Precedence Grammars

LR parsing is Bottom-Up. We want to find the parse that reverses the derivation that always expands the right-most non-terminal symbol.

Example for the grammar E::=E+T|T   T ::= T*F|F F::= id
We derive and parse the string x+y*z
Derivation:

E
E+T
E+T*F
E+T*z
E+F*z
E+y*z
T+y*z
F+y*z
x+y*z

The parse we want reverses this.

Again the grammar is E::=E+T|T T ::= T*F|F F::= id

We parse the string x+y*z

x+y*z

F+y*z

T+y*z

E+y*z

E+F*z

E+T*z

E+T*F

E+T

E

Terminology:

1. A prime phrase is the right side of any grammar rule.
2. A handle is the prime phrase that represents one step in the reversal of a right-most derivation.

Examples from the previous bottom-up parse. On each line the prime phrases are underlined and the handle is indicated with H.

x+y*z
H

E+y*z
H

E+T*F

H

We will develop a series of increasingly general classes of grammars, building towards LR(k) grammars.

Def. A <u>parenthesis grammar</u> is one in which

a) The right hand side of every rule is enclosed in parentheses.
b) Parentheses occur nowhere else.
c) No two rules have the same right hand side.

Example: S::= (aA)   A ::= (Aa) | (a) | (SA)

Consider parsing (a ((a (a)) ((a) a)))

(a ((a (a)) ((a) a)))
(a ((a A) ((a) a)))
(a (S ((a) a)))
(a (S (A a)))
(a (S A))
(a A)
S

The parentheses make the prime phrases disjoint. The handle is always the leftmost prime phrase.

We can parse a parenthesis language with a stack machine:

a) Start with an empty stack.
b) At each step, if ")" is at the top of the stack, perform a reduction by popping the stack to the first "(" and pushing the appropriate non-terminal on the stack.
c) The Start symbol should be on the stack at the end of the input.

Try this with the previous example. It works.

Def. A simple precedence grammar is one in which we can insert symbols "<", "=", and ">" to produce a language (treating "<" and ">" as parentheses) that can be parsed like a parenthesized grammar.

To parse a simple precedence language we need a precedence table. The entries are the new symbols "<", "=" and ">", indexed by the symbols that could be on the stack (all terminals and non-terminals) and any symbols we might push on the stack (also all terminals and non-terminals).

At each step we insert the symbol from the table between the current stack top and the new symbol to be pushed on. If the table entry is ">" we do a reduction before pushing anything new onto the stack.

We always start with "<" and the first token on the stack, and at EOF push ">". We should end with the Start symbol on the stack.

Example. Grammar S::=Aab A::=aS | c

Precedence table:

	S	**A**	**a**	**b**	**c**
S			>		
A			=		
a	=	<	<	=	<
b			>		
c			>		

Try using this to parse acabab or aacababab

To generate the precedence table we need two relations:
$\mathcal{L}$(A) is the set of left-most symbols of strings generated from A.
$\mathcal{R}$(A) is the set of right-most symbols of strings generated from A.

These are easy to generate. For the grammar

S ::= Aab	$\mathcal{L}$(S)={A, a, c}	$\mathcal{R}$(S)={b}
A ::= aS | c	$\mathcal{L}$(A)={a, c}	$\mathcal{R}$(A)={S, c, b}

To build the precedence table apply the following rules. The grammar is a simple precedence grammar if this can be done unambiguously.

1. Table[x,y] is "=" if there is a grammar rule A ::= αxyβ.
2. Table[x,y] is "<" if there is a non-terminal symbol A where Table[x,A] is "=" and y is in $\mathcal{L}$(A). (We know we are starting a new A-rule)
3. Table[x,y] is ">" if there is a non-terminal symbol A where Table[A, y] is "=" and x is in $\mathcal{R}$(A). (Do the reduction to A before pushing y.)
4. Table[x,y] is ">" if there is a non-terminal symbol A where Table[A,y] is "<" and x is in $\mathcal{R}$(A). (Do the reduction to A as soon as possible.)

It should be easy to apply these rules to the grammar

S ::= Aab $\mathcal{L}$(S)={A, a, c} $\mathcal{R}$(S)={b}

A ::= aS | c $\mathcal{L}$(A)={a, c} $\mathcal{R}$(A)={S, c, b}

and get the table

	S	**A**	**a**	**b**	**c**
S			>		
A			=		
a	=	<	<	=	<
b			>		
c			>		

Problem: If we try to apply these rules to the grammar

E ::= E+T | T $\mathcal{L}$(E)={E,T,F,id} $\mathcal{R}$(E)={T,F,id}

T ::= T*F | F $\mathcal{L}$(T)={T,F,id} $\mathcal{R}$(E)={F,id}

F ::= id $\mathcal{L}$(F)={id} $\mathcal{R}$(F)={id}

we get the table

	E	**T**	**F**	**+**	*****	**id**
E				=		
T				>	=	
F				>	>	
+		< =	<			<
*****			=			<
id				>	>	

A weak precedence grammar is one where we can build the precedence table and have conflicts only between "<" and "=", and in such a way that we can still parse successfully. This means

a) There are no conflicts between "<=" and ">".
b) The right hand side of each grammar rule is unique.
c) If there are rules $A::=\alpha y\gamma$ and $B::=\gamma$ then we cannot have $y <= B$. This allows us to distinguish between possible handles.

Our common arithmetic grammar

E ::= E+T | T   T ::= T*F | F   F ::= id

is a weak precedence grammar. The precedence table is

	E	**T**	**F**	**+**	*****	**id**
E				=		
T				>	=	
F				>	>	
+		<=	<			<
*****			=			<
id				>	>	

We can parse expressions such as x+y+z and x+y*z

Problem: Weak precedence grammars have trouble recognizing errors; we often need to read well past the bad token before we recognize the error. As a result, they are seldom used in practice. This is just a step towards the derivation of LR(k) grammars.

Example: S::=a+x+E $\mathcal{L}$(S)={a} $\mathcal{R}$(S)={E,a}
E::=a+E | a $\mathcal{L}$(E)={a} $\mathcal{R}$(E)={E,a}

	S	**E**	**a**	**+**	**x**
S					
E					
a				=	
+		=	<		=
x				=	

If we try to parse a+a+a+a+a+a, the parser reads to the end of the string, reduces it all to an E, and then fails because this isn't the start symbol.