编译原理总结

By August

总述：

1 前面 1~4 章部分主要根据陈老师课件，师兄的总结，龙书整理。这里陈老师说会考 4 道大题。

2 本总结对应龙书的 6~9 章部分没有整理，请大家翻阅其他两份资料(《编译原理押题王》《编译原理复习资料》) 和看书理解重点章节 (6.1, 6.2, 6.6 ,8.1, 8.2, 8.4, 8.6, 8.8, 9.1, 9.4) 。

3 后面章节对应冯老师第 11,12,14 次课件总结。

4 前面两章概念较多，大概看看就可以。后面第三，第四章和 3 章 ppt 内容是重点。一些提到的书上的页数可能不同，可以自行更改。

5 设有简易目录，主要是例题详解部分。

6 打印是可以先将全部字体设为黑色，以免红色字体看不清，之后可根据电子版勾画（如果觉得粗体就可以了，这步可以省略。。。）

7 本学渣知识水平有限，有不足的地方希望大家谅解。

祝大家考试顺利！

目录

第一章

一、 基本概念：

1. 词法分析（Lexical Analysis or Scanning）

• 任务1：将字符序列转化为**单词（token）序列**（涉及到第三章，**正则表达式 -> NFA -> DFA -> 最小化DFA，都要掌握 重点必考**）

• 任务2：给出单词的类别和一些相关属性值

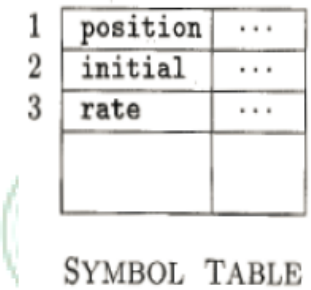
属性值放在公用的**符号表（Symbol Table）**中，词法分析过程返回单词的属性值存放的地址。

例子：对于下面的式子

```
position = initial + rate * 60
```

词法分析的结果返回

$\langle \mathbf{id}, 1\rangle\ \langle = \rangle\ \langle \mathbf{id}, 2\rangle\ \langle + \rangle\ \langle \mathbf{id}, 3\rangle\ \langle * \rangle\ \langle 60 \rangle$

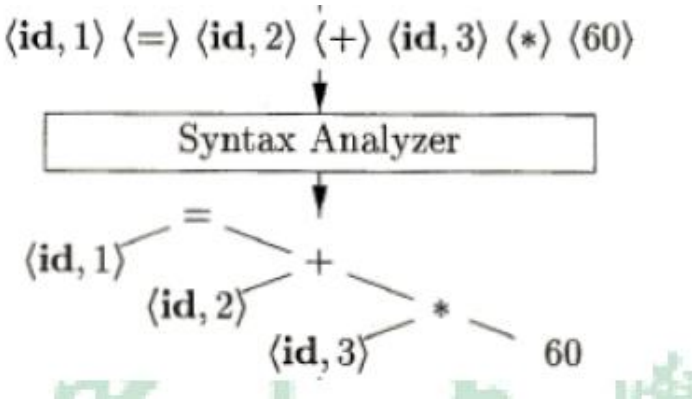


SYMBOL TABLE

2. 语法分析（Syntax Analysis）

任务：在分词的基础上建立**语法分析树。（第四章内容，都是重点）**

通常词法分析是作为一个过程被语法分析不断调用，每次调用取得一个单词

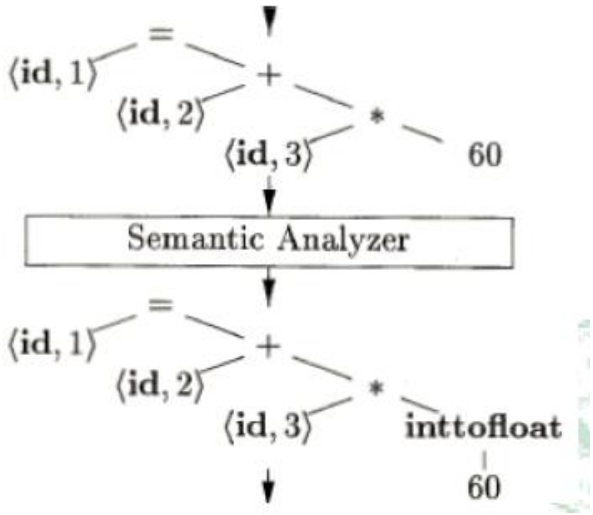


3. 语义分析（Semantic Analysis）

任务1：**类型检查**

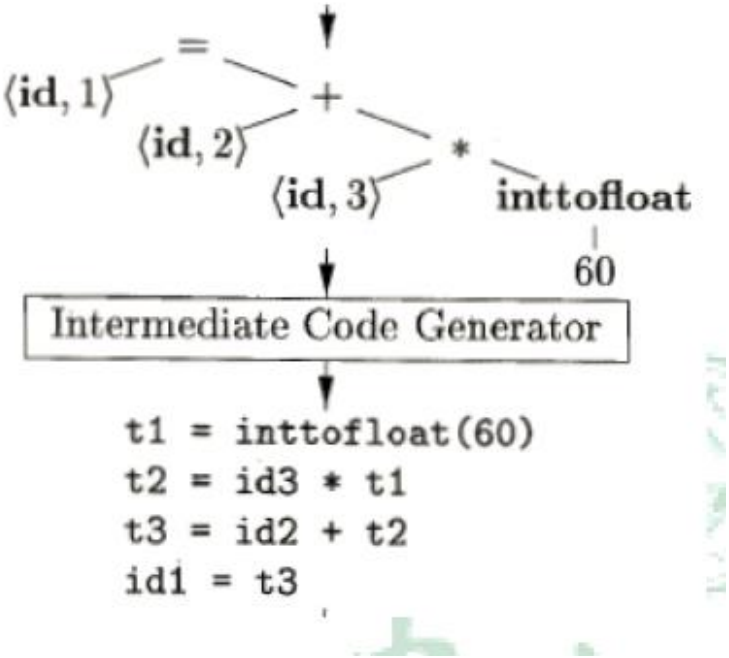
任务2：**类型转换**

其它



4. 中间代码生成（ Intermediate Code Generation）

- 中间代码介于源语言和目标语言之间
- 既容易由源语言转化，又容易转化为目标语言
- 如三地址语言
  - 生成中间代码的好处：实现了前端和后端的分离



5. 代码优化（ Code Optimization ）

目标：对中间代码进行优化，提高代码运行速度，缩短代码长度。

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Code Optimizer

```
t1 = id3 * 60.0
id1 = id2 + t1
```

6. 代码生成（ Code Generation）

任务：生成最终代码

```
t1 = id3 * 60.0
id1 = id2 + t1
```

Code Generator

```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

概念总结： 编译器的结构

编译器能够把源程序映射为在语义上等价的目标程序，这个映射过程由两个部分组成：**分析部分**和**综合部分**。

分析（analysis）部分把源程序分解成为多个组成要素，并在这些要素之上加上语法结构，然后使用这个结构来创建该源程序的一个中间表示；还会收集有关源程序的信息，并把信息存放在一个称为**符号表**的数据结构中。分析部分经常被称为编译器的**前端（front end）**。

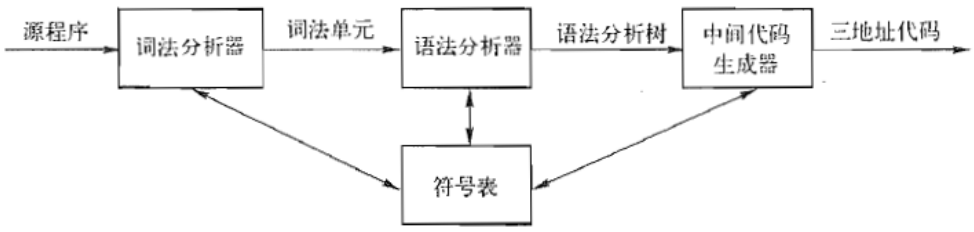


图 2-3 一个编译器前端的模型

综合（synthesis）部分根据中间表示和符号表中的信息来构造用户期待的

目标程序。综合部分称为**后端（back end）**。

管道设计：字符流->**词法分析器**->符号流->**语法分析**->语法树->**语义分析**->语法树->**中间代码生成器**->中间表示形式->**机器无关代码优化器**->中间表示形式->**代码生成器**->目标机器语言->**机器相关代码优化器**->目标机器语言。

（1） **词法分析（lexical analysis）**：又称为**扫描（scanning）**，读入组成源程序的字符流，并且将它们组成为有意义的**词素（lexeme）**序列。产生词法单元（token）作为输出：<token-name, attribute-value>

（2）**语法分析（syntax analysis）**：又称为**解析（parsing）**，使用词法分析器生成的各词法单元的第一个分量来创建树形的中间表示，一个常用的表示方法是**语法树（syntax tree）**。

（3）**语义分析（semantic analysis）**：使用语法树和符号表中的信息来检查源程序是否和语言定义的语义一致，并进行**类型检查（type checking）**以及可能有的**自动类型转换（coercion）**。

（4）中间代码生成：一种中间表示形式称为**三地址代码（three-address code）**，这种中间表示由一组类似于汇编语言的指令组成，每个指令具有三个运算分量。每个运算分量都像一个**寄存器**。

（5）**代码优化**：改进中间代码，提高运行速度，缩短代码长度。

（6）**代码生成**：生成最终代码，至关重要的方面是合理分配寄存器以存放变量的值。

第二章

涉及到的内容是后面3-6章内容的简要概括，这里就给出一些概念，详细的总结在后面章节都有包括。

1）引言

编译器在分析阶段把一个源程序划分成各个部分，并生成源程序的内部表示形式。这种内部表示称为中间代码。然后，编译器在合成阶段将这个中间代码翻译成目标程序。

一个程序设计语言的**语法（syntax）**描述了该语言的程序的正确形式，而该语言的**语义（semantics）**则定义了程序的含义，即每个程序在运行时做什么事情。

我们**用上下文无关文法来描述语法，上下文无关文法还可以指导程序的翻译过程**，也就是**语法制导翻译（syntax-directed translation）**。

2）语法定义

文法自然地描述了大多数程序设计语言构造的层次化语法结构。例如

stmt-> if (expr) stmt else stmt

这样的规则称为**产生式（production）**。在一个产生式中，

像关键字if和括号这样的词法元素称为**终结符号（terminal）**。

像 expr和 stmt 这样的变量表示终结符号的序列，称为**非终结符号（nonterminal）**。

一个**上下文无关文法（context-free grammar）**由四个元素组成：

1）一个**终结符号集合**，也称为词法单元。终结符号是该文法所定义的语言的基本符号的集合。

2） 一个**非终结符号的集合**，也称为语法变量。每个非终结符号表示一个终结符号串的集合。

3）一个**产生式集合**，其中每个产生式包括一个称为产生式头或左部的**非终结符号**，一个箭头，和一个称为产生式体或右部的**由终结符号及非终结符号组成的序列**。

4）指定一个**非终结符为开始符号**。

如果某个非终结符号是某个产生式的头部，我们就说该产生式是该非终结符号的产生式。

一个终结符号串是由零个或多个终结符号组成的序列。

零个终结符号组成的串称为**空串（empty string）**，记为 ε。

根据文法**推导**符号串时，我们首先从开始符号出发，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（language）**。

语法分析（parsing）的任务是：接受一个终结符号串作为输入，找出从文法的开始符号推导出这个串的方法。

语法分析树用图形的方式展现了**从文法的开始符号推导出相应语言中的符号串**的过程。

一个文法的语言的另一个定义是指任何能够由某棵语法分析树生成的符号串的集合。

为一个给定的终结符号串构建一棵语法分析树的过程成为对该符号串进行语法分析。

一个文法可能有多棵语法分析树能够生成同一个给定的终结符号串。这样的文法称为具有二

义性（ambiguous）。

3）语法制导翻译

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。

语法制导定义（syntax-directed definition）把每个文法符号和一个属性集合相关联，并且把每个产生式和一组语义规则（semantic）相关联，这些规则用于计算与该产生式中符号相关联的属性值。

树的遍历将用于描述属性的求值过程，以及描述一个翻译方案中的各个代码片段的执行过程。一个树的遍历（traversal）从根结点开始，并按照某个顺序访问树的各个结点。

4）语法分析

大多数语法分析方法都可以归入以下两类：自顶向下（top-down）方法和自底向上（bottom-up）方法。这两个术语指的是语法分析树结点的构造顺序。

自顶向下方法可以较容易地手工构造出高效的语法分析器。不过，自底向上方法可以处理更多种文法和翻译方案，所以直接从文法生成语法分析器的软件工具常常使用自底向上方法。

递归下降分析方法（recursive-descent parsing）是一种自顶向下的语法分析方法，它使用一组递归过程来处理输入，当出现如下所示的“左递归”产生式时，分析器就会出现无限循环：

```
expr -> expr + term
```

因为产生式A->Aa 的右部最左符号是A 自身，非终结符合A 和它的产生式就称为左递归（left recursive）。

语法分析的结果是源代码的一种中间表示形式，称为中间代码。抽象语法树中的各个结点代表了程序构造，一个结点的子结点给出了该构造有意义的子构造。

另一种表示方法是三地址代码，它是一个由三地址指令组成的序列，其中每个指令只执行一个运算。

5）符号表

符号表（symbol table）是存放有关标识符的信息的数据结构。当分析一个标识符的声明的时候，该标识符的信息被放入符号表中。当在后来使用这个标识符时，比如它作为一个表达式的因子使用时，语义动作将从符号表中获取这些信息。

第三章 词法分析（Lexical Analysis）

正则表达式是一种可以很方便地描述词素模式的方法。将对正则表达式进行转换：首先转换为不确定有穷自动机，然后再转换为确定有穷自动机。

正则表达式—>NFA—>DFA—>最小化

1）词法分析器的作用 （书上页数）

词法分析是编译的第一阶段。

主要任务：读入源程序的输入字符、将它们组成词素（Lexemes），生成并输出一个词法单元序列（a sequence of tokens），每个词法单元对应于一个词素。

中英文对照图：

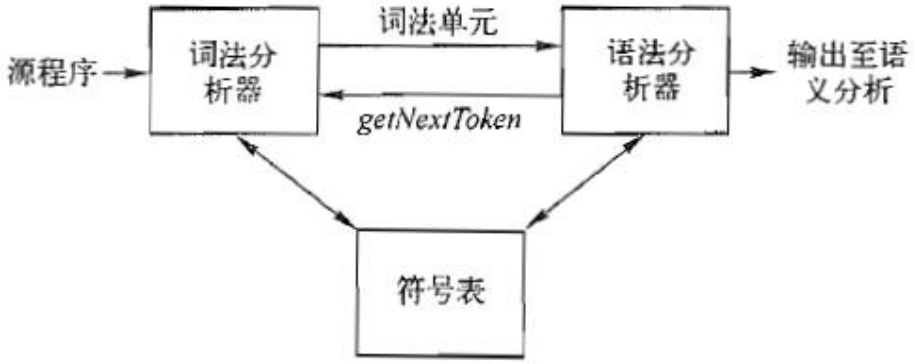
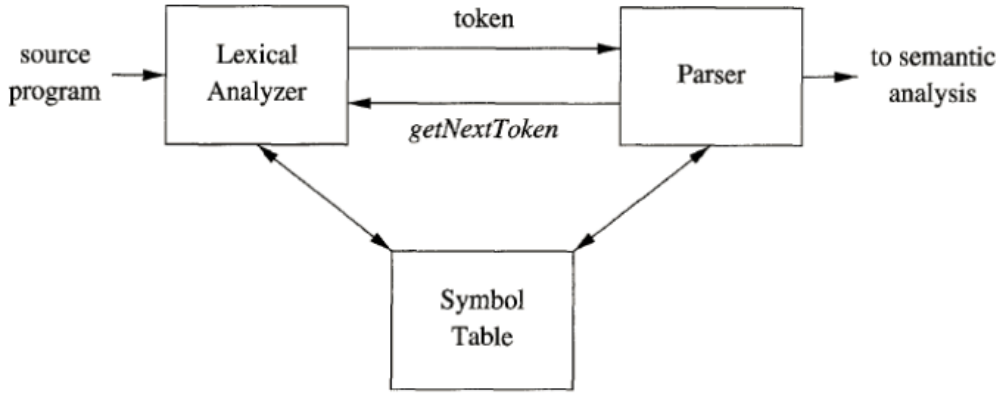


图 3-1 词法分析器与语法分析器之间的交互



*为什么把编译过程划分为词法分析和语法分析？

简化编译器的设计、提高编译器的效率、增强编译器的可移植性

* 词法单元、模式和词素

词法单元(Tokens)：一个词法单元名和一个可选的属性值组成。

模式(Patterns)：描述了一个词法单元的词素可能具有的形式。

词素(Lexemes)：是源程序中的一个字符序列。

词法单元属性(attribute value)，当多个词素可以和一个模式匹配，需要一个属性值作为区分。

Tokens’ s classes（词法单元类别）：关键字（keyword）， 运算符（operators）， 所有标识符（identifiers）， 常量（constant）， 标点符号（punctuation symbol）

1）每个关键字有一个词法单元。一个关键字的模式就是该关键字本身。

2）表示运算符的词法单元。它可以表示单个运算符，也可以像图3-2中的comparison那样，表示一类运算符。

3）一个表示所有标识符的词法单元。

4）一个或多个表示常量的词法单元，比如数字和字面值字符串。

5）每一个标点符号有一个词法单元，比如左右括号、逗号和分号。

e.g

```
E = M * C ** 2
```

中的词法单元名字和相关的属性值可写成如下的名字-属性对序列：

<id, 指向符号表中E的条目的指针>
<assign_op>
<id, 指向符号表中M的条目的指针>
<mult_op>
<id, 指向符号表中C的条目的指针>
<exp_op>
<number, 整数值2>

2）词法单元的规约（书上页数）

（1）字母表（alphabet）和语言（languages）

字母表：字符的集合

语言：字母表Σ上的语言，是由Σ中字符组成的字符串的集合

例如：

Σ={0, 1}

则{001, 100100}，{}，{1, 11, 111, 1111, …}都是定义在字母表Σ的语言

（2） 串（string）的术语

串s 的前缀prefix：从s 尾部删除0~n 个符号的串。

串s 的后缀suffix：从s 头部删除0~n 个符号的串。

串s 的子串substring：删除s 的某前缀和某后缀之后的串。

串s 的真前缀、后缀、子串：既不是空也不是s。

串s 的子序列：从s 中删除0~n 个符号得到的串。

（3）语言上的运算

语言是一种集合，所以集合的运算也适用于语言

OPERATION	DEFINITION AND NOTATION
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

正则表达式（用来表示一个模式）

e. g

例3.4 令 $\Sigma = \{a, b\}$。

1）正则表达式 **a|b** 表示语言 $\{a, b\}$。

2）正则表达式(**a|b**)(**a|b**)表示语言 $\{aa, ab, ba, bb\}$，即在字母表 Σ 上长度为 2 的所有串的集合。可表示同样语言的另一个正则表达式是 **aa|ab|ba|bb**。

3）正则表达式 **a*** 表示所有由零个或多个 a 组成的串的集合，即 $\{\epsilon, a, aa, aaa, \cdots\}$。

4）正则表达式(**a|b**)* 表示由零个或多个 a 或 b 的实例构成的串的集合，即由 a 和 b 构成的所有串的集合 $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \cdots\}$。另一个表示相同语言的正则表达式是 (**a***|**b***)*。

5）正则表达式 **a|a*b** 表示语言 $\{a, b, ab, aab, aaab, \cdots\}$，也就是串 a 和以 b 结尾的零个或多个 a 组成的串的集合。 □

单目后缀运算符+表示一个正则表达式及其语言的正闭包。

(r*= r+| ε =rr*=r*r)

单目后缀运算符？的意思是“零个或一个出现”。

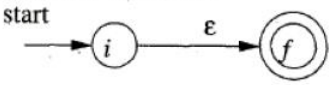
a1|a2|a3|.....|an = [a1a2a3...an]

3）词法单元的识别（关于NFA，DFA，重难点，必考！）

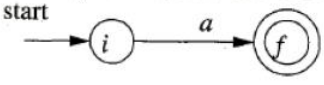
Step1 从正则表达式（regular expression）到NFA（非确定有限自动机）：

A 基本转换，对每一个字符：

BASIS: For expression ϵ construct the NFA

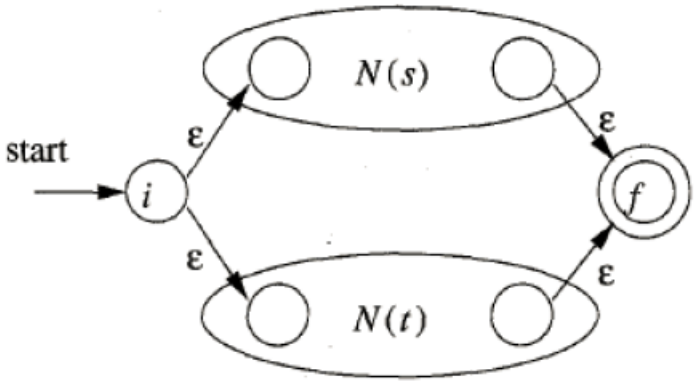


For any subexpression a in Σ, construct the NFA



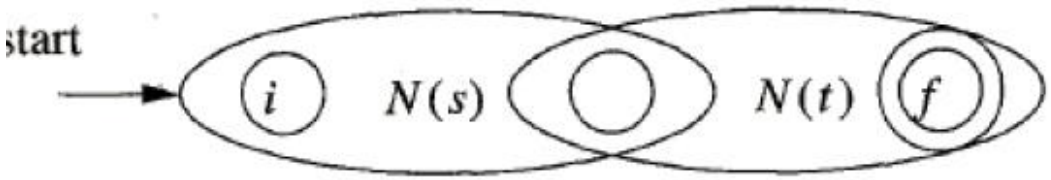
B（s|t）：

Suppose $r = s|t$. Then $N(r)$, the NFA for r, is constructed as



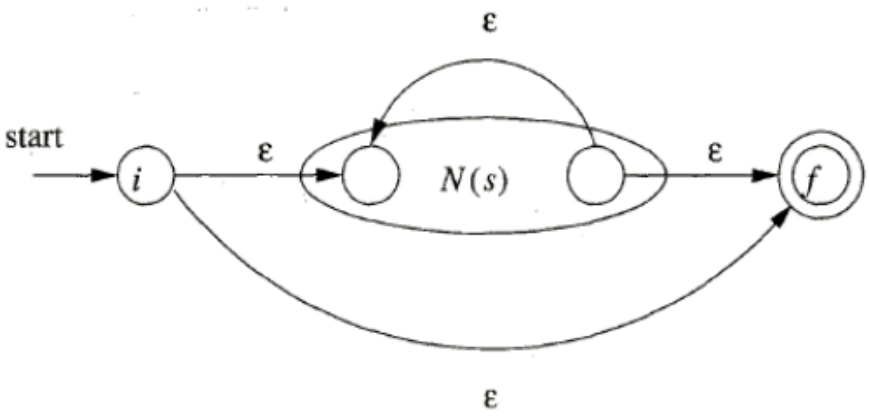
C st:

Suppose $r = st$. Then construct $N(r)$ as



D s*:

Suppose $r = s^*$. Then for r we construct the NFA $N(r)$



Step 2： 从NFA到DFA（确定的有限自动机）

DFA（没有 ε 的转换）

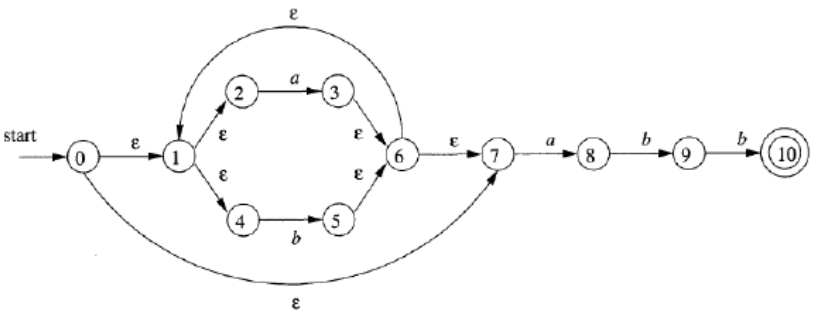
一些记号：
符号的闭包（ε作为输入时，当前状态和下一状态等效，即在同一个闭包集合里）

ϵ-*closure*(s)	Set of NFA states reachable from NFA state s on ϵ-transitions alone.

符号集合的闭包（T是多个s的集合）

ϵ-*closure*(T)	Set of NFA states reachable from some NFA state s in set T on ϵ-transitions alone; $= \cup_{s \text{ in } T}$ ϵ-*closure*(s).

推荐使用列表法来将NFA化为DFA，最重要的是计算闭包集合时一定要细心！
e.g

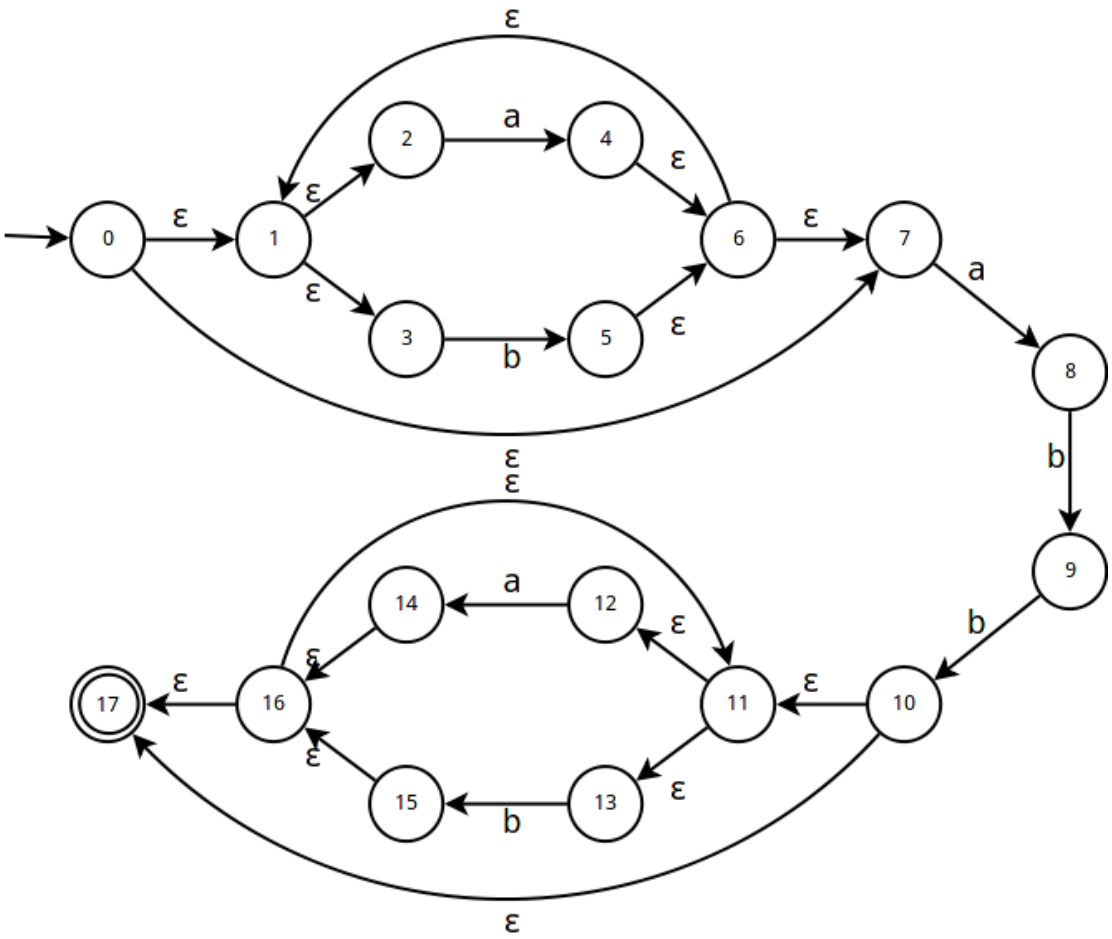


NFA states	DFA state	a	b
{0, 1, 2, 4, 7}	A	B	C
{1, 2, 3, 4, 6, 7, 8}	B	B	D
{1, 2, 4, 5, 6, 7}	C	B	C
{1, 2, 4, 5, 6, 7, 9}	D	B	E
{1, 2, 4, 5, 6, 7, 10}	E	B	C

Step 3： 最小化DFA（难点）
算法描述见书上3.6.2。
之后会给出一道关于有限状态机的完整例题，会有详细说明。

第三章NFA,DFA例题详解

根据以下正则表达式 ，构造NFA，确定DFA，并化简DFA。
(a|b)*abb(a|b)*
1 构造NFA：（注意要标明终止状态）
如何判断NFA的终止状态：该节点出度为 0 （只有指向该节点的有向边，没有从该节点指向其他节点的有向边）

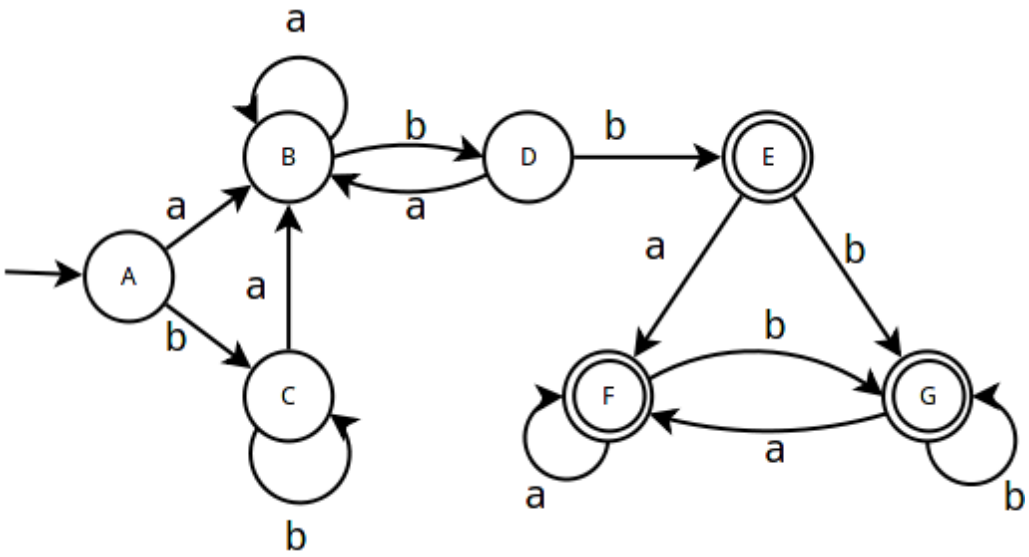


2 NFA → DFA （列表法，找闭包集合）

DFA

NFA State	DFA State	a	b
{0,1,2,3,7}	A	B	C
{1,2,3,4,6,7,8}	B	B	D
{1,2,3,5,6,7}	C	B	C
{1,2,3,5,6,7,9}	D	B	E
{1,2,3,5,6,7,10,11,12,13,17}	E	F	G
{1,2,3,4,6,7,8,11,12,13,14,16,17}	F	F	G
{1,2,3,5,6,7,11,12,13,15,16,17}	G	F	G

！！注意：新状态中，E,F，G 三个状态包含了 NFA 的终止状态（第 17 个状态，所以在 DFA 中 E，F，G 都是终止状态（也叫接受状态）。



3 最小化DFA状态数：

① 将终止状态和非终止状态划分为两个组，形成最初的划分P1

P1: {{A，B，C，D}， {E，F，G}}

② 对于 P1 中每个组，检查该组中的每个元素经过**所有转换函数**后到达的状态是否在P1中的同一组，有在不同组的即形成一个新的划分。

解释： 如{A，B，C，D}中，虽然每个元素经过“a”后，都还是到达{A，B，C，D}，但是经过“b”后，A，B，C三个元素到达{A，B，C，D}组， D到达{E，F，G}组。（**注意要验证所有转换函数，假设该组中还存在一个“c”的转换，还要验证c，全部满足才可以确定在同一组**）

所以{A，B，C，D} 被分为 {A，B，C}， {D} 两组。

同理：{E，F，G}中每个元素经过“a”和“b”后，都还是在同一组，即{E，F，G}中，所以该组不用继续划分。

现在我们得到一个新的划分P2，与P1不同**（算法终止就是P（i+1）和 P(i)相同）**

P2: {{A，B，C}，{D}，{E，F，G}}

③ 对于P2中的每个组，按照第②步的做法，检查划分，得到P3

P3: {{A，C}，{B}，{D}，{E，F，G}} (B 经过函数 “b”后到达了另一组)

④对于P3中的每个组，按照第②步的做法，检查划分，得到P4

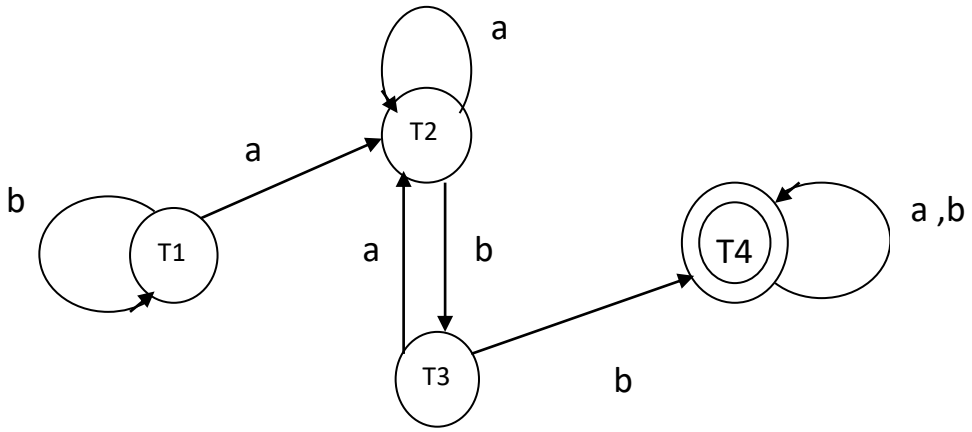
P4: {{A，C}，{B}，{D}，{E，F，G}}

此时和P3一致，停止划分，结束。

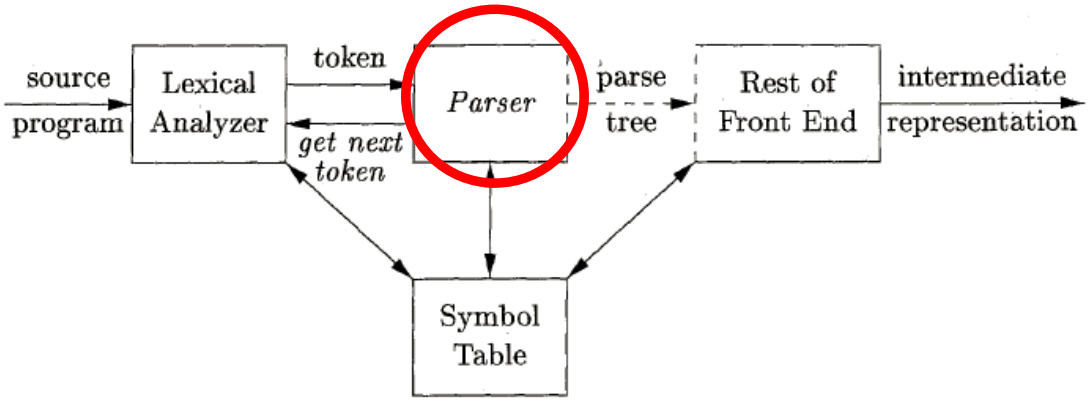
对于新状态列出表格，画出最小化DFA的状态图

分组	新状态	a	b
{A, C}	T1	T2	T1
{B}	T2	T2	T3
{D}	T3	T2	T4
{E, F, G}	T4	T4	T4

最小化DFA状态图：



第四章 语法分析（Syntax Analysis）



语法分析要解决的问题包括：

• 从词法分析中获得的每个属性字（token）在语句中，

或在整个程序中扮演什么角色

例如：同样一个数字，在程序中可能是一个加数，可能是数组的下标， 可能是循环的次数，可能是某个函数的参数等

• 检查语句或程序是否符合程序语言的语法

1）上下文无关文法

组成：终结符 + 非终结符 + 唯一的开始符号 + 一组产生式（某个非终结符）

文法是比正则表达式表达能力更强的表示方法，每个正则语言都是一个上下文无关语言，但是反之不成立。

文法总结：

Class	Grammar	Restriction	Recognizer
3	Regular	$A \to aB$ or $A \to a$, where $A, B \in N \wedge a \in \Sigma \cup \{\varepsilon\}$. $A \to \varepsilon$ permitted if A is the start symbol and does not appear on the right of any production.	Finite-State Automaton (**FSA**)
2	Context-Free	$A \to \alpha$, where $A \in N \wedge \alpha \in (\Sigma \cup N)^*$.	Push-Down Automaton (**PDA**)
1	Context-Sensitive	$\alpha \to \beta$, where $\alpha, \beta \in (\Sigma \cup N)^* \wedge \alpha \neq \varepsilon \wedge \lvert\alpha\rvert \leq \lvert\beta\rvert$. β can't be ε, unless α is the start symbol and does not appear on the right of any production.	Linear-Bounded Automaton (**LBA**)
0	Unrestricted	$\alpha \to \beta$, where $\alpha, \beta \in (\Sigma \cup N)^* \wedge \alpha \neq \varepsilon$.	Turing Machine (**TM**)

Useful in Practice

Useful in Theory

Practice 正则表达式— FSA 有限状态机— 不能递归

上下文无关文法— PDA 下推自动机— 可以递归

上下文有关文法— LBA 线性限定

Theory 无限制— 图灵模型

处理文法的语法分析器分为三类：①通用的（很少用） ②自顶向下③自底向上
语法分析器的输入总是按照从左向右的方式被扫描，每次扫描一个符号。

chomsky 文法详解：

0型文法

设G=（VN，VT，P，S），如果它的每个产生式α→β是这样一种结构：α∈(VN∪VT)*且至少含有一个非终结符，而β∈(VN∪VT)*，则G是一个0型文法。0型文法也称短语文法。一个非常重要的理论结果是：0型文法的能力相当于图灵机(Turing)。或者说，任何0型文语言都是递归可枚举的，反之，递归可枚举集必定是一个0型语言。0型文法是这几类文法中，限制最少的一个，所以我们在试题中见到的,至少是0型文法。

1型文法

1型文法也叫上下文有关文法，此文法对应于线性有界自动机。它是在0型文法的基础上每一个α→β,都有|β|>=|α|。这里的|β|表示的是β的长度。

注意：虽然要求|β|>=|α|，但有一特例：α→ε也满足1型文法。

如有A->Ba则|β|=2,|α|=1符合1型文法要求。反之,如aA->a，则不符合1型文法。

2型文法

2型文法也叫上下文无关文法,它对应于下推自动机。2型文法是在1型文法的基础上,再满足：每一个α→β都有α是非终结符。如A->Ba,符合2型文法要求。

如Ab->Bab虽然符合1型文法要求,但不符合2型文法要求，因为其α=Ab，而Ab不是一个非终结符。

3型文法

3型文法也叫正规文法，它对应于有限状态自动机。它是在2型文法的基础上满足:A→α|αB（右线性）或A→α|Bα（左线性）。

如有：A->a,A->aB,B->a,B->cB，则符合3型文法的要求。但如果推导为:A->ab,A->aB,B->a,B->cB或推导\为:A->a,A->Ba,B->a,B->cB则不符合3型方法的要求了。具体的说,例子A->ab,A->aB,B->a,B->cB中的A->ab不符合3型文法的定义,如果把后面的ab,改成“一个非终结符＋一个终结符”的形式（即为aB）就对了。例子A->a,A->Ba,B->a,B->cB中如果把B->cB改为B->Bc的形式就对了,因为A→α|αB（右线性）和A→α|Bα（左线性）两套规则不能同时出现在一个语法中,只能完全满足其中的一个,才能算3型文法。

2）推导（**很大可能会考最左和最右。**）

最左推导：E -> (E + E) -> (id + E) -> (id + id)

最右推导：E -> (E + E) -> (E + id) -> (id + id)

句子：我们将由开始符号推导出来的**只含有终结符号**的串称为这个文法的**句子**。

语言：一个文法G定义的**语言**就是这个文法**所有句子的集合，记为L(G)**。

判断一个终端符号串x是否符合G的语法就是判断x是否属于L(G)。

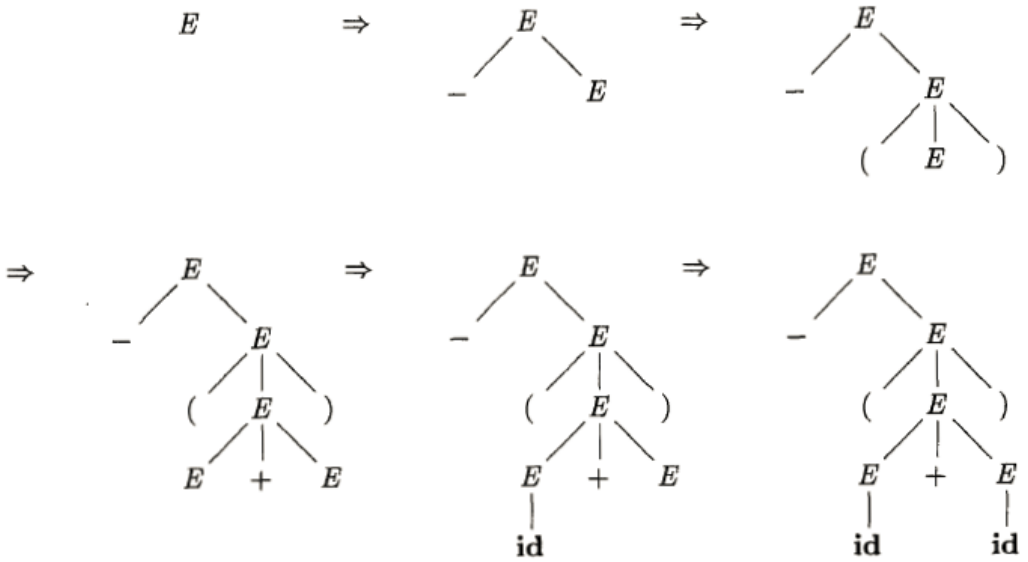
!! 注：

1）正则语言是上下文无关语言的真子集

2）如果一个语言不是上下文无关的，那它就是上下文有关语言。

语法分析树是推导的图形表示形式，它过滤掉了推导过程中对非终结符号应用产生式的顺序。一棵语法树的**叶子结点**的标号既可以是非终结符号，也可以是终结符号。**从左到右排列这些符号就可以得到一个句型**，称为这棵树的结果（yield）或边缘（frontier）。

一个最左推导的语法分析树：



二义性ambiguous：如果一个文法可以为某个句子生成多棵语法分析树，那么它就是二义性的。换句话说，就是对同一个句子有多个最左推导或是最右推导。

3）设计文法

（1）消除文法二义性（P122）

基本思想是让一个then 和一个else 之间出现的语句必须是已匹配的，不让else 悬空。

$$E \to E+E \mid E*E \mid (E) \mid \mathrm{id}$$

$$\Downarrow$$

$$\begin{aligned} E &\to E + T \mid T \\ T &\to T * F \mid F \\ F &\to (E) \mid \mathbf{id} \end{aligned}$$

（2）左递归的消除（P123）：

A -> Aα | β 替换为

A -> βA`

A` -> αA` | ε

详细算法： 书上算法4.8（P124）

（3）提取左公因子：

提取左公因子是一种文法转换方法，它可以产生适用于预测分析技术或**自顶向下分析技术的文法**。

对于每个非终结符，找出它的两个或多个选项之间的最长公共前缀，提取出来。

例如将A -> αβ1 | αβ2 |...|αβn | γ

替换为 A -> αA’ | γ

A’ -> β1 | β2 |...| βn

4）自顶向下的语法分析 （重点掌握 First，Follow集合的构造，LL（1）文法的预测分析表的构造）

自顶向下语法分析可以被看作是**为输入串构造语法分析树**的问题，它从语法分析树的根结点开始，按照先根次序创建这棵语法分析树的各个结点。自顶向下语法分析也可以被看作寻找输入串的**最左推导**过程。

在自顶向下语法分析过程中，**FIRST 和FOLLOW 使得我们可以根据下一个输入符号来选择应用哪个产生式**。

注： 自顶向下语法分析不能处理左递归，因此要用该方法时，要消除左递归。

通用的递归下降分析技术有可能回溯。

FIRST集

定义：FIRST(α)定义为可从α推导得到的串的首符号的集合。

算法：

①如果α是一个**终结符**，那么**FIRST(α)=α**

②如果α是一个非终结符，且α->Y1Y2...是一个产生式，若ε在FIRST(Y1) ... FIRST(Yi-1)中，则FIRST(Yi)中的所有符号一定在FIRST(α)中

③如果α->ε，那就将ε加入到FIRST(α)中

FOLLOW集

定义：FOLLOW(α)定义为可能在某些句型中紧跟在α右边的终结符的集合。

算法：

①将$放到FOLLOW(S)中，其中S 是开始符号，$是结束标志

②如果存在A->αBβ，那么FIRST(β)中除ε以外的所有符号都在FOLLOW(B)中

③如果存在A->αB，或A->αBβ且FIRST(β)包含ε，那么FOLLOW(A)中的所有符号都在FOLLOW(B)中

LL(1)文法

第一个L：输入字符串从左边开始扫描。

第二个L：得到的推导是最左推导。

1：向前看1 个输入符号。

左递归和二义性的文法都不可能是LL(1)的。

满足条件：

一个文法满足LL(1)文法时，当且仅当任意两个不同的产生式A->α|β满足：

①不存在终结符a 使得α和β都能推导出以a 开头的串

②α和β中最多只有一个可以推导出空串

（前两条等价于说FIRST(a)与FIRST(b)是不相交的集合）

③如果β->ε，那么α不能推导出任何以FOLLOW(A)中某个终结符开头的串。α->ε也类似。（也即FIRST(a)和FOLLOW(A)是不相交的集合）

所有的一切限制就是为了检查当前符号就可以确定要用那个产生式去推导。

构造预测分析表：

对每个产生式 A->α：

①每个FIRST(α)中的终结符a，将A->α加入到表中M[A,a]

②如果ε在FIRST(α)中,那么对于FOLLOW(A)中的每个终结符b,将A->α加入到表中M[A,b]。

如果ε在FIRST(α)中，且$在FOLLOW(A)中，也将A->α加入到表中M[A,$]。

第四章LL（1）文法 预测分析表构造例题详解

根据以下文法G，构造预测分析表M，判断是否为LL（1）文法（即书上4.17的例题）

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \mathbf{id}
\end{array}
$$

1 构造每个非终结符的First和Follow集，这里按照之前的算法构造。一般先计算First集

合，然后计算Follow集合。

Step 1 通过规则 ①如果α是一个终结符，那么FIRST(α)=α 得到以下表格：

	Frist	Follow
E		
E’	+, ε	
T		
T’	*, ε	
F	(, id	

Step 2 通过规则 ：②“如果α是一个非终结符，且α->Y1Y2...是一个产生式，若ε在FIRST(Y1) ...FIRST(Yi-1)中，则FIRST(Yi)中的所有符号一定在FIRST(α)中”。

因为 T-> FT’ ,First(F) = { (， id },所以First(T) = { (， id };

又因为 E->TE’, First(T) = { (， id }, 所以First(E) = { (， id }

	Frist	Follow
E	(, id	
E’	+, ε	
T	(, id	
T’	*, ε	
F	(, id	

Step 3 根据规则“①将$放到FOLLOW(S)中，其中S 是开始符号，$是结束标志”和规则“②如果存在A->αBβ，那么FIRST(β)中除ε以外的所有符号都在FOLLOW(B)中”

E 为开始符号，F->(E) | id, 所以Follow（E）= {), $}

E’ -> +TE’ | ε，所以 Follow（T）= First(E’) - ε = {+}

T’ -> *FT’ | ε，所以 Follow（F）= First(T’) - ε = {*}

	Frist	Follow
E	(, id	), $
E’	+, ε	
T	(, id	+
T’	*, ε	
F	(, id	*

Step 4 根据规则 “③如果存在A->αB，或A->αBβ且FIRST(β)包含ε，那么FOLLOW(A)中的所有符号都在FOLLOW(B)中”

因为 E->TE’,所以 Follow（E’） += Follow(E) = {), $}

因为E’ -> +TE’ | ε （E’-> ε），所以 Follow（T）中加入Follow（E’），Follow（T）={+,), $}

因为T->FT’,所以Follow(T’) += Follow(T) = {+,), $}

因为T’ ->*FT’ | ε （T’-> ε），所以 Follow（F）中加入Follow（T’），Follow（F）

={*, +,), $}

	Frist	Follow
E	(, id	), $
E’	+, ε	), $
T	(, id	+,) ,$
T’	*, ε	+,), $
F	(, id	*, +,), $

至此，完成First和Follow集合的构造。

2 根据构造的First和Follow集合构造预测分析表
规则如下：
①每个FIRST(α)中的终结符a，将A->α 加入到表中M[A,a]
②如果 ε 在FIRST(α)中，那么对于FOLLOW(A)中的每个终结符b，将A->α 加入到表中M[A,b]。如果 ε 在FIRST(α)中，且 $ 在FOLLOW(A)中，也将A->α 加入到表中M[A, $]。

依次对每个产生式进行分析：
① E -> TE’： First(TE’) = First(T) ={(, id}，所以将“E -> TE’”填入“（”和“id”中。
② E’->+TE’ | ε： First(+TE’ | ε)={+, ε}，因为此时First中有了 ε，所以Follow（E’）中的终结符“)”“$”要填入产生式E’->ε 。
接下的产生式都类似操作。

非终结符	输入符号					
	id	+	*	(	)	$
E	E -> TE’			E -> TE’		
E’		E’ ->+TE’			E’ -> ε	E’ -> ε
T	T->FT’			T->FT’		
T’		T’ -> ε	T’ ->*FT’		T’ -> ε	T’ -> ε
F	F->id			F->(E)		

至此，预测分析表构造完成。

3 判断是否为LL（1）文法。
判断方法：检查预测分析表中每一个表格是否有两个或多个产生式，如果有，则不是LL（1）文法，反之，则是。
该题中是LL（1）文法。

反例：以下分析表不是LL（1）文法

非终结符号	输入符号					
	a	b	e	i	t	\$
S	$S \to a$			$S \to iEtSS'$		
S'			$S' \to \epsilon$ $S' \to eS$			$S' \to \epsilon$
E		$E \to b$				

5）自底向上的语法分析 （比自顶向下语法分析应用范围更广）

自底向上的语法分析过程可以看成一个串规约为开始符号的过程**（即可以看做推导的反向操作）**。

归约（reduction）：一个与某产生式体相匹配的特定子串被替换为该产生式头部的非终结符号。

句柄（handle）：是和某产生式体匹配的子串，也就一个可归约的可行前缀。

句柄右边的串一定只包含终结符号。**（这里请看书上4.5.2节中关于句柄的解释，仔细理解）**

可行前缀是一个最右句型的前缀，并且它没有越过该最右句型的最右句柄的右端。

一个移入-归约语法分析器可采取如下四种可能的动作：（详细过程请看书上4.5.3的例子）

①**移入（shift）**：将下一个输入符号移到栈的顶端。

②**归约（reduce）**：**被归约的符号串的右端必然是栈顶**。语法分析器在栈中确定这个串的左端，并决定用哪个非终结符号来替换这个串。

③**接受（accept）**：宣布语法分析过程成功完成。

④**报错（error）**：发现一个语法错误，并调用一个错误恢复子例程。

概念重难点理解：句子（sentence），句型（sentential form），短语（phrase），句柄（handle）

1)句子和句型：

句子：由开始符号S可以推导出来的**只含有终结符号的符号串**叫做该文法的一个句子。

句型：由开始符号S可以推导出来的**所有符号串**都是该文法的句型。

也就是说句子只有终结符号，句型可以有非终结符号。句子和句型都是由开始符号S可以推导出来的。

2）短语和句柄

短语：

文法G[S]： S =>* xUy，U =>+ α，称α是句型xαy相对于U的**短语**。

S =>* xUy，U =>α，称α是句型xαy相对于U的**直接短语**或**简单短语**。

1．αδβ是必须是句型，即有 **S** $\Rightarrow^*$ αδβ。

2．另外，还要有 A $\Rightarrow^+$ δ，即有 S $\Rightarrow^*$ α**A**β $\Rightarrow^+$αδβ。（α**A**β也是个句型）

3．有了以上的条件，于是 δ 就是**句型** αδβ 的（记住，而不是α**A**β的！）**短语**。

4．如果还有 **A→δ**，即它是个产生式，则δ 就是**句型** αδβ 的**直接**短语。

句柄：

最左直接短语称**句柄**(Handle)。

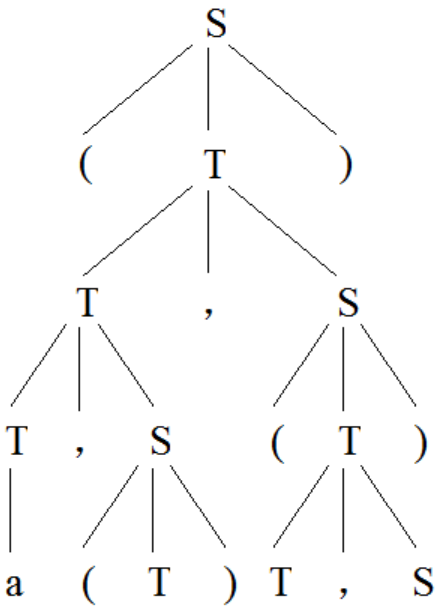
例6.3 句型α = (a, (T), (T, S))的语法树如右图所示。它有7个分枝结点，分别以这7个分枝结点为根结点，得7棵子树。再分别由这7棵子树的叶子结点（从左到右）组成7个符号串，它们是句型α的7个短语。这7个短语是：

- ① (a, (T), (T, S))
- ② a, (T), (T, S)
- ③ a, (T)
- ④ (T, S)
- ⑤ a
- ⑥ (T)
- ⑦ T, S

句型α的语法树有3棵简单子树。由这3棵简单子树的叶子结点组成的符号串，则是句型α的3个简单短语：

① a ② (T) ③ T, S

句型α的语法树的最左简单子树的叶子结点a便是句型α的句柄。



句柄在移进归约中的例子

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (\ E\ ) \mid \mathbf{id}
\end{aligned}
$$

STACK	INPUT	ACTION
$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	shift
$\$ \mathbf{id}_1$	$* \mathbf{id}_2 \$$	reduce by $F \rightarrow \mathbf{id}$
$\$ F$	$* \mathbf{id}_2 \$$	reduce by $T \rightarrow F$
$\$ T$	$* \mathbf{id}_2 \$$	shift
$\$ T *$	$\mathbf{id}_2 \$$	shift
$\$ T * \mathbf{id}_2$	$\$$	reduce by $F \rightarrow \mathbf{id}$
$\$ T * F$	$\$$	reduce by $T \rightarrow T * F$
$\$ T$	$\$$	reduce by $E \rightarrow T$
$\$ E$	$\$$	accept

为什么是移进而不是将T归约为E？

因为如果归约的话， 就得到 E * id ，然而 E*id 不能由开始符号推导得到，所以不是一个句型，所以要移进。

6) 自底向上的语法分析中的 LR语法分析技术（重点掌握LR（0），SLR ，LR（1））

概念：

目前最流行的自底向上语法分析器都基于LR(k)语法分析的概念。其中，“L”表示对输入进行从左到右的扫描，“R”表示反向构造出一个最右推导序列，而k表示在做出语法分析决定时向前看k 个输入符号。

只要存在这样一个从左到右扫描的移入-归约语法分析器，它总是能够在某文法的最右句型的句柄出现在栈顶时识别出这个句柄，那么这个文法就是LR 的。

7) LR(0)文法&SLR(1)文法

项指明了在语法分析过程中的给定点上，我们已经看到了一个产生式的哪些部分。

一个称为规范LR(0)项集族（canonical LR(0) collection）的一组项集提供了构建一个确定有穷自动机的基础。

为了构造一个文法的规范 LR(0)项集族，我们定义了一个增广文法（augmented grammar）和两个函数：CLOSURE和 GOTO。

如果 G 是一个以S 为开始符号的文法，那么G 的增光文法G'就是在G 中加上新开始符号S'和产生式S'->S 而得到的文法。引入这个新的开始产生式的目的是告诉语法分析器何时应该停止语法分析并宣称接受输入符号串。只有当用S'->S 进行归约时，输入符号串被接受。

项集的闭包（CLOSURE）算法

假设I 是文法G 的一个项集：

①将I 中各项加入到CLOSURE(I)中

②如果A->α ·Bβ在闭包中，B->γ是一个产生式，并且B->·γ不在闭包中，就加入其中。

③重复②直到没有新项加入为止

其中项可以分为两类

①内核项：包括初始项S'->·S 以及点不在最左端的所有项

②非内核项：除了S'->·S 以外的点在最左端的所有项

项集的GOTO函数

假设I 是一个项集而X 是一个文法符号。
GOTO(I,X)被定义为I 中所有形如[A->α ·Xβ]的项所对应的项[A->αX· β]的集合的闭包。直观的讲，GOTO 函数用于定义一个文法的LR(0)自动机中的转换。

语法分析表

①根据ACTION 和GOTO 函数写出对应状态集的移入状态，用Sn 表示
②当A->α ·时进行规约。若是LR(0)，对应所有输入符a 将ACTION[i,a]设置为 rn，若是SLR(1)，则**仅对应FOLLOW（A）中的输入符a 将ACTION[i,a]设置为rn。**
若在同一个项集中出现例如如下情况的时候，则出现了移入-归约冲突，表面SLR 文法在这里不可用，因为不够强大，不能记录足够的信息。

S->L·=R
R->·

可以出现在一个移入-归约语法分析器的栈中的**最右句型前缀称为可行前缀（viable prefix），它不可能越过最右句型的最右句柄的右端。**

第四章LR（0）语法 预测分析表构造例题详解

根据以下拓广文法，画出LR（0）自动机，列出LR（0）分析表，判断是否为LR（0）文法。
拓广文法为：

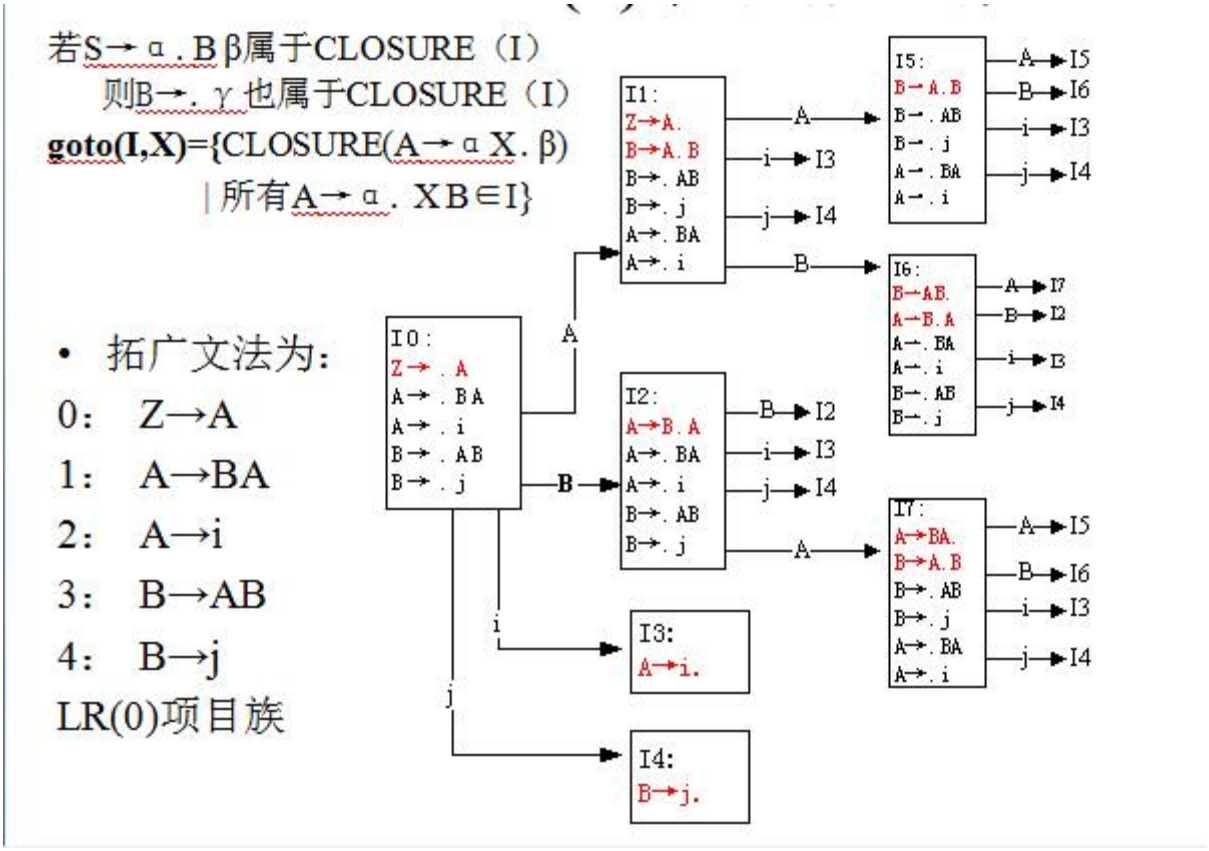
0： Z→A
1： A→BA
2： A→i
3： B→AB
4： B→j

1 构造LR(0)自动机

构造步骤：

Step 1 每个状态I里面要算闭包，就是要把“.”后面的符号的产生式也包含在其中。
算法：①将I 中各项加入到CLOSURE(I)中
②如果A->α ·Bβ在闭包中，B->γ是一个产生式，并且B->·γ不在闭包中，就加入其中。

Step 2 状态之间计算goto,主要就是把当前状态的“.”号后移一位。
算法：假设I 是一个项集而X 是一个文法符号。
GOTO(I,X)被定义为I 中所有形如[A->α ·Xβ]的项所对应的项[A->αX· β]的集合的闭包。



2 根据构造的项目族构造分析表

解释：

Sj： A→α . X β ∈ I_i，goto(I_i,X)=I_j， 如果X ∈V_T，则ACTION[I_i,X]=S_j

j： A→α . X β ∈ I_i，goto(I_i,X)=I_j，如果X ∈V_N，则GOTO[$I_{i,X}$]=j

Acc： S’ →S . ∈ I_i，则ACTION[I_i,#]=Acc

rk： A→α . ∈ I_i，该项目为k产生式的，则ACTION[I_i,v]=rk， v ∈V_T

状态	ACTION			GOTO	
	i	j	#	A	B
I0	S3	S4		1	2
I1	S3	S4	Acc	5	6
I2	S3	S4		7	2
I3	r2	r2	r2		
I4	r4	r4	r4		
I5	S3	S4		5	6

I6	S3/r3	S4/r3	r3	7	2
I7	S3/r1	S4/r1	r1	5	6

这里特别注意：LR(0)分析表中的“移进-归约”冲突。

解释：假设一个LR（0）项目集规范族中有如下项目集合：

{ X → α.bβ

A → γ.

B → δ.}

即存在移进-归约冲突和归约-归约冲突

所以诞生了SLR分析表。

3 不是LR（0）文法， 因为LR（0）文法必须满足：①移进项目与归约项目不能并存同一项目集。②多个归约项目不能并存同一项目集。

怎么利用构造的LR（0）语法分析表处理一个输入串： 书上例4.45详细看一下

第四章SLR语法 预测分析表构造例题详解

知识准备：

SLR以LR(0)项和LR(0)自动机为基础，要求知道输入文法的每个非终结符号A的Follow（A）。

- SLR中，每个项目集，如果有归约项目[A→α.，B→β.]，有移进项目[D→α.bγ]
- 要求：FOLLOW（A）、FOLLOW（B）互不相同，且不含b。

例题：对以下文法构造SLR分析表：

文法G(E)

E→E+T | T

T→T*F | F

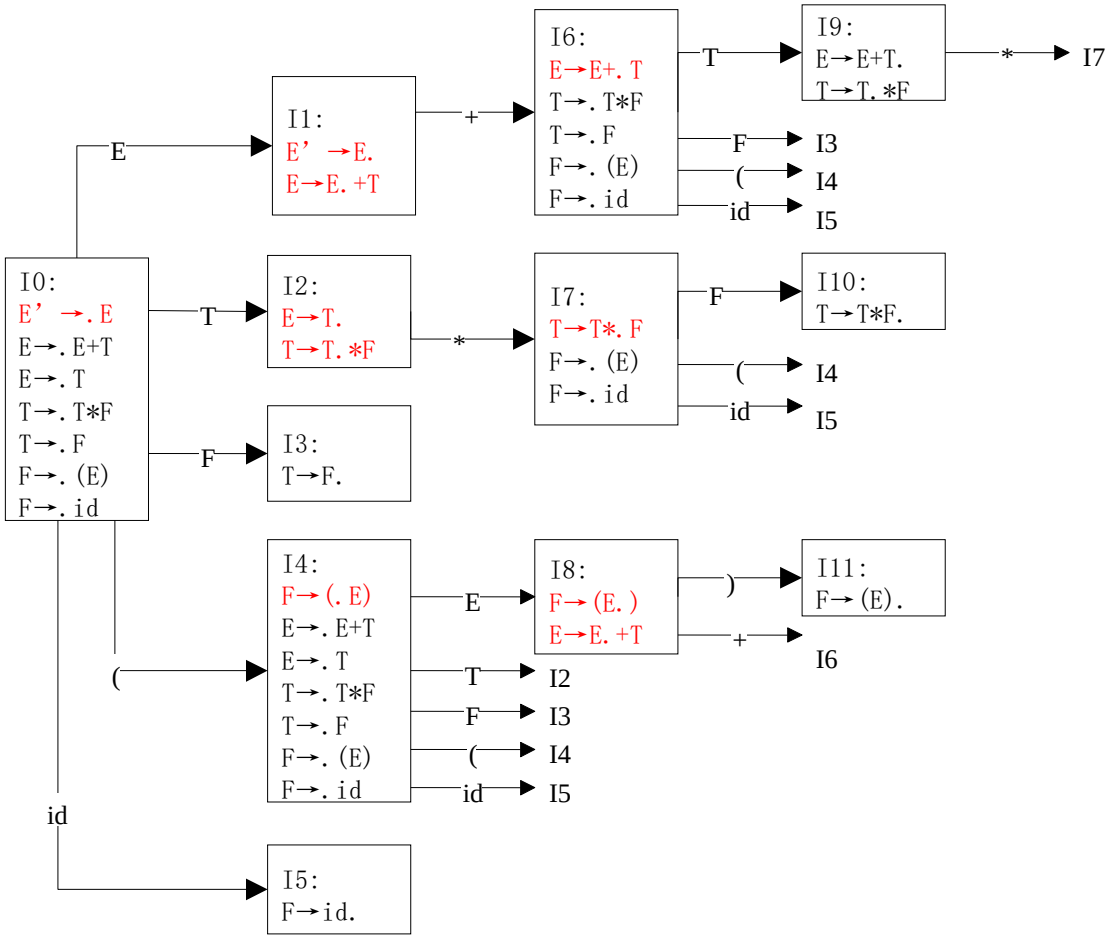
F→(E) | id

1 构造first和follow集合，主要是用到follow集合，之前有构造方法的详细解释，这里直接给出结果：

	FIRST	FOLLOW
E’	id,(	#
E	id,(	),+,#
T	id,(	*,),+,#
F	id,(	*,),+,#

2 画出LR（0）的自动机

0 E’ →E
1 E→E+T
2 E→ T
3 T→T*F
4 T→F
5 F→(E)
6 F→id



3 构造 SLR 分析表

解释：

Sj： A→α.X β ∈ I_i，goto(Ii,X)=I_j，如果X ∈V_T，则ACTION[Ii,X]=S_j

j： A→α.X β ∈ I_i，goto(Ii,X)=I_j，如果X ∈V_N，则GOTO[Ii,X]=j

Acc：S’ →S . ∈I_i，则ACTION[Ii,#]=Acc

rk： A→α. ∈I_i，该项目为k产生式的，则ACTION[I_i,v]=rk， v ∈FOLLOW(A) ，A不等于S’ 。（这里与LR（0）不同）

解释：

比如 I2 中， 有E -> T. 和 T -> T.*F

有E -> T. 所以对follow（E）中的所有终结符v,有ACTION[2,v]= r2(E→ T)。

又因为T -> T.*F， “*”为终结符，所以ACTION[2,*] = S7。

状态	ACTION						GOTO		
	id	+	*	(	)	#	E	T	F
0	S5			S4			1	2	3
1		S6				Acc			
2		r2	S7		r2	r2			
3		r4	r4		r4	r4			
4	S5			S4			8	2	3
5		r6	r6		r6	r6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		r1	S7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

4 判断是否为SLR（1）文法。

是，因为每个单元格中没有冲突。

结论：

SLR文法都是无二义性的。

8) LR(1)文法&LALR(1)文法

项集的闭包算法：

LR(1) 的项目： A→α．β，b

LR(1) 的闭包： A→α.Bβ，b 的闭包 B→.γ，FIRST（βb）

对于闭包的解释： B为一个非终结符，FIRST（βb）由两部分组成： 之前的b（A→α．β，b） 和 B后面符号串β的所有first

语法分析表：

基本与LR(0)的构造方法一致，区别在于规约。

LR(1)：如果[A->α •, a]，且A≠S'，那么将ACTION[i,a]设置为规约A->α 。

LALR(1)：在LR(1)的基础上将相同的项集合并，又不存在冲突，那就是LALR。

第四章规范LR（1）语法 预测分析表构造例题详解

对以下文法构造LR（1）分析表：

G[S’]

0 S’ →S

1 S→CC

2 C→cC

3 C→d

1 构造非终结符的 First集合

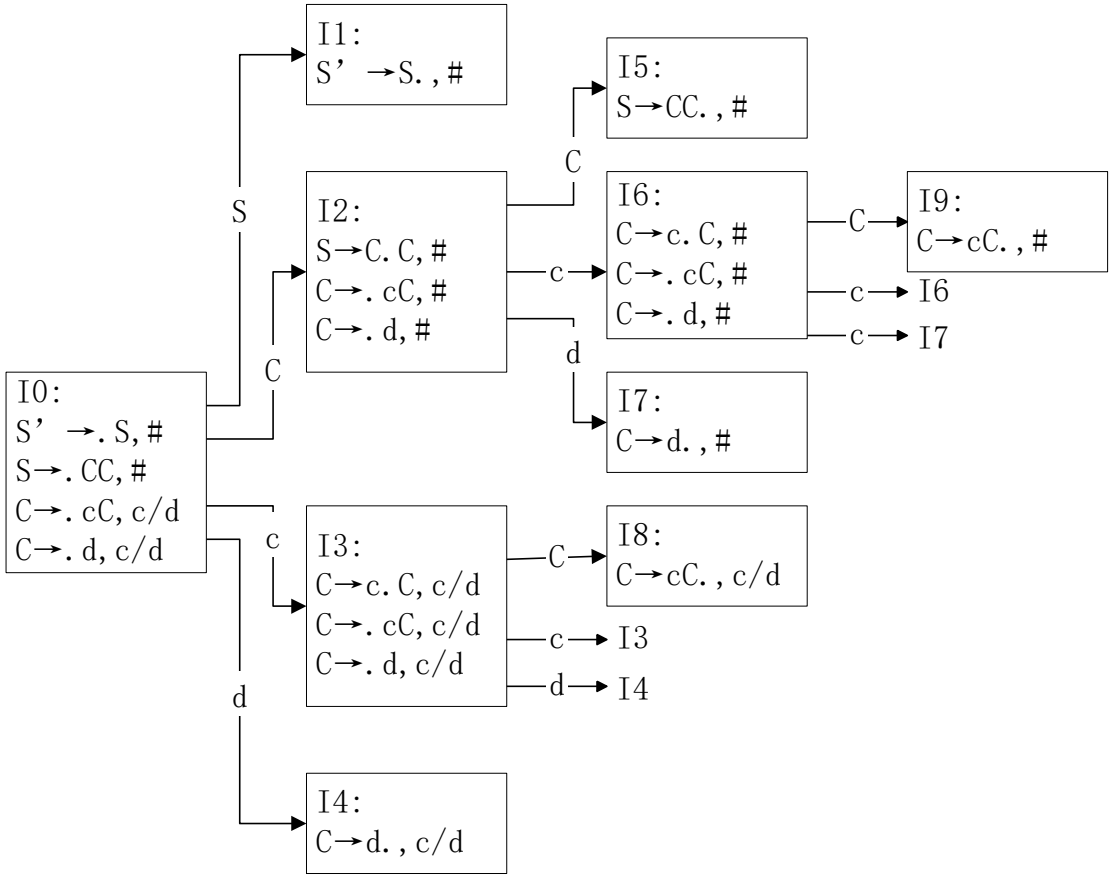
2 构造LR（1）的自动机，这里最重要的就是闭包中计算**向前看符号（逗号后面的符号）。**

解释：

比如在I0中计算闭包时：由“S->.CC,#” 闭包得到 “C->.cC, c/d ”和 “C ->.d, c/d”。

“C->.cC, c/d” 中 “c/d” 的来源：

因为在“S->.CC,#”中，第一个“C”为“A→α.Bβ,b”中的B，第二个“C”为“A→α.Bβ,b” 中的β。因此First（β） = First(C) = {c,d}， 所以向前看符号有两个： c/d。



3 构造LR（1）分析表

基本与LR(0)的构造方法一致，区别在于规约。

LR(1)：如果[A->α •, a]，且A≠S’，那么将ACTION[i,a]设置为规约A->α 。

比如I4中 “C->d. , c/d” ,则ACTION[4,c] = r3 (C->d)， ACTION[4,d] = r3 (C->d)

状态	ACTION	GOTO

	c	d	#	S	C
0	S3	S4		1	2
1			Acc		
2	S6	S7			5
3	S3	S4			8
4	r3	r3			
5			r1		
6	S6	S7			9
7			r3		
8	r2	r2			
9			r2		

第四章规范LALR语法 预测分析表构造例题详解

针对以上的LR（1），对应的合并。

LR（1）的项目集同心：除向前看符号外，项目集内容相同；

状态	ACTION			GOTO	
	c	d	#	S	C
0	S3，6	S4，7		1	2
1			Acc		
2	S3，6	S4，7			5
3，6	S3，6	S4，7			8，9
4，7	r3	r3	r3		
5			r1		
6	S6	S7			9
7			r3		
8，9	r2	r2	r2		
9			r2		

第五章 语法制导的翻译

由于不是考试重点，这里只列出一些相应概念，和11级的部分考试重点。
详细看一下书上5.1，5.2，5.3，理解一下

1）语法制导定义

语法制导定义（Syntax-Directed Definition,SDD）是一个上下文无关文法和属性及规则的结合。属性和文法符号相关联，而规则和产生式相关联。

非终结符的两种属性

①综合属性（Synthesized Attribute）：由子节点或本身的属性值来定义的。
②继承属性（Inherited Attribute）：由父节点，本身或兄弟节点的属性值来定义的。

终结符只有综合属性。

一个只包含综合属性的 SDD 称为 S属性(S-attribute)的 SDD。一个没有副作用的 SDD 有时也称为属性文法（attribute grammar）。

一个属性文法的规则仅仅通过其他属性值和常量来定义一个属性值.

我们对一棵语法分析树的某个结点的一个属性进行求值之前，必须首先求出这个属性值所依赖的所有属性值。

2）SDD的求值顺序

注释语法分析树（Annotated Parse Tree）：显示各个属性的值的语法分析树。例如教材例5.3（图5.5）

第五章SDD注释分析树例题详解

八、（8 分）考虑以下语法制导定义（SDD）：

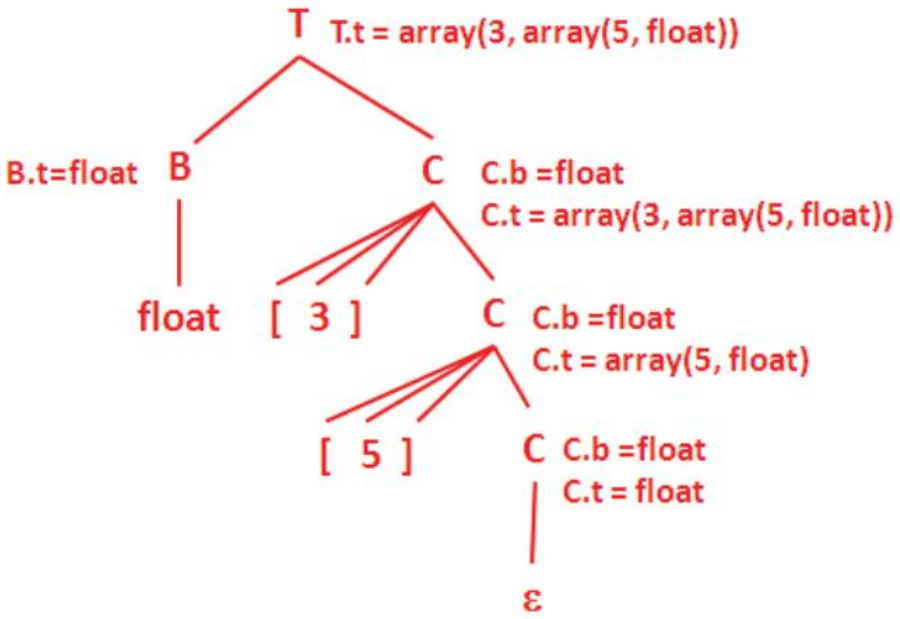
语法规则	语义规则
$T \rightarrow BC$	$T.t = C.t$ $C.b = B.t$
$B \rightarrow$ int	$B.t = integer$
$B \rightarrow$ float	$B.t = float$
$C \rightarrow$ [num] C_1	$C.t = array(\text{num}.val, C_1.t)$ $C_1.b = C.b$
$C \rightarrow \varepsilon$	$C.t = C.b$

注：假设 num 对应非负整数，num.val 代表这一非负整数的值.

1. **（2 分）在这一 SDD 中，哪些属性是综合属性？哪些属性是继承属性？**
2. **（6 分）对输入串 float[3][5]构造带注释的分析树（annotated parse tree）.**

【参考答案】

1. T.t，C.t 和 B.t 是综合属性，C.b 是继承属性.
2.



依赖图（Dependency Graph）：在语法分析树的基础上，显示属性求值的过程顺序图。例如教材例5.5（图5.7）。一般来说先自顶向下计算继承属性，再自底向上计算综合属性。

拓扑排序：

当只有综合属性存在时，只有一种顺序。

当有继承+综合属性，且无环时，存在多种顺序。

当存在环时，没有合法的顺序。

S属性的定义

如果一个SDD 的每个属性都是综合属性，它就是S 属性的。

L属性的定义

满足L 属性的SDD 的每个属性必须是其一：

①综合属性

②继承属性，但存在A->X1X2...Xn，其中Xi.a 继承属性只能与A 有关或是与Xi-1之前的文法符号有关，并保证不存在环。

3）语法制导的翻译方案

语法制导的翻译方案（Syntax-Directed Translation Scheme，SDT）是在其产生式体中嵌入了程序片段的一个上下文无关文法。这些程序片段称为语义动作。

两类重要的SDD

①基本文法用LR 技术分析，且SDD 是S 属性的（自底向上）

②基本文法用LL 技术分析，且SDD 是L 属性的（自顶向下）

后缀翻译方案

所有动作都在产生式最右端的SDT。

形如E -> E1 + T {E.val = E1.val + T.val; }

产生式语义动作

从SDT中消除左递归

消除左递归的技巧是对两个产生式A->Aα|β进行替换。替换为

A->βR

R->αR| ε

把一个L属性的SDD转换为一个SDT的规则

①把计算某个非终结符A 的继承属性的动作插入到产生式体中紧靠在A 的本次出现之前的位置上。

②将计算一个产生式头的综合属性的动作放在这个产生式体的最右端。

可以见书上例5.18，知道要把继承属性提前放在哪里就可以了。

4）实现L属性的SDD

L属性的SDD的自底向上语法分析

可以用自底向上额方法来完成任何自顶向下方式完成的翻译过程，技巧是：

①根据上面L 属性的SDD 转换为一个SDT 的规则，置放继承属性

②对于每个内嵌的语义动作，向这个文法引入一个标记的非终结符来替换它。每个这样的位置都有一个不同的标记，并且对于任意一个标记M 都有一个产生式M->ε 。

③如果标记非终极符M 在某个产生式A->α{a}β中替换了语义动作a，对a 进行修改到a'，并且将a'关联到M->ε上。

a'用到A 或α的值记为继承属性，其余均为综合属性。

见书上例子5.25，假设一个LL 文法中存在一个产生式A->BC，而继承属性B.i 是根据继承属性A.i 按照某个公式B.i = f(A.i)得到的，也即SDT 片段是 A->{B.i = f(A.i)} BC 引入标记M，有继承属性M.i 和综合属性M.s。

则这个SDT 可以写成

A->MBC

M->{M.i = A.i; M.s = f(M.i)}

由于有三道题来自冯老师对应章节的ppt，所以根据ppt和对应的龙书上的知识点进行总结。

以下根据冯老师第11（code generation） ，12（Intermediate code && local optimizations），14章ppt（register allocation）总结。

第11节 code generation 代码生成

利用简单的MIPS汇编指令集和堆栈机（stack machine）翻译

一、MIPS汇编指令架构

1 概述：

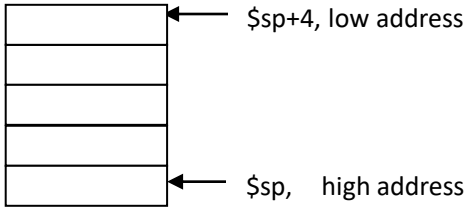
——32个32bits的通用寄存器：32 general purpose registers (32 bits each)

We will use $sp, $a0 and $t1 (a temporary register)

The accumulator is kept in MIPS register $a0

The address of the next location on the stack is kept in MIPS register $sp

The top of the stack is at address $sp + 4



——现场保护AR（Activation Record:

函数调用和函数的代码依赖于现场保护（AR）的设计。

AR的设计：

1）结果一直在 accumulator 中，则不需要现场保护。

2）现场保护实参 ：

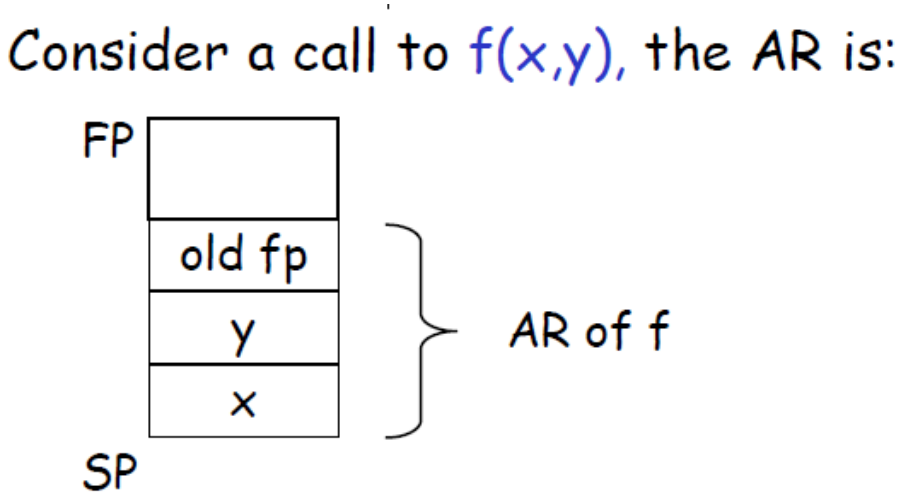
For f(x1,…,xn) push xn,…,x1 on the stack， These are the only variables in this language.

利用栈的机制，保证在进入函数和退出函数时 $sp 的值不变。所以函数调用需要返回地址。

——现场保护机制（利用fp（frame pointer））

fp 指针存储在 $fp 寄存器中

实例:



2 常用指令：

读写指令：

```
lw reg1 offset(reg2) // Load 32-bit word from address reg2 + offset into reg1
sw reg1 offset(reg2) // tore 32-bit word in reg1 at address reg2 + offset
```

基本算术运算指令：

```
add reg1 reg2 reg3  // reg1 ← reg2 + reg3
sub reg1 reg2 reg3  // reg1 ← reg2 - reg3
addiu reg1 reg2 imm  // reg1 ← reg2 + imm, “u” means overflow is not checked
li reg imm      // reg ← imm
```

跳转指令：

```
beq reg1 reg2 label   // Branch to label if reg1 = reg2
b label   // Unconditional jump to label
jr reg    // Jump to address in register reg
```

函数调用指令（现场保护）：

```
jal label  //Jump to label, save address of next instruction in $ra
jr reg     //Jump to address in register reg
```

二、 MIPS汇编代码生成方法

1 概述

对于每个表达式 e ，在$a0 寄存器中算出e的值，保存$sp寄存器和栈里的内容

定义： 代码产生函数 “cgen(e)” 的结果为表达式e生成的代码。

2 常见表达式的代码生成

1）常数i ：

```
cgen(i) = li $a0 i
```

2）加法

```
cgen(e1 + e2) =
        cgen(e1)
        sw $a0 0($sp)
        addiu $sp $sp -4
    cgen(e2)
    lw $t1 4($sp)
    add $a0 $t1 $a0
    addiu $sp $sp 4
```

3）减法

```
cgen(e1 - e2) =
            cgen(e1)
            sw $a0 0($sp)
            addiu $sp $sp -4
            cgen(e2)
            lw $t1 4($sp)
            sub $a0 $t1 $a0
            addiu $sp $sp 4
```

4）if语句

```
cgen(if e1 = e2 then e3 else e4) =
        cgen(e1)
        sw $a0 0($sp)
        addiu $sp $sp -4
        cgen(e2)
        lw $t1 4($sp)
        addiu $sp $sp 4
        beq $a0 $t1 true_branch // e1 = e2 则跳转到e3
        false_branch:
        cgen(e4)
        b end_if
        true_branch:
        cgen(e3)
        end_if:
```

5）函数调用

```
cgen(f(e1,…,en)) =
        sw $fp 0($sp)
        addiu $sp $sp -4
        cgen(en)
        sw $a0 0($sp)
        addiu $sp $sp -4
        …
        cgen(e1)
        sw $a0 0($sp)
        addiu $sp $sp -4
        jal f_entry
```

- The caller saves its value of the frame pointer
- Then it saves the actual parameters in reverse order
- The caller saves the return address in register $ra
- The AR so far is 4*n+4 bytes long

6）函数定义

```
cgen(def f(x1,…,xn) = e) =
        move $fp $sp
        sw $ra 0($sp)
        addiu $sp $sp -4
        cgen(e)
        lw $ra 4($sp)
        addiu $sp $sp z
        lw $fp 0($sp)
        jr $ra
```

Note: The frame pointer points to the top, not bottom of the frame

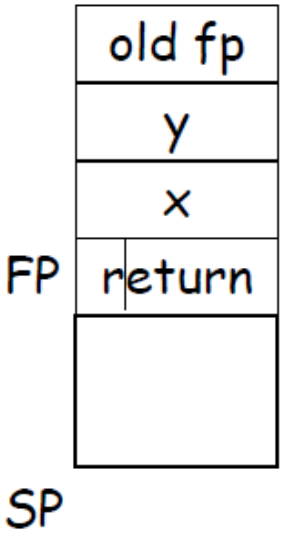
- The callee pops the return address, the actual arguments and the saved value of the frame pointer
- z = 4*n + 8

7）变量

函数的变量仅仅是函数的参数，这些参数都在AR中，被调用时压入栈。

Problem: 因为栈只在中间结果保存时才会增加，所以这些参数相对于sp指针的偏移位置时不确定的。

实例：



- X is at fp + 4
- Y is at fp + 8

第12节 code generation中间代码生成和Local Optimizations局部优化

中间代码生成

一、中间语言（intermediate languages）

中间语言与机器无关，等价于high-level的汇编语言。

特点：

使用任意标号寄存器名；

使用和汇编语言相似的控制结构；

使用较高层次操作码（opcode） 。

二、三地址代码（Three-Address Intermediate Code） （龙书 6.2节）

一条指令的右侧最多只有一个运算符的表达式。

三地址代码基于两个概念：地址和指令。

地址：

1、名字 Name (variables in source code)

- May be implemented as a pointer or reference to its entry in the symbol table.

2、常量 Constant

- Type conversions must be considered.

3、**编译器生成的临时变量** Compiler-generated temporary

- 优化 （Useful for optimization）.
  寄存器分配 （Register allocation）

指令：

x = y *op* z // arithmetic and logical

x = *op* y // negation and conversion

x = y // copy

- **goto** L // unconditional jump

if x **goto** L // conditional jump

if False x **goto** L // conditional jump

if x *op* y **goto** L // relational operation

- **param** x1 // parameter passing

param x2

...

param xn

call p, n // procedure call

y = **call** p, n // function call

return y // return a value

注意 ：四元式，三元式是三地址指令在某个数据结构中的表示方法

四元式表示：*op，arg1，arg2，result:*

其中op 包含运算符的内部编码，而result 字段用来表示临时变量。

例如，t1 = b * t2 ，其四元式依次为<*, b, t2, t1>

三元式表示：*op，arg1，arg2*

表示形式的唯一不同在于**arg2 是利用位置表示不同的临时变量的**，等于省下了表示临时变量的那个指针空间，但是在优化的时候可能会出现问题。因此**间接三元式（indirect triple）**另外未回一个指令列表来进行操作。

例如，t1 = minus c ，其三元式依次为<minus, c, >

t2 = b * t1 ，其三元式依次为<*, b, 0>(这里的0 代表上面那个式子)

中间语言举例：

id 表示寄存器，可以用常数替换id， op为常见运算符（+，-，*，/）

```
P→ S P | ε
S → id := id op id
| id := op id
| id := id
| push id
| id := pop
| if id relop id goto L
| L:
| jump L
```

三、基本块（basic block） 和 流图 （control-flow graph） （龙书8.4）

1 基本块概念

中间代码被划分为基本块，每个基本块是满足以下条件的**最大的连续三地址指令**

A 控制流只能从基本块的第一个指令进入该块，即不允许直接跳转到基本块的中间指令。

B 控制流在离开基本块之前不会停机（halt）或跳转只有，除了基本块最后一个指令可以停机或跳转。

基本块实例;

（1）. L:

（2）. t := 2 * x

（3）. w := t + x

（4）. if w > 0 goto L’

2 流图基本概念

流图的node ：

基本块形成流图的结点。

流图的edge：

基本块A到基本块B的有一条边当且仅当B中的第一个指令可能紧跟在A的最后一个指

令之后执行，有以下两种情况：

1） A的结尾有一个到B的开头的跳转语句

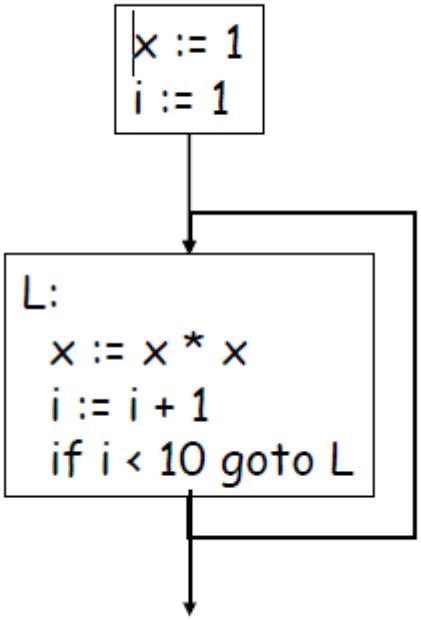
2） A的结尾不存在跳转语句，但是按照原来三地址语句序列，B紧跟在A后。

A称作B的前驱（predecessor），B称作A的后记（successor）。

通常流图还会增加入口（entry）和出口（exit）结点，不包含任何指令。

3 流图的表示

实例：



局部优化

1 概述

局部优化是指对基本块的优化。

优化的概念：

即对程序或中间代码进行各种等价变换，使得从变换后的程序出发，能生成更有效的目标代码。目的在于，既要设法缩小存储空间，又要尽量提高运行速度，而且常常偏重于提高运行速度。

优化可以分为全局优化和局部优化。局部优化是指在基本块上的优化。全局优化指的是在非线性程序块上的优化。

2 局部优化的常见方法（龙书 8.5, 9.1）

1）代数简化 algebraic simplification

a 可以被删除的冗余表达式：

x := x + 0

x := x * 1

b 可以化简的表达式：

x := x * 0 ⇒ x := 0

y := y ** 2 ⇒ y := y * y

x := x * 8 ⇒ x := x << 3

x := x * 15 ⇒ t := x << 4; x := t - x

2）常量合并（Constant Folding）

形如 x:= y op z ，如果y和z是常数的话，编译时就可以计算出y op z的值。

3）消除死代码 （即永远不会执行的代码和对程序结果没有影响的代码）

如：if 2 < 0 jump L。

控制流优化：在控制流中，也要删除不可到达的基本块

4）单赋值形式 （Single Assignment Form）

单赋值形式是指每个寄存器只在赋值操作左边出现了一次。

以下非单赋值形式的代码优化：

x := z + y

a := x

x := 2 * x

可以优化为：

b := z + y

a := b

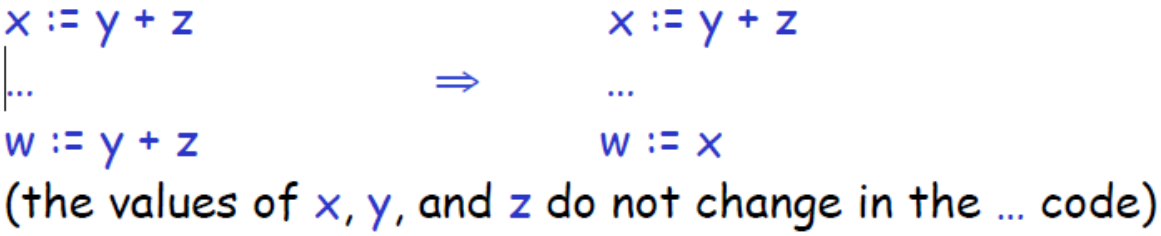
x := 2 * b

(b is a fresh register)

5）消除公共子表达式（Common Subexpression Elimination）

如果表达式E在某次出现之前已经被计算过，并且E中的变量值自从上次计算后没有改变过。那么E的这次出现就是一个公共子表达式。

如：



6）复制传播（copy propagation）

在单赋值形式前提下，如果有形如 w:=x 的赋值表达式，则在该块的后序子序列中将w替换为x。

如：

b := z + y b := z + y

a := b ⇒ a := b

x := 2 * a x := 2 * b

3 局部优化综合应用实例

初始代码：

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```

代数简化：

```
• a := x * x
b := 3
c := x
d := c * c
• e := b << 1
f := a + d
g := e * f
```

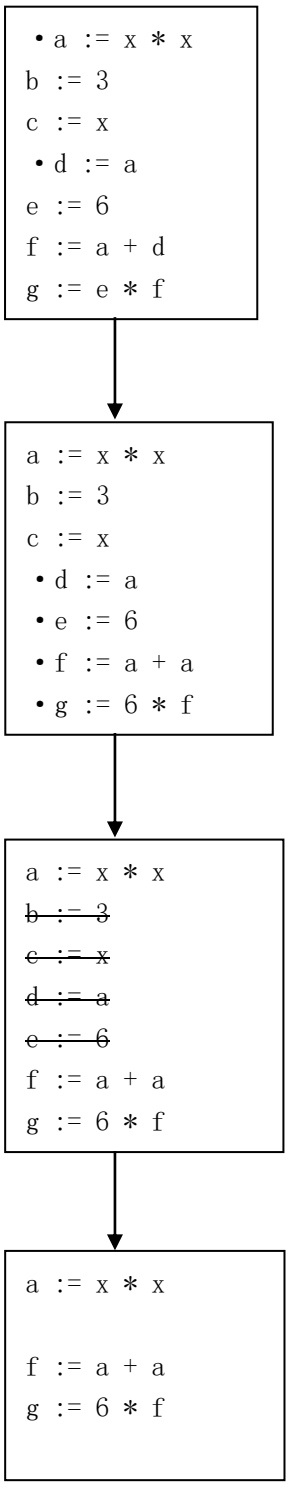
复制传播：

```
a := x * x
• b := 3
• c := x
• d := x * x
• e := 3 << 1
f := a + d
g := e * f
```

常量合并：

```
a := x * x
b := 3
c := x
d := x * x
• e := 6
f := a + d
g := e * f
```

消除公共
子表达式：


• a := x * x
b := 3
c := x
• d := a
e := 6
f := a + d
g := e * f
复制传播：
a := x * x
b := 3
c := x
• d := a
• e := 6
• f := a + a
• g := 6 * f
死代码消除：
a := x * x
b := 3
c := x
d := a
e := 6
f := a + a
g := 6 * f
最后结果：
a := x * x
f := a + a
g := 6 * f

第14节 Register Allocation寄存器分配

一、概述

由于寄存器的数量有限制，一般都比较少，所以无法为中间代码的每一个变量都从头到尾分配一个寄存器。所以需要进行**全局寄存器分配**，使得在有限个数寄存器的情况下，程序依然能够正常运行。

二、知识准备

1 计算流图中每一时刻的活跃变量

1）概述：

活跃的定义：如果一个变量的值在将来还需要使用，则称该变量是活跃的。

所以看一个变量的活性通常需要对一个基本块进行反向遍历（backward scanning回溯进行）

举例：

```
1 a=0;
2 b=a+1;
3 c=c+b;
4 a=b*2;
5 if(a<N)  goto  2;
6 return  c;
```

上述流程中：

b的活跃范围是{2->3, 3->4}。

a在{1->2, 4->5->2}活跃，但在{2->3->4}之间不是活跃的。

由于存在循环，c进入程序断时就是活跃的，直至程序断退出。

2）算法：Calculate liveness in a basic block:

从基本块的最后一个语句开始：

```
Initialize: for each variable v,
  v.liveness = v is temporary ? false : true; //临时变量活跃性为false
For each i: x = y + z
  Attach information of x, y and z to instruction i;
  x.liveness = false; //左值为false
  y.liveness = z.liveness = true;
```

注意： 流图中从最后面的基本块开始，每个基本块中从最后一个语句开始

2 图的 k-着色算法（NP完全问题）

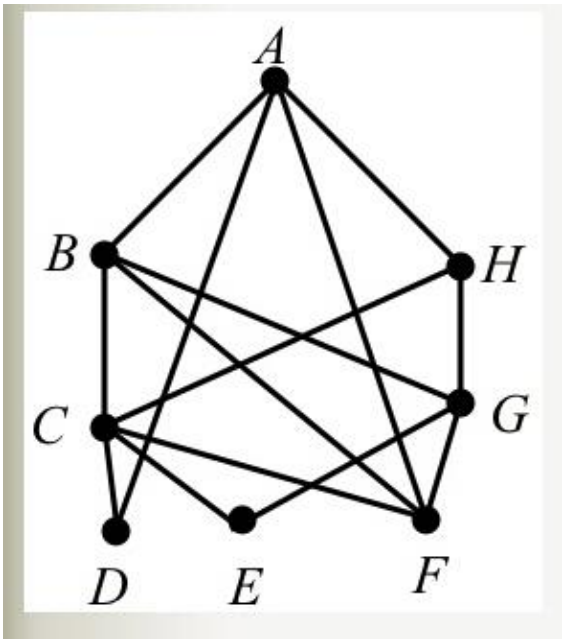
Welch Powell 法

a).将 G 的结点按照度数递减的次序排列.

b).用第一种颜色对第一个结点着色,并按照结点排列的次序，对与前面着色点不邻接的每一点着以相同颜色.

c).用第二种颜色对尚未着色的点重复步骤 b).用第三种颜色继续这种作法，直到所有点着色完为止。

举例：



解：

按度数递减次序排列各点

C A B F G H D E

•第一种颜色：C, A, G

•第二种颜色：B, H, D, E

•第三种颜色： F

所以图是三色的。另外图不能是两色的,因为图中有 A,B,F 两两相邻,所以 x(G)=3

三、图着色算法分配全局寄存器

1 算法思想

共享寄存器的前提：

设有临时变量t1,t2，只有当两个变量在程序的**任意时刻都不同时活跃（live）**时，这两个临时变量可以共享一个寄存器。

算法思想：

将寄存器分配问题抽象为图着色问题。即用**k种颜色（k个寄存器）给图中结点（临时变量）着色，使得任意相邻的两点颜色不同（即图中有边相连的两个临时变量分配到不同的寄存器）。**

2 算法基本步骤

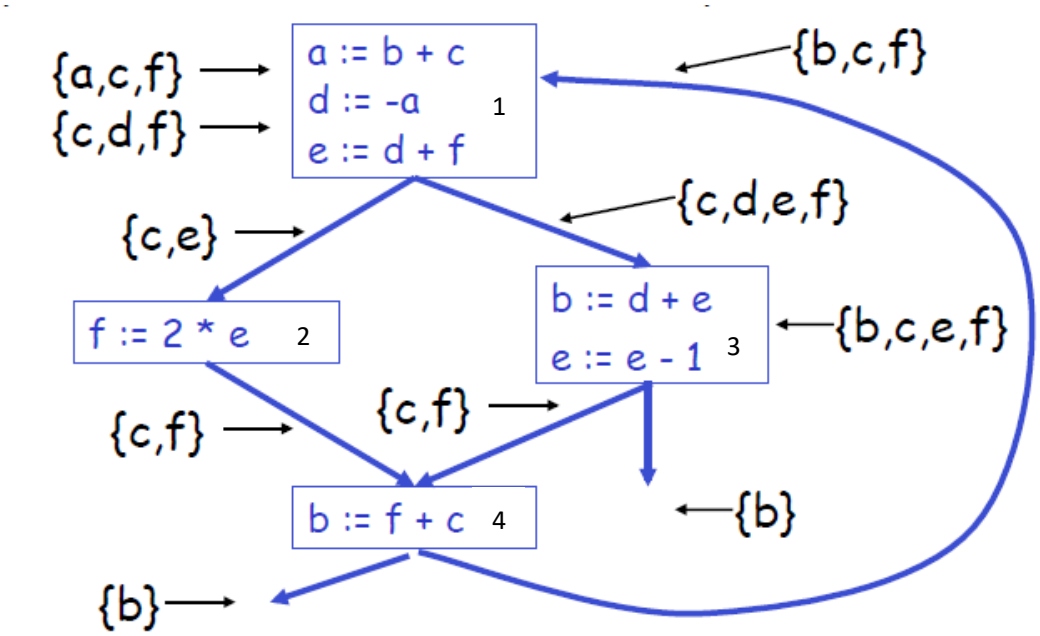
通过流图计算程序中每一时刻的活跃变量。

构造RIG（register-interference graph）寄存器冲突图。

根据图着色算法为每个临时变量分配对应的寄存器。

图着色分配寄存器例题详解 （根据冯老师ppt例题，有详细解释）

对以下流图使用图着色算法进行寄存器分配。限制使用4个寄存器（r1 ~ r4）：



1 计算流图中每个时刻的活跃变量

解释：

这是一个内层循环的流图（最后一个基本块又指向了第一个基本块）。

按照之前所说的算法，回溯分析，包括基本块之间是回溯的，基本块内部也是回溯的。

分析：

基本块中数字为序号，为了描述方便，用“number”表示该基本块

Step 1

从最后一个基本块4开始：

因为有“b:= f+c”，所以左值c，f为live变量。即2->4， 3->4 都有live{c,f}。

因为b为右值，所以dead。

这里有个内层循环，即从4->1：

从4->1，因为1 中需要的左值寄存器变量为b,c,f(d和a 都是暂时变量)。 所以4->1的live为 {b,c,f}

Step 2

回溯，4->2，因为2 中： “f:= 2*e”，f 成为右值，所以dead；e为左值，所以live。c依然为live，所以 1->2 的live为{c,e}

Step 3

回溯，4->3，在3中从最后一个语句回溯。

因为3到一个未知块有live{b}，到4有live{c,f}，3中最后一句为 “e:=e-1”，e为左值，e也活跃。

所以3中第一句到第二句的live为{b,c,e,f}

回溯，在3中第一个语句为“b:=d+e”,即d,e为live，b为dead。{b,c,e,f}变为{c,d,e,f}

所以 1->3的第一句，live为{c,d,e,f}

Step 4

回溯 2->1，3->1。从1中最后一个语句开始。

因为最后一句为e:=d+f，d,f为live，e为dead。根据1->2的live{c,f} 和 1->3的live{c,d,e,f}，c依然为live

所以1中第二句到第三句的live为{c,d,f}

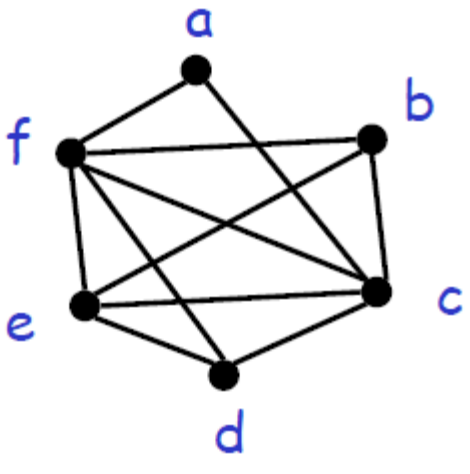
回溯，从第二句到第一句。

因为第二句为“d:=-a”,所以a为live，d为dead，根据第二句到第三句的live{ c,d,f }，c,f依然为live。

所以1中第一句到第二句的live为{c,a,f}

2 根据每个时刻的 live 构建 RIG

一旦两个变量在同一个集合中出现，他们之间就会有一条边。



3 根据图着色算法分配寄存器

Step 1

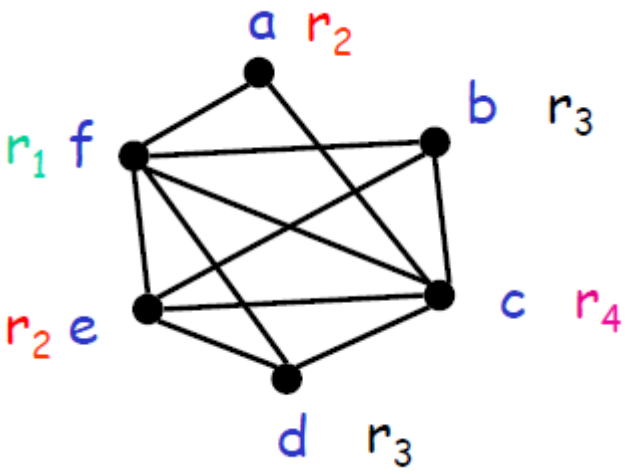
按照度数给节点排序： {f,e,d,c,b,a}

Step 2

根据之前的算法着色，这里用寄存器r1 ~ r4代表颜色

给 f 分配 r1,并且给未分配的所有图中与f不相邻的结点分配r1(这里没有)。
给 e 分配 r2，并且给给未分配的所有图中与e不相邻的结点:{a}分配r2。
给 d 分配 r3，并且给给未分配的所有图中与d不相邻的结点{b}分配r3。
给 c 分配 r4，并且给给未分配的所有图中与d不相邻的结点分配r4(这里没有)。
至此，分配完成。

分配后的图：



4 根据新分配的寄存器优化流图
对应变量用寄存器代替即可：