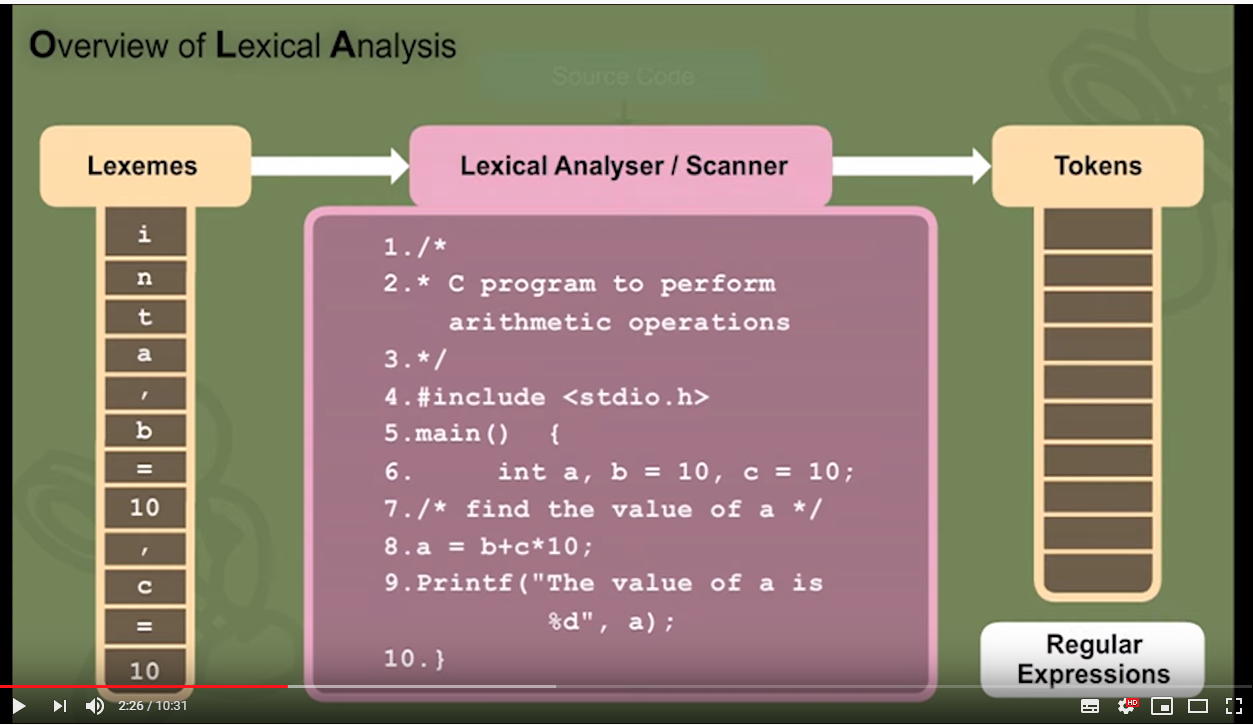
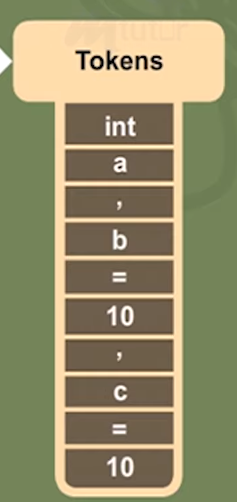
Lexical词汇的

Lexical anaylsis 是一个把一串char转化成一串token的处理过程



这些最原始的char叫做lexemes，通过Scanner，转化成有词义的token



一个用来进行lexical anlysis的program或function叫做Lexical analyzer,lexer或scanner

Scnnaer必须有一个叫做parser的function，用来提取源代码中下一个token, //parser提取器，解析器

一个编程语言的lexical specificcation通常由一组描述scanner的规则所描述。这一系列规则可以被lexical analyzer generator例如lex或flex所理解，所以通常是regular expressions

Lexical analyzer通常自动生成由lex这种工具，或者手写，阅读源码，分析词汇，分类城token，输出token stream

这个过程叫做tokenizing

如果scanner找到invalid token，会报lexical error

第二页

Scanner的作用：

首先会removal of comments

因为comments不是program的一部分

Multiple-line comments

Comments in comment //使用counter，

/\*

\*

\*

\* /\* \*/

\*/

第三页，remove空格,tab，回车

有可能不用空格鉴别token吗，可以，通过关键词类似1<这种符号

对编译器指令的解释

#include #ifdef #ifndef这种都是指示，告诉compiler重新定位Input开头

可能是由pre-compiler完成的

Symbol table的初始创建

每一个identifier(ID)创建的时候都会给他一个对应的symbol table entry

Lexical analyzer不能创造一个完整的entry

可以把文字转换成对应的值并且附一个值

把Input file会转换成token stream

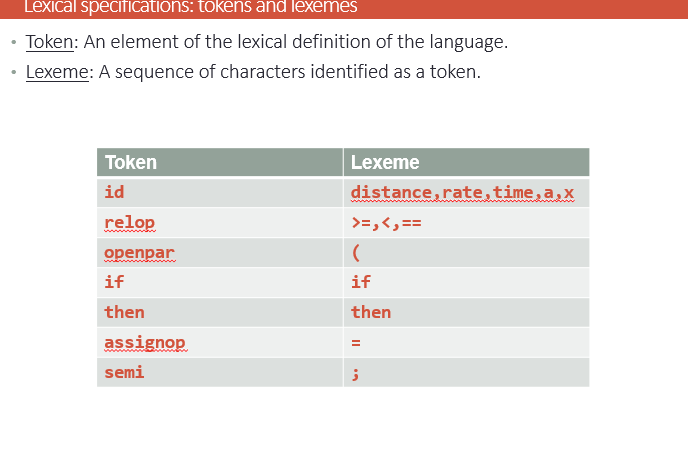
Input file一开始是一串char

Lexical specifications:

词汇规范:文字、操作符、关键字、标点符号

Token, 你语言里的一种词汇lexical定义 //char能形成的类别 （类）

Lexeme：一串char被认作是一个token //那些被分好类的一串char （实例）



设计lexical analyzer

过程：

构建一组regular expressions来描述任意形式的有效token

把RE转化成NFA

把NFA转化成DFA

把DFA转换成state transition table，state表格

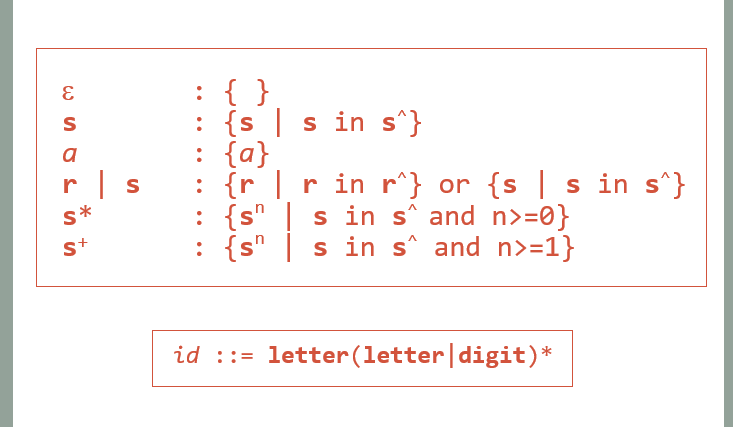
构建table

构建能读取table的算法

这也是一个scanner generator的过程，所以你使用scanner generator实际上是一样的

例如lex,flex等等

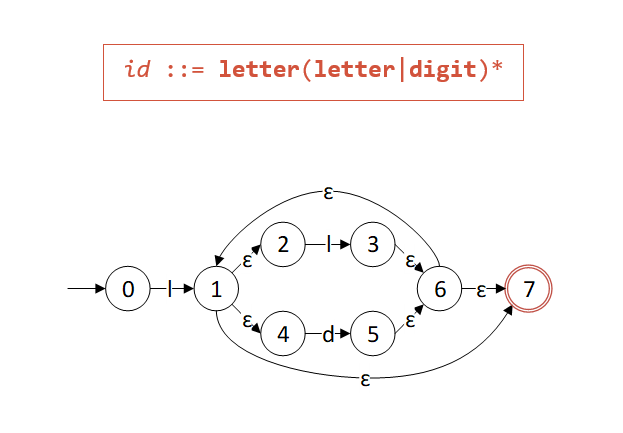
Regular expression:就是集合，描述了每种set由什么构成



S^,那些可以由definition of s构建成的string

Id^, 可以由id这个definition构建成string

Tompson’s construction :一个算法把regular expression转化成NFA



L是letter,d是digit，e是空

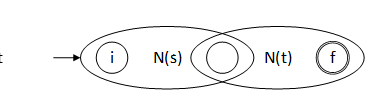
其实不写e也是合格的NFA，有e是因为这是thompson算法所产生的结果

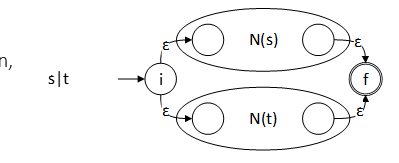
Thompson通过recursively把一个expression分解成子expression

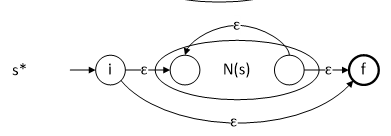
每一个subexpression对应一个子图

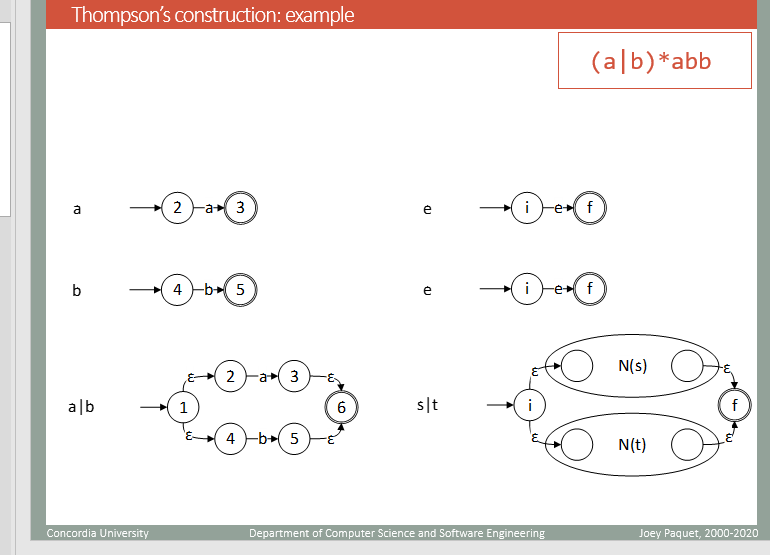
Subgraph被如何组合取决于链接方式，

St（直接连接）

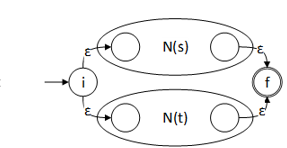


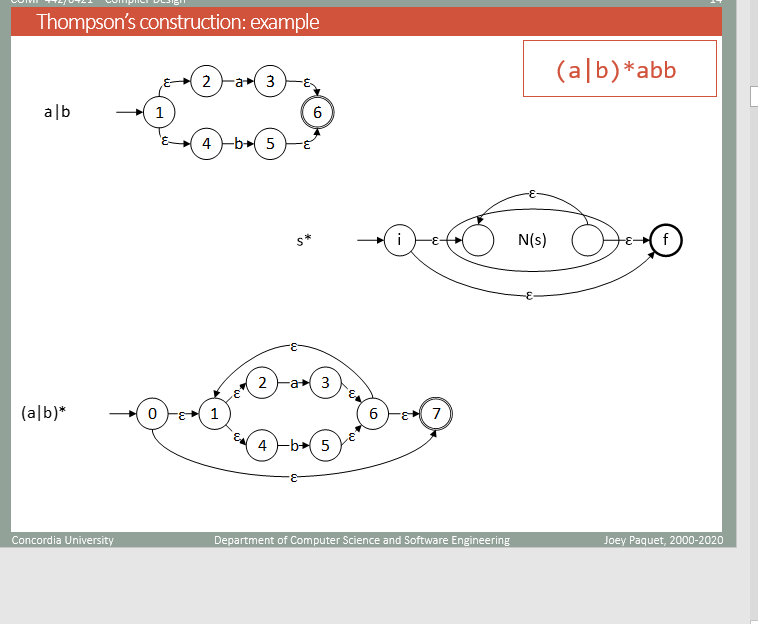
S|t（或关系）

S\*（循环）



Ab之前都是两种状态，因为产生a的状态与产生b的状态不是一种状态

A|b是s|t生成方式



然后再把a|b代入s\*

中间那种空的小圆就是start,end，用来给你嵌套

这个算法是把regular转化成NFA

Rabin-scott

把NFA转换成DFA

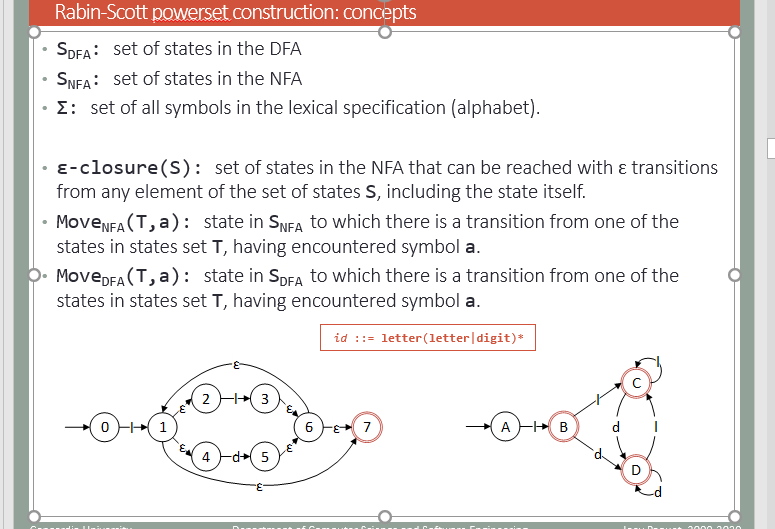
在这里的理解是把一组NFA的state转换成一个新state

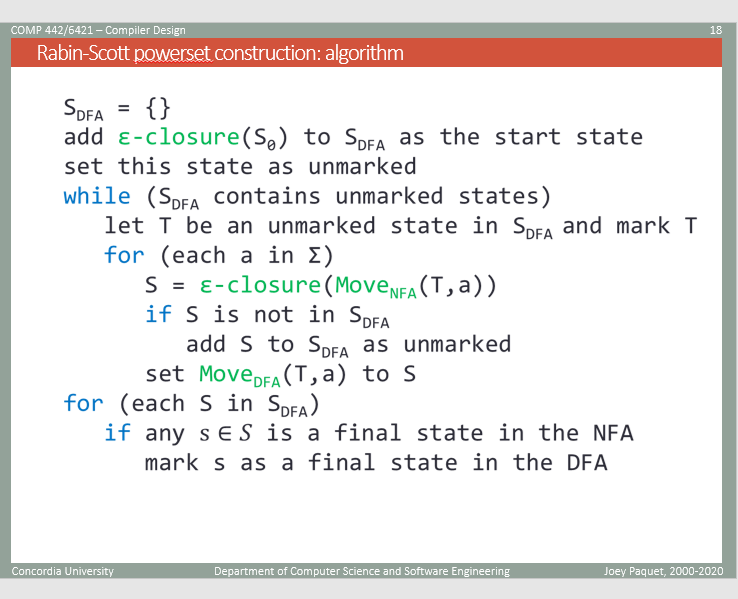
SDFA: DFA的所有state组成的set

SNFA：NFA所有state组成的set

alphabet

S是一组NFA的state组成的set,（不是全集） 通过e-closure这个操作符以后可以到达的新set of states //这个操作符指通过e 空transition //这时还是一组NFA STATE，然后我们添加一个大写字母来代表DFAstate





流程：

一开始DFA的state集合是空的

然后把e-closure(S0)加到SDFA里作为start state

新加入的state都设置成Unmarked

while DFA里存在 unmarked时候

E-CLOSURE这个unmarked state （这时还是DFA的单独state(也是一组NFA STATE）

创建一个新的DFA state S, 他等于unmarked 展开后每一个单独NFA STATEe closure以后再集合，如果新的S不在 SDFA里，加入SDFA并标记为unmarked,

不管怎样，都吧测试过的标记为Marked

一直循环

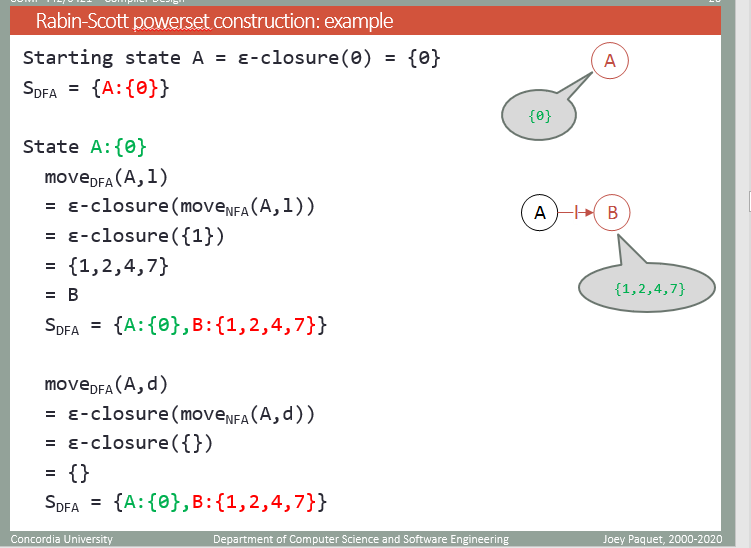
直到所有的都是Marked

然后如果任意DFA里的state包含的子state有·final state，那他就是DFA里的final state

例如124到B

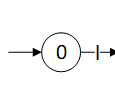
Move NFA,例如2到3，move了一个l

Move DFA,例如B到C，move 了一个L



Red is unmarked, green is marked

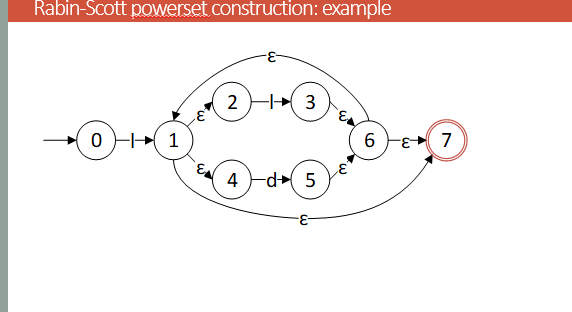
第一步给Sdfa一个e-closure S0作为start state

E-closure(0)代表从0经过e所能达到的state的集合，这里显然还是0

所以S（DFA）唯一state就是{A:{0}}//state名字为A，所代表的是原NFA里的几个状态的集合，这里集合只有一个元素0

然后对于每一个alphabet中的元素试图进行moveDFA

实际上就是对原来的集合e-closure进行moveNFA



从0只能移到1，

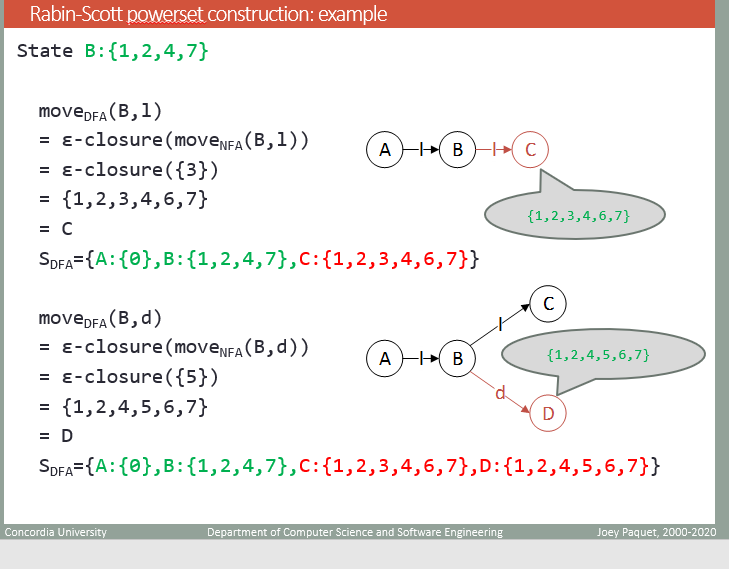
Eclosure{1}={1,2,4,7}

把它命名成B//B里有7所以B是终点之一

然后try d，哪都去不了

所以是空集

这里不添加

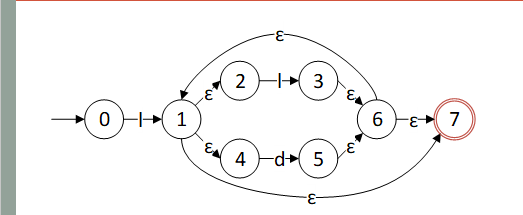


Move Bl

就是1247都试试

会reach 3

3 eclosure完以后会遇到6,6遇到17等等

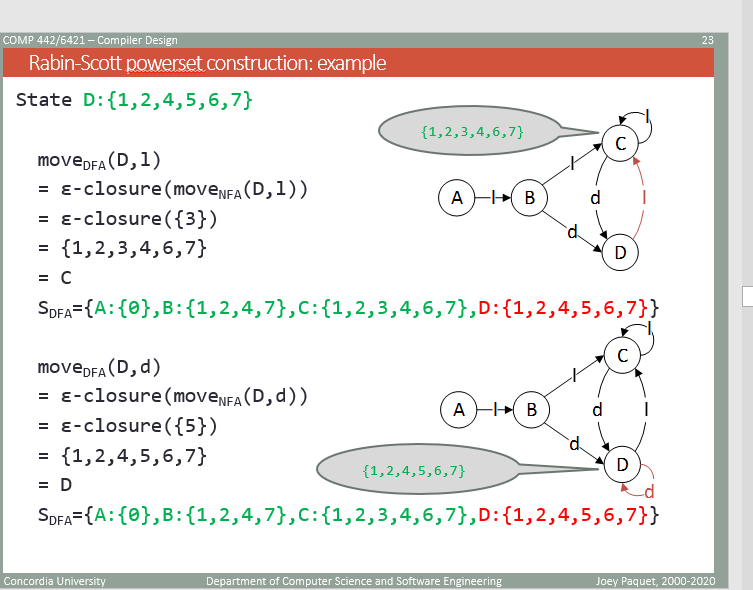


B到d, 1247加d，变成5,5遇到61….

这时CD是unmark

所以要对CD进行测试

C进行L是还是自己，C进行d是D，没有新东西，所以C是marked

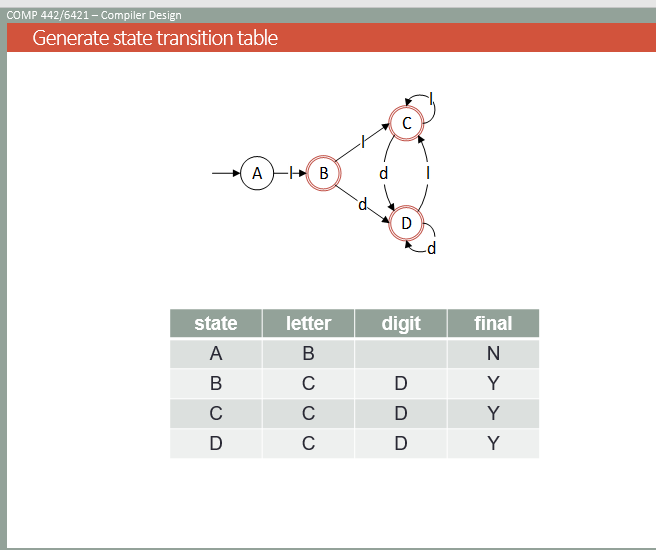


D经过l是C

D经过d是D

那么D是marked

然后找4个最终状态里面谁有final state



生成table(因为我们的algorithm需要用的是table)

Backtracking

一个token在下一个char被读到的时候，我们才能意识到这个token（例如int a =b ,我们要一直读到a，才能意识到int是个token）

但这个char已经是下一个token的第一个字母了 (例如chi shi,这时光标已经在chi s|hi了)

例如x<1,只有当1被阅读到才能识别<（因为有可能是小于等于），这是我们需要回溯一个char来继续token识别而不是跳过第二个token的第一个char，（chi |shi）

解决方案：把这些所有特例写到state transition table里

Ambiguity含混

有一些token的lexmes语义是另一部分token的一部分// 就是有些token（小的）可能也是另外一些token的一部分（大的），他们单独有效，组成一个combo也有效，不能武断的截断他们

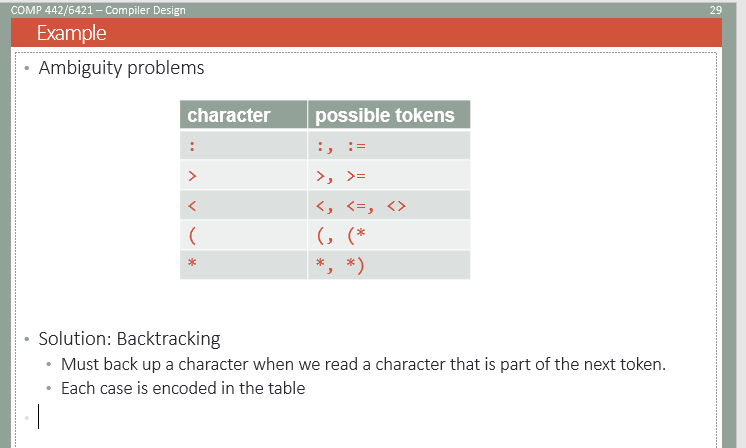
例如n-1,是n – 1还是 N -1

解决方案

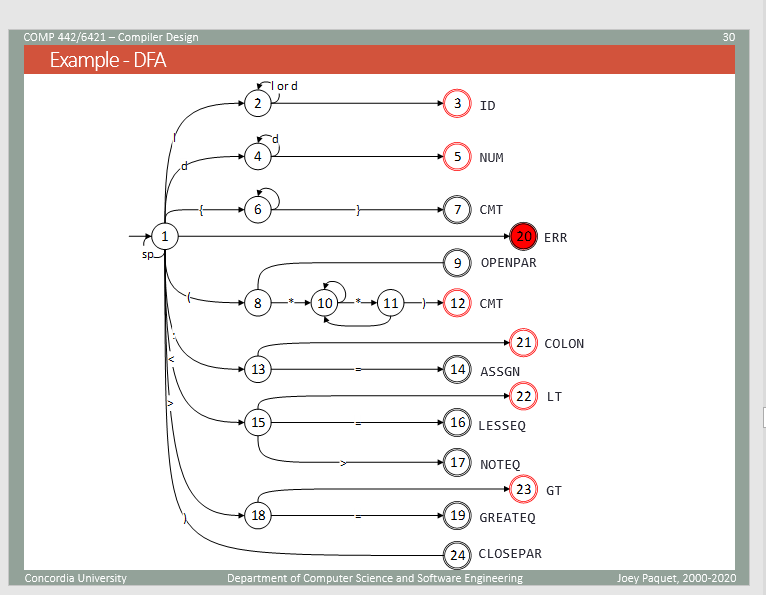
延迟这个决定，给syntactic analyzer

在lexical sppecification里面禁止给number机前面加前缀

与syntactic analyzer一起找出答案



表格中那些有可能有含混的我们都加了例子，如果不在例子里面，（读到下一个token了）截断，backtrack



Token的鉴别就是一个DFA

letter开头，然后L or D 就是ID，这些都不谈了

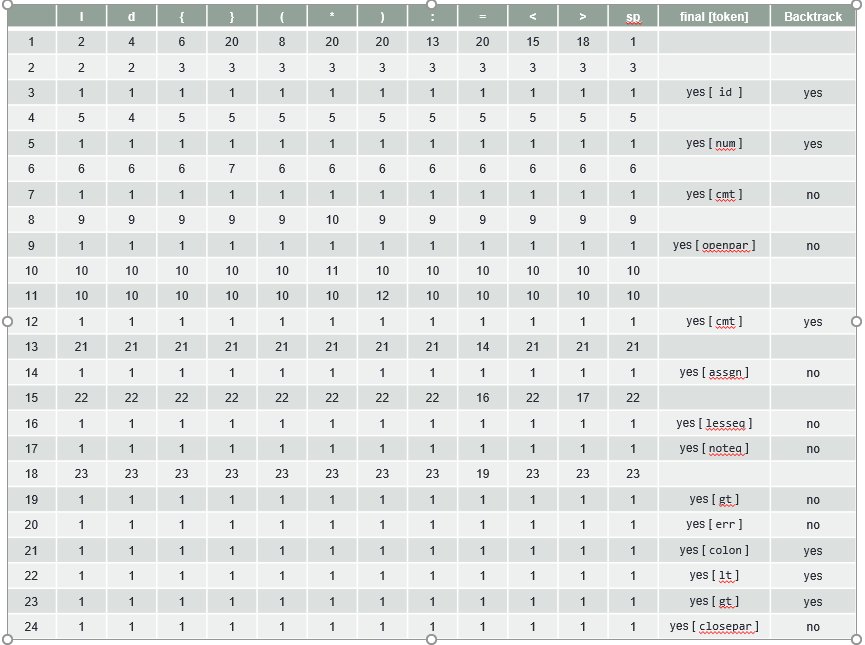
<开头的

这种就是ambiguilty

如果到下一个token前啥都没有，是lessthan

如果到下一个token前有个等于，就是less equal

这个DF会转化成state transition tabl3e



怎么看

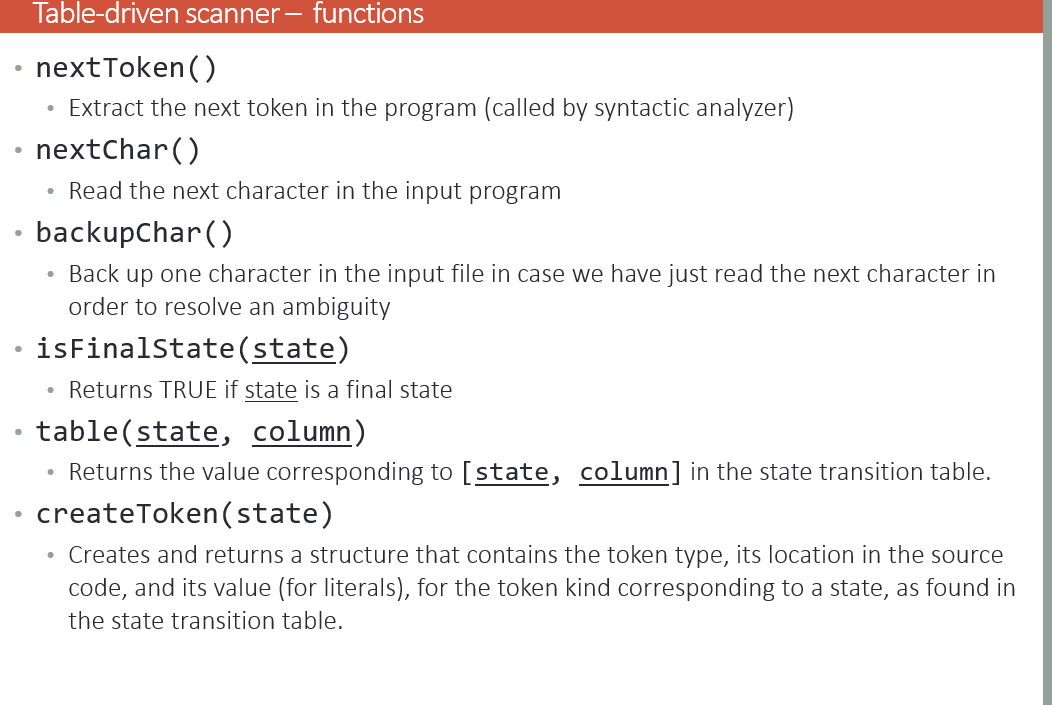
第一列是在哪个state

经过了第一行哪个输入token

就会转到对应的状态

最终到FINAL TOKEN

final token所有的都写1



Error Recovery in lexical analysis

不能只找到一个error就停止

Panic mode:跳过error chars直到下一个有效char被读到

guess mode:用一一匹配的方式在error与valid之间，但是很少用

结论：

lexical analyzer generator例如lex

简单快捷，但必须使用软件，没法handle罕见情况

table-driven lexical analyzer

方便易于使用，所有不同的table都可以用同一套analyzer

但是创建table可能出现问题

hand-written

容易建立

有可能有error，而且不方便维护，因为一更改就要写新的

为什么要把lexical analyzer 与syntactic analyzer区分开

modularity/maintainability 模块化，系统更加模块化，更容易保持

efficiency: 模块化代表更好的切分任务·

reusability:可以更改整个lexical analyzer而不更改其他部分