *Term* * *Factor*	5,5
9	*Expr* - *Term* * <id,y>	9,5
7	*Expr* - *Factor* * <id,y>	7,3
8	*Expr* - <num,2> * <id,y>	8,3
4	*Term* - <num,2> * <id,y>	4,1
7	*Factor* - <num,2> * <id,y>	7,1
9	<id,x> - <num,2> * <id,y>	9,1

Handles for rightmost derivation of x - 2 * y

This is the inverse of Figure 3.9 in EaC

Handle-pruning, Bottom-up Parsers

The process of discovering a handle & reducing it to the appropriate left-hand side is called *handle pruning*

Handle pruning forms the basis for a bottom-up parsing method

To construct a rightmost derivation

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow w$$

Apply the following simple algorithm

```
for $i \leftarrow n$ to $1$ by $-1$
    Find the handle $\langle A_i \rightarrow \beta_i , \mathbf{k}_i \rangle$ in $\gamma_i$
    Replace $\beta_i$ with $A_i$ to generate $\gamma_{i-1}$
```

This takes *2n* steps

Handle-pruning, Bottom-up Parsers

One implementation technique is the *shift-reduce parser*

```
push INVALID
token ← next_token( )
repeat until (top of stack = Goal and token = EOF)
    if the top of the stack is a handle $A \to \beta$
        then    // reduce $\beta$ to A
            pop $|\beta|$ symbols off the stack
            push A onto the stack
        else if (token ≠ EOF)
            then // shift
                push token
                token ← next_token( )
        else    // need to shift, but out of input
            report an error
```

How do errors show up?

- failure to find a handle
- hitting EOF & needing to shift (final else clause)

Either generates an error

Figure 3.7 in EAC

Back to x - 2 * y

Stack	Input	Handle	Action
$	id – num * id	*none*	shift
$id	– num * id		

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Back to x - 2 * y

Stack	Input	Handle	Action
$	id – num * id	*none*	shift
$id	– num * id	9,1	red. 9
$ *Factor*	– num * id	7,1	red. 7
$ *Term*	– num * id	4,1	red. 4
$ *Expr*	– num * id		

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Back to x - 2 * y

Stack	Input	Handle	Action
$	id – num * id	*none*	shift
$id	– num * id	9,1	red. 9
$ *Factor*	– num * id	7,1	red. 7
$ *Term*	– num * id	4,1	red. 4
$ *Expr*	– num * id	*none*	shift
$ *Expr* –	num * id	*none*	shift
$*Expr* – num	* id		

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Back to x - 2 * y

Stack	Input	Handle	Action
$	id – num * id	*none*	shift
$id	– num * id	9,1	red. 9
$ *Factor*	– num * id	7,1	red. 7
$ *Term*	– num * id	4,1	red. 4
$ *Expr*	– num * id	*none*	shift
$ *Expr* –	num * id	*none*	shift
$*Expr* – num	* id	8,3	red. 8
$ *Expr* – *Factor*	* id	7,3	red. 7
$*Expr* – *Term*	* id		

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Back to x - 2 * y

Stack	Input	Handle	Action
$	id – num * id	*none*	shift
$id	– num * id	9,1	red. 9
$ *Factor*	– num * id	7,1	red. 7
$ *Term*	– num * id	4,1	red. 4
$ *Expr*	– num * id	*none*	shift
$ *Expr* –	num * id	*none*	shift
$*Expr* – num	* id	8,3	red. 8
$ *Expr* – *Factor*	* id	7,3	red. 7
$*Expr* – *Term*	* id	*none*	shift
$*Expr* – *Term* *	id	*none*	shift
$ *Expr* – *Term* * id			

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Back to x - 2 * y

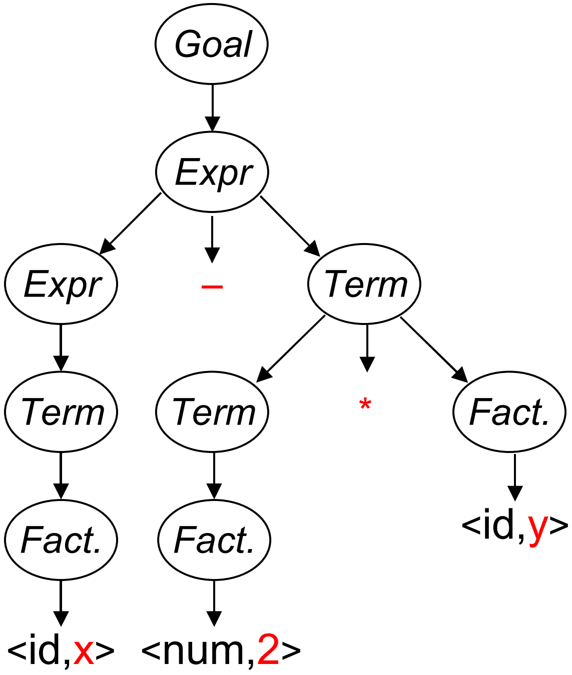
Stack	Input	Handle	Action
$	id – num * id	*none*	shift
$id	– num * id	9,1	red. 9
$ *Factor*	– num * id	7,1	red. 7
$ *Term*	– num * id	4,1	red. 4
$ *Expr*	– num * id	*none*	shift
$ *Expr* –	num * id	*none*	shift
$*Expr* – num	* id	8,3	red. 8
$ *Expr* – *Factor*	* id	7,3	red. 7
$*Expr* – *Term*	* id	*none*	shift
$*Expr* – *Term* *	id	*none*	shift
$ *Expr* – *Term* * id		9,5	red. 9
$ *Expr* – *Term* * *Factor*		5,5	red. 5
$ *Expr* – *Term*		3,3	red. 3
$ *Expr*		1,1	red. 1
$ *Goal*		*none*	accept

5 shifts +
9 reduces +
1 accept

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Example

Stack	Input	Action
$	id – num * id	shift
$id	– num * id	red. 9
$ *Factor*	– num * id	red. 7
$ *Term*	– num * id	red. 4
$ *Expr*	– num * id	shift
$ *Expr* –	num * id	shift
$*Expr* – num	* id	red. 8
$ *Expr* – *Factor*	* id	red. 7
$*Expr* – *Term*	* id	shift
$*Expr* – *Term* *	id	shift
$ *Expr* – *Term* * id		red. 9
$ *Expr* – *Term* * *Factor*		red. 5
$ *Expr* – *Term*		red. 3
$ *Expr*		red. 1
$ *Goal*		accept



Shift-reduce Parsing

Shift reduce parsers are easily built and easily understood

A shift-reduce parser has just four actions

- *Shift* — next word is shifted onto the stack
- *Reduce* — right end of handle is at top of stack
    - Locate left end of handle within the stack
    - Pop handle off stack & push appropriate *lhs*
- *Accept* — stop parsing & report success
- *Error* — call an error reporting/recovery routine

Accept & *Error* are simple

Shift is just a push and a call to the scanner

Reduce takes |*rhs*| pops & 1 push

If handle-finding requires state, put it in the stack ⇒ 2x work

Handle finding is key
- handle is on stack
- finite set of handles

⇒ use a DFA !

An Important Lesson about Handles

To be a handle, a substring of a sentential form γ must have two properties:

  - It must match the right hand side β of some rule $A \rightarrow \beta$
  - There must be some rightmost derivation from the goal symbol that produces the sentential form γ with $A \rightarrow \beta$ as the last production applied

- Simply looking for right hand sides that match strings is not good enough
- Critical Question: How can we know when we have found a handle without generating lots of different derivations?
  - Answer: we use look ahead in the grammar along with tables produced as the result of analyzing the grammar.
  - *LR(1)* parsers build a DFA that runs over the stack & finds them

LR(1) Parsers

- LR(1) parsers are table-driven, shift-reduce parsers that use a limited right context (1 token) for handle recognition
- LR(1) parsers recognize languages that have an LR(1) grammar

Informal definition:

A grammar is LR(1) if, given a rightmost derivation

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \mathit{sentence}$$

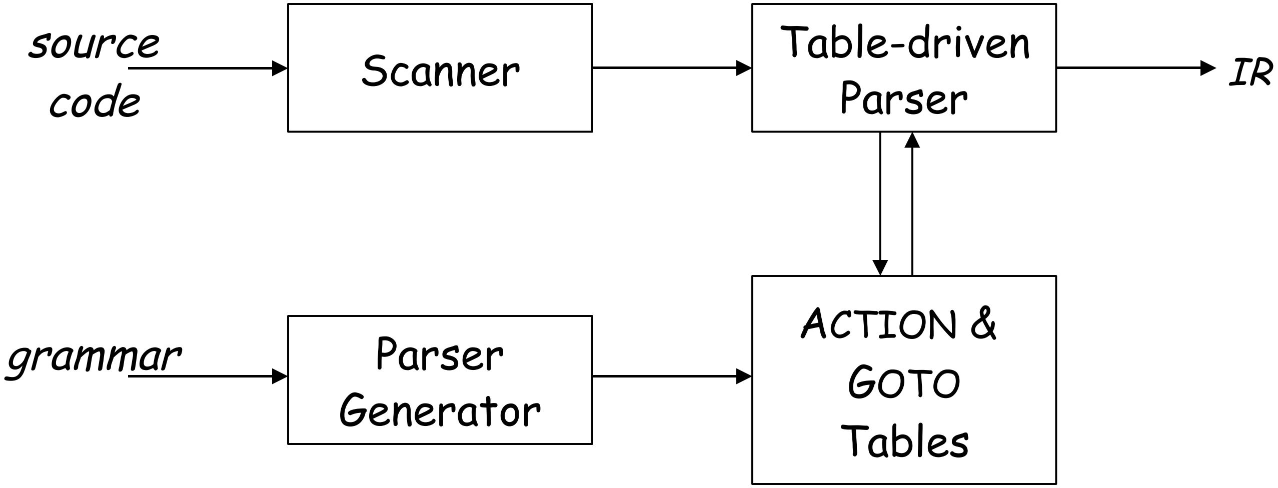
We can

1. *isolate the handle of each right-sentential form* γ_i, and
2. *determine the production by which to reduce,*

by scanning γ_i from left-to-right, going at most 1 symbol beyond the right end of the handle of γ_i

LR(1) Parsers

A table-driven LR(1) parser looks like



Tables *can* be built by hand

However, this is a perfect task to automate

LR(1) Skeleton Parser

```
stack.push(INVALID); stack.push($s_0$);
not_found = true;
token = scanner.next_token();
do while (not_found) {
    s = stack.top();
    if ( ACTION[s,token] == "reduce $A \rightarrow \beta$" ) then {
        stack.popnum(2*|$\beta$|); // pop 2*|$\beta$| symbols
    s = stack.top();
    stack.push($A$);
    stack.push(GOTO[s,$A$]);
    }
    else if ( ACTION[s,token] == "shift $s_i$" ) then {
        stack.push(token); stack.push($s_i$);
        token ← scanner.next_token();
    }
    else if ( ACTION[s,token] == "accept"
                    & token == EOF )
        then not_found = false;
    else report a syntax error and recover;
}
report success;
```

The skeleton parser

- uses ACTION & GOTO tables
- does |*words*| shifts
- does |derivation| reductions
- does 1 accept
- detects errors by failure of 3 other cases

LR(1) Parsers (parse tables)

To make a parser for *L(G)*, need a set of tables

Remember, this is the left-recursive SheepNoise; EaC shows the right-recursive version.

The grammar

1	*Goal*	→	SheepNoise
2	*SheepNoise*	→	SheepNoise baa
3		|	baa

The tables

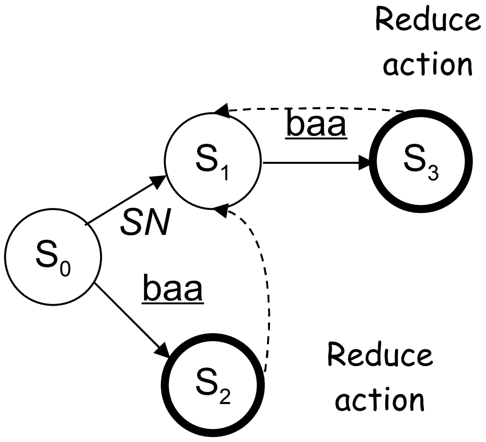
ACTION		
State	EOF	baa
0	—	shift 2
1	accept	shift 3
2	reduce 3	reduce 3

GOTO	
State	*SheepNoise*
0	1
1	0
2	0
3	0

LR(1) Parsers

How does this LR(1) stuff work?

- Unambiguous grammar ⇒ unique rightmost derivation
- Keep upper fringe on a stack
  - → All active handles include top of stack (TOS)
  - → Shift inputs until TOS is right end of a handle
- Language of handles is regular (finite)
  - → Build a handle-recognizing DFA
  - → ACTION & GOTO tables encode the DFA
- To match subterm, invoke subterm DFA & leave old DFA's state on stack
- Final state in DFA ⇒ a *reduce* action
  - → New state is GOTO[state at TOS (after pop), *lhs*]
  - → For *SN*, this takes the DFA to s_1



Control DFA for SN

Building LR(1) Parsers

How do we generate the *ACTION* and *GOTO* tables?

- Use the grammar to build a model of the DFA
- Use the model to build *ACTION* & *GOTO* tables
- If construction succeeds, the grammar is LR(1)

The Big Picture

- Model the state of the parser
- Use two functions *goto(s, X)* and *closure(s)*
  - → *goto()* is analogous to *move()* in the subset construction
  - → *closure()* adds information to round out a state
- Build up the states and transition functions of the DFA
- Use this information to fill in the *ACTION* and *GOTO* tables

Terminal or non-terminal

What can go wrong?

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

- 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

EaC includes a worked example

What is set *s* contains $[A \rightarrow \gamma \bullet, \underline{a}]$ and $[B \rightarrow \gamma \bullet, \underline{a}]$?

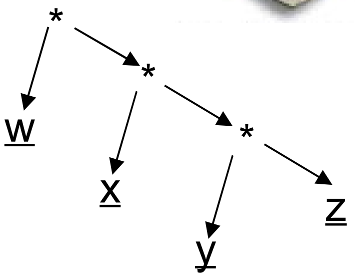
- Each generates “reduce”, but with a different production
- Both define ACTION$[s, \underline{a}]$ — cannot do both reductions
- This fundamental ambiguity is called a *reduce/reduce error*
- Modify the grammar to eliminate it *(PL/I’s overloading of (...))*

In either case, the grammar is not LR(1)

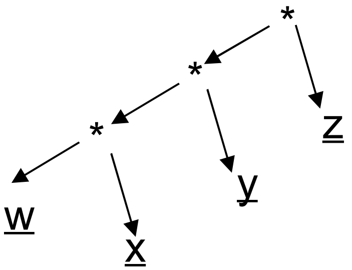
Left Recursion versus Right Recursion

- 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



w * (x * (y * z))

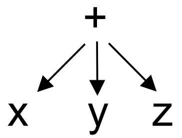


((w * x) * y) * z

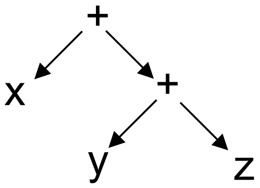
Associativity

- What difference does it make?
- Can change answers in floating-point arithmetic
- Exposes a different set of common subexpressions

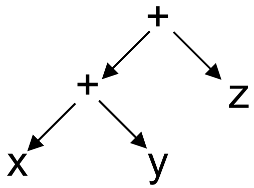
- Consider x+y+z



Ideal operator



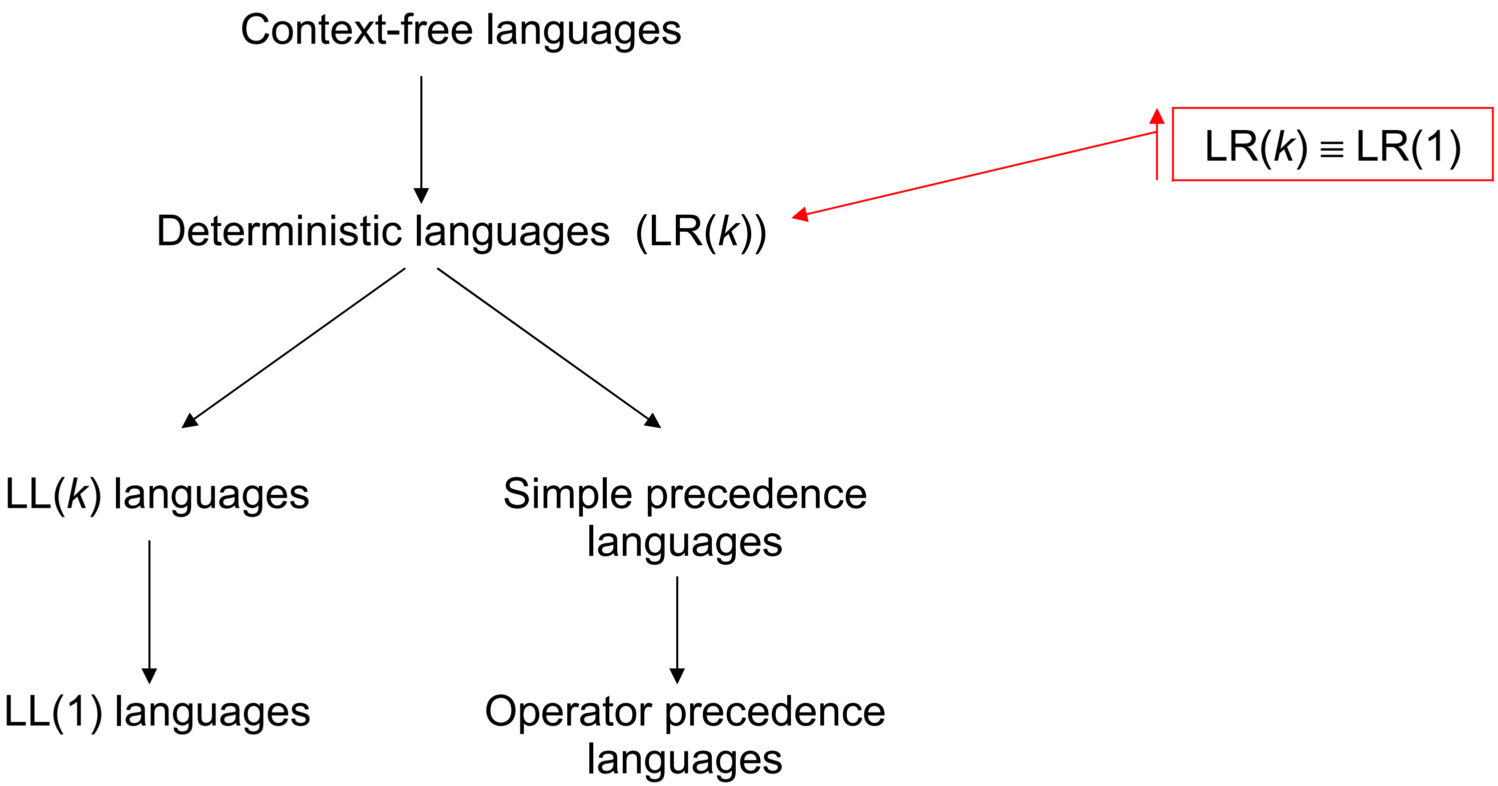
Left association



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



Context-free languages

Deterministic languages (LR(k))

LR(k) ≡ LR(1)

LL(k) languages

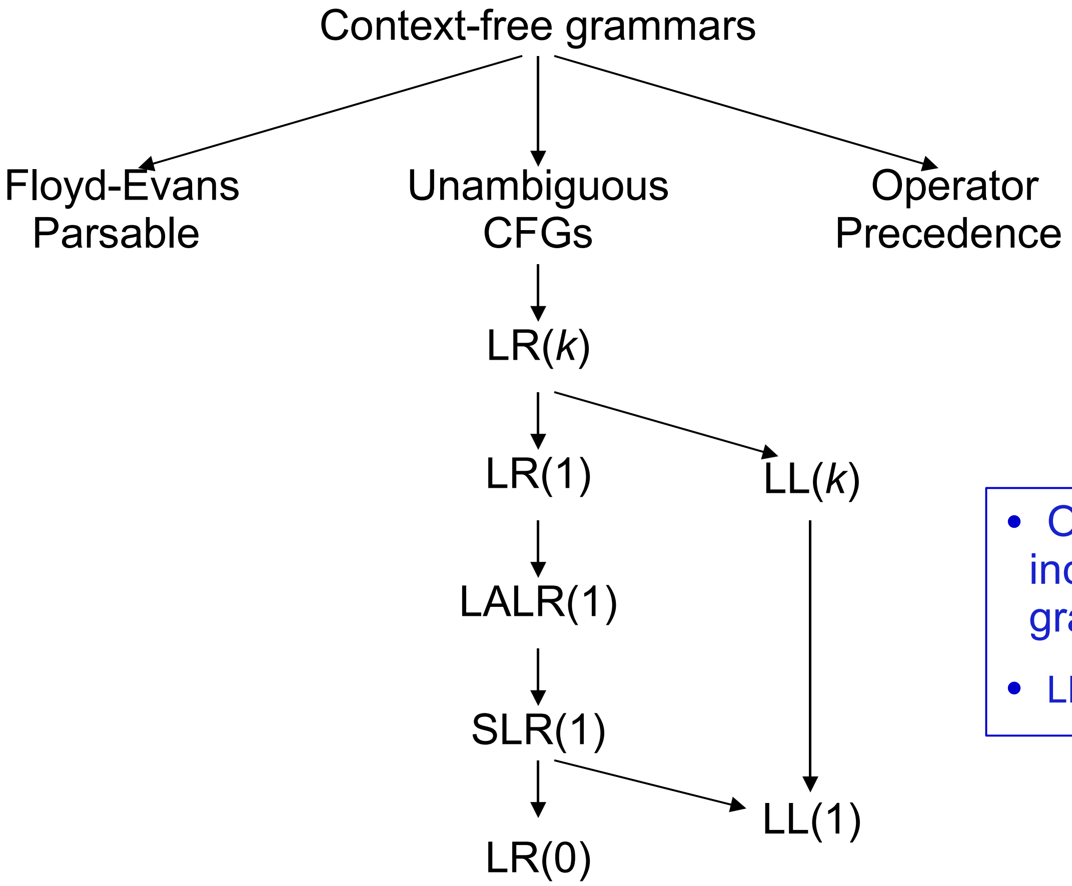
Simple precedence languages

LL(1) languages

Operator precedence languages

The inclusion hierarchy for context-free languages

Hierarchy of Context-Free Grammars


Context-free grammars

Floyd-Evans Parsable

Unambiguous CFGs

Operator Precedence

LR(*k*)

LR(1)

LL(*k*)

LALR(1)

SLR(1)

LR(0)

LL(1)

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

The inclusion hierarchy for context-free grammars

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

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*

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

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 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 program?

(let me count the ways …)

- declared g[0], used g[17]
- wrong number of args to fie()
- “ab” is not an int
- 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

Beyond Syntax

These questions are part of context-sensitive analysis

- Answers depend on values, not parts of speech
- Questions & answers involve non-local information
- Answers may involve computation

How can we answer these questions?

- Use formal methods
  - → Context-sensitive grammars?
  - → Attribute grammars? *(attributed grammars?)*
- Use *ad-hoc* techniques
  - → Symbol tables
  - → *Ad-hoc* code *(action routines)*

In scanning & parsing, formalism won; different story here.