

编译原理复习

编译原理复习

Chapter1 编译概述

- 1.1 翻译程序与编译程序
- 1.2 编译过程和编译程序的基本结构
- 1.3 编译程序的生成方法

Chapter2 文法和语言的基本知识

- 2.1 概述
- 2.2 字母表和符号串的基本概念
 - 2.2.1 字母表和符号串
 - 2.2.2 符号串的运算
- 2.3 文法和语言的形式定义
 - 2.3.1 形式语言
 - 2.3.2 文法的形式定义
 - 2.3.3 语言的形式定义
 - 2.3.4 规范推导和规范规约
 - 2.3.5 递归规则与文法的递归性
- 2.4 短语、直接短语和句柄
 - 2.4.1 短语和直接短语
 - 2.4.2 句柄

2.5 语法树和文法的二义性

- 2.5.1 推导和语法树
- 2.5.2 文法的二义性

2.6 文法和语言的分类

2.7 有关文法的实用限制和变换

Chapter3 词法分析与有穷自动机

- 3.1 词法分析程序的功能
- 3.2 单词符号及输出单词的形式
 - 3.2.1 语言的单词符号
 - 3.2.2 词法分析程序输出单词的形式
- 3.3 语言单词符号的两种定义方式
 - 3.3.1 正规式与正规集
 - 3.3.2 正规文法与正规式
- 3.4 正规式与有穷自动机
 - 3.4.1 确定有穷自动机 (DFA)
 - 3.4.2 非确定有穷自动机 (NFA)
 - 3.4.3 由正规式R构造NFA
 - 3.4.4 NFA确定化为DFA的方法
 - 3.4.5 DFA的化简
 - 3.4.6 有穷自动机到正规式的转换
- 3.5 正规文法与有穷自动机
 - 3.5.1 右线性正规文法到有穷自动机的转换方法
 - 3.5.2 左线性正规文法到有穷自动机的转换方法
 - 3.5.3 有穷自动机到正规文法的转换

Chapter4 语法分析

- 4.1 语法分析程序的功能
- 4.2 自上而下分析法
 - 4.2.1 非确定的自上而下分析法的基本思想
 - 4.2.2 文法左递归性的消除
 - 4.2.3 文法回溯的消除

- 4.2.4 LL(1)文法
- 4.2.5 某些非LL(1)文法到LL(1)文法的改写
- 4.2.6 递归下降分析法
- 4.2.7 预测分析法与预测分析表的构造
- 4.3 自下而上分析法的一般原理
- 4.4 算符优先分析法
 - 4.4.1 方法概述
 - 4.4.2 算符优先文法的定义
 - 4.4.3 算符优先关系表的构造
 - 4.4.4 算符优先分析算法的设计
- 4.5 LR分析法
 - 4.5.1 LR分析器的工作原理和过程
 - 4.5.2 LR(0)分析法
 - 4.5.3 SLR(1)分析法
 - 4.5.4 LR(1)分析法

Chapter5 语法制导翻译技术和中间代码生成

- 5.1 概述
- 5.2 属性文法
- 5.3 语法制导翻译概述
- 5.4 中间语言
 - 5.4.1 逆波兰式（后缀式）
 - 5.4.2 三元式
 - 5.4.3 树形表示
 - 5.4.4 四元式
- 5.5 自下而上语法制导翻译

Chapter7 代码优化

- 7.1 优化概述
- 7.2 局部优化
- 7.3 循环优化

Chapter1 编译概述

- 编译程序是系统软件。

1.1 翻译程序与编译程序

- 翻译程序：把源语言所写的源程序翻译成等价的目标语言的目标程序。
- 编译程序：一种翻译程序，它将高级语言所写的源程序翻译成等价的机器语言或汇编语言的目标程序。
- 采用编译方式在计算机上执行用高级语言编写的程序，需分阶段进行
 1. 编译阶段、运行阶段
 2. 编译阶段、汇编阶段、运行阶段

1.2 编译过程和编译程序的基本结构

- 词法分析
 - 对构成源程序的字符串从左到右进行扫描和分解，根据语言的词法规则，识别出一个一个具有独立意义的单词。
 - 词法规则是单词符号的形成规则，它规定了哪样的字符串构成一个单词符号。
- 语法分析

- 在词法分析的基础上，根据语言的语法规则从单词符号串中识别出各种语法单位(如表达式、说明、语句等)，并进行语法检查。
- 语法规则是语法单位的形成规则。
- 语义分析和中间代码生成
 - 对语言的各种语法单位赋予具体的意义。
 - 对每种语法单位进行静态的语义审查，然后分析其含义，并用另一种语言形式(比源语言更接近于目标语言的一种中间代码或直接用目标语言)来描述这种语义。
- 代码优化
 - 对前阶段产生的中间代码进行等价变换或改造，以期获得更为高效的目标代码。
- 目标代码生成
 - 将中间代码转换成特定机器上的绝对指令代码或可重定位的指令代码或汇编指令代码。
- 在编译程序的各个阶段中，都要涉及表格管理和错误处理。

1.3 编译程序的生成方法



Chapter2 文法和语言的基本知识

2.1 概述

- 对程序设计语言的描述是从语法、语义和语用三个因素来考虑。
 - 语法是对语言结构的定义。
 - 语义是描述了语言的含义。
 - 语用则是从使用的角度去描述语言。
- 形式语言理论是编译的重要理论基础。

2.2 字母表和符号串的基本概念

2.2.1 字母表和符号串

- 字母表：元素的非空有穷集合，例如， $\Sigma = \{a, b, c\}$
 - 字母表中至少包含一个元素
 - 任何语言的字母表指出了该语言中允许出现的一切符号
- 符号(字符)：字母表中的元素称为符号或称为字符
- 符号串(字)：符号的有穷序列称为符号串
 - 建立在某个字母表上

- 由字母表上的有穷多个符号组成
- 不包含任何符号的符号串，称为空符号串，用 ϵ 表示。

2.2.2 符号串的运算

- **符号串的连接**: 设 x 和 y 是符号串，则串 xy 称为它们的连接。
 - $\epsilon x = x\epsilon = x$
- **集合的乘积**: 设 A 和 B 是符号串的集合，则 A 和 B 的乘积定义为 $AB = \{xy \mid x \in A, y \in B\}$
 - $\{\epsilon\}A = A\{\epsilon\} = A$
- **符号串的幂运算(符号串连接)**: 设 x 是符号串，则 x 的幂运算定义为

$$\begin{aligned} x^0 &= \epsilon \\ x^1 &= x \\ x^2 &= xx \\ &\dots \\ x^n &= xx\dots x = xx^{n-1} \end{aligned}$$

- **集合的幂运算(集合乘积)**: 设 A 是符号串的集合，则集合 A 的幂运算定义为

$$\begin{aligned} A^0 &= \{\epsilon\} \\ A^1 &= A \\ A^2 &= AA \\ &\dots \\ A^n &= AA\dots A = AA^{n-1} \end{aligned}$$

- **集合 A 的正闭包 A^+ 与闭包 A^*** :

$$\begin{aligned} A^+ &= A^1 \cup A^2 \cup \dots \cup A^n \dots \\ A^* &= A^0 \cup A^1 \cup A^2 \cup \dots \cup A^n \dots = \epsilon \cup A^+ \end{aligned}$$

正闭包表示 A 上元素构成的所有符号串的集合，闭包比正闭包多一个空符号串。

2.3 文法和语言的形式定义

2.3.1 形式语言

- **序列的集合称为形式语言**，每个形式语言都是某个字母表上按某种规则构成的所有符号串的集合。
- 形式语言是指不考虑语言的具体意义，即**不考虑语义**。
- 描述形式语言的两种方法：
 - 当语言为有穷集合时，用**枚举法**来表示语言。
 - 当语言为无穷集合时，用**文法**来描述语言。（递归思想）

2.3.2 文法的形式定义

- **规则(产生式)**: 规则是一个符号与一个符号串的有序对 (A, β) ，通常写作 $A \rightarrow \beta$
 - ' \rightarrow '表示定义为。
 - 一组规则定义了一个语言的语法结构。
 - **非终结符号**: 规则左部，每个非终结符号表示一定符号串的集合。
 - **终结符号**: 组成语言的基本符号，是一个语言的不可再分的基本符号。
- **文法**: 规则的非空有穷集合，通常表示成四元组 $G = \{V_N, V_T, P, S\}$

- 文法是对语言结构的定义和描述
- V_N : 是规则中非终结符号的集合
- V_T : 是规则中终结符号的集合
- P : 是文法规则的集合
- S : 开始符号/识别符号
- 约定:
 - 第一条规则的左部是开始符号
 - 对文法G不用四元式显式表示, 只将规则写出。
- 设计一个文法来描述一个语言, 关键是设计一组规则生成语言中的符号串, 首先分析语言中串的结构特征。
- 描述语言的文法不唯一。

2.3.3 语言的形式定义

- 已知文法, 求得定义的语言。
- **直接推导**: 从符号串 xAy 直接推导出 xay 仅使用一次规则
- 推导和规则的区别:
 1. **形式上的区别**, 推导用“ \Rightarrow ”表示, 规则用“ \rightarrow ”表示。
 2. 对文法G中任何规则 $A \rightarrow a$, 我们有 $A \Rightarrow a$, 即**推导的依据是规则**。
- **推导**: $\alpha_0 \xrightarrow{+} \alpha_n$ (双箭头), 从 α_0 出发, 经**1步或若干步**或者说使用若干次规则可推导出 α_n 。
- **广义推导**: $\alpha_0 \xrightarrow{*} \alpha_n$ (双箭头), 从 α_0 出发, 经**0步或若干步**可推导出 α_n 。
- 直接推导的长度为1, 推导的长度大于等于1, 而广义推导的长度大于等于0。
- **句型和句子**:
 - 如果 $S \xrightarrow{*} x$ (双箭头), $x \in (V_N \cup V_T)^*$, 则称符号串 x 为文法G[S]的**句型**。
 - 如果 $S \xrightarrow{*} x$ (双箭头), $x \in V_T^*$, 则称符号串 x 为文法G[S]的**句子**。
 - 只要证明符号串对文法G存在一个推导, 就可证明符号串是文法G的一个句子。
- **语言**: 文法G[S]产生的**所有句子的集合**称为文法G所定义的语言, 记为 $L(G[S])$.
 - 一旦文法给定, 语言也就确定。
 - $L(G)$ 是 V_T^* 的子集。
- 给定一种语言, 能确定其文法, 但这种文法不是唯一的。
- 给定一个文法, 就能从结构上唯一确定其语言。

2.3.4 规范推导和规范规约

- **最左推导**: 对于一个推导序列中的每一步直接推导 $\alpha \Rightarrow \beta$, 都是对 α 中的最左非终结符进行替换。
- **最右推导(规范推导)**: 对于一个推导序列中的每一步直接推导 $\alpha \Rightarrow \beta$, 都是对 α 中的最右非终结符进行替换。
- **规范句型**: 用规范推导推导出的句型。
- **规约**: 用 $\dot{\Rightarrow}$ 表示归约, 设 $A \rightarrow a$ 是文法G中的一个规则, 则有 $xAy \Rightarrow xay$, $xay \dot{\Rightarrow} xAy$ 。
- **最左归约(规范规约)**: 规范推导的逆过程。

2.3.5 递归规则与文法的递归性

- **递归规则**
 - $A \rightarrow A \dots$ 称为规则左递归
 - $A \rightarrow \dots A$ 称为规则右递归
 - $A \rightarrow \dots A \dots$ 称为规则递归

- 文法的递归性

- 若文法中有推导 $A \xrightarrow{+} A \dots$ 称文法左递归
- 若文法中有推导 $A \xrightarrow{+} \dots A$ 称文法右递归
- 若文法中有推导 $A \xrightarrow{+} \dots A \dots$ 称文法递归
- 在文法中使用递归规则，使得我们能用有限的规则去定义无穷集合的语言。当一个语言是无穷集合时，则定义该语言的文法一定是递归的。

2.4 短语、直接短语和句柄

2.4.1 短语和直接短语

- 短语**: 令G是一个文法，S是文法的开始符号，假定 $\alpha\beta\delta$ 是文法G的一个句型，如果有 $S \xrightarrow{*} \alpha A \delta$ ，且 $A \xrightarrow{+} \beta$ ，则称 β 是相对于非终结符A的句型 $\alpha\beta\delta$ 的短语。
- 直接短语**: 若 $A \Rightarrow \beta$ ，则为直接短语。
- 短语和直接短语的区别：
 - 直接短语中的第二个条件表示有文法规则 $A \rightarrow \beta$ 。
 - 每个直接短语都是某规则右部。

2.4.2 句柄

- 句柄**: 一个句型的最左直接短语。
- 短语、直接短语和句柄都是针对某一句型的，都是指句型中的哪些符号子串能构成短语和直接短语。
- 根据短语定义，可以从句型的推导过程中找出其全部短语、直接短语和句柄。
- 用语法树求句型的短语、直接短语和句柄非常直观、简单。

2.5 语法树和文法的二义性

2.5.1 推导和语法树

- 语法树**: 对句型的推导过程给出一种图形表示。
- 一棵语法树表示了一个句型的种种可能的(但未必是所有的)不同推导过程，包括最左(最右)推导。
- 简单子树：只有单层分枝的子树。
- 句型的短语、直接短语和句柄的直观解释是：
 - 短语**: 子树的末端结点形成的符号串是相对于子树根的短语。
 - 直接短语**: 简单子树的末端结点形成的符号串是相对于简单子树根的直接短语。
 - 句柄**: 最左简单子树的末端结点形成的符号串是句柄。

2.5.2 文法的二义性

- 若一个文法存在某个句子对应两棵不同的语法树，则说这个文法是二义性的。
- 若一个文法中存在某个句子，它有两个不同的最左(最右)推导，则这个文法是二义性的。

2.6 文法和语言的分类

- 0型文法（无限制文法）
 - 文法的每条规则 $\alpha \rightarrow \beta$ 结构如下： $\alpha \in (V_N \cup V_T)^+$, $\beta \in (V_N \cup V_T)^*$, 而且 α 中至少含一个非终结符。
 - 描述的语言是0型语言。
 - 0型语言由图灵机识别。
- 1型文法（上下文有关文法）
 - 文法每条规则的形式为 $\alpha A \beta \rightarrow \alpha \mu \beta$, 其中 $A \in V_N$, $\alpha, \beta \in (V_N \cup V_T)^*$, $\mu \in (V_N \cup V_T)^+$ 。

- 1型语言由线性界限自动机识别。
- 2型文法（上下文无关文法）
 - 每一条规则的形式为 $A \rightarrow \beta$, 其中 $A \in V_N$, $\beta \in (V_N \cup V_T)^*$ 。
 - 2型语言由下推自动机识别。
- 3型文法（正规文法）
 - 右线性文法：每一条规则的形式为 $A \rightarrow aB$ 或 $A \rightarrow a$, 其中 $A, B \in V_N$, $a \in V_T^*$ 。
 - 左线性文法：每一条规则的形式为 $A \rightarrow Ba$ 或 $A \rightarrow a$, 其中 $A, B \in V_N$, $a \in V_T^*$ 。
 - 正规文法产生的语言称为正规语言。
 - 3型语言由有穷自动机识别。

$$L_0 \supset L_1 \supset L_2 \supset L_3$$

2.7 有关文法的实用限制和变换

- 对文法的实用限制有以下两点
 1. 文法中不能含有形如 $A \rightarrow A$ 的规则（有害规则）
 2. 文法中不能有多余规则：
 - 某条规则 $A \rightarrow a$ 的左部符号 A 不在所属文法的任何其他规则右部出现，即在推导文法的所有句子中始终都不可能用到的规则。
 - 对文法中的某个非终结符 A , 无法从它推出任何终结符号串来。

Chapter3 词法分析与有穷自动机

- 有穷自动机是构造词法分析程序的理论基础。

3.1 词法分析程序的功能

- 对字符串表示的源程序从左到右地进行扫描和分解，根据语言的词法规则识别出一个一个具有独立意义的单词符号。

3.2 单词符号及输出单词的形式

3.2.1 语言的单词符号

- 语言的单词符号：语言中具有独立意义的最小语法单位
 - 关键字、标识符、常数、运算符、界符。

3.2.2 词法分析程序输出单词的形式

- 词法分析程序所输出的单词符号通常表示成如下的二元式：（单词种别，单词自身的值）
- 为处理方便通常让每种单词对应一个整数码。
- 关键字：可将其全体视为一种，也可以一字一种。
- 标识符：一般统归为一种。
- 常数：可统归为一种，也可按类型（整型、实型、布尔型等）分种。
- 运算符和界符：可采用一符一种的分法，也可以统归为一种。
- 若一个种别只含一个单词符号，那么种别编码就完全代表了自身的值。

3.3 语言单词符号的两种定义方式

- 多数程序设计语言的单词符号都能用正规文法或正规式来定义。
- 正规式定义简介清晰，正规文法定义易于识别。

3.3.1 正规式与正规集

- 设有字母表 $\Sigma=\{a_1, a_2, \dots, a_n\}$ ，在字母表 Σ 上的正规式和它所表示的正规集可用如下规则来定义：
 - Φ 是 Σ 上的正规式，它所表示的正规集是 Φ ，即空集 $\{\}$ 。
 - ϵ 是 Σ 上的正规式，它所表示的正规集仅含一空符号串，即 $\{\epsilon\}$ 。
 - a_i 是 Σ 上的一个正规式，它所表示的正规集是由单个符号 a_i 所组成，即 $\{a_i\}$ 。
 - 如果 e_1 和 e_2 是 Σ 上的正规式，它们所表示的正规集分别为 $L(e_1)$ 和 $L(e_2)$ ，则：
 - $e_1 \mid e_2$ 是 Σ 上的一个正规式，它所表示的正规集为 $L(e_1 \mid e_2) = L(e_1) \cup L(e_2)$ 。
 - e_1e_2 是 Σ 上的一个正规式，它所表示的正规集为 $L(e_1e_2) = L(e_1)L(e_2)$ 。
 - $(e_1)^*$ 是 Σ 上的一个正规式，它所表示的正规集为 $L((e_1)^*) = (L(e_1))^*$ 。
- 闭包运算的优先级最高，连接运算次之，或运算最低。
- 正规文法对应于正规式，正规集即为语言。
- 如果正规式 R_1 和 R_2 描述的正规集相同，则我们称正规式 R_1 与 R_2 等价，记为 $R_1 = R_2$ 。
- 正规式的性质：
 - $A \mid B = B \mid A$
 - $A \mid (B \mid C) = (A \mid B) \mid C$
 - $A(BC) = (AB)C$
 - $A(B \mid C) = AB \mid AC$
 - $(A \mid B)C = AC \mid BC$
 - $A\epsilon \mid \epsilon A = A$
 - $A^* = AA^* \mid \epsilon = A \mid A^* = (A \mid \epsilon)^*$
 - $(A^*)^* = A^*$

3.3.2 正规文法与正规式

- 正规文法和正规式都是描述正规集(语言)的工具，可相互转化。
- 正规文法到正规式的转换
 - 将正规文法中的每个非终结符表示成关于它的一个正规式方程，获得一个联立方程组。
 - 依照求解规则：
 - 若 $x = ax \mid \beta$ (或 $x = ax + \beta$) 则解为 $x = a^*\beta$
 - 若 $x = xa \mid \beta$ (或 $x = xa + \beta$) 则解为 $x = \beta a^*$
 - 运用正规式的分配律、交换律和结合律，求关于文法开始符号的正规式方程组的解。
- 正规式到正规文法的转化
 - 令 $V_T = \Sigma$
 - 对任何正规式 R 选择一个非终结符 Z 生成规则 $Z \rightarrow R$ 并令 $S = Z$
 - 若 a 和 b 都是正规式，对形如 $A \rightarrow ab$ 的规则转换成 $A \rightarrow aB$ 和 $B \rightarrow b$ 两规则 (右线性)，其中 B 是新增的非终结符
 - 对已转换的文法中，形如 $A \rightarrow a^*b$ 的规则，进一步转换成 $A \rightarrow aA \mid b$ (右线性)。
 - 不断利用前两个进行变换，直到每条规则最多含有一个终结符为止。

- 对于 $A \rightarrow cA^*b$
 - 右线性: $A \rightarrow cB, B \rightarrow aB \mid b$
 - 左线性: $A \rightarrow Bb, B \rightarrow Ba \mid c$
- 用正规文法表示的语言和用正规式表示的集合可以用有穷自动机来识别。

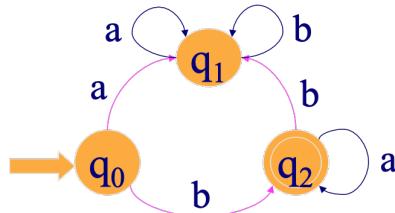
3.4 正规式与有穷自动机

3.4.1 确定有穷自动机 (DFA)

- 一个确定有穷自动机 M 是一个五元组 $M = (Q, \Sigma, f, S, Z)$
 - Q : 有穷状态集合
 - Σ : 有穷输入字母表
 - f : 从 $Q \times \Sigma$ 到 Q 的单值映射, $f(q_i, a) = q_j \quad (q_i, q_j \in Q, a \in \Sigma)$
 - 唯一地确定了下一个要转移的状态
 - $S \subseteq Q$: 唯一的一个初态
 - $Z \subseteq Q$: 终态集
- 状态转换矩阵:

字符 状态	a	b
q_0	q_1	q_2
q_1	q_1	q_1
q_2	q_2	q_1

- 状态转换图:
 - 初始箭头, 终止双圈。



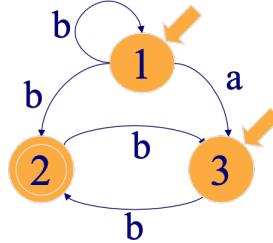
- 对 Σ^* 中的任何符号串 β , 若存在一条从初态结到终态结的道路, 在这条路上所有弧的标记连结成的符号串等于 β , 则称 β 为 DFA M 所识别(或接受)。 M 所识别的符号串的全体记为 $L(M)$, 称为 DFA M 所识别的语言。
- 若 M 的初态结同时又是终态结, 则 ϵ 可为 M 所识别。

3.4.2 非确定有穷自动机 (NFA)

- 一个非确定有穷自动机 M 是一个五元组 $M = (Q, \Sigma, f, S, Z)$
 - f : 多值函数, $f(q_i, a) = \{\text{某些状态的集合}\}$, 是集合
 - 允许 $f(q_i, \epsilon) = \{\text{某些状态的集合}\}$, 即允许边上标记是 ϵ 。
 - $S \subseteq Q$, 是非空初态集。
- 状态转移矩阵:

字符 状态	a	b
1	{3}	{1,2}
2	Φ	{3}
3	Φ	{2}

- 状态转换图:



- 对于NFA，同一个符号串 β 可由多条路来识别。
- 对于每个NFA M 存在DFA M' ，使 $L(M) = L(M')$ 。
- 利用有穷自动机构造词法分析程序的方法：
 - 从语言单词的描述中构造出非确定的有穷自动机，再将非确定的有穷自动机转化为确定的有穷自动机，并将其化简为状态最少化的DFA，然后对DFA的每一个状态构造一小段程序将其转化为识别语言单词的词法分析程序。

3.4.3 由正规式R构造NFA

1. 引进初始结点X和终止结点Y，把R表示成拓广转换图。



2. 分析R的语法结构，用如下规则对R中的每个基本符号构造NFA

(1) $R=\Phi$, 构造NFA如图所示



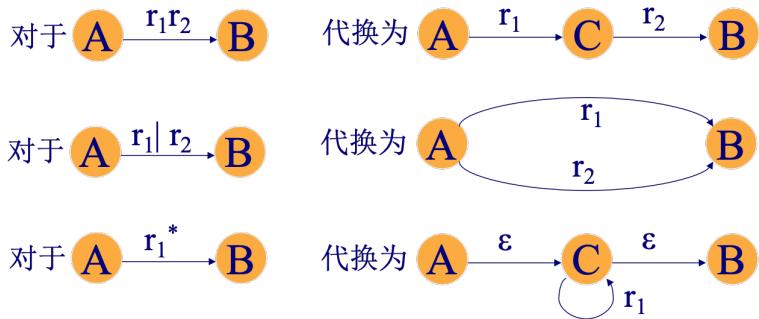
(2) $R= \epsilon$, 构造NFA如图所示



(3) $R= a$ ($a \in \Sigma$), 构造NFA如图所示



(4) 若R是复合正规式，则按下图的转换规则对R进行分裂和加进新结，直至每个边上只留下一个符号或 ϵ 为止。



3. 整个分裂过程中，所有新结均采用不同的名字，保留X, Y为全图唯一初态结和终态结。

试构造正规式 $R = 0(1^*)^* \mid 01$ 的NFA

首先利用正规式的等价化简正规式， $\because (1^*)^* = 1^*$, $\therefore R = 01^* \mid 01 = 0(1^* \mid 1) = 01^*$

3.4.4 NFA确定化为DFA的方法

- 基本思想：
 - DFA的每一个状态代表NFA状态集合的某个子集，这个DFA使用它的状态去记录在NFA读入输入符号之后可能到达的所有状态的集合。
- 利用构造 ϵ -闭包的方法将NFA确定化为DFA
- 状态集合 I 的 ϵ -闭包：
 - ϵ -closure(I) 表示所有那些从 I 中的元素出发经过 ϵ 道路所能到达的NFA的状态所组成的集合， I 中任何状态也在其中，因为它们是通过 ϵ 通路到达自身的。该集合对DFA来说是一个状态。
- 从 NFA N 构造 DFA M 的算法：

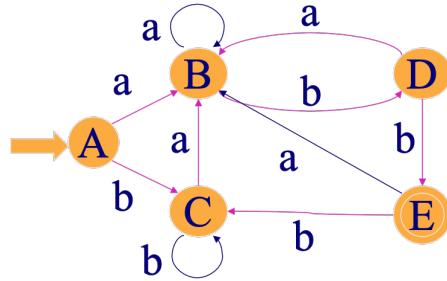
已知 NFA $N = (Q, \Sigma, f, x, \{y\})$, 开始令 $Q' = \{\}$

```

求DFA M的初态  $\epsilon$ -CLOSURE({x}),并置为无标记送入Q';
while ( Q' 中存在一个无标记的状态 T={ s1, s2, s3, ..., sn } )
  { 标记T;
    for ( 每个输入符号a )
      { J = f( {s1, s2, s3, ..., sn }, a )
        = f( s1, a ) ∪ f( s2, a ) ∪ ... ∪ f( sn, a );
        U =  $\epsilon$ -CLOSURE( J );
        if (U不在Q'中 && U不为空) 置U为无标记并送入Q';
        if (U不为空) 置M[T, a] = U;
        if (U中至少有一个是N的终态) U为M的终态;
      }
  }
}
  
```

- 等价的DFA M的状态转移矩阵和状态转换图

Q'	a	b
A { X, 0, 1 }	{ 0, 1, 2 }	{ 0, 1 }
B { 0, 1, 2 }	{ 0, 1, 2 }	{ 0, 1, 3 }
C { 0, 1 }	{ 0, 1, 2 }	{ 0, 1 }
D { 0, 1, 3 }	{ 0, 1, 2 }	{ 0, 1, Y }
E { 0, 1, Y }	{ 0, 1, 2 }	{ 0, 1 }



3.4.5 DFA的化简

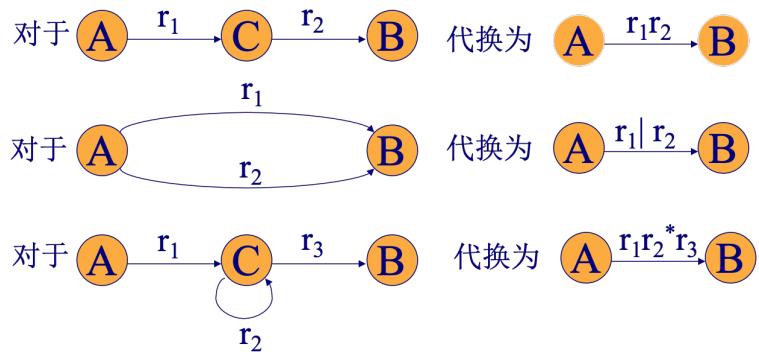
- 任何正规语言都有一个唯一的状态数目最少的DFA。
- 化简了的DFA满足两个条件
 1. 没有多余状态。
 2. 它的状态集中没有两个状态是互相等价的。
- 多余状态：从该自动机的开始状态出发，任何输入串也不能到达的状态。
- 等价状态：
 - 设DFA $M = (Q, \Sigma, f, S_0, F)$, $s, t \in Q$, 若对任何 $a \in \Sigma^*$, $f(s, a) \in F$ 当且仅当 $f(t, a) \in F$, 则称状态 s 和 t 是等价的。
 - 如果 s 和 t 不等价, 则称 s 和 t 是可区别的。
 - 终态与非终态是可区别的。
- 两个状态等价的条件：
 1. 一致性条件：状态 s 和 t 必须同时为终态或非终态。
 2. 蔓延性条件：对于所有输入符号 a , 状态 s 和 t 必须转到等价的状态里。
- 化简方法——删去多余状态，合并等价状态
 1. 判断有无多余状态。
 2. 将DFA M 的状态集 Q 分成两个子集：终态集 F 和非终态集 \bar{F} , 形成初始分划 Π 。（既为开始，又为终止，则视为终止）
 3. 对 Π 使用如下方法建立新分划 Π_{NEW} ：

对 Π 的每个状态子集 G ：

 1. 把 G 分划成新的子集，使得 G 的两个状态 s 和 t 属于同一子集，当且仅当对任何输入符号 a ，状态 s 和 t 转换到的状态都属于 Π 的同一子集。
 2. 用 G 分划出的所有新子集替换 G ，形成新的分划 Π_{NEW} 。
 4. 如果 $\Pi_{NEW} = \Pi$ ，则执行第5步；否则令 $\Pi = \Pi_{NEW}$ ，重复第3步。
 5. 分划结束后，对分划中的每个状态子集，选出一个状态作代表，而删去其它一切等价的状态，并把射向其它状态的箭弧改为射向这个作为代表的状态。

3.4.6 有穷自动机到正规式的转换

- 先尝试化简
 1. 在 M 的转换图上添加两个结点：X结和Y结。从X结用 ϵ 连线连结到 M 的所有初态结点，从 M 的所有终态结点用 ϵ 连线连结到Y结，从而构成一新的非确定有穷自动机 M' ，它只有一个初态结X和一个终态结Y。
 2. 逐步消去 M' 中的其它结点，直至只剩下X, Y结点。在消除结点过程中，逐步用正规式来标记相应的箭弧。



3.5 正规文法与有穷自动机

3.5.1 右线性正规文法到有穷自动机的转换方法

- 设给定了一个右线性正规文法 $G = (V_N, V_T, P, S)$, 则相应的有穷自动机 $M = (Q, \Sigma, f, q_0, Z)$
 - 令 $Q = V_N \cup \{D\}$ ($D \notin V_N$), $Z = \{D\}$ $\Sigma = V_T$ $q_0 = S$
 - 对 G 中每一形如 $A \rightarrow aB$ ($A, B \in V_N, a \in V_T \cup \{\epsilon\}$) 的产生式, 令 $f(A, a) = B$
 - 对 G 中每一形如 $A \rightarrow a$ ($A \in V_N, a \in V_T$) 的产生式, 令 $f(A, a) = D$
 - 对 G 中每一形如 $A \rightarrow \epsilon$ ($A \in V_N$) 的产生式, 令 A 为接受状态或令 $f(A, \epsilon) = D$

3.5.2 左线性正规文法到有穷自动机的转换方法

- 设给定了一个左线性正规文法 $G = (V_N, V_T, P, S)$, 则相应的有穷自动机 $M = (Q, \Sigma, f, q_0, Z)$
 - 令 $Q = V_N \cup \{q_0\}$ ($q_0 \notin V_N$), $Z = \{S\}$ $\Sigma = V_T$
 - 对 G 中每一形如 $A \rightarrow Ba$ ($A, B \in V_N, a \in V_T \cup \{\epsilon\}$) 的产生式, 令 $f(B, a) = A$
 - 对 G 中每一形如 $A \rightarrow a$ ($A \in V_N, a \in V_T \cup \{\epsilon\}$) 的产生式, 令 $f(q_0, a) = A$

3.5.3 有穷自动机到正规文法的转换

- 先尝试化简
- 设给定了一个有穷自动机 $M = (Q, \Sigma, f, q_0, Z)$, 则相应的正规文法 $G = (V_N, V_T, P, S)$
 - 令 $V_N = Q, V_T = \Sigma, S = q_0$
 - 若 $f(A, a) = B$ 且 $B \notin Z$ 时, 则将产生式 $A \rightarrow aB$ 加到 P 中
 - 若 $f(A, a) = B$ 且 $B \in Z$ 时, 则将产生式 $A \rightarrow aB \mid a$ 或将产生式 $A \rightarrow aB, B \rightarrow \epsilon$ 加到 P 中
 - 若文法的开始符号 S 是一个终态, 则将产生式 $S \rightarrow \epsilon$ 加到 P 中

Chapter4 语法分析

4.1 语法分析程序的功能

- | | | |
|--|-----------------------|--|
| <ul style="list-style-type: none"> 词法分析后的单词串 → | 语
法
分
析
器 | <p>→ 语法成分构成的语法树/错误表</p> <ul style="list-style-type: none"> 语法分析的方法: 自上而下语法分析法、自下而上语法分析法 自上而下的分析法: <ul style="list-style-type: none"> 从文法的开始符号出发, 根据文法规则正向推导出给定句子的一种方法 从树根开始, 往下构造语法树, 直到建立每个叶的分析方法 |
|--|-----------------------|--|

- **自下而上的分析法**:

- 从给定的输入串开始，根据文法规则逐步进行归约，直至归约到文法开始符号的一种方法
- 从语法树的末端开始，步步向上归约，直至根结点的分析方法

4.2 自上而下分析法

4.2.1 非确定的自上而下分析法的基本思想

- 对任何输入串W试图用一切可能的办法，从文法的开始符号出发，自上而下地为它建立一棵语法树（使用最左推导）
- 穷举试探过程，试探发生回溯
- 由于对输入串从左向右进行扫描，使用**最左推导**，才能保证按从左到右扫描顺序匹配输入串
- **确定的自上而下分析法**对语言文法的限制条件：
 1. 描述语言的文法**无左递归**
 2. 描述语言的文法**无回溯**

4.2.2 文法左递归性的消除

- 当一个文法是左递归文法时，采用自上而下分析法会使分析过程进入**无穷循环**之中
- 对含直接左递归的规则进行等价变换，消除左递归：
 - 设关于非终结符A的直接左递归的规则为 $A \rightarrow A\alpha \mid \beta$ ，其中 α 、 β 是任意的符号串， α 不等于 ϵ ， β 不以A开头
 - 对A的规则可改写成如下右递归形式： $A \rightarrow \beta A'$, $A' \rightarrow \alpha A' \mid \epsilon$

4.2.3 文法回溯的消除

- 第一种情况：文法中相同左部的规则，其右部左端第一个符号相同而引起回溯
 - $A \rightarrow de \mid d$
 - 可改写
- 第二种情况：文法中相同左部的规则，其中某一右部能推出 ϵ 串
 - $A \rightarrow Bx, B \rightarrow x \mid \epsilon$
 - 不可改写
- 在自上而下分析过程中，为了避免回溯，对描述语言的文法有一定的要求
 - 若用A匹配输入串时，能根据当前读到的输入符号a唯一地选择一条规则去匹配输入串
 - 即要求描述语言的文法是**LL(1)文法**

4.2.4 LL(1)文法

- **FIRST集**: $\text{FIRST}(\alpha) = \{ a \mid \alpha \xrightarrow{*} a \dots \text{且 } a \in V_T \}$
 - 若 $\alpha \xrightarrow{*} \epsilon$, 则规定 $\epsilon \in \text{FIRST}(\alpha)$
- **FOLLOW集**: $\text{FOLLOW}(A) = \{ a \mid S \xrightarrow{*} \dots A a \dots \text{且 } a \in V_T \}$
 - 若有 $S \xrightarrow{*} \dots A$, 则规定 $\$ \in \text{FOLLOW}(A)$
- **SELECT集**: $\text{SELECT}(A \rightarrow \alpha) = \begin{cases} \text{FIRST}(\alpha) & \text{若 } \alpha \xrightarrow{*} \epsilon \\ \text{FIRST}(\alpha) \setminus \{\epsilon\} \cup \text{FOLLOW}(A) & \text{若 } \alpha \xrightarrow{*} \epsilon \end{cases}$
- LL(1)文法判断：
 - 一个上下文无关文法G是LL(1)文法，当且仅当对 G 中每个非终结符A的任何两个不同的规则 $A \rightarrow \alpha \mid \beta$, 满

是 $\text{SELECT}(A \rightarrow \alpha) \cap \text{SELECT}(A \rightarrow \beta) = \emptyset$

- 其中 α 、 β 中至多只有一个能推出 ϵ 串
- 对LL(1)文法，若当前非终结符 A 面临输入符号 a 时，可根据 a 属于哪一个SELECT集，唯一地选择一条相应规则去准确地匹配输入符号 a

4.2.5 某些非LL(1)文法到LL(1)文法的改写

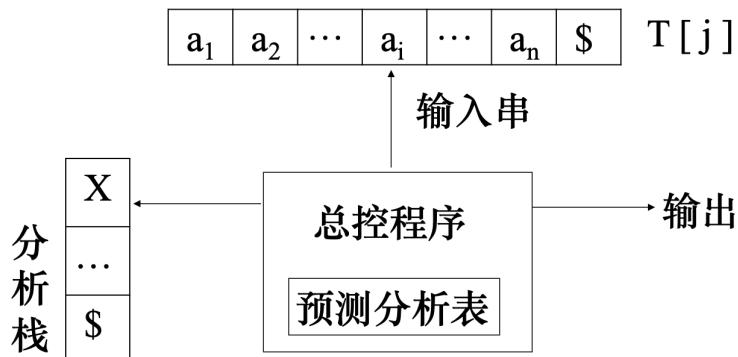
- 构造确定的自上而下分析程序要求对给定语言的文法必须是LL(1)文法
- 对某些非LL(1)文法而言，可通过消除左递归和反复提取公共左因子对文法进行等价变换，可能将其改造为LL(1)文法
- 提取公共左因子：
 - 当文法中含有形如 $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n$ 的规则
 - 将文法改写成： $A \rightarrow \alpha A', A' \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$

4.2.6 递归下降分析法

- 确定的自上而下分析法，要求文法是LL(1)文法
 - 当遇到终结符 a 时，则编写语句：if (当前读来的输入符号== a) 读下一个输入符号；
 - 当遇到非终结符 A 时，则编写语句调用 $A()$ ；
 - 当遇到规则 $A \rightarrow \epsilon$ 时，则编写语句：if (当前读来的输入符号 $\notin \text{FOLLOW}(A)$) error();
 - 当某个非终结符的规则有多个候选式时，按LL(1)文法的条件能唯一地选择一个候选式进行推导
- 构造一个识别某文法句子的递归下降分析程序：
 - 消去文法左递归
 - 提取公共左因子
 - 判断LL(1)
 - 写程序
 - 函数 Scaner()：读进源程序的下一个单词符号，并将它放在全局变量sym
 - 函数 error()：出错处理程序
- 对文法要求高，必须是LL(1)文法，同时由于递归调用较多，影响分析器的效率

4.2.7 预测分析法与预测分析表的构造

- 预测分析法(LL(1)分析法)是确定的自上而下分析的另一种方法
- 由预测分析表、分析栈、总控程序组成



- 预测分析器的总控程序根据栈顶符号和当前输入符号 a 来决定分析器的动作
 - 将规则右部逆序放入 S 栈中，若右部为 ϵ ，则 ϵ 不进 S 栈
 - 总控程序对于不同的LL(1)文法都是相同的，而预测分析表对于不同的LL(1)文法是不相同的
- 构造预测分析表的方法(首先判断是否为LL(1)文法)：

- 计算文法G的每一非终结符的FIRST集和FOLLOW集
- 对文法的每个规则 $A \rightarrow a$, 若 $a \in \text{FIRST}(a)$, 则置 $M[A, a] = A \rightarrow a$
- 若 $\epsilon \in \text{FIRST}(a)$, 对任 $b \in \text{FOLLOW}(A)$, 则置 $M[A, b] = A \rightarrow a$
- 把分析表中每个未定义的元素标上出错标志error(表中用空白格表示)

	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

句子 $id + id * id \$$ 的分析过程

分析栈	输入串
$\$ E$	$id + id * id \$$
$\$ E'T$	$id + id * id \$$
$\$ E'T'F$	$id + id * id \$$
$\$ E'T'id$	$id + id * id \$$
$\$ E'T'$	$+id * id \$$
$\$ E'$	$+id * id \$$
$\$ E'T +$	$+id * id \$$
$\$ E'T$	$id * id \$$
...	...
$\$$	$\$$

- 分析栈从开始符号开始, 以\$结束
- 若一个文法G的分析表M不含多重定义元素, 则该文法是LL(1)文法

4.3 自下而上分析法的一般原理

- 按照移进—归约法的原理建立起来的一种语法分析方法
 - 将输入符号按从左到右扫描顺序移入栈中, 边移入边分析, 当栈顶符号串形成某条规则右部时就进行一次规约
 - 若最后栈中剩下句子右界符"\$"和文法的开始符号, 则所分析的输入符号串是文法的正确句子
- 实现自下而上分析法的关键问题是如何精确定义可归约串这个直观概念, 以及怎样识别可归约串
 - 规范归约分析法——用句柄刻画可归约串
 - LR分析法——根据历史、现实、展望三者信息来确定栈顶符号串是否形成句柄
 - 简单优先分析法——根据文法符号之间的优先关系来确定栈顶符号串是否形成句柄
 - 算符优先分析法——用最左素短语刻画可归约串

4.4 算符优先分析法

4.4.1 方法概述

- 规定运算符之间(确切地说终结符之间)的优先关系和结合性质, 然后借助这种关系, 比较相邻运算符的优先级来确定句型的可归约串并进行归约
- 算符优先分析法的关键: 用合适的方法去定义任何两个可能相邻出现的终结符号a和b之间的优先关系
- 任何两个相邻终结符号a 和 b之间的优先关系有三种

$a < b$	a的优先级低于b
$a = b$	a的优先级等于b
$a > b$	a的优先级高于b

- 优先关系与出现的左右次序有关
- 算符优先分析法借助优先关系表寻找句型的可归约串

	a	b
a	>	<	
b	<		
.....			

4.4.2 算符优先文法的定义

- 算符文法: 设有文法G, 若G中没有形如 $U \rightarrow \dots VV \dots$ 的规则, 其中V和W为非终结符, 则称G为算符文法
 - 任何一个规则右部都不存在两个非终结符相邻的情况
 - 性质
 - 在算符文法中任何句型都不含两个相邻的非终结符
 - 若Ab或bA出现在算符文法的句型 β 中, 其中 $A \in V_N$, $b \in V_T$, 则 β 中任何含b的短语必含有A
- 定义任意两个终结符号之间的优先关系
 - 设G是一个算符文法, a和b是任意两个终结符, P、Q、R是非终结符, 算符优先关系定义如下:
 - $a \sqsubseteq b$ 当且仅当G中含有形如 $P \rightarrow \dots ab \dots$ 或 $P \rightarrow \dots aQb \dots$ 的规则
 - $a \lhd b$ 当且仅当G中含有形如 $P \rightarrow \dots aR \dots$ 的规则, 且 $R \xrightarrow{+} b \dots$ 或 $R \xrightarrow{+} Qb \dots$
 - $a \rhd b$ 当且仅当G中含有形如 $P \rightarrow \dots Rb \dots$ 的规则, 且 $R \xrightarrow{+} \dots a$ 或 $R \xrightarrow{+} \dots aQ$
 - 算符优先文法: 不含 ϵ 规则的算符文法G, 任意两个终结符号对(a,b)在三种关系中只有一种关系成立, 则称G是算符优先文法

4.4.3 算符优先关系表的构造

- 对文法每个非终结符A 定义两个集合:
 - $\text{FIRSTVT}(A) = \{b \mid A \xrightarrow{+} b \dots \text{或 } A \xrightarrow{+} Bb \dots, b \in V_T, B \in V_N\}$
 - $\text{LASTVT}(A) = \{a \mid A \xrightarrow{+} \dots a \text{或 } A \xrightarrow{+} \dots aB, a \in V_T, B \in V_N\}$
- 构造 $\text{FIRSTVT}(A)$ 可遵循以下两条规则
 - 若有产生式 $P \rightarrow a \dots$ 或 $P \rightarrow Qa \dots$, 则 $a \in \text{FIRSTVT}(P)$,
 - 若有产生式 $P \rightarrow Q \dots$ 则 $\text{FIRSTVT}(Q) \subseteq \text{FIRSTVT}(P)$ 。
- 构造 $\text{LASTVT}(A)$ 可遵循以下两条规则

- 若有产生式 $P \rightarrow \dots a$ 或 $P \rightarrow \dots aQ$, 则 $a \in \text{LASTVT}(P)$;
- 若有产生式 $P \rightarrow \dots Q$, 则 $\text{LASTVT}(Q) \subseteq \text{LASTVT}(P)$ 。

- 算法 (扫描文法规则) :

- 寻找 ab 或 aRb , 则有 $a \sqsubseteq b$
- 寻找 aB , 则有 $a \triangleleft \text{FIRSTVT}(B)$
- 寻找 Ab , 则有 $\text{LASTVT}(A) \triangleright b$
- 对于 \$ 的处理
 - 对 $\text{FIRSTVT}(S)$ 中的所有 b , 置 $\$ \triangleleft b$
 - 对 $\text{LASTVT}(S)$ 中的所有 a , 置 $a \triangleright \$$
 - 置 $\$ \sqsubseteq \$$

4.4.4 算符优先分析算法的设计

- 算符优先分析法并不是一种规范归约的分析方法**
- 仅在终结符号之间定义优先关系, 未对非终结符定义优先关系, 从而无法使用优先关系表去识别由单个非终结符组成的可归约串
- 算符优先分析法不用句柄刻画可归约串, 而是用最左素短语刻画可归约串
- 素短语:**
 - 至少包含一个终结符, 并且除自身之外, 不再包含其它的素短语
- 最左素短语:**
 - 句型最左边的素短语

例如, 有文法 $G[E]$:

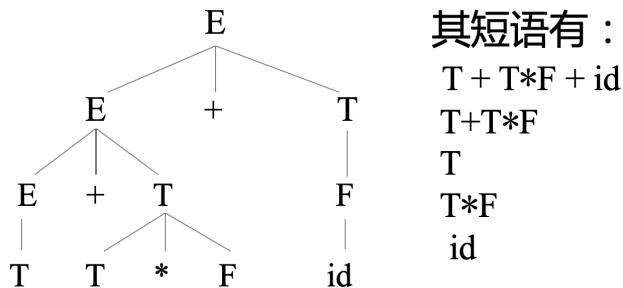
$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid \text{id}$$

求该文法句型 $T + T * F + \text{id}$ 的素短语和最左素短语

首先给出句型 $T + T * F + \text{id}$ 的语法树



由素短语定义可知 $T * F$ 和 id 是素短语, $T * F$ 为最左素短语 (T 是该句型的句柄, 而不是素短语)

- 识别句型最左素短语**

- 任何句型都没有相邻的两个非终结符, $\$ N_1 a_1 N_2 a_2 \dots N_n a_n N_{n+1} \$$
其中每个 N_i 为非终结符或空, a_i 为终结符 ($1 \leq i \leq n$)
- 一个算符优先文法 G 的任何句型的最左素短语是满足下列条件的最左子串 $N_i a_i N_{i+1} a_{i+1} \dots a_j N_{j+1}$:

$a_{i-1} < a_i$

$a_i = a_{i+1} = \dots = a_{j-1} = a_j$

$a_j > a_{j+1}$

对上述句型 $\$ T + T * F + id \$$, 算符优先分析形式为: $\$ N_1 a_1 N_2 a_2 N_3 a_3 a_4 \$$, 因有 $\$ < + < * > + < id > \$$, 得 $T * F$ 是最左素短语

- 最左素短语中的终结符号具有相同的优先关系, 由于最左素短语中的符号是当时最先要归约的串, 其优先关系先于最左素短语之外的符号, 所以使用一个用于存放文法符号的先进后出栈, 并利用优先关系表, 可以确定最左素短语是否已形成来决定分析器的动作
 - 若当前栈顶的终结符号和待输入符号的优先关系是 $<$ 或者 $=$, 则表示栈顶符号串未形成最左素短语, 移进输入符号
 - 若当前栈顶的终结符号和待输入符号的优先关系是 $>$, 则表示已找到最左素短语的尾, 按优先关系在栈内向前寻找最左素短语的头, 然后规约最左素短语
 - 若出现两个终结符号之间不存在优先关系, 则表示存在语法错误, 调用出错处理程序
- 在算法中, 没有指明应将栈顶的最左素短语规约到哪一个非终结符, 因为非终结符在分析过程中对识别最左素短语没有影响, 可取任意名字代替非终结符。

对输入串 $id + id \$$ 的算符优先分析过程

	+	*	id	()	\$
E → E + T T	>	<	<	<	>	>
T → T * F F	>	>	<	<	>	>
F → (E) id	>	>			>	>
(<	<	<	<	=	
)	>	>			>	>
\$	<	<	<	<	=	

S栈	优先关系	当前符号	输入流	动作
\$	<	id	+ id \$	移进
\$ id	>	+	id \$	归约
\$ N	<	+	id \$	移进
\$ N +	<	id	\$	移进
\$ N + id	>	\$		归约
\$ N + N	>	\$		归约
\$ N	=	\$		结束

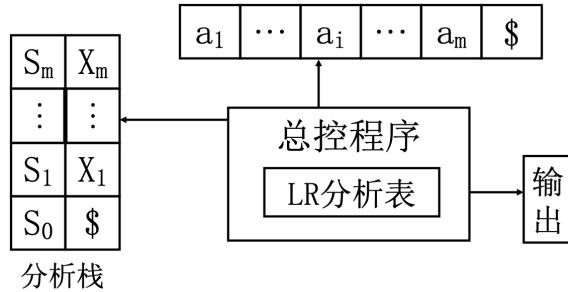
- 算符优先分析法跳过了所有单非产生式(规则右部只含有单个非终结符)之间的归约, 这样算符优先分析比规范归约要快得多, 然而由于忽略非终结符在归约过程中的作用, 可能导致把本来不是句子的输入串误认为是文法句子

4.5 LR分析法

- 对于大多数用无二义性上下文无关文法描述的语言都可以用LR分析法进行有效的分析

4.5.1 LR分析器的工作原理和过程

- 规范归约分析法的关键：在分析过程中如何确定分析栈栈顶的符号串是否形成句柄



- LR分析法将历史和展望信息抽象成状态，并放在分析栈中
- LR分析器的每一步分析工作，都是由栈顶状态和现行输入符号所唯一确定的
- 一张LR分析表由分析动作(ACTION)表和状态转换(GOTO)表组成
 - 状态转换表元素 $GOTO[S_i, X]$ 规定了当状态 S_i 面临文法符号 X 时，应转移到的下一个状态
 - 分析动作表元素 $ACTION[S_i, a]$ 规定了当状态 S_i 面临输入符号 a 时应执行的动作
 - 移进**：把状态 $S_j = ACTION[S_i, a]$ 和输入符号 a 移入分析栈。—Shift(j, a)
 - 归约**：当栈顶符号串 α 形成句柄，且文法中有 $j: A \rightarrow \alpha$ 的规则，其中 $|\alpha| = \beta$ ，则归约动作是从分析栈栈顶去掉 β 个文法符和 β 个状态，并把 A 和 $GOTO[S_{i-\beta}, A] = S_j$ 移入分析栈中。—reduction(j)
 - 接受(acc)**：表示分析成功。此时，分析栈中只剩文法开始符号 S 和当前读到的输入符号是 '\$'。即输入符号串已经结束。
 - 报错**：表示输入串含有错误，此时出现栈顶的某一状态遇到了不该遇到的输入符号。

4.5.2 LR(0)分析法

- 在分析的每一步，只需根据当前栈顶状态而不必向前查看输入符号就能确定应采取的分析动作
- 构造LR分析表的基本思想：
 - 从给定的上下文无关文法直接构造识别文法所有规范句型活前缀的DFA，然后再将DFA转换成一张LR分析表
- 规范句型活前缀**：规范句型的前缀，这种前缀不包含句柄右边的任何符号
- 在LR分析过程中的任何时刻，栈中的文法符号应是某一规范句型的活前缀，由此对句柄的识别就变成对规范句型活前缀的识别
- 活前缀与句柄之间的关系有下述三种情况：
 - 活前缀中已含有句柄的全部符号，此时某一规则 $A \rightarrow \alpha$ 的右部符号串 α 已出现在栈顶，其相应的分析动作是用此规则进行归约

$$A \rightarrow \alpha.$$

- 活前缀中只含有句柄的一部分符号，此时意味着形如 $A \rightarrow \alpha_1 \alpha_2$ 规则的右部子串 α_1 已出现在栈顶， α_2 正期待着从余留的输入串中进行归约得到

$$A \rightarrow \alpha_1 \cdot \alpha_2$$

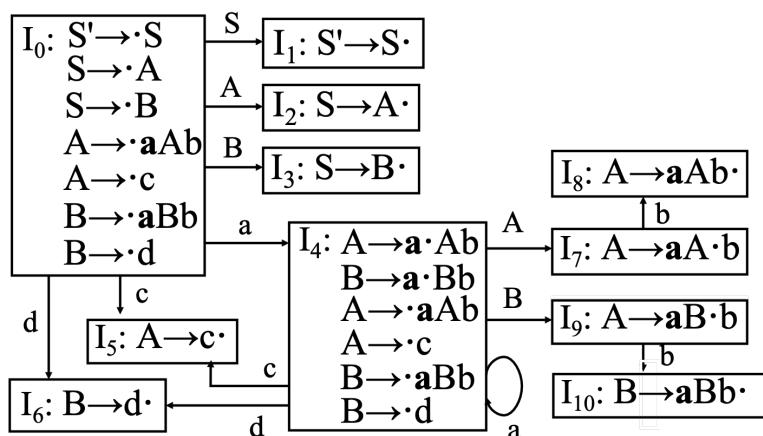
- 活前缀中全然不含有句柄的任何符号，此时意味着期望从余留输入串能看到由某规则 $A \rightarrow \alpha$ 的右部 α 所推出的符号串

$$A \rightarrow \cdot \alpha$$

- 把文法G中右部标有圆点的规则称为G的一个LR(0)项目
- 一个LR(0)项目指明了对文法规范句型活前缀的不同识别状态，文法G的全部LR(0)项目是构造识别文法所有规范句型活前缀的DFA的基础
- 可以根据圆点位置和圆点后是终结符还是非终结符，将一个文法的全部LR(0)项目进行分类：
 1. 归约项目：形如 $A \rightarrow \alpha \cdot$ ，它表示一个规则的右部已分析完，句柄已形成，应该按此规则进行归约
 2. 移进项目：形如 $A \rightarrow \alpha \cdot a\beta$ ，它表示期待从输入串中移进一个符号，以待形成句柄
 3. 待约项目：形如 $A \rightarrow \alpha \cdot B\beta$ ，它表示期待从余留的输入串中进行归约得到B，然后分析A的右部
 4. 接受项目：形如 $S' \rightarrow S \cdot$ ，它表示整个句子已经分析完毕，可以接受
- 构成识别文法规范句型活前缀DFA的每一个状态是由若干个LR(0)项目所组成的集合，称为LR(0)项目集。在这个项目集中，所有的LR(0)项目识别的活前缀是相同的，可以利用闭包函数（CLOSURE）求DFA一个状态的项目集
- 为了使“接受”项目唯一，对文法G进行拓广： $S' \rightarrow S$
- 状态转移函数GO
 - 设I是拓展文法G'的任一个项目集，X为一文法符号
 - $GO(I, X) = CLOSURE(J)$, $J = \{A \rightarrow aX \cdot \beta | A \rightarrow a \cdot X\beta \in I\}$

识别文法G活前缀的DFA

0. $S' \rightarrow S$
1. $S \rightarrow A$
2. $S \rightarrow B$
3. $A \rightarrow aAb$
4. $A \rightarrow c$
5. $B \rightarrow aBb$
6. $B \rightarrow d$

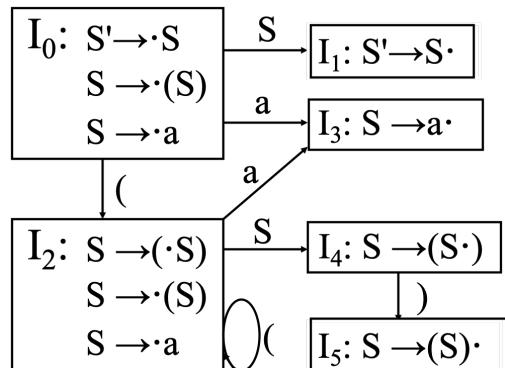


状态	ACTION					GOTO		
	a	b	c	d	\$	S	A	B
0	S_4		S_5	S_6		1	2	3
1					acc			
2	r_1	r_1	r_1	r_1	r_1			
3	r_2	r_2	r_2	r_2	r_2			
4	S_4		S_5	S_6		7	9	
5	r_4	r_4	r_4	r_4	r_4			
6	r_6	r_6	r_6	r_6	r_6			
7		S_8						
8	r_3	r_3	r_3	r_3	r_3			
9		S_{10}						
10	r_5	r_5	r_5	r_5	r_5			

- DFA中的每一个状态都是终态，构成识别一个文法活前缀的DFA的状态(项目集)的全体称为这个文法的LR(0)项目集规范族
- LR(0)文法：文法G的拓广文法G'的LR(0)项目集规范族中的每个项目集不存在移进项目和归约项目同时并存、多个归约项目同时并存
- 构造的LR(0)分析表不含多重定义时，称这样的分析表为LR(0)分析表，能构造LR(0)分析表的文法称为LR(0)文法

考虑文法G[S]: $S \rightarrow (S) \mid a$

首先将文法拓广，并给出每条规则编号：0. $S' \rightarrow S$ 1. $S \rightarrow (S)$ 2. $S \rightarrow a$



	ACTION			GOTO	
	a	()	\$	S
0	S_3	S_2			1
1				acc	
2	S_3	S_2			4
3	r_2	r_2	r_2	r_2	
4				S_5	
5	r_1	r_1	r_1	r_1	

((a))的分析过程

步骤	栈中状态	栈中符号	输入串	分析动作
1	0	\$	((a))\$	移进
2	02	\$((a))\$	移进
3	022	\$((a))\$	移进
4	0223	\$((a))\$	归约
5	0224	\$((S)\$	移进
6	02245	\$((S))\$	归约
7	024	\$(\$)\$	移进
8	0245	\$(\$)	\$	归约
9	01	\$S	\$	接受

- LR(0)分析器的特点：
 - 不需要向前查看输入符号就能归约，即当栈顶形成句柄，不管下一个输入符号是什么，都可以立即进行归约而不会发生错误

4.5.3 SLR(1)分析法

- 为了解决移进—归约或归约—归约冲突，采用对含有冲突的项目集向前查看一个输入符号的办法，这种分析法称为简单的LR分析法，即SLR(1)分析法
- 若一个LR(0)项目集中有m个移进项目和n个归约项目：

$$I : \{A_1 \rightarrow \alpha_1 \cdot a_1 \beta_1, A_2 \rightarrow \alpha_2 \cdot a_2 \beta_2, \dots, A_m \rightarrow \alpha_m \cdot a_m \beta_m, B_1 \rightarrow r_1 \cdot, B_2 \rightarrow r_2 \cdot, \dots, B_n \rightarrow r_n \cdot\}$$
 - 对所有移进项目向前看一符号集合 $\{a_1, a_2, \dots, a_m\}$ 和 $\text{FOLLOW}(B_1), \text{FOLLOW}(B_2), \dots, \text{FOLLOW}(B_n)$ 两两相交为 \emptyset 时，则项目集中冲突可用下述规则解决：
 - 设 a 为当前输入符：
 - 若 $a \in \{a_1, a_2, \dots, a_m\}$ 则移进
 - 若 $a \in \text{FOLLOW}(B_i), i=1, 2, \dots, n$ ，则用 $B_i \rightarrow r_i$ 进行归约
 - 此外报错
- 如果对于一个文法的某些LR(0)项目集或LR(0)分析表中所含有的动作冲突都能用SLR(1)方法解决，则称这个文法是SLR(1)文法
- 若文法的SLR(1)分析表不含多重定义元素，则称文法G为SLR(1)文法
- 仍存在一些文法，其项目集中的移进—归约冲突或归约—归约冲突不能用SLR(1)方法解决
- 用SLR(1)方法解决动作冲突时，仅孤立地考察对于归约项目 $A \rightarrow a \cdot$ ，只要当前面临输入符号 $a \in \text{Follow}(A)$ 时，就确定使用规则 $A \rightarrow a$ 进行归约，而没有考察符号串 a 所在规范句型的环境。因为如果栈里的符号串为 $\$ \delta a$ ，归约后变为 $\$ \delta A$ ，当前读到的输入符号是 a ，若文法中不存在以 $\delta A a$ 为前缀的规范句型，那么，这种归约无效

4.5.4 LR(1)分析法

- LR(1)分析法的思想：
 - 当试图用某一规则 $A \rightarrow a$ 归约栈顶的符号串 a 时，不仅应该察看栈中符号串 δa ，还应向前扫视一个输入符号 a ，只有当 $\delta A a$ 的确构成文法某一规范句型的前缀时，才能用此规则进行归约
- LR(1)项目：
 - 二元组 $[A \rightarrow \alpha \cdot \beta, a]$ ， $A \rightarrow \alpha \cdot \beta$ 是一个LR(0)项目，每个 a 是终结符，称为展望符或搜索符
 - 当 $\beta \neq \epsilon$ 时，搜索符是无意义的
 - 当 $\beta = \epsilon$ 时，搜索符 a 明确指出当 $[A \rightarrow a \cdot \beta, a]$ 是栈顶状态的一个LR(1)项目时，仅在输入符号是 a 时才能

用 $A \rightarrow a$ 归约，而不是对 $\text{FOLLOW}(A)$ 中的所有符号都用 $A \rightarrow a$ 归约

- 构造LR(1)项目集族的方法：
 1. 构造LR(1)项目集I的闭包函数：若项目 $[A \rightarrow a \cdot B\beta, a]$ 属于 $\text{CLOSURE}(I)$, $B \rightarrow r$ 是文法中的一条规则, $b \in \text{FIRST}(\beta a)$, 则 $[B \rightarrow \cdot r, b]$ 也属于 $\text{CLOSURE}(I)$
 2. 构造转换函数：令 I 是一个LR(1)项目集, X 是一个文法符号, 函数 $\text{GO}(I, X) = \text{CLOSURE}(J)$, $J = \{[A \rightarrow aX \cdot \beta, a] \mid [A \rightarrow a \cdot X\beta, a] \in I\}$
- 对LR(1)的归约项目不存在任何无效归约，在多数情况下同一个文法的LR(1)项目集的个数比LR(0)项目集的个数要多
- 如果一个文法的LR(1)分析表不含多重入口时，或者任何一个LR(1)项目集中没有移进—归约冲突或归约—归约冲突，则称该文法为LR(1)文法
- 当一个文法是LR(0)文法，则一定也是SLR(1)文法，也是LR(1)文法，反之不一定成立

Chapter5 语法制导翻译技术和中间代码生成

5.1 概述

- 语义分析的任务：
 - 静态语义审查，审查每个语法结构的静态语义，即验证语法结构合法的程序，是否真正有意义。
 - 例如：表达式， $A + B * C$, 对运算对象进行类型检查，对变量进行先定义后使用检查。
- 执行真正的翻译：如果静态语义正确，语义处理则要执行真正的翻译，即生成程序的某种中间代码的形式或直接生成目标代码。
- 常采用语法制导翻译法，它不是一种形式系统，但它比较接近形式化。语法制导翻译法使用属性文法为工具来描述程序设计语言的语义。

5.2 属性文法

- 属性：对文法的每一个符号，引进一些属性，这些属性代表与文法符号相关的信息，如类型、值、存储位置等。与属性相关的信息，即属性值，可以在语法分析过程中计算和传递。
- 属性加工的过程即是语义的处理过程。
- 属性分为两类：综合属性和继承属性
 - 综合属性：其计算规则按“自下而上”方式进行，即规则左部符号的某些属性根据其右部符号的属性和(或)自己的其他属性计算而得。
 - 继承属性：其计算规则按“自上而下”方式进行，即规则右部符号的某些属性根据其左部符号的属性和(或)右部其他符号的某些属性计算而得。
- 属性文法：属性文法包含一个上下文无关文法和一系列语义规则。
- 语义规则：为文法的每一个规则配备的计算属性的规则（描述语义处理的加工动作）。
 - 这些语义规则附在文法的每个产生式上，在语法分析过程中，执行语义规则描述的动作，从而实现语义处理。

5.3 语法制导翻译概述

- 基本思想：
 - 为文法的每个产生式都配备一个语义动作或语义子程序。
 - 在语法分析的过程中，每当使用一条产生式进行推导或归约时，就执行相应产生式的语义动作，从而实现语义处理。
 - 括号内为语义处理的加工动作。

$$S \rightarrow \dots \quad \{ \dots \}$$

$$\dots \quad \dots$$

$$A \rightarrow xy \quad \{ \dots \}$$

$$\dots \quad \dots$$

- 语法制导翻译法使用**属性文法**为工具来说明程序设计语言的语义。

- 语法制导翻译法**: 在语法分析过程中, 依随分析的过程, 根据每个产生式所对应的语义子程序(或语义规则描述的语义处理的加工动作)进行翻译的方法。

设有简单算术表达式的文法 $E \rightarrow E + E | E * E | (E) | digit$, 为文法每一产生式设计相应的求值的语义描述(语义动作):

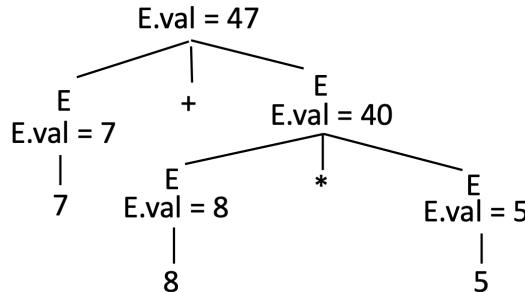
$$1. E \rightarrow E^{(1)} + E^{(2)} \quad \{ E.val = E^{(1)}.val + E^{(2)}.val \}$$

$$2. E \rightarrow E^{(1)} * E^{(2)} \quad \{ E.val = E^{(1)}.val * E^{(2)}.val \}$$

$$3. E \rightarrow (E^{(1)}) \quad \{ E.val = E^{(1)}.val \}$$

$$4. E \rightarrow digit \quad \{ E.val = Lex.digit \}$$

句子 $7+8*5$



5.4 中间语言

5.4.1 逆波兰式 (后缀式)

- 逆波兰式**去除了原表达式中的括号, 并将运算对象写在前面, 运算符写在后面。

中缀表达式 逆波兰式

a*b ab*

(a+b)*(c+d) ab+cd+*

- 优点

- 不再有括号, 且运算符出现的顺序体现了中缀表达式的运算顺序。

- 易于计算机处理。

- 一般表达式计值时, 要处理两类符号, 一类是运算对象, 另一类是运算符, 通常用两个工作栈分别处理, 但处理用逆波兰式表示的表达式却只用一个工作栈。
- 自左到右顺序扫描逆波兰式, 若当前符号是运算对象则进栈, 若当前符号是运算符, 设为K元运算符, 则将栈顶的K个元素依次取出, 同时进行K元运算, 并将运算结果置于栈顶, 表达式处理完毕时, 其计算结果自然呈现在栈顶。
- 逆波兰形式可以推广到其他语法结构:

赋值语句 逆波兰式

V=E VE=

条件语句 逆波兰式

if E S₁; else S₂ ES₁S₂ ¥

-a + b * (-c + d) 的逆波兰式: a@ bc @ d + * +

i↑(i / (i - i)) 的逆波兰式: i i i - / ↑

5.4.2 三元式

- **三元式**主要由三部分组成: (OP, arg1, arg2)

- OP: 运算符
- arg1, arg2: 第一和第二个运算对象
- 当OP是一目运算时, 常常将运算对象定义为arg1
- 运算对象是指向符号表的某一项或指向三元式表的某一项

a+b*c 的 三元式序列: (1) (*, b, c) (2) (+, a, (1))

- 三元式的特点:

- 三元式出现的顺序和语法成分的计值顺序相一致。
- 三元式之间的联系是通过指示器实现的。

- **间接三元式**:

- 三元式表: 用来存放各三元式本身
- 间接码表: 按执行各三元式的顺序, 依次列出各三元式在三元式表中的位置
- 间接三元式表中**不存放重复的三元式**

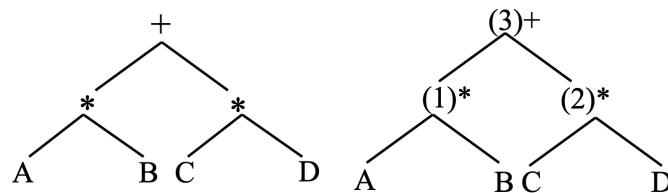
语句: X = (A + B) * C, Y = D ↑ (A + B)

三元式序列	间接三元式	三元式表
(1) (+, A, B)	间接码表	三元式表
(2) (*, (1), C)	(1)	(1) (+, A, B)
(3) (=, X, (2))	(2)	(2) (*, (1), C)
(4) (+, A, B)	(3)	(3) (=, X, (2))
(5) (↑, D, (4))	(1)	(4) (↑, D, (1))
(6) (=, Y, (5))	(4)	(5) (=, Y, (4))

5.4.3 树形表示

- **树形表示**: 三元式的翻版

语句: A * B + C * D



- 末端结点表示一个运算对象, 每一个内结点表示一个一元或二元运算符。

5.4.4 四元式

- 四元式主要由四部分组成：(OP, arg1, arg2, result)
 - OP: 运算符
 - arg1, arg2: 第一和第二个运算对象
 - 当OP是一目运算时，常常将运算对象定义为arg1
 - result是编译程序为存放中间运算结果而临时引进的变量，也可以是用户自定义的变量

语句: $X = a * b + c / d$

- (1) (*, a, b, T₁)
- (2) (/, c, d, T₂)
- (3) (+, T₁, T₂, T₃) 不能写为 (3) (+, T₁, T₂, X)
- (4) (=, T₃, -, X)

- 四元式的特点：

- 四元式出现的顺序和语法成分的计值顺序相一致。
 - 四元式之间的联系是通过临时变量实现的，这样易于调整和变动四元式。
 - 便于优化处理。
- 四元式更直观的形式：三地址代码
 - 三地址代码: result := arg1 OP arg2
 - 三地址语句：语句中是三个量的赋值语句，每个量占一个地址

语句: $X = a * b + c / d$

- (1) (*, a, b, T₁)
- (2) (/, c, d, T₂)
- (3) (+, T₁, T₂, T₃)
- (4) (=, T₃, -, X)

相应的三地址语句序列为：

- (1) T₁=a*b
- (2) T₂=c/d
- (3) T₃=T₁+T₂
- (4) X=T₃

-a + b * (-c + d) 的三地址语句：

- (1) t₁=@ a
- (2) t₂=@ c
- (3) t₃=t₂ + d
- (4) t₄=b*t₃

(5) $t_5 = t_1 + t_4$

$i \uparrow (i / (i - i))$ 的三地址语句：

(1) $t_1 = i - i$

(2) $t_2 = i / t_1$

(3) $t_3 = i \uparrow t_2$

5.5 自下而上语法制导翻译

- 设计：

- 分析源结构和目标结构
- 自下而上的语法制导翻译特点：栈顶形成句柄，归约时执行相应语义动作
- 给出从源结构到目标结构的变换方法

设文法及其相应的语义动作如下：

$S \rightarrow bTc \quad \{ \text{print "1"} \}$

$S \rightarrow a \quad \{ \text{print "2"} \}$

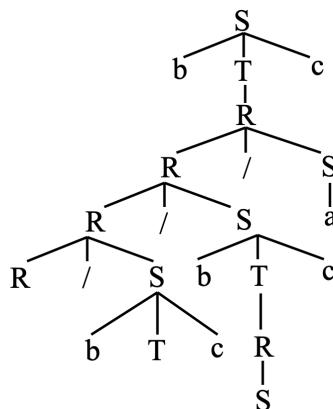
$T \rightarrow R \quad \{ \text{print "3"} \}$

$R \rightarrow R/S \quad \{ \text{print "4"} \}$

$R \rightarrow S \quad \{ \text{print "5"} \}$

采用自下而上语法制导翻译，给出输入串 $bR / bTc / bSc / ac$ 的翻译结果

首先对输入串 $bR / bTc / bSc / ac$ 画出语法树：



再考虑归约时执行相应语义动作

依次寻找句柄，规约的同时执行相应规则的语义动作，翻译结果为 1453142431

改：输入是 $bR / bTc / bSc / ac$ ，输出为 1453142431，给出相应语义动作(翻译方案)

简单算术表达式求值的语义描述，设有简单算术表达式的文法 $E \rightarrow E + E | E * E | (E) | digit$

1. $E \rightarrow E^{(1)} + E^{(2)} \quad \{ E.val = E^{(1)}.val + E^{(2)}.val \}$

2. $E \rightarrow E^{(1)} * E^{(2)} \quad \{ E.val = E^{(1)}.val * E^{(2)}.val \}$

$3. E \rightarrow (E^{(1)})$	$\{E.val = E^{(1)}.val\}$
$4. E \rightarrow digit$	$\{E.val = Lex.digit\}$

简单算术表达式翻译到四元式的语义描述，设有简单算术表达式的文法 $E \rightarrow E + E | E * E | (E) | i$

源结构 目标结构

$$\begin{aligned} a*b+c*d & \quad (1) \quad T_1 = a*b \\ & \quad (2) \quad T_2 = c*d \\ & \quad (3) \quad T_3 = T_1 + T_2 \end{aligned}$$

语义函数 $emit(T = arg_1 OP arg_2)$ ，功能是生成一个三地址语句，并送到输出文件中

语义函数 $newtemp()$ ，功能是产生一个新的临时变量名字，并回送新的临时变量名的整数码，如 T_1, T_2 等
对临时变量有两种不同的处理方法：

1. 送到符号表
2. 不进符号表，临时变量单词值部分用整数码表示

语义过程 $Lookup(i.name)$ ，功能是审查 $i.name$ 是否出现在符号表中，在则返回其指针，否则返回NULL

语义变量 $E.place$ ：表示存放非终结符 E 值的变量名在符号表中的入口地址或临时变量名的整数码

- 1. $E \rightarrow E^{(1)} + E^{(2)}$**
 $\{ E.place = newtemp();$
 $emit(E.place' = 'E^{(1)}.place' + 'E^{(2)}.place)$
 $\}$
- 2. $E \rightarrow E^{(1)} * E^{(2)}$**
 $\{ E.place = newtemp();$
 $emit(E.place' = 'E^{(1)}.place' * 'E^{(2)}.place)$
 $\}$
- 3. $E \rightarrow (E^{(1)})$**
 $\{ E.place = E^{(1)}.place; \}$
- 4. $E \rightarrow i$**
 $\{ p = Lookup(i.name);$
 $if (p != NULL) E.place = p;$
 $else error();$
 $\}$

算术表达式 $a+b*c$ 翻译到三地址语句的过程：

为文法构造 LR 分析表如下图：

状态	ACTION						GOTO
	+	i	*	()	\$	
0		S_3		S_2			1
1	S_4	S_3	S_5	S_2		acc	
2		S_3		S_2			6
3	r_4		r_4	S_2	r_4	r_4	
4		S_3		S_2			7
5		S_3		S_2			8
6	S_4		S_5		S_9		
7	r_1		S_5		r_1	r_1	
8	r_2		r_2		r_2	r_2	
9	r_3		r_3		r_3	r_3	

状态栈	符号栈	语义栈	输入串
0	\$	—	a+b*c\$
03	\$a	— —	+b*c\$
01	\$E	— a	+b*c\$
014	\$E+	— a —	b*c\$
0143	\$E+b	— a — —	*c\$
0147	\$E+E	— a — b	*c\$
01475	\$E+E*	— a — b —	c\$
014753	\$E+E*c	— a — b — —	\$
014758	\$E+E*E	— a — b — c	\$ $t_1=b*c$
0147	\$E+E	— a — t_1	\$ $t_2=a+t_1$
01	\$E	— t_2	\$ acc

文法G[S]如下： $S \rightarrow S(S)$, $S \rightarrow \epsilon$, 试写出一个语法制导定义, 它对输入符号串翻译的结果是输出配对的括号个数

为S引入一个综合属性num, 用以表示该文法符号所表示的语法成分中配对的括号数。首先将文法扩展, 并设计语法制导的语义动作为:

$S' \rightarrow S \quad \{ \text{print}(S.\text{num}) \}$

$S \rightarrow S1(S2) \quad \{ S.\text{num}=S1.\text{num}+S2.\text{num}+1 \}$

$S \rightarrow \epsilon \quad \{ S.\text{num}=0 \}$

有一语法制导翻译方法如下: 其中“+”表示符号连接运算:

$S \rightarrow B \quad \{ \text{print}(B.\text{reverse}) \}$

$B \rightarrow a \quad \{ B.\text{reverse}=a \}$

$B \rightarrow b \quad \{ B.\text{reverse}=b \}$

$B \rightarrow Ba \quad \{ B.\text{reverse}=a+B.\text{reverse} \}$

$B \rightarrow Bb \quad \{ B.\text{reverse}=b+B.\text{reverse} \}$

若输入符号串abab, 则采用自底向上语法分析方法时, 输出为baba

Chapter7 代码优化

7.1 优化概述

- 代码优化：
 - 对程序实施各种等价变换，使得变换后程序能生成高效率的目标代码
 - 目标代码占用的存储空间少
 - 目标代码运行时间短
- 代码优化的种类
 - 根据是否涉及具体的计算机分为：
 1. 与机器有关的优化（主要在目标代码级上进行）
 - 对寄存器的优化
 - 多处理器的优化
 - 特殊指令的优化
 2. 与机器无关的优化（主要在中间代码级上进行）
 - 根据优化对象所涉及的程序范围分为：
 1. 局部优化
 2. 循环优化
 3. 全局优化
- 局部优化：局限于程序基本块范围内的一种优化
 - 合并已知量
 - 删除公共子表达式(删除多余的运算)
 - 删除无用赋值
- 循环优化：对循环中的代码进行优化
 - 代码外提
 - 删除归纳变量
 - 强度削弱
- 全局优化：在整个程序范围内进行的优化，需进行数据流分析，花费代价很高

例如 设有一个计算两个向量内积的源程序段

```
s = 0;
for ( i=1; i<=20; i++ )
    s = s + A[i]*B[i];
```

编译程序对其进行词法分析、语法分析、语义分析和中间代码的生成，得到如下一组三地址语句序列表示的中间代码：

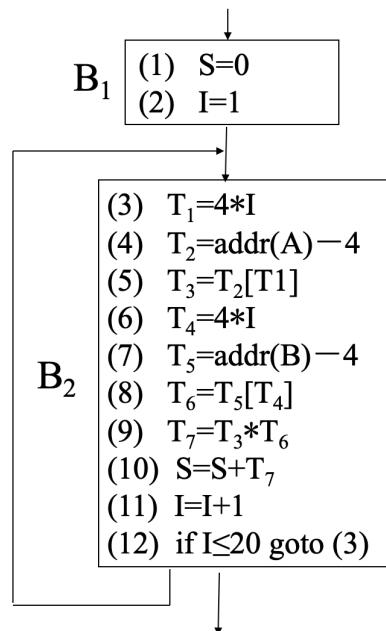
$\langle A[i] \rangle = \langle A[1] \rangle + l * \text{sizeof}(\text{数组元素}) - 1 = \text{addr}(A) - 1 + l * 4$

地址计算的不变部分： $\text{addr}(A) - 1 \Rightarrow T_2$

地址计算的可变部分： $l * 4 \Rightarrow T_1$

用 $T_2[T_1]$ 表示数组元素的地址

若一个机器字有四个字节，按字节编址： $T_1 = 4 * l$, $T_2 = \text{addr}(A) - 4 * 1$

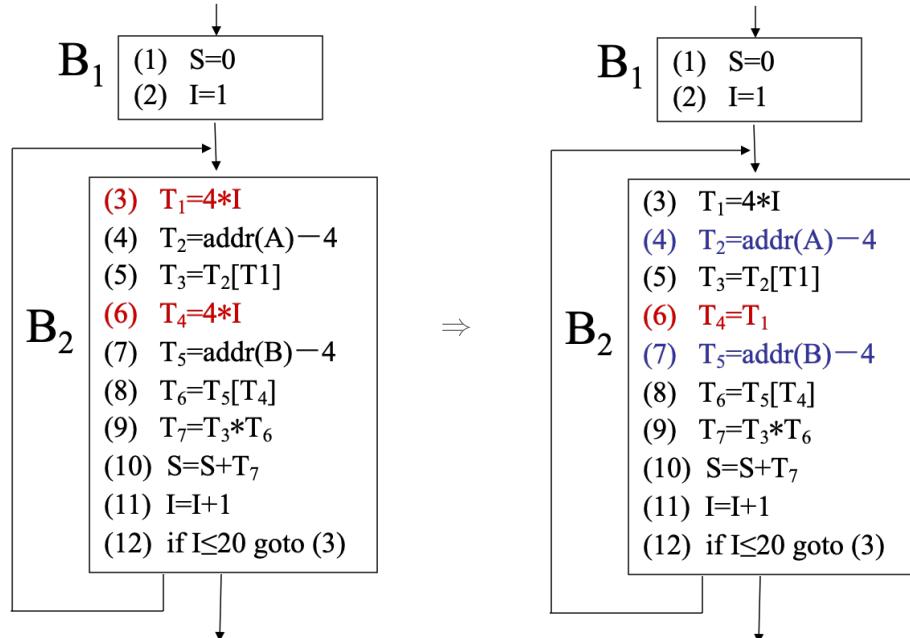


- 在上图所示的中间代码中，根据程序流向的特点，分为B₁、B₂两块：

- B₁是循环体外的语句序列
- B₂是可重复执行的循环体语句序列

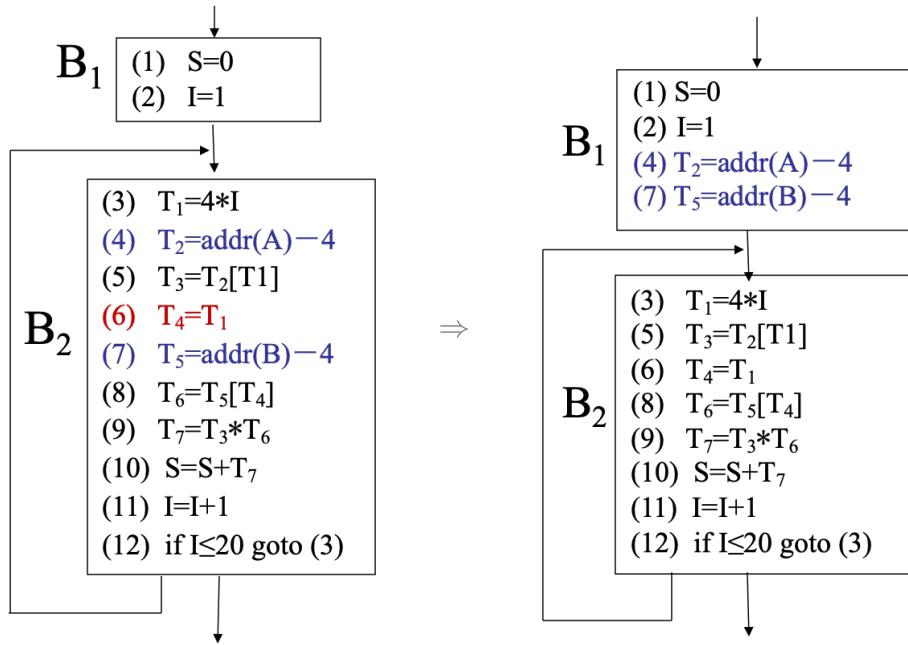
- 删除公共子表达(删除多余运算)

- 公共子表达式：在一个基本块内计算结果相同的子表达式
- 对相同的子表达式只在第一次出现时计算且仅计算一次，将重复计算的表达式删除



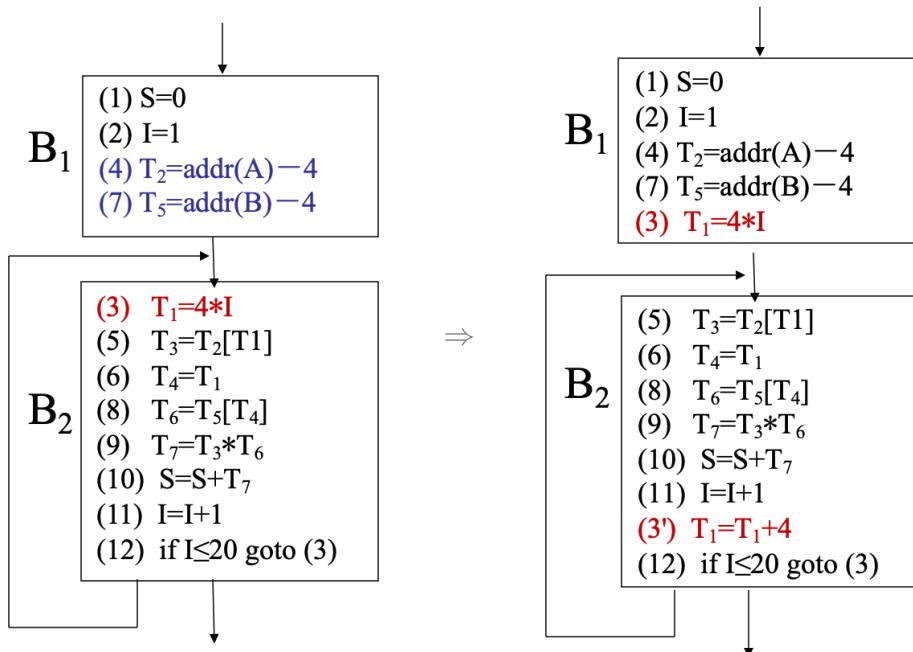
- 代码外提

- 将循环中的不变运算提到循环体前面



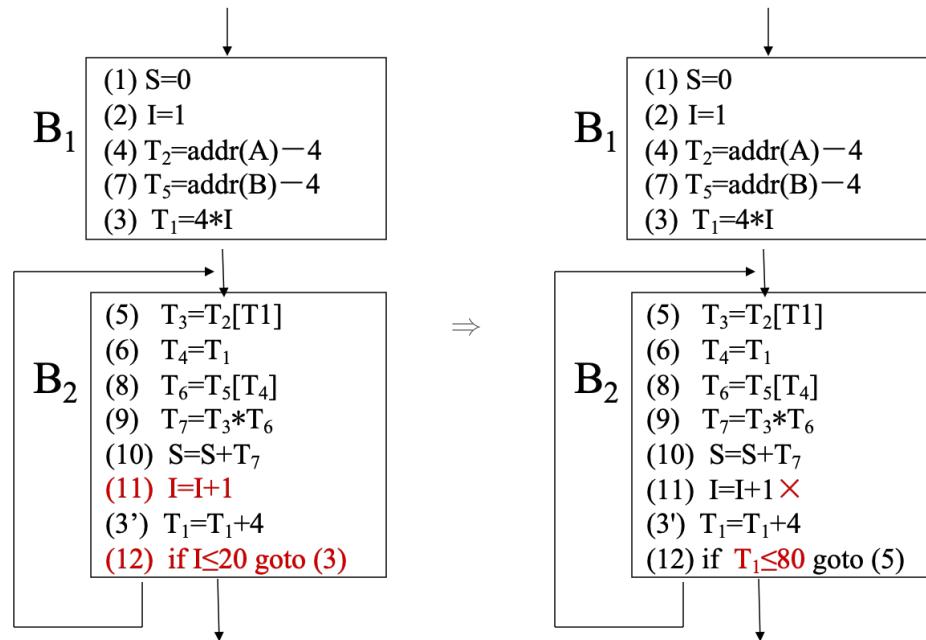
- 强度削弱

- 在不改变运算结果的前提下，将程序中执行时间长的运算替换成执行时间短的运算
- 对计算机上的二进制算术运算，作加法一般比作乘法的速度快



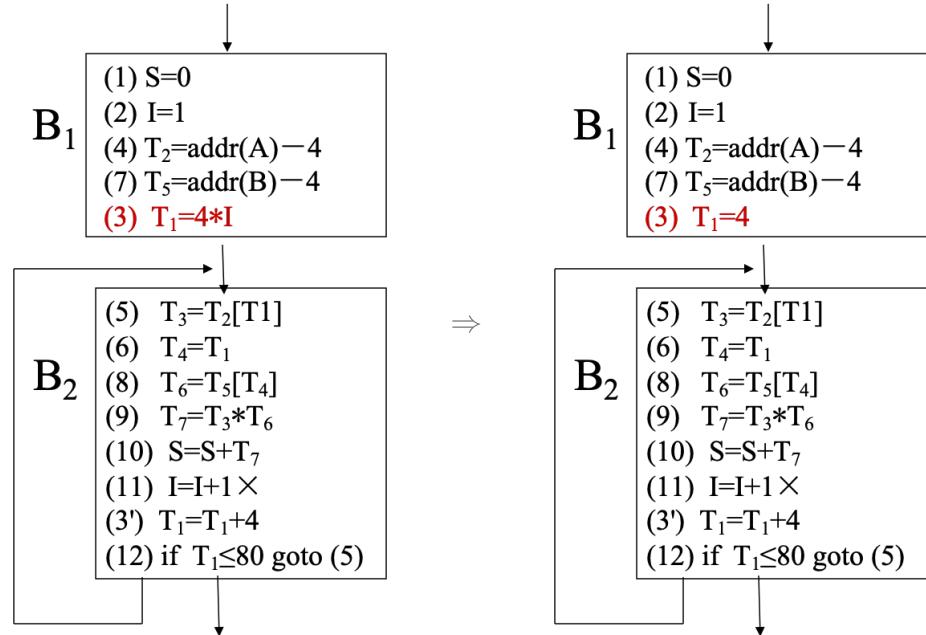
- 变换循环控制条件(删除归纳变量)

- 在B₂中存在变量 T₁ 与循环控制变量 I 保持线性关系，则可由 T₁ 取代 I
- 变量 I 是引用 I 值来计算 I 值，称为对 I 递归赋值，I 称为循环中的归纳变量



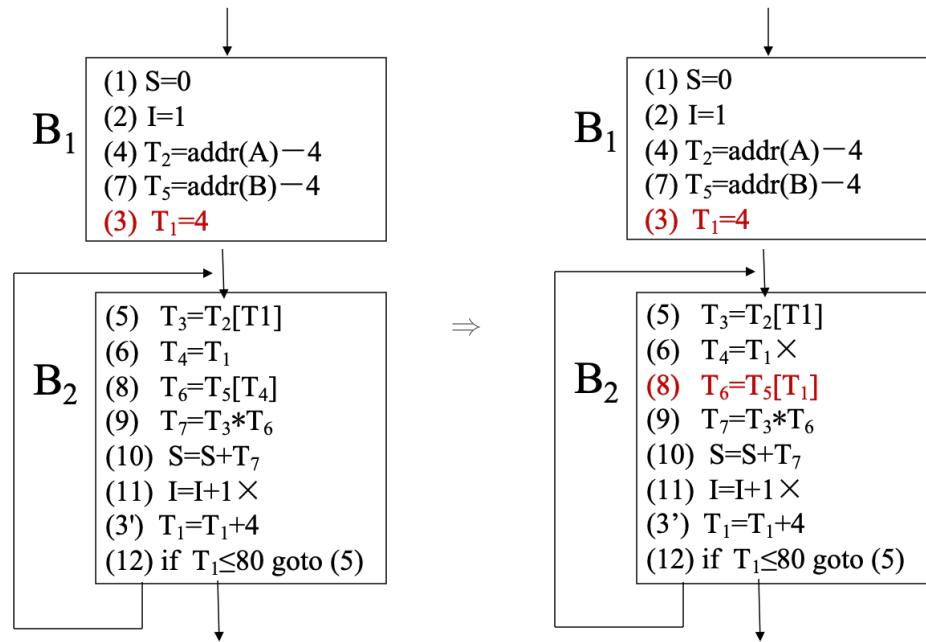
- 合并已知量

- 已知量是指常数或在编译时就能确定其值的变量
- 合并已知量是指若参加运算的两个对象在编译时都是已知量，则可以在编译时直接计算出它们的运算结果，不必生成目标



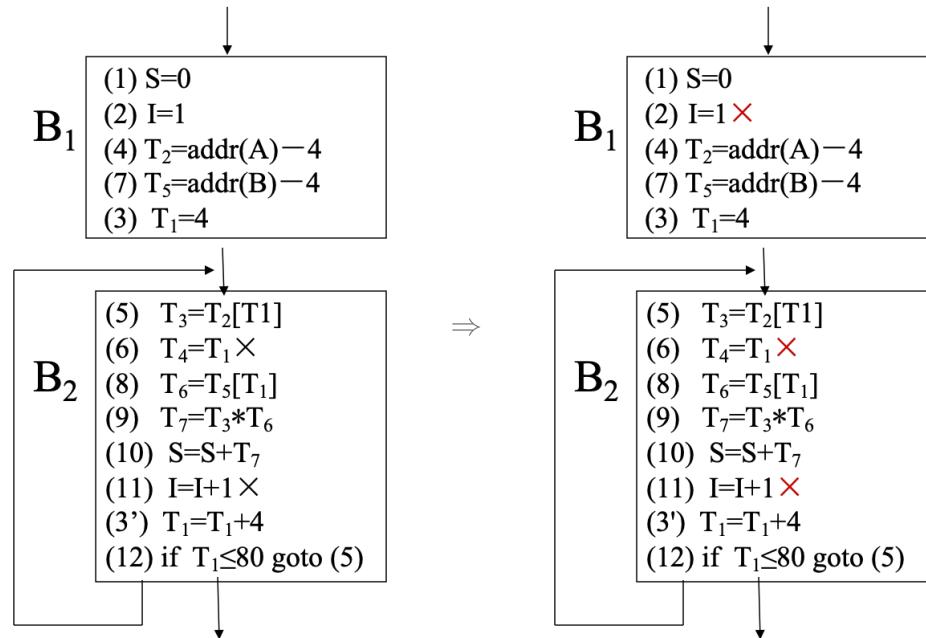
- 复写传播

- 尽量不引用那些在程序中仅仅只传递信息而不改变其值，也不影响其运行结果的变量

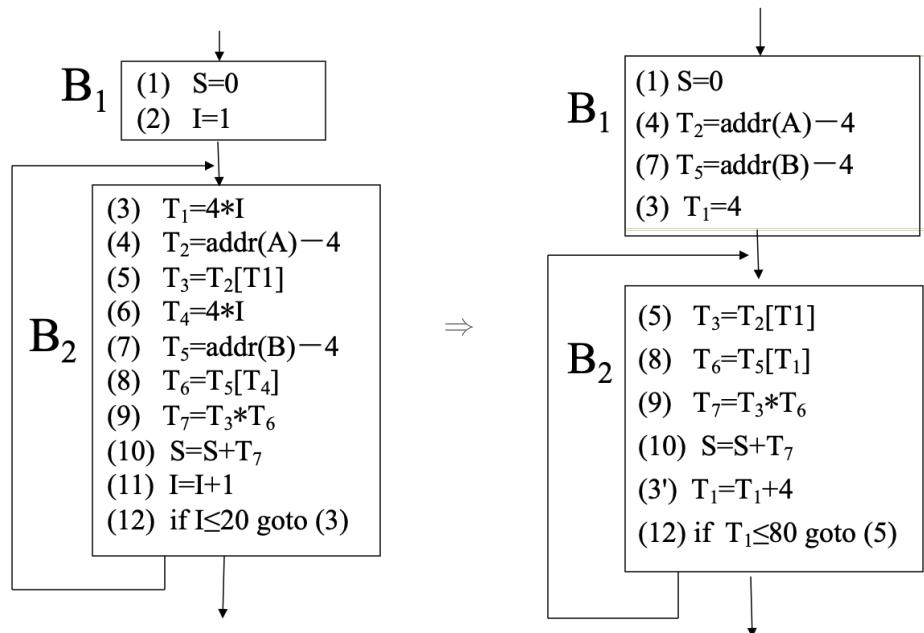


- 删除无用赋值

- 对赋值语句X=Y, 若在程序的任何地方都不引用X, 这时该语句执行与否对程序运行结果没有任何作用, 这种语句称为无用赋值语句, 可以删除



- 最终结果



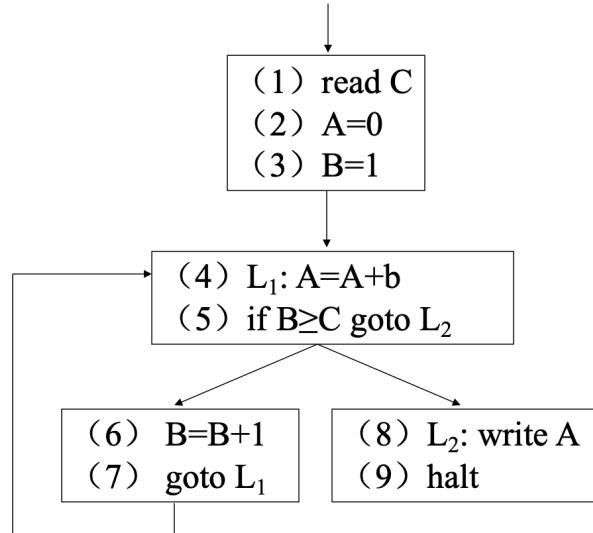
- 比较优化前后的代码序列可以发现经过优化后最终得到的代码序列具有以下特点：
 - 减少了循环体内可执行代码，由10条减至6条
 - 减少了作乘法运算的次数，由3次降为1次
 - 减少了全程范围内使用的变量，如 I 、 T_4 等变量

7.2 局部优化

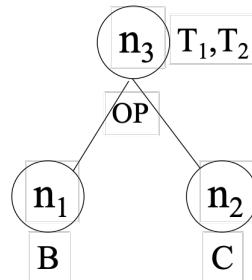
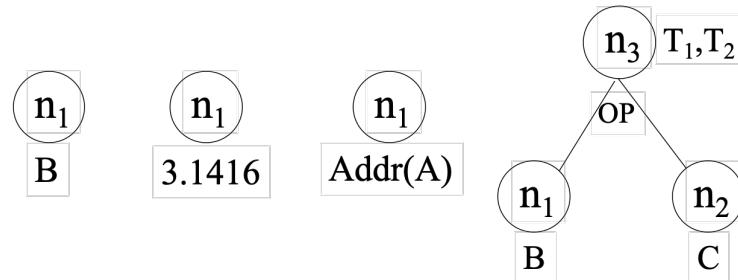
- 局部优化是指局限于程序基本块范围内的一种优化
- 基本块：**
 - 所谓基本块是指程序中一组顺序执行的语句序列，其中只有一个入口语句和一个出口语句
- 划分基本块的方法
 - 从四元式序列确定基本块的入口语句：
 - 四元式序列的第一个语句
 - 由条件转移语句或无条件转移语句转移到的语句
 - 紧跟在条件转移语句后面的语句
 - 确定基本块的出口语句：
 - 下一个入口语句的前导语句
 - 转移语句(包括转移语句本身)
 - 停语句(包括停语句本身)
- 入口语句和出口语句之间组成一个基本块
- 删去那些不属于任何基本块的语句，因为控制流无法到达这些语句

例 给以下四元式序列划分基本块

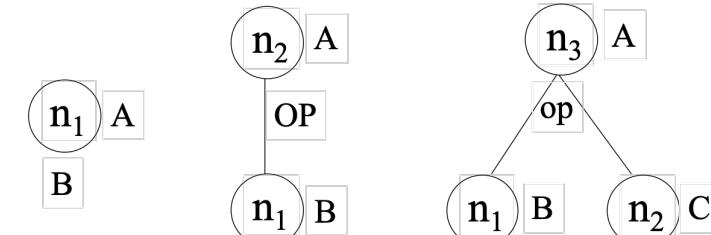
→(1) read C	四元式序列的第一个语句
→(2) A=0	
→(3) B=1	下一个入口语句的前导语句
→(4) L ₁ : A=A+b	由条件转移语句或无条件转移语句转移到的语句
→(5) if B≥C goto L ₂	转移语句(包括转移语句本身)
→(6) B=B+1	紧跟在条件转移语句后面的语句
→(7) goto L ₁	转移语句(包括转移语句本身)
→(8) L ₂ : write A	由条件转移语句或无条件转移语句转移到的语句
→(9) halt	停语句(包括停语句本身)



- 利用DAG(无环路有向图)实现局部优化的思想：基本块→DAG→还原
- 结点带有标记的DAG：



- 四元式与DAG：



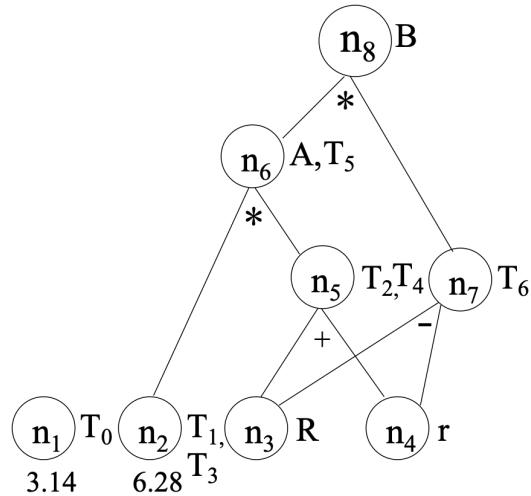
(1) A=B

(2) A= op B

(3) A=B op C

构造以下基本块的DAG：

- (1) $T_0 = 3.14$
- (2) $T_1 = 2 * T_0$
- (3) $T_2 = R + r$
- (4) $A = T_1 * T_2$
- (5) $B = A$
- (6) $T_3 = 2 * T_0$
- (7) $T_4 = R + r$
- (8) $T_5 = T_3 * T_4$
- (9) $T_6 = R - r$
- (10) $B = T_5 * T_6$



按照构造DAG结点的顺序，对每一个结点写出其相应的四元式表示

- | | |
|-----------------------|----------------------|
| (1) $T_0 = 3.14$ | (1) $T_0 = 3.14$ |
| (2) $T_1 = 2 * T_0$ | (2) $T_1 = 6.28$ |
| (3) $T_2 = R + r$ | (3) $T_3 = 6.28$ |
| (4) $A = T_1 * T_2$ | (4) $T_2 = R + r$ |
| (5) $B = A$ | (5) $T_4 = T_2$ |
| (6) $T_3 = 2 * T_0$ | (6) $A = 6.28 * T_2$ |
| (7) $T_4 = R + r$ | (7) $T_5 = A$ |
| (8) $T_5 = T_3 * T_4$ | (8) $T_6 = R - r$ |
| (9) $T_6 = R - r$ | (9) $B = A * T_6$ |
| (10) $B = T_5 * T_6$ | |
- ⇒

设优化后的上述一组四元式序列为 G' , G' 较之优化前的一组四元式 G , 不但代码总数减少, 而且对 G 中四元式 (2) $T_1 = 2 * T_0$ 和 (6) $T_3 = 2 * T_0$, 在 G' 中已被优化为 $T_1 = 6.28$, $T_3 = 6.28$, 这是**合并已知量**的结果; 对 G 中四元式 (5) $B = A$, 直至在下一个 B 被赋值时都没有被引用, 显然是**无用赋值**, G' 中已被删除; 对 G 中具有公共子表达式的四元式 (3) $T_2 = R + r$, (7) $T_4 = R + r$, G' 中被优化为直接赋值 $T_4 = T_2$, 避免了对**公共子表达式** $R + r$ 的重复计算。

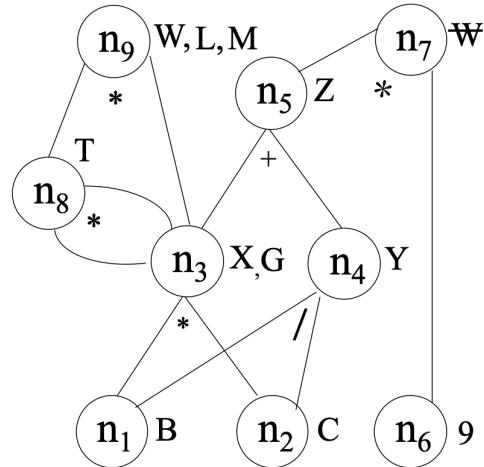
综上所述, 利用DAG可以很方便地进行基本块内的优化处理

试对下面基本块进行优化

1. 应用DAG对该基本块进行优化, 给出优化后的语句序列

2. 给出当只有L在基本块出口后为活跃时的优化结果

X=B*C
Y=B/C
Z=X+Y
W=9*Z
G=B*C
T=G*T
W=T*G
L=W
M=L



按照构造DAG结点次序重写四元式序列

X=B*C
G=X
Y=B/C
Z=X+Y
T=G*T
W=T*G
L=W
M=L

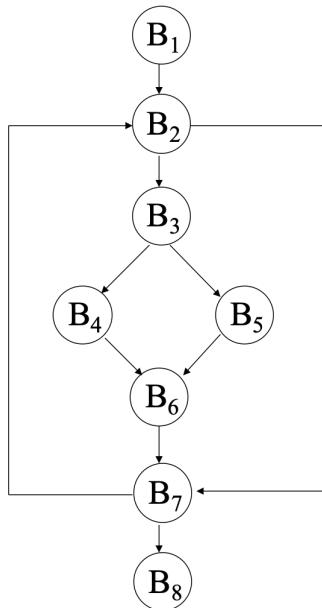
因为只有L在基本块出口后为活跃，即只有L在基本块出口处要引用，因此其他变量赋值语句可删去，其优化结果为：

G=B*C
T=G*T
L=T*G

7.3 循环优化

- 循环查找
 - 1. 求流图中所有结点n的必经结点集D(n)
 - 2. 求流图中的回边
 - 3. 根据回边求循环
- 控制流程图（程序流图/流图）G

- G 是一个三元组 $G=(N, n_0, E)$, 其中:
 - N 表示流图的有限结点集, 流图中每一个结点对应程序中的一个基本块, 因此, N 实质上就是一个程序的基本块集
 - n_0 表示唯一的首结点, 流图的首结点是包含程序第一个语句的基本块
 - E 表示流图的有向边集
- 为了找出流图中的循环, 需分析流图中结点间的控制关系, 因此引入必经结点和必经结点集的概念
- 必经结点: 在程序流图中, 对任意两个结点 a 和 b , 若从流图的首结点出发, 到达 a 的任一通路都要经过 b , 则称 b 是 a 的必经结点, 记为 $b \text{ DOM } a$
- 必经结点集: 流图中结点 a 的所有必经结点的集合称为结点 a 的必经结点集, 记为 $D(a)$
- 求流图中结点 n 的必经结点集 $D(n)$ 的算法思想:
 - {结点 n } \cup { n 的所有前驱结点的必经结点的交}, 即如果某结点 d 是结点 n 的所有前驱结点的必经结点, 则 d 为 n 的必经结点



流图中各结点的必经结点集:

$$\begin{aligned}
 D(B_1) &= \{B_1\} \\
 D(B_2) &= \{B_1, B_2\} \\
 D(B_3) &= \{B_1, B_2, B_3\} \\
 D(B_4) &= \{B_1, B_2, B_3, B_4\} \\
 D(B_5) &= \{B_1, B_2, B_3, B_5\} \\
 D(B_6) &= \{B_1, B_2, B_3, B_6\} \\
 D(B_7) &= \{B_1, B_2, B_7\} \\
 D(B_8) &= \{B_1, B_2, B_7, B_8\}
 \end{aligned}$$

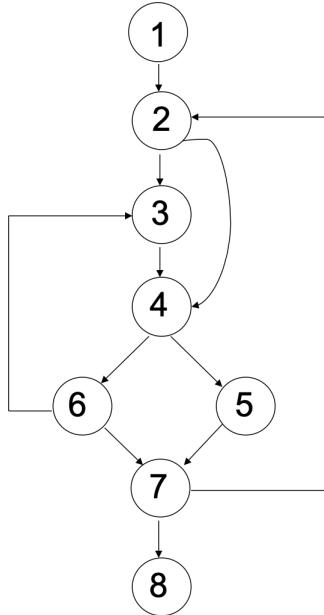
- 求流图中的回边
 - 设 $a \rightarrow b$ 是流图中的一条有向边, 若 $b \text{ DOM } a$, 则称 $a \rightarrow b$ 是流图中的一条回边
- 因为 $B_2 \text{ DOM } B_7$, 所以 $B_7 \rightarrow B_2$ 是流图中的回边, 且是流图中的唯一回边
- 求循环的算法思想:
 - 回边 n 到 d 组成的循环: d 是循环的入口, n 是循环的出口, 循环出口 n 的前驱属于循环, 前驱不是 d , 再求前

驱，直到入口d为止

由回边 $B_7 \rightarrow B_2$ 组成的循环是 $\{B_2, B_3, B_4, B_5, B_6, B_7\}$

设有如下图所示的程序流图：

1. 求流图中各结点 n 的必经结点集 $D(n)$
2. 求出该流图中的回边
3. 求出该流图中的循环



$$\begin{aligned}D(1) &= \{1\} \\D(2) &= \{1, 2\} \\D(3) &= \{1, 2, 3\} \\D(4) &= \{1, 2, 4\} \\D(5) &= \{1, 2, 4, 5\} \\D(6) &= \{1, 2, 4, 6\} \\D(7) &= \{1, 2, 4, 7\} \\D(8) &= \{1, 2, 4, 7, 8\}\end{aligned}$$

$7 \rightarrow 2$ 为回边，回边组成的循环： $\{2, 3, 4, 5, 6, 7\}$

- 对循环中的代码可以实行：

- 代码外提
- 强度削弱
- 删除归纳变量