编译原理复习资料

特别提示：看作业题目

版本：0.1
日期：2012-01-10
整理：CodePlane（wdx&wx）
备注：请结合课本与作业还有 PPT 看。

一、引论

在一个程序可以运行之前，它首先要被翻译成一种能够被计算机执行的形式。完成这项翻译工作的软件系统称为编译器。

1) 语言处理器

简单地说，一个编译器就是一个程序，它可以阅读以某一种语言（源语言）编写的程序，并把该程序翻译成为一个等价的、用另一种语言（目标语言）编写的程序。编译器的重要任务之一是报告它在翻译过程中发现的源程序中的错误。

2) 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。

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。

分析部分经常被称为编译器的前端（front end）而综合部分称为后端（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。

- **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>
- **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）**。
- **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）**。
- 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**。

- **代码优化**：改进中间代码，提高运行速度，缩短代码长度。
- **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。

3) 乔姆斯基文法的判断

四种文法就是规定产生式的左边和右边的字符的组成规则的。判断的时候根据生成式左边和右边的特征来进行判断。

首先，应该明确，四种文法，从 0 型到 3 型，其规则和约定越来越多，限制条件也越来越多，所以我们判断时可以从最复杂的 3 型开始依次进行判断。

- **3 型文法（正则表达式）**

左边必须只有一个字符，且必须是非终结符；

其右边最多只能有两个字符，且当有两个字符时必须有一个为终结符而另一个为非终结符。当右边只有一个字符时，必须为终结符。

对于 3 型文法中的所有产生式，其右边有两个字符的产生式，这些产生式右边两个字符中终结符和非终结符的相对位置一定要固定，也就是说如果一个产生式右边的两个字符的排列是：终结符 + 非终结符，那么所有产生式右边只要有两个字符的，都必须前面是终结符而后面是非终结符。反之亦然，要么，就全是：非终结符 + 终结符。

- **2 型文法（上下文无关文法）**

与 3 型文法的第一点相同，即：左边必须有且仅有一个非终结符。

2 型文法所有产生式的右边可以含有若干个终结符和非终结符（只要是有限的就行，没有个数限制）。

2 型文法的特点就是可以递归，只要一个表达式是可以递归的，那么肯定不是 3 型文法。

- **1 型文法（上下文有关文法）**

1 型文法所有产生式左边可以含有一个、两个或两个以上的字符，但其中必须至少有一个非终结符。

与 2 型文法第二点相同。

1 型文法的特点在于可以较 2 型文法表达更多的东西。例如，对于

$$L1 = \{w2w \mid w = (0, 1)^*\}$$

可以看到这个文法能表示的字符串模式，是与 w 的内容有关的，例如如果 w=0 或 1，或者是多个相同的或是对称的字符串（如 000、111、101、010 等），那么能表示的字符串的为 020（121），可能表示的模式就是以 2 为中心对称的一个字符串也即 S->0S0|1S1|2。

但是如果 w=01，那么字符串就变成了 01201，与上面的字符串模式不同，也就是说，这个文法能够表示的语言形式，是与 w 的内容有关的，要想描述，文法的左边至少要有两个非终结符。

因此这个文法无法用上下文无关文法来表示。

- **0 型文法（图灵机）**

只要是人能够描述出来的东西，都属于这个类型，就不废话了。

最后取其最高的符合规则，来进行判断。

二、简单语法制导翻译器

这一部分是后面四章的一个总体介绍，许多基本但是关键的术语可以在这里找到。

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：

- 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。
- 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。
- 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。
- 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。

三、词法分析

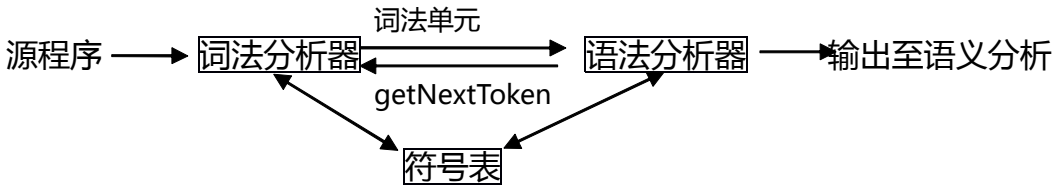
正则表达式是一种可以很方便地描述词素模式的方法。将对正则表达式进行转换：首先转换为不确定有穷自动机，然后再转换为确定有穷自动机。

正则表达式—>NFA—>DFA—>最小化

1) 词法分析器的作用

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，生成并输出一个词法单元序列，每个词法单元对应于一个词素。



- 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：一个词法单元名和一个可选的属性值组成。

模式：描述了一个词法单元的词素可能具有的形式。

词素：是源程序中的一个字符序列。

词法单元属性，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。

2) 词法单元的规约

- 串的术语

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。

- 语言上的运算

并，连接，闭包，正闭包。（注意长度！并的长度只有 1，连接的长度为 2）

- 正则表达式

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]
要懂得如何在自然语言和正则表达式之间转换，具体的见第一次作业！

3) 词法单元的识别

状态转换图（transition diagram）有一组被称为状态（state）的结点或圆圈。状态图中的边（edge）从图的一个状态指向另一个状态。每条边的标号包含了一个或多个符号。

- 状态转换的重要约定

某些状态称为**接受状态**或**最终状态**，表明已经找到了一个词素。有一个状态被指定为**开始状态**，也称**初始状态**，该状态由一条没有处罚结点的、标号 start 的边指明。

4) 有穷自动机（finite automata）

有穷自动机是**识别器（recognizer）**，只能对每个可能的输入串简单地回答“是”或“否”。

有穷自动机分为两类：

不确定的有穷自动机（Nondeterministic Finite Automata，NFA），对其边上的标号没有任何限制。一个符号标记离开同一状态的多条边，并且空串 ε 也可以作为标号。

确定的有穷自动机（Deterministic Finite Automata，DFA），对于每个状态及自动机输入字母表中的每个符号，有且只有一条离开该状态、以该符号为标号的边。

确定的和不确定的有穷自动机能识别的语言的集合是相同的，这个集合中的语言称为**正则语言（regular language）**。不管是 NFA 还是 DFA，我们都可以将它表示为一张**转换图（transition graph）**。图中的结点是状态，带有标号的边表示自动机的转换函数。双圈表示接受状态

- 不确定的有穷自动机的组成：

一个有穷的状态集合 S；
一个输入符号集合Σ，即输入字母表（input alphabet）；
一个转换函数 (transition function）为每个状态和符号给出后继状态 (next state）的集合。
S 中的一个状态 s0 被指定为开始状态，或者说初始状态。
S 的一个子集 F 被指定为接受状态（或者说终止状态的）集合。

- 转换表

也可以将一个 NFA 表示为一张转换表（transition table），表的各行对应于

状态，各列对应于输入符号和ε。对应于一个给定状态和给定输入的条目是将 NFA 的转换函数应用于这些参数后得到的值。

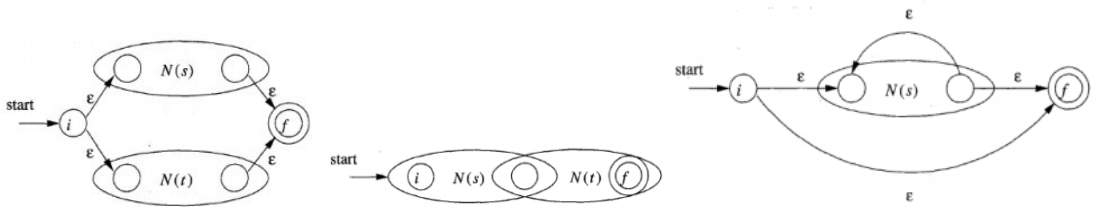
- 确定的有穷自动机

是不确定有穷自动机的一个特例，**DFA 没有输入ε之上的转换动作，对于每个状态 s 和每个输入符号 a，有且只有一条标号为 a 的边离开 s**。一个 NFA 总可以变成 DFA，DFA 总可以被最小化。

5) 正则表达式—>NFA—>DFA—>最小化

- **正则—>NFA**：

并：　　连接：　　闭包：



- **NFA—>DFA**：

算法：

①找到ε-closure(0)，即状态集 A 作为开始状态；
②for (每个输入符号 a)　计算ε-closure(move(A, a)) ,成为一个新的状态集 B...；
③重复步骤②直到没有新的状态集产生；

ε-closure (s) 的算法：
for (每个 s 集合中的符号 a) 如果 a 接收ε到达 b，就把 b 也加进闭包中。

move (T , a) 的算法：
能够从 T 中某个状态 s 出发通过标号为 a 的转换到达的 NFA 状态的集合。

- **DFA—>最小化**：

任何正则语言都有一个唯一的状态数目最少的 DFA。
目的：①消除死状态 ②消除冗余
算法：

①初始划分状态集为非接受状态组和接收状态组；
②划分每一个组直至不能分割，最坏情况可能各自为组。若两个状态 s 和 t 在同一小组中当且仅当对于所有的输入符号 a，状态 s 和 t 在 a 上的转换都到达已分组中的同一组，就把不同状态分成小组。由于这个描述起来太抽象，直接举个例子：{A、B、C}、{D}两个组，A、B 接收状态 a 后都转到组{A、B、C}中，而 C 接收状态 a 后转到{D}组，则 C 独立成组，结果变成{A、B}、{C}、{D}。
③从最后分好的组中每个组选取一个状态作为该组的代表，然后画出最小的 DFA

ATT：具体习题见第二次作业！

四、语法分析

↑ 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。

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

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）。

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。

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。

ATT：要懂得自然语言，正则表达式，上下文无关文法之间的等价转换，看第六周作业！

2) 设计文法

- 消除二义性：

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。

- 左递归的消除：

A -> Aα | β　　替换为　A -> βA`
A` -> αA` | ε

- 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 -> $\alpha\beta_1 | \alpha\beta_2 | ... | \alpha\beta_n | \gamma$　替换为　A -> αA' | γ
A' -> $\beta_1 | \beta_2 | ... | \beta_n$

3) 自顶向下的语法分析

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。

- **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->$Y_1Y_2...$是一个产生式，若ε在 FIRST(Y_1) ... FIRST(Y_{i-1})中，则 FIRST(Y_i)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。

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：

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, $]。

语法分析的详细过程可以参考书上例 4-21。

ATT：以上算法都很抽象，关键是结合例子学会怎么算，看第七周作业。

4) 自底向上的语法分析

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。

规约(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

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：

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。

5) LR(0)文法&SLR(1)文法

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。

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。

6) LR(1)文法&LALR(1)文法：

项集的闭包算法：

```
for( I 中每个项 [A->α·Bβ, a] )
    for( G'中每个产生式 B->γ )
        for( FIRST(βa) 中的每个终结符 b)
            将[B->·γ,b]加入到集合中
```

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。

ATT：跟 LL(1)一样，这 4 种 LR 文法重点还需要自己体会一下，结合例子学会怎么算，看第九周作业。

- LL 和 LR 都通用的解题步骤：

①算 FIRST 集合 FOLLOW 集
②画自动机，写语法分析表
③具体分析过程

- 判断文法顺序：

LR(0) -> SLR(1) -> LR(1) -> LALR(1)
s 和 r 冲突 合并相同项集不冲突
前两个是用同一个自动机，后两个是用另一个自动机。

五、语法制导的翻译

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）

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。

ATT：要会画注释语法分析树和依赖图，见第十周作业。

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 $A->X_1X_2...X_n$，其中 Xi.a 继承属性只能与 A 有关或是与 X_{i-1} 之前的文法符号有关，并保证不存在环。

ATT：要会根据产生式写语义规则，SDD 的表，见第十一周作业。

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 $E \rightarrow E_1 + T \quad \{E.val = E_1.val + T.val;\}$
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)}

六、中间代码生成

静态检查包括**类型检查（type checking）**，类型检查保证运算符被应用到兼容的运算分量。
将给定源语言的一个程序翻译成特定的目标机器代码的过程中，一个编译器可能构造出一系列中间表示。高层的表示接近于源语言，而地层的表示接近于目标机器。语法树是高层的表示，它刻画了源程序的自然的层次性结构，并且适用于静态类型检查这样的处理。

1) DAG

语法树中的各个节点代表了源程序中的构造，一个节点的所有子节点反映了该节点对应构造的有意义的组成成分。为表达式构建的**有向无环图（Directed**

Acyclic Graph，DAG）指出了表达式中的公共子表达式。

可以根据表达式直接写出 DAG，但是需要注意同一个子元素只能出现一次，画图的时候需要注意了。（具体可见例 6.1）

2）三地址代码

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。

- 四元式表示：*op*，arg_1，arg_2，*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，$t_1 = b * t_2$ ，其四元式依次为 <*, b, t_2, t_1>

- 三元式表示：*op*，arg_1，arg_2

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，t_1 = minus c ，其三元式依次为 <minus, c, >

$t_2 = b * t_1$ ，其三元式依次为 <*, b, 0>(这里的 0 代表上面那个式子)

3）表达式的翻译

表达式中的运算：使用 S 的属性 code 和 E 的属性 addr 和 code，为一个语句生成三地址代码。属性 S.code 和 E.code 分别表示 S 和 E 对应的三地址代码。属性 E.addr 则表示存放 E 的值的地址，一个地址可以是变量名字、常量或编译器产生的临时量。

详见书上图 6-19

将数组元素存储在一块连续的存储空间里就可以快速地访问它们。假设每个数组元素的宽度是 w，那么数组 A 的第 i 个元素的开始地址为 base+i×w，其中 base 是 A[0]的相对地址。对于多维的数组也类似。

一个二维数组通常有两种存储方式，即**按行存放**和**按列存放**。

ATT：主要掌握例 6.12，数组元素的寻址，看第十四周作业第一题

4）控制流

回填技术可以用来在一趟扫描中完成对布尔表达式或控制流语句的目标代码生成。非终结符号 B 的综合属性 truelist 和 falselist 将用来管理布尔表达式的跳转代码中的标号。

此文法中有一个标记非终结符号 M。它引发的语义动作在适当的时刻获取将要生成的下一条指令的下标。

B->B1 || M B2 | B1 && M B2 | !B1|(B1) | E1 rel E2 | true | false

就是 M 是用来索引下一个判断指令的标号的。

例如对于表达式 x < 100 || x > 200 && x != y

一开始生成的语句分别为

100: if x < 100 goto _

101: goto _

102:if x > 200 goto _

103: goto _

104: if x != y goto _

105: goto _

并且根据规则，需要在不同语句的 goto 后面填写上具体要跳转到的代码位置。例如在第 102 句，因为是&&的判断，x> 200 后还需要判断下一部分，所以需要在 goto 后面填写 104（注释分析树中的这里的 M 为 104），然后因为一开始是一个||判断，所以只有在 x < 100 不成立的时候，才需要跳转，也就是说

需要把 101 行的 goto 添加 102（注释分析树中的这里的 M 为 102），以指示下一条指令在哪里。注意只有&&和||操作需要回填，其他的仍旧保留为下划线。最后得到的结果应该是：

100: if x < 100 goto _
101: goto 102
102:if x > 200 goto 104
103: goto _
104: if x != y goto _
105: goto _

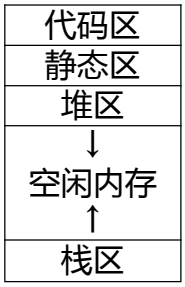
ATT：掌握 truelist,falselist，具体看第十四周作业第二题和例 6.24

七、运行时刻环境

编译器创建并管理一个运行时刻环境（run-time environment），它编译得到的目标程序就运行在这个环境中。

1) 栈分配

每当一个过程（函数、过程、方法和子例程的统称）被调用时，用于存放该过程的局部变量的空间被压入栈；当这个过程结束时，该空间被弹出这个栈。这种安排不仅允许活跃时段不交叠的多个过程调用之间共享空间，而且允许我们以如下方式为一个过程编译代码：它的非局部变量的相对地址总是固定的，和过程调用的序列无关。



了解**控制链**是指向调用者的活动记录

2) 垃圾回收

不能被引用的数据通常被称为**垃圾（garbage）**。垃圾回收是重新收回那些存放了不能再被程序访问的对象的存储块。

人工存储管理很容易出问题。常见的错误有两种形式：一直未能删除不能被引用的数据，称为**内存泄露（memory-leak）**；引用已经被删除的数据，称为**悬空指针引用（dangling-pointer-dereference）**错误。自动垃圾回收通过回收所有垃圾而消除了内存泄露问题。

- **引用计数的垃圾回收器**

简单但有缺陷，每个对象都有一个用于存放引用计数的字段。当一个对象从可达变成不可达的时候，该回收器就可以将该对象确认为垃圾；当一个对象的引用计数为 0 时，该对象会被删除。

主要缺点是①不能回收不可达的循环数据结构②开销较大

- **基本的标记-清扫式回收器（基于跟踪）**

找出所有不可达的对象，并将它们放入空闲空间列表，开销较大。

- **标记压缩垃圾回收器（基于跟踪）**

在本地压缩对象，降低存储需求。

- **复制回收器（基于跟踪）**

主要缺点是复制的开销很大、只有一半的堆内存会使用到。

八、代码生成

代码生成器必须有效地利用目标机器上的可用资源且高效运行。实践中，我们必须使用那些能够产生良好但不一定最优的代码的启发技术。

代码生成器有三个主要任务：①指令选择 ②寄存器分配和指派 ③指令排序

1) 目标语言

- **程序和指令的代价**

①指令 LD R0，R1 把寄存器 R1 中的内容复制到 R0 中。因为不要求附加的内存字，代价是 1。
②指令 LD R0，M 把内存位置中的内容 M 加载到寄存器 R0 中。因为内存位置 M 的地址在紧跟着指令的字中，代价是 2。
③指令 LD R1，*100（R2） 把值 contents(contents(100+contents(R2)))加载到寄存器 R1 中。代价是 2。

2) 基本块和流图

- **基本块**

每个基本块是满足下列条件的最大的连续三地址指令序列。
①控制流只能从基本块中的第一个指令进入该块。也就是说，没有跳转到基本快中间的转移指令。
②除了基本块的最后一个指令，控制流在离开基本块之前不会停机或者跳转。

- **划分基本块算法**

主要就是选择首指令，根据首指令到下一个首指令之前（不含）形成一个基本块。
①中间代码第一个三地址指令，是首指令
②任意一个条件或无条件转移指令的**目标指令**，是首指令
③紧跟在一个条件或无条件转移指令**之后的指令**，是首指令

- **在每一基本块中，确定活跃性（后续使用信息）**：

①假设所有的非临时变量都是活跃
②从块中最后一个语句开始反向扫描，对于每句 i: x = y + z 将 x 设为“不活跃”，y 和 z 设置为“活跃”，并把他们的下一次使用设置为语句 i。（必须按顺序，先设不活跃，再设活跃！）

- **流图**

根据上面划分的基本块画出流图，用箭头把 goto 语句指向对应的语句即可，在头尾分别加上 ENTRY 和 EXIT 块，具体见书上例 8.8 的图。

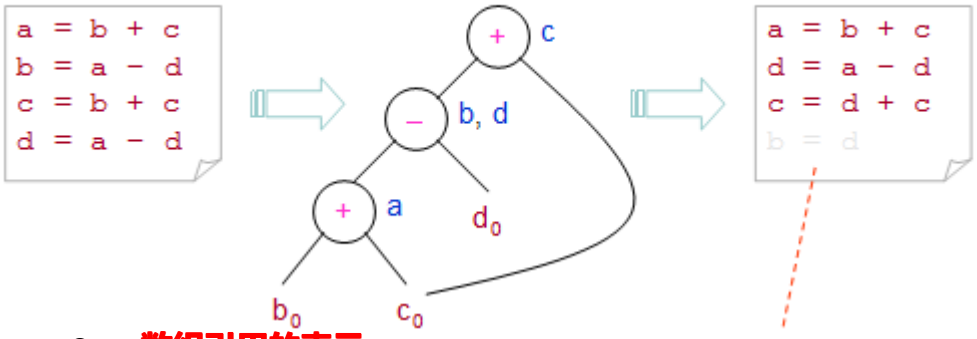
3) 基本块的优化

仅仅通过对各个基本块本身进行**局部**优化，我们就常常可以实质性地降低代码运行所需的时间。更加彻底的**全局**优化将是下一章的内容。

主要采用三种方法：①消除局部公共子表达式（重复计算一个已经计算得到的值的指令）②消除死代码（计算得到的值不会被使用的指令） ③重新排序

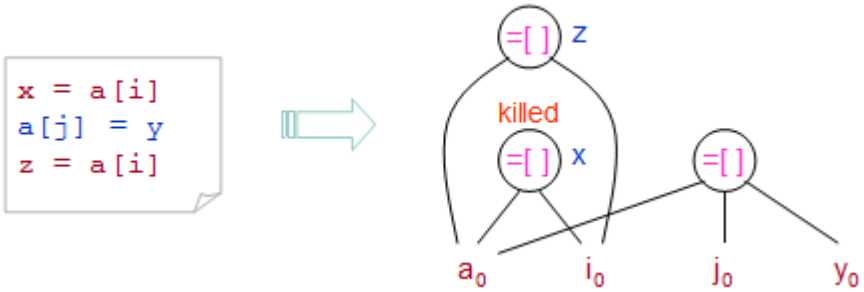
- **消除局部公共子表达式条件**

①表达式 b 不活跃
②存在另一个表达式与 b 等价
例：



- **数组引用的表示**

数组比较特别，应当把整个数组视为一个整体，每次都必须重新进行计算，不然可能会像下面一样得到预期外的效果。下面先是对 x 赋值，然后一旦要对数组中的元素进行赋值，那么 x 的那个结点就被杀死了，杀死了的意思是在考虑公共子表达式的时候不再考虑它，所以在 z 的时候，还需要重新建立一个新的结点。例：



九、机器无关优化

大部分全局优化是基于数据流分析（data-flow analyse）技术实现的。数据流分析技术是一组用以收集程序相关信息的算法。

只需要看一个例子，就是 Lecture12 的 ppt 的 P6-12。