

中国科学技术大学《编译原理》笔记分享 (二)

前两章的笔记可参考:

[爱喝热水：中国科学技术大学《编译原理》笔记分享（一）](#) 45 赞同 · 1 评论文章

笔记目录

第三单元:词法分析(Part 2)

- RE转换成NFA: Thompson算法
- NFA转换成DFA: 子集构造算法
- DFA的最小化: Hopcroft算法
- 从DFA生成分析算法

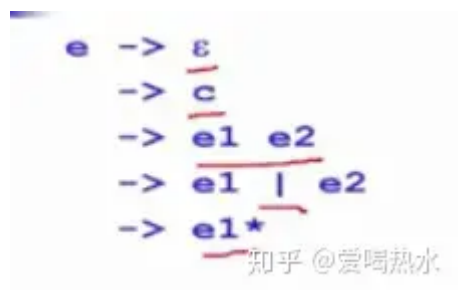
第四单元:语法分析 (Part 1)

- 第一讲: 语法分析的任务
- 第二讲: 上下文无关文法和推导
- 第三讲: 分析树和二义性文法
- 第四讲: 自顶向下分析
- 第五讲: 递归下降分析算法

三、词法分析 (Part 2)

RE转换成NFA: Thompson算法

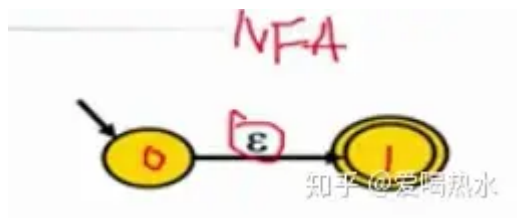
- 基于对RE的结构做归纳
- 对基本的RE直接构造
- 对复合的RE递归构造
- 递归算法, 容易实现。



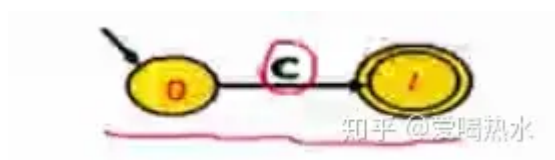
对上图的五种正则表达式，前两种可以直接构造。后三种需要递归构造

1.直接构造

第一种:从0号节点到1号节点不需要接受字符即可到达。

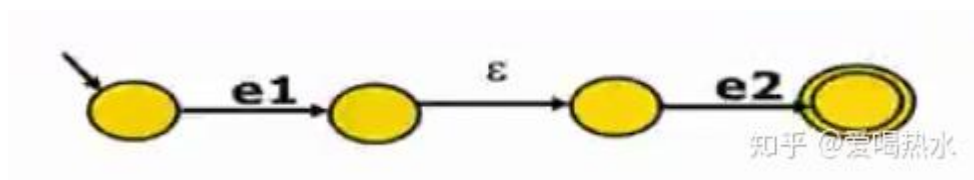


第二种:从0号节点到1号节点需要接受一个指定的字符c



2.递归构造

第三种:先构造识别e1，再构造识别e2。最后用一个空串讲两者连接起来。



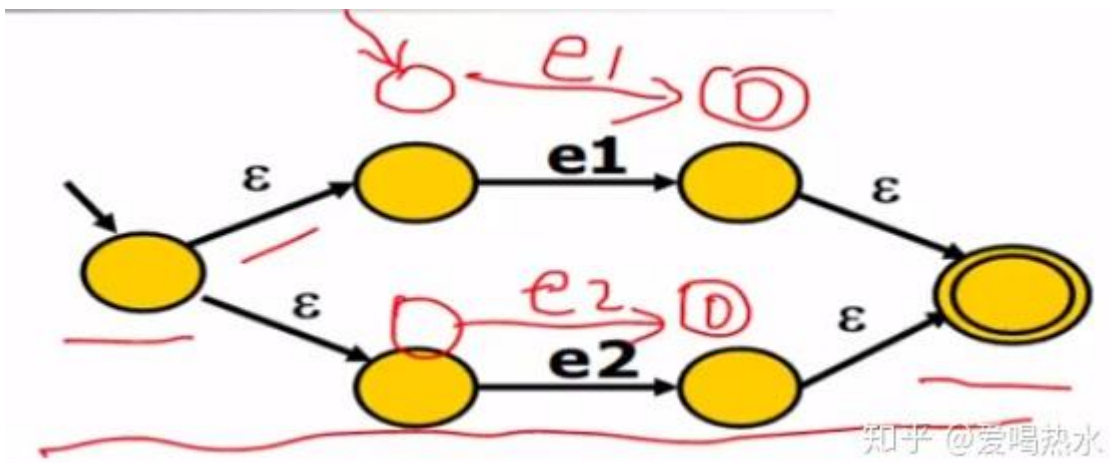
正则表达式连接

为什么不画成下面的形式？



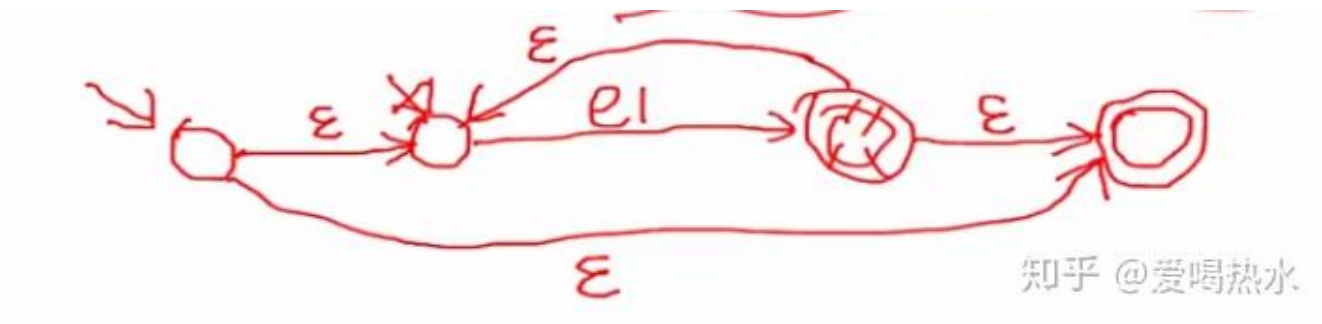
因为递归不工整。

第四种:先分别构造e1和e2的识别，再通过两个节点和空串将其连接。使其可以接受其中一个字符就达到接受状态。



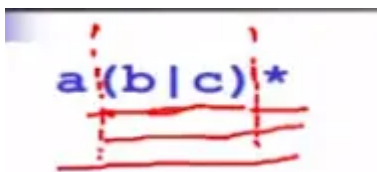
正则表达式选择

第五种:

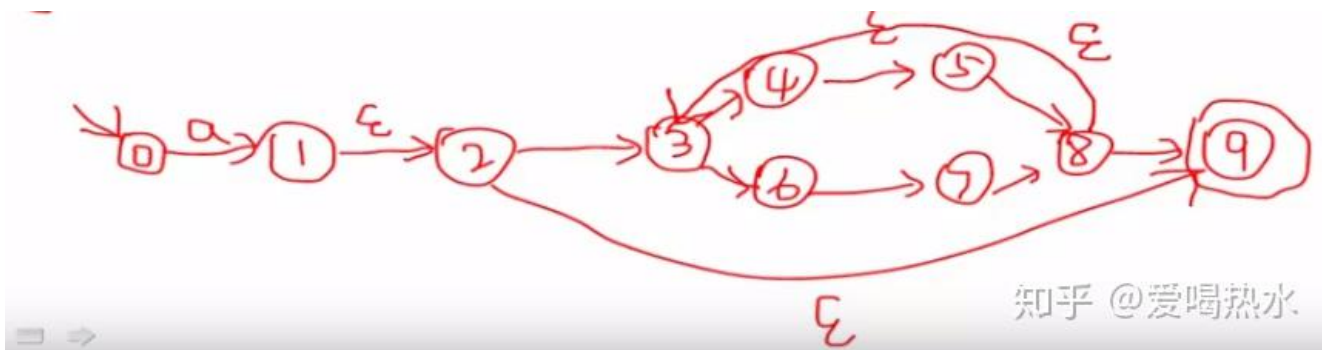


正则表达式闭包

例子:



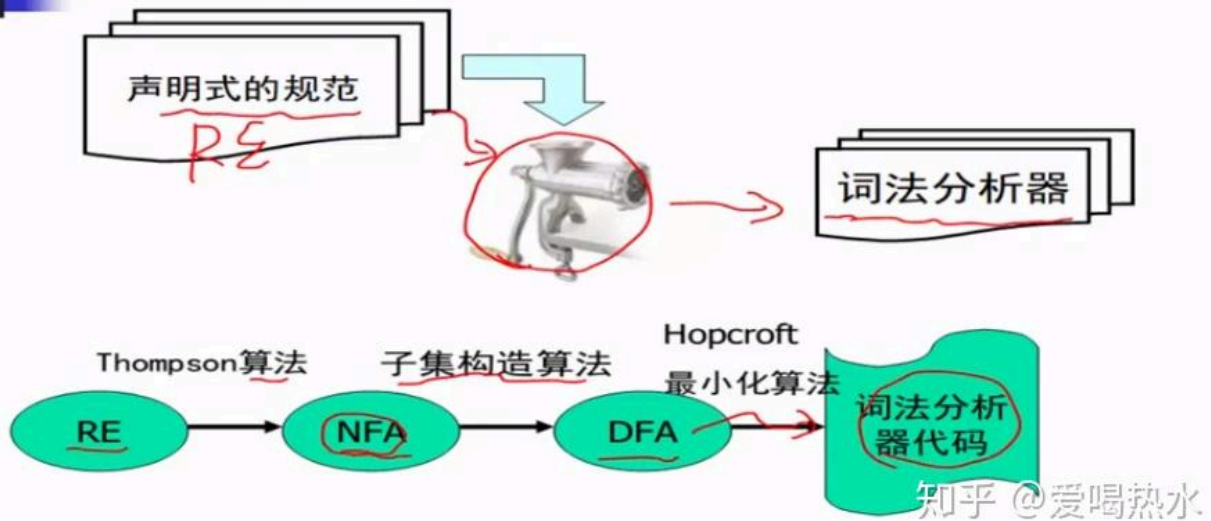
优先级: 括号 > 闭包 > 连接



例子所对应的NFA

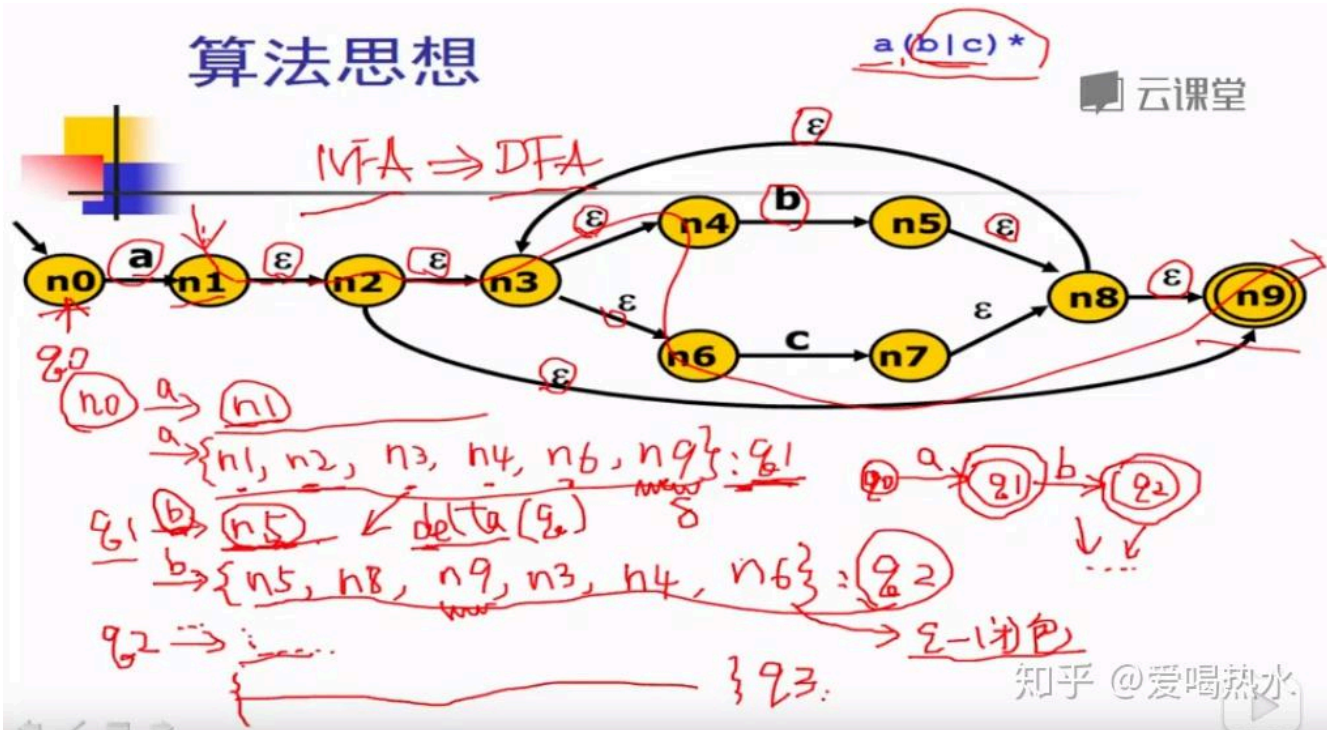
NFA转换到DFA:子集构造算法

回顾：自动生成



NFA的转移边上包含 ϵ ，所以转移是不确定的。因此我们需要使用子集构造算法将NFA转换到DFA。

算法思想

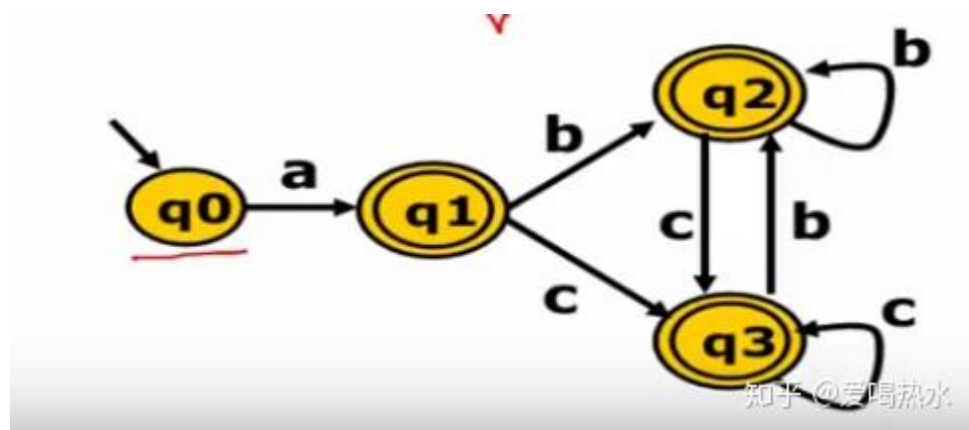
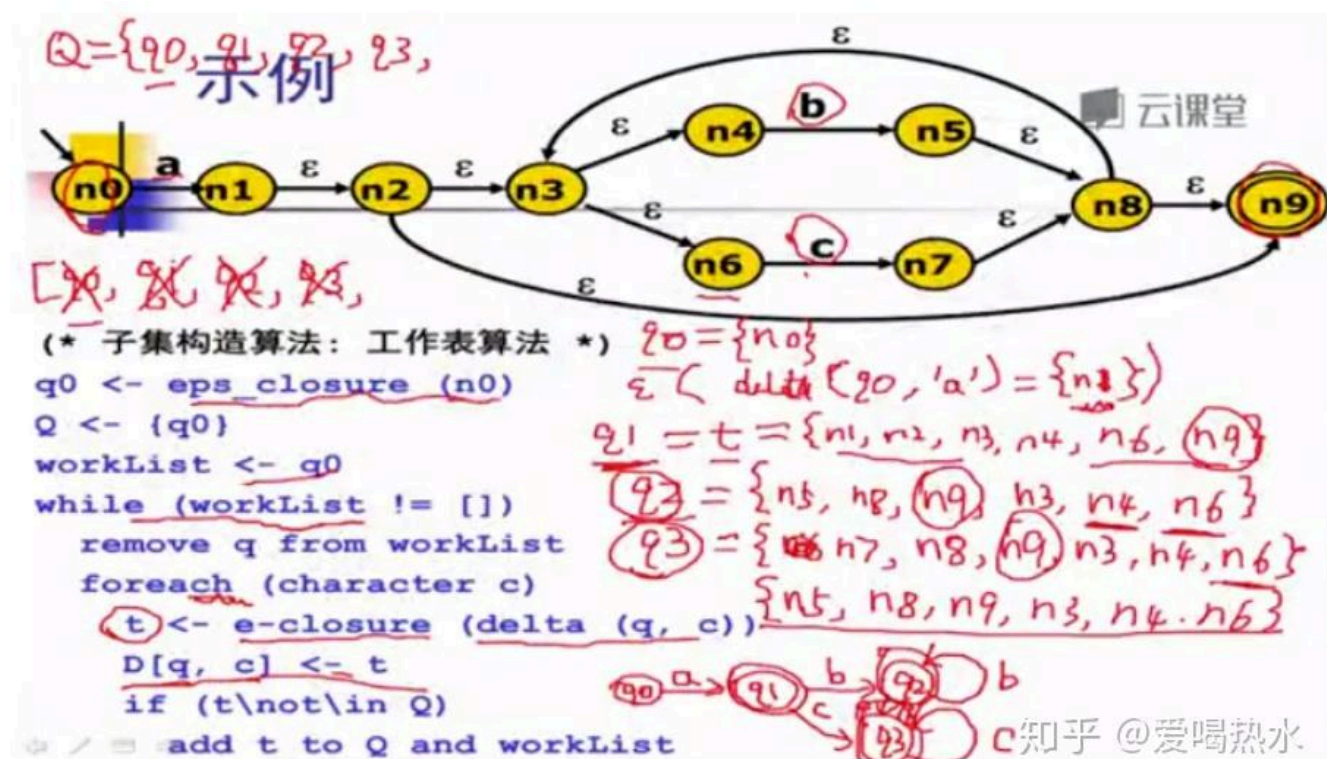


因为走 ϵ 是没有代价的，所以当 n_0 吸收字符 a 到达 n_1 后，还可以通过走 ϵ 边到达更多节点。

1. 我们可以将 n_0 吸收字符 a 可以走到的节点看作一个集合 q_1 。 $q_1: \{n_1, n_2, n_3, n_4, n_6, n_9\}$, 共有6个节点。
2. 在集合 q_1 的基础上，输入字符 b 所到达的节点构成集合 q_2 。 $q_2: \{n_5, n_8, n_9, n_3, n_4, n_6\}$
3. 在 q_2 的基础上，输入字符所到达的节点构成集合 q_3 。

4.最后根据各个集合来构造自动机。

最后的全过程以及子集构造算法伪代码如图



最终的DFA

对算法的讨论

■ 不动点算法

- 算法为什么能够运行终止

■ 时间复杂度

- 最坏情况 $O(2^N)$
- 但在实际中不常发生
 - 因为并不是每个子集都会出现

$10 \quad n_0 \sim n_9$

$$2^{10} = 1024 : 4$$

知乎 @爱喝热水

基于深度优先的闭包计算

ϵ -闭包的计算：深度优先

/* ϵ -closure: 基于深度优先遍历的算法 */

set closure = {};

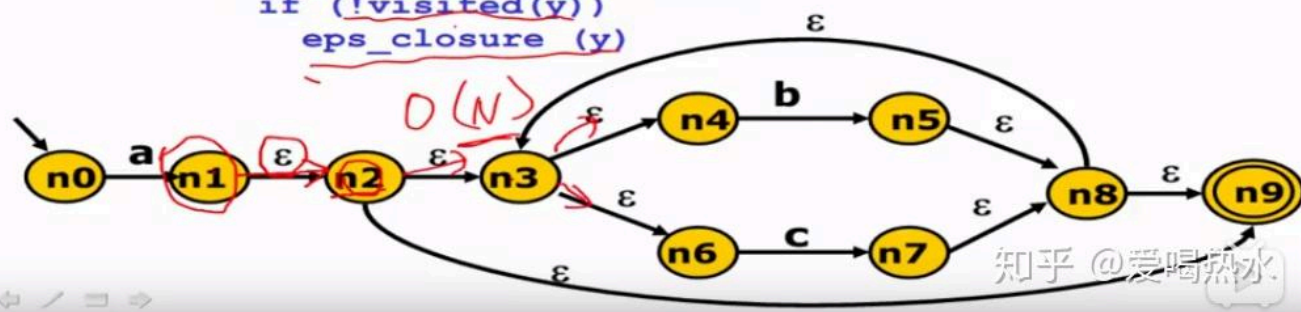
void eps_closure (x)

closure += {x};

foreach (y: x $\xrightarrow{\epsilon}$ y)

if (!visited(y))

eps_closure (y)



基于宽度优先的闭包计算

ϵ -闭包的计算：宽度优先

```
/*  $\epsilon$ -closure: 基于宽度优先的算法 */  
set closure = {};  
Q = []; // queue  
void eps_closure (x) =  
    Q = [x];  
    while (Q not empty)  
        q <- dequeue (Q)  
        closure += q  
        foreach (y: q  $\xrightarrow{\epsilon}$  y)  
            if (!visited(y))  
                enqueue (Q, y)
```

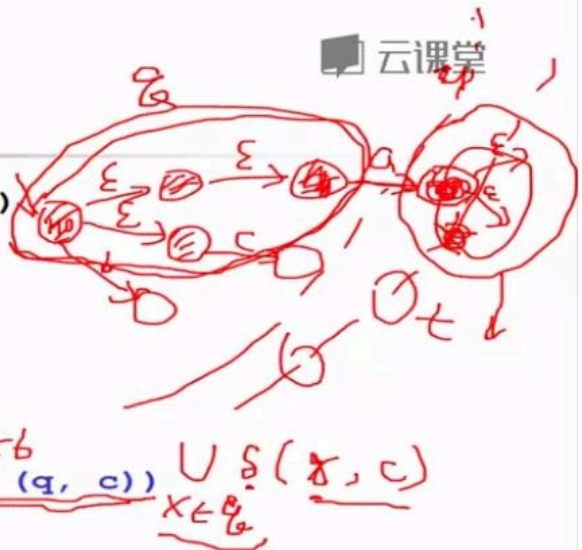


知乎 @爱喝热水

子集构造算法分析：

子集构造算法

```
(* 子集构造算法：工作表算法 *)  
q0 <- eps_closure (n0)  
Q <- {q0}  
workList <- q0 队列  
while (workList != [])  
    remove q from workList  
    foreach (character c) 256  
        t <- e-closure (delta (q, c))  
        D[q, c] <- t  
        if (t \not\in Q)  
            add t to Q and workList
```



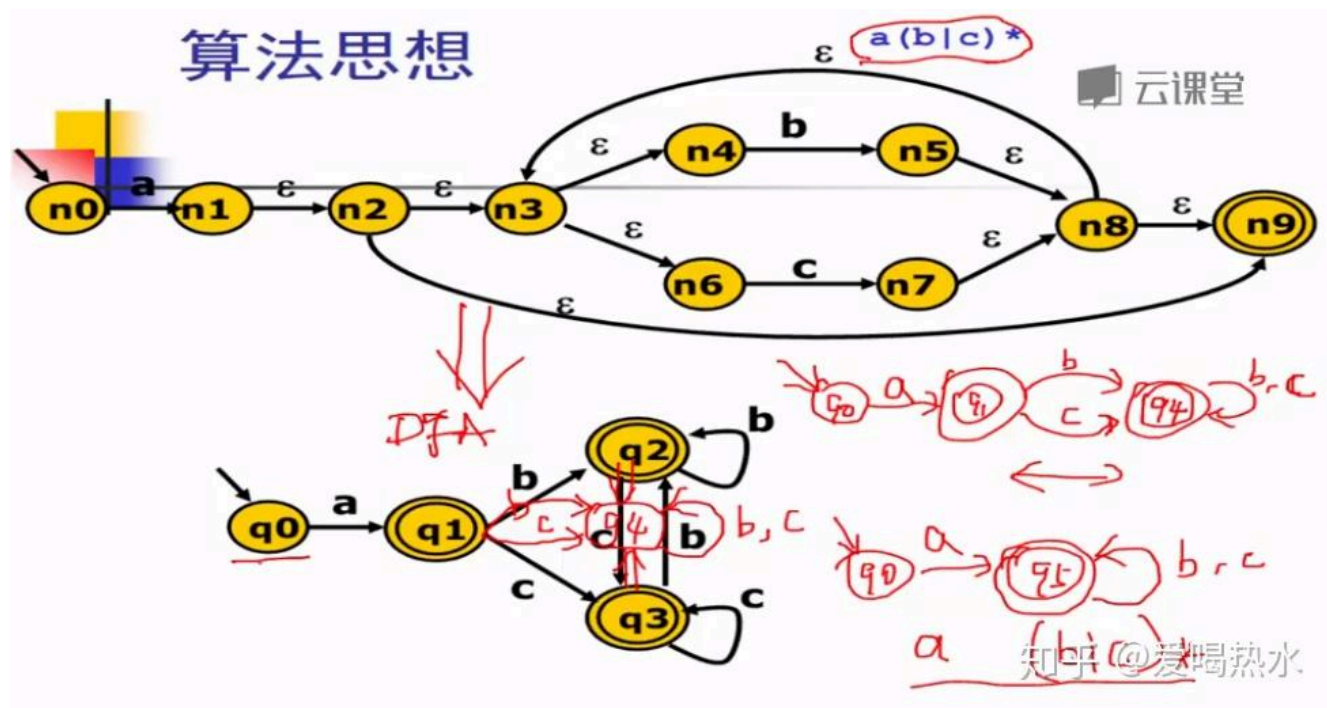
知乎 @爱喝热水

```
q0 <- eps_closure (n0) //首先计算起始状态的闭包得到q0  
Q <- {q0} //Q是最后得到的DFA的状态集  
workList <- q0 //workList常用队列实现  
while (workList != [])  
    remove q from workList //拿出队首元素  
    foreach (character c) //如果是ASCII则要256次  
        t <- e-closure (delta (q, c)) //delta函数是q读入c之后所能到达的节点  
        D[q, c] <- t  
        if (t \not\in Q)  
            add t to Q and workList
```


DFA的最小化：Hopcroft算法

为什么需要DFA最小化算法？

在前面的章节我们通过Thompson算法和子集构造算法，得到了DFA。通过Hopcroft算法使DFA最小化，进而使DFA输出成词法分析器的代码。



通过子集构造算法得出的DFA的每个状态并不是必须的，我们可以将其中的某些相同状态合并起来。（接受状态之间与非接受状态之间合并）

因为q2和q3都接受b, c两个字符且都在他们两个或者自身之间转化。而q1接受b,c可以转化到这两个状态上。因此我们可以将q2,q3合并成一个新的状态q4。

合并q2,q3形成q4

但是这样还不是最简便的形式，我们还可以将q1,q4合并，形成q5。

最小NFA

由此可见，我们通过Thompson算法和子集构造算法得到的DFA还需要通过Hopcroft算法来进行最小化。当DFA的边或者节点越小，它所占用的资源也越少，算法的效率也会越高。


```

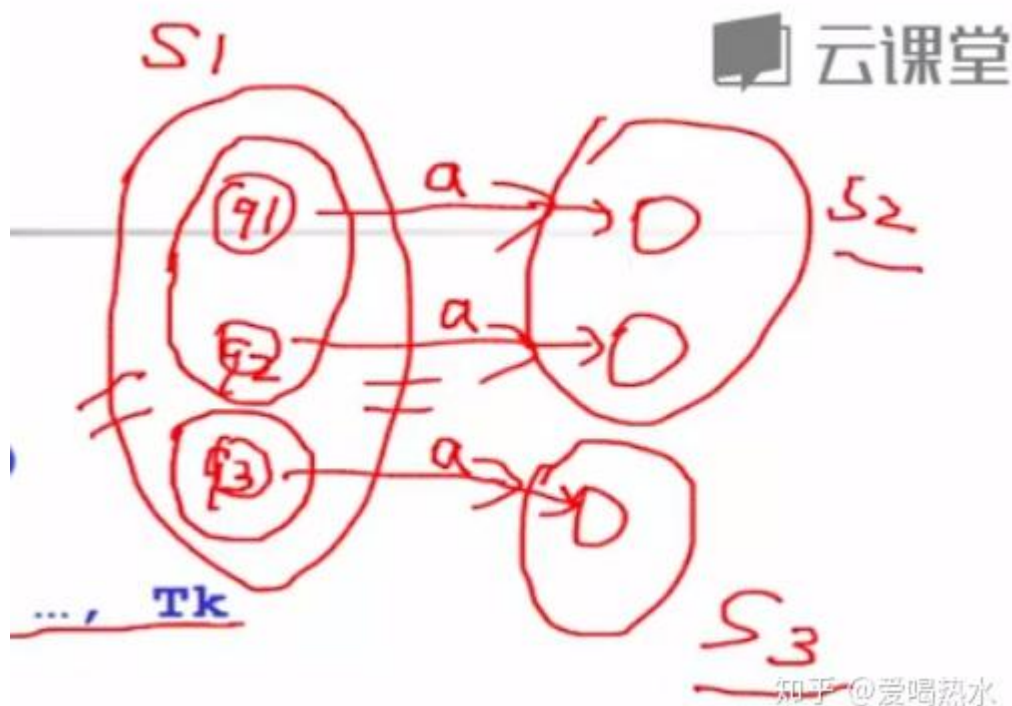
//Hopcroft算法
//基于等价类的思想
split(S)
  foreach (character c)
    if (c can split S) //如果c能切分开S
      split S into T1, ..., Tk
hopcroft ()
  split all nodes into N, A //把所有节点切分为两个子集, 一个是N(非接受状态), 一个是A(接受状态)
  while (set is still changes)
    split(S)

```

split函数

1.遍历每一个字符, 如果该字符可以切分这个集合。那我们就将这个集合分成若干个小集合。但是这些小集合必须是真子集。

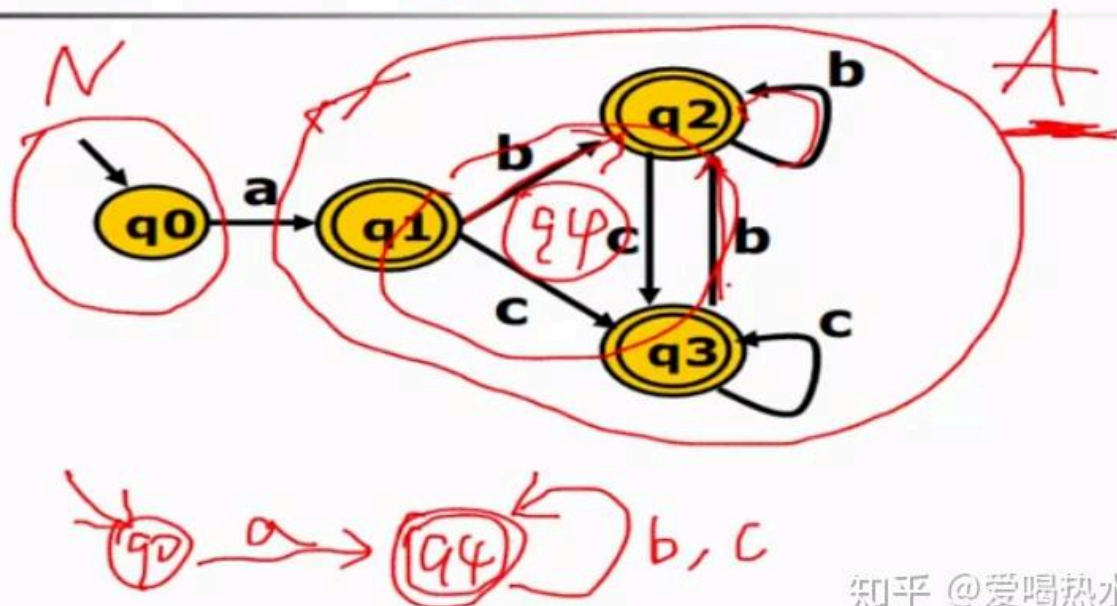
正如下图, 对于 q_1, q_2, q_3 读入字符 a 都可进行状态转移, 但 q_1, q_2 转移到的等价类集合是 s_2 , q_3 读入 a 转入等价类 s_3 。又因为 s_2, s_3 是不同的等价类, 而 q_1, q_2 对 a 的行为是一致的。对于 q_3 , 我们则认为它“叛变”了, 它对于 a 的转移上没有到达 s_2 而是到达了 s_3 。因此我们可以得出 a 字符将 s_1 集合切分成了两个集合。



例子1:

示例1

$a(b|c)^*$

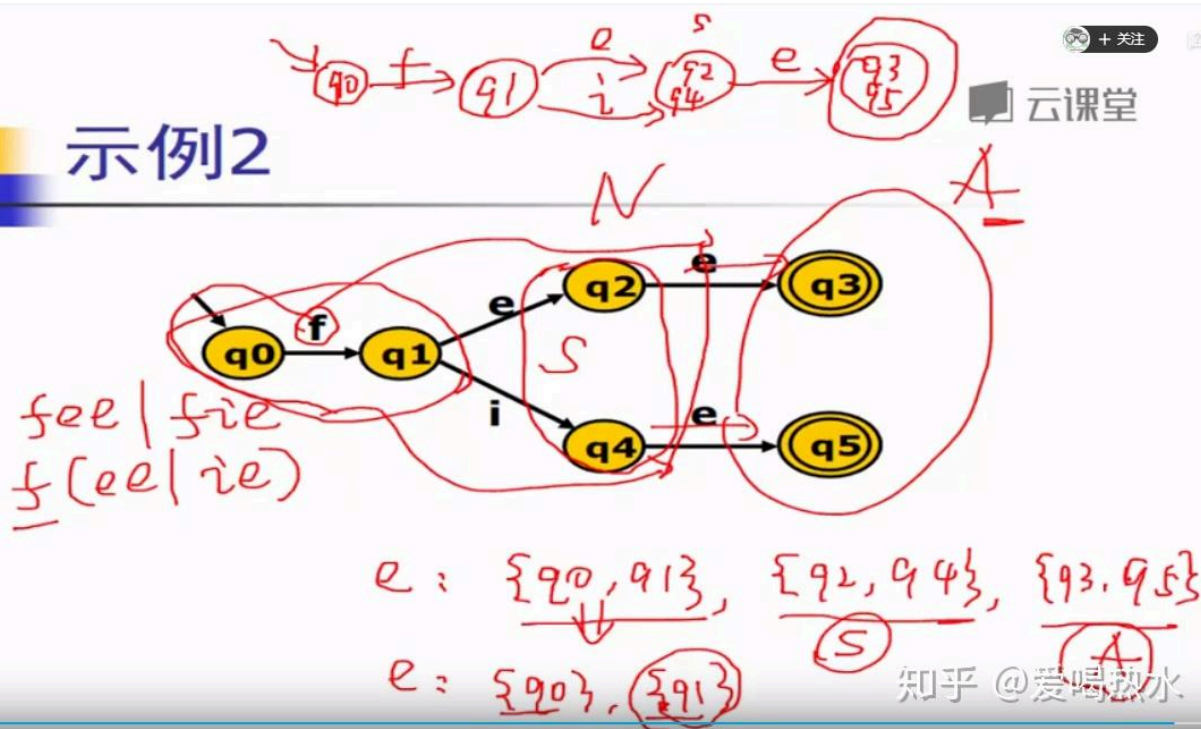


知乎 @爱喝热水

1. 首先将其切分为两个集合，N集合(非接受状态)和A集合(接受状态)
2. 我们再看这两个集合是否可以继续切分
3. 对于集合N只有一个状态了，因此不可继续切分。
4. 对于集合A，当其中的状态读入b字符时，我们发现不能状态转换出集合A。因此b不能区分出q1,q2,q3三种状态。同理c也不可以。
5. 因此可以将A集合中的三个状态浓缩成状态q4。

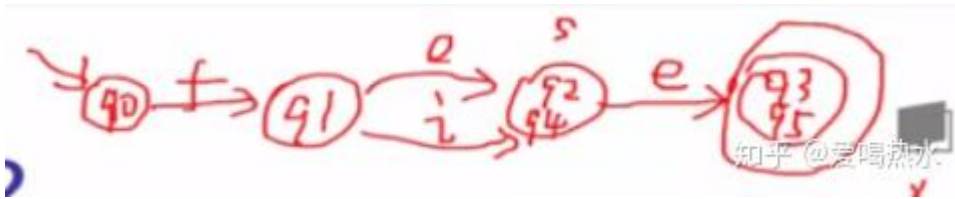
例子2:

示例2




知乎 @爱喝热水


1. 对于例2，同理先将其切分为两个集合N、A。
2. 对于集合A，它们都不接受任何字符，也没有转移。所以不可进行切分了。
3. 对于集合N，当状态 q_0 、 q_1 接受字符e的时候仍在集合N内，而状态 q_2 、 q_4 接受字符e后则转移到了集合A。因此可以将集合N拆分为 $\{q_0, q_1\}$ 、 $\{q_2, q_4\}$
4. 对于集合 $\{q_0, q_1\}$ 、 $\{q_2, q_4\}$ 进行相同的划分处理，最后可得自动机如下图。



从DFA生成分析算法



DFA的代码表示



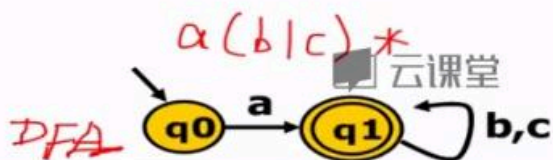
- 概念上讲，DFA是一个有向图
- 实际上，有不同的DFA的代码表示
 - 转移表 (类似于邻接矩阵)
 - 哈希表
 - 跳转表
 - . . .
- 取决于在实际实现中，对时间空间的权
衡

知乎 @爱喝热水

转移表

最终的代码由转移表和词法分析驱动代码(从文本中获得输入，再根据转移表中的相应表项来决定给定的输入是否能被DFA接受)组成。

转移表



状态\字符	a	b	c
0	1	-1	
1		1	1

```
char table[M][N];

table[0]['a']=1;
table[1]['b']=1;
table[1]['c']=1;
// other table entries
// are ERROR
```

具体驱动代码:

```
nextToken() //下一个记号或单词
state = 0 // 代表状态, 等于0的时候说明走到了q0
stack = []
while (state!=ERROR)
    c = getChar()
    if (state is ACCEPT) //是否为接受状态
        clear(stack) //清空
        push(state)
    state = table[state][c] //查表, 看能转换到什么地方
while(state is not ACCEPT)
    state = pop();
    rollback();//回滚
```

驱动代码

```
nextToken()
state = 0
stack = []
while (state!=ERROR)
    c = getChar()
    if (state is ACCEPT)
        clear(stack)
        push(state)
    state = table[state][c]
while(state is not ACCEPT)
    state = pop();
    rollback();
```



状态\字符	a	b	c
0	1		
1	ERROR	1	1

```
char table[M][N];

table[0]['a']=1;
table[1]['b']=1;
table[1]['c']=1;
// other table entries
// are ERROR
```

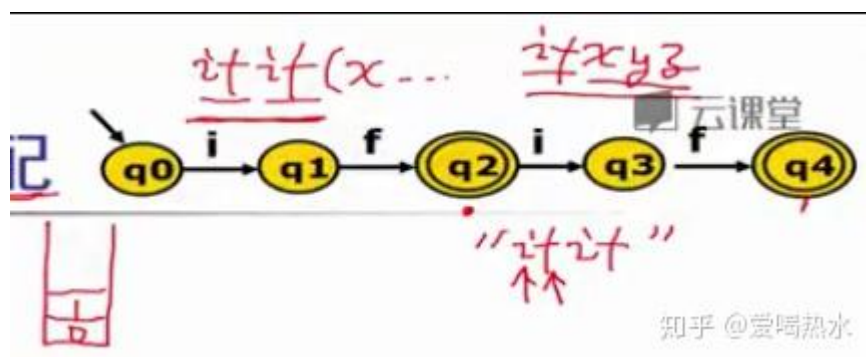
知乎 @爱喝热水

最长匹配

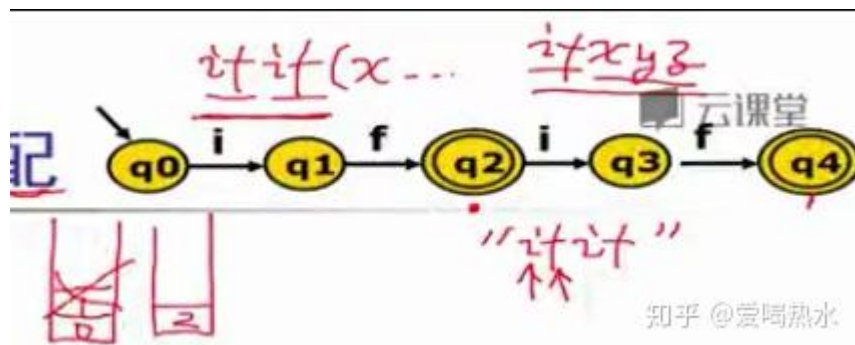
大多数情况识别采取最长匹配策略。比如当有两个关键字if和iff。当识别出if的时候，我们会继续向后识别，看是否有iff。如果有则结果为iff，否则则回退并返回if。

我们可以根据关键字if和iff来构造如下的DFA

1.初始情况下栈是空的，当读入到if时，栈的状态如下



2.当到达q2时，因为q2的接受状态，因此要清空栈，并压入2。



3.当读入i之后，则转移到了q3，因此要将3压入栈中。此时栈的状态如下

4.当再次读入第二个f后，转移到了q4状态。因为q4是接受状态，因此要清空栈并将4压入栈中。此后，4再接受任何字符都会转移到非法状态。所以识别过程结束。

总结:

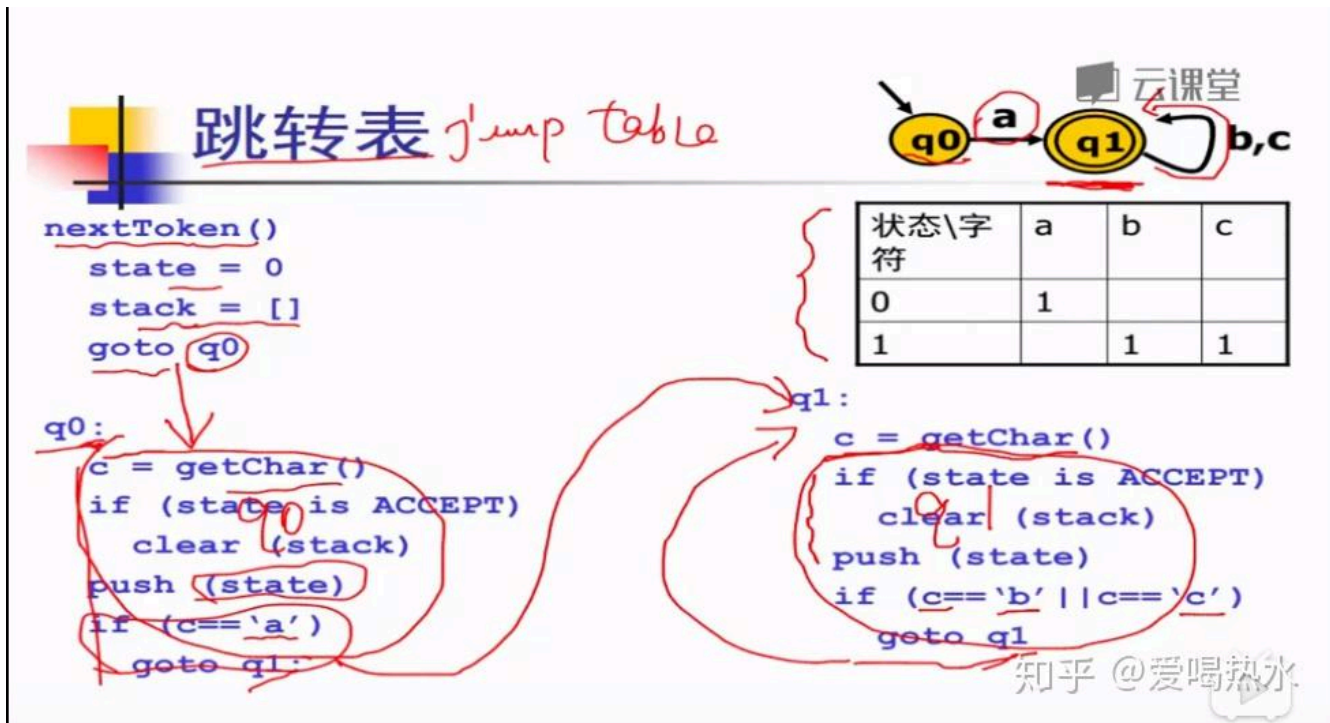
一般遵循最长匹配原则，及时到达了接受状态也会继续向后匹配寻求更长的匹配。

栈每次从最近的一个接受状态开始作为栈底。

识别失败，则栈进行pop操作，将字符的回滚到最近的一个接受状态上来。

跳转表

除了转移表，DFA还有另一种的跳转表的代码表示。



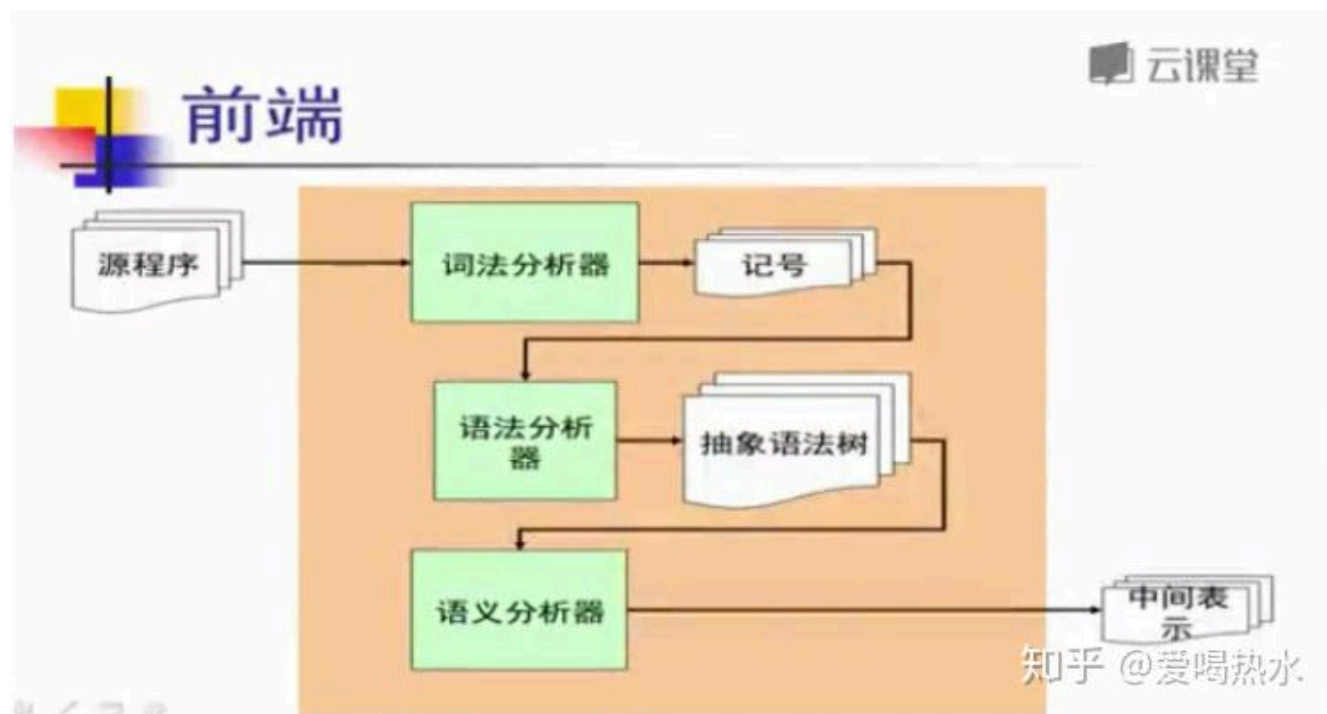
1. 对于跳转表的nextToken函数，我们有一个初试状态和空栈。
2. 首先由goto跳转到q0
3. 到达q0的内部后，如果此时处于接受状态则清空栈(原理如转移表)，然后将状态压入栈中。如果后面接受的字符是a，则跳转到q1。
4. 后面的q1状态同理
5. 每段代码可以看作一个状态。

对于跳转表，不需要维护大的数组，节约了内存。每次只需要执行仅仅一小段代码。效率高。

对于转移表，占内存，加载代码速度相对跳转表慢。

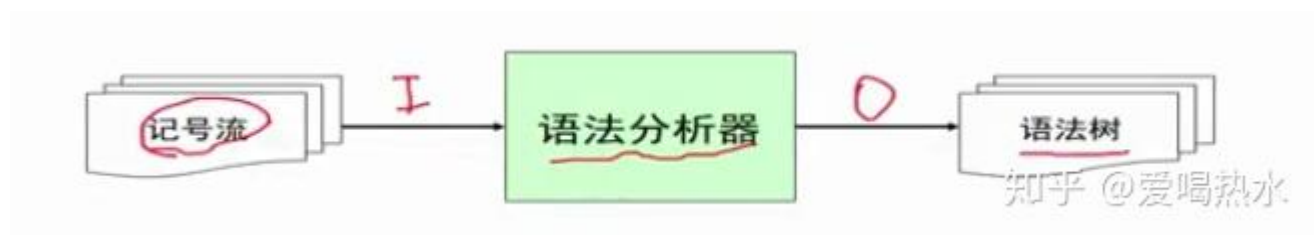
具体采用哪种实现方式还要根据具体场景具体分析。

四、语法分析 (Part1)



语法分析阶段，在编译器设计的早期主要是用于检查输入的记号中的语法是否合法，如果合法则可能直接生成目标体系结构的代码，否则则会返回相应的信息来指导程序员对其进行修改，进行修改之后则可以重复上述流程。后来的语法分析变的更加复杂，语法分析可能要生成一个叫抽象语法树的中间表示。

语法分析器的任务



1. 将记号流输入语法分析器，语法分析器输出抽象语法树这个中间表示
2. 语法分析器判断输入的记号流是否合法
3. 语法分析器需要一个标准来判别输入是否合法，所以语法分析器还有第二种输入
4. 语法分析器还要隐含的输出记号流是否合法(Yes/No)

语法分析器处理错误的例子

```
if ((x > 5)
    y = "hello"
else
    z = 1,
```

语法分析

Syntax Error: line 1, missing)

Syntax Error: line 2, missing ;

Syntax Error: line 4, expecting ; but got ,

知乎 @爱喝热水

当输入上面的源代码后，编译器会给出对应的错误提示：

1. 第一行缺少右括号
2. 第二行缺少了；
3. 第四行期望得到; 但是却得到了,

编译器可以精确定位错误的位置。程序的开发过程也是一个不断取悦编译器的过程

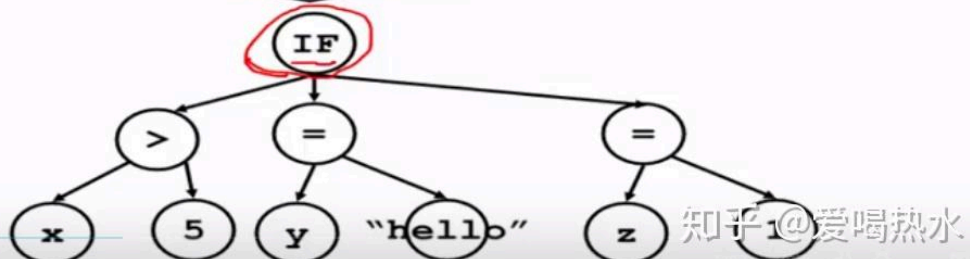
1. 编译器会指出程序中的各种语法错误，并给出诊断信息
2. 程序员根据诊断信息来修改源代码直到通过编译器
3. 通过编译后，就开始语法树的构建(后端需要的数据结构)
4. 语法分析器生成抽象语法树，并将其放入内存中

云课堂

例子：语法树构建

```
if (x > 5)
    y = "hello";
else
    z = 1;
```

语法分析



知乎 @爱喝热水

1. 例子中为三层语法树，分别根据字符判断>,,=
2. 构建的抽象语法树包含后期要用到的所以信息，我们后续的流程只要看抽象语法树就可以了。

路线图

- 数学理论：上下文无关文法（CFG）
 - 描述语言语法规则的数学工具
- 自顶向下分析
 - 递归下降分析算法（预测分析算法）
 - LL分析算法
- 自底向上分析
 - LR分析算法

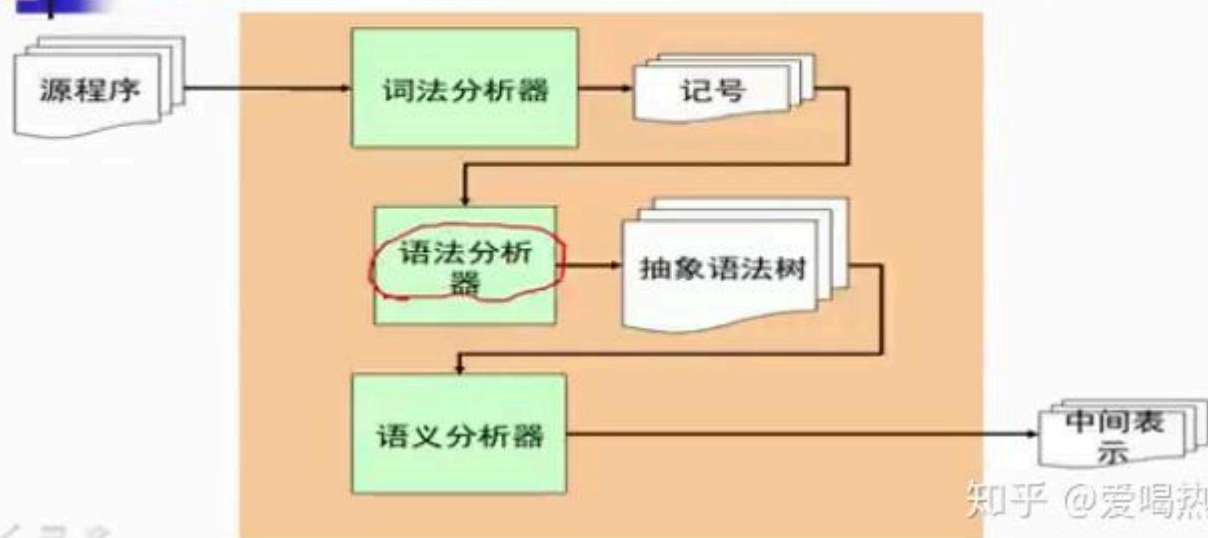
知乎 @爱喝热水

上下文无关法和推导

上下文无关文法是什么？

上下文无关文法是描述程序语法的一个强有力的数学工具，通过对这样一个数学工具认真研究之后，我们可以根据文法来设计一些高效的算法。我们之前讨论过前端的一个核心的结构，语法分析其实在整个前端中处于一种比较核心的地位。

前端



知乎 @爱喝热水

将记号流输入到语法分析器中，语法分析器输出抽象语法树。但是在这个过程中要判断是否满足语法规则，所以我们需要告诉编译器语法规则是什么？为了描述语法规则，我们就需要一些数学工具。

形式化的描述语法

示例

■ 自然语言中的句子的典型结构：

■ 主语 谓语 宾语

■ 名字 动词 名词

RRR

■ 例子：

■ 名词：{羊、老虎、草、水}

■ 动词：{吃、喝}

■ 句子：

羊 吃 草

羊喝水

羊吃水

老虎吃老虎

老虎吃草

草吃老虎

... 知乎 @爱喝热水

对于上图中的句子，有些是符合语法规则的，但是有些就不符合语法规则(草吃老虎，老虎吃草)。所以我们可以看出，紧靠语法规则，我们是很难判断一个语句是否合法的。

我们先将语法规则形式化

形式化

S \rightarrow N V N
N \rightarrow sheep
| tiger
| grass
| water
V \rightarrow eat
| drink

非终结符：{S, N, V}
终结符：{s, t, g, w, e, d}
开始符号：S

$S \rightarrow N V N$
 $\downarrow \quad \downarrow \quad \downarrow$
 $\rightarrow s e g$
 $\rightarrow \dots$

知乎 @爱喝热水

通过形式化，我们可以知道合法的句子里面包含了名词和动词，动词和名词又分别包含什么。

上下文无关文法

- 上下文无关文法 G 是一个四元组：

$$G = (T, N, P, S)$$

- 其中 T 是终结符集合 $\{\dots\}$
- N 是非终结符集合 $\{\dots\}$
- P 是一组产生式规则 $\{\dots\}$
 - 每条规则的形式： $X \rightarrow \beta_1 \beta_2 \dots \beta_n$ $n \geq 0$
 - 其中 $X \in N$, $\beta_i \in (T \cup N)$
- S 是唯一的开始符号（非终结符）
 - $S \in N$

知乎 @爱喝热水

数学作为工具要结合具体例子来理解

例子：

上下文无关文法的例子

$E \rightarrow \text{num}$
 $E \rightarrow \text{id}$
 $E \rightarrow E + E$
 $E \rightarrow E * E$

$E \rightarrow \text{num}$

$E \rightarrow \text{id}$

$E \rightarrow E + E$

$E \rightarrow E * E$

$G = (N, T, P, S)$

非终结符： $N = \{E\}$

终结符： $T = \{\text{num}, \text{id}, +, *\}$

开始符号： E

产生式规则集合：

$\{\dots\}$ 4

BNF: {Backus
Naur

$\langle E \rangle \rightarrow$
 num
 id

知乎 @爱喝热水

如果我们严格写出四条产生式，结果应该是：

$E \rightarrow \text{num}$

$E \rightarrow \text{id}$

$E \rightarrow E + E$

$E \rightarrow E * E$

因为左边四条规则是完全一样的，所以我们用一个'|'来进行简化，从而将左边的共同部分简化。


我们还规定所以的大写符号为非终结符，小写符号均为终结符。

在文献中常用BNF范式来区分终结符与非终结符：

- 非终结符常用一对尖括号括起
- 所有终结符要加上下划线

推导的概念

云课堂



推导

$\bigcirc \rightarrow p_1 \dots p_n$

- 给定文法G，从G的开始符号S开始，用产生式的右部替换左侧的非终结符
- 此过程不断重复，直到不出现非终结符为止
- 最终的串称为句子

$$\begin{aligned} S &\rightarrow NVN \\ \cdot & \rightarrow N \cdot d N \\ &\rightarrow N d \cdot g \\ &\rightarrow \underline{t d g} \end{aligned}$$

句子

S	→	N	V	N
N	→	s		
			t	
			g	
			w	
v	→	a		
			d	

知乎 @爱喝热水

1. 一开始我们只有一个非终结符S，然后我们用其右部替换左边的非终结符得到NVN
2. 替换后S则消失了，我们继续用右部替换左边的非终结符NVN
3. 对于NVN三个非终结符我们可以随意选择一个进行替换，我们这次选择对V进行替换得到NdN
4. 此时我们再任意选择一个N进行替换，得到Ndg
5. 再替换最后一个N得到tdg，这时候串中已经没有非终结符了，这样的串我们称其为句子

课后题:

S可以推导出多少个不同的句子?

$4 \times 2 \times 4 = 32$ 个 N 有 4 种, V 有 2 种

最左推导和最右推导

云课堂

最左推导和最右推导

- 最左推导: 每次总是选择最左侧的符号进行替换

$$\begin{aligned} S &\rightarrow N V N \\ &\rightarrow s V N \\ &\rightarrow s d N \\ &\rightarrow s d s \end{aligned}$$

```
S -> N V N
N -> s
    | t
    | g
    | w
V -> e
    | d
```

知乎 @爱喝热水

同理, 最右推导就是每次总是选择最右侧的非终结符进行替换

语法分析

云课堂

语法分析

- 给定文法 G 和句子 s , 语法分析要回答的问题: 是否存在对句子 s 的推导?

$S = s e s Y$
 $s s s N$
!

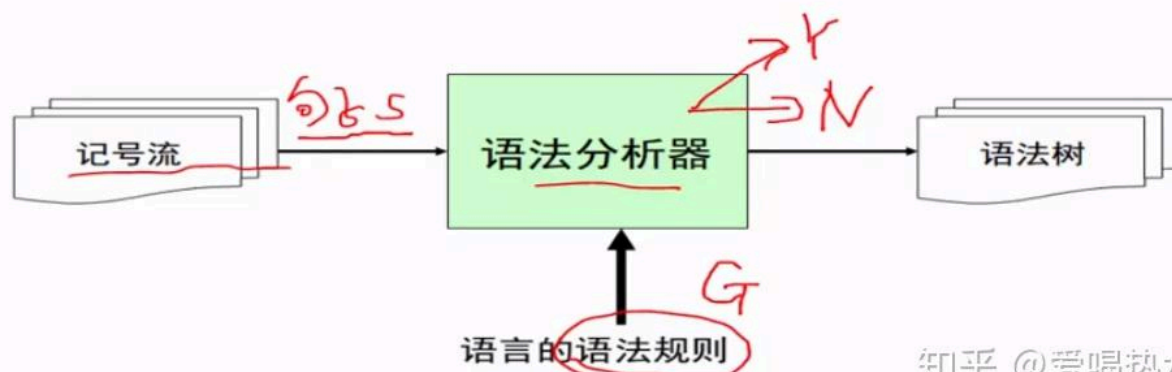
G

```
S -> N V N
N -> s
    | t
    | g
    | w
V -> e
    | d
```

知乎 @爱喝热水

语法分析器的任务

语法分析器的任务

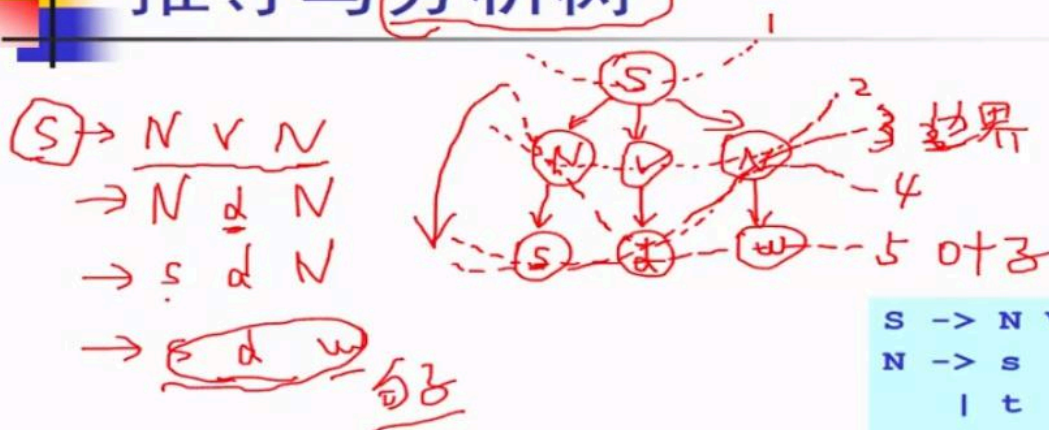


知乎 @爱喝热水

- 句子 s 和语言的语法规则 G 输入到语法分析器
- 语法分析器还要回答 G 中是否存在对句子 S 的推导，回答Yes/No
- 负责的语法分析器应该给程序员反馈一个出错信息

语法分析:分析树与二义性

推导与分析树



S	\rightarrow	NVN
N	\rightarrow	s
		t
		g
		w
V	\rightarrow	e
		d

G

知乎 @爱喝热水

1. 在任何的一个推导步骤中，我们可以画出一个抛面，比如第一步的时候只有一个 s 节点
2. 经过第一步的推导后 s 变成了 NVN ，这时候就可以画出2号抛面
3. 第二步将 V 换成 d ，形成了 NdN ，就有了3号抛面。
4. 同理当第一个 N 被替换成 s ，则产生4号抛面 sdN

我们将每一个抛面称为语法推导的边界。最终的边界是这个句子被推导出来的结果。

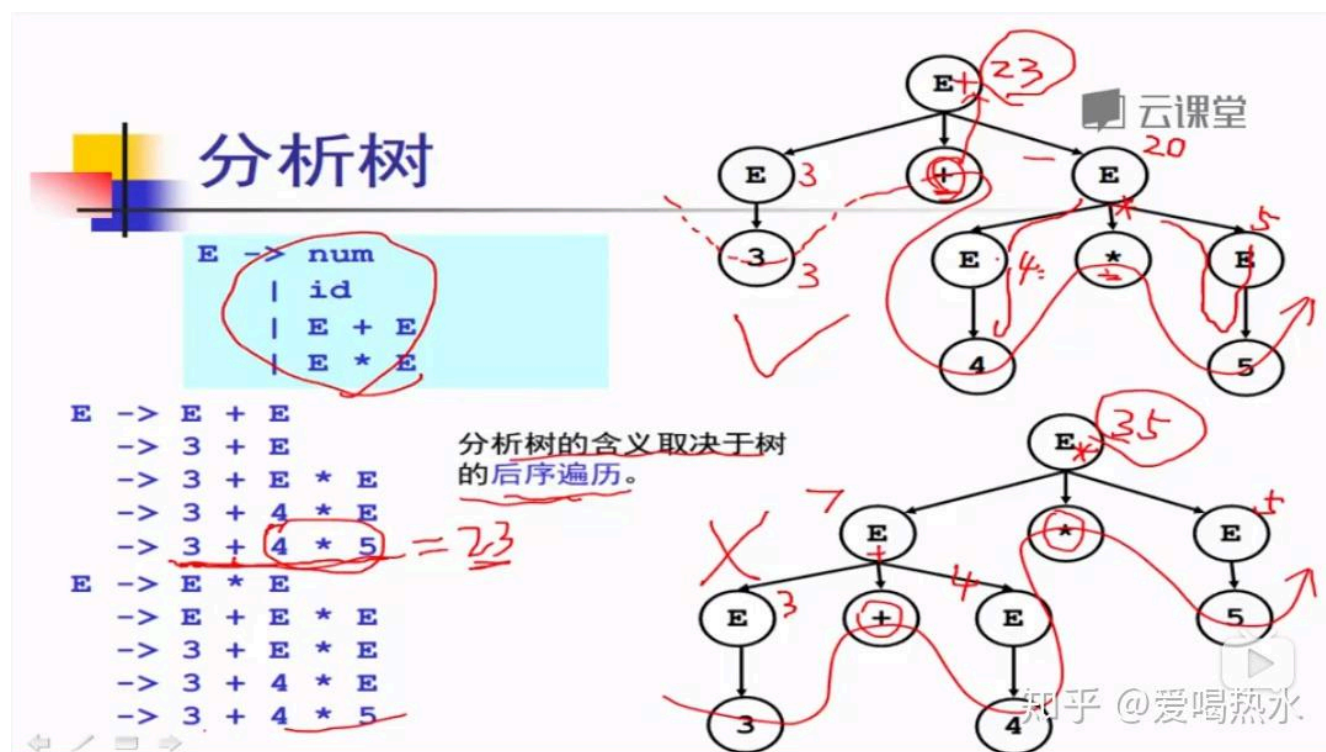
分析树

分析树

- 推导可以表达成树状结构
 - 和推导所用的顺序无关（最左、最右、其他）
- 特点：
 - 树中的每个内部节点代表非终结符
 - 每个叶子节点代表终结符
 - 每一步推导代表如何从双亲节点生成它的直接孩子节点

知乎 @爱喝热水

表达式例子



在这个例子中，我们来看能否通过上图左上角的语法规则G推导出 $3+4*5$ 这个句子

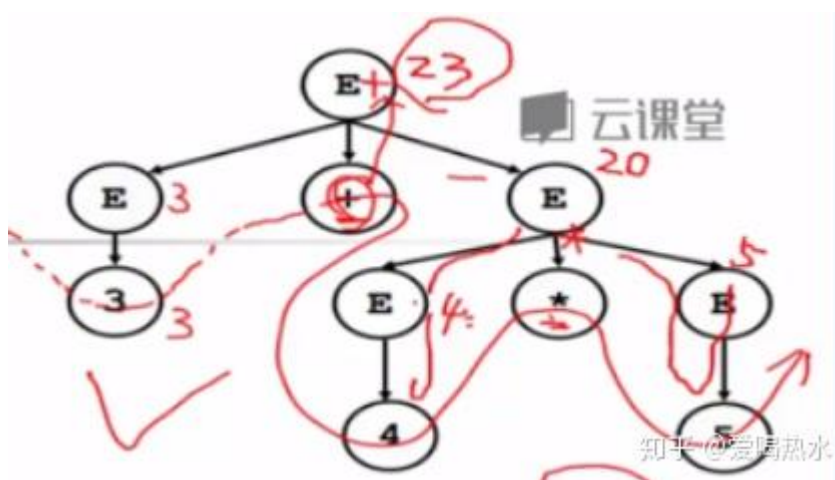
1. 从开始符号E开始进行替换，我们需要从后两种中选择(前两种是终结符，不符合)。选择第三种得到 $E \rightarrow E+E$
2. 接下来我们从 $E+E$ 中任意选择一个进行展开，我们先选最左边的E将其展开为3，得到 $3+E$
3. 然后我们将剩下E进行替换，得到 $3+E * E$
4. 再将最左边的E进行替换得到 $3+4 * E$
5. 最后将最右边的E替换成5，得到 $3+4 * 5$
6. 在这个推导中，我们采用的是最左推导，得到的推导结果是Yes

另一种推导

1. 先选择 $E * E$ 进行替换
2. 将最左边的E替换为 $E+E$ 得到 $E+E * E$
3. 继续进行最左推导，得到 $3+E * E$
4. 同上，得到 $3+4 * E$
5. 最后替换得到 $3+4 * 5$

我们根据两种不同的推导方式可以得到不同的分析树:

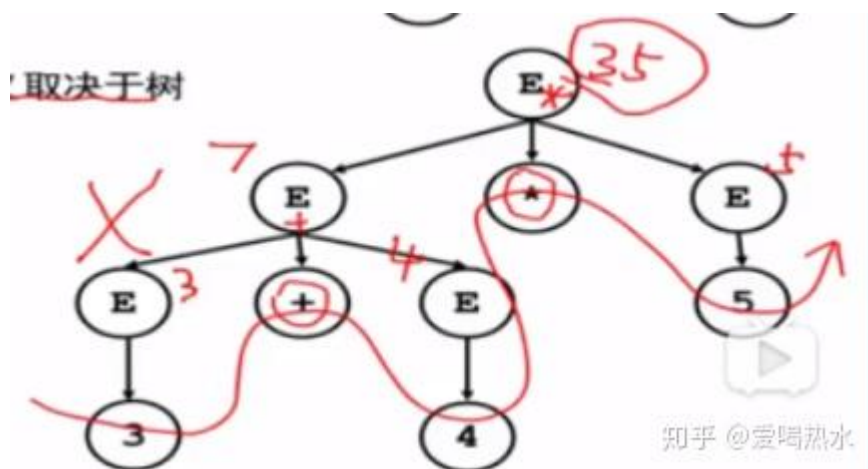
第一种推导的分析树:



第一种推导的分析树

将所有叶子节点连出来可以得到我们推导的句子，先进行乘法再进行加法

第二种推导的分析树:



将叶节点进行连接，同样可以生成我们期望的句子，但是它是先进行加法，再进行乘法

因为分析树的含义取决于树的后序遍历的顺序，所以这两种结构的树的含义有所不同。对于第一棵树结果为23。第二棵树结果为35。按照一般的常识(乘法的优先级高于加法的优先级)，第一种的结果是对的。

因此我们可以看出，从相同的规则可能会推导出完全不同的结果，程序存在着歧义。

二义性文法

二义性文法

- 给定文法G，如果存在句子s，它有两棵不同的分析树，那么称G是二义性文法
- 从编译器角度，二义性文法存在问题：
 - 同一个程序会有不同的含义
 - 因此程序运行的结果不是唯一的
- 解决方案：文法的重写

二义性文法

- 给定文法G，如果存在句子s，它有两棵不同的分析树，那么称G是二义性文法
- 从编译器角度，二义性文法存在问题：
 - 同一个程序会有不同的含义
 - 因此程序运行的结果不是唯一的
- 解决方案：文法的重写

- 从编译器角度，二义性文法存在问题：
 - 同一个程序会有不同的含义
 - 因此程序运行的结果不是唯一的

- 同一个程序会有不同的含义
- 因此程序运行的结果不是唯一的

- 因此程序运行的结果不是唯一的

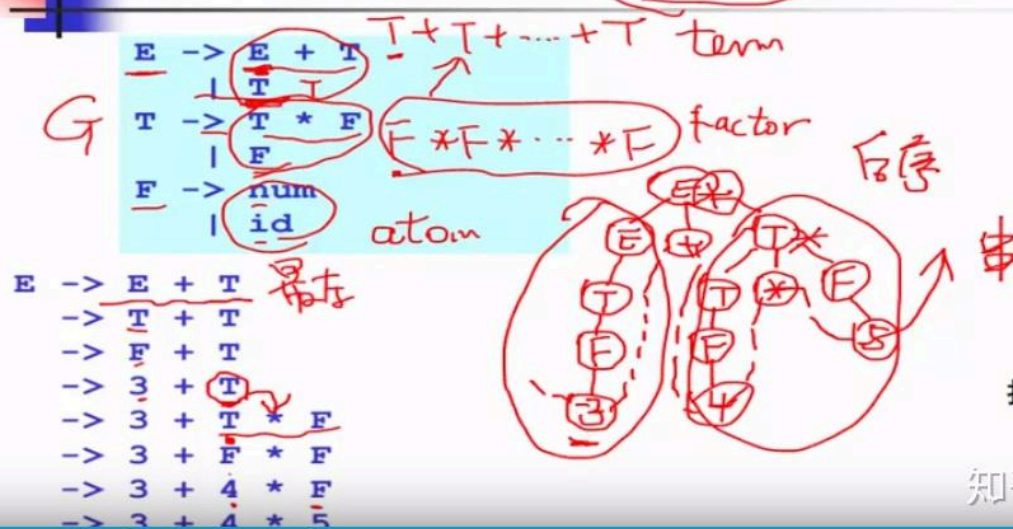
- ## ■ 解决方案：文法的重写

知乎 @爱喝热水

表达式文法的重写

文法的重写需要具体问题具体分析，不存在一个算法可以使给定的任意文法从二义性转换到非二义性。

表达式文法重写



对于 $E \rightarrow E + T$

| T

因为左边的 E 和右边的 E 是相同的，所以可以进行递归

$E \rightarrow E + T$

$\rightarrow E + T + T$ // 将一个 E 替换成 $E + T$

$\rightarrow E + T + T + T$ // 同理

.....

$\rightarrow T + T + T + \dots + T$

对于 $T \rightarrow T * F$

| F

同理最后可以替换成

$T \rightarrow F * F * F * \dots * F$

我们通常将 T 看作一个 term 项， F 看作一个 factor

比如 $1 * 2 + 3 * 4 + 5 * 6$ 中， $1 * 2$ 是一个 term 项， 1 或 2 就是 factor

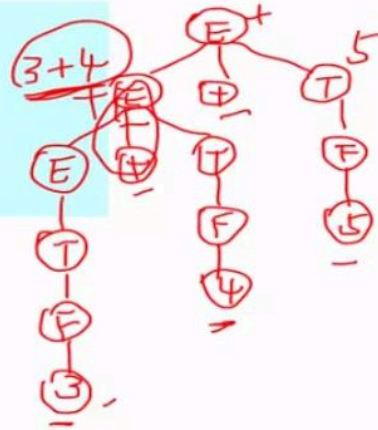
表达式文法的重写

```

E -> E + T
    | T
T -> T * F
    | F
F -> num
    | id
  
```

```

E -> E + T
-> E + T + T
-> T + T + T
-> F + T + T
-> 3 + T + T
-> 3 + F + T
-> 3 + 4 + T
-> 3 + 4 + F
-> 3 + 4 + 5
  
```



左结合

$$\underline{3+4+5}$$

推导这个句子

$$3+4+5$$

知乎 @爱喝热水

保证了左结合性

自顶向下分析1

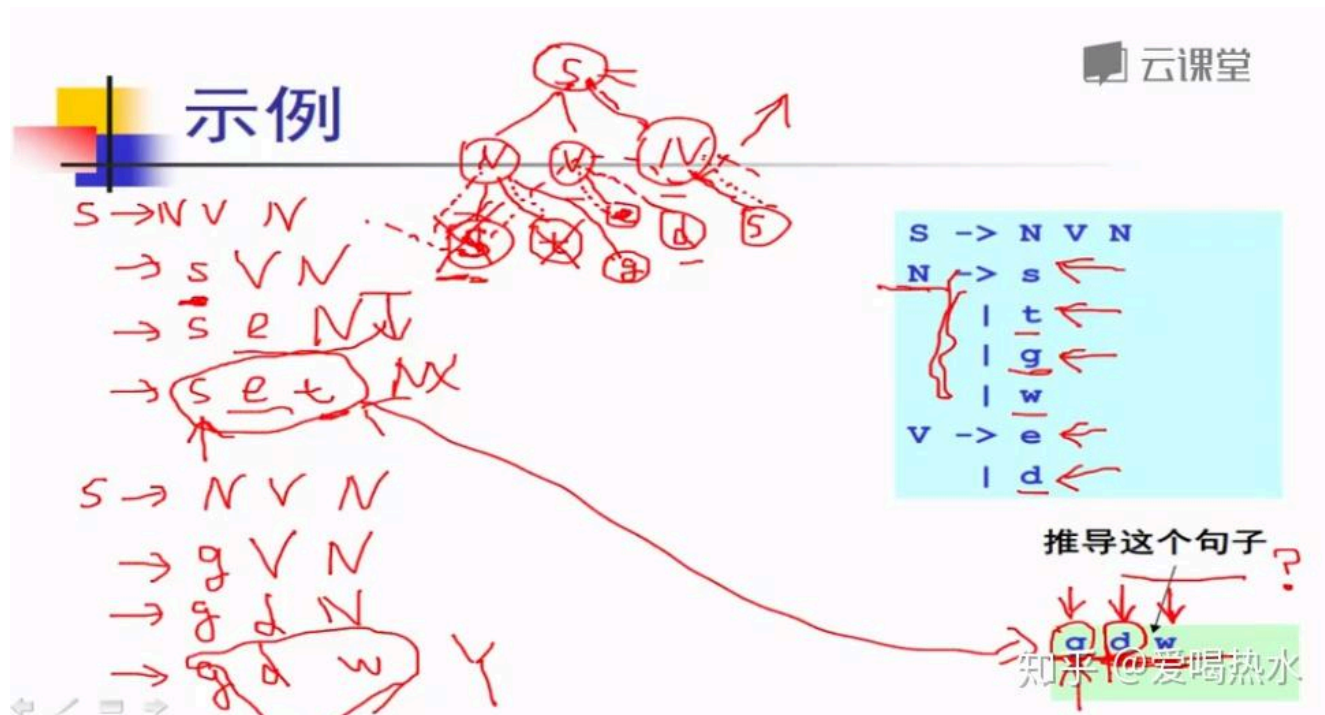
自顶向下分析的算法思想

- 语法分析：给定文法 G 和句子 s ，回答 s 是否能够从 G 推导出来？
- 基本算法思想：从 G 的开始符号出发，随意推导出某个句子 t ，比较 t 和 s
 - 若 $t=s$ ，则回答“是”
 - 若 $t \neq s$ ，则？ “否” $t' \stackrel{?}{=} s$ $t'' \stackrel{?}{=} s$
- 因为这是从开始符号出发推出句子，因此称为自顶向下分析
 - 对应于分析树自顶向下的构造顺序

知乎 @爱喝热水

1. 从 G 的开始符号出发，随意推导出某个句子 t ，比较 t 和 s
2. 如果 $t=s$ ，则回答“是”
3. 若 $t \neq s$ ，不可以直接回答否。我们需要回溯，将之前的过程推翻，重新推导 t' ，然后再看 t' 和 s 是否相等。如果 t' 和 s 不相等则继续推导 t' 。我们不断重复这个过程，直到我们枚举出 $t(n)=s$ ，或者推导出的所有结果都不能使其等于 s 。

示例



在上面的示例中，我们将推导的串和目标串进行匹配

1. 首先，我们将S推导成NVN
2. 然后我们再将最左边的N替换为s
3. 这时候我们发现第一个字母是s，与目标句子gdw中的第一个字母g不匹配。
4. 所以我们要进行回溯，将s去掉，然后我们再次尝试t,g,w。第二次我们用t来替换N
5. t显然与g不匹配，接下来我们用g来进行匹配。这次就成功了

自顶向下算法分析

```
tokens[]; // all tokens
i=0;
stack = [S] // S是开始符号，stack中存放所有终结符和非终结符
while (stack != [])
if (stack[top] is a terminal t)
if (t==tokens[i++]) //比较t是不是等于tokens[i++]，如果相等就将t从栈中弹出，不相等则回溯
pop();
else backtrack();
else if (stack[top] is a nonterminal T)
pop(); push(the next right hand side of T)
```

例子:

- 我们首先将需要推导的句子gdw存入tokens中
- i一开始指向g

- 开辟一个栈，一开始栈里只有一个起始符号S,此时top指向-1

- 此时栈不为空，我们需要判断栈顶元素是终结符还是非终结符。显然，此时栈顶为非终结符，所以我们将其弹出，然后将它右边没有考虑过的符号压入栈中。需要注意的是，压入栈中的顺序为N2、V、N1(这样的顺序将符号压入栈中相当于对树进行后序遍历)。



- 此时栈中有三个非终结符，所以栈不为空，我们先来判断栈顶的元素是否为终结符，因为N1为非终结符，所以我们将其弹出，然后将它右边的下一个没有考虑过的符号压入栈中。此时栈的情况如下图。
- 循环进入下一轮，因为这时候栈顶是终结符，所以我们进入if中，我们判断s是否等于g，显然不等于，所以进行回溯，将s弹出去，再将N1压回来。此时栈的状态如下。回溯的时候tokens中的i也要往前一位。

- 此时再次进行判断，首先栈不为空，因为栈顶元素N1为非终结符，所以我们将N1弹出去，将N1的下一个没有考虑过的右部符号压入。因为我们上面已经考虑过了s，所以这时候我们压入t。所以此时栈的情况如下。

我们可以看出，算法的主要思想就是回溯。一个个符号尝试，不行就回溯。所以这是一个相当昂贵的算法

算法的讨论



算法的讨论

- 算法需要用到回溯
 - 给分析效率带来问题
- 而就这部分而言（就所有部分），编译器必须高效
 - 编译上千万行的内核等程序
- 因此，实际上我们需要线性时间的算法
 - 避免回溯
 - 引出递归下降分析算法和LL(1)分析算法

知乎 @爱喝热水

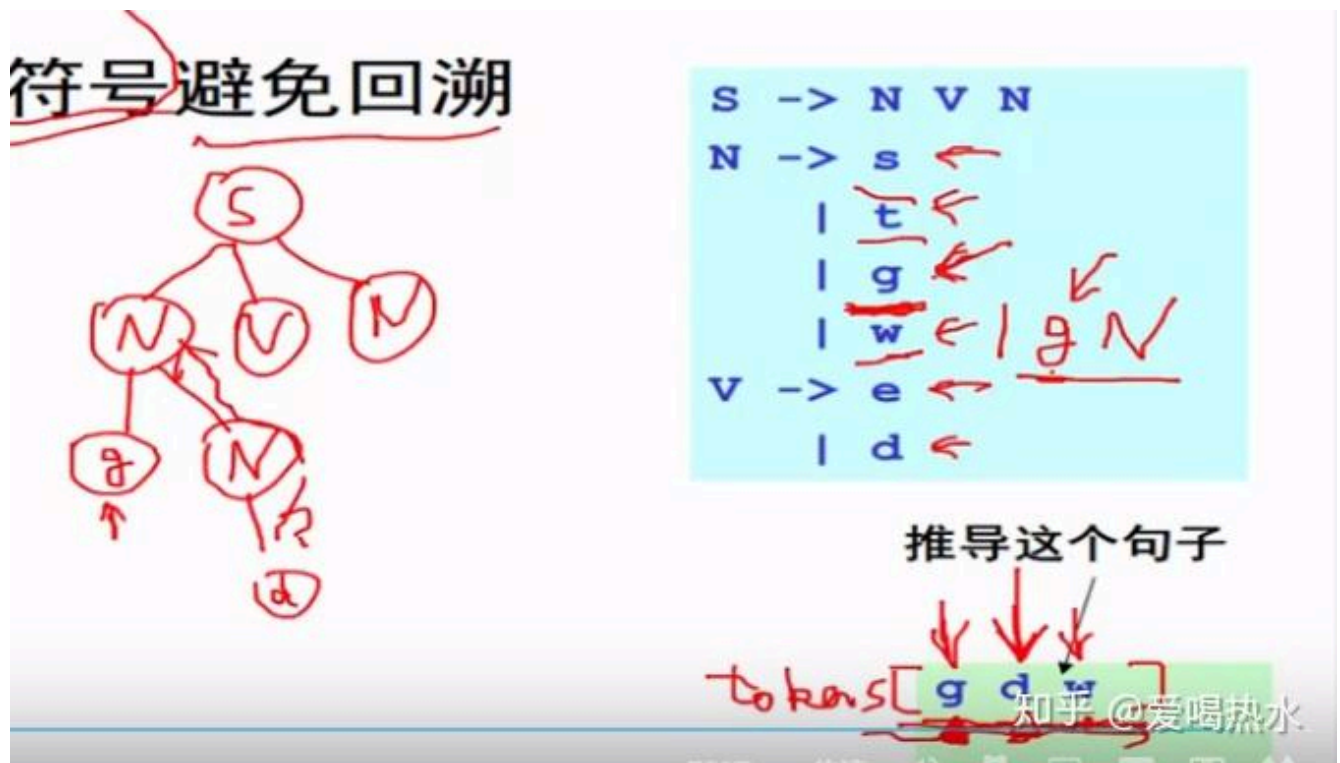
云课堂

自顶向下算法的最大问题就是因为需要回溯，所以造成效率相对低下。这就引出了递归下降分析算法和LL(1)分析算法，其主要思想是利用前看符号来避免回溯。

1. 首先建立抽象语法树

2. 当需要替代N的时候我们有4个选择, s、t、g、w, 我们这时候可以看看输入串, 因为输入串为gdw, 第一个字符为g, 而N的右部又有g, 所以我们可以直接选择g。
3. 接下来就是V, 我们同样可以通过前看符号来确定第二个为d, 同理第三个可以确定为w。
4. 这样我们不需要进行回溯了

但是有这样一种情况, 当N可以取两个符号, 如gN(如下图所示)



这时候再次进行选择的时候, 虽然知道是g开头, 但是我们不知道该选g还是gN。虽然我们两个都可以选, 但是如果选择不对, 还是会造成回溯。

递归下降分析算法

递归下降分析算法

■ 也称为预测分析

- 分析高效（线性时间）
- 容易实现（方便手工编码）
- 错误定位和诊断信息准确
- 被很多开源和商业的编译器所采用
 - GCC 4.0, LLVM, ...

■ 算法基本思想：

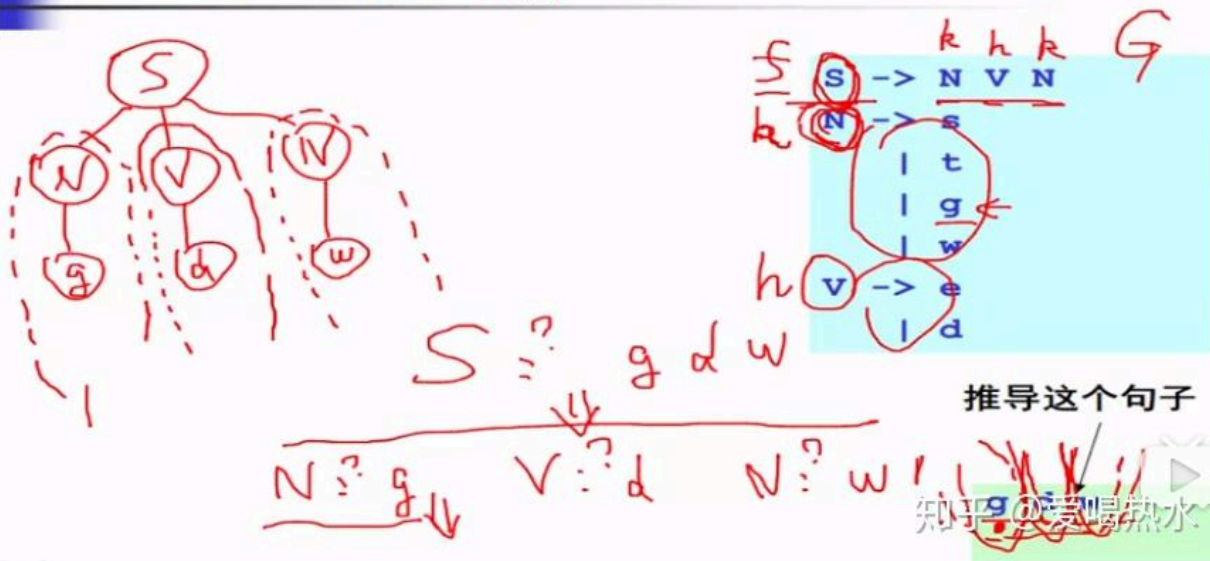
- 每个非终结符构造一个分析函数
- 用前看符号指导产生式规则的选择

知乎 @爱喝热水

示例

分治法

云课堂



我们先画出语法分析树

- N由g来替换
- V由d来替换
- N由w来替换

可以看出，每一个子树都和推导的句子的部分对应。这样的——对应的思想，即为计算机中的分治法。将初始符号S能否推导出一个句子，转换为将S分为N、V、N，再递归的变成，N是否能推导出g，V是否能推导出d，N能否推导出w。将一个大问题，一步步拆分为一个个小问题。

可以写出三个函数

- 分析S的函数叫f
- 分析N的函数叫k
- 分析V的函数叫h

函数f可以递归调用k,h。

```
parse_S()  
    //递归调用N、V、N函数  
    parse_N()  
    parse_V()  
    parse_N()  
parse_N()  
    token = tokens[i++] //读取句子中的下一个字符  
    if (token==s||token==t|| //如果读取到的符号在{s,t,g,w}的四个字符中，则匹配成功  
        token==g||token==w)  
        return;  
    error("..."); //否则报错  
parse_V()  
    token = tokens[i++]  
    ...// leave this part
```

一般的算法框架

```

parse_X()
  token = nextToken()
  switch(token)
    case α: // β 11 ... β 11
    case ...: // β 21 ... β 21
    case ...: // β 31 ... β 31
    ...
  default: error ("...");

```

$x \rightarrow$

β_{11}	β_{12}	β_{1j}	β_{1k}
\vdots	\vdots	\vdots	\vdots
β_{21}	\dots	β_{2j}	j
\vdots	\vdots	\vdots	\vdots
β_{31}	\dots	β_{3k}	k
\vdots	\vdots	\vdots	\vdots

$B \rightarrow b \dots$

$D \rightarrow d \dots$

$\text{token} = a$
 $\text{parse_B()};$

$\text{token} = c$
 $\text{parse_D()};$

知乎 @爱喝热水

1. 左侧为非终结符 x ，右侧有若干个右部，如 $\beta_{11}...\beta_{1i}$ ，共有 i 个。从 $\beta_{21}...\beta_{2j}$ 都是终结符或者非终结符，共有 j 个符号。同理， $\beta_{31}...\beta_{3k}$ 共有 k 个符号。
2. 于是我们就有了这样的一个名为`parse_X()`的函数，`token`等于下一个`token`，然后对`token`分情况进行讨论，如果是第一个情况，就对应第一条的 $\beta_{11}...\beta_{1i}$ 。同理，如果是第二个情况则走到第二个分支上，否则将报错。
3. 在 $\beta_{11}...\beta_{1i}$ 的 i 个字符中，其中可能有终结符，也有可能非终结符。
4. 例如，当 i 等于4的时候，假设 $\beta_{11}...\beta_{14}$ 分别为`aBcD`。有两个为终结符，有两个为非终结符。
5. 当我们发现`token=a`的时候，我们将走到`a`对应的分支上，当处理完`a`，我们将读入大写`B`，这时候就要递归调用`parse_B`了。（当读入的为终结符的时候则进行比较，非终结符则递归调用）
6. 再比较`c`，最后再递归调用`parse_D`
7. 处理完成

对算术表达式的递归下降分析

对算术表达式的递归下降分析

```
// a first try
```

```
parse_E()
```

```
3 token = tokens[i++]
```

```
if (token==num)
```

```
    ? // E+T or T
```

```
    else error("...");
```

parse_E();

* parse_T();

→ parse_T();

```
E -> E + T num
    | T num
T -> T * F
    | F
F -> num
```

对这个句子做语法分析

知乎@爱喝热水

我们写出parse_E函数，先读入一个数组，然后再判断要走哪一个分支，如果走第一个分支，就先调用parse_E函数，再吃掉+号，最后再调用parse_T。如果走第二条分支则直接调用parse_T。

对算术表达式的递归下降分析

```
// a first try
```

```
parse_E()
```

```
3 token = tokens[i++]
```

```
if (token==num)
```

```
    ? // E+T or T
```

```
    else error("...");
```

parse_E();

* parse_T();

→ parse_T();

```
E -> E + T num
    | T num
T -> T * F
    | F
F -> num
```

对这个句子做语法分析

知乎@爱喝热水

在判断要走哪一条分支的时候，我们可能会进行回溯。但是我们可以根据已有的经验避免掉回溯。

对算术表达式的递归下降分析

```
// a second try
parse_E()
  parse_T()
  token = tokens[i++]
  while (token == '+')
    parse_T()
    token = tokens[i++]
  Eof
parse_T()
  parse_F()
  token = tokens[i++]
  while (token == '*')
    parse_F()
    token = tokens[i++]

```

$E \rightarrow E + T$
 $\quad \quad \quad | T$
 $T \rightarrow T * F$
 $\quad \quad \quad | F$
 $F \rightarrow \text{num}$

$T + T + \dots + T$ Eof
 $T * T * \dots * T$
 $F * F * \dots * F$ 0 - ∞

对这个句子做语法分析

知乎 @爱喝热水
3+4*5