Syntax analysis parsing（数据处理）token sequence来辨别程序的syntactic（句式上的）结构

Parser 的输出是一种对程序结构intermediate representation//中间阶段的展示，通常是个parse tree（分析树），用基于formal grammar的树结构来代替analytical analysis后生成的token串//formal grammar用来描写语言的语法

通常使用context-free grammar来循环的描述语法结构，这样才能组成能运行的程序

作为结果的parse tree然后被分析，augmented(扩张)，并在compiler中被转换成之后的形式

Parser可以手写也可以由parser generator生成

SYNTACTI ANALYZER 的 Roles:

分析程序的结构以及他的组件declaration, definition, statement, expressions

检查语法错误

驱动前端的执行

Chomsky使用generative grammar来完成了grammatical analysis

使用的是Generative grammar:

主要目的：通过一些步骤来建立sentences:

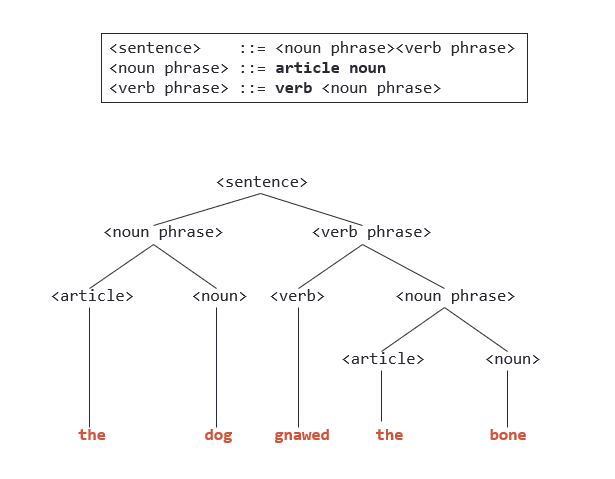
第一步是用一组语法规则来描述一些抽象概念//什么是Noun什么是verb这种

Productions:用来描述抽象概念的一组grammatical rules

Refine提纯:把analysis细化成一些lexical词义上的元素

语法分析（parsing）包括了构建规则**Productions**，可以用这个**production**s来构建句子

一个valid sentence被以parse tree的形式表示。



上面的是rule

句子被分解成：名词形式（主语）+动词形式

名词形式被分为 冠词article+名词

动词形式被分为 动词+名词形式（宾语）= 动词加冠词加名词

然后这个句子就被切割成了一组lexical element

SYNTAX:描述**怎样**构建一个有效的sentence //语法上的

Semantics:描述有效sentence的**意义 //实际意义**

有时候一些语法正确的句子可能没有意义

例如the bone walked the dog

不可能完全验证英语句子是否在意义上有效

因为英语的句子有时是含混的ambiguous

程序语言必须是non-ambiguous，非含混的

在程序语言中，semantics通过把程序转译成exe来描述意义

Grammar:

一个grammar是quadruple(由四个元素组成) (t,n,s,r)

T:一组 有限terminal symbol //小写字母也是最终形式

N:一组有限non-terminal symbol //大写字母，中间形式，也是开头

S:一个独特的starting symbol，属于N

R:一组有限的productions// the rule to transform A to B //转化规则

Context free grammar通常都有以下的形式



如果有A，那就能transformed到B

A属于非终端terminal,B有可能是非终端terminal，也有可能是terminal symbol，或者两者结合

词语就是terminal symbol //最终目的

<verb>就是non terminal symbol

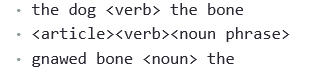
Terminal symbol就是我们要的最终形式（通常小写） //例如dog

Non terminal就是初始形式或中间形式（通常大写） //例<noun>

A->B

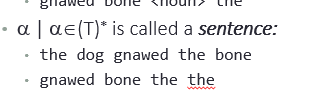
B->y

**Sentential form**: 既有T又有N //中间态



完全由N组成也是合格的sential form

**Sentence:**完全由terminal组成 //最终态



Backnus-Naur Form

这两个都是人名

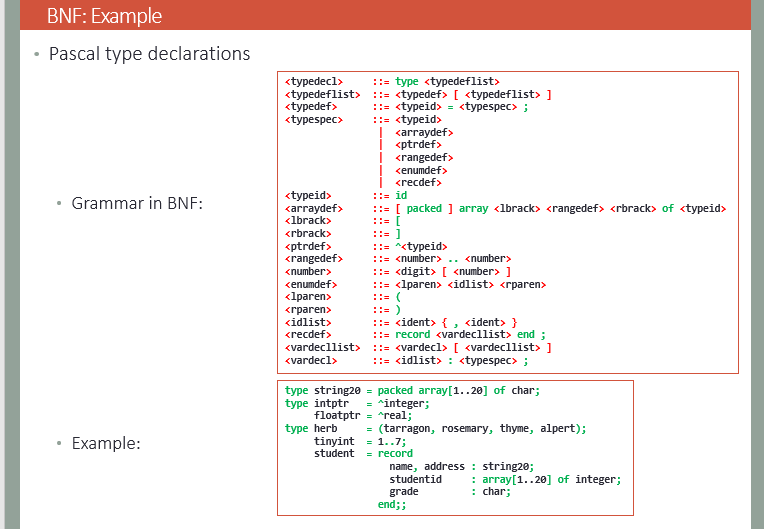
Non-terminal被放在方括号里

用::==来代替箭头//推出符号

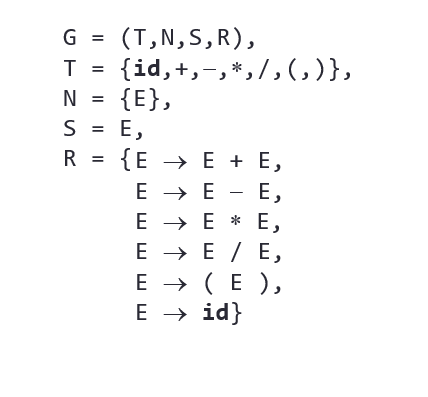
Vertical bar竖线|用来表示**alternative或**

Curly brace圆括号用来表示无限次的**重复Repetitions**

Square brackets方括号来表示**可选择的Optionality**



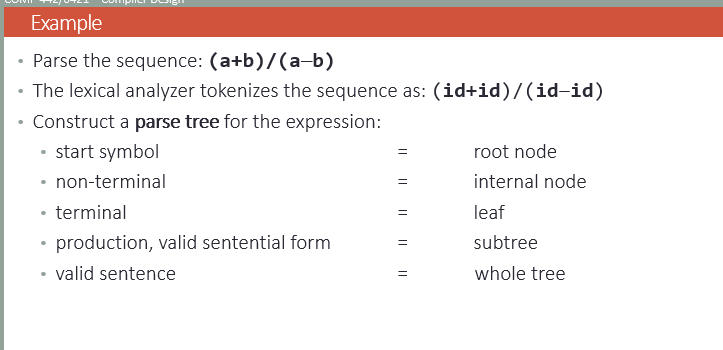
简单的arithmetic expressions //四则运算的context free grammar表示



N: non-terminal symbol

T: tokens //terminal，也就是最终形成的token

R:relation



目的：我们要分析句子(a+b)/(a-b)

Lexical analyzer把这句话处理成token串 Id+id / id – id

在syntax analyze中：我们希望构建parse tree

Root是start symbol S

中间过渡的internal node是non-terminal

Leaf是terminal

中间产物，valid sentential: 子树

Valid sentence:整颗树

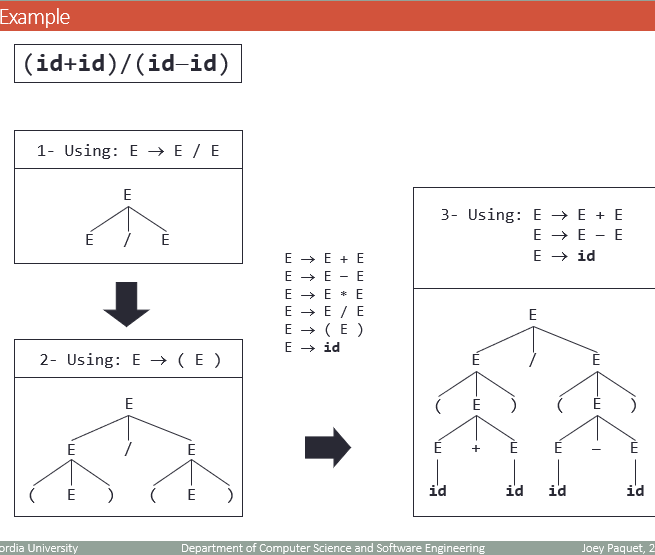
**Top-down parsing**

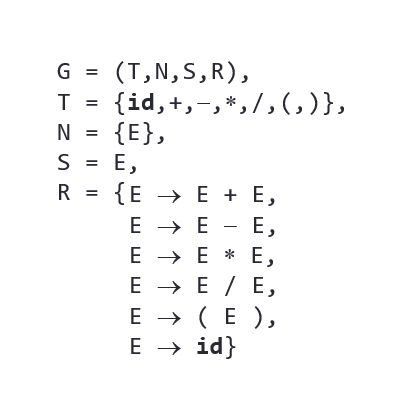
从root开始也就是starting symbol

想向下build tree，同时接受以下两种input形式

第一种input:一串token，从左往右 //原始处理数据

第二种input:grammar的规则 //处理规则





初始是个E(E永远是初始root)，（图示方法不是标准的，更像直觉）//程序上是不停地测，如果不对，回退，直到测出结果

我们的输入是XX/XX

所以第一步E变成E/E

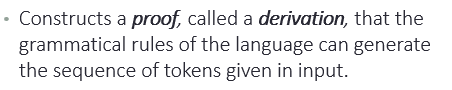
然后加上括号 E->(E)

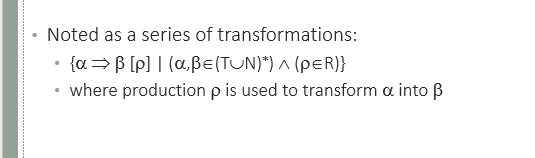
然后左边变成E+E,右边变成E-E

然后E变成id

但是上面那个方法是靠直觉的

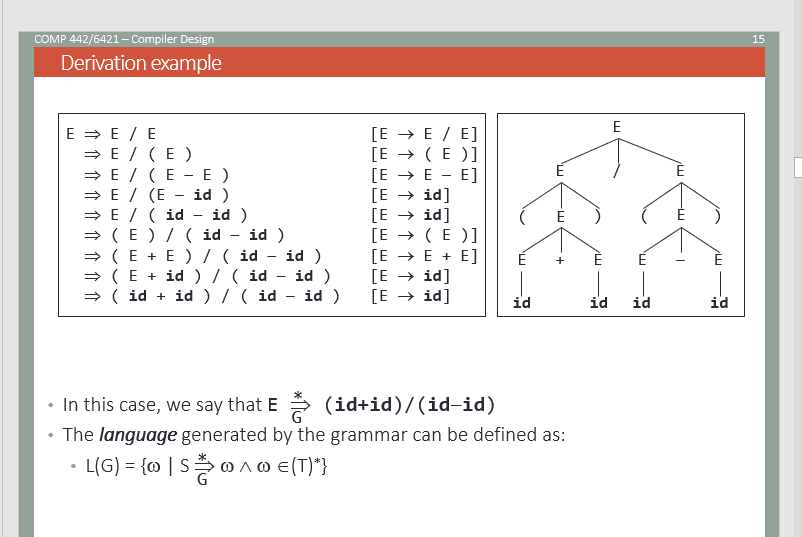
**Derivation推导：**构建一个有效证明的过程，在这里指利用语法规则用给定token生成一个句子





合理的derivation过程：

利用一系列语法规则ρ，把开始的E一步一步转换成sentence

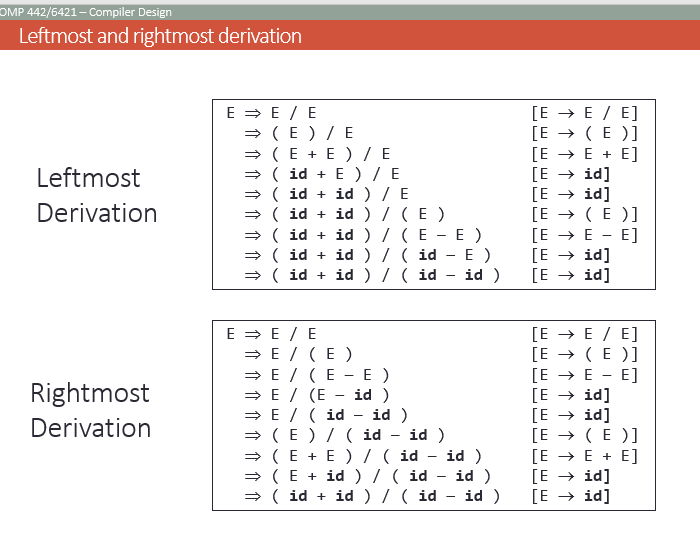


E通过我们的grammar能推导成（id+id）/（id-id）的形式

这个grammar生成的语言可以描述成



S经过grammar能得到的所有terminal组成的集合，就是这个grammar的language



Derivation又分为leftmost derivation, rightmost derivation，

区别在于趋向于process处理哪一侧的non-terminal symbol

左边E/E

处理左边non-terminal

(E)/E

左边仍然是non terminal

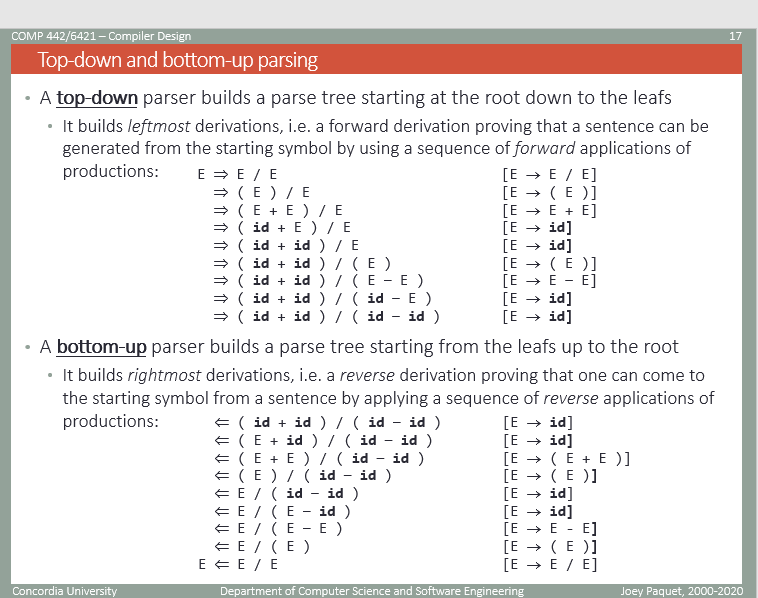
(e+e)/e

Id+e/e

Id+id/e

Id+id/(e)

…………



Top-down parser从root自上往下构建到leaf，按照的是leftmost derivation,

Top down 是一种前向推导，通过一个starting symbol，来生成一个sentence,前向是因为它使用规则是顺序的

Bottom-up从leaf构建到root来创立parse tree

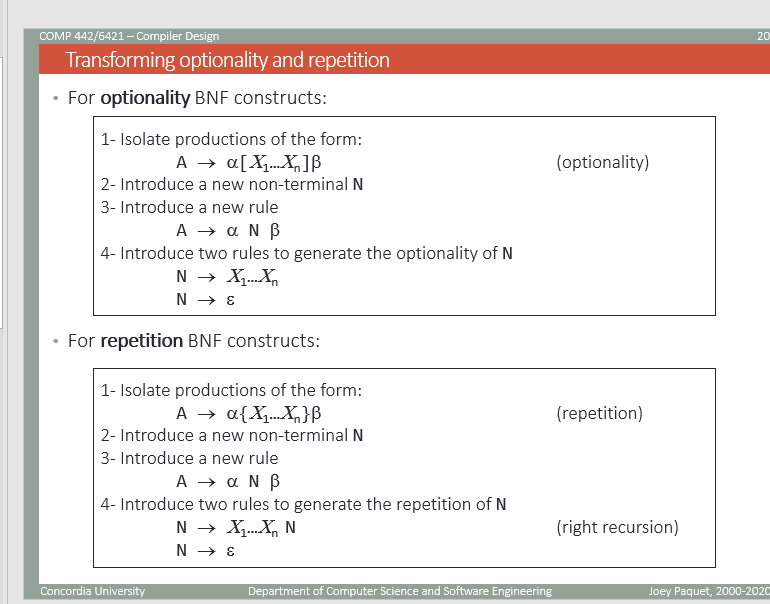
Bottom-up这里使用的是rightmost derivation，但看起来反而是反的，这就是bottom-up一个要注意的地方，同时它运用的grammar也是逆向的

Grammar transformation

Extended BNF这个构建方式支持：选择optionality也就是’方括号’ 与重复也就是圆括号

用来表明阐述grammar很合适

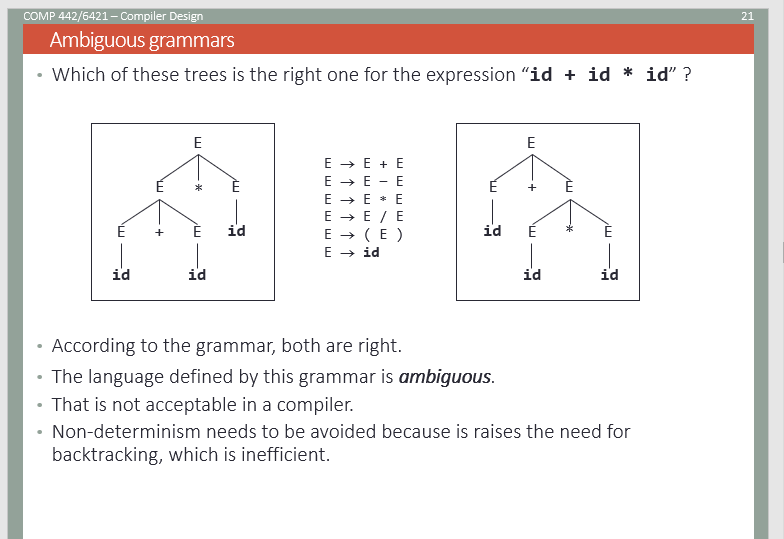
但是还是被移除了，因为他们与标准生成parsing technique不兼容



第一个是选择性：你可以生成ax1x2..xnB,也可以生成AB

第二个是repetition：你可以生成ax1xnx1xnx1xn….b

Ambiguous grammar含混的语法



根据语法，左右两个图都是对的

这个语法描述的语言是含混的

这对compiler来说是不可接受的

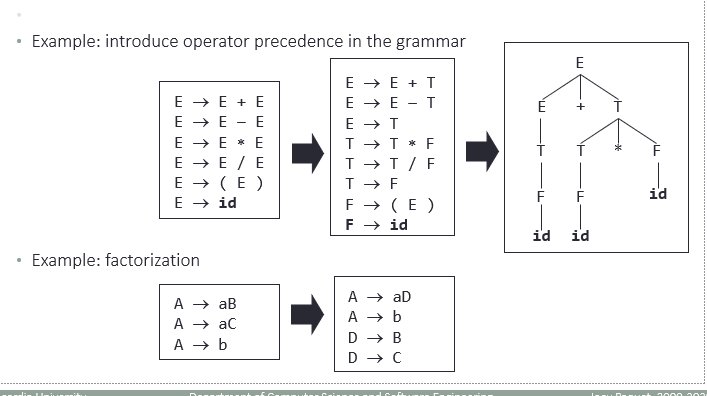
Non-determinism （非唯一解）需要被避免因为我们有时候需要backtracking回溯

解决含混性：

将操作优先级整合到parser中（让compiler更为复杂，很少这么做）

或者实装backtracking(让compiler复杂，效率低)

更改语法（优先解）//就是这一串规则



Factorization:因式分解

把A->aB A->aC转换成

A->aD

D->B

D->C

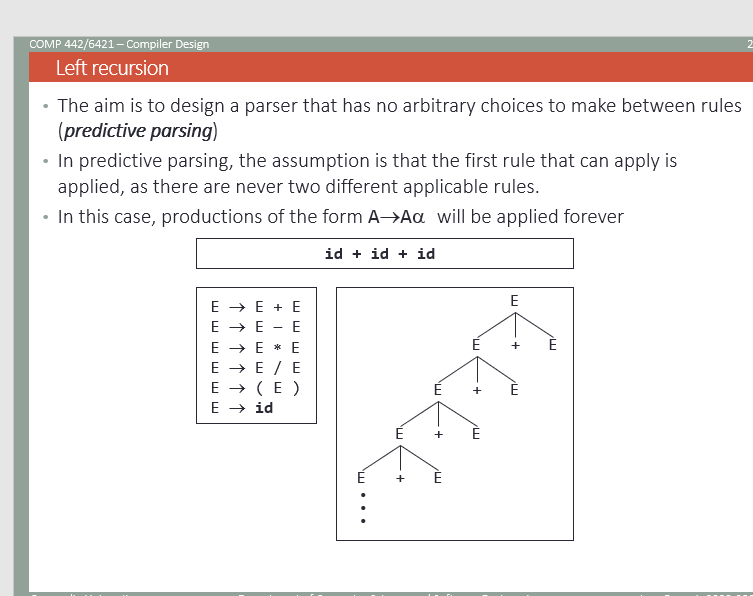
Left recursion：

设计parser的目的是不作出选择（predictive parsing）

Prective parsing中，假设如果我们可以使用第一个rule，那就用第一个，永远不会同时有两个不同的ruLe

Left Recursion的具体形式

A->Aa|…



E转成E+E,然后转左边的，

那么我们就会不停地用E->E+E而不会用E->id

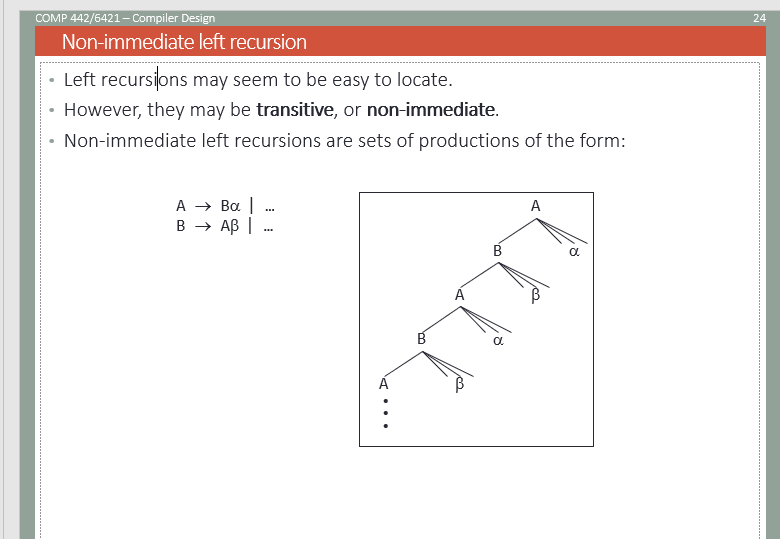
这个叫做non-immediate left recursion,仍然是left recursion的一种

形式是

A->Bα|…

B->Aβ|…

就是换了一次仍然是A->A,,,,



Left recursion容易发现，

但是non-immediate left recursion会transitive或non-immediate//不停循环

Left recursion

A->Aa

Non-immediate left recursion

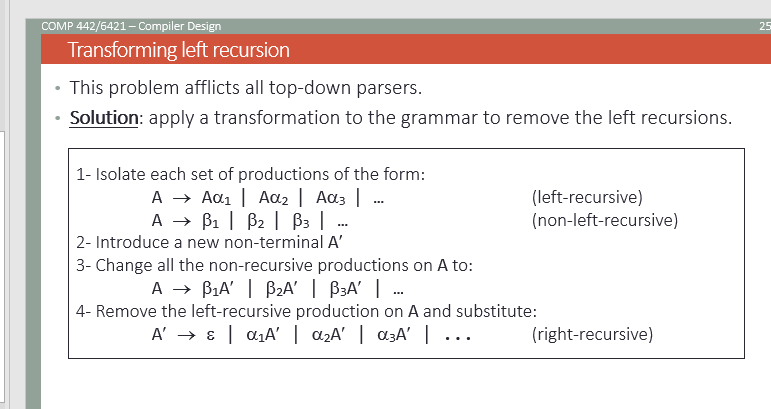
A->Ba

B->Ab

区别在于左边是不同的大写字母，然后经过循环步骤又能绕回初始字母

所以我们要改变left recursion

假设有一个left Recursion为A->Aα|β



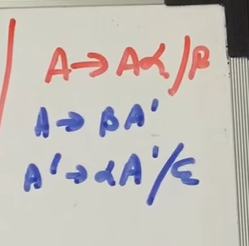
第一步：把left-recursive与Non-left-recursive（注意这里不是B，而是β，已经合格了）分开

第二步引入新的non-terminalA’(大写字母)

第三步，把Non-recursive productions加上新字母

A->B1A’。。。。。

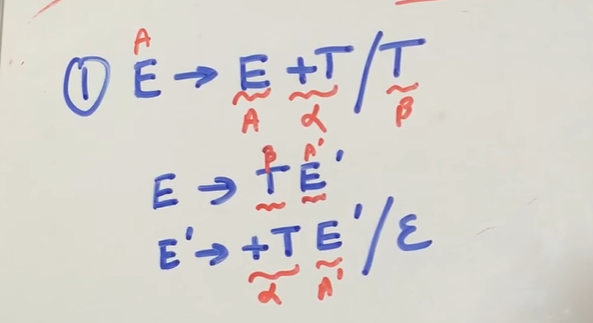
**其实就是公式**



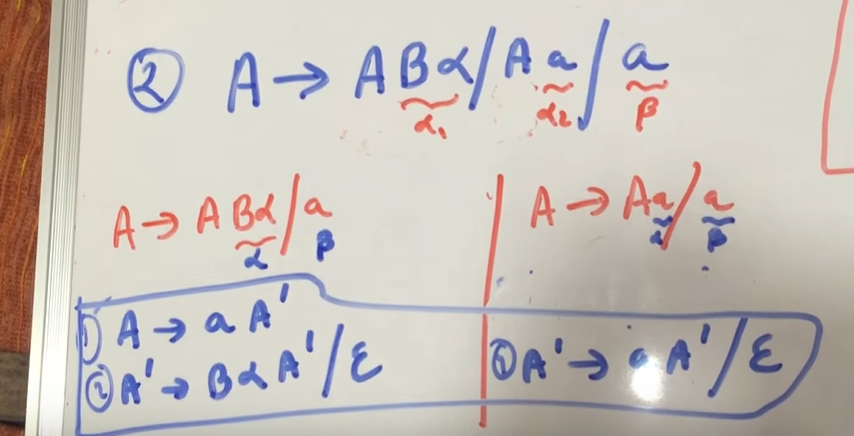
第一步，找到β找到α

第二步引入A’建立β公式 A->ΒA’

第三步A’在左建立α公式 A’->Αa’/e

例子

α并不是一个单纯terminal，只要不是A的整个combo例如+T就是α



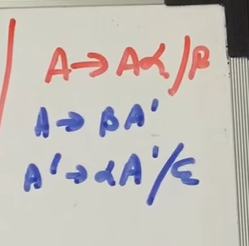
这里Bα是 α1, α是α2， α是β

第一步A->α A’

然后α1代入公式第二步A’->BαA’

然后α2代入公式 A’ -> α A’/e

**为什么能成功的原理**

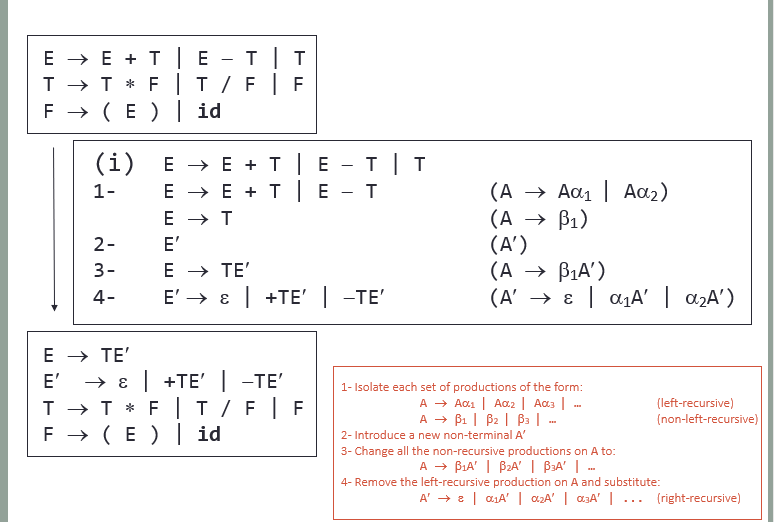


例如观察这个式子，不管左边再怎么循环，最后得到的词语一定是βα\*

而且β有且仅有一个

因此第一步先把β用了

然后然后不停地换α直到你想停止变成空集



课件例子：

E->E+T|E-T|T

显然T是β ，+T -T是我们的α1 α2

Step1:E->T

Step2:E’->+TE’|-TE’|e

然后处理T

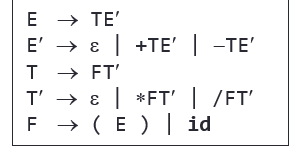
F是β，\*F 与/F是我们的α

T->F

T’->\*FT’|/FT’|e

F已经OK了

所以最终结果是



Non-recursive ambiguilty

这里的含混指的不是循环，而是不知道选哪个

以这个A->αβ1|αβ2|…为例

parser并不能找到我们要的β，因为开头的字母α是相同的

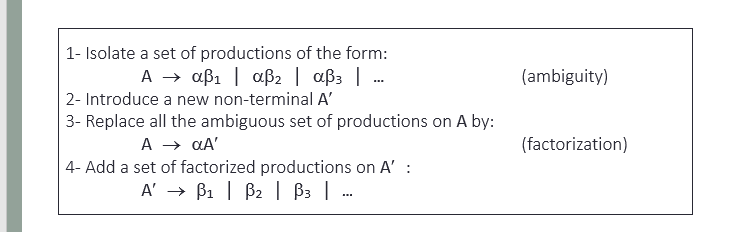
尽管是端点，还是含混的

但他们都是以α开始的，含混的地方只是在于β

所以我们先把α确定好

然后用字母A’来代替含混的β123

然后parser就可以处理不同的B1B2B3（因为是第**一**个**不同**字母）



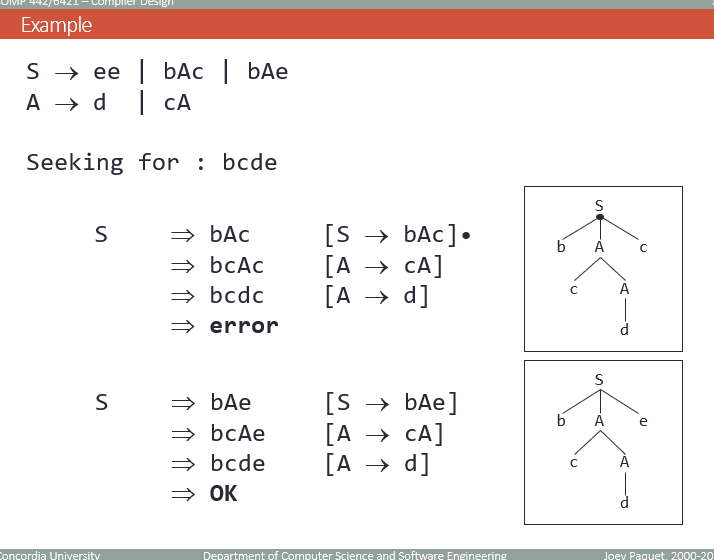
Backtracking

有可能利用含混的语法写一个parser

在这个例子中，由于有不同的可能，parser一个一个慢慢展开

如果没有展开我们需要的parse tree，大不了回溯，然后找另外一种展开可能

这叫brute-force method



我们想生成bcde

S->bAc

bcAc

bcdc

错了

然后我们会退到第一步然后S

这种方式是可以的，但是更慢，这里比较简单，随着tree越来越复杂，回溯会越来越繁琐

Backtracking

风险大，难实施

但是最近发现了catching的技巧，减少了生成的subtree，让backtracking成为可能

Predictive parsing:

Parser必须决定使用哪一个right-hand sides来继续//就是A|B|C这种选一个

然后唯一已知信息就是下一个token

是一种没backtracking，没含混性的top-down parsing

Recursive descent predictive parser

递归 下降 预测 分析器

每一个non-terminal symbol大写字母都会有一个function

可以决定使用正确的right-hand-side

怎么选择正确的right-hand side ，在function里面用if语句以及**FIRST** | **FOLLOW** set

每个function都能识别terminal symbol并call other function来识别对应的non-terminal symbol

Parse tree就在nesting of function calls中建立

容易实施

编码工作量大

难以维护

语法的改变导致代码的改变

Table-driven predictive parser

使用一个parsing table来告诉parser来使用哪个righthand-side

对所有parser有一套标准的parsing algorithm

只有language改变的时候，只需要改变table，算法本身是恒定的

容易为辅

如果人工写parsing table，很难，而且很有可能产生error

有工具生成parsing table

下节课会讲

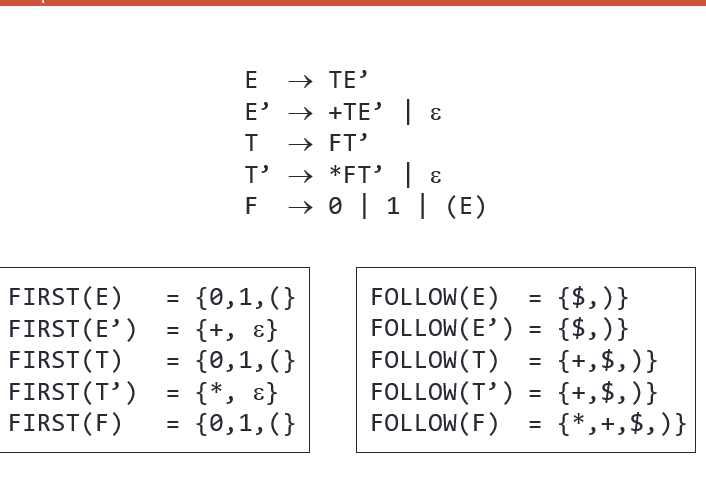
First and Follow sets

当parsing试图处理Non-terminal symbol决定使用哪个right-hand-side的时候

如果所有的right-hand-side开头都是terminal symbol小写字母，选择是很简单的

如果right-hand-side是大写字母开头，那么parser需要知道这个non-terminal最终会产生怎样的小写sequence，我们不可能一下得出所有sequence，但是我们能较快的得出所有可能的sequence的**第一个terminal的可能性**（FIRST集合）

如果FIRST包含了e，那么我们就要找到non-terminal是怎样选择才能到达这个e选项（FOLLOW）//就是这个non-terminal慢慢转化的过程



E能达到的所有terminal开头：暂时无法看出，但肯定与T相同

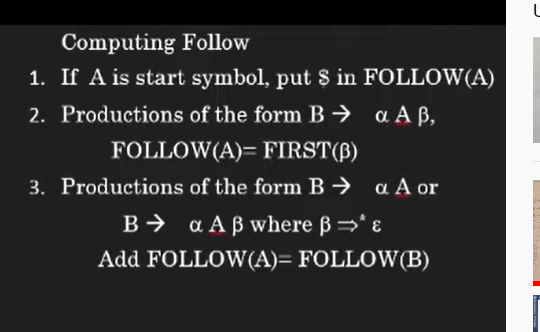
E’能达到的,+或者空集

T能达到的，无法看出，与F相同

T’能达到的，\*或者E

F能达到的0或1或(

然后倒推T和E



然后计算FOLLOW，

第一部分：只应用12与3的前半句

第一句：E肯定是开头，因此有一个dollar符号

然后E->TE’符合形式 α A, α是T ，A是E’

因此FOLOW (E’)=FOLLOW(E)

第二句：E‘->+TE’ ,符合形式2， +为α，T为A，β为 E’ ，因此

Follow（T）为 FIRST(E’) 为 {+} //为什么没有e，因为FOLLOW里不能有e，所有的e要舍去

第三句：T推出FT’ 符合形式3， follow (t’)=follow(T)

第四句：T’推测出\*FT’,符合形式2， Follow (F)=FIRST(T’）={\*}

第五句：F推出(E)符合形式2， follow(E)=first()),因为）是一个terminal

因此FOLLOW（E）={$}U{ ）} //follow e加上这个右括号

第二部分，处理空集

第二句 ：E'->+TE' ，因为E'是空集，所以符合定律3的后半部分，

因此FOLLOW （T）=follow(E'),注意这里是add关系，就是{+}add E'的部分，并不代表E'会被改变

第四句， FOLLOW F被 add follow (T')

所以E很好确认，开头加上第一部分第五句， 为DOLLOAR )

E'决定他的就是ADD FOLLOW E, 所以完全等于E

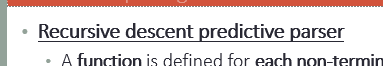
T在{+}的基础上根据第二部分第二句，add follow E' .因此是 +$)

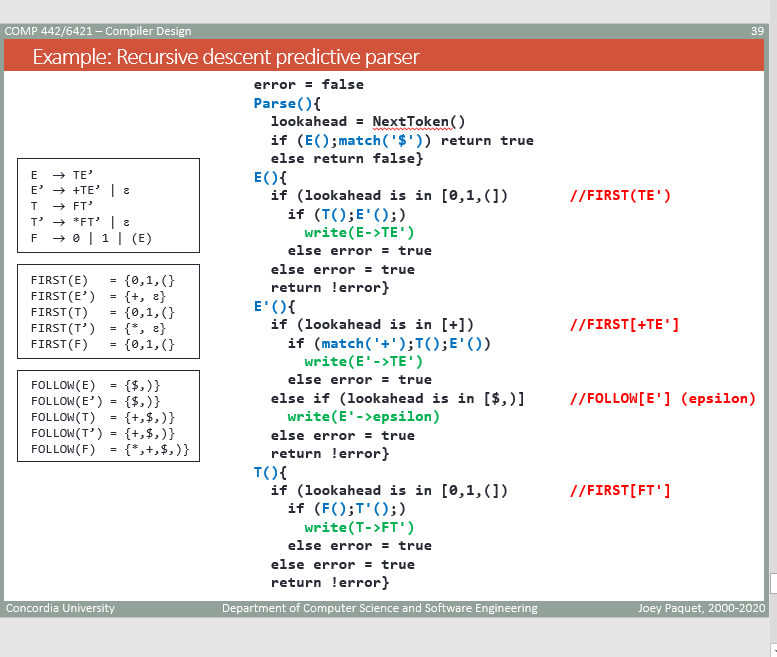
T' 就是单纯的FOLLOW T

FOLLOW F 是\*的基础上 add follow t' 为 \*+$)

那么我们得到了FIRST FOLLOW SET

就可以进行





为每一个大写字母构建function