# Example Parser Tree:

E

E = E

| |

id id

The parser will take the tokens and create a parse tree from it the parser tree only need to be read from left to right at the leaf nodes

## Production

The rhs of a rule for example S -> aSb /e, aSb and e are 2 productions

## Context Free Grammar

If the lhs only have one nonterminal (not a token received from the lexer (expression etc.))

# Left most derivation (LMD):

S -> aS|Sa|a -> S can be replaced with 3 different things and ***in LMD the parser will choose the leftmost production*** (here it's aS) In ***Right Most Derivation (RMD) the parser will choose the rightmost production*** (here it's a)

If each rule only has ***at most one nonterminal*** (S in the example) ***or******the grammar is unambiguous then LMD = RMD***

# Recursive Grammar:

If at least one rule has the same nonterminal in both LHS and RHS (i.e., S -> aSb | S)

Recursive grammar will ***always generate infinite strings.***

(ab,aabb,aaabbb, ...)

# Non-Recursive Grammar:

If no rule has the same nonterminal in both LHS and RHS

(i.e., S -> a | b)

It will always ***generate finite number of strings.***

# Left Recursive Grammar (LRG):

If the LHS exists in the ***first part*** of a RHS production

(i.e., S -> Sb | b) (S -> bS | b is not left recursive)

# Right Recursive Grammar (RRG):

If the LHS exists in the ***last part*** of a RHS production

(i.e., S -> bS | b) (S -> Sb | b is not right recursive)

# General Recursive Grammar (GRG):

If the LHS exists in a RHS production ***but not in the first or***

***last position***

(i.e., S -> aSb | c)

# There are 2 types of recursion:

**Direct Recursion**: LRG, RRG, GRG

**Indirect Recursion**: LRG, RRG, GRG

**Direct example**: S -> Sa | b

**Indirect example**: S -> Aa

A -> Sb

(S won’t derive to itself directly but through A)

# Convert Left Recursion Grammars to Right Recursion Grammar:

Introduce a ***temporary variable.***

A -> Aa | B (Left Recursion)

A -> BA'

A' -> aA' | € (Right Recursion)

--------------------------------

A -> Aab|c|d (Left Recursion)

A -> cA' | dA'

A'-> abA' | € (Right Recursion)

--------------------------------

S -> SaS | SbS | c (Left Recursion)

S -> cS'

S'-> aSS' | bSS' | € (Right Recursion)

--------------------------------

E -> E + E | E \* E | (E) | id (Left Recursion)

E -> (E)E' | id E'

E' -> +EE' | \*EE' | €

# Ambiguous Grammar:

If *a* ***grammar both have left and right recursion*** (S -> SaS | B) (they don't have to be in the same production)

S -> AB

A -> SA | a (Right Recursion)

-> Ambiguous

B -> BS | b (Left Recursion)

# Grammar with Common Prefixes:

If the RHS of more than one production starts with the same sequence of symbols (i.e., S -> ab | abc | ad)

# Left Factoring:

The process of removing common prefixes from the grammar

S -> ab | abc | ad (We take the longest common one so ab)

S -> abS' | ad

S'-> € | c | (We introduced a new one, but now a is common)

S -> a S''

S'' -> bS' | d

S' -> € | c (Now there are no common prefixes)

# First:

The set of all terminals that may begin in RHS.

For example: S-> € | a | b First(S): {€,a,b}

S -> aS | bS | € First(S): {a,b,€}

If the first part of the production is nonterminal, then it will inherit the first of that nonterminal (If it has more non terminals after it will inherit all of them).

(S->ABC would inherit First(A, B,C))

### For example:

E->TE' First(E) = {T} -> nonterminal so we have to check First(T)

T->FT' First(T) = {F} -> -"- have to check First(F)

F->(E)|id First(F) = {‘(‘, id} -> **The First of E,T,F are all {’(’,id}**

# Follow:

Follow(A) is the set of ***all terminals that follow the right of A***

(If the LHS is in one of the RHS then the right symbol of the RHS)

S -> a | b | € Follow(S)= {$} (Dollar is the default)

S -> Sa | Sb | € Follow(S)= {$,a,b}

# LL(1) grammar:

A grammar is LL(1) ***if the parse table for it doesn't have multiple entries***

## Bottom-up parser (shift/reduce parser):

The process of constructing the parse tree starting with the input string and getting the start symbol of the grammar.

## Top-down parser:

You start with the start symbol then work your way up to the string.

## Handle:

Sequence of grammar symbols that match the RHS of any production.

## Handle pruning:

The process of finding and reducing the handle

## Reduce:

Substituting the LHS for one of the productions on the stack

# LR Parsers:

LR parsers take grammar with **no ambiguity.**

**May or may not depend** on lookahead symbol.

LR(K) -> K = Number of lookahead symbols

## LR(K):

Reverse of RMD

Scanning from left to right

## Procedure to construct LR Parse Table:

1. Obtain the Augmented Grammar.
2. Construct the canonical collection of LR(0) items.
3. Draw the DFA.
4. Construct the Parse Table.

## Augmented Grammar:

The grammar which is obtained from adding one more production (because we always must start with S)

### Example:

S -> AB S'-> S

A -> a -> S -> AB

B -> b A -> a

B -> b

## LR(0) item:

Any production which has '.' anywhere on RHS

### Example:

A -> .xyz -> 1 item

A -> x.yz -> 1 item

A -> xy.z -> 1 item

The rules above are **Non-Final Items** because there is at ***least one symbol after the '.'*** (**Shift**)

A -> xyz.

The rule above is **Final Item** because there are ***no symbols after the '.'*** (**Reduce**)

What is after the '.' that is what you are going to scan.

# Functions used to generate LR (0) items:

1. **Closure(I):**
   1. Add everything from input to output-
   2. If A -> ŁBx is in I and B -> x is in grammar G then add B -> .x to closure of I  
      Input (A -> Ł.Bx) -> CLOSURE -> Output(A -> Ł.Bx, B -> .x) ( We needed to add B -> ,x because B is a non-terminal)
   3. Repeat step b. for every newly added item.
2. **Goto (I, x)**
   1. The Closure of A -> ŁB.x such that A -> Ł.Bx is in I

I (A -> Ł.Bx, B ->.x) -> GOTO(I, B) -> I (A -> ŁB.x)

Example:

goto(A -> Ł.Bx, B) = A -> ŁB.x (Just moved the ‘.’ to the right)

# Example LR(0) item generation:

Grammar: A -> aA/b

1. **Augmented Grammar**

A’->A  
 A->aA  
A->b

* 1. **I0: Closure(A’->’.’A)**

A’->’.’A  
A->’.’aA  
A->.b

* 1. goto(I0,A)=I1   
     I1:A’ -> A. (moved the dot to the right)  
     goto(I0,a)=I2  
     I2:A->a.A  
      A->.aA  
      A->.b  
     goto(I0,b)=I3  
     I3:A->b.

Now we must repeat the gotos with I2 because that still has Non-Final Items (there are productions where the ‘.’ is not the final), while in the others we don’t  
  
The Canonical Collection of the Grammar:

C= {I0, I1, I2….}

1. **Draw the DFA**
2. **Construction of LR Parse Table**

# Conflicts in LR(k)

1. SR-Conflict (**Shift-Reduce Conflict**): If any state hast ***both shift and reduce option***
2. RR-Conflict (**Reduce-Reduce Conflict**): If any state has ***more than one final item***

# Relation Between LL(1) and LR(0)

Every LL(1) Grammar need not be LR(0) Grammar.

Every LR(0) Grammar need not be LL(1) Grammar.

(Sometimes the grammar is both, sometimes it’s only one of them)