Lecture01：概述&预备知识

Lecture02：实验导引

Lecture03：词法分析

词法分析概述

词法分析程序的设计与实现（状态转换图、扩展巴克斯范式EBNF 描述词法规则）

词法分析程序自动构造

Lecture04：自顶向下语法分析

基本思想（按给定文法和终结符号串进行推导）

确定的自顶向下分析

消除第一类不确定（带回溯的自顶向下分析、最左推导）

消除第二类不确定（自顶向下预测分析）

**LL(1) 分析**

第一个 “L” 代表从左向右扫描单词符号，第二个 “L” 代表产生的是最左推导，“1” 代表向前查看（lookahead）一个单词符号。

First 集合：一个句型α若可以推导出另一个以终结符 a 开头的句型，那么 a 属于First（α）；若α可以推导出 ε，那么 ε 属于 First（α）

Follow 集合：，若 G 中存在一个包含子串 Xa 的句型，那么 a 属于 Follow(X)；G 中存在一个以 X 结尾的句型，那么 # 属于 Follow(X)。

预测集合 PS (A->α) 定义如下：

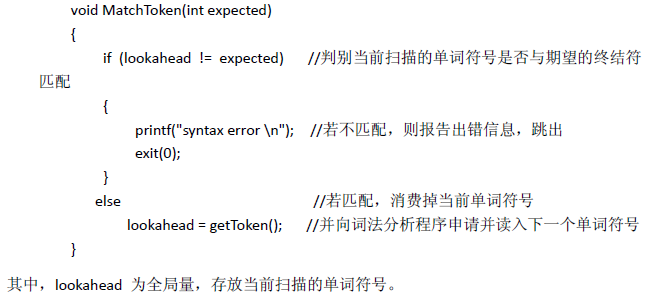
如果ε不属于First (α)，那么 PS (A->α) = First (α) ；

如果ε∈First (α)，那么 PS (A->α) = ( First (α) – {ε} ) ∪ Follow(A)



LL（1）分析的实现

**递归下降 LL（1）分析程序**



**表驱动 LL（1）分析程序**

预测分析表：表的每一行对应一个非终结符，表的每一列对应某个终结符或输入结束符#，表中的项是一个产生式集合。

对文法 G 的每个产生式，若它的预测集合 PS(A->α) 中包含 a∈VT∪{#}，那么将A->α加入 表项M[A,a]。当 M(A,a) 不含产生式时，就对应一个出错位置。

一个文法 G 的预测分析表中每个表项最多只包含一个产生式，当且仅当 G是 LL(1) 文法。对于LL（1）文法，预测分析表也称为**LL（1）分析表**。

**表驱动 LL（1）分析程序**的工作原理为：

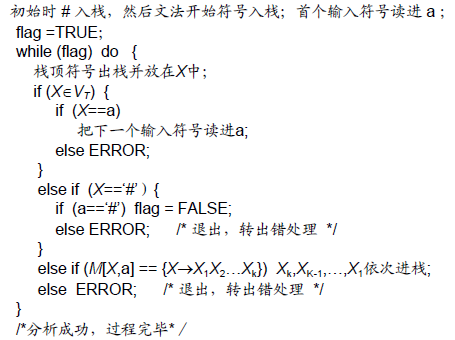
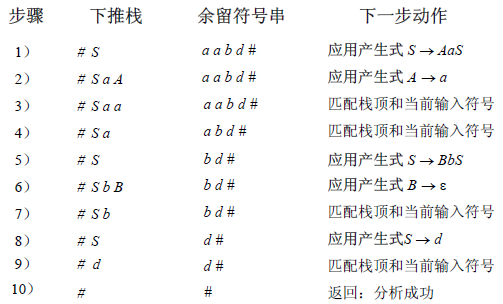
初始时，下推栈只包含 #；首先将文法开始符号入栈；然后执行如下步骤：

（1）若栈顶为终结符，则判断当前读入的单词符号是否与该终结符相**匹配**，若匹配，再读取下一单词符号继续分析；若不匹配，则进行错误处理。

（2）若栈顶为非终结符，则根据该非终结符和当前输入单词符号**查LL（1）分析表**，若相

应表项中是产生式（唯一的），则将此非终结符出栈，并把产生式右部符号**从右至左**入栈；若表项为空，则进行错误处理。

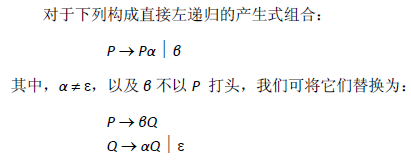
（3）重复（1）和（2），直到栈顶为 # 同时输入也遇到结束符 # 时，分析结束。

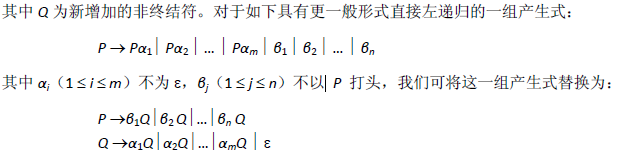
 

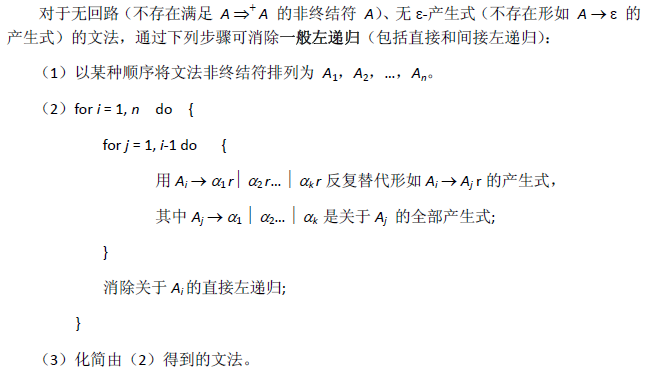
**文法变换：消除左递归、提取左公因子（构造出所期望的文法类型）**

**左递归**的，直接左递归，间接左递归。

包含左递归或左公因子的文法一般不是LL(1) 的。

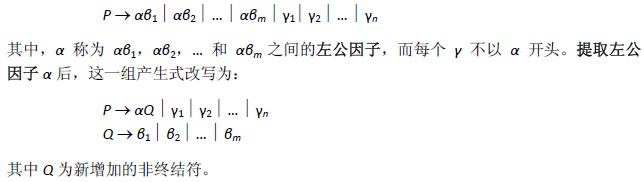






如果对非终结符的不同排序方式，依上述步骤消除一般左递归后所得到的文法从形式上看可能会有区别，但它们之间是相互等价的。

**提取左公因子**



**LL(1) 分析中的出错处理**

一是报错，发现错误时应尽可能准确指出错误位置和错误属性；二是错误恢复，尽可能进行校正，使编译工作可以继续下去，提高程序调试的效率。

对于表驱动 LL（1）分析，在以下两种情况下需要报错：栈顶的终结符与当前输入符号不匹配；非终结符A 位于栈顶，面临的输入符号为a，但分析表 M 的表项 M[A , a] 为空。

短语层恢复

Lecture05：自底向上语法分析

基本思想（按给定文法和终结符号串进行归约）

两种不确定（在每一步归约中，选择哪个产生式，以及匹配哪个位置上的子串）

对于文法的某个句型或句子，如果一个子串根据某个产生式归约后，得到文法的另一个句型，那么我们称这样的子串为可归约串；如果没有这样的产生式，那么它就是不可归约的串。总是选择某个“可归约串”进行归约，可大大减少回溯。

短语和直接短语





某个句型的短语：先给出以所考虑的句型为果实的分析树，每个非叶子结点（的子树果实串）与该句型的一个短语是一一对应的。

直接短语：在句型的分析树中，只包含直接子孙的非叶子结点（的子树果实串）与该句型的

直接短语是一一对应的。

句柄（只适合于右句型，如果文法无二义，则句柄是唯一的）

句型的直接短语不唯一，通常的做法是选择最左边的直接短语进行归约。



一个终结符串是合法的，即属于 L(G)，当且仅当它是一个右句型。

可以在自底向上分析的每一步中只使用句柄进行归约，如果我们把这种按照句柄进行归约的过程逆过来，实际上就相当于一个最右推导过程。

某个句型的句柄：任何一个句子一定是右句型，做它的最右推导，最后一步产生式的右边即是句柄。或者，画分析树，先找出所有的直接短语，最左边的直接短语就是句柄。

移进-归约分析（实现自底向上分析最常用的技术）

基于有限状态控制的分析引擎根据当前状态、分析栈当前内容、余留的输入符号串来唯一确定如下动作之一，然后进入新状态：

Reduce: 依确定的产生式序列对位于栈顶的短语进行**归约**。

Shift: 从输入序列**移进**一个单词符号。

Error: 发现语法错误，进行错误处理/恢复。

Accept：分析成功。

在进行Reduce 动作时可将短语、直接短语、或句柄替换为某个非终结符。LR 分析方法中，是对句柄进行归约。

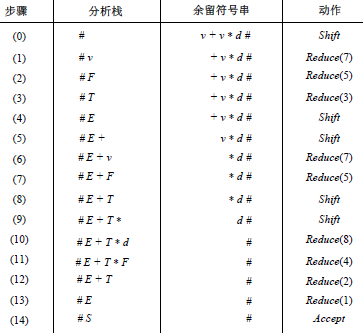
移进-归约分析过程确定化的关键是解决好两类冲突：移进-归约冲突和归约-归约冲突。

**移进-归约冲突**是指到达一个不能确定下一步应该移进还是应该归约的状态。

**归约-归约冲突**是指到达一个可以对多个短语进行归约的状态。

为解决这些冲突，经常采用向前查看确定数目的单词符号、规定特定优先关系等技术。然而，不可能找到通用的解决方法适合于任何文法。

为简化状态控制，通常会借助于一张分析表，称之为表驱动的移进-归约分析方法。



**LR 分析方法**

LR 中的“L” 代表从左向右扫描单词符号，“R” 代表产生的是最右推导。

最右推导的逆过程对应于自底向上的每一步对句柄进行归约。

分析过程中使用的分析表称为 LR分析表，分析栈称为状态分析栈，简称状态栈。（之前的分析栈称为符号分析栈，简称符号栈）

**ACTION表：**告诉分析引擎，栈顶状态为k, 当前输入单词符号是 a 时应该完成的动作。

ACTION [k,a] = si， Shift：状态 i 移进栈顶

ACTION [k,a] = rj， Reduce：按第 j 条产生式归约

ACTION [k,a] = acc， Accept：分析完成

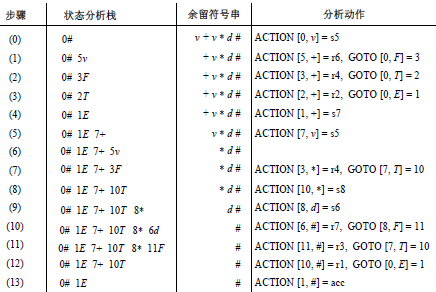
ACTION [k,a] = err，Error：发现错误 （常标为空白）

**GOTO表**：GOTO[i,A] = j ，告诉分析引擎，在依产生式 A->β归约之后，位于栈顶的状态如何改变。依产生式 A->β归约时，要将栈顶的|β|个状态弹出，假设此时位于栈顶的状态是 i，那么就将新状态为 j 移进栈顶。

LR分析表的每行对应分析引擎的一个状态；初始状态通常表示为0。

ACTION 表的每一列对应一个输入单词符号或#，表项取值为 err 时用空白表示。

GOTO 表的每一列对应一个文法非终结符。



LR（0）、SLR（1）、LR（1） 和LALR（1） 等分析方法有着不同的分析能力，所适合的文法分别称为 LR（0） 文法、SLR（1） 文法、LR（1） 文法和 LALR（1） 文法。这些方法采用了结构上是一致的分析表，即LR 分析表，只是构造方法上存在差异。

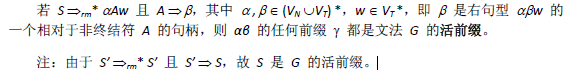
“0” 代表向前查看 0 个符号，“1” 代表向前查看 1 个符号，SLR（1）代表 Simple LR（1），LALR（1） 代表Look Ahead LR（1）。

**LR（0） 分析**

文法G = (VN , VT , P , S )增加产生式S’->S，S’不属VN∪VT，得增广文法G’ = (VN , VT , P , S’ )。

增广文法的开始符号不会出现在任何产生式的右部，若原文法的开始符号已经满足这个条件，那么可以不作这样的改造。

活前缀





**LR（0）有限状态机**

一个**LR（0） 项目**是在右端某一位置有圆点的产生式。

圆点标志着已分析过的串与该产生式匹配的位置，根据该位置和圆点后是终结符还是非终结符或为空，把项目分为以下几种：

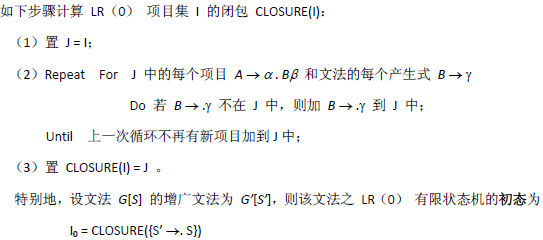
移进项目: 形如A->α.aβ，其中a为终结符，α、β为（非）终结符或空。

待约项目: 形如A->α.Bβ

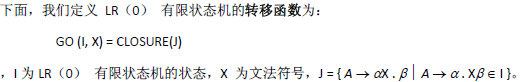
归约项目: 形如A->α.

接受项目: 形如S’->S.

LR（0）有限状态机的任一状态是某个LR（0）项目集 I 的闭包CLOSURE(I)。





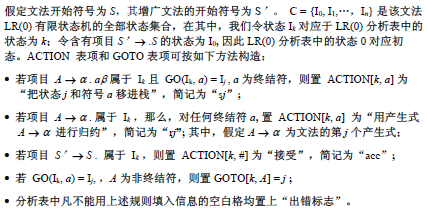


该方法构造的 LR（0）有限状态机可以看作一个以（非）终结符为字母表的 DFA。

如果假定除了（没有绘出的）死状态是非终态外，其余都是终态，那么可有结论：该DFA的语言是文法 G 的所有活前缀的集合。，即任何句型按该DFA不会错过任何可归约的句柄，该DFA 的所有可接受的符号串都是活前缀。

**LR（0）分析表**

LR（0）分析表可以直接从LR（0）有限状态机得到



LR（0） 分析表的一些表项可能会被多重定义，即被不止一个 rj 或 sj 赋值。这种情况下不合适进行LR 分析过程。如果各表项均无多重定义，则称它为相应文法 G的一张 LR（0）表，并称 G 为一个 LR（0）文法。

**文法 G 是 LR（0）文法，当且仅当它 的LR（0）有限状态机中的每个状态都满足：**

**不同时含有移进项目和归约项目，即不存在移进-归约冲突；**

**不含有两个以上归约项目，即不存在归约-归约冲突。**

**SLR（1） 分析**

通过向前查看一个输入符号，可以根据这一符号是否属于要归约的非终结符的Follow 集来决定是否进行归约。如果可以使LR（0） 有限状态机中所有状态的冲突问题得到解决，那么就可以进行正常LR 分析。称这种方法为SLR（1）分析方法，它的基本原理是检查LR（0） 有限状态机的所有状态是否有冲突，并借助于以下两个方面来解决冲突问题：

如果每一状态归约项中要归约的非终结符的 Follow 集互不相交，则可解决归约-归约冲突。

如果每一状态归约项中要归约的非终结符的 Follow 集与该状态所有移进项要移进的符号集互不相交，则可以解决移进-归约冲突。

只需对 LR（0）分析表进行简单修改，使得归约表项只适用于相应非终结符Follow 集中的输入符号，就可得到 SLR（1） 分析表。

SLR（1） 分析表中一些表项可能会被多重定义，即被不止一个 rj 或 sj 赋值。这种情况不合适进行 LR 分析。如果各表项均无多重定义，则称它为相应文法 G的一张 SLR（1） 表，并称 G 为一个 SLR（1） 文法。

**文法 G 是 SLR（1） 文法，当且仅当它 的LR（0） 有限状态机中的每个状态都满足：**

**对该状态的任何项目A->α.aβ（a为终结符），不存在项目 B->γ.，使得a属于Follow(B)，即不存在移进-归约冲突。**

**对该状态的项目A->α.和 B->β.，满足Follow(A)交Follow(B) =空，即不存在归约-归约冲突。**

SLR（1）分析和LR（0）分析区别在归约句柄时，前者要向前查看一个输入符号，看是否属于要归约的非终结符的 Follow集，若是则归约，否则出错处理。而后者无论下一个输入符号是什么都进行归约。LR（0）分析出错处理会比SLR（1）分析晚一些。区别体现在表上：

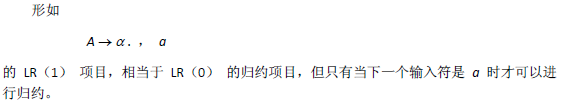
在 LR（0） 表的 ACTION 表中，归约表项总是整行出现的，即每一个归约动作都是对应所有输入符；同时，在任何一行，不会既有移进表项又有归约表项。

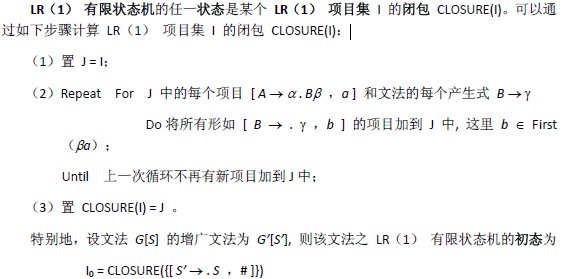
在 SLR（1） 表的 ACTION 表中，归约表项只适用于相应非终结符 Follow 集中的输入符号；在同一行，可以既有移进表项又有归约表项。

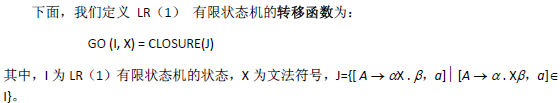
**LR（1）分析**

SLR（1）分析方法只考虑到所归约非终结符的 Follow 符号。但一个输入符号属于所归约非终结符的 Follow集合，未必就是句柄可以后跟的符号。

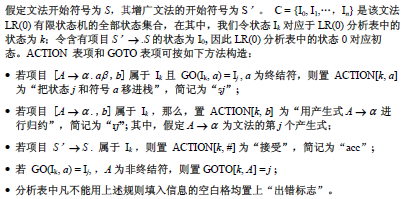
**LR（1）项目**是在 LR（0）项目基础上增加一个终结符，称为向前搜索符，表示产生式右端完整匹配后所允许在余留符号串中的下一个终结符。







**LR（1） 分析表**



一个文法的 LR（1） 分析表中的一些表项有可能会被不止一个 rj 或 sj 赋值。这种情况不合适进行 LR分析的。如果各表项均无多重定义，则称它为相应文法 G 的一张 LR（1） 表，并称 G 为一个 LR（1） 文法。

文法 G 是 LR（1） 文法，当且仅当它 的LR（1） 有限状态机中的每个状态都满足：

如果该状态含有项目 [A->α.aβ, b] （a为终结符），那么就不会有项目[B->γ., a] ；反之亦然。这表明该状态不存在移进-归约冲突。

该状态中的所有归约项目的向前搜索符互不相交，即不同时含有项目 [B->α., a] 和 [B->β., a]。这表明该状态不存在归约-归约冲突。

**LALR（1） 分析**

可以采用 SLR（1） 分析的文法一定可以采用 LR（1） 分析；但反之不成立。表明：在所有以确定方式工作的分析程序中，LR（1） 分析程序已经达到了最强的分析能力。

通过合并 LR（1）有限状态机中的同芯状态，可得到相应的 LALR（1） 有限状态机。

LR（1）项目 [A->α.β, a ] 中的A->α.β部分称为该项目的芯。对于两个状态，如果只考虑每个项目的芯时它们是完全相同的项目集合，那么这两个状态就是**同芯状态**。

**合并同芯状态**是将相互之间是同芯的状态合并为同一个状态，即将所有这些状态的项目集合全部并起来。（合并向前搜索符）

对于两个同芯状态，由 GO 函数得到的针对任何符号的后继状态仍然同芯。

所有合并后的新状态、未合并的状态以及改造后的GO 函数，一起构成了一个新的LR（1） 有限状态机，称之为 LALR（1） 有限状态机。

合并同芯状态后，不会产生新的移进-归约冲突，但有可能产生新的归约-归约冲突。

一个 LR（1） 文法，如果将其LR（1） 有限状态机的同芯状态合并后所得到的LALR（1） 有限状态机中的全部状态无归约-归约冲突，则该文法是一个 LALR（1） 文法。

LALR（1）状态数目实际上与LR（0） 有限状态机相同，分析要强于SLR（1） 分析。

**二义文法在LR 分析中的应用**

如果对语义（即可以产生的语言）进行一些合限定，那么有可能构造出相应的LR 分析程序。

规定\*的优先级高于 +，以及\* 和 + 都服从左结合性。

规定最近嵌套匹配原则。

**LR 分析中的错误处理**

自底向上分析处理语法错误更准确。原因在于推导时仅观察可推导出的输入串的一部分，而归约时可归约的输入串整体已全部出现，并且输入符号是在查看后才被移进的。

应急恢复

即兴恢复

**几类分析文法之间的关系**

所有的LR（0）文法都是SLR（1）文法。

所有的SLR（1）文法都是 LALR（1）文法。

所有的LALR（1）文法都是 LR（1）文法。

所有的LL（1）文法都是 LR（1）文法。

所有的LR（1）文法都是 无二义文法。

这些结论的逆命题是不成立的。

如果已知一个文法是二义的，那么它一定不是 LR（1）文法或者 LL（1）文法。

Lecture06：符号表

符号表的作用

符号表自创建后便开始被用于收集符号（标识符）的属性信息。

在语义分析中，符号表所登记的内容将用于上下文语义的合法性检查的依据。

同一标识符可能在程序不同地方出现，而有关该符号的属性是在不同情况下收集的，需检查标识符属性在上下文中的一致性和合法性。

在目标代码生成阶段，符号表是对符号名进行地址分配的依据。

符号表的组织与结构还需要体现符号的作用域与可见性信息。

表的常见属性

符号的名字、符号的类别、符号的类型（决定存储格式与运算操作）、符号的存储类别和存储分配信息、符号的作用范围与可见性等等。

符号表的实现

创建符号表、释放符号表空间；

插入、查询、修改、删除表项。

体现符号表的功能和作用，考虑符号表操作的方便性和高效性，考虑节省内存空间。

线性表、有序表、二叉树、Hash表。

符号表至少应该在静态语义分析之前已经创建，最常见的情况是在语法分析的同时创建。

符号表体现作用域与可见性

作用域之间可以嵌套，不会交错

对于程序的某一特殊点而言，该点所在的作用域称为当前作用域

当前作用域与包含它的程序单元所构成的作用域称为开作用域

不属于开作用域的作用域称为闭作用域

可见性是指在程序的某一特定点，哪些符号是可访问的。**可见性规则**：

1、在程序的任何一点，只有在该点的开作用域中声明的符号才是可访问的。

2、若符号在多个开作用域声明，则把离该符号的某个引用最近的声明作为该引用的解释。

3、新的声明只能出现在当前作用域。

多符号表组织：每个作用域都有各自的符号表

单符号表组织：所有嵌套的作用域共用一个全局符号表

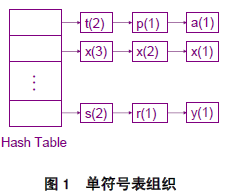
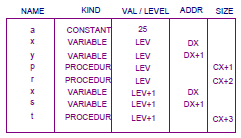
作用域与单符号表组织：

所有嵌套的作用域共用一个全局符号表。

每个作用域都对应一个作用域号

仅记录开作用域中的符号。

当某个作用域成为闭作用域时，从符号表中删除该作用域中所声明的符号。

作用域与多符号表组织

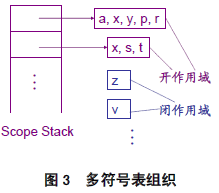
每个作用域都有各自的符号表。

需要维护一个符号表的作用域栈，每个开作用域对应栈中的一个入口，当前的开作

用域出现在该栈的栈顶

当一个新的作用域开放时，新符号表将被创建，并将其入栈。

在当前作用域成为闭作用域时，从栈顶弹出相应的作用域。



Lecture07：实验导引二

Lecture08：语法制导的语义计算基础

词法和语法正确的源程序，编译程序就可以对它进行语义分析。若发现程序存在静态一致性或完整性方面的问题，则报告静态语义错误。否则称它通过了静态语义检查，编译程序可将其翻译到后续的中间表示形式，即中间代码生成。

在编译程序的实现中，一种经典的方法是由语法分析程序的分析过程来主导语义分析的过程，本课程将其称为**语法制导的语义计算**。

为描述完成什么样的语义计算，需要我们在语法定义的基础上建立适当的语义计算模型。

**基于属性文法的语义计算**

不存在任何上下文无关文法接受某语言。如果对 G[S] 附加某种限定条件，使其只产生满足这一限定条件的字符串，则可能接受该语言。

在文法 G[S] 基础上，为文法符号关联有特定意义的**属性**，并为产生式关联相应的语义动作或条件谓词，称之为**属性文法**，并称文法 G[S] 是这一属性文法的基础文法。

设文法符号 X 关联一个属性 a，我们用 X.a 来表示对这个属性的访问。

为明确指出一个属性对应于当前产生式中哪个位置的文法符号，在书写同一文法符号时会用到文法符号的下标形式。*A* -> *A*1*a*

每个产生式A都关联一个语义计算规则的集合（花括号内的部分），每个语义计算规则或者是一个语义动作，或者是一个条件谓词。语义动作一般形式为*b* := *f*(*c*1, *c*2,*…*, *ck*)（**语义函数**）

形如 X.a := Y.b 的语义动作称为**复写规则。**

**对于给定的属性文法，在基于语法分析过程进行语义计算时，使用某个产生式完成一步分析时将执行相应的语义动作，而前提是必须满足相应的条件谓词。**

对关联于产生式A->α的语义动作 b := f(c1, c2,…, ck)，如果 b 是 A 的某个属性, 则称 b 是 A 的**综合属性**。计算综合属性是对父结点的属性赋值，“自底向上”传递信息。

对关联于产生式A->α的语义动作 b := f(c1, c2,…, ck)，如果 b 是产生式右部某文法符号X 的某属性, 则称 b 是X 的**继承属性**。计算它是对子结点的属性赋值，“自顶向下”传递信息。

**遍历分析树进行语义计算**

构造输入串的语法分析树，构造依赖图。若该依赖图是无圈的，则按造此无圈图的一种拓扑排序对分析树进行遍历，从而计算所有的属性值；若依赖图含有圈，则这一步骤失效。

**依赖图**是一个有向图。

可以用**带标注语法分析树**表示属性的计算结果。

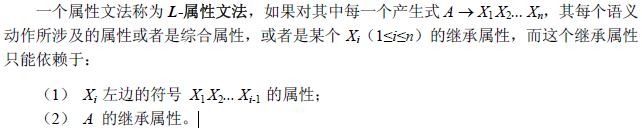
通过遍历分析树进行属性计算的方法有一定通用性，但它是在语法分析遍之后进行的。

实际编译程序中，语法制导的语义计算大都采取单遍过程，即语法分析过程的同时就完成相应的语义动作。属性计算仅对应一个自顶向下或是自底向上的简单过程。要求对属性文法进行某种限制。

**S-属性文法和L-属性文法**

只包含综合属性的属性文法称为S-属性文法。

L-属性文法既可以包含综合属性，也可以包含继承属性，但要求产生式右端某文法符号的继承属性的计算只取决于该符号左边符号的（继承）属性



S-属性文法是L-属性文法的一个特例。

**基于S-属性文法的语义计算（自底向上）**

若S-属性文法的基础文法可用LR 分析进行语法分析，那么可通过扩充分析栈中的域，形成**语义栈**来存放综合属性的当前取值。若该符号有多属性，可对应多元组的形式。

**基于L-属性文法的语义计算**（自顶向下深度优先从左至右遍历分析树）

将L-属性文法描述的语义计算融入到LL(1)预测分析过程之中

**基于翻译模式的语义计算**

翻译模式在形式上类似于属性文法，但允许由{}括起来的语义动作出现在产生式右端的任何位置，以此显式地表达属性计算的次序。

S-翻译模式：仅涉及综合属性，通常将语义动作集合置于相应产生式右端的末尾。

L-翻译模式：包含综合属性和继承属性，需满足：

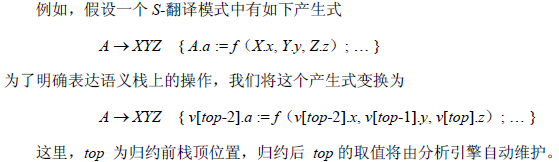
（1）产生式右端某个符号继承属性的计算必须位于该符号之前，其语义动作不访问位于它右边符号的属性，只依赖于该符号左边符号的（继承）属性；

（2）产生式左部非终结符的综合属性的计算只能在所用到的属性都已计算出来后进行，通常将相应的语义动作置于产生式的尾部。

**基于S-翻译模式的语义计算**，形式上与S-属性文法一致，自底向上。

为进一步解释基于自底向上分析（如LR分析）的语义计算过程，要讨论每个产生式归约时需要执行的**语义计算代码片断**：

假设语义栈由向量v表示，归约前栈顶位置为top，栈上第i个位置所对应符号的综合属性值x用v[i]. x表示。



**基于L-翻译模式的自顶向下语义计算**

将L-翻译模式描述的语义计算过程融入递归下降分析程序的改造过程之中

称改造后的分析子函数（过程）为语义计算子函数（过程），并称改造后的递归下降分析程序称为递归下降（预测）语义计算程序，或**递归下降（预测）翻译程序。**

相应于分析子函数的设计，改造后子函数代码的流程不同之处是要将语义动作嵌入其中，具体可描述为：

若遇到一个终结符 X，首先将其综合属性x的值保存至专为 X.x 而声明的变量；然后，判断当前读入的输入符号是否与该终结符相匹配，若匹配，则继续读取下一个输入符号；若不匹配，则报告和处理语法错误。

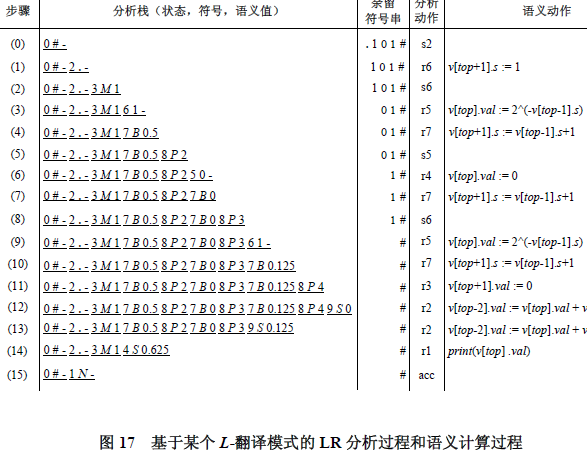
若遇到一个非终结符B，利用相应于 B 的子函数ParseB产生赋值语句c := ParseB(b1，b2，…，bk)，其中**参量 b1，b2，…，bk 对应于 B的各继承属性**，**变量c对应B的综合属性**（若有多个综合属性，则可使用记录类型的变量）。

若遇到一个语义动作集合，则直接复制其中每一语义动作所对应的代码，只是需要注意将属性的访问替换为相应变量的访问。

**基于L-翻译模式的自底向上语义计算**

先解决继承属性的普通函数求值问题，再解决其访问一致性，我们的目标是：通过变换翻译模式（如增加新的文法符号，增加相应的复写规则和产生式），使嵌在产生式中间的语义动作集中仅含复写规则，并使得在自底向上的语法分析过程中，文法符号的所有继承属性均可以通过归约前已出现在分析栈中的综合属性唯一确定地进行访问。

需要注意的是：不可以改变L-翻译模式的特性。若不是L-翻译模式，则不能保证归约前需要访问的综合属性已出现在分析栈中。

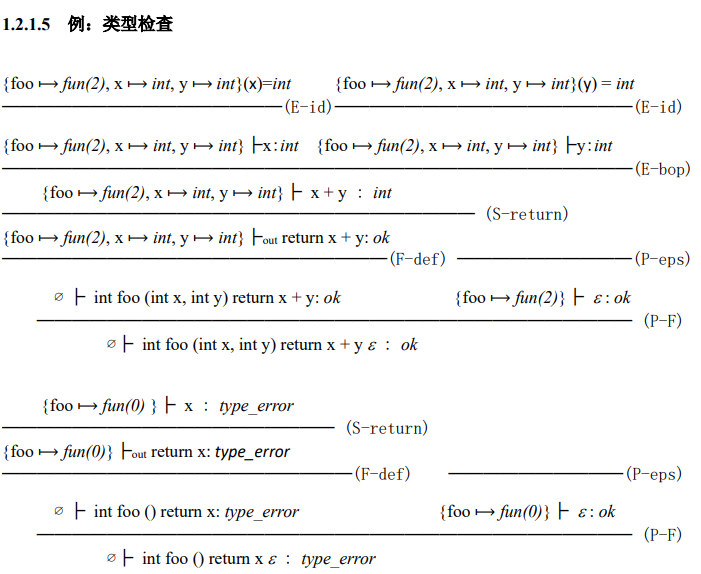


Lecture09：实验导引（三）

Lecture10：静态语义分析与中间代码生成

**语法制导的静态语义分析**

包括控制流检查、唯一性检查、名字的上下文相关性检查、类型检查



实现类型检查程序的算法通常可以借助于翻译模式进行规范/描述。例子见讲义。

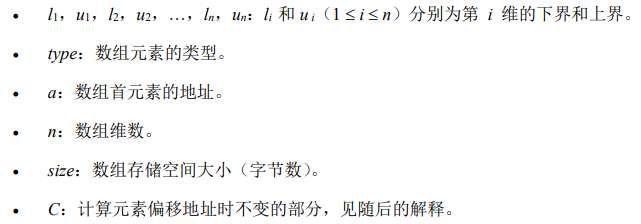
**语法制导的中间代码生成**

在 TAC 层次，变量名字所对应的存储位置信息（相对基地址的偏移量）总是可以从符号表中得到。变量的取值即为其名字对应的存储位置上存储单元的内容。

表达式、语句的翻译：（略）

声明的翻译：计算偏移量，存入符号表，注意区分堆上的空间（全局变量）和栈上的。

数组的翻译：维护数组的有关信息（内情向量），静态数组内情向量可放在符号表中；动态可变数组在运行时建立相应的内情向量。



拉链与代码回填：用于处理控制流的S-翻译模式。当生成的转移语句不能确定目标语句标号时，先将目标语句标号的位置用 ‘\_’ 表示，并将该转移语句的地址加入到某个链表（真链、假链、next链）中；当这个目标语句标号可以确定时再将其回填至 ‘\_’ 处。

Lecture11：运行时存储组织

运行时存储组织的作用与任务（略）

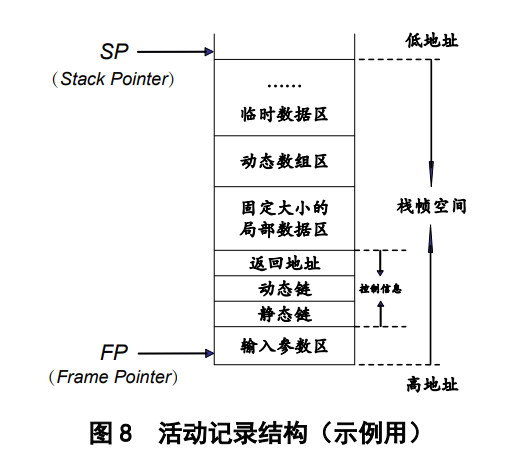
**程序运行时存储空间的布局**

代码区、数据区。

**存储分配策略**

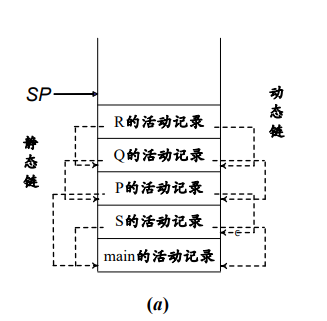
静态、栈式、堆式。

**活动记录**

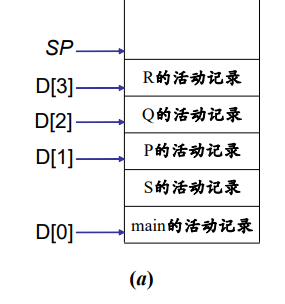


分配动态数组的空间：编译阶段先在初始活动记录中分配内情向量单元（数组的上界）和数组开头的指针的存储位置。运行时更改具体的值和栈指针。

静态链用于**嵌套过程的非局部量访问**，而动态链用于返回。静态链指向**当前过程的直接外过程**运行时的最新活动记录（的基址）。动态链指向**调用过程**的活动记录（的基址）。



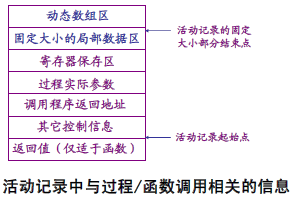
Display表记录各嵌套层当前过程的活动记录在运行栈上的起始位置，规定大小上界。可以只在活动记录保存一个 Display 表项，而在静态存储区或专用寄存器中维护一个全局 Display表。



嵌套语句块的非局部量访问仅需要借用其所属的过程级活动记录（调用栈）

动态作用域：针对嵌套过程中非局部量的使用，若遵循静态作用域规则，要沿着过程活动记录的静态链（或 display 表项）查找最近一个过程中所声明的同名变量；若遵循动态作用域规则，则要沿着过程活动记录的动态链查找最近一个过程中所声明的同名变量。

**过程调用与参数传递**



在调用前，调用函数/过程（caller）完成：

（1）保存caller-saved寄存器；（2）参数传递（寄存器、压栈） ；（3）保存其它控制信息（静态链、display 表等）（4）保存PC作为返回地址（5）跳转到目标地址

然后，在调用期间，被调用函数/过程（callee）完成：

（1）为局部量设置新 SP 寄存器的值（2）保存旧 FP 寄存器的值（3）保存callee-saved寄存器（4）设置新 FP；（5）启动被调用函数/过程体的执行。

在调用收尾阶段：

（1）如果被调用过程是函数，则返回值；（2）恢复callee-saved寄存器（3）恢复调用时的 FP 和 SP 指针（4）恢复PC。

最后，控制返回到caller，恢复保存的caller-saved寄存器，继续caller的执行。

Lecture12：:代码优化和目标代码生成

**基本块、流图和循环**

基本块（Basic Block）是指程序中一个顺序执行的语句序列，其中只有一个入口语句和一个出口语句。执行时只能从其入口语句进入，从其出口语句退出。

最大基本块

基本块的入口语句可以是下面任意三类语句：（1）程序的第1条语句；或者，（2）条件跳转语句或无条件跳转语句的跳转目标语句；或者，（3）条件跳转语句后面的相邻语句。

划分基本块的方法：

（1）先求出各个基本块的入口语句；

（2）对每一入口语句，构造其所属的基本块。它是由该语句到下一入口语句（不包括下一入口语句），或者到某个跳转语句（包括该跳转语句），或者到某个停语句（包括该停语句）之间的语句序列组成的。

可以为构成程序的基本块增加控制流程信息，方法是构造一个有向图，称之为流图或控制流图（CFG，Control-Flow Graph）。流图以基本块集为结点集的有向图；第一个结点为含有程序第一条语句的基本块，称为首结点。

流图中，某一个基本块运行之后可以到达的所有基本块是该基本块的后继基本块，可以直接运行并到达某一个基本块的所有基本块是该基本块的前驱基本块。

对任意两个结点m和n而言，如果从流图的首结点出发，到达n的任一通路都要经过m，则称m是n的支配结点（dominator），记为m DOM n

流图中结点n的所有支配结点的集合，称为结点n的支配结点集，记为D(n)

任意结点是自身的支配结点，首结点是任意结点的支配结点。

假设 n->d 是流图中的有向边，如果 d DOM n 则称 n->d 是流图中的一条回边（back edge）。

如果已知有向边n->d是回边，就可求出包含该回边的自然循环（natural loop），简称循环.

为做好代码生成和代码优化工作，通常需要收集整个程序流图的一些特定信息，并把这些信息分配到流图中的程序单元（如基本块、循环、或单条语句等）中。这些信息称为数据流信息，收集数据流信息的过程称为**数据流分析**（data-flow analysis）。

数据流方程用于描述流入和流出某程序单元或程序中不同点之间的数据流信息之间的联系。

到达-定值数据流分析

对变量 A 的定值（definition）是指一条（TAC）语句赋值或可能赋值给 A。最普通的定值是对 A 的赋值或读值到 A 的语句，该语句的位置称作 A 的定值点。

变量 A 的定值点 d 到达某点 p，是指流图中从 d 有一条路径到达 p 且该通路上没有 A 的其它定值。

**为了求出到达点 p 的各个变量的所有定值点**，对程序中所有基本块B，定义下面几个集合：

in[B]：到达基本块 B 入口处的各个变量的所有定值点集合。

out[B]：到达 B 出口处的各个变量的所有定值点集合。

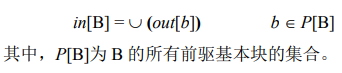
gen[B]：B 中定值的并能够到达 B 出口处的所有定值点集合，即 B 所“产生”的定值点集合。

kill[B]：B 外能够到达 B 的入口处，但所定值的变量在 B 中被重新定值；即 B 所“杀死”的定值点集合。

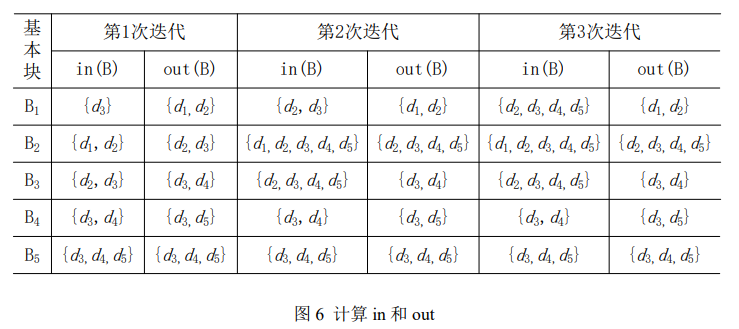
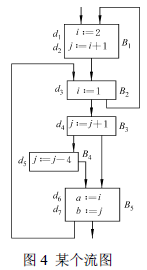
对于 out[B]，其计算方法为所有该基本块入口处的定值点集合 in[B]中去除当前基本块“杀 死”的定值点，再加入当前基本新“产生”的定值点，因此有：



某定值点d到达B的入口处，当且仅当它到达B的某一前驱基本块的出口处。



gen[B]和 kill[B] 是每个基本块 B 的固有属性，均可直接从基本块本身给出。



用迭代法解不动点方程。收敛时的 in 和 out 就是最后求出的结果

活跃变量数据流分析

一些数据流信息的获得依赖于从程序流图反方向进行计算，活跃变量信息就是其中的一种。

对程序中的某变量A 和某点p 而言，如果存在一条从p 开始的通路，其中引用了A 在点p的值，则称A 在点p 是活跃的。否则称A 在点p 是死亡的。

**为了求出各基本块B 出口处的活跃变量集合**，我们定义以下集合：

LiveIn[B]：B 入口处为活跃的变量的集合。

LiveOut[B]：B 的出口处的活跃变量的集合。

Def[B]：B 中定值的且定值前未曾在B 中引用过的变量集合。

LiveUse[B]：B 中被定值之前要引用变量的集合。

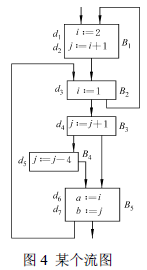
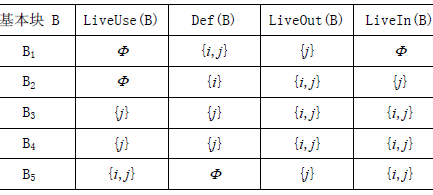


变量在B 中定值前有引用，或在B 出口处活跃且没有在B 中被定值，那么它在B 入口处就是活跃的。



变量在B 出口处活跃，当且仅当它在B 的某个后继基本块入口处活跃。

LiveUse[B]和Def[B]是每个基本块B 的固有属性，均可直接从基本块本身给出。

同样是迭代法解方程。

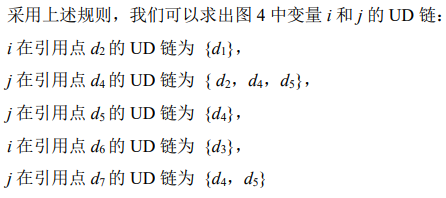
UD链

利用引用-定值数据流信息可以求出到达基本块B中某点p的任一变量A的所有定值点。

假设在程序中某点u引用了变量A的值，则把能到达u的A的所有定值点的全体，称为A在引用点u的引用-定值链（Use-Definition Chaining），简称 UD 链。

（1）如果在基本块B中，变量A的引用点u之前有A的定值点d，并且A在点d的定值可以到达u，那么A在点u的UD链就是{d}。

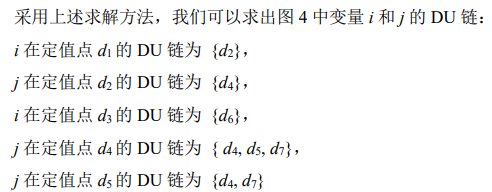
（2）如果在基本块中，变量A的引用点u之前没有A的定值点，那么，in[B]中A的所有定值点均到达u，它们就是A在点u的UD链。



DU链

假设在程序中某点 p 定义了变量 A 的值，从 p 存在一条到达 A 的某个引用点 s 的路径，且该路径上不存在 A 的其他定值点，则把所有此类引用点 s 的全体称为 A 在定值点 p 的定值－引 用链（Definition-Use Chaining），简称 DU 链。

（1）如果 B 中 p 后面没有 A 的其它定值点，则 B 中 p 后 面 A 的所有引用点加上 LiveOut[B]中 A 的所有引用点，就是 A 在定值点 p 的 DU 链；  
（2）如果 B 中 p 后面有 A 的其它定值点，则从 p 到与 p 距离最近的那个 A 的定值点之间的 A 的所有引用点， 就是 A 在定值点 p 的 DU 链。



窥孔优化

是指在语句/指令序列上滑动一个包含几条语句/指令的窗口（称为窥孔），发现其 中不够优化的语句/指令序列，用一段更有效的序列来替代它，使整个代码得到改进。

局部优化

构造基本块有向无圈图（DAG，Directed Acyclic Graph）进行局部优化

