编译原理复习

## 第一章

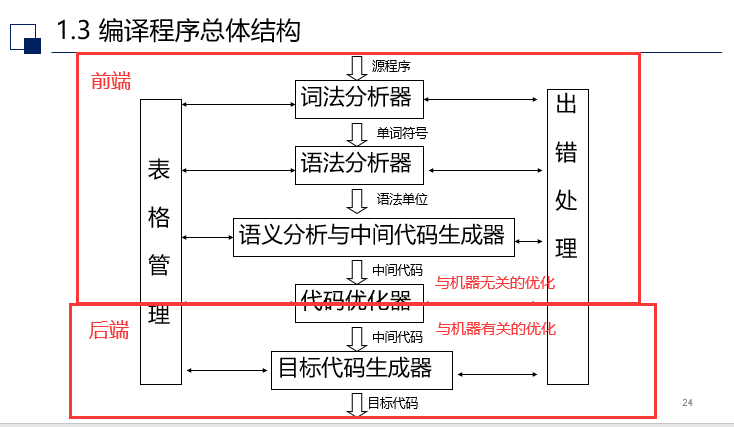
1. 编译系统的结构

编译系统 = **编译程序** + **运行系统**

编译程序：将**源程序完整地**转换成**机器语言程序或汇编语言程序**，然后再处理、执行的翻译程序。

翻译程序：将某一种语言描述的程序翻译成**等价的**另一种语言描述的程序

解释程序：一边解释一边执行的翻译程序





词法分析器作用：词法分析器从左到右扫描组成源程序的字符串，并将其转换成单词（token）串；同时要：查词法错误，进行标识符登记——符号表管理。

语法分析器作用：实现组词成句，将token串组成各类语法成分，构造语法分析树，指出语法错误，指导翻译等。

语义分析一般和语法分析同时进行，成为语法制导翻译。作用：获取标识符的属性，进行语义检查，完成子程序的静态绑定、变量的静态绑定等。

中间代码生成器作用：生成简单规范、与机器无关、易于优化与转换的中间代码，便于移植。

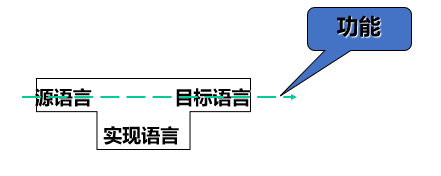
代码优化器作用：对中间代码进行优化处理，使程序运行能够尽量节省存储空间，更有效地利用机器资源，使得程序的运行速度更快，效率更高。

目标代码生成器作用：将中间代码转换成目标机上的机器指令代码或汇编代码。

表格管理作用：管理各种符号表，查、填源程序中出现的符号和编译程序生成的符号，为编译的各个阶段提供信息。

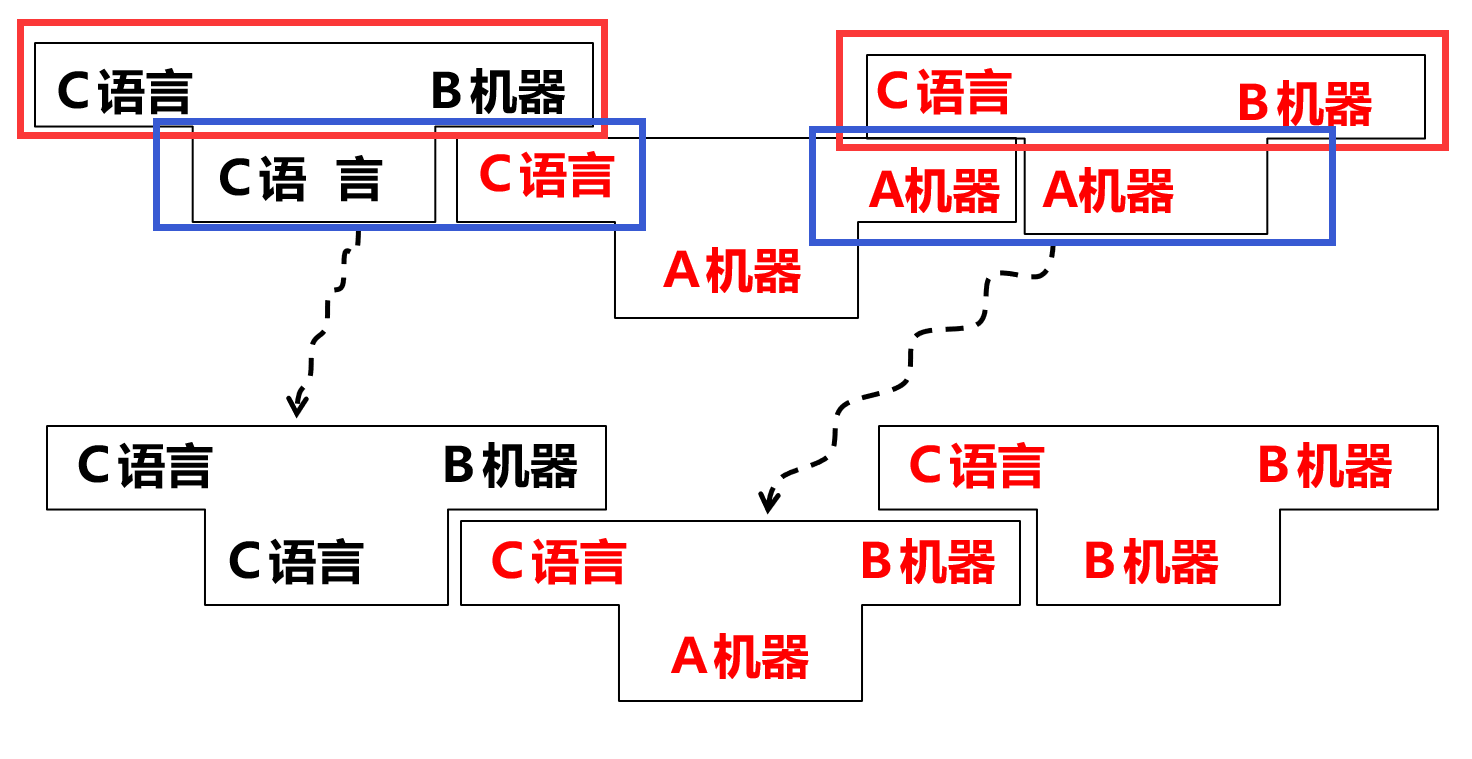
错误处理：进行各种错误的检查、报告、纠正，以及相应的续编译处理。

1. 编译程序的生成
2. T形图：



要深刻理解每个T形图本质上都是一个程序。实现语言为机器语言可以直接在对应机器上运行，但为编写方便一般要求实现语言为高级语言。

1. 生成过程：



实现语言要尽量使用高级语言。经典问题有自展问题，移植（交叉编译）问题，本机编译器利用问题等。

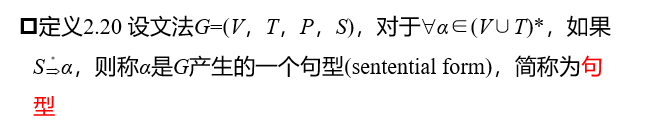
自展问题：要在A机器上实现L语言，可以先用A机器语言实现L语言的子集L1，再用L1实现一个更大的子集L2，再用L2实现L3，……，直到最终实现L语言为止。

移植问题：已经在A机器语言上实现L语言的编译器，也想要在B机器上也实现L语言的编译器。

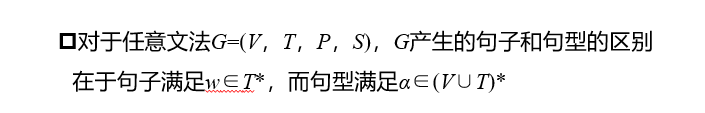
本机编译器利用问题：已经在A机器语言上实验L语言的编译器，想要继续在A语言上实现另一个语言L`的编译器。

## 第二章

1. 关键概念：
2. 句型：



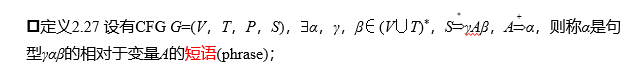
1. 句子：



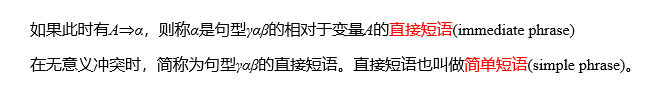
1. 句柄：



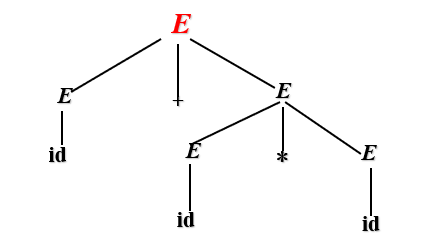
1. 短语：



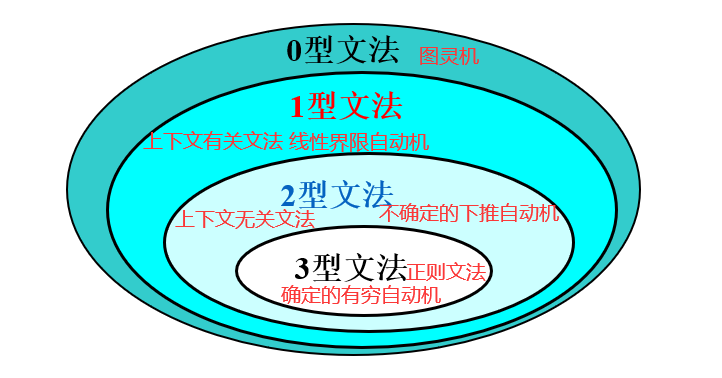
1. 直接短语（简单短语）：



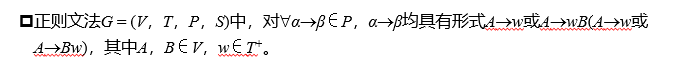
练习：写出短语、直接短语和句柄：



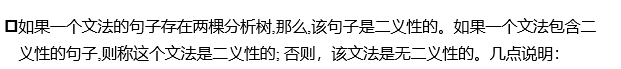
1. 文法层级：



其中正则（三级）文法：（**必须包含一个非终结符**）



1. 文法二义性：

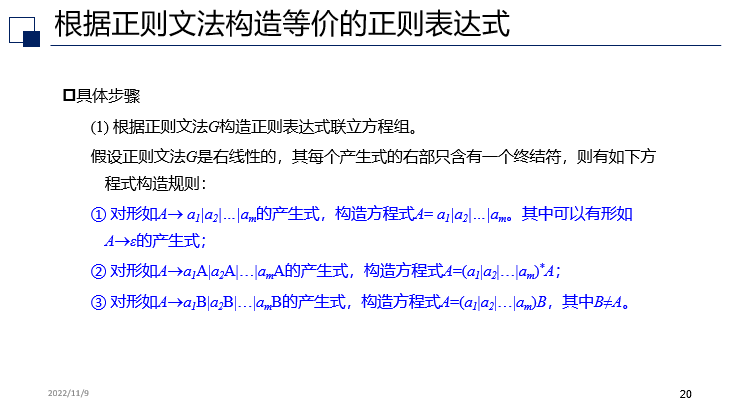


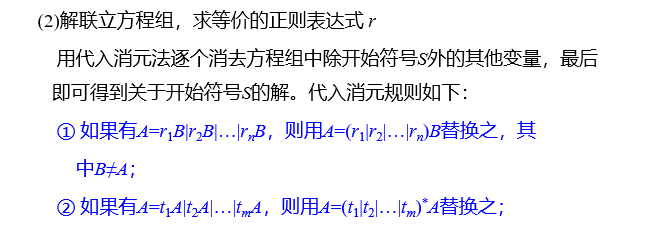
文法的二义性影响句子的分析，有时可以将其消除，但文法的二义性是不可判定的，没有一般的消除文法二义性的方法，而且固有二义性语言的文法的二义性是不可消除的。但是，可以通过给文法附加一些条件来解决它所含二义性所导致的分析冲突问题，例如手动规定符号之间的优先级，增加新的语法变量予以消除歧义等。

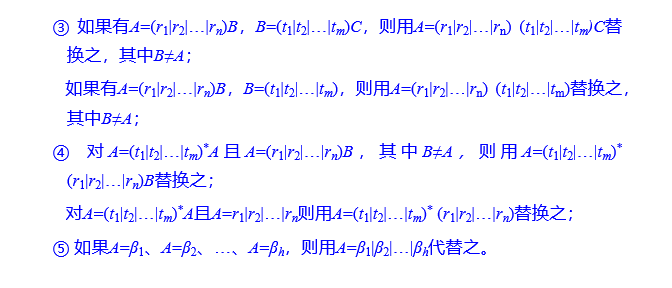
## 第三章

1. 词法分析器的功能：输入源程序，输出单词符号序列（tokens）。
2. 源程序的输入缓冲与预处理
   1. 超前搜索和回退：遇到双字符运算符，如要区分\*和\*\*时需要超前读入一个字符。
   2. 缓冲区：避免每读一个字符都要访问一次磁盘。
   3. 双缓冲区问题：避免在读入字符时，同一个token被分割。过程：若读入eof，结束；若读入缓冲区1末尾，重载缓冲区2，指针+1（移到缓冲区2开始）；若读入缓冲区2末尾，重载缓冲区1，指针移到缓冲区1开始；其余情况指针+1。
   4. 剔除空白字符。
3. 词法分析阶段的错误处理：
   1. 非法字符检查
   2. 关键字拼写错误检查
   3. 不封闭错误检查
   4. 重复说明检查
   5. 错误恢复与续编译：反复删掉剩余输入最前面的字符，直到能正确发现一个单词
4. 单词应该被分成多个种类，并被表示成(种别，属性值)的二元组形式；
5. 正则表达式符号：
   1. |：或
   2. \*：克林闭包
   3. +：正闭包
   4. ？：零个或一个
   5. 直接相连：拼接
6. 正则表达式和正则文法的相互转换：

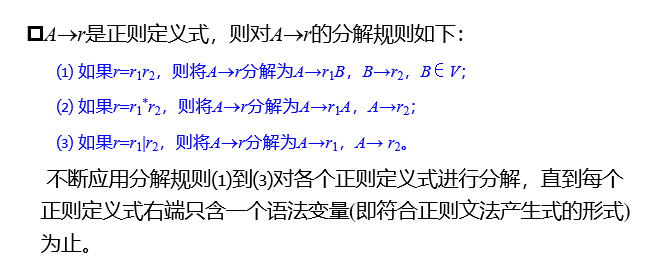
文法->表达式：





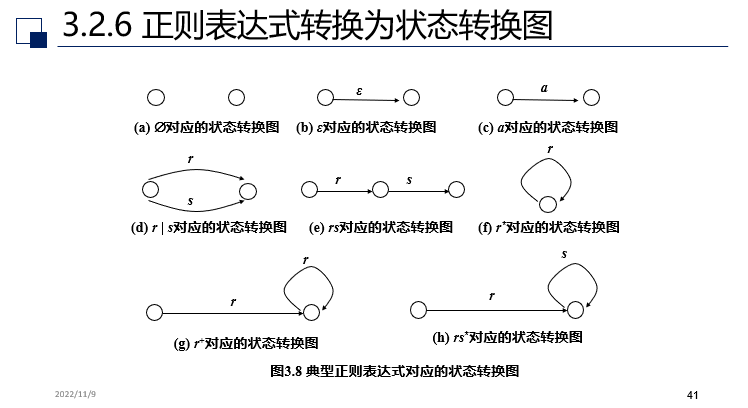


表达式->文法：



只有拼接需要引入新变量B。

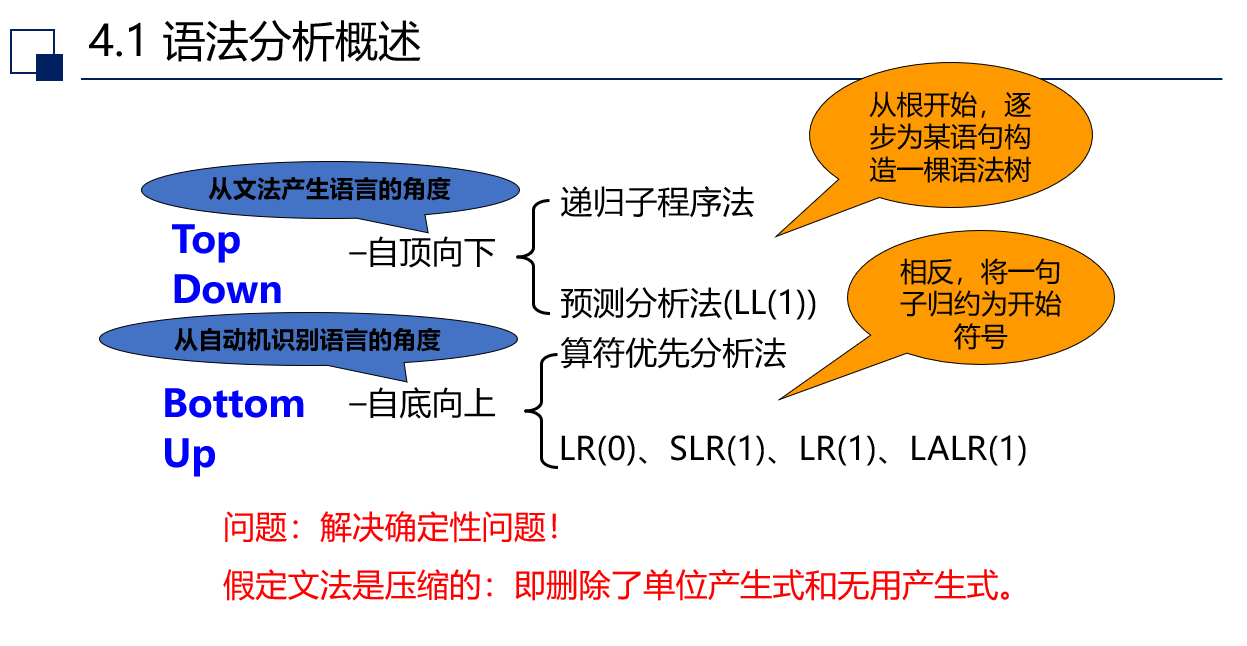
1. 正则表达式和NFA转换：



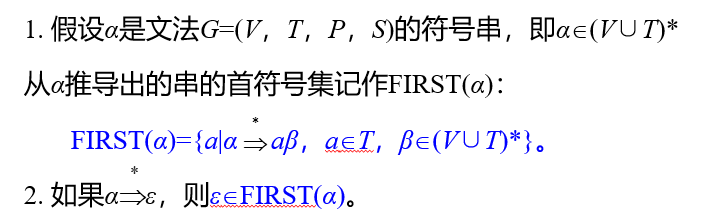
1. 通过DFA状态转换图构造出相应的分析程序

## 第四章

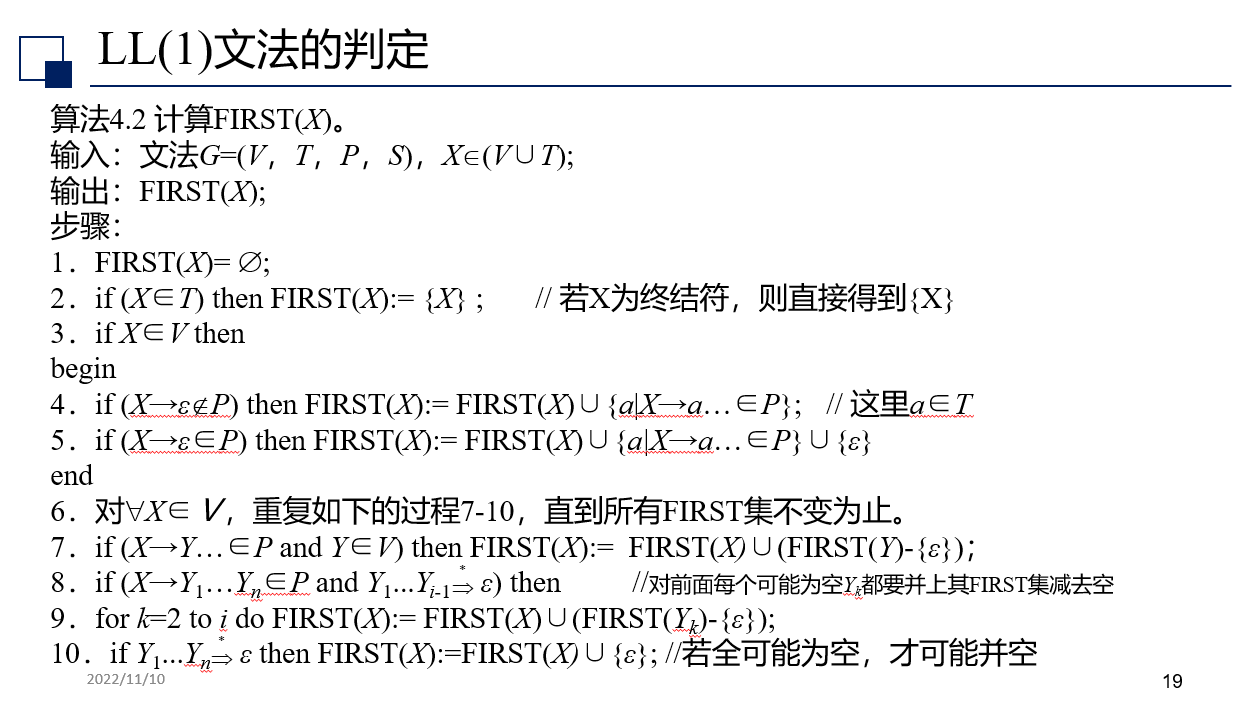
1. 语法分析概述：



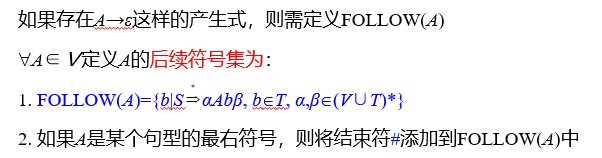
1. 自顶向下分析法遇到的问题：
   1. 二义性问题：存在两个最左推导，语法分析程序无法决定采用哪个推导
   2. 回溯问题：若推导的顺序不对，需要进行大量回溯，效率低下
   3. 左递归引起无穷推导：推导顺序是从左往右的，若存在左递归则会无穷推导，陷入无限递归导致栈溢出
2. 使用自顶向下分析法需要对文法进行改造：
   1. 消除二义性：引入新的语法变量。掌握例题中if-else语句改写和运算表达式改写。
   2. 消除左递归：分为消除直接左递归和消除间接左递归。直接左递归如A->Aa|B改写为A->BA`，A`->aA`|；间接左递归先代入成直接左递归，然后按直接左递归解法消除左递归。
   3. 提取左因子：如果语法变量A有多个候选式存在公共前缀，则自顶向下的语法分析程序将无法根据当前的输入符号选用哪个候选式来替换A，只能试探，从而出现降低效率的回溯问题。对形如A->aB1|aB2|C1|C2的表达式，改写为A->aA`|C1|C2，A`->B1|B2，直到每个候选式都没有公共前缀为止。
3. FIRST集求法：
   1. FIRST集定义：



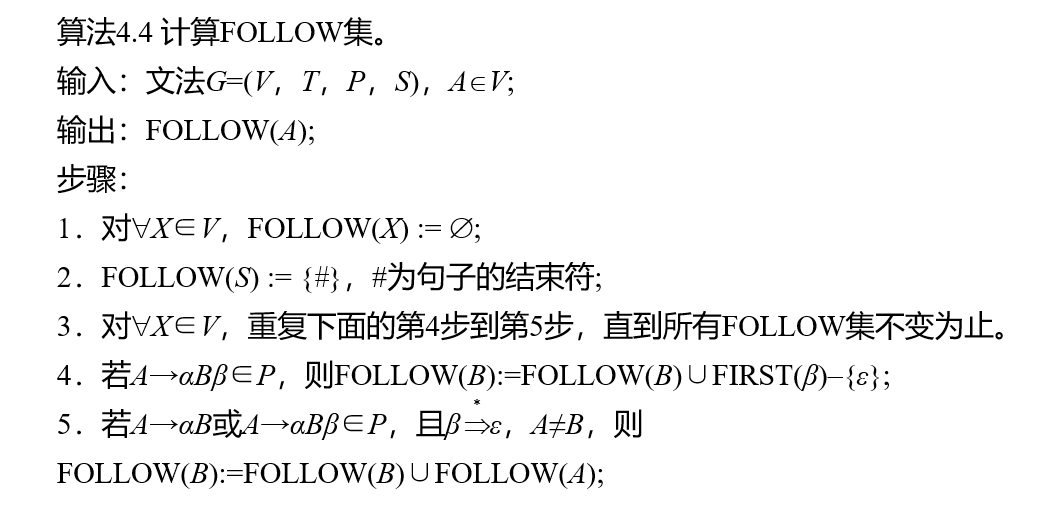
* 1. 求法：



1. FOLLOW集求法：
   1. FOLLOW集定义：



* 1. 求法：

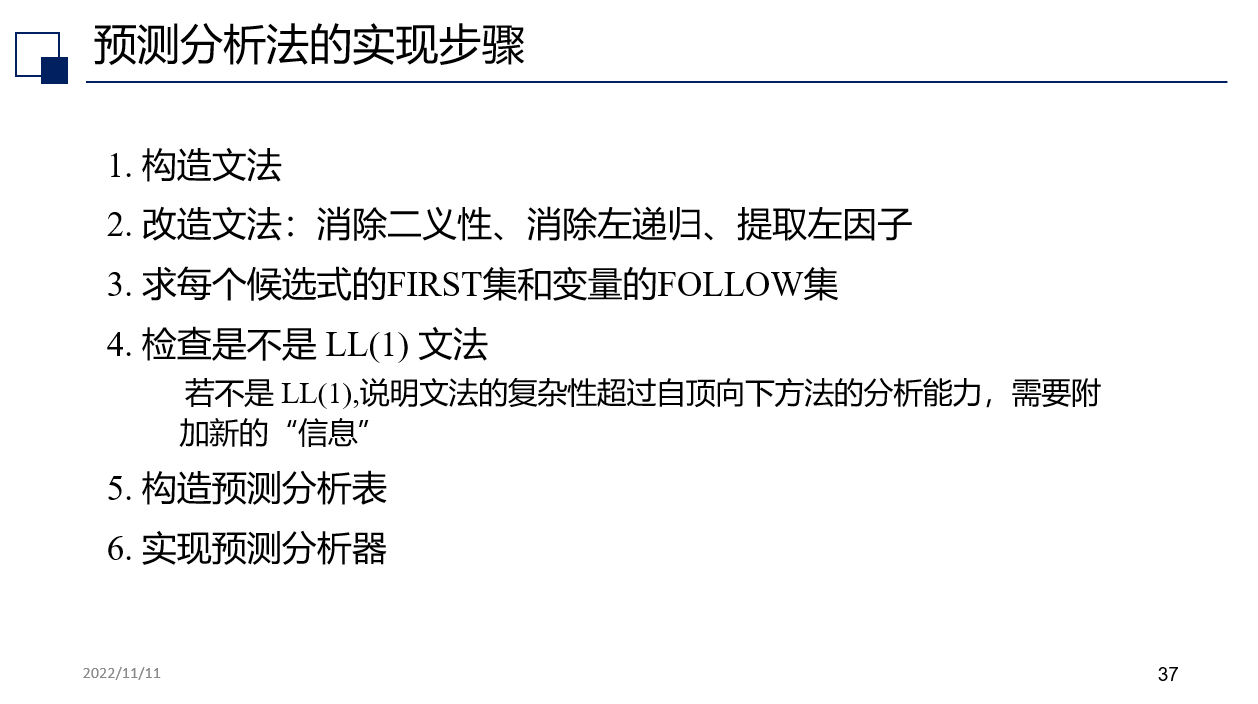


1. LL(1)文法定义：

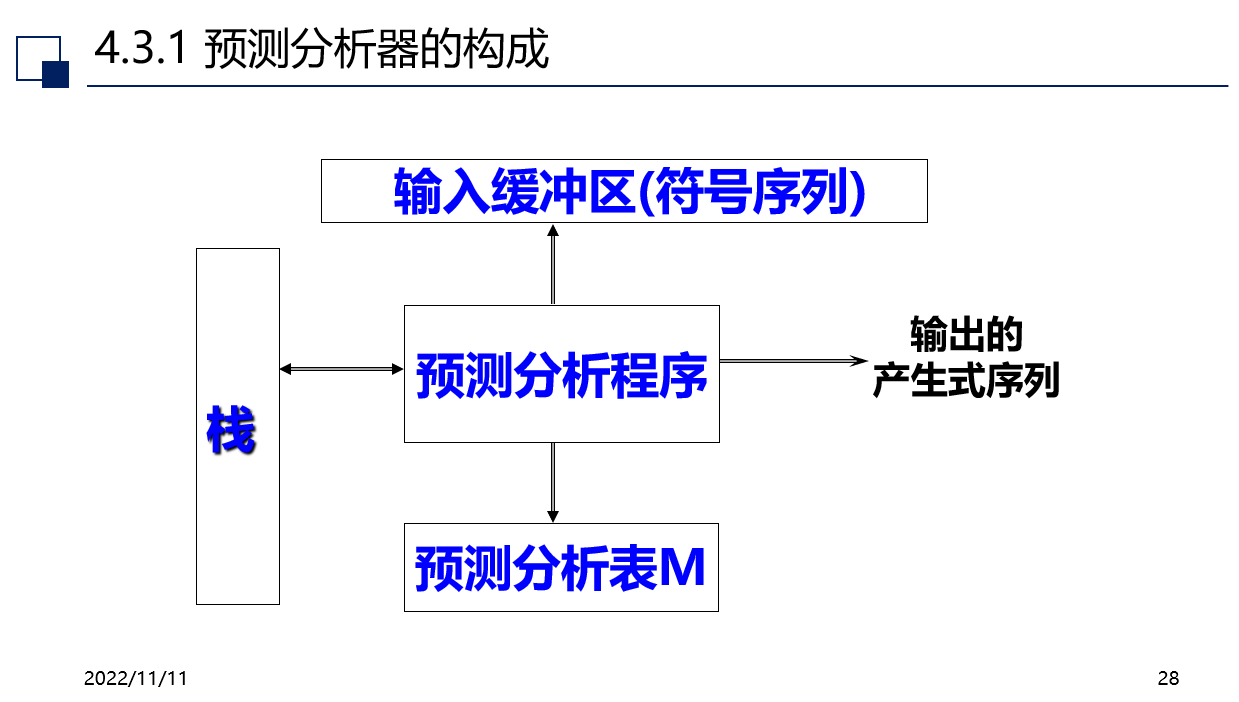


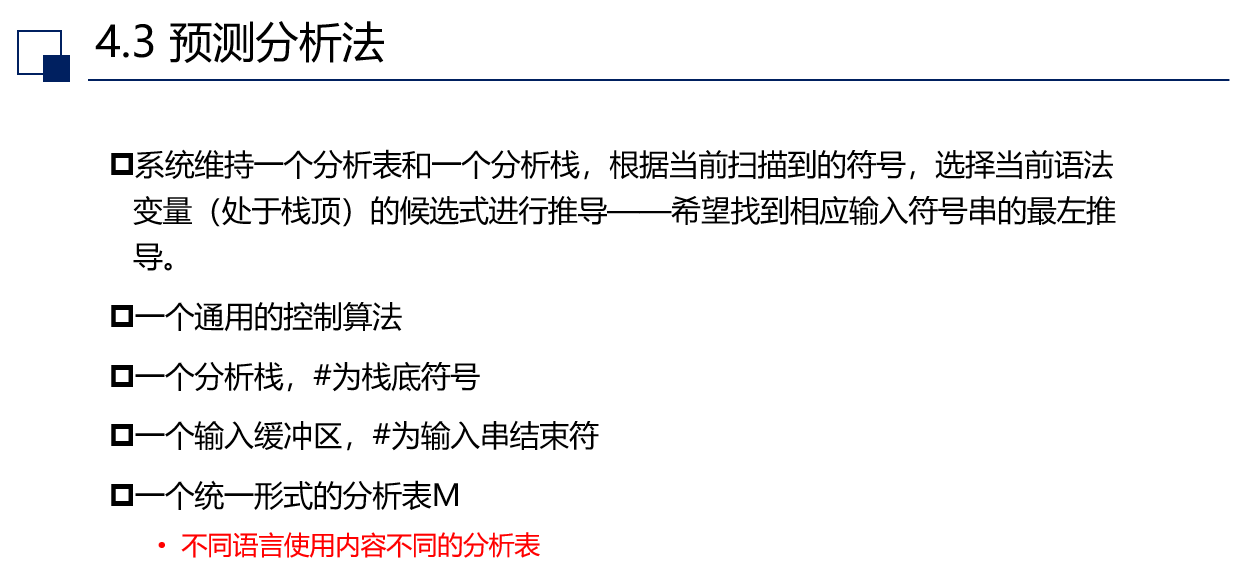
判定时，需要对每个拥有多个产生式右部的语法变量A进行判断：求解FOLLOW(A)和每个右部α的FIRST(α)，比较FIRST(α)之间是否有交集，再比较FOLLOW(A)和每个FIRST(α)是否有交集。

1. 预测分析法：
   1. 构成：
   2. 步骤：

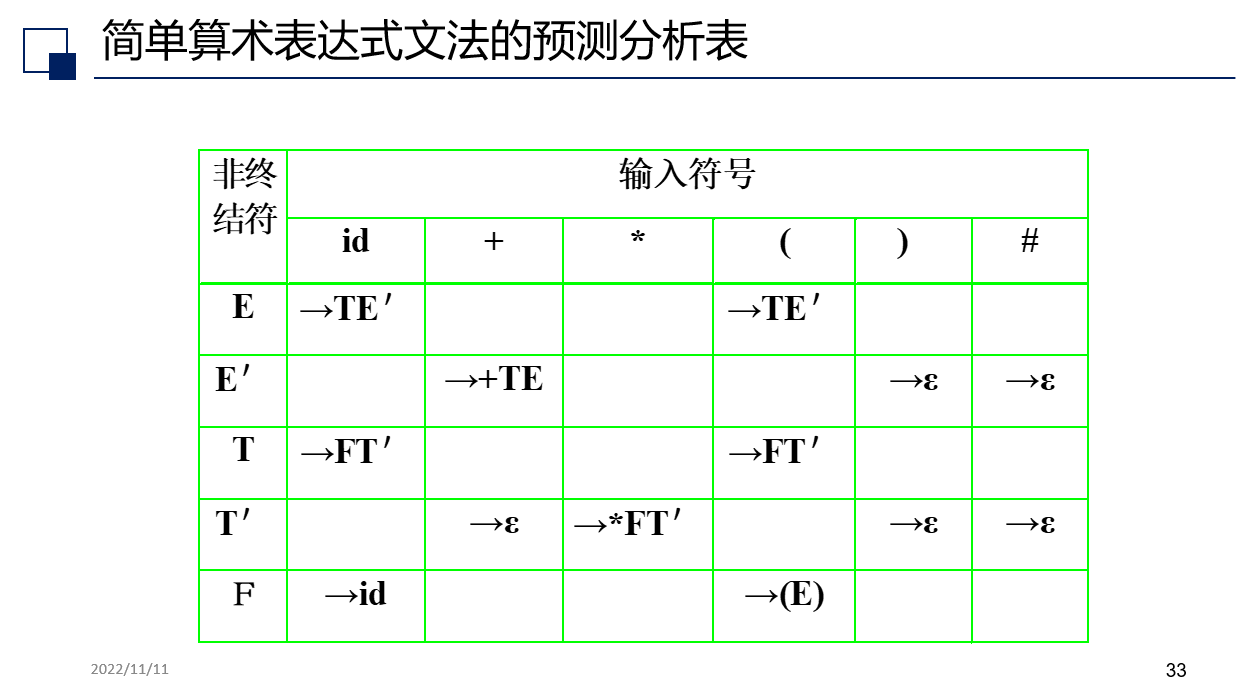


* 1. 过程：



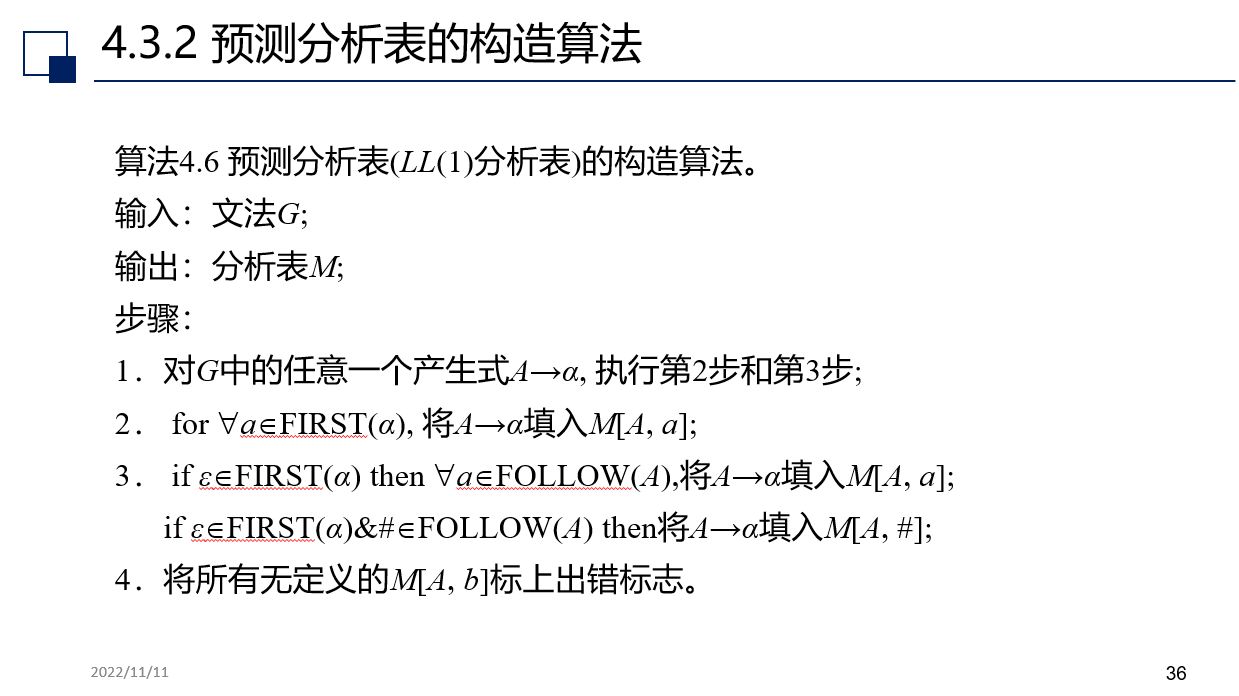


* 1. 过程：

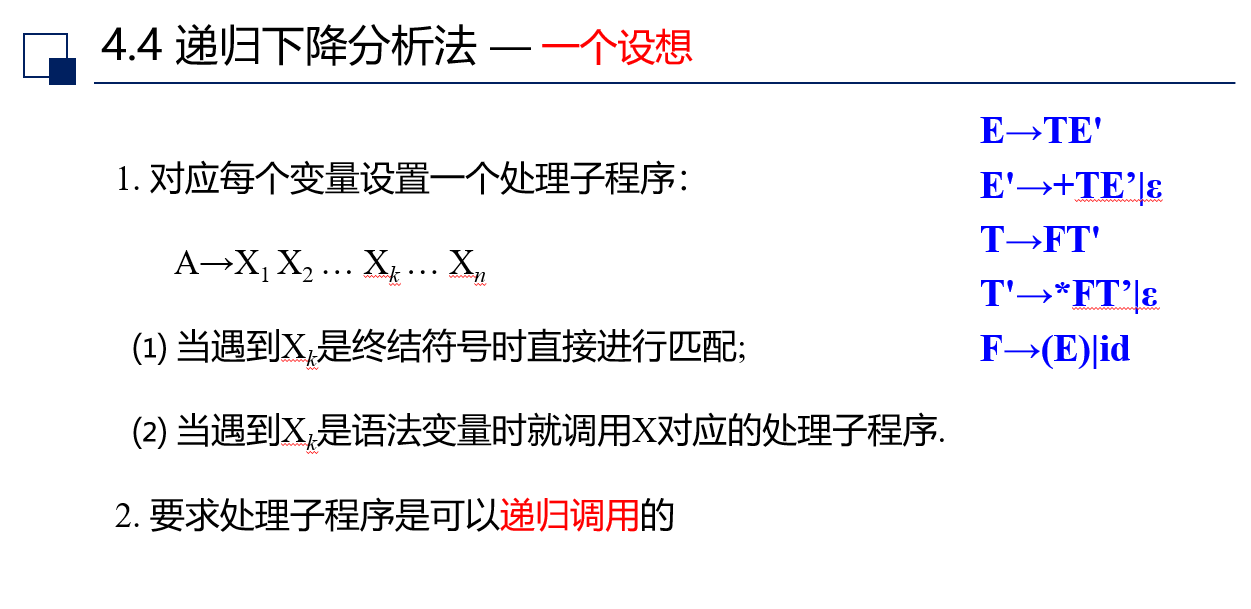


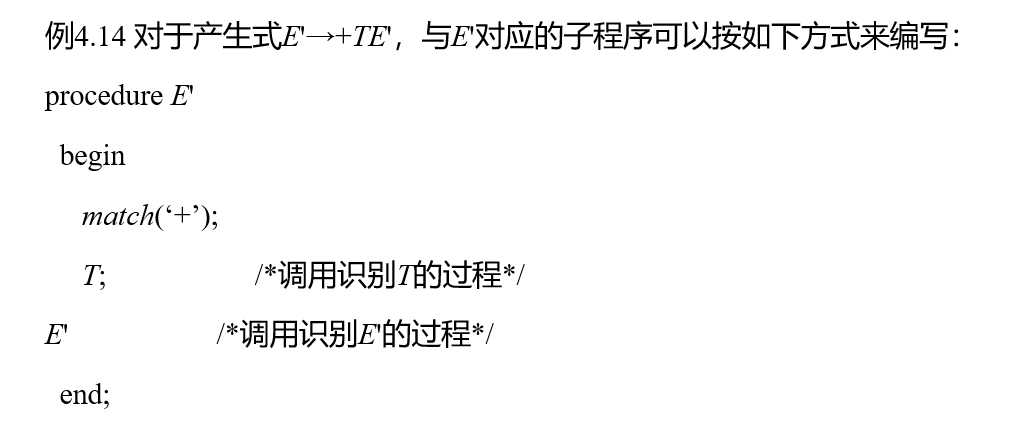
以此分析表为例，当栈顶元素为E，剩余输入元素为id时，弹出E，将TE`压入栈内（T在栈顶）；当栈顶元素为t，剩余输入元素也为t时，弹出栈顶t，剩余输入串指针后移。重复此过程，直到栈内只有初始元素#为止。

* 1. 预测分析表的构造：



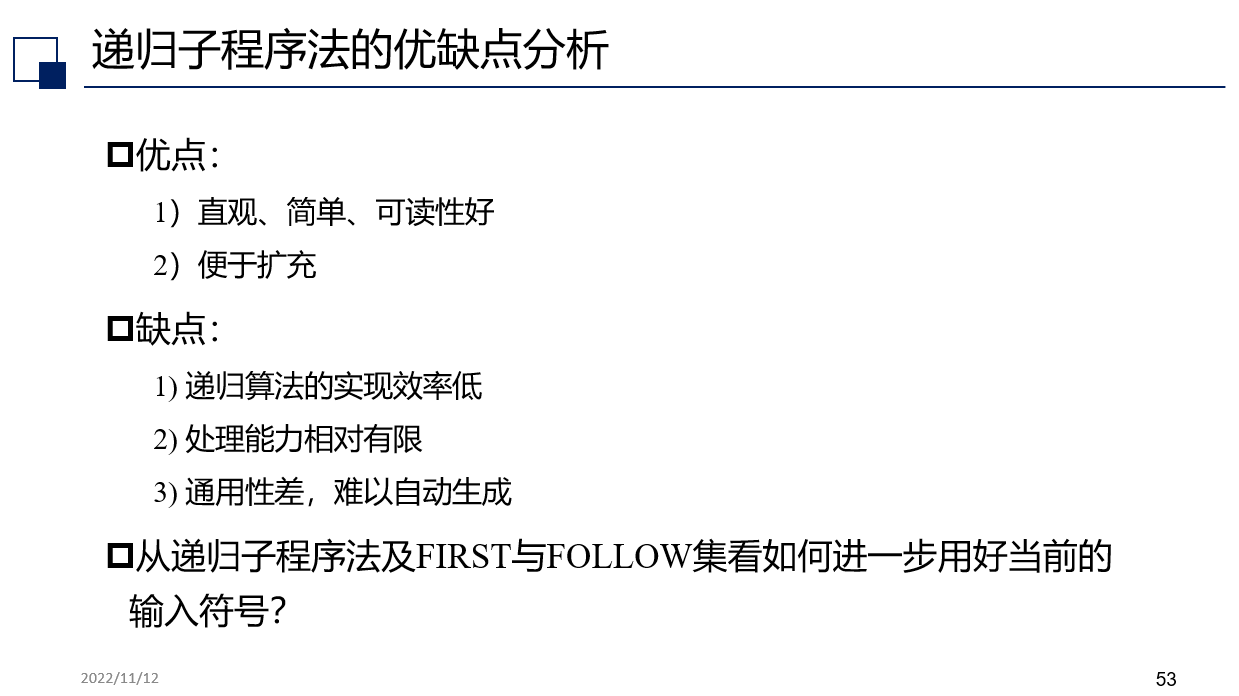
1. 递归下降分析法：





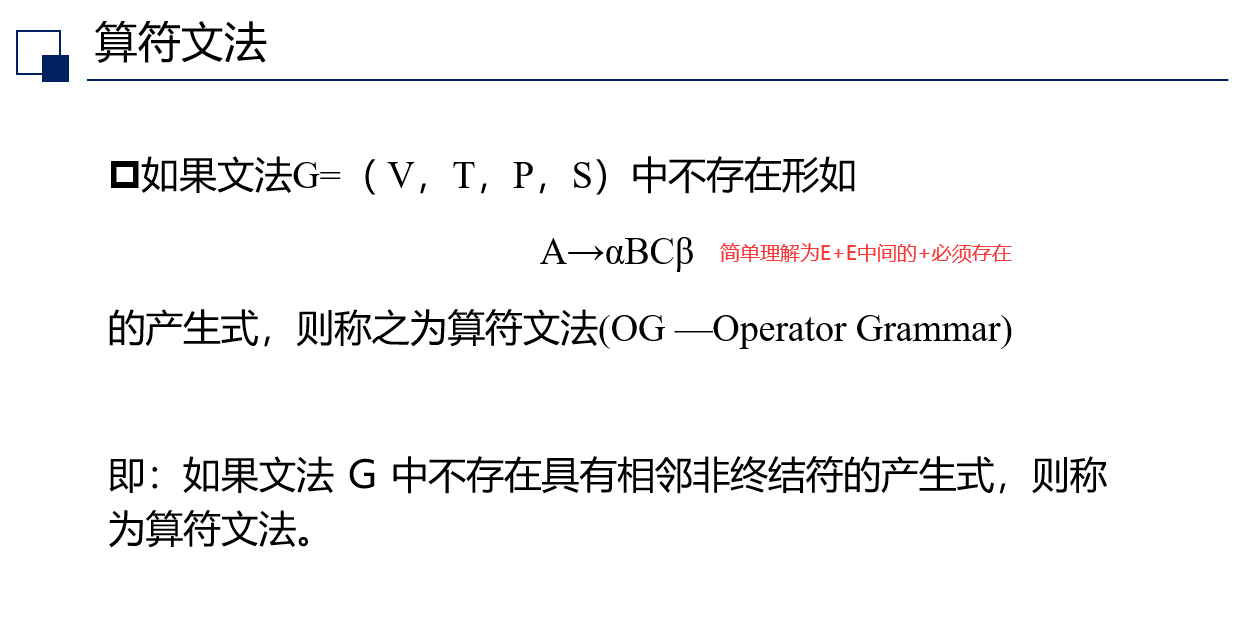
有多个候选式要用提前用if-else语句判断提前读入字符lookahead是否属于该候选式的FIRST集；遇到闭包要用while语句判断lookahead是否属于闭包部分的FIRST集。

match(t)中判断lookahead是否与t相同，若相同则将lookahead向后移，否则报错。

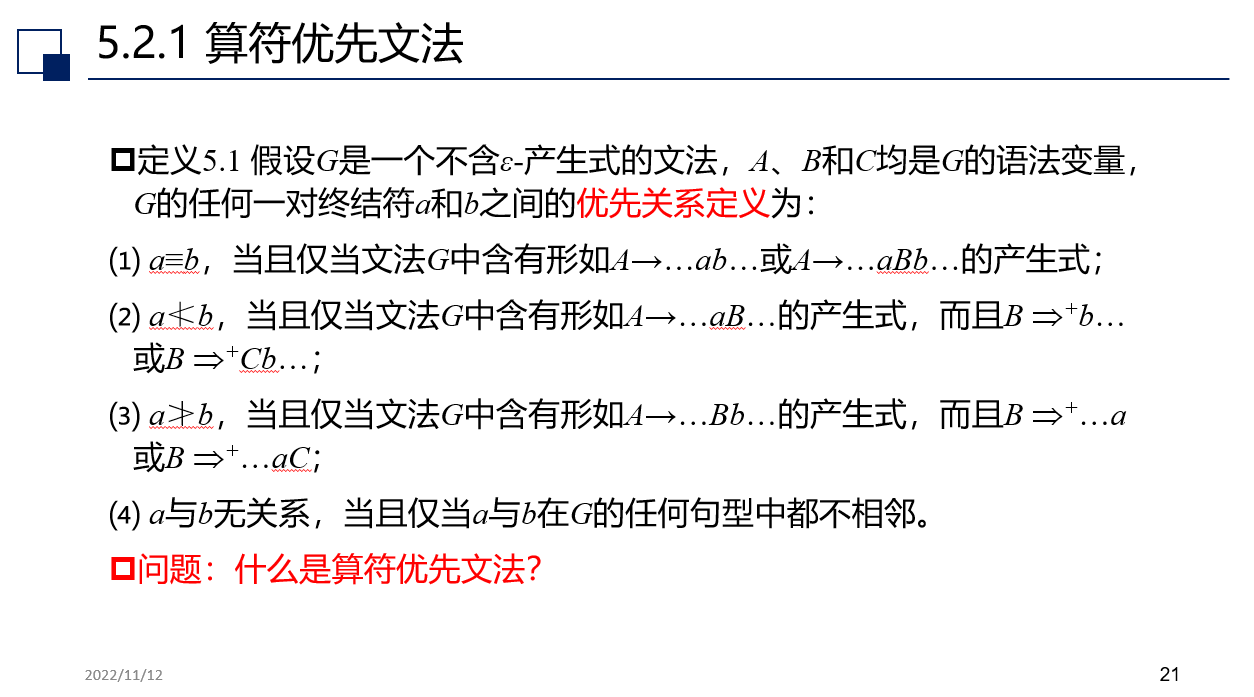


## 第五章

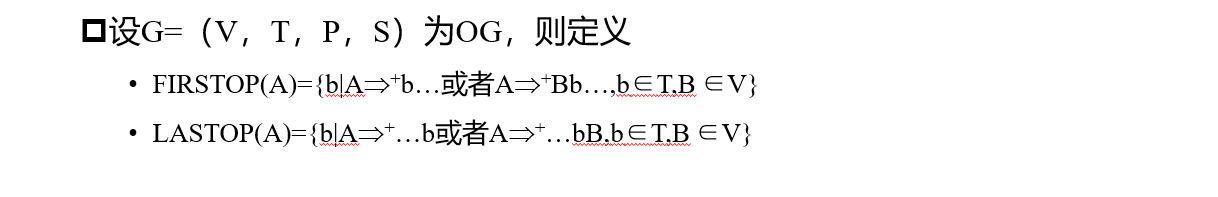
1. 算符优先文法
   1. 算符文法定义：

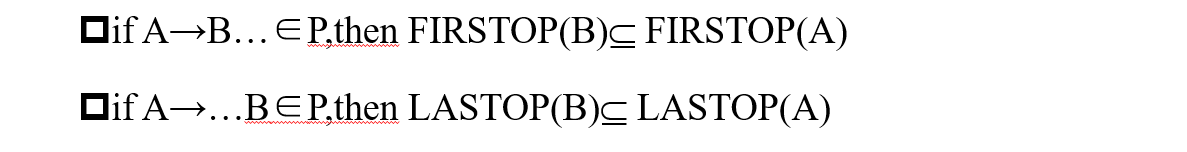


* 1. 算符优先文法优先级定义：

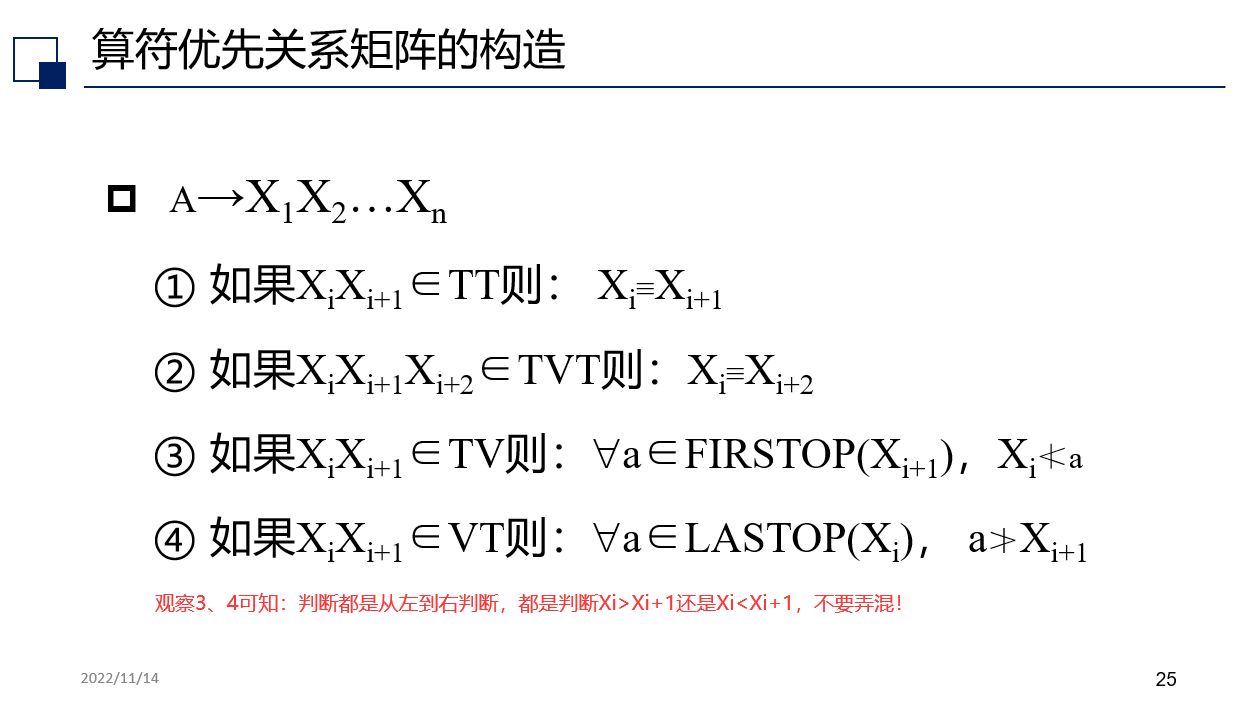


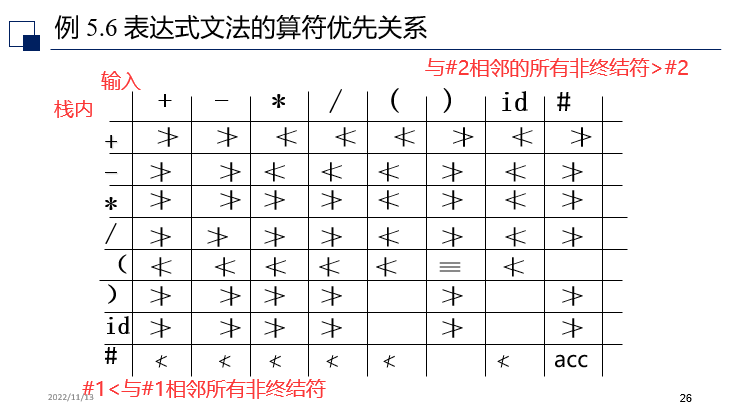
* 1. FIRSTOP、LASTOP集定义：





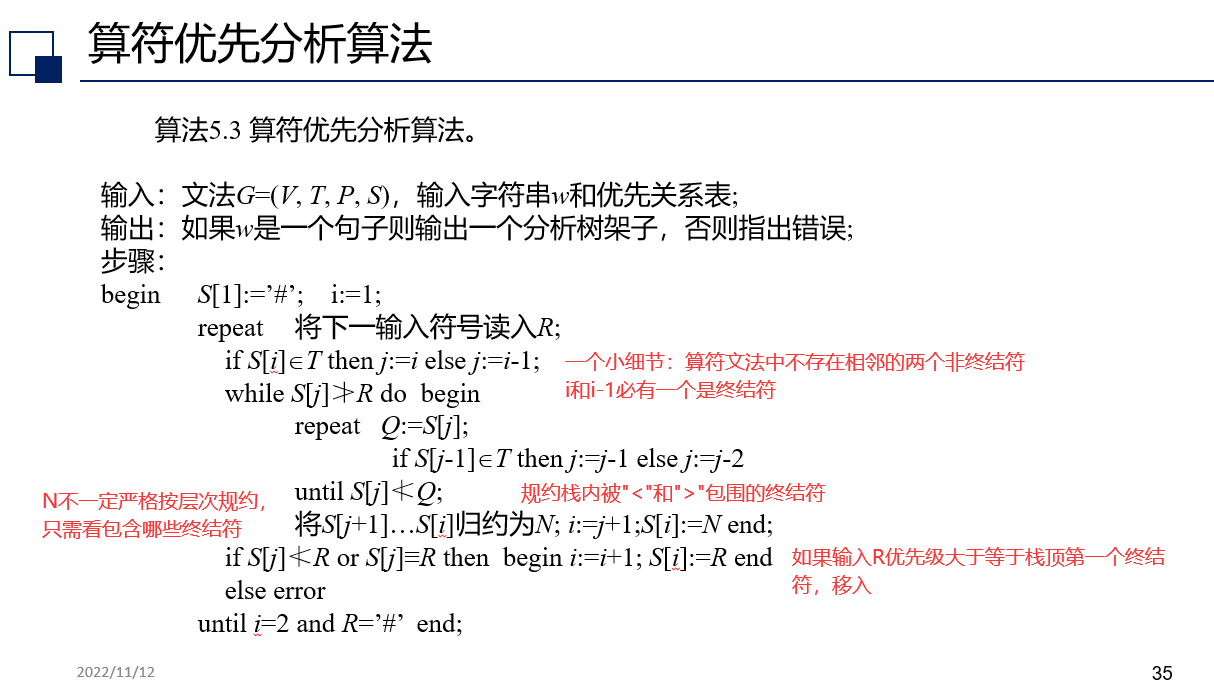
* 1. 构造优先关系矩阵及图示

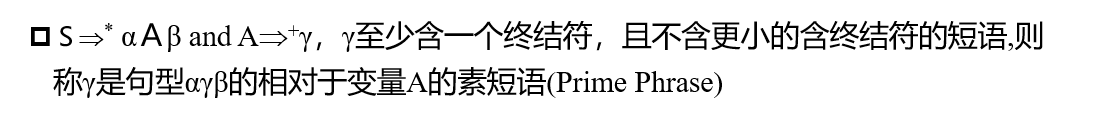




拓展文法S`->#S#，于是有#<FIRSTOP(S)，又有LASTOP(S) >#

* 1. 算符优先分析算法过程：





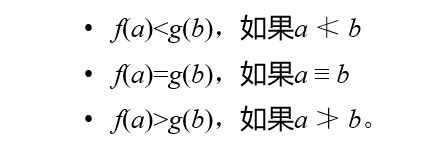
算符优先分析算法规约的是最左素短语，某句型最左素短语可以通过比较该句型非终结符的优先关系，最先得到”<=…=>”形式的，被”<”和”>”包围着的句型子集就是最左素短语。

注意：算符优先文法分析跳过了所有**单语法变量产生式**所对应的规约步骤，有可能将本来不属于该文法语言的输入串认为符合该文法。



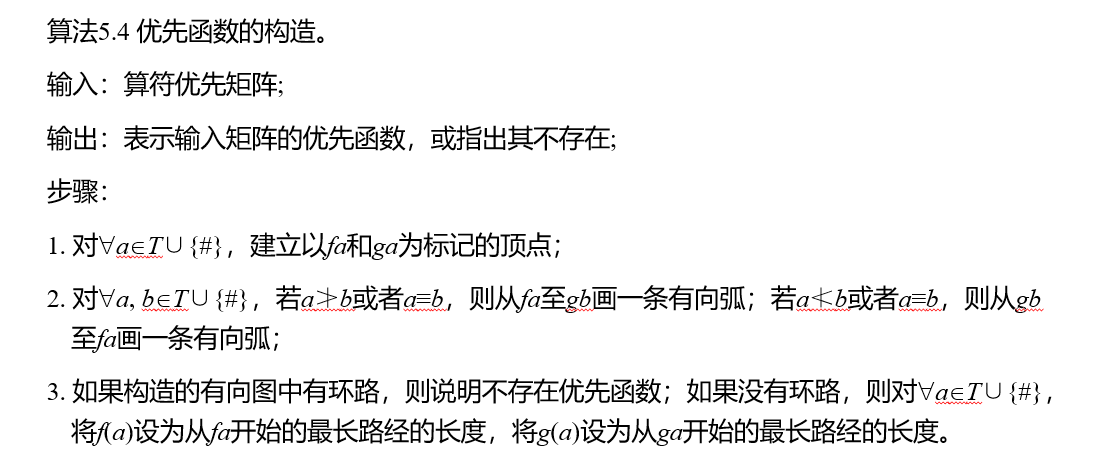
优先函数：

存储n^2的算符优先矩阵内存空间有所浪费，可以对每个非终结符建立对自然数的映射（n^2 -> 2n），使得：



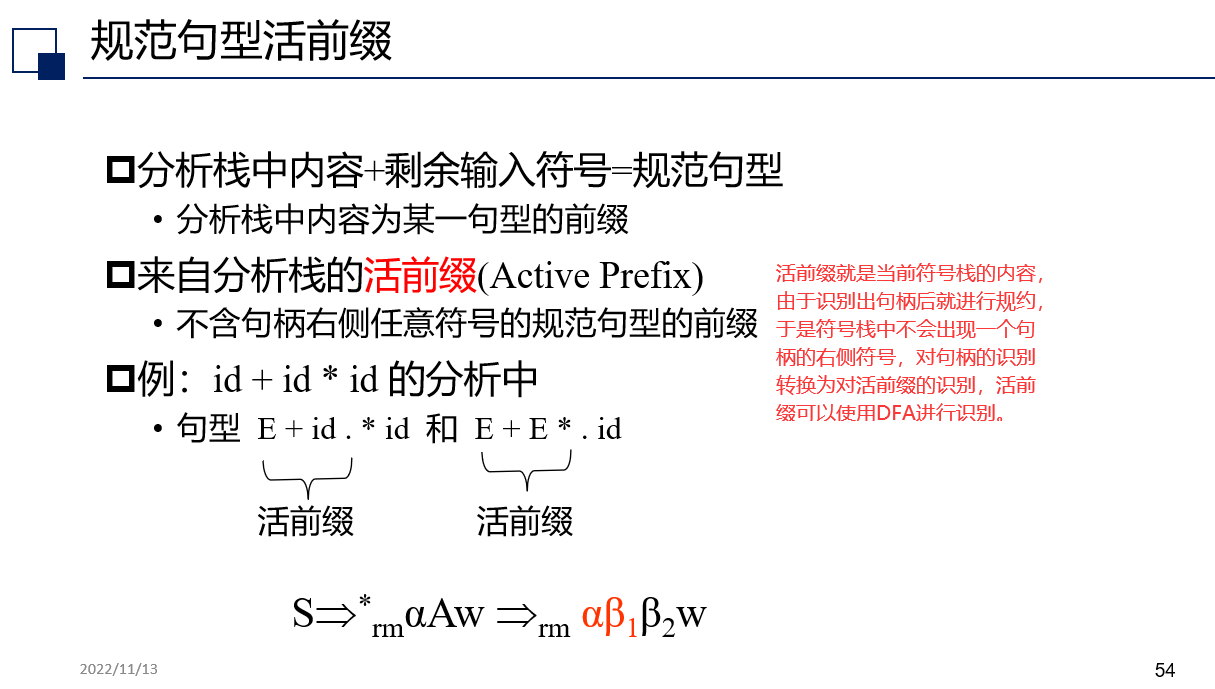
缺点：算符优先矩阵内的空白被填充，原本不能比较的a和b借由f(a)和g(b)得以比较。

优先函数的构造方法：



构造有向图，其中f(a)和g(a)的值可以从f(#)和g(#)开始利用动规的思想求解。

1. 状态法
   1. LR(k)：L指从左向右扫描输入字符串，R指最右推导对应的最左规约，k指超前看的符号个数。
   2. 活前缀识别：



* 1. 首先先对文法进行拓展，再对拓展文法对项目集求闭包，得到项目集规范族。可以用项目集规范族构建状态转移图，再根据状态转移图填写LR分析表，不同文法分析方法有不同的填写方法，主要有LR(0)、SLR(1)、LR(1)、LALR(1)。其中，识别能力LR(1)>LALR(1)>SLR(1)>LL(1)?LR(0)。
  2. LR分析表图示：



以上图为例简单说明一下LR分析过程：

若当前状态栈栈顶是0，当前剩余输入符号为a，则将a移入到符号栈中，并将状态3压入到状态栈栈顶。

若当前状态栈栈顶是4，当前剩余输入符号为#，则按照第三个产生式规约，符号栈和状态栈pop一个位置（|b|），然后再将产生式左部B压入到符号栈中，再根据此时状态栈栈顶，若状态栈栈顶为0，则按goto表中压入状态2。

* 1. LR(0)、SLR(1)、LR(1)、LALR(1)之间的区别：

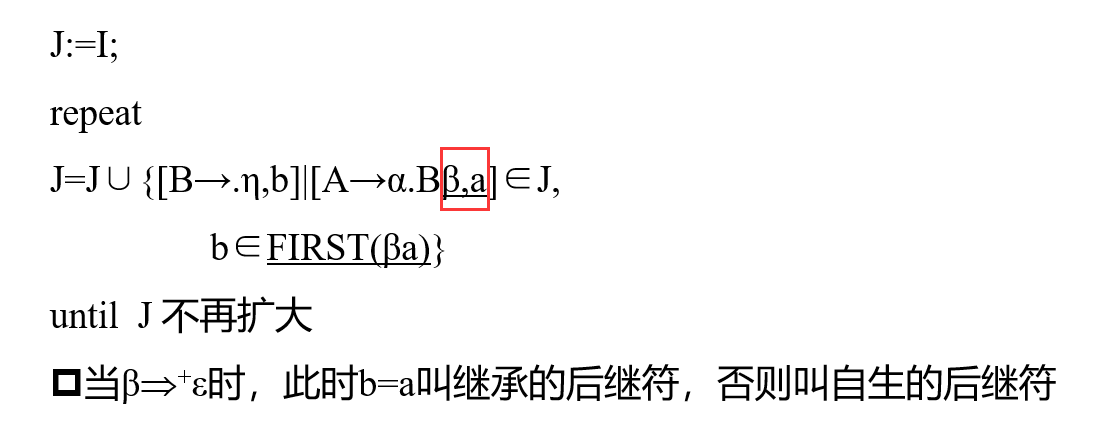
LR(0)文法在规约时，并不考虑当前输入的非终结符，全部进行规约，容易产生规约-规约冲突和移进-规约冲突。

SLR(1)文法在LR(0)的基础上，在规约时考虑当前输入的非终结符，只有当前输入的终结符是要规约的产生式左部语法变量的FOLLOW集成员时，才进行规约。

LR(1)文法在SLR(1)的基础上，在规约时考虑当前输入的非终结符，只有当前输入的终结符是当前位置的搜索符（后继符）时，才进行规约。搜索符集是产生式左部语法变量的FOLLOW集的子集。LR(1)文法产生的状态数比其他文法都要更多，也更为复杂。

LALR(1)文法在LR(1)的基础上，对同心集进行合并。同心集是具有相同的心（即每个产生式的状态完全一致），只是搜索符不同的项目集。合并同心集可以简化LR(1)的状态个数，节省空间，提升效率。但有可能产生新的问题：1.虽然不产生冲突，但是会推迟发现语法问题；2.会产生新的**规约-规约**冲突，而不是移进-规约冲突。

* 1. 对LR(1)文法，其每个项目集的搜索符依据

 中的b来进行求解。

* 1. 对二义性文法的应用：

虽然任何二义性文法都不是LR文法，但通过手动设定优先级，规定符号是左结合（一般是规约）还是右结合（一般是移进），修改LR分析表，消除二义性造成的冲突，依旧能利用LR分析法对二义性文法进行识别。

使用二义性文法的好处有：减少语法分析器的状态数，减少单位产生式规约。典型案例有简单算术表达式文法中的\*和+，if-else语句等。

## 第六章

1. 语法制导翻译的基本思想：

在进行语法分析的同时，完成相应的语义处理。

1. 语法制导定义：

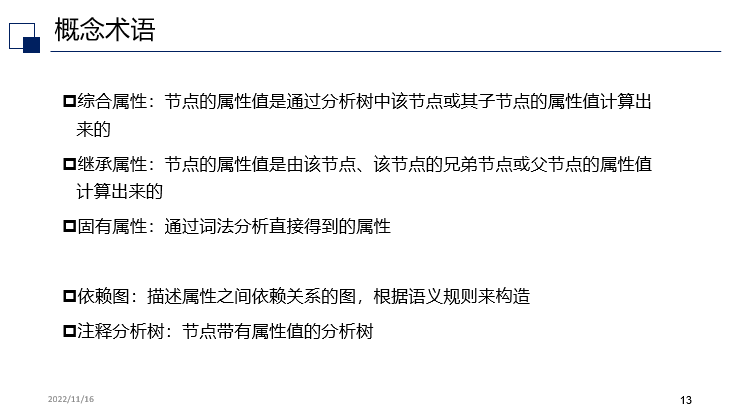
语法制导定义通过将属性与文法符号关联、将**语义规则与产生式关联**来描述语言结构的翻译方案。

1. 语法制导翻译模式：

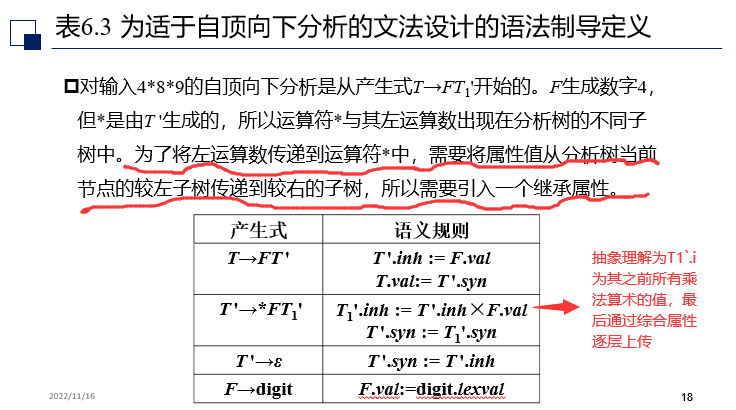
翻译模式通过将属性与文法符号关联，并将**语义规则插入到产生式的右部**来描述语言结构的翻译方案。

语法制导定义侧重于为每个产生式制定相应的语义规则，翻译模式侧重于将这些语义规则插入到产生式右部合适的位置上。

1. 重要的概念术语：

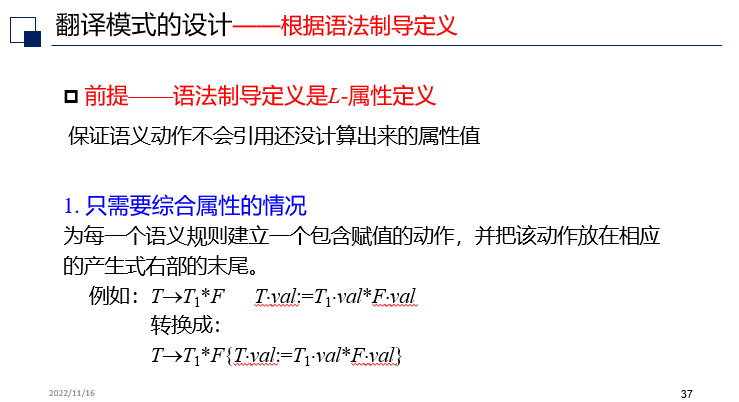


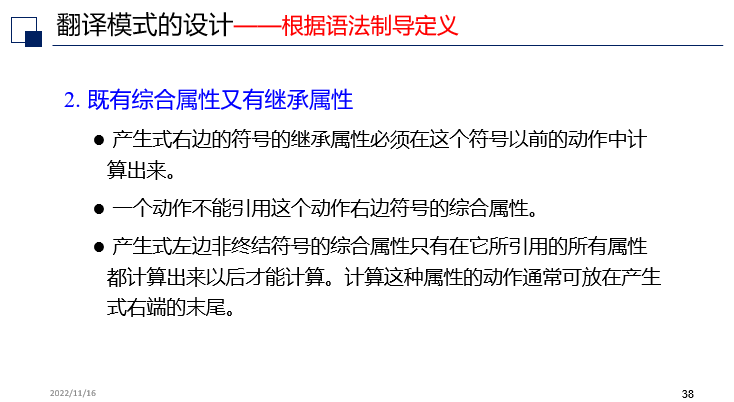
1. S-属性定义：只含有综合属性的语法制导定义称为S-属性定义。S-属性定义天然适合自底向上的语法分析（LR分析），但也不是不能用自顶向下分析法进行语义分析。
2. L-属性定义：当且仅当它的每个属性或者是综合属性，或者是依赖于左兄弟节点、父节点的继承属性，或者是依赖于其本身的综合属性或继承属性（前提是该属性不能在依赖图中形成回路）。



由于继承属性不方便用自底向上的方法进行分析（如产生式左部的语法变量需要将产生式右部规约才能产生，无法先得到其继承属性的值），一般来说，L-属性定义可以用自顶向下分析法进行语义分析，但也不是不能利用自底向上的方法进行分析。**特别地，对任意LL(1)文法，一定可以利用LR分析法进行自底向上的语义分析，这是因为LL(1)文法在经过一定处理后一定会变成LR(1)文法。**

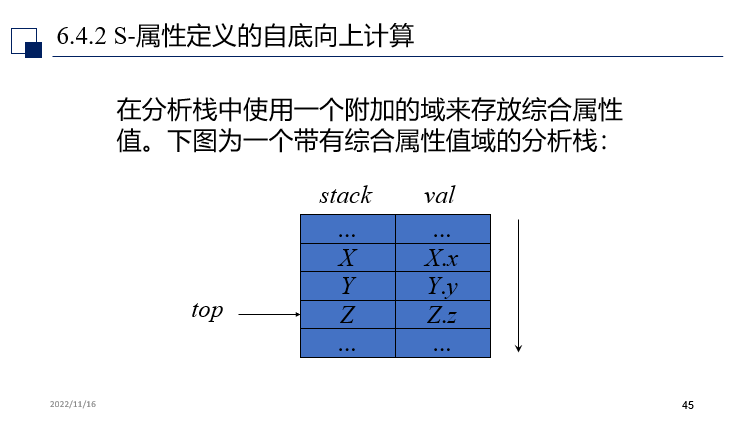
1. 翻译模式的设计原则：





1. S-属性定义的自底向上翻译

在LR分析的基础上，加入一个分析栈，分析栈的形式可以有多种。例如教科书中的分析栈结构：



若stack中的符号没有综合属性则用一个占位符替代。保证栈中符号和val值一一对应。在归约到产生式时，自然地执行对应的语义规则，由于分析栈中符号和val一一对应，可以容易地得到旧变量综合属性值，因此也能够轻易地计算新压入变量地综合属性值。

1. L-属性定义的自顶向下翻译
   1. 由于S-属性定义是L-属性定义的特例，于是本方法也适用于S-属性定义的自顶向下翻译。
   2. 翻译模式下消除左递归的方法：
      1. 简单情形下，如果该左递归的产生式中的语义规则没有计算其他属性，例如只是简单地打印语句，则可以完全将该语义规则当作终结符操作，不需要修改语义规则、增加其他属性等，按照原有方法消除左递归即可，当该语义规则移动到栈顶时执行即可。
      2. 在S-属性下，考虑这样一种左递归的产生式：

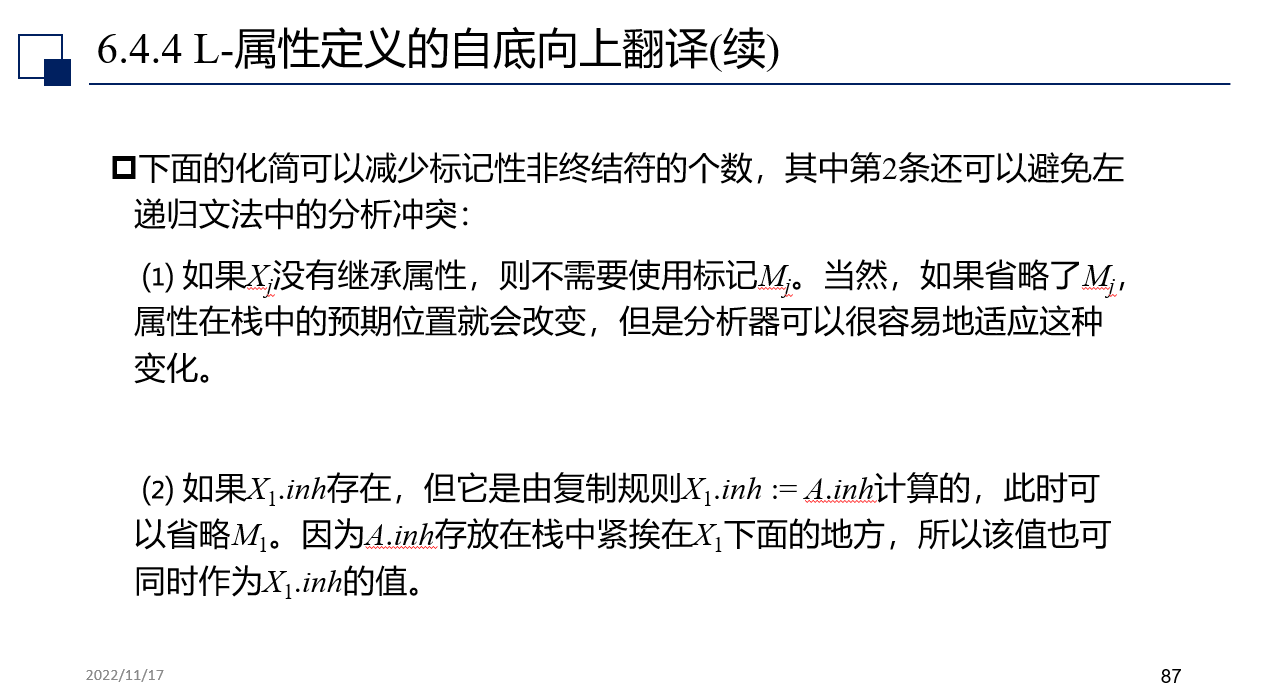
消除左递归后变为：

为其构造翻译方案如下：

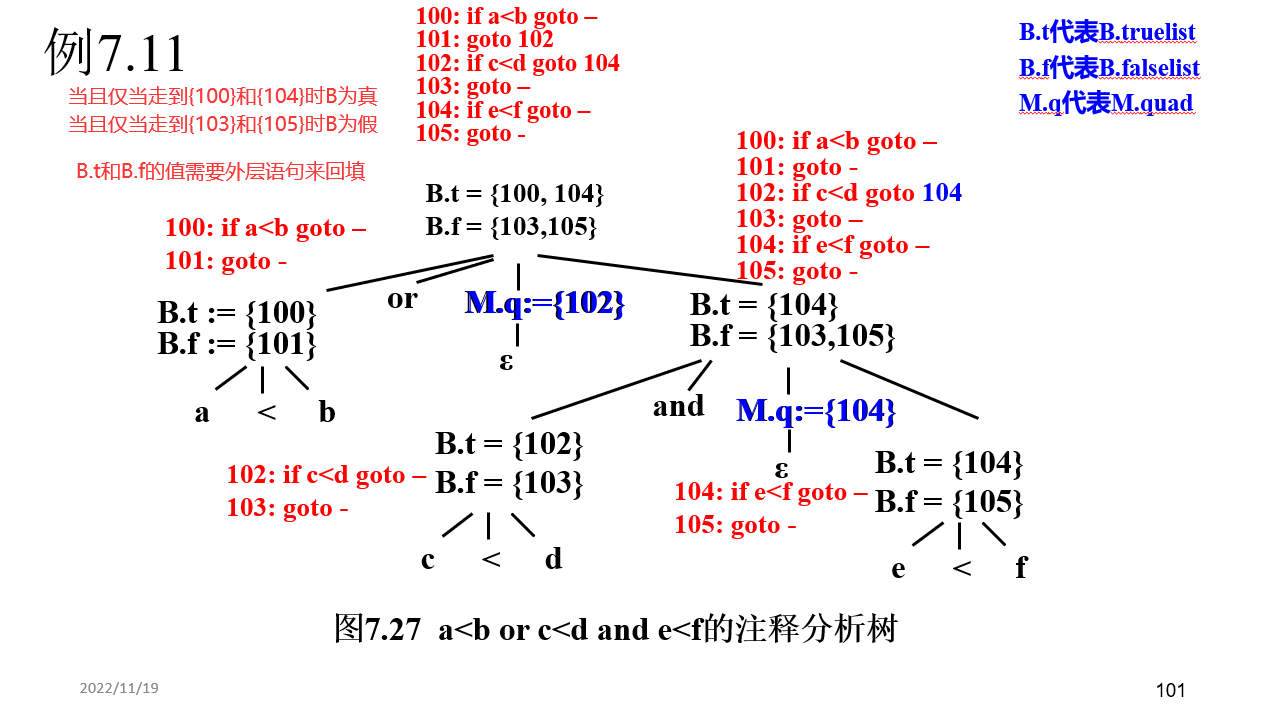
这是一个简单地从左到右计算的L-属性定义。

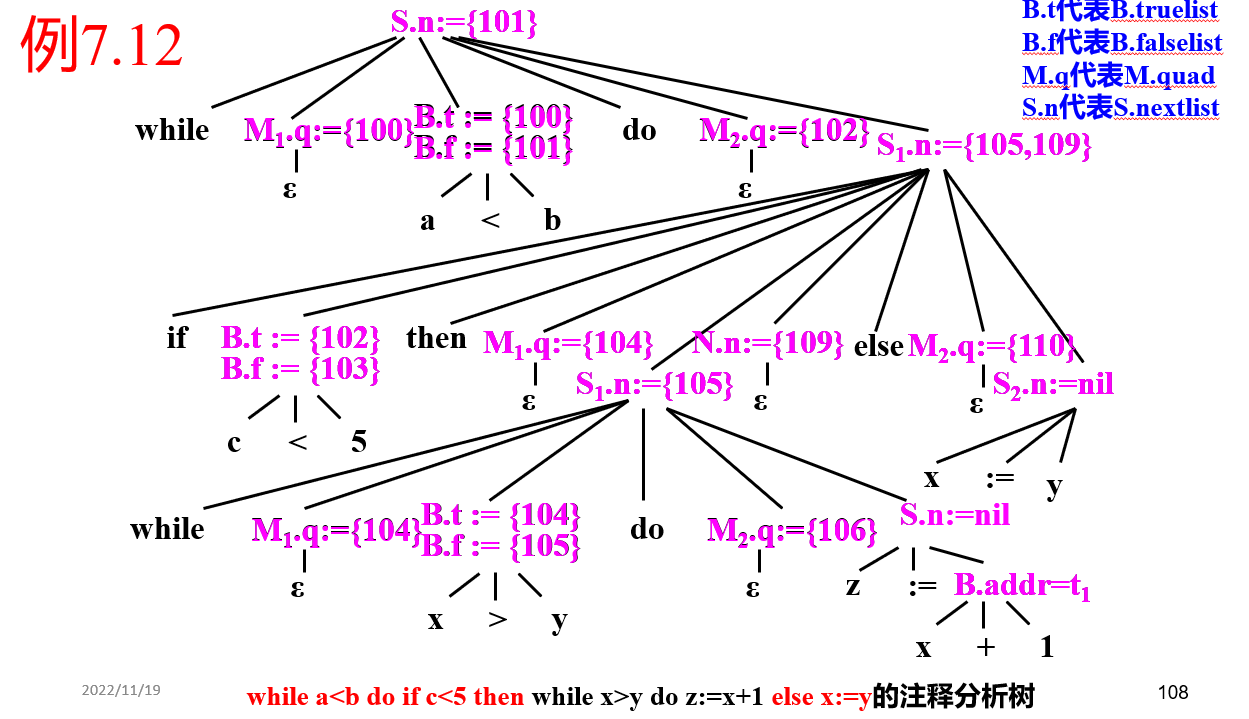
* 1. L-属性自顶向下翻译的两种方法：
     1. 递归下降分析法：
        1. 对每个非终结符A构造一个函数A，将非终结符A的各个**继承属性**作为函数A的**形式参数**，将非终结符A的**综合属性**作为函数A的**返回值**。
        2. 按第四章所讲的递归下降分析法构造函数（主体流程不变），设置一些中间变量保存需要的值，然后根据语义动作做相关操作。最后返回该变量的syn属性。
     2. 调用显式栈LL(1)翻译法：
        1. 该方法语义信息和文法符号不同步，语义栈和语法栈操作不同步。
        2. 用语义动作符号e1、e2、e3等来表示每个语义动作，当这些符号位于栈顶时，运行这些语义动作。
        3. 难点是如何这些语义动作转化为对栈元素的操作，何时pop，pop出来的元素意义是什么？何时push，push元素的意义是什么？这实际上和文法有关，类似于要思考该文法变量的前一个操作是什么，可以从哪里来。

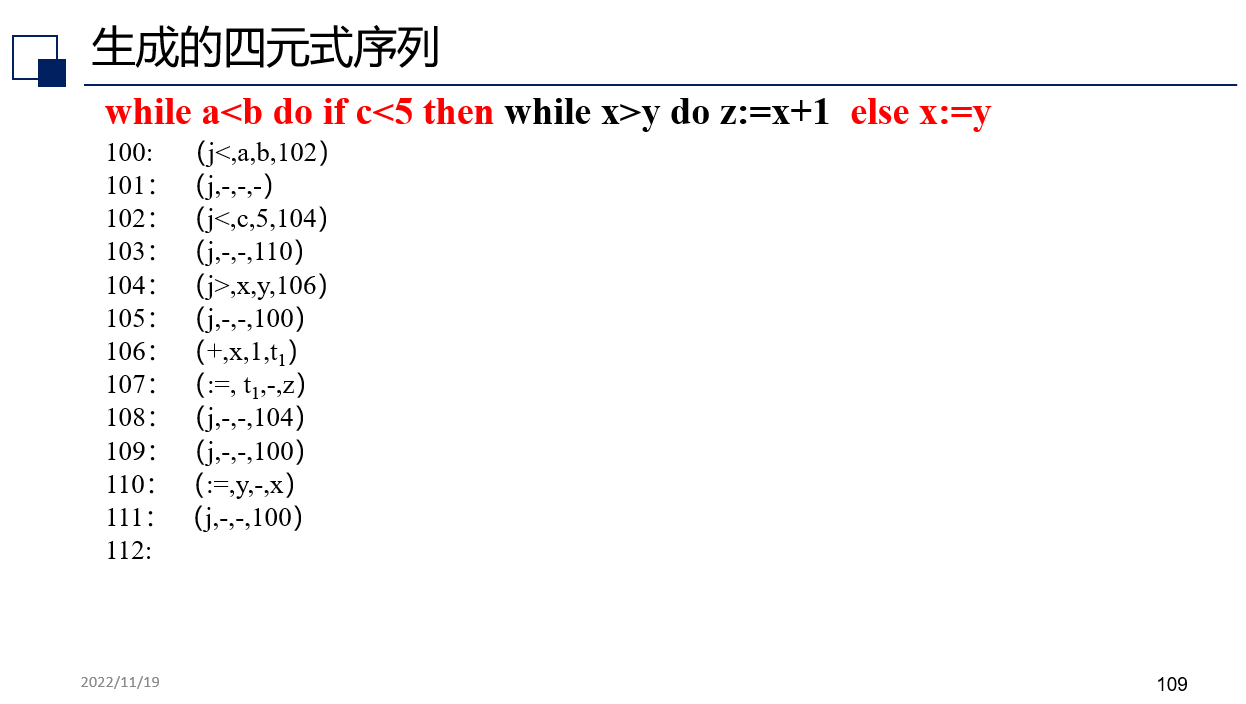
1. L-属性定义的自底向上翻译
   1. 该方法可以用于任意LL(1)文法语义分析。
   2. 给出一个标准的L-属性定义，对嵌入在产生式右部中间的计算继承属性的动作，使用一个标记性非终结符M替代，每个标记性非终结符M都有一个形如的产生式。
   3. 对每个被取代的中间动作，将其修改为，并放在的末尾。首先明确，中间动作作用是用来计算其右某个非终结符的继承属性，它可能需要一些其左部的继承属性做参数来计算。
   4. 中，为M设置若干继承属性来复制这些参数值，以正常方式计算属性，得到的值赋给M.syn。
   5. 计算属性看似是非法的，因为从产生式并不能得到前面符号的继承属性值。但在实际分析过程中，前面符号的继承属性值已经存放在分析栈内。现在所要求的是他们具体在栈内的哪个位置。
   6. 先说明结论：**每个非终结符X的继承属性都放在堆栈中紧靠X将被规约出来的位置之下（即X的位置是top，那么其继承属性值为top-1）**，这和我们利用标记性非终结符M有关。
   7. 对每个产生式，引入n个标记性非终结符用来计算的继承属性，产生式改变为，其中的综合属性就是的继承属性值，亦即上文所说的相对位置。特别对于A，若需要A的继承属性值，则对应整个产生式都归约后，A被压入到M1的位置，其继承属性在M1前一个位置上。
   8. 下面两条规则可以化简标记性非终结符的个数。



## 第七章



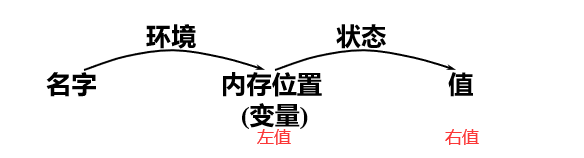




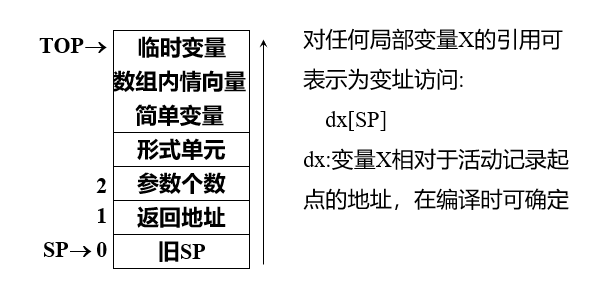
特别要注意调用生成语句代码gencode()时会使得nextquad自增1。最好一边写代码一边写明nextquad的值。

## 第九章

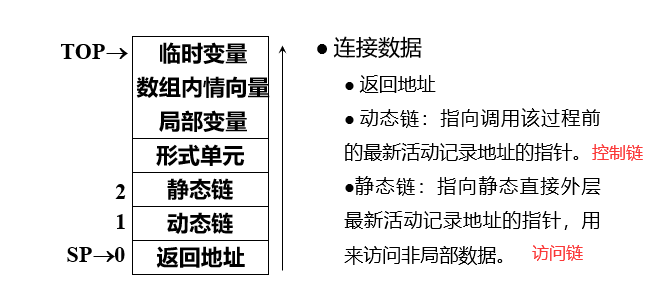
1. 关键概念：
   1. 静态策略：在编译时即可做出决定的策略
   2. 动态策略：知道程序执行时才能做出决定的策略
   3. 名字：编译时的名字
   4. 变量：运行时改名字所代表的**内存位置**
   5. 标识符：一串字符串，标识符一定是名字，但名字不一定是标识符，如结构体X.Y
   6. 环境和状态：



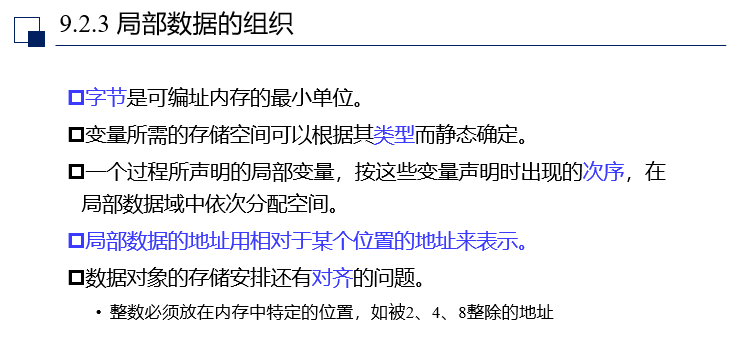
1. 声明的作用域：
   1. 如果只考察程序就可以确定某个声明的作用域，则称该语言使用**静态作用域**；否则称该语言使用**动态作用域**。面向对象语言是用public等关键字对作用域进行显式控制。
   2. 静态作用域采用最近作用域规则，一般可以用程序块来区分。
   3. 动态作用域规则相对于时间。如使用宏定义#define A x+1，使用A时要判断x是什么，这需要去寻找最近一次定义名字x的声明是什么。
2. 过程及其活动：
   1. 过程：将不同语言中的“过程、方法、函数”统称为“过程”。
   2. 过程定义是一个声明，最简单形式是把一个标识符和一个语句联系起来。标识符是过程名，语句是过程体。
   3. 过程调用是执行被调用过程的过程体。
   4. 过程定义中的某些标识符有特殊意义，称为形参。调用过程时将表达式作为实在参数传递给被调用的过程，以替代形参。
   5. 过程体的每次执行叫做该过程的一个活动。
   6. a和b是过程的活动，那么他们的生存期或者不交迭，或者嵌套。
   7. 如果同一个过程的一次新活动可以在前一次活动结束前开始，则称这样的过程是递归的。
3. 传值四种方式：
   1. 传值：计算实参并将其右值传递给被调用过程
   2. 传地址：将实参的地址传递给被调用过程，对类似a+b表达式这样的实参，要先开辟新的存储单元存放a+b，再将该存储单元的地址传给被调用过程
   3. 传值结果：将地址和右值都传给被调用过程，形参复制右值的值，计算出结果后将形参的值复制到实参的地址当中。
   4. 传名：用实参表达式对形参进行文字替换。类似于宏定义替换。
4. 活动记录：
   1. 过程中的每个活动所需要的信息用一块连续的存储区来管理，这块存储区叫做活动记录。
   2. 运行时，每当进入一个过程就有一个相应的活动记录压入栈顶。活动记录一般都有局部变量区域、返回值区域等。·
   3. 非嵌套语言的活动记录内容：



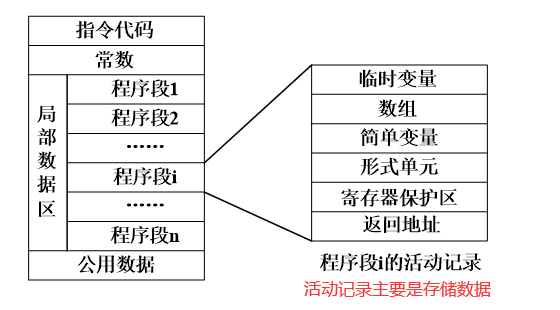
* 1. 嵌套语言的活动记录内容：



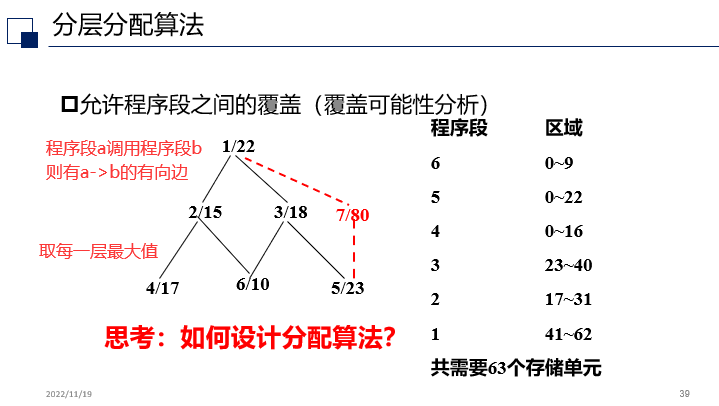
1. 局部数据的组织：



1. 全局数据的组织：
   1. 静态存储分配策略：在编译时就能确定数据空间的大小，为每个数据项目确定处在运行时刻的存储空间中的位置。
   2. 动态存储分配策略：在编译时不能确定数据空间大小，比如有递归、动态申请内存等情况时，需要采取栈式或堆式存储分配。
2. 静态存储分配策略：

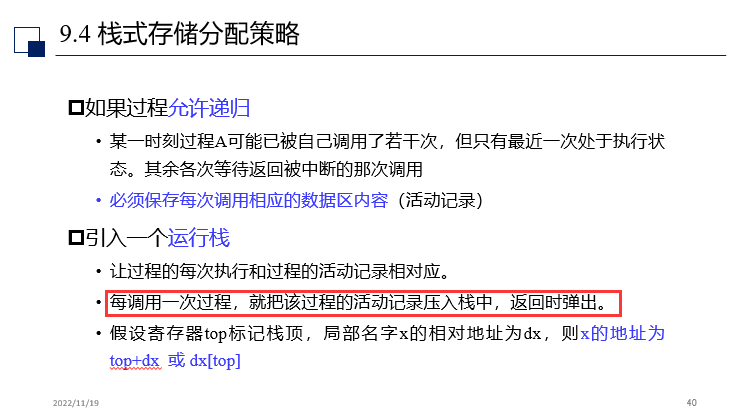


不需要运行栈，程序段编号顺序排放。

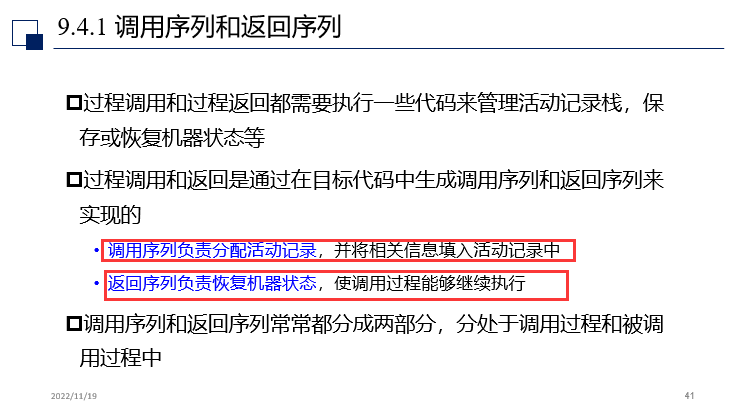


1. 栈式存储分配策略：

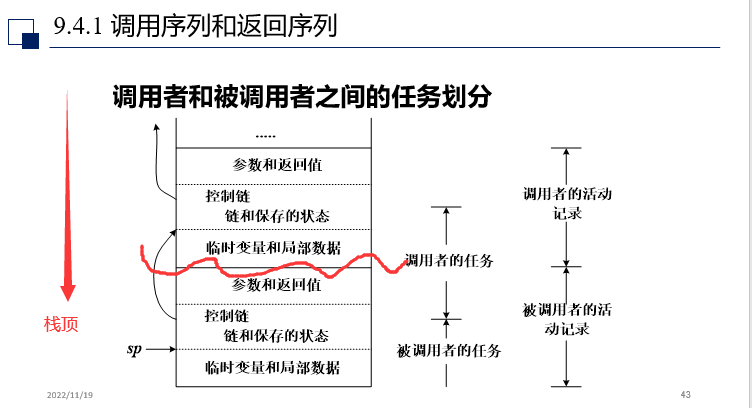
引入运行栈，用栈来模拟过程的执行过程。



* 1. 调用序列和返回序列：



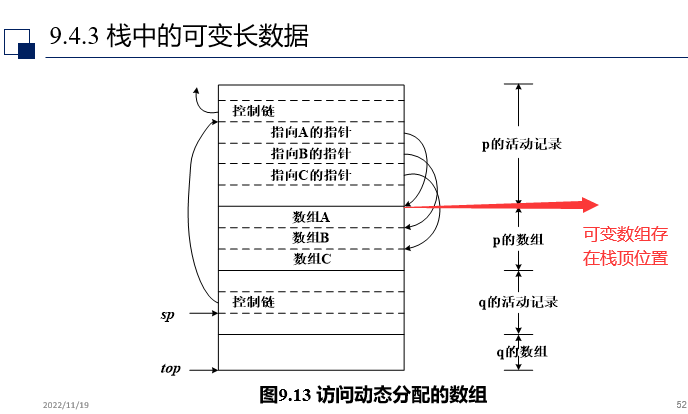
* 1. 调用者和被调用者之间的任务划分：



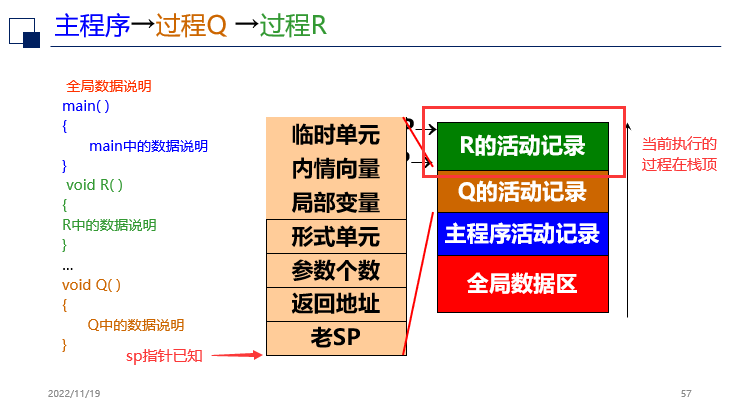
* 1. 可能产生的目标代码：



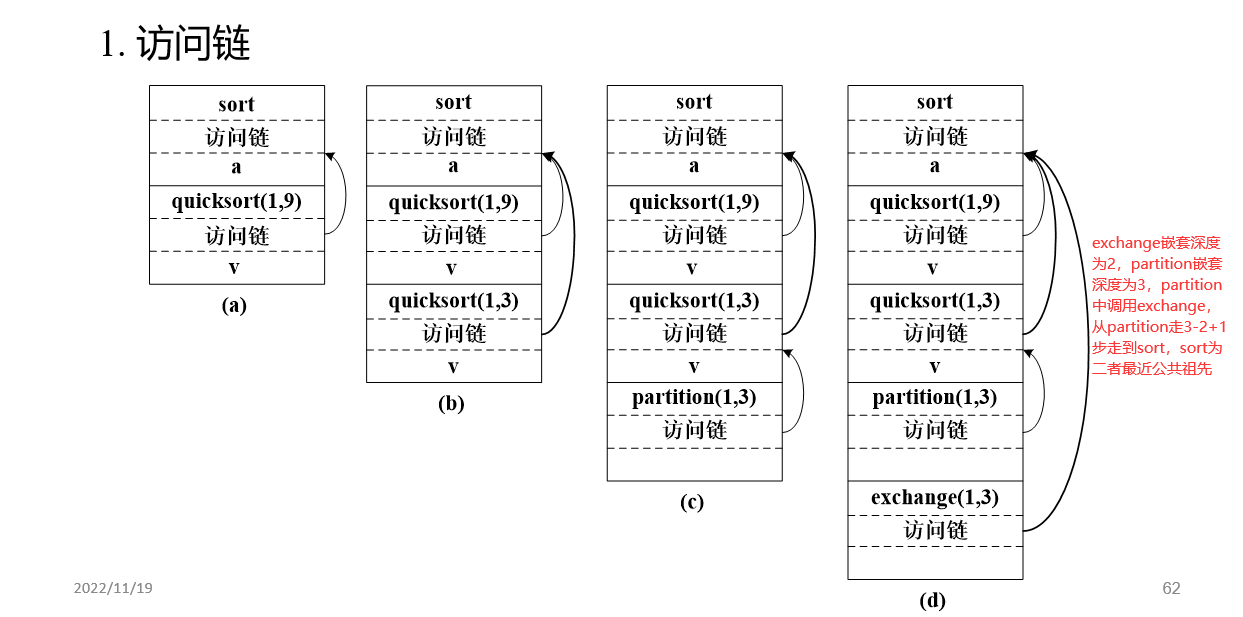
* 1. 栈中可变长数据：



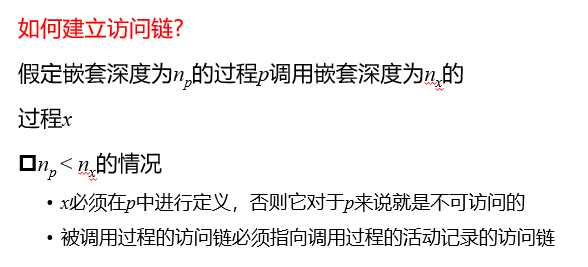
1. 栈中非局部数据的访问：
   1. 无过程嵌套的静态作用域：不允许函数嵌套定义。
      1. 局部变量就在栈顶的活动记录中，通过sp指针可以定位。
      2. 无须深入栈中取数据，无须访问链。
      3. 过程可以作为参数传递，也可以作为结果返回。

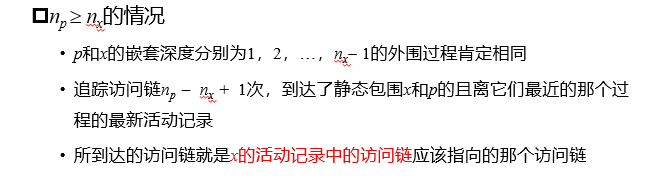


* 1. 有过程嵌套的静态作用域：
     1. 使用变量需要找到该变量所在的活动记录。
     2. 引入访问链和嵌套深度来寻找活动记录。
     3. 建立访问链：

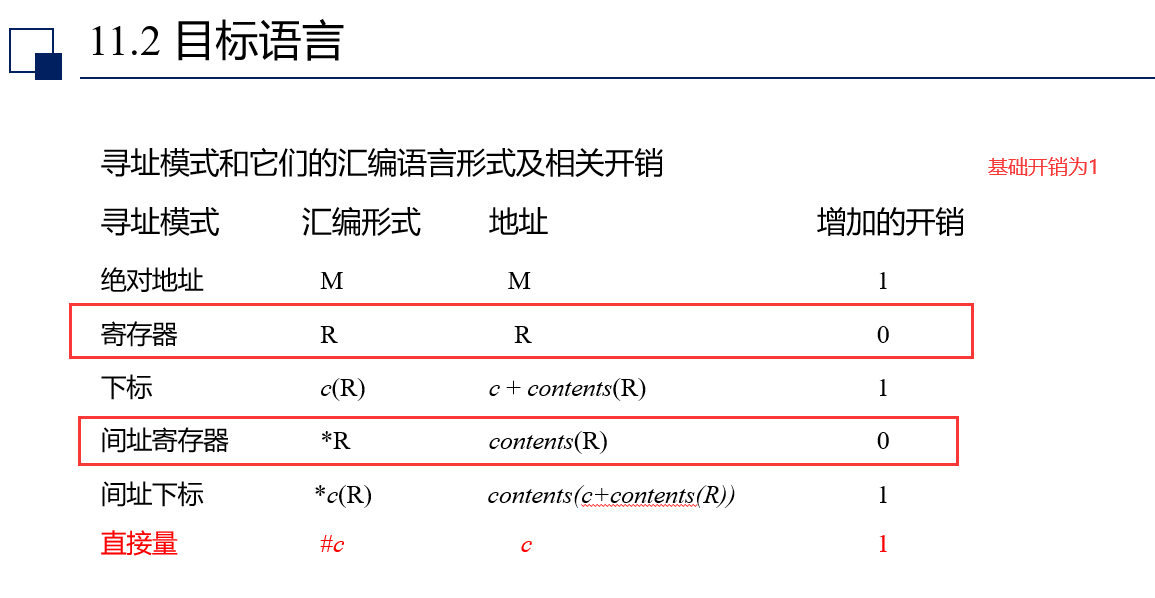


为被套用的过程x指向套用它的过程





## 第十一章



## 必须要掌握的题型：

1. 编译器的整体结构（编译器所包含的各个部分及其相应的输入输出和功能作用）
2. **T形图题型**
3. **根据要求写正则表达式**
4. 正则表达式与正则文法之间的转换
5. 正则表达式与状态转换图之间的转换
6. **求文法FIRST集和FOLLOW集**
7. **LL(1)构造预测分析表**
8. 算符优先文法
9. **LR(1)**
10. 语法制导翻译思想
11. 布尔表达式回填

## 必须要掌握的一些例题：

1. 改写S->SaS|SbS|ScS|d为非二义性文法
2. E->E\*E|E+E二义性文法改写
3. if-else语句的二义性改写