第一章 引论

1: 编译器和解释器？

答：编译器是一个程序，它可以阅读以某一种语言(源语言)编写的程序，并把该程序翻译成为一个等价的、用另一种语言编写的程序。编译器最重要的任务之一是报告它在翻译过程中发现的源程序中的错误。

解释器是另一种语言处理器。它并不通过翻译的方式，而是直接利用用户提供的输入执行源程序中的指定操作。

ps：JAVA 语言结合了编译器和解释器，Java源程序首先被编译成字节码的中间层，然后由一个虚拟机对字节码解释执行。实现一处编译处处执行。

2: 一个编译器由分析(也称为前端，扫描源程序的语法语义，并生成源程序的符号表和中间表示)和综合(也称为后端，将中间表示和符号表在一起构造出用户期待的目标程序)组成。

3: 词法分析、语法分析、语义分析的实现流程？

答：词法分析。将源程序的字符流，按照符号表将它们组织为有意义的词素序列，并将生成的词法单元传递给语法分析。

语法分析。将词法生成器传递过来的词法单元建立起语法树，树中节点表示运算，子节点表示运算的分量。

语义分析。收集类型信息，并作类型检查，编辑器检查每个运算符是否具有匹配的运算分量。

4: 中间代码生成、代码优化、代码生成三者的作用？

答：中间代码生成的中间代码易于被翻译成机器语言；代码优化可以加快编译速度或者能耗更低；代码生成需要将中间代码合理分配寄存器以存放变量的值。并生成机器代码。

5: 编译技术的应用？

答：1，高级程序设计语言的实现。2，针对计算机体系机构的优化。3，新计算机体系结构的设计。4，程序翻译

6: 静态作用域和动态作用域？

答：如果仅通过阅读程序就可以确定一个声明的作用域，则称为静态作用域(C,C++,JAVA等)，反之为动态作用域。

例如：＃define a (x+1) 当调用a时，x取值多少取决于当时的环境，所以属于动态作用域。

第二章 一个简单的语法制导翻译器

1: 上下文无关文法定义？

答：上下文无关文法由一个终结符号集合、一个非终结符号集合、一个产生式集合、指定一个非终结符号为开始符号，四部分组成。文法自然地描述了大多数程序设计语言构造的层次语法结构。

2: 文法推导过程？

答：从开始符号出发，不断将某个非终结符号替换为该非终结符号的某个产生式的体。可以从开始符号推导得到的所有终结符号串的集合称为该文法定义的语言。

3: 什么是语法分析树？

答：1，根节点的标号为文法的开始符号。2，每个叶子结点的标号为一个终结符号或空。3，每个内部结点的标号为一个非终结符号。

4，如果 A 为某个内部结点标号，并且它的子节点从左至右为X1，X2，X3,...,Xn，则必然存在产生式 A -> X1X2X3...Xn, 其中 X1...Xn 既可以是终止也可以是非终止符号。

4: 何为文法二义性？

答：一个文法可能有多棵文法树。即一个终结符号串是两颗以上文法分析树的结果。

5: 语法制导翻译过程定义？

答：语法制导翻译是通过向一个文法的产生式附加一些规则或程序片段而得到的。语法制导翻译方案是一种在文法产生式中附加一些程序片段来描述翻译结果的表示方法。

6: 综合属性，某个属性在语法分析树结点 N 上的值是由 N 的子节点以及 N 本身的属性值确定，那么这个属性就称为综合属性。综合属性有个特点：只需要对语法分析树自底向上遍历，就可以计算出属性的值。

7: 语义动作：被嵌入到产生式体中的程序片段称为语义动作。一个语义动作用花括号括起来，并写入到产生式体中，它的执行位置也由此指定。

8: 两种语法制导翻译方法。

（1）通过将属性值附加在语法分析树的结点上，从而得到翻译结果。

（2）通过在产生式中附加语义动作（程序片段），来逐步生成相同的翻译结果。

9: 语法分析定义：语法分析是决定如何使用一个文法生成一个终结符号串的过程。

10: 语法分析可以归为两大类：自顶向下分析法和自底向上分析法。其中自顶向下分析法先挑选第一个非终结符号为根节点，然后为根节点选择一个产生式，然后在它的子节点中自左向右依次为非终结符号选择产生式，以此类推。

递归下降分析法是自顶向下分析法中的一种。该方法是用一组递归过程处理输入，文法的每一个非终结符号都对应一个相关联的过程。

第三章 词法分析

1: 词法分析是将源码解析成一组长度相等的二元组，例如：if (i == 9) m = 9; （单词类别，自身值）解析成如下：

（10（在定义符表中的序号），\_）,

（21（‘（’在定义符表中的序号），\_）,

（id，（i 在符号表中的指针）），

（23（在定义符表中的序号），\_）,

（id，9的指针）,

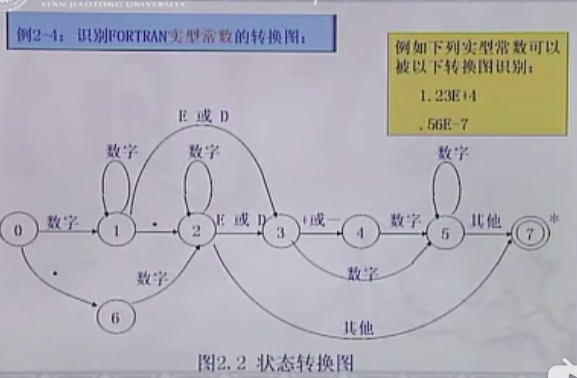
（22（‘）’在定义符表中的序号），\_）,

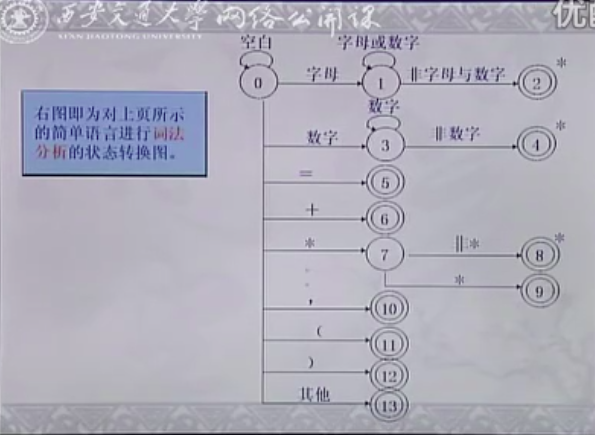
（id，（m的指针））等等

在词法分析时通常有一对输入缓冲区作为输入缓冲。

2: 状态转换图定义及其作用？

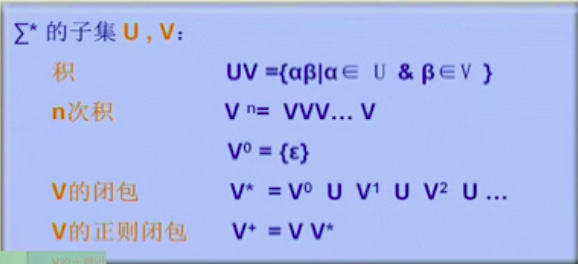
答：状态转换图是构造词法分析器的一个中间步骤，由一组被称为状态的结点或圆圈组成。转换图中的每个状态代表了可能在这个过程中出现的情况，以及应该做的相关处理。是构造词法分析程序的有力工具。用来识别单词。如下图：

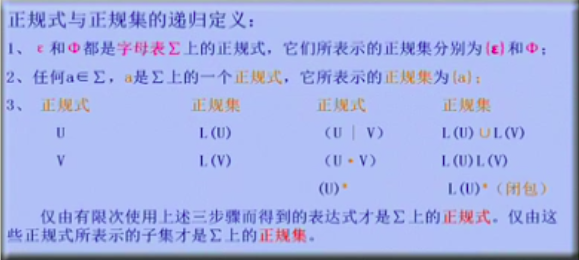




3: 正规式和正规集

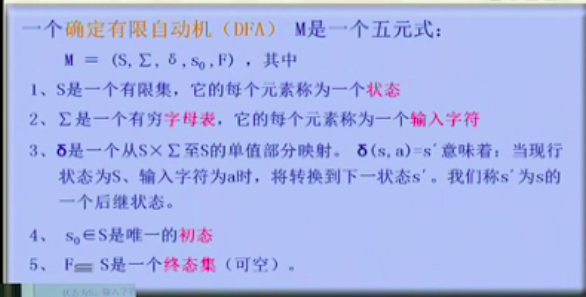
正规式是由集合经过或、积、闭包等运算形成的一个式子。正规集表示该正规式所表示的一个集合。例如：令 R ＝ {A, B, C, ..., Z}，令 M ＝ {0，1，2，...，9}。则 R(R|M)\* 为一个正规式（\* 代表闭包运算）,该正规式表示所有以字母开头，并且由字母数字组成的 “标识符” 集合。正规式可以用来描述所有类型的单词。 标识符、数字、关键字等等。





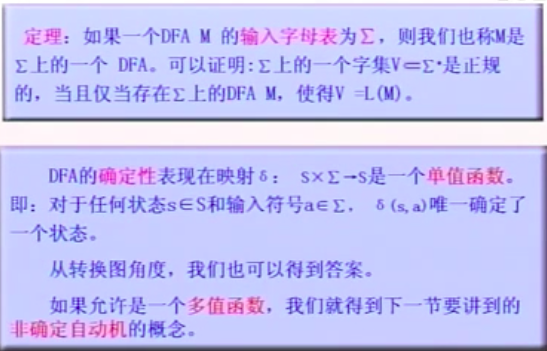
4: 有限状态自动机（DFA）定义

答：有限自动机是一个五元式，为了将状态图精确化。记作 M ＝ （S、∑、δ、s0、F）。S 对应状态图中的结点，经过输入符号 a 后转变成 S', 记作 δ（S，a）＝ S', 其中 a 为 ∑ 中的一个符号。S0 是 S 的唯一的初态。F 是 S 的终态集（可空）。一个 DFA 可以用一个矩阵表示，行表示状态，列表示输入字符，矩阵元素代表 δ（S，a）的值。该矩阵称为状态转换矩阵。



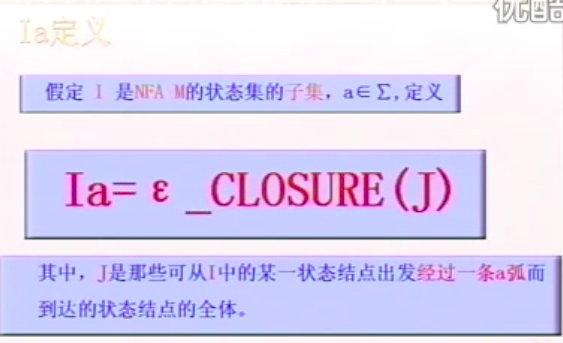
5: 定理：如果一个 DFA M 的输入字母表为 ∑，则我们也称 M 是 ∑ 上的一个 DFA。可以证明：∑ 上的一个子集 V 是正规的，当且仅当存在 ∑ 上的 DFA M，使得 V＝L(M)。

DFA 的确定性表现在映射 δ：S \* ∑ -> S' 是一个单值函数。即 对于任何状态 s 属于 S 和输入符号 a 属于 ∑，δ（s，a）唯一确定了一个状态 S'。如果允许是一个多值函数，则是非确定自动机的概念即 NFA。



6: NFA 非确定自动机。和 DFA 的区别在于 δ，也就是状态图上的弧的标志，如果出现字(多个字符连在一起叫字)或者空则是 NFA，否则 DFA。

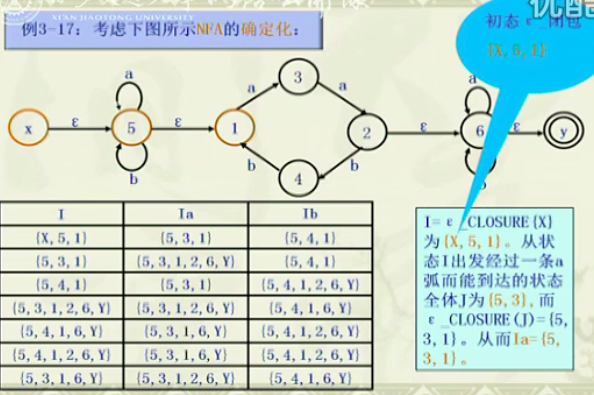
7: I 的 ℇ 闭包定义：I 属于 M，s 为 I 上任意一个状态，那么从 s 出发经过任意条 ℇ 弧而能到达的任何状态 s' 都属于 I 的 ℇ 闭包。



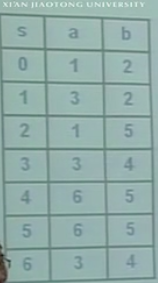
8: NFA 可以通过子集算法变换成 DFA 便于计算。

子集算法：将初始状态做 I 的 ℇ 集合为第一组初态 S1。将 S1 和 M 中的每一个符号进行运算的到对应的状态，将得到的状态做为矩阵第二行，继续进行上述的运算，直到的到全部的状态。

9: DFA 化简算法，将 M 划分成终态组和非终态组。使用 ∑ 中的每一个符号来 “打” 这两个分组中的每一个元素，结果一致的可以合并为一个。



对上表的集合元素重命名，简化为DFA：



10 Lex 过程及原理

（1）生成每种单词的语法规则（正规式）传递给 LEX，Lex 给每种单词起个名称 Pi。

（2）Lex 内部将所有的正规式生成对应的 NFA，并根据子集算法变换成 DFA 后进行简化操作。

（3）控制器读取单词并根据 DFA 生成的转换矩阵表，进行转换。最终生成二元式。

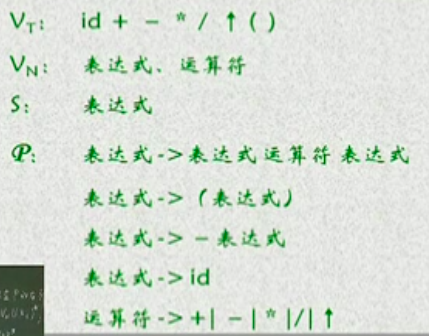
第四章 语法分析器

1: 文法：描述语言的语法结构的形式规则。目的是解决语言的有穷说明问题，包含对语法的描述，但是却不表达任何语义。

2: 上下文无关文法定义？

答：上下文无关文法由一个终结符号集合、一个非终结符号集合、一个产生式集合、指定一个非终结符号为开始符号，四部分组成。文法自然地描述了大多数程序设计语言构造的层次语法结构。G（Vt、Vn、S、P），S 属于 Vn 称为开始符号，P 有限的产生式集合

例：算术表达式的文法



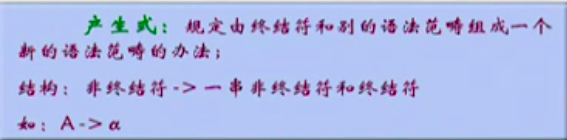
3: 终结符号和非终结符号，开始符号

终结符号是用于组成语言中的串的基本符号，与程序语言中的“单词”是同义词。

非终结符号是标记某种串的集合的特定符号，与“语法变量”，“语法范畴”是同义词；如：表达式，运算符都表示一个串的集合。

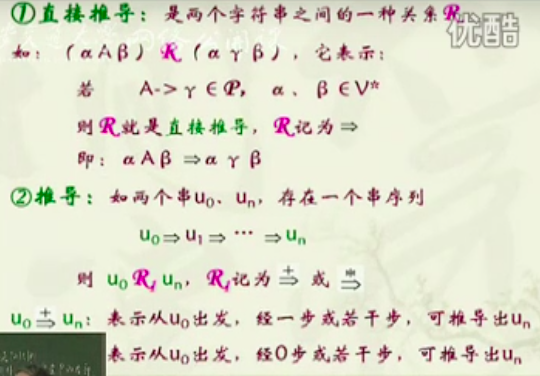
开始符号：一个 Vn 标记感兴趣的语法范畴。

产生式：A -> a：非终结符号 -> 一串非终结符号和终结符号



4 : 文法推导过程？

答：从开始符号出发，不断将某个非终结符号替换为该非终结符号的某个产生式的体。可以从开始符号推导得到的所有终结符号串的集合称为该文法定义的语言。



5: 什么是语法分析树？

答：1，根节点的标号为文法的开始符号。2，每个叶子结点的标号为一个终结符号或空。3，每个内部结点的标号为一个非终止符号。

4，如果 A 为某个内部结点标号，并且它的子节点从左至右为X1，X2，X3,...,Xn，则必然存在产生式 A -> X1X2X3...Xn, 其中 X1...Xn 既可以是终止也可以是非终止符号。

6: 何为文法二义性？

答：一个文法可能有二棵文法树。即一个终结符号串是两颗以上文法分析树的结果

处理二义性对语法分析不便，希望：（1）判断二义性。（2）控制充分条件，消除二义性

7: 为了避免二义性的限制：

（1）G 中不含有 P->P

（2）每一个 P 都必须有用处。 即 P 在句型中出现过，且对 P 不存在不终结的回路。

消除二义性的方法：使用代数方法重写一个文法代替原来的文法，并且消除二义性。

8: 自上而下分析主旨：从文法开始符号出发，自上而下的为输入串建立一个语法树。

实现：每一个非终结符号对应一个递归子程序，在只生成两个串的文法，过程无需递归，而对生成无数个串的文法，递归是不可避免的。

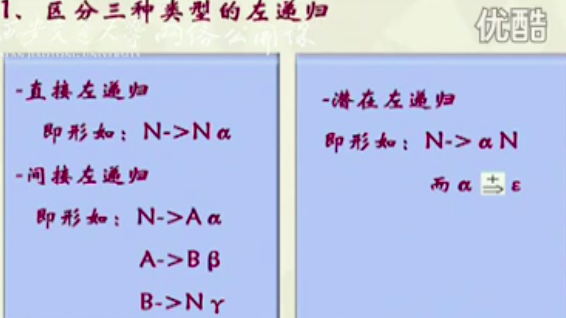
递归子程序：是一个布尔过程，一旦发现它的某个候选式与输入串匹配，它就按此式扩充语法树，并返回 true，指针移过已匹配子串。否则，返回 false，保持原来的语法树和指针不变。

9: 回溯

例如 A -> ab | a, 在分析过程中使用 ab 替换 A 后发现并不能正确生成语句，则回溯到 a 替换 A 继续尝试，直到分析成功。因此必将 A 的输入指针存放在一个局部变量中。

当产生式很多时，回溯会导致语法匹配过程很慢，影响编译效率，应该尽量消除回溯。

10：左递归：递归下降语法分析器有可能进入无限循环。当出现如下所示的左递归产生式时，分析就会出现无限循环：A -> Aa，在这里，产生式体最左边的符号和产生式头部的非终结符号相同。所以 A 将被递归调用，进入无限循环。

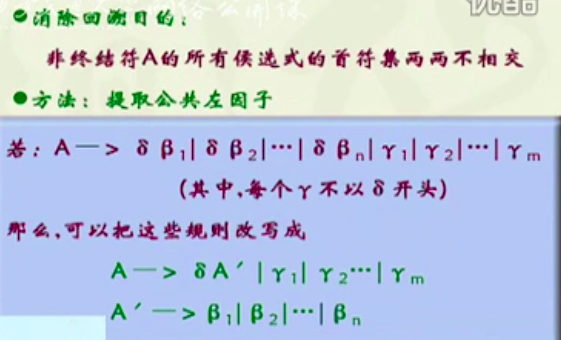


＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊ 第15讲 0 min

11: 左递归的消除公式：A -> Aa | 𝛽, 变换成 A -> 𝛽A', A' -> aA' | ℇ

12: 避免回溯的定理：A 属于 Vn，就会有许多的 FIRST（ai），如果 A 的任意两个候选式 ai，aj（ai，aj可以是终结或者非终结）均有 FIRST（ai）∩ FIRST（aj） ＝ 𝜙，意味着 A 的每一个候选式a推导后所得的字符串第一个 Vt 均不同。于是，对于输入符号 a，如果 a ∈ First（ai），则 a ∉ FIRST（aj）（i ≠ j），因此对 A 的展开式无疑选择 ai 才可能命中。

简单说，就是某一个非终结符号的所有产生式的开头终结符号都不同。这样在选择产生式的时候就无需回溯，一次匹配失败则可以认定失败。



12: 递归下降分析法：指在语法分析过程中会多次出现直接／间接递归的情况。例如：

E -> TE'

E' -> +TE' | ℇ

T -> FT'

T' -> \*FT' | ℇ

F -> (E) | i

递归下降分析文法要求消除左递归和回溯。每一个 Vn 符号都对应一个独立的递归程序，句子匹配的过程是一个递归的过程。

13: 预测分析法 由四部分组成：

（1）一个输入：含有要分析的串。

（2）一个栈：包含了一系列的文法符号，栈底是＃，S 先进栈。

（3）分析表：二维数组M[A][a], 其中 a ∈ Vt，A ∈ Vn

（4）输出：根据分析表内元素做规定的语义动作。

相比递归分析法，预测分析法避免了递归的过程效率更高，而且可以更早的发现错误。



14: 预测分析过程以及栈的实质？

栈里实际上是还没有被匹配的残缺的句型，最开始时是开始符号。分析时分三种情况，设栈顶符号为 X（终结非终结都有可能），当前输入指针为 a（Vt）

（1）a == X == # 句子匹配成功，属于该句型

（2）a == X != # 该字符匹配成功，a 后移，X 弹栈

（3）X 为非终结符号，则查找预测分析表，找到对应的产生式，将 X 弹栈，并用找到的产生式倒序入栈。

15: FOLLOW（A）计算规则：

（1）＃属于 FOLLOW（S）

（2）若存在 A > aB𝛽，则FIRST（𝛽）∈ FOLLOW（B），除 ℇ 外。

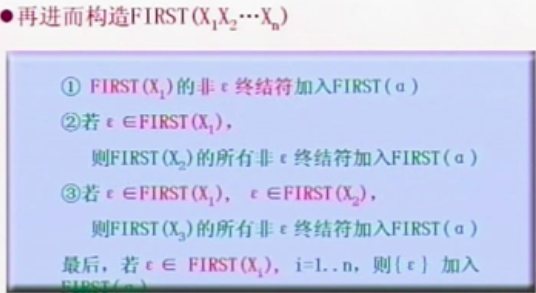
（3）若有 A -> aB 或 A-> aB𝛽 且 ℇ ∈ FIRST（𝛽），则 FOLLOW（A）∈ FOLLOW（B）

FIRST(A) 计算规则：

（1）若 X ∈ Vt 则 FIRST（A）是 {X}

（2）若 X ∈ Vn 且 X -> 𝛼a, 则 {𝛼} ∈ {X}。若 X -> 𝜀 则 {𝜀} ∈ {X}

（3）若 X ∈ Vn 且 X -> Y……, Y ∈ Vn, 则 FIRST（Y）∈ FIRST{X}



16 : 预测分析表如何构造？

按照 a => ? 将产生式分成三种。4

（1）A => a.....

（2）A => ℇ

预测分析法使得每次的输出都是有效的（如果是错误则在开始就会报错），优化了递归下降带来的缺点。

17: 构造预测分析表，输入 G1 文法，输出分析表 M

（1）对文法的每一个 A -> 𝛼，做（2）和（3）

（2）对于任意 a ∈ FIRST（𝛼），把 A -> 𝛼 加进 M[A,a]

（3） 若 ℇ ∈ FIRST（𝛼），则把 A -> 𝛼 加进 M[A,b]，b ∈ FOLLOW（A）

18: LL(1) 文法满足条件是当且仅当 G1 的任意两个不同产生式 A -> 𝛼 | 𝛽 满足下面条件

（1）FIRST（𝛼）和 FIRST（𝛽）是不相交的集合

（2）当 ℇ ∈ FIRST（𝛽）时 FIRST（𝛼）和 FOLLOW（A）不相交，反之当 当 ℇ ∈ FIRST（𝛼）时 FIRST（𝛽）和 FOLLOW（A）不相交。

当不满足 LL（1）语法时可以根据程序语言的特性舍弃多余的 M［A，a］

19: 自底向上推导：需要使用一个栈为辅助结构，将栈中的终结符号不断的规约，最终规约成某一条产生式的过程。

规范推导：即最右推导。

规范句型：由规范推导所得到的句型称为规范句型。

规范规约：是关于句型 a 的一个最右推导的逆过程，也称为最左规约。

20: 短语定义：G，S，S =>\* a A 𝛿,如果 S =>\* a 𝛽 𝛿 且 A =>\* 𝛽，则称 𝛽 是句型 a 𝛽 𝛿 的相对于 A 的一个短语。(=>\* 指的是推导 0 次或多次出来的)

直接短语：若 A -> 𝛽, 则称 𝛽 是句型 a 𝛽 𝛿 的相对于 A 的一个直接短语。

句柄：最左直接短语。

20: 实现规范规约的一个方便途径是使用一个栈和一个输入缓冲区，用 ＃ 表示栈底和输入的结束。

分析程序的动作：

（1）移进，下一输入符号入栈顶。

（2）把句柄按产生式的左部进行规约，规约的时候需要从栈顶循环遍历到栈底，找到可规约的句柄。

（3）分析程序报告成功。（检测到两边都出现 ＃）

21: 直观算符优先文法：只有加减乘除的文法，通过运算符优先级表就可以实现。

根据运算符栈内外的符号优先级决定入栈还是计算。

22: 算符优先文法定义：

（1）任何产生式的右部都不存在两个连续的非终结符号，例如 ...QR...，则我们称之为算符文法。

（2）假设 G 是一个不含有 ℇ 产生式的算符文法。对于任何一对终结符号 a，b 有：

a=b,当且仅当 G 中含有形如 p -> ...ab... 或 p -> ...aQb... 的产生式；例如：（E）则 （ ＝ ）

a < b, 当且仅当 G 中含有形如 P -> ...aR... 的产生式，而 R =>+ b...,或 R ＝> Qb...;

a > b. 当且仅当 G 中含有形如 P -> ...Rb... 的产生式，而 R =>+ ...a,或 R ＝> ...aQ;

（3）两个非终结符号的关系至多有三种之一 > = <，则称 G 为一个算符优先文法。

简单来说，谁先规约谁优先。

23: 算符优先表的构造：

构造出列为 a1,a1,...,am，行为 a1,a1,...,am 的一张算符优先表。

首先需要两个集合：FIRSTVT（p），LASTVT（p）。

FIESTVT（p）＝｛a｜p =>+ a... 或 p =>+ Qa...，a 属于 Vt 而 Q 属于 Vn｝

LASTVT（p）＝｛a｜p =>+ ...a 或 p =>+ ...aQ，a 属于 Vt 而 Q 属于 Vn｝

根据上面的两个集合，找出候选式中 ...aP... 那么任意 b 属于 FIRSTVT(P) 有 a < b; 同理，...Pb... 中对于任意 a 属于 LASTVT（P）有 a > b

24: FIESTVT 和 LASTVT 的计算算法

（1.1）P -> a... 或 P -> Qa...，则 a ∈ FIRSTVT(P)

（1.2）若 a ∈ FIRSTVT(Q)，且有产生式 P -> Q...，则 a ∈ FIRSTVT(P)

（2.1）P -> ...a 或 P -> ...aQ，则 a ∈ LASTVT(P)

（2.2）若 a ∈ LASTVT(Q)，且有产生式 P -> ...Q，则 a ∈ LASTVT(P)

上述两个算法的确定需要借助栈来求得。算法简单描述如下：

构造出列为 P1,P2,...,Pn，行为 a1,a1,...,am 的一张 FIRSEVT 表。

先计算出所有的 Pn->a1 和 Pn->Qa1, 这个 Pn 代表以 a1 为开始符号，将 [Pn,a1] 标记为 true, n然后将这些 Pn 加入一个栈，不断的取栈顶，找到形如 Pm －> Pn... 的产生式，此时将 Pm 也入栈，同时将 [Pm,a1] 也标记为 true，直到栈为空。

之后在找所有的 Pn->a2 和 Pn->Qa2，直到求到 am 结束为止。至此找到所有的非终结符号的 FIRSTVT 集合。LASTVT 算法同理。

24: 短语：G，S，S=> a A 𝛿,如果 S => a 𝛽 𝛿 且 A => 𝛽，则称 𝛽 是句型 a A 𝛿 的一个短语。

素短语：指至少包含一个终结符号，并且除了自身外不再含有任何更小的素短语。

最左素短语：句型最左边的素短语。

算符优先算法，每次寻找栈顶的素短语（最左素短语）然后进行规约。

25: 最左素短语是满足一下条件的最左子串

算符优先文法一般形式：＃N1a1 N2a2 ... Nnan Nn+1＃ ai ∈ Vt, Ni ∈ Vn|ℇ

满足一下＝＝以下条件的最左子串: Nj aj...Ni ai Ni+1，其中：

aj-1 < aj, aj = aj+1 = ... = ai-1 = ai, ai > ai+1

即：ai < 最左素短语 > aj，找最左素短语时不考虑其中的 Vn 符号。

26: 规范规约与算符优先文法的比较：

（1）规范规约找最左直接短语（句柄）；而算符优先找最左素短语。

（2）规范规约中的句柄是文法符号串，而算符优先文法的最左短语至少含有一个 Vt 符号。

（3）规范规约找句柄是穷举的方式，找错了需要回溯；而算符优先找 ...<====>...。

（4）规范规约速度慢于算符优先，原因是算符优先忽略了单非产生式。

27: 由于算符优先表占用空间较大，所以可以采用算符优先函数的方式优化空间。定义如下：

每个终结符号 𝜃 与两个自然数f（𝜃）和 g（𝜃）对应，使得：

若 𝜃1 < 𝜃2，则 f（𝜃1）< g（𝜃2）

若 𝜃1 = 𝜃2，则 f（𝜃1）= g（𝜃2）

若 𝜃1 > 𝜃2，则 f（𝜃1）> g（𝜃2）.

函数 f 称为入栈优先函数，g 称为比较优先函数。注：f 和 g 不一定能确定，如果存在则存在无数个 f 和 g，其中任意两个相差常数，类似于不定积分。

28: 建立 f 和 g 函数的步骤（如果有的话）：

（1）将所有优先级相等的 vt 符号归为一组。

（2）建立有向图，结点为上一步骤所归纳的所有组。对于任何 a 和 b，

若 a > b，则画 Fa -> Gb 弧，意味着f(a) > g(b).

若 a < b，则画 Gb -> Fa 弧，意味着f(a) < g(b).

（3）如果上一步构成环，则 f g 不存在。否则 f（a）＝ 从 Fa 组开始的路径和。g（a）＝从 Gb 组开始的路径和。

29: LR 分析法（规范规约的一种）。

LR 分析程序的实质：分析栈＋DFA。根据 DFA 中的状态转换表，决定当前输入操作入栈、规约、成功、失败，四个动作。其中：

sj 代表入栈操作，并且添加状态 j 如栈。

rj 代表规约，使用第 j 条产生式规约。产生式规约几个输入字符，对应的状态拿掉几个。

acc 代表成功。

30: LR 语法分析器

首先按照 Yacc 的语法规则输入，Yacc 自动生成 LR 语法分析表。控制程序每次查询 LR 语法分析表和当前栈中的状态，以及当前的输出来确定入栈、规约、接受、报错四种状态的输出。

第五章 中间代码生成（语义分析）

1: 语法制导翻译：在语法分析的过程中，随着分析的步步进展，根据每个产生式所对应的语义子程序（语义动作）进行翻译（生成中间代码）的办法。

即：每次规约都有一个动作，规约结束后动作出结果。

2: 标记说明：描述语义动作时，需要赋予每个文法符号X以种种不同方面的值，如X.type（类型），X.VAL（值）。

一个产生式中同一个符号多次出现，用上角标来区分 E -> E + E 表示为 E -> E(1) + E(2)

每个产生式内部的语义动作，写在该产生式之后的花括号内。

3: 中间代码的必要性：大部分的编译器都不直接产生目标代码，因为产生代码需要做优化，涉及到寄存器的分配问题，在语义分析阶段，很难有效的分配它们。

4: 逆波兰表达式定义：一般，若 e1，e2 为任意的后缀表达式，𝜃 为任意双目运算符，则 𝜃 作用于 e1 e2 所表达的结果用后缀 e1e2 𝜃 表示。

推而广之，𝜃 为 k 目运算符，则 𝜃 作用于 e1 e2 ... ek 所表达的结果用后缀 e1e2...ek 𝜃 表示。常用于算符表达式。

例如 用 ? 表示 if－then－else 则

if a then if c-d then a+c else a\*c else a+b -> a cd- ac+ ac\* ? ab+ ?

上述过程可以利用栈来计算，遇到运算符，则根据对应的k目取出位于栈顶的k个运算量来进行计算并将结果写回栈顶。

5: 后缀表达式的缺点：根据定义一个 if(e != 0)－then(x)－else(y) 的运算需要将 e != 0, x, y 依次入栈后进行计算，但是如果 x或者y中需要使用 e 为 除数，则会产生数学异常。

解决方案是引入标号，在后缀式中加入条件／无条件转移算符。将所有的后缀式存储在一维数组 POST[1...N] 中，每个元素是运算符或者运算分量(指向符号表)。转移算符如下：

p jump -> 转到 POST[p]

e1e2pjlt -> e1 < e2 时，转到 POST[p]

e p jez -> 若 e=0,转到 POST[p]

if e then x else y 变换为：e p1 jez x p2 jump p1:y p2:

6: 语法制导生成后缀式（逆波兰表达式），通过为每个产生式添加语义动作，则在语法分析时每次规约都会产生一条语义动作。例如：

E -> E(1) op E(2) {E.code = E(1).code || E(2).code || op}/{print op}

E -> (E(1)) {E.code = E(1).code}/{}

E -> id {E.code = id}/{print id}

7: 三元式构成 op ARG1 ARG2

ARG1 和 ARG2 都是指示器，指向符号表的某项或者是三元式表自身的某项。op 通常用整数编码。例如：

X = A + B \* C 的三元式

（1）\* B C －－－ B C 中存放的是词法分析时保存的符号表地址

（2）+ A （1）－－－ （1）代表三运式表中的序号，可以是任意有效值

（3）= X （2）

8: 语法制导生成三元式的语义动作

E -> E(1) op E(2) ｛E.VAL＝ TRIP(op, E(1).VAL, E(2).VAL)｝

E -> (E(1)) {E.VAL = E(1).VAL}

E -> -E(1) {E.VAL = TRIP(@, E(1).VAL, -)}

E -> id {E.VAL = ENTRY(id)}

注解：ENTRY 代表获取符号表入口的语义函数。

TRIP语义函数生成一个三元式，并加入到三元式表中，最后将表中对应的号码交给 E.VAL。

9: 用树型结构来表示一个表达式或者语句

简单变量或者常数的树就是该变量或者常数自身。一般叶子表示运算量，内节点表示 OP。

10: 语法制导产生树：

E -> E(1) op E(2) ｛E.VAL＝ NODE(op, E(1).VAL, E(2).VAL)｝

E -> (E(1)) {E.VAL = E(1).VAL}

E -> -E(1) {E.VAL = UNARY(@, E(1).VAL)}

E -> id {E.VAL = LEAY(id)}

11: 四元式相比三元式多了一个存放结果的临时变量，这个临时变量可以不限制的添加，在优化时压缩。这个临时变量可以通过符号表的方式存放，也可以单独开辟一块区域来存储。虽然比三元式代价高一些，但是避免了三元式调整顺序带来的问题。

几个语义变量和过程：

NEWTEMP：函数过程。每次调用时，它都回送一个代表新临时变量名的整数码作为函数值。

ENTRY（i）函数过程，查找对应符号表的位置

E.PLACE 和非终结符号E相联系的语义变量，表示存放E的值的变量名在符号表的入口或者整数码（为临时变量的话）

GEN(OP, ARG1, ARG2, RESULT): 语义过程，把四元式（）填入四元式表

12: 简单算术赋值表达式的语义动作描述：

（1）A -> i:=E {GEN(:=, E>PLACE, \_, ENTRY(i))}

（2）E -> E(1) + E(2) {E.PLACE := NEWTEMP, GEN(+, E(1).PLACE, E(2).PLACE, E.PLACE)}

（3）E -> -E(1) {E.PLACE := NEWTEMP, GEN(@, E(1).PLACE, \_, E.PLACE)}

（4）E -> (E(1)) {E.PLACE := E(1).PLACE}

（5）E -> i {E.PLACE := ENTRY(i)}

注意：算术表达式应该能够对运算量进行类型转换，所以需要进行类型转换的四元式，语义规则和四元式如下：

{IF E(1).MODE = int AND E(2).MODE = int

Then E.MODE = int

ELSE E.MODE = r}

四元式：GEN（itr, A1, \_, T）

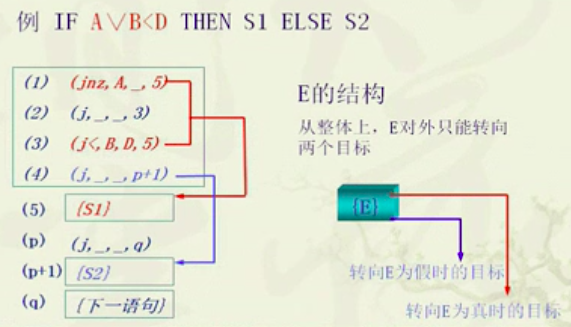
13: 布尔表达式

文法：E -> E ∧ E | E ∨ E | ¬E | (E) | i | i rop i, rop 代表表达式

布尔表达式用于求值或者条件控制

翻译布尔表达式的第一种方法是像处理算数表达式的方式来处理。

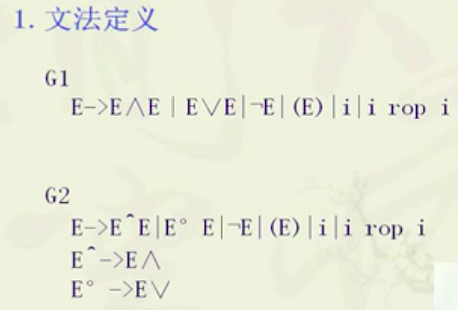
14: IF 语句的四元式表达：IF E THEN S1 ELSE S2 的翻译



困难点在于不知道 E 到底有多长，所以不知道最终的真跳转和假跳转的序号。可以采用拉链－返填法解决。即将跳转到真语句和假语句的四元式组成两个链。如下图：



15: 布尔表达式的文法以及语义动作：



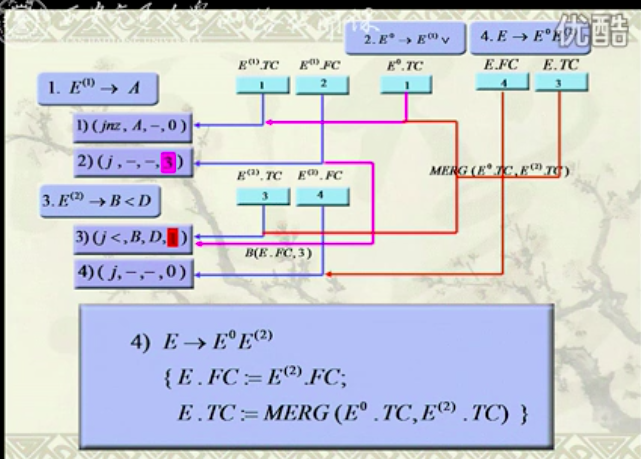


注：其中 NXQ 代表当前四元式栈中的四元式个数。随着四元式个数增加而加一。

BACKPATCH 是返填动作，将 NXQ 填入 E(1).FC 指向的四元式。

MERG: 将两个链合并成一个，即将某一个链的尾结点指向另外一个的链头。

16: 布尔表达式 A∨B<D 的语义生成过程：



17: 设计标号语句时分成两种情况：

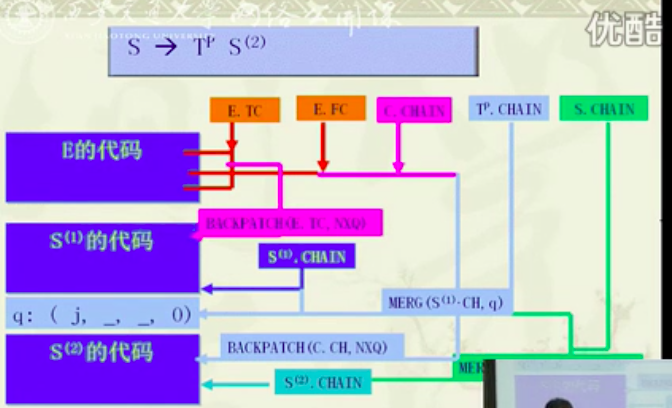
（1）先定义标号语句，后调用：

先将标号语句存入到符号表中，后续的调用过程根据标号名字查找符号表即可。

（2）先调用后定义标号语句：

利用拉链返填法将定义语句之前的所有调用语句拉成一个链，链头存放在符号表中的地址（标号语句未定义之前符号表中地址栏为空），当编译到标号语句的定义时，根据链头进行返填，直到地址为0。

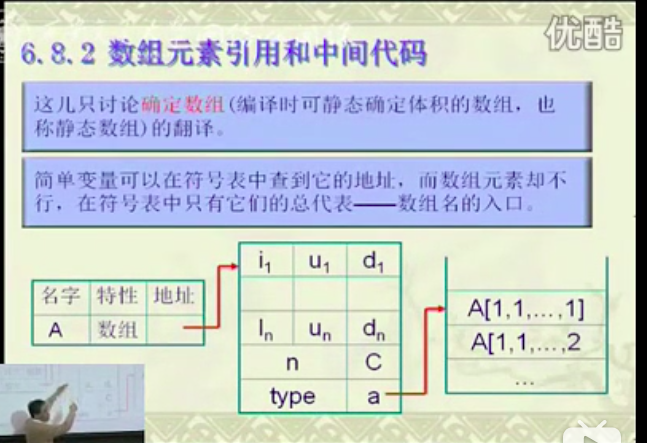
18: IF 语句的语法语义结构 32讲，.chain 用来记录语句执行完的出口。

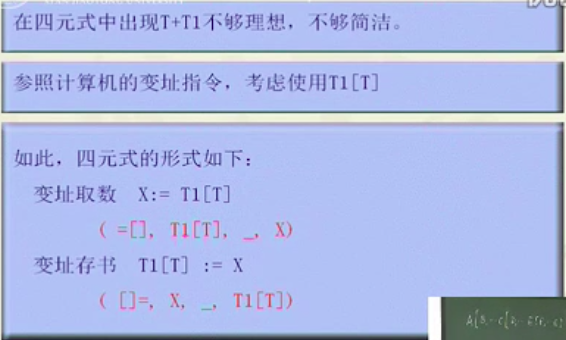


19: 对于 Case 语句的翻译，可以构造开关表，运行时查表即可，如果 case 项很多也可以构造哈希表。

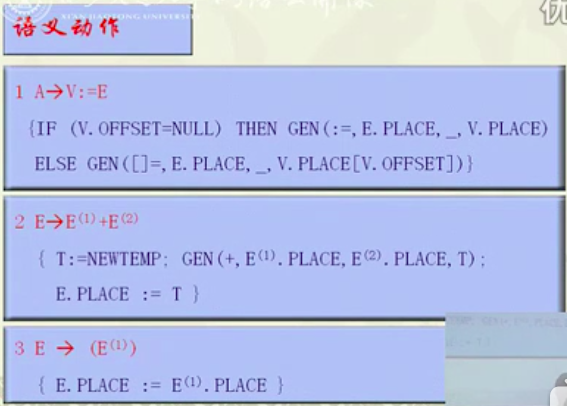
20: 数组元素在符号表中的存放形式以及四元式的表达：

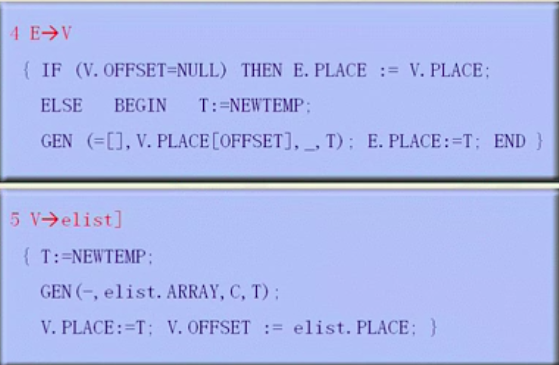
其中，第二张表称为数组的信息表，表中记录了数组每一维的上界、下界和界差（i，u，d）。如下图

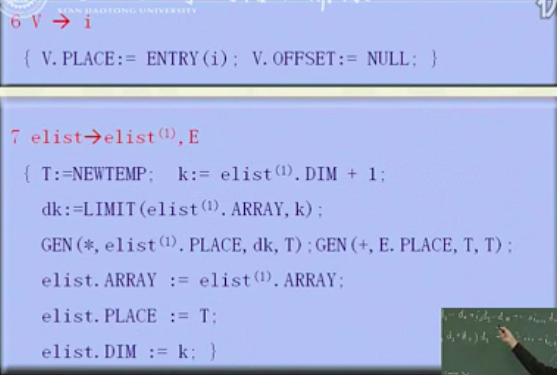


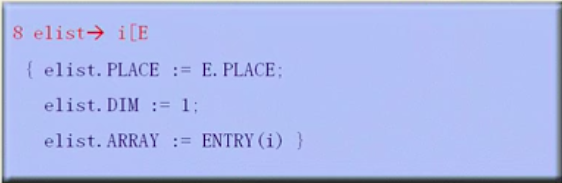


21：数组元素的语义动作：

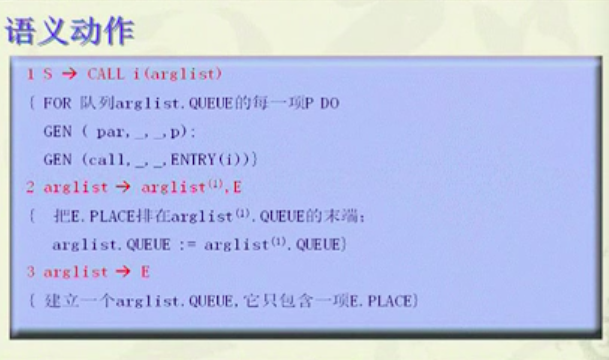




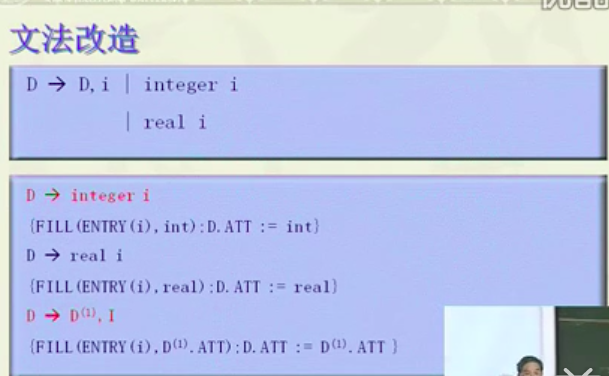




22: 过程调用的语义动作：



23: 变量说明语句的文法：



第六章 符号表