编译原理笔记

Pionpill * Nekasu †

本文档为作者学习《编译原理》‡一书时的笔记，

2022 年 1 月 6 日

前言：

本篇笔记用于应付学校期末考试，本人对这方面也没有进行深入研究。内容浅显，不适合考研等深入学习。

编译原理这门课挺难的，本文只能提供一个基于原教程的精炼与复习，针对晦涩的部分或者跳过的理解性例子，会在对应页下给出注释，但有些东西笔者也不是很理解，望谅解。

本文使用 LaTeX 语言书写，原文编译环境为 TeXLive + VSCode。文中大部分数字编号，超链接，目录均可实现点击跳转。若无法实现基础的目录点击跳转，那可能是你的 pdf 阅读器有问题，建议使用免费的 Adobe Acrobat Reader DC[1]或 Chrome[2] 浏览器。得益于 LaTeX 的强大排版功能以及 TikZ 绘图库，本文所有图片文字均为矢量形式，如有不清楚地方放大即可，如果放大了还不清楚，那你可能拿到的是 n 手资源。

此外，本篇笔记是对原书的提炼总结，只能辅助原书进行学习，有大量例子等理解性文字并未进行说明，如有需要请购买原书。本笔记文案多为原书摘抄或个人总结，图片为本人使用 TikZ 绘制，若需进行引用，可前往下载 LaTeX 源代码[3]，并遵守 GPL-v3 协议。

本篇笔记换过一次仓库，之前的版本请前往：`https://github.com/Pionpill/Notebook/tree/Pionpill/Lessons`

针对学校考试，红色为重要 (必考)，蓝色为次重点，绿色为理解性内容，灰色为上课跳过或者不重要的内容。

由于原书错误太多，而且内容过于形式化，第四章开始，本文主要参考哈工大编译原理课程`https://www.bilibili.com/video/BV1zW411t7YE`。

2022 年 1 月 6 日

*笔名：北岸，电子邮件：673486387@qq.com，Github：`https://github.com/Pionpill`

†笔名：小十六的伏特加，电子邮箱：1428147954@qq.com，Github：`https://github.com/Nekasu`

‡《程序设计语言编译原理》：陈火旺，国防工业出版社，2020 年印刷

[1]下载连接 (pro 为付费版本)：`https://get.adobe.com/cn/reader`

[2]以及使用 chrome 核心的 edge 浏览器，小心国产浏览器

[3]`https://github.com/Pionpill/Notebook/tree/Pionpill/Lessons`

目录

一、引论

1.1 什么是编译原理

计算机上执行一个高级语言程序通常分为两步：第一步，用一个编译程序把高级语言翻译成机器语言程序；第二部，运行所得到的机器语言程序求计算结果。

通常所说的翻译程序是这样一个程序，它能够把某一种语言程序 (称为源语言程序) 转换成另一种语言程序 (称为目标语言程序)，前后者逻辑上是等价的。这样的一个翻译程序就称为编译程序。

高级语言除了先编译后执行外，有时也可 “解释” 执行。一个源语言的解释程序就是这样的程序，它以该语言写的源程序作为输入，但不产生目标程序，而是边解释边执行源程序本身。

1.2 编译过程概述

编译程序的工作过程一般可分为五个阶段：词法分析，语法分析，语义分析与中间代码产生，优化，目标代码生成。

词法分析

词法分析的任务是：输入源程序，对构成源程序的字符串进行扫描和分解，识别出一个个单词。这一过程类似于英文翻译中认识每一个单词的意义。

语法分析

语法分析的任务是：在词法分析的基础上，根据语言的语法规则，把单词符号串分解成各类语法单位，如 “短语”，“子句”，“句子”,“程序段” 等。通过语法分析，确定整个输入串是否构成语法上正确的 “程序”。语法分析所依循的是语言的语法规则。语法规则通常用上下文无关文法描述。词法分析是一种线性分析，而语法分析是一种层次结构分析。

语义分析和中间代码产生 (例子见书 P3)

这一阶段的任务是：对语法分析所识别出的各类语法范畴，分析其含义，并进行初步翻译 (产生中间代码)。这一阶段通常包括两个方面的工作。首先，对每种语法范畴进行静态语义检查。如果语义正确，则进行另一方面工作，即进行中间代码的翻译。这一阶段所依循的是语言的语义规则。通常使用属性文法描述语义规则。

“翻译” 仅仅在这里才开始涉及到。所谓的 “中间代码” 是一种含义明确，便于处理的记号系统，它通常独立于具体的硬件。

优化

优化的的任务在于对前端产生的中间代码进行加工变换，以期在最后阶段能产生更为高效 (省时间省空间) 的目标代码。优化的主要方面有：公共子表达式的提取，循环优化，删除无用代码等等。有时为了便于 “并行运算”，还可对代码进行并行化处理。优化所依循的原则是程序的等价变换规则。

目标代码生成

这一阶段的任务是：把中间代码 (优化处理过后) 变换成特定机器上的低级语言代码。这阶段实现了最后的翻译，它的工作有赖于硬件系统结构和机器指令含义。

目标代码的形式可以是绝对指令代码或可重定位的指令代码或汇编指令代码。如目标代码是绝对指令代码，则这种目标代码可立即执行。如目标代码是汇编指令代码，则需汇编器汇编后才能进行。

现代多数编译程序产生的是可重定位的指令代码。这种目标代码在运行前必须借助一个连接装配程序把各个目标模块 (包括系统提供的库模块) 连接在一起。确定程序变量 (常数) 在主存中的位置，装入内存中指定的起始地址，使之称为一个可运行的绝对指令代码程序。

1.3 编译程序的结构

1.3.1 编译程序总框

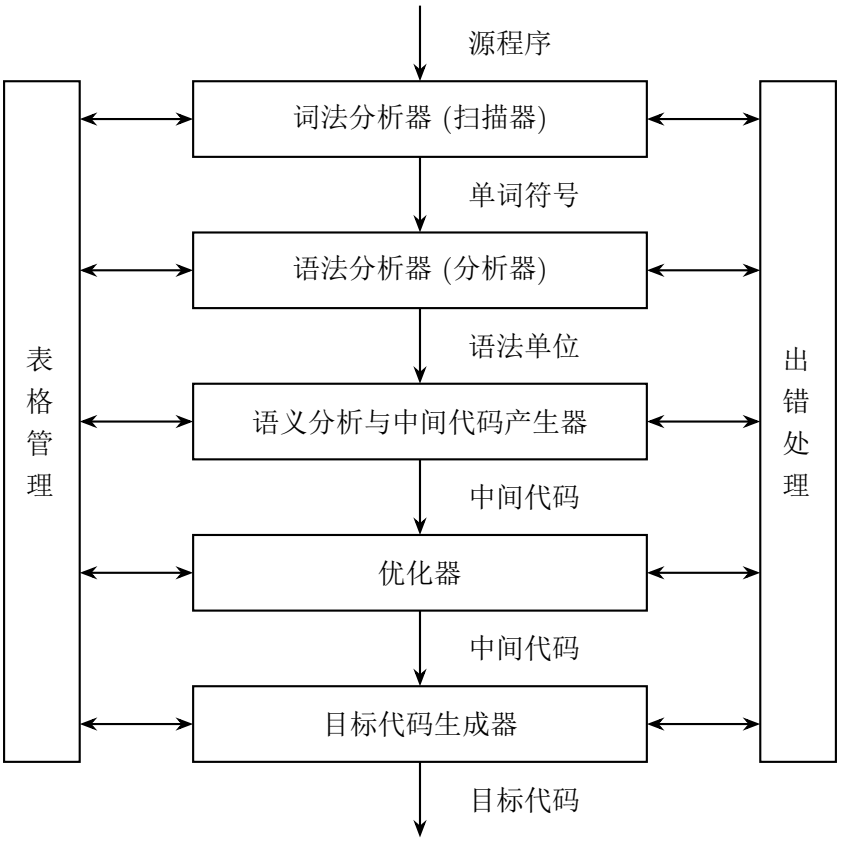


图 1.1　编译程序总框

有的编译程序在识别出各类语法单位后，构造并输出一棵表示语法结构的语法树，然后，根据语法树进行语义分析和中间代码产生。还有许多编译程序在识别出语法单位后并不真正构造语法树，而是调用相应的语义子程序。在这种编译程序中，扫描器，分析器和中间代码产生器并非截然分开，而是相互穿插。

1.3.2 表格与表格管理

编译程序在工作过程中需要保持一系列的表格，以登记源程序的各类信息和编译各阶段的进展情况。在编译程序使用的各种表格中，最重要的是符号表。它用来登记源程序中出现的每个名字以及名字的各种属性。

编译各阶段都涉及到构造，查找或更新有关表格。

1.3.3 出错处理

编译程序应能对出现在源程序中的错误进行处理。这部分工作由专门的一组程序 (出错处理程序) 完成。一个好的编译程序应找到错误，并最小化错限制错误造成的影响，使得源程序的其他部分能够继续被编译，以便进一步发现其他可能的错误。

编译过程的每一阶段都可能检测出错误，绝大部分错误出现在编译的前三阶段。源程序中的错误通常可分为语法错误与语义错误两大类。

- 语法错误

  指源程序中不符合语法 (词法) 规则的错误，可在语法分析与词法分析时检测出来。例如非法字符，括号不匹配等。

- 语义错误

  指源程序中不符合语义规则的错误，一般在语义分析时检测出来，有的语义错误需要在运行时检测出来。例如类型不一致，作用域错误等

1.3.4 遍

编译过程具体实现时，受不同限制，往往将编译程序组织为若干遍 (pass)。所谓“遍”就是对源程序或源程序中间结果从头到尾扫描一次，并作有关的加工处理，生成新的中间结果或目标程序。通常，每遍的工作由从外存上获得的前一遍地中间结果开始，完成它所含的有关工作之后，再把结果记录于外存。既可以将几个不同阶段合为一遍，也可以把一个阶段的工作分为若干遍。

遍数多一点有一个好处，即整个编译程序的逻辑结构可能清晰一点。但遍数多势必增加输入/输出所消耗的时间。

1.3.5 编译前端与后端

概念上，我们有时把编译程序分为编译前端和编译后端。前端主要由与源语言有关但与目标无关的部分组成。包括：词法分析，语法分析，语义分析与中间代码产生，有的代码优化工作也能包含在内。[4]后端包括编译程序中与目标机有关的那部分，如与目标机有关的代码优化和目标代码生成等。通常，后端不依赖于源语言而仅依赖于中间语言。

[4]考试时，代码优化属于编译后端。

可以取编译程序的前端，改写其后端以生成不同目标机上的相同语言的编译程序。也可以将几种源语言编译成相同的中间语言，然后为不同的前端配上相同的后端。

1.4 编译程序与程序设计环境

程序设计环境：编译程序与程序设计工具一起构成。程序设计工具包括：编辑程序，连接程序，调试工具等。

集成化的程序设计环境：特点是将相互独立的程序设计工具集成起来，以便为程序员提供完整的，一体化的支持，从而进一步提高程序开发效率，改善程质量。

下面以 Ada 语言的程序设计环境 APSE 为例，介绍程序设计环境的基本构成与主要工具。

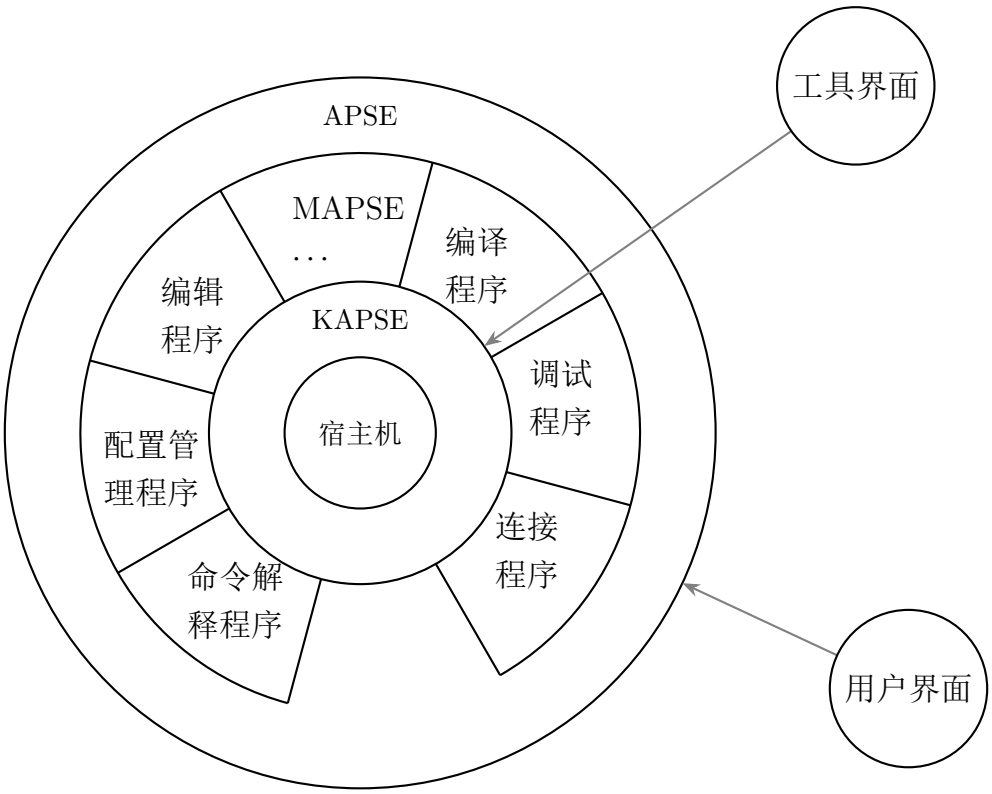


图 1.2　Ada 程序设计环境

最内层是宿主计算机系统,包括硬件,宿主操作系统和其他支撑软件;第一层是核心 APSE (KAPSE)，包含环境数据库，通信即运行时支撑功能等。第二层，最小 APSE (MAPSE)，包含 Ada 程序开发及维护的基本工具，这些工具包括编译程序，编辑程序，连接程序，调试程序等。第三层，APSE，在 MAPSE 外面再机上更广泛的工具构成完整的 APSE。

在一个程序设计环境中，编译程序起着中心作用。连接程序，调试程序等的工作直接依赖于编译程序所产生的结果。而其他工具的构造常常要用到编译的原理，方法和技术。

1.5 编译程序的生成

我们用一种 T 形图来表示源语言 S，目标语言 T 和编译程序实现语言 I 之间的关系。

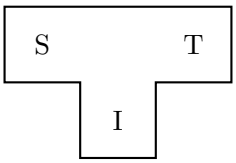


图 1.3 T 形图

如果 A 机器上已有一个用 A 机器代码实现的某高级语言 L_1 的编译程序，则我们可以用 L_1 语言编写另一种高级 L_2 的编译程序，把学好的 L_2 编译程序经过 L_1 编译程序编译后就可得到 A 机器代码实现的 L_2 编译程序。

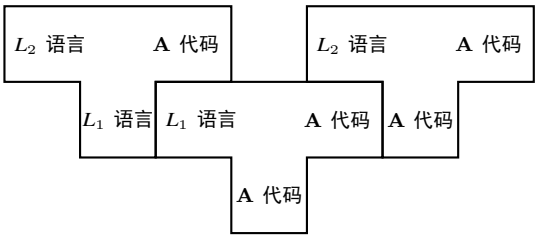


图 1.4 用 L_1 语言编写编译程序

采用一种所谓 “移植” 方法，我们可以利用 A 机器上已有的高级语言 L 编写一个能够在 B 机器上运行的高级语言 L 的编译语言，然后把该源程序经过 A 机器上的 L 编译程序编译后得到能在 A 机器上运行的产生 B 机器代码的编译程序，用这个编译程序再一次编译上述编译程序源程序就得到了能在 B 机器上运行的产生 B 机器代码的编译程序。

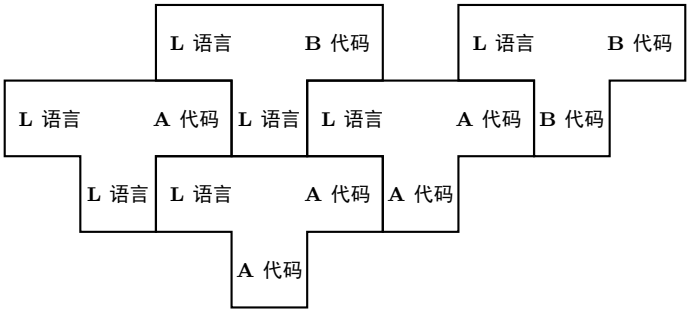


图 1.5 编译程序 “移植”

自编译方法：先对语言的核心部分构造一个小小的编译程序 (可用低级语言实现)，再以它为工具构造一个能够编译更多语言成分的较大编译程序。如此扩展下去，就能像滚雪球一样，越滚越大。通过这一系列自展途径而形成编译程序的过程叫做自编译过程。

在某一台机器上为某种语言构造一个编译程序，必须掌握以下内容：

1. 源语言：对被编译的源语言，要深刻理解其结构 (语法) 和含义 (语义)。

2. 目标语言：假定目标语言是机器语言，那么就必须搞清楚硬件的系统结构和操作系统的功能。

3. 编译方法：把一种语言程序翻译为另一种语言程序方法很多，但必须准确地掌握一二。

二、高级语言及其语法描述

2.1 程序语言的定义

2.1.1 语法

任何语言程序就可看成是一定字符集 (字母表) 上的一字符串 (有限序列)。所谓的语法是指这样的一组规则，用它可以形成和产生一个合适的程序。这些规则的一部分称为词法规则，另一部分称为语法规则 (产生规则)。

如：0.5 * X1 + C。可看作常数 0.5，标识符 X1，C。运算符 *，+。这些为单词符号。表达式称为语法范畴，或语法单位。

语言的单词符号是由词法规则所确定的。词法规则规定了字母表中哪样的字符串是一个单词符号。

词法规则是指单词符号的形成规则。包括各类型的常数，标识符，基本字，算符等。语法规则规定了如何从单词符号形成更大的结构 (即语法单位)，换言之，语法规则是语法单位的形成规则。一般程序语言的语法单位由：表达式，语句，分程序，函数等。

2.1.2 语义

语义问题：对于一个语言来说，不仅要给出它的词法，语法规则，而且要定义它的单词符号和语法单位的意义。

所谓一个语言的语义是指这样一组规则，使用它可以定义一个程序的意义。这些规则称为语义规则。现在还没有一种公认的形式系统，借助于它可以自动地构造出使用的编译程序，书上介绍的是基于属性文法的语法制导翻译方法。

一个程序的层次结构大体上如下：

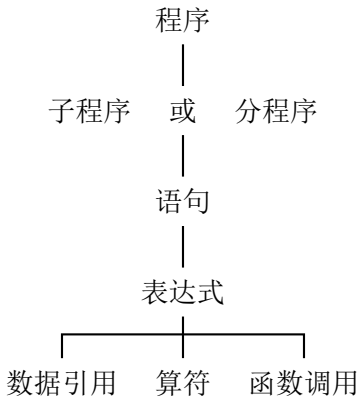


图 2.1 程序的层次结构

2.2 高级语言的一般特性

2.2.1 高级语言的分类

从语言范型分类，当今的大多数程序设计语言可分为四类。

- 强制式语言

  强制式语言也称过程式语言。其特点是命令驱动，面向语句。一个强制式语言程序由一系列的语句组成，每个语句的执行引起若干存储单元中的值得改变。

```
语句1;
语句2;
......
语句n;
```

  这类语言有：C，Pascal，FORTRAN，Ada

- 应用式语言

  应用式语言更注重程序所表示的功能，而不是一个语句接着一个语句地执行。程序的开发过程从前面已有的函数出发构造出更复杂的函数。

```
函数n(...函数2(函数1(数据))...)
```

  这类语言有：LISP，ML

- 基于规则的语言

  基于规则的语言程序的执行过程式：检查一定的条件，当它满足值，则执行适当的动作。也称逻辑程序设计语言。

```
条件1 -> 动作1
条件2 -> 动作2
......
条件n -> 动作n
```

- 面向对象语言

  面向对象语言如今已成为最流行、最重要的语言。它主要的特征是支持封装性、继承性和多态性等。把复杂的数据和用于这些数据的操作封装在一起，构成对象；对简单对象进行扩充继承简单对象的特性，从而设计出复杂的对象。通过对象的构造可以使面向对象程序获得强制式语言的有效性，通过作用于规定数据的函数的构造可以获得应用式语言的灵活性和可靠性。

  这类语言有：JAVA，C++

2.2.2 程序结构

一个高级语言程序通常由若干子程序段构造，许多语言还引入了类，程序包等更高级的结构。下面以说明 JAVA [5]语言程序结构。

[5]FORTRAN 等语言例子见书 P16，本人没学过故不写

JAVA 是一种面向对象的高级语言，它很重要的方面是类及继承的概念，同时支持多态性和动态绑定特征。

相信各位都能了解 JAVA，这里不写了。

2.2.3 数据类型与操作

一个数据类型通常包括以下三种要素。

- 用于区别这种类型的数据对象的属性。
- 这种类型的数据对象可以具有的值。
- 可以作用于这种类型的数据对象的操作。

初等数据类型

一个程序语言必须提供一定的初等类型数据成分，并定义对于这些数据成分的运算。常见的初等数据类型有：

- **数值数据**：如整数，实数，复数以及这些类型的双长 (或多倍长) 精度数，对他们可施行算数运算 (+,-,*,/等)。
- **逻辑数据**：多数语言有逻辑型 (布尔型) 数据，对它们可施行逻辑运算 (and,or,not 等)。
- **字符数据**：字符型或字符串型数据。
- **指针类型**：指针的值指向另一些数据。

标识符：由字母或数字组成的以字母为开头的一个字符串。

计算机的名字仅代表一个抽象的存储单位，还必须指出它的属性。名字还包含类型和作用域等，相信读者都明白，这里不写了。

这小节内容多为数据结构的内容以及高级语言的基础知识，不写了。

2.3 程序语言的语法描述

2.3.1 字母表上的运算

设 Σ 是一个有穷字母表，它的每个元素 $(a, b, \cdots)$ 称为一个符号。Σ 上的一个符号串是指由 Σ 中的符号所构成的一个有穷序列。不包含任何符号的序列被称为空字，记作 ϵ。用 Σ^* 表示 Σ 上所有符号串的全体。则有以下运算规则:

- 字母表乘积:

$$\Sigma_1\Sigma_2 = \{ab | a \in \Sigma_1, b \in \Sigma_2\} \tag{2.1}$$

  例如 $\{0,1\}\{a,b\} = \{0a, 0b, 1a, 1b\}$

- 字母表幂运算:

$$\begin{cases} \Sigma^0 = \{\epsilon\} \\ \Sigma^n = \Sigma^{n-1}\Sigma (n \geqslant 1) \end{cases} \tag{2.2}$$

  字母表的 n 次幂，就是长度为 n 的符号串构成的集合。

- 字母表 Σ 的正闭包:

$$\Sigma^+ = \Sigma \cup \Sigma^2 \cup \Sigma^3 \cup \dots \tag{2.3}$$

- 字母表 Σ 的正则闭包 (克林闭包):

$$\Sigma^* = \Sigma^0 \cup \Sigma \cup \Sigma^2 \cup \Sigma^3 \cup \dots \tag{2.4}$$

设 Σ 是一个字母表，$\forall x \in \Sigma^*$，x 称为是 Σ 上的一个串。串即字母表中的一个又穷序列。串 s 的长度记为 $|s|$。

2.3.2 上下文无关法

文法是描述语言的语法结构的形式规则 (即语法规则)。所谓上下文无关法是指这样一种文法：它所定义的语法范畴 (语法单位) 是完全独立于这种范畴可能出现的环境的。也即遇到某个语法单位时，完全不考虑它的上下文环境，而直接进行处理 (例如运算)。下文所指的文法无特别说明都是上下文无关文法。

例如我们将下面自然语言进行文法分析并绘制分析树。

He gave me a book.

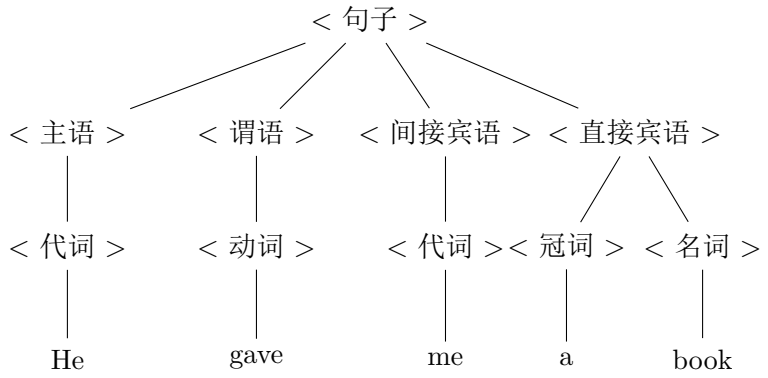


图 2.2 语法树：He gave me a book.

一个上下文无关法包括四个组成部分：一组终结符号，一组非终结符号，一个开始符号，一组产生式。在上例中，它们分别是：

- 终结符号：组成语言的基本符号

He,gave,me,a,book

- 非终结符号：代表语法范畴，语法概念

< 句子 >,< 主语 >,< 代词 > 等

- 开始符号：代表所定义语言中我们最感兴趣的语法范畴

< 句子 >

- 产生式：定义语法范畴的一种书写规范

< 直接宾语 > → < 冠词 >< 名词 >

下面重点介绍产生式，一个产生式的形式是：

$$A \to \alpha$$

其中箭头 (有时也用::=) 左边的 A 是一个非终结符，称为产生式的左部符号；箭头右边的 α 是由终结符号或/与非终结符号组成的一符号串，称为产生式的右部。

为了书写方便，若干个左部相同的产生式，如:

$$\begin{aligned} P &\to \alpha_1 \\ P &\to \alpha_2 \\ &\vdots \\ P &\to \alpha_n \end{aligned}$$

可合并为一个，缩写成

$$P \to \alpha_1 | \alpha_2 | \cdots | \alpha_n$$

其中，每个 α_i 有时也称为是 P 的一个候选式。

后面的讨论中，通常将大写字母 A,B,C... 或汉语词组代表非终结符号，a,b,c... 小写字母代表终结符, 用 α, β, γ 代表由终结符和非终结符组成的符号串 (也即要进行匹配的符号串)。

当表示推导一步时，可以使用 $\Rightarrow$ 代替 $\to$。例如

$$E \Rightarrow (E) \Rightarrow (E+E) \Rightarrow (E+i) \Rightarrow (i+i)$$

如果存在：

$$\alpha_1 \Rightarrow \alpha_2 \cdots \alpha_n$$

则我们称这个序列是从 α_1 到 α_n 的一个推导。称 α_1 可推导出 α_n。我们用 $\alpha_1 \overset{+}{\Rightarrow} \alpha_n$ 表示：从 α_1 出发，经一步或者若干步可推导出 α_n。而用 $\alpha_1 \overset{*}{\Rightarrow} \alpha_n$ 表示：从 α_1 出发，经 0 步或者若干步可推导出 α_n。换言之，$\alpha_1 \overset{*}{\Rightarrow} \beta$ 表示：$\alpha = \beta$ 或 $\alpha_1 \overset{+}{\Rightarrow} \beta$ 。

假定 G 是一个文法，S 是它的开始符号。如果 $S \overset{*}{\Rightarrow} \alpha$，则称 α 是一个句型。仅含终结符号的句型就是一个句子。文法 G 所产生的语句的全体是一个语言，记作 L(G)。

为了对句子的结构进行确定性分析，我们往往只考虑最左推导和最右推导。所谓最左推导是指：任何一步 $\alpha \Rightarrow \beta$ 都是对 α 中的最左非终结符进行替换的。最右推导同样。

2.3.3 语法分析树和二义性

可以用一张图表示一个句型的推导，这种表示称为语法分析树，或简称为语法树。语法树的根结由开始符号所标记。随着推导的展开，当某个非终结符被它的某个候选式所替换时，这个非终结符的相应结就产生出下一代新结，候选式中自左至右的每个符号对应一个新结，并用这些符号标记其相应的新结。每个新结和服结间都有一条连线。在一棵语法树生长过程中的任何时刻，所有那些没有后代的端末结自左至右排列起来就是一个句型。

例如如下文法及其对应分析树[6]：

$$E \Rightarrow (E) \Rightarrow (E+E) \Rightarrow (E*E+E) \Rightarrow (i*E+E) \Rightarrow (i*i+E) \Rightarrow (i*i+i)$$

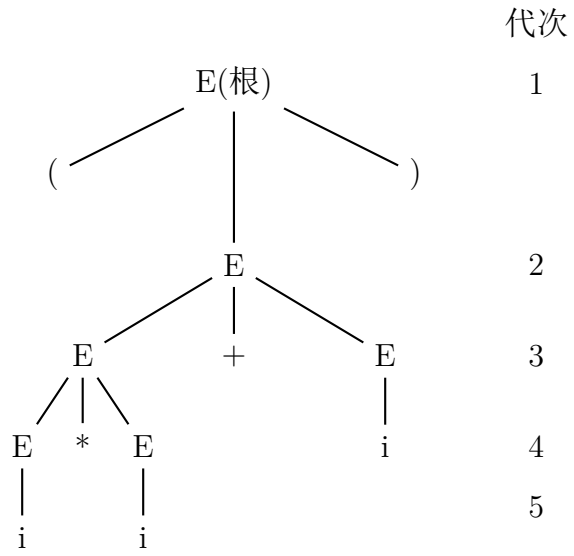


图 2.3　分析树示例

一棵语法树表示了一个句型种种可能的 (未必是所有) 不同推导过程，这样一棵语法树是这些不同推导过程的共性抽象，是它们的代表。但一个句型不唯一对应一棵语法树。值得注意的是，我们可以用完全不同的方法生成同一个句子。

如果一个文法存在某个句子对应两棵不同的语法树 (最左/最右推导)，则称这个文法是二义的。对于一个程序而言。常常希望它的文法是无二义性的。因为我们希望对它的每个语句的分析是唯一的。但是，只要我们能够控制和驾驭文法的二义性，文法的二义性的存在并不一定是一件坏事。

二义性问题是不可判定的，我们所能做的事是为无二义性寻找一组充分条件 (未必是必要的)。例如在四则运算中，规定 * 运算符优先级高于 + 运算符[7]。

作为描述程序语言的上下文无关法，有以下限制。

- 文法中不含任何下面形式的产生式(除了引起二义性没有任何用处)：

$$P \to P$$

- 每个非终结符 P 必须有用处，也即必须存在含 P 的句型。对 P 不存在永不终结的回路。

2.3.4 形式语言鸟瞰

乔姆斯基[8]将文法定义为四元组：$G = (V_T, V_N, S, \mathcal{P})$，各部分意义如下：

[6]书 P31-32 提供了更多的例子

[7]这里结合原书的例子更好理解，大意为：一开始认为 * 和 + 优先级相同，导致最左和最右推导不同，出现二义性。

[8]1956 年建立形式语言的描述

- V_N: 非空有限集，其中每个元素都为非终结符，也称为非终结符集。
- V_T: 也是非空有限集，其中每个元素均为终结符，也称终结符集。
- $\mathcal{P}$: 产生式的有限集。
- S: 文法的开始符号，至少要在一条产生式的左部中出现。

乔姆斯基把文法分成四种类型：0 型，1 型，2 型，3 型。这几类文法的差别在于对产生式施加不同的限制，类型越高，限制越强。

- **0 型文法**

  0 型文法: 左部至少有一个非终结符的文法。

  0 型文法也称为无限制文法。
- **1 型文法**

  1 型文法: 在 0 型文法的基础上，左部长度小于等于右部长度。

  1 型文法也称为上下文有关文法。这意味着对非终结符进行替换时务必考虑上下文，且一般不允许替换成空串 ϵ。

  例如，若存在 $\alpha A\beta \to \alpha\gamma\beta$ 是 1 型文法 G 的一个产生式，α, β 不为空，那么非终结符 A 只有在 α, β 这样一个上下文环境总才能被替换为 γ。
- **2 型文法**

  2 型文法: 左部有且仅有一个非终结符的文法。

  2 型文法也称上下文无关文法。也即不用考虑上下文。
- **3 型文法**

  3 型文法也称为正则文法 (Regular Grammar), 3 型文法分为两种 (满足 2 型文法的基础上):
  - 右线性文法: $A \to wB/A \to w$
  - 左线性文法: $A \to Bw/A \to w$

三、词法分析

3.1 对于词法分析器的要求

3.1.1 词法分析器的功能和输出形式

词法分析器的功能是输入源程序，输出单词符号。程序语言的单词符号一般可分为下列五种。

1. **关键字**：由程序语言定义的具有固定意义的标识符。有时被称为保留字或基本字。例如 begin,if,while..... 这些字通常不用作一般标识符。
2. **标识符**：用来表示各种名字，如变量名，数组名，过程名等。
3. **常数**：一般用整型，实型，布尔型等等
4. **运算符**：如 +，-，*，/ 等
5. **界符**：如逗号，括号，分号，注释符等

词法分析器所输出的单词符号常常表示为以下二元式：

(单词种别，单词符号的属性值)

单词种别通常采用整数编码。如果一个种别只含一个单词符号，那么，对于这个单词符号，种别编码就完全代表它自身了。若一个种别含有多个单词符号，那么，对于它的每个单词符号，除了给出种别编码之外，还应给出有关的单词符号的属性信息。

单词符号的属性是指单词符号的特征或特性。属性值则是反应特性或特征的值。例如，对于某个标识符，常将存放它的有关信息的符号表项的指针作为属性值。

考虑如下 C++ 代码段：

while(i>=j) i–;

经词法分析器处理后，它将被转换为如下的单词符号序列：

```
<while,->
<(,->
<id,指向i的符号表项的指针>
<>=,->
<id,指向j的符号表项的指针>
<),->
<id,指向i的符号表项的指针>
<--,->
<;,->
```

3.1.2 词法分析器作为一个独立子程序

把词法分析器安排为一个独立阶段的好处是：它可以使整个编译程序的结构更简洁，清晰和条理化。

我们可以把词法分析器安排成一个子程序，每当语法分析器需要一个单词符号时就调用这个子程序。每一次调用，词法分析器就从输入串中识别出一个单词符号，把它交给语法分析器。后文假定词法分析器按这种方式工作。

3.2 词法分析器的设计

3.2.1 输入，预处理

词法分析器工作的第一步是输入源程序文本。输入串一般是放在一个缓冲区中，称为输入缓冲区。词法分析器的工作 (单词符号的识别) 可以直接在这个缓冲区中进行。在多数情况下，会对输入串进行预处理。

对于许多程序语言来说，空白符，跳格符，回车符，换行符等编辑性字符除了在文字常数中之外，在别处的任何出现都没有意义，而注解部分几乎允许出现在程序中的任何地方。它们不是程序语言的必要组成部分，它们存在的意义仅是提高程序的易读性和易理解性。对于它们，预处理可以将其剔掉。

有些语言将空白符 (一个或数个) 用作单词符号之间的间隔，即界符。这种情况下，预处理时可把相继的若干个空白结合成一个。

我们可以设想构造一个预处理子程序，它能够完成上面所述的任务。每当词法分析器调用它时，它就处理出一窜确定长度的输入自读，并将其装进词法分析器所指定的缓冲区中 (扫描缓冲区)。这样分析器就可以在此缓冲区中直接进行单词符号的识别，而不必照管其他繁琐事务。

分析器对扫描缓冲区进行扫描时一般用两个指示器，一个指向当前正在识别的单词的开始位置 (新单词首字符)，另一个用于向前搜索以寻找单词的终点。

无论扫描缓冲区设得多大都不能保证单词符号不会被它的边界所打断。因此，扫描缓冲区最好的使用一个如下所示的一分为二的区域。

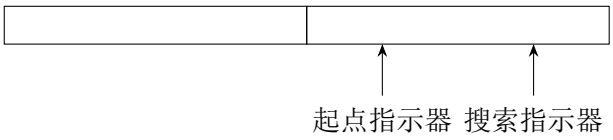


图 3.1　扫描缓冲区

假定每半区可容 120 个字符，而这两个半区又是互补使用的。如果收缩指示器从单词起点出发搜索到半区得边缘但尚未到达单词得终点，那么就应调用预处理程序，令其把后续的 120 个输入字符装进另半区。我们认定，在搜索指示器对另半区进行扫描得时期内，线性单词得终点必能够达到。这意味着对标识符和常数得长度实际上必须加以限制 (例如不多于 120 字符)。

3.2.2 单词符号得识别：超前搜索

词法分析器得结构如图3.2所示，当词法分析器调用预处理子程序处理出一串输入字符放进扫描缓冲区之后，分析器就从此缓冲器中逐一识别单词符号。当缓冲区里的字符串被处理完之后，他又调用预处理程序装入新串。

下面介绍单词符号识别的一个简单方法——超前搜索。

关键字的识别[9]

在一些关键字不加以保护 (用户可以将关键字用作普通标识符) 的语言中，为了识别关键字，我们必须超前扫描多个字符，超前到能够确定词性的地方为止。

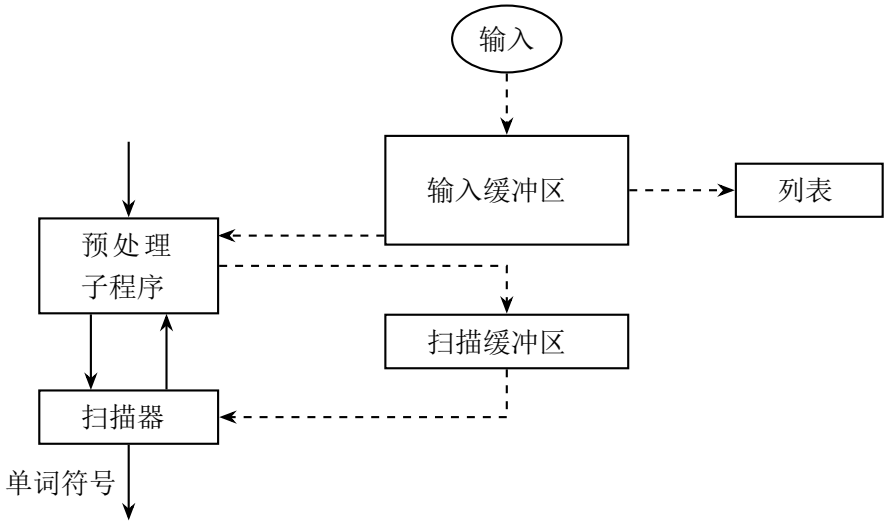


图 3.2　词法分析器

标识符的识别

多数语言的标识符时字母开头的 “字母/数字” 串，而且在程序中标识符的出现都后跟着算符或界符，因此识别标识符大多没有困难。

常数的识别

多数语言算术常数的表示大体相似，对它们的识别也是很直接的。但对于某些语言的常数的识别也需要超前搜索的方法。

逻辑 (或布尔) 常数和用引用括起来的字符串常数都很容易识别。但 FORTRAN 有需格外处理。

算符和界符的识别

词法分析器应将那些由多个字符复合成的算符和界符拼合成一个单词符号。因为这些字符串是不可分的整体，在这里同样需要超前搜索。

[9]书 P39-40 有一个 FORTRAN 语言的例子

3.2.3 状态转换图

状态转换图是设计词法分析程序的一种好途径。转换图是一张有限方向图。在状态转换图中，结点代表状态，用圆圈表示。状态之间用箭弧连结。箭弧上的标记 (字符) 代表在射出结点状态下可能出现的输入字符或字符类。

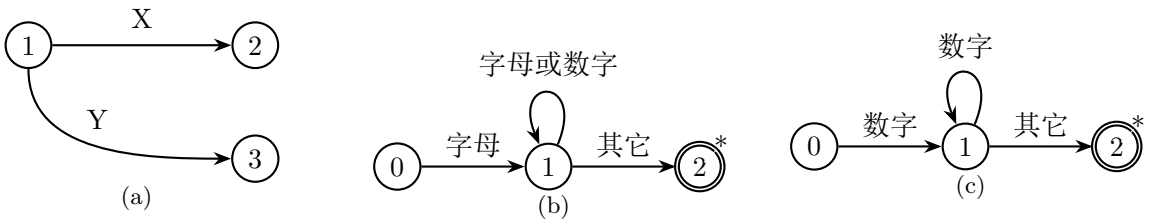


图 3.3 状态转换图

例如在 3.5(a) 表示：在状态 1 下，若输入字符为 X，则读进 X，并转换到状态 2. 若输入字符为 Y，则读 Y，转换到状态 3。

一张在转换图只包含有限个状态 (即有限个结点)，其中有一个被认为是初态，而且实际上至少要有一个终态 (双圈表示)。

一个状态转换图可用于识别 (或接受) 一定的字符串。例如，识别标识符的转换图 3.5(b)。终态结上打个星号 * 意味着多读进了一个不属于标识符部分的字符，应把它退还给输入串[10]。

3.2.4 状态转换图的实现

转换图容易用程序实现。最简单的办法是让每个状态结点对应一小段程序。下面引用一组全局变量和过程，将它们作为实现转换图的基本成分：

1. ch
   字符变量，存放最新读进的源程序字符。
2. strToken
   字符数组，存放构成单词符号的字符串。
3. GetChar
   子程序过程，将下一输入字符读到 ch 中，搜索指示器前移以字符位置。
4. GetBC
   子程序过程，检查 ch 中的字符串是否为空白。若是，则调用 GetChar 直至 ch 中进入一个非空白字符。
5. Concat
   子程序过程，将 ch 中的字符串连接到 strToken 之后。
6. IsLetter 和 IsDigit
   布尔函数过程，它们分别判断 ch 中的字符是否为字母和数字。

[10]书 P42-44 有一个完整的简单语言进行词法分析绘制状态转换图的例子

7. Reverse
整型函数过程，对 strToken 中的字符串查找保留字表，若它是一个保留字则返回编码，否则返回 0。

8. Retract
子程序过程，将搜索指示器回调一个字符位置，将 ch 置为空白字符。

9. InsertId
整型函数过程，将 strToken 中的标识符插入符号表，返回符号表指证。

10. InsertConst
整型函数过程，将 strToken 中的常数插入常数表，返回常数表指针。

对于不含回路的分叉结点来说，可让它对应一个 switch 语句或一组 if...else...then 语句，如图3.4(a)；对于含回路的状态结点来说，可让它对应一个由 while 语句或 if 语句构成的程序段，如图3.4(b)。

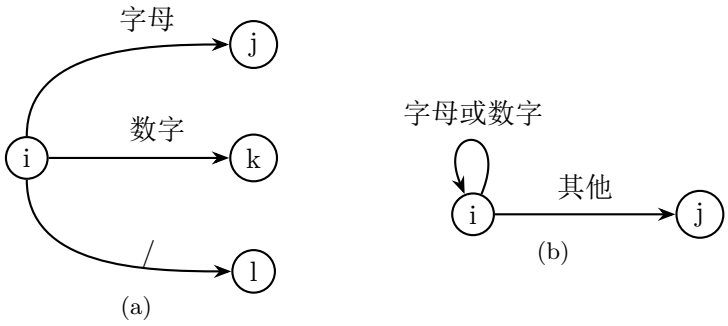


图 3.4　不含回路与含有回路的转换图

3.3 正则表达式与有限自动机

3.3.1 正规式与正规集

对于字母表 Σ，我们感兴趣的是它们的一些特殊字集，即所谓正规集。我们将使用正规式[11]这个概念来表示正规集。

可以将正则式理解为一种表示字符串 (句子) 的语法，而正则集表示利用该语法所能表示的所有字符串的集合。例如正则式 A = (a|b)c 表示以 a 或 b 开头，c 结尾的字符串，而这种字符串有两个 ac 和 bc，其对应的正则集为 L(A) = {ac,bc}。[12]

下面是正规式和正规集的递归定义。

- 特殊字符串都是正规式

ϵ 和 ϕ 都是 Σ 上的正规式，它们所表示的正规集分别是 $\{\epsilon\}$ 和 ϕ。

- 单个字符串是正规式

[11]更常用的叫法为正则表达式 (Regular Expression)，简称 RE，关于使用 RE 匹配字符串的方法可以参考我的另一篇笔记：https://github.com/Pionpill/Notebook/blob/Pionpill/related/RegEx/syntax.md

[12]由于上面的形式化语言不好理解，这段为本人自己给出的解释，仅供参考。

任何 $a \in \Sigma$，a 是 Σ 上的一个正规式，它所表示的正规集为 $\{a\}$。

- 正规式运算法则

  假定 U 和 V 都是 Σ 上的正规式，它们所表示的正规集分别记为 $L(U)$ 和 $L(V)$，那么 $(U|V), (U \cdot V), (U)^*$ 也都是正规式。它们所表示的正规集分别为：$L(U) \cup L(V), L(U)L(V)$ 连续积,$(L(U))^*$(闭包)。

  - $U|V$：表示正则式 U 或 V，例如 (ac|b) 表示符号 ac 或者 b。
  - $U \cdot V$：表示正则式 U 后跟着 V，例如 (ac)(b) 表示符号 acb。
  - $(U)^*$：表示任意个 U 中字符串的组合，例如 a^* 对应正则集 $\{\epsilon$,a,aa,aaa,...$\}$。$(ab)^*$ 对应正则集 $\{\epsilon$,ab,abab,ababab,...$\}$，ab 为一个整体。

仅由有限次使用上述三步骤而得到的表达式才是 Σ 上的正规式。仅由这些正规式所表示的字集才是 Σ 上的正规集。

正规式的运算符 “|” 读作 “或”，“ • ” 读作 “连接”，“*” 读作 “闭包”。规定运算符优先级如下：$^* \rightarrow \bullet \rightarrow |$。连接符 “ • ” 可省略不写[13]。

若两个正规式所表示的正规集相同，则认为二者等价。两个等价的正规式 U，V 记为 U = V。如果 U,V,W 均为正规式，由以下关系成立：

- U|V=V|U (交换律)
- U|(V|W) = (U|V)|W (结合律)
- U(VW) = (UV)W (结合律)
- U(V|W) = UV|UW　(V|W)U = VU|WU (分配律)
- ϵU = Uϵ = U

3.3.2 确定有限自动机 (DFA)

一个确定有限自动机 (DFA)M 是一个五元式:

$$M = (S, \Sigma, \delta, s_0, F)$$

- S: 一个有限集，它的每个元素称为一个状态。
- Σ: 一个有穷字母表，每个元素称为一个输入字符。
- δ: 一个从 $S \times \Sigma$ 到 S 的单值部分映射。$\delta(s, a) = s'$ 表示：当前状态为 s, 输入字符为 a, 将转换到下一状态 s'。称 s' 为 s 的后继状态。
- $s_0 \in S$: 唯一初态。
- $F \subseteq S$: 终态集 (可空)。

显然，一个 DFA 可用一个矩阵表示，该矩阵的行表示状态，列表示输入字符，矩阵元素表示 $\delta(s, a)$ 的值。这个矩阵称为状态转换矩阵。例如如下 DFA：

$$M = (\{0, 1, 2, 3\}, \{a, b\}, \delta, 0, \{3\})$$

其中，δ 为：

[13]书 P47 有几个正规集例子

$$\delta(0,a)=1 \qquad \delta(0,b)=2$$
$$\delta(1,a)=3 \qquad \delta(1,b)=2$$
$$\delta(2,a)=1 \qquad \delta(2,b)=3$$
$$\delta(3,a)=3 \qquad \delta(3,b)=3$$

他所对应的状态转换矩阵为[14]：

表 1　状态转换矩阵

状态	a	b
0	1	2
1	3	2
2	1	3
3	3	3

一个 DFA 也可表示成一张 (确定的) 状态转换图。假定 DFA M 含有 m 个状态和 n 个输入字符，那么，这个图含有 m 个状态结点，每个结点顶多有 n 条箭弧射出和别的结点相连接，即绘图时状态对应结点，输入字符对应连线。每条箭头用 Σ 中的一个不同输入字符作为标记，整张图含有一个初态结点和若干个 (可以是 0 个) 终态结点。

对于 Σ^* 中的任何字 α，若存在一条从初始结点到终态结点的通路，且这条通路上所有弧的标记符连接成的字等于 α，则称 α 可为 DFA M 所识别 (读出或接受)。若 M 的初态结点同时又是终态结点，则空字 ϵ 可为 M 所识别 (接受)。DFA M 所能识别的字的全体记为 L(M)。

例如上面例子对应的状态转换图：

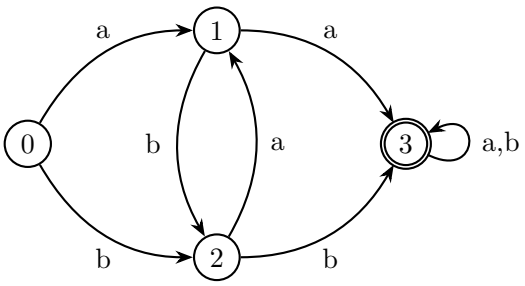


图 3.5　状态转换图

如果 DFA M 的输入字母表为 Σ，则我们也称 M 是 Σ 上的一个 DFA[15]。可以证明：Σ 上的一个字集 $V \subseteq \Sigma^*$ 是正规的，当且仅当存在 Σ 上的 DFA M，使得 V=L(M)[16]。

DFA 的确定性表现在映射 $\delta : S \times \Sigma \to S$ 是一个单值函数。也就是说，对任何状态 $s \in S$ 和输入符号 $a \in \Sigma$，$\delta(s,a)$ 唯一地确定了下一状态。如果允许 δ 是一个多值函数[17]，就得到了

[14]书上比较难理解，建议看这个视频：https://www.bilibili.com/video/BV1zW411t7YE?p=15
[15]听君一席话，如听一席话。简言之这句话是废话
[16]这句话也看不懂，建议忽略
[17]一个 x 对应多个 y

非确定的自动机。

最长子串匹配原则：

- 当输入串的多个前缀与一个或多个模式匹配时，总是选择最长的前缀进行匹配。
- 在达到某个终态后，只要输入带上还有符号，FA 就继续前进，以便寻找尽可能长的匹配。

例如在下图中，如果我们遇到了 <= , 则既可以在 1 终结也可以在 2 终结，但是按最长字串匹配原则应该在 2 终结。

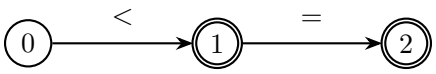


图 3.6　最长字串匹配原则

3.3.3 非确定的有限自动机 (NFA)

一个非确定有限自动机 (NFA)M 是一个五元式:

$$M = (S, \Sigma, \delta, s_0, F)$$

- S, Σ, F 意义与 NFA 相同。
- $s_0 \subseteq S$: 一个非空初态集。
- δ: 一个从 $S \times \Sigma^*$ 到 S 的子集映照。即 $\delta : S \times \Sigma^* \to 2^S$。

显然，一个含有 m 个状态和 n 个输入字符的 NFA 可以表示成如下状态转换图：该图含有 m 个状态结点，每个结点可射出若干条箭弧与别的结点相连，每条弧用 Σ^* 中的一个字 (不一定要不同的字而且可以是空字) 做标记 (称为输入字)，整张图至少含有一个初态结点以及若干个 (可以是 0 个) 终态结点。某些结点既可以是初态结点也可以是终态结点。

对于 Σ^* 中的任何字 α，若存在一条从某一初态结点到某一终态结点的通路，且这条通路上所有弧的标记符连接成的字 (忽略标记为空的弧) 等于 α，则称 α 可为 NFA M 所识别 (读出或接受)。若 M 的某些结点既是初态结点又是终态结点，或者存在一条从某个初态结点到某个终态结点的 ϵ 通路，则空字 ϵ 可为 M 所识别 (接受)。

例如下图[18] NFA 所能识别的是所有含有相继两个 a 或相继两个 b 的字。

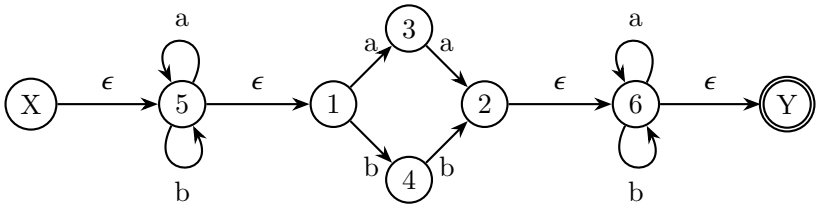


图 3.7　非确定有限自动机

[18] 书上原图有误，1 → 4

显然，DFA 是 NFA 的特例。但是，对于每个 NFA M 存在一个 DFA M″, 使得 L(M) = L(M″)[19]。

将 NFA 状态转换图转换为 DFA 状态转换图时有以下替换规则：

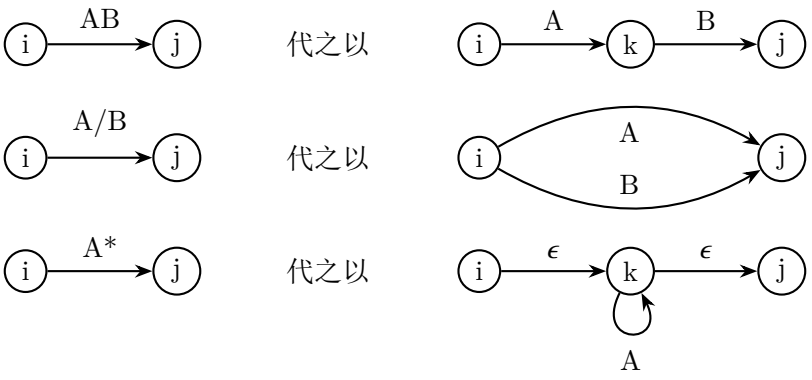


图 3.8 替换规则

正规式 $(a|b)^*(aa|bb)(a|b)^*$ 对应的 NFA 如图 3.7 所示，对应的状态转换矩阵如下：

表 2 正规式的状态转换矩阵

I	I_a	I_b
{X，5，1}	{5，3，1}	{5，4，1}
{5，3，1}	{5，3，1，2，6，Y}	{5，4，1}
{5，4，1}	{5，3，1}	{5，4，1，2，6，Y}
{5，3，1，2，6，Y}	{5，3，1，2，6，Y}	{5，4，1，6，Y}
{5，4，1，6，Y}	{5，3，1，6，Y}	{5，4，1，2，6，Y}
{5，4，1，2，6，Y}	{5，3，1，6，Y}	{5，4，1，2，6，Y}
{5，3，1，6，Y}	{5，3，1，2，6，Y}	{5，4，1，6，Y}

对上表中的所有状态子集重新命名，得到如下状态转换矩阵。

表 3 重命名的状态转换矩阵

s	**a**	**b**
0	1	2
1	3	2
2	1	5
3	3	4
4	6	5
5	6	5
6	3	4

[19]证明过程见书 P49-50

与上表相对应的状态转化图如下所示，显然这和图3.7是等价的。

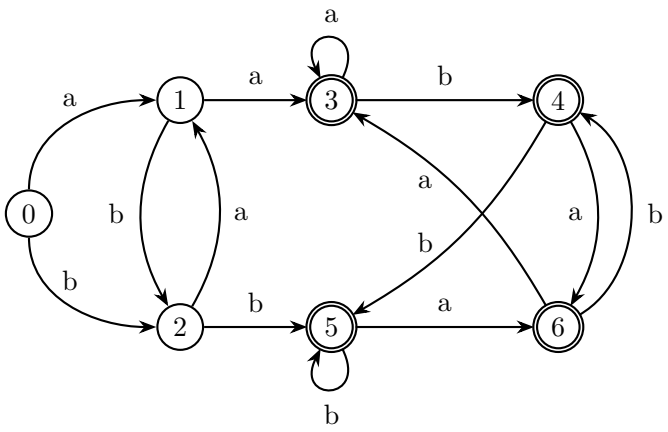


图 3.9　未化简的 DFA

3.3.4 正规文法与有限自动机的等价性

对于正规文法 G 和有限自动机 M，如果 L(G) = L(M)，则称 G 和 M 是等价的。并有以下结论[20]：

- 对每一个右线性正规文法 G 或左线性正规文法 G，都存在一个有限自动机 (FA)M，使得 L(M) = L(G)。
- 对每一个 FA M，都存在一个右线性文法 G_R 和左线性文法 G_L，使得 L(M) = L(G_R) = L(G_L)。

3.3.5 正规式有限自动机的等价性

有以下两个结论[21]：

- 对任何 FA,M 都存在一个正规式 r, 使得 L(r)=L(M)。
- 对任何正规式 r, 都存在一个 FA M, 使得 L(M)=L(r)。

总而言之一句话，三者是等价的：正则文法 ↔ 正则表达式 ↔ FA。

3.3.6 确定有限自动机的化简

一个确定有限自动机 M 的化简指的是：寻找一个状态数比 M 少的 DFA M′，使得 L(M) = L(M′)。

假定 s 和 t 是 M 的两个不同状态，我们称 s 和 t 是等价的：如果从状态[22] s 出发能读出某个字 w 而停于终态，那么同样，从 t 出发也能读出同样的字 w 而停于终态；反之，若从 t 出发能读出某个字 w 而停于终态，则从 s 出发也能读出同样的 w 而停于终态。

[20]证明见书 P51-52。

[21]证明见书 P53-56。

[22]转换图中的结点

如果 DFA M 的两个状态 s 和 t 不等价，则称这两个状态是可区别的。例如，终态与非终态是可区别的。因为，终态能读出空字 ϵ，非终态则不能。

一个 DFA M 的状态最少化过程旨在将 M 的状态集分割成一些不相交的子集，使得任何不同的两子集中的状态都是可区别的，而同一子集中的任何两个状态都是等价的。最后，在每个子集中选出一个代表，同时消去其他等价状态。

3.4 总结与题型

3.4.1 正则表达式与有限自动机的转换

这章为笔者添加的，主要是对一些方法的归纳，用于做题，仅供参考。

从 RE 直接构建 DFA 比较困难，常用的方式是从 RE 构建 NFA 再构建 DFA。

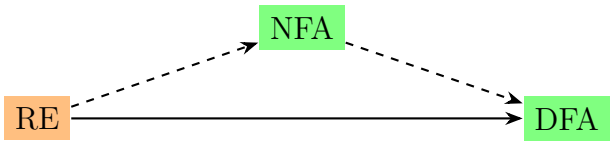


图 3.10　RE-FA

RE → NFA

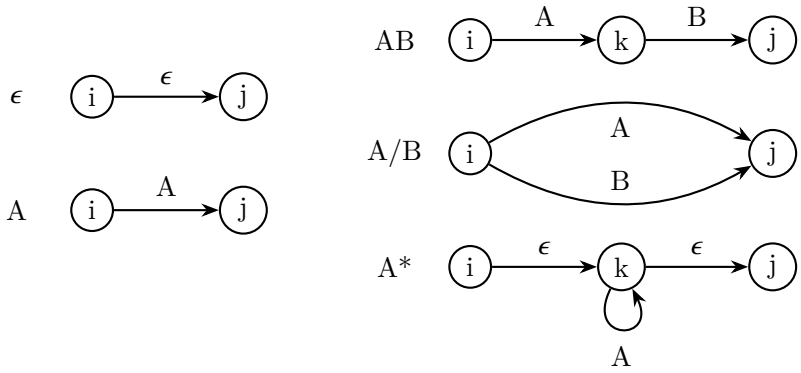


图 3.11　RE 到 NFA 替换规则

假如有 r=(a|b)*abb 转换为 NFA 过程如下 (XY 分别表示开始与结束)，注意画法不止一种：

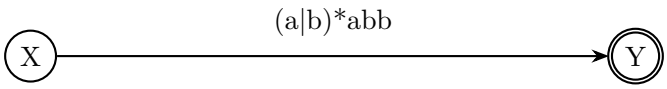


图 3.12　一: 创建初始到结束结点并连接

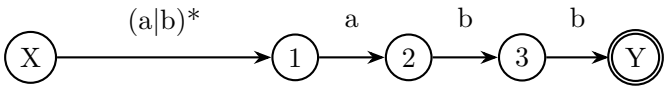


图 3.13　二：初步分解连接运算

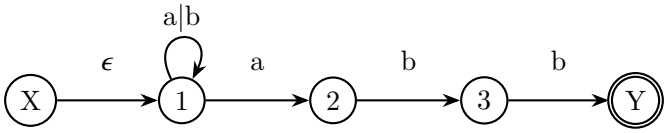


图 3.14　三：分解闭包

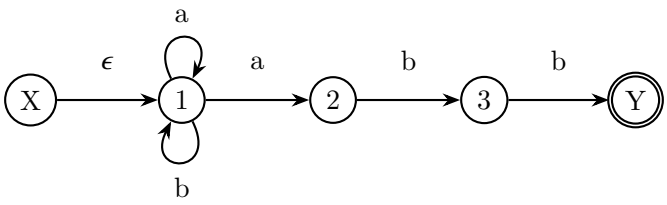


图 3.15　三：分解或运算

NFA → DFA

假设有以下 NFA：

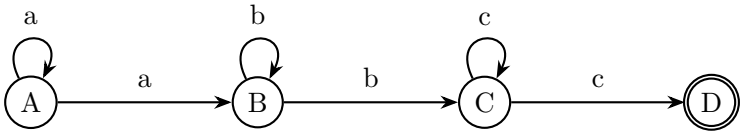


图 3.16　目标 NFA

首先，画转换表：

表 4　目标 NFA 的转换表

状态	a	b	c
A	{A,B}	ϕ	ϕ
B	ϕ	{B,C}	ϕ
C	ϕ	ϕ	{C,D}
D	ϕ	ϕ	ϕ

按以下规则绘图：

- 如果某个状态集 (相对于 NFA，例如 (A,B,C)) 中的一个子状态传入字符能达到下一个字符 (例如传入 a，A->X,BC 不能)，则该状态集也能达到下一个字符 (即 (A,B,C) 整体能到达 X)。也即采用 any(A,B,C) 运算。

- 如果某个状态集中包含终止状态，则这个状态集为终止状态。
- 只要下一个状态集与已有的状态集不同，就算是子集，也算作新状态。

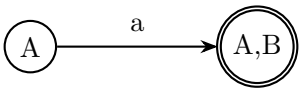


图 3.17　一：从起始结点 A 出发，查表绘制下一个状态

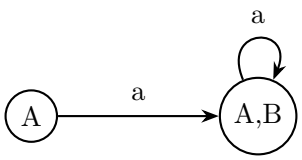


图 3.18　二：从 A,B 出发，遇到 a 继续进入 A,B

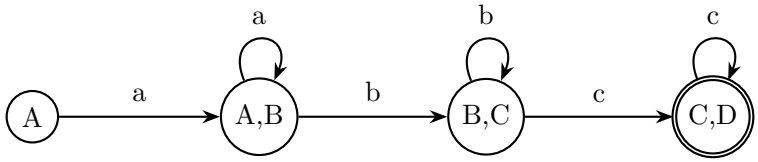


图 3.19　三：按此方法继续构造

带有 ϵ 边的转换，有这样一个 NFA：

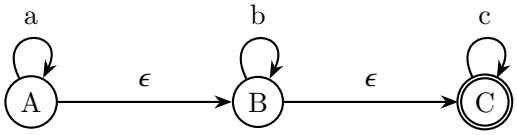


图 3.20　带 ϵ 的 NFA

同样画表：

表 5　目标 NFA 的转换表

状态	a	b	c
A	{A,B,C}	{B,C}	{C}
B	ϕ	{B,C}	{C}
C	ϕ	ϕ	{C}

由于不需要遇到任何符号，都能进入 A,B,C 所以它们都是初始状态。由于 C 的加入，这个状态集又是终止状态。

图 3.21　既全是初始状态又是终止状态

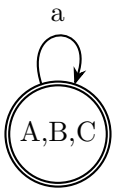


图 3.22　遇到 a 进入自身

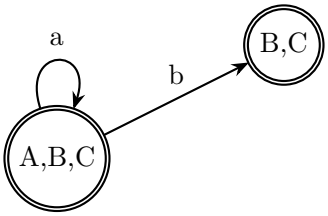


图 3.23　遇到 b 进入新状态

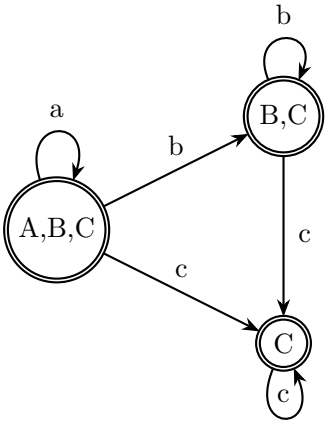


图 3.24　采用同样方法继续绘制

3.4.2 DFA 化简

DFA 的化简指的是: 寻找一个状态数比 M 少的 DFA M'，使得 $L(M) = L(M')$。其求解步骤如下:

1. 将 DFA M 的状态集分为两个子集: 终态集和非终态集。
2. 对每个子集 G, 如果面对某个输入符号得到的后继状态不属于同一个子集，则将 G 进一步划分。
3. 重复执行上一步，直到不能再划分。
4. 在每个子集中选一个状态作代表，消去其他状态，得到最少状态的等价 DFA M'。

例子: 有如下 DFA，初始状态为 1:

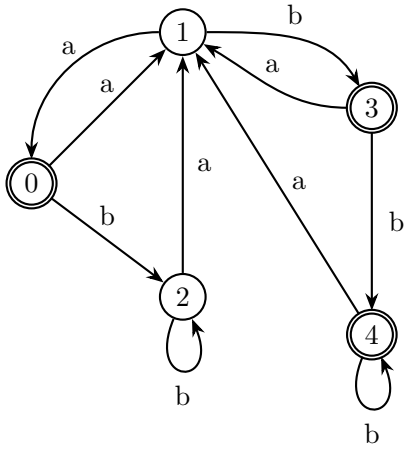


图 3.25　DFA 化简

1. 区分终态集与非终态集:

   令 $x = \{1, 2\}, y = \{0, 3, 4\}$

2. 划分:

   发现 $\{1, 2\}_a = \{0\}$, $\{1, 2\}_b = \{2, 3\}$ x 可划分为 $\{1\}, \{2\}$。

   $\{1, 3, 4\}_a = \{1\}$, $\{1, 3, 4\}_b = \{2, 4\}$ 可划分为 $\{0\}, \{3, 4\}$。$\{3, 4\}_b = \{4\}$ 不可再划分。

3. 简化绘图: 故令 $A = \{1\}, B = \{2\}, C = \{0\}, D = \{3, 4\}$, 则有:

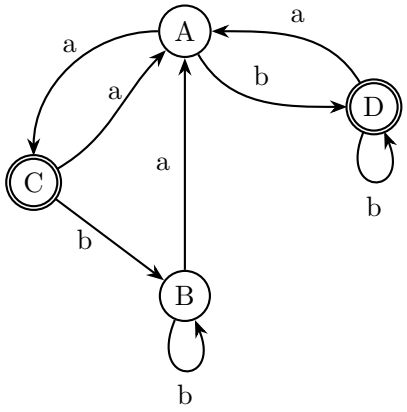


图 3.26　化简后的 DFA

四、语法分析——自上而下

4.1 语法分析器的功能

语法分析是编译过程的核心部分。它的任务是在词法分析识别出单词符号串的基础上，分析并判定程序的语法结构是否符合语法规则。

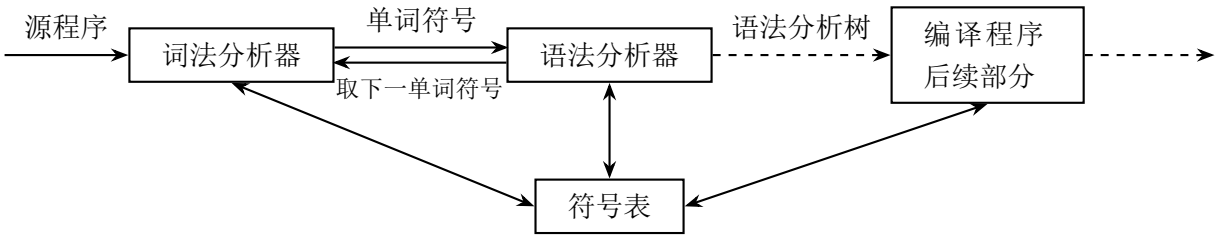


图 4.1　语法分析器在编译程序中的地位

语法分析器的工作本质上就是按文法的产生式，识别输入符号串是否为一个句子。这里的输入串指单词符号 (文法的终结符) 组成的有限序列。

对一个文法，给出一串 (终结) 符号时，如何判断它是不是该文法的一个句子？此时要判断能否从文法的开始符号出发推导出这个输入串。或者从概念上，就是要建立一棵与输入串相匹配的语法分析树。

按照语法分析树的建立方法，可以将语法分析办法分成两类，即自上而下分析发法和自下而上分析法。

推导与归约

推导：用产生式的右部替换左部，即正向的语法分析过程归约：用产生式的左部替换右部，即推导的逆过程

最左推导[23]

最左推导，即在推导过程中，总是选择每个句型的最左非终结符进行替换。与之对应的逆过程称为最右归约。相对的也有最右推导和最左归约。

在自顶向下分析中，采用的是最左推导与最右归约。

4.2 自上而下分析面临的问题

自上而下语法分析，顾名思义，就是从文法的开始符号出发，向下推导，推出句子。自上而下推导的主旨是：对任何输入串，试图用一切可能的办法，从文法开始符号 (根结) 出发，自上而下地为输入串建立一棵语法树。或者说，为输入串寻找一个最左推导。

下面举一个简单的例子[24]，假定有文法：

(1) S → xAy

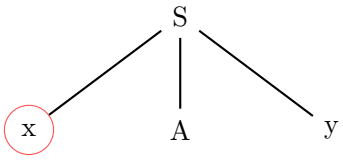
[23]这部分书上没有仔细讲，这里简单说一下，仅供参考。

[24]参考书 P67-68，这里为本人理解，叙述为白话，更准确但难懂的形式化语言建议参考原书

(2) A → **|*

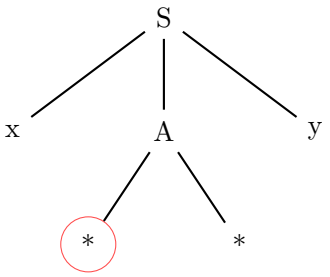
以及输入串 α: x*y，下面构造 α 的语法树。

1. 首先按文法开始符号产生根结 P，并让指示器 IP[25] 指向输入串的第一个符号 x。然后用 S 的规则 (此处只有一条) 绘制树[26]。



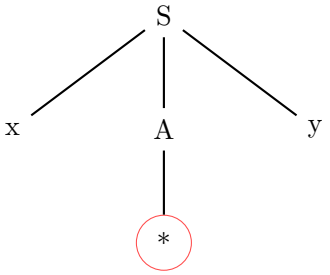
IP: x

2. 其次，需要用 S 的子结从左至右匹配整个输入串 α，左边第一个子结为终结符 x 和 IP 指向的符号匹配，于是让 IP 调整为指向下一输入符号 *，让第二个字结 A 进行匹配。A 有两个候选，这里试着用第一个候选进行匹配，于是有新的语法树：



IP: *

3. 然后，子树 A 的最左子结和 IP 所指的符号 * 相符，然后我们再把 IP 指向下一符号并让 A 的第二个子结进入工作。发现第二子结 * 和 IP 指向的 y 不匹配。因此 A 宣告失败，回溯查看 A 是否有其他候选表达式。注销之前的语法树，绘制新的语法树，IP 恢复为进入 A 时的原值，重新指向 *。

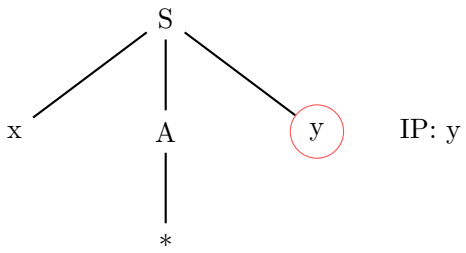


IP: *

4. 此时将 IP 所指的 * 与 A 的字节 * 匹配，成功匹配后，IP 指向 y，获取语法树第三个字节 y 进行匹配，成功。至此语法树匹配成功，证明是 α 是一个句子。

[25] IP 逐个指向输入串 α 中的符号

[26] 圆圈代表当前匹配的结点



这种自上而下分析法存在许多困难和缺点。

- 文法左递归

左递归: 存在非终结符 P，有如下表达式：

$$P \overset{+}{\Rightarrow} P\alpha$$

含有左递归的文法将使上述的分析过程陷入无限循环。因此，使用自上而下分析法必须消除文法的左递归。

- 回溯

如果走了一大段错路，就必须抛弃所有的工作 (中间代码，表格记录) 重来，浪费时间。因此，最好应该设法消除回溯。

- 虚假匹配

当某一个非终结符用某一候选项匹配成功时，这种成功可能仅是暂时的。这需要更复杂的回溯技术处理这种虚假匹配线性。一般来说，难以消除虚假匹配线性，但从最长的候选项开始匹配，会产生较少的虚假现象。

- 出错位置

当最终报告分析不成功时，难以知道输入串中出错的具体位置。

- 穷举的代价

可以看出，自上而下分析是一种穷举算法。这必然会导致效率低，代价高的问题。甚至严重的低效问题在实践中没有价值。

4.3 LL(1) 分析法

为了克服自上而下分析的缺陷：左递归，回溯。这节将研究递归子程序分析算法及其变种—预测分析法。

4.3.1 消除左递归

假定关于非终结符 P 的规则为[27]：

$$P \to P\alpha|\beta$$

[27]这里有很好的解释：https://www.bilibili.com/video/BV1zW411t7YE?p=20

其中，β 不以 P 开头，那么，我们可以把 P 的规则改写为如下的非直接左递归形式：

$$P \to \beta P^{'}$$
$$P^{'} \to \alpha P^{'} | \epsilon \qquad (\epsilon\text{为空字})$$

这种形式和原来的形式是等价的，也即从 P 推出的符号串是相同的。

举个例子：

$$E \to E + T | T \longleftrightarrow \begin{cases} E \to TE^{'} \\ E^{'} \to +TE^{'} | \epsilon \end{cases}$$
$$T \to T * F | F \longleftrightarrow \begin{cases} T \to FT^{'} \\ T^{'} \to *FT^{'} | \epsilon \end{cases}$$

一般而言，假定 P 关于的全部产生式：

$$P \to P\alpha_1 | P\alpha_2 | \cdots | P\alpha_m | \beta_1 | \beta_2 | \cdots | \beta_n$$

其中，每个 α 不为 ϵ，每个 β 不以 P 开头，那么消除左递归后的规则为：

$$P \to \beta_1 P^{'} | \beta_2 P^{'} | \cdots | \beta_n P^{'}$$
$$P^{'} \to \alpha_1 P^{'} | \alpha_2 P^{'} | \cdots | \alpha_m P^{'} | \epsilon$$

使用这个办法，可以将见诸于表面上的直接左递归消除 (即把直接左递归改成直接右递归)。但这并不能消除整个文法的左递归性。

消除间接左递归只需要将产生间接左递归的式子直接带入，变成直接左递归之后再消除即可。

4.3.2 消除回溯，提左因子

为了消除回溯就必须保证候选项获得成功的匹配。假设 A 有 n 个候选项 $\alpha_1, \alpha_2, \cdots, \alpha_n$。A 所面临的第一个输入符号为 a，如果 A 能够根据不同的输入符号指派相应的候选 α_i 去匹配，就没有回溯了。

令 G 是一个不含左递归的文法，对 G 的所有非终结符的每个候选 α 定义它的终结首符集 FIRST(α) 为：

$$\text{FIRST}(\alpha) = \{a | \alpha \overset{*}{\Rightarrow} a \cdots, a \in V_T\}$$

特别的，若 $\alpha \overset{*}{\Rightarrow} \epsilon$，则规定 $\epsilon \in \text{FIRST}(\alpha)$。换句话说，FIRST($\alpha$) 是 α 的所有可能推导的开头终结符或 ϵ。

如果非终结符 A 的所有候选首符集两两不相交，那么，当要求 A 匹配输入串时，A 就能根据它所面临的第一个输入符号 a，准确地指派某一个候选前去执行任务。这个候选就是那个终结首符集含 a 的 α。

并非所有 enfant 的候选的终结首符集都两两不相交。其解决方案是：提取公共左因子。例如，假定关于 A 的规则是：

$$A \to \delta\beta_1 | \delta\beta_2 | \cdots | \delta\beta_n | \gamma_1 | \gamma_2 | \cdots | \gamma_m \quad (\text{每个}\gamma\text{不以}\delta\text{开头})$$

那么，可以把这些规则改写成：

$$A \to \delta A^{'} | \gamma_1 | \gamma_2 | \cdots | \gamma_m$$

$$A^{'} \to \beta_1 | \beta_2 | \cdots | \beta_n$$

经过反复提取左因子，就能够把每个非终结符的所有候选首符集变成两两不相交。

4.3.3 FOLLOW 集和 FIRST 集

FOLLOW 集

假定 S 是文法 G 的开始符号，对于 G 的任何非终结符 A，可能在某个句型中紧跟在 A 后边的终结符 a 的集合，记为 FOLLOW(A)。

$$\text{FOLLOW}(A) = \{a | S \overset{*}{\Rightarrow} \cdots Aa \cdots, a \in V_T\}$$

若 $S \overset{*}{\Rightarrow} \cdots A$，则规定 $\# \in \text{FOLLOW}(A)$。换句话说，FOLLOW(A) 是所有句型中出现在紧接 A 之后的终结符或 #(结束符)。

SELECT 候选集

产生式 $A \to \beta$ 的候选集是指可以选用该产生式进行推导时对应的输入符号的集合，记为 SELECT($A \to \beta$)[28]。

SELECT 有如下规则:

- SELECT($A \to a\beta$) = $\{a\}$
- SELECT($A \to \epsilon$) = FOLLOW(A)

FIRST 集

串首终结符，即串首第一个符号，并且是终结符，简称首终结符。

给定一个文法符号串 X，X 的串首终结符集 FIRST(X) 定义：

- 可以从 X 推导出的所有串首终结符构成的集合。
- 如果 $X \overset{*}{\Rightarrow} \epsilon$，那么 $\epsilon \in \text{FIRST}(X)$。

FIRST(X) 算法：不断应用下列规则，直到没有新的终结符或 ϵ 可以被加入到任何 FIRST 集合中为止。

- 如果 X 是一个终结符，那么 FIRST(X) = X。
- 如果 X 是一个非终结符，有 $X \to Y_1 \cdots Y_k \in P(k \geqslant 1)$，那么如果对于某个 i，a 在 FIRST(Y_i) 中，且 Y_i 前的所有非终结符的 FIRST 集中都存在 ϵ。就把 a 加入到 FIRST(X) 中。如果所有 Y_i 的 FIRST 集中都有 ϵ，就把 ϵ 加入到 FIRST(X)。
- 如果 X$\to \epsilon$，就把 ϵ 加入到 FIRST(X)。

有了 FIRST 集后，产生式 $A \to \alpha$ 的可选集 SELECT:

- 如果 $\epsilon \notin FIRST(\alpha)$, 那么 $SELECT(A \to \alpha) = FIRST(\alpha)$
- 如果 $\epsilon \in FIRST(\alpha)$, 那么 $SELECT(A \to \alpha) = (FIRST(\alpha) - \{\epsilon\}) \cup FOLLOW(A)$

[28] 也即遇到对应输入符号时可以使用这个产生式

4.3.4 LL(1) 文法

通过上面一系列讨论，我们可以找出满足构造不带回溯的自上而下分析的文法条件。

1. 文法不含左递归。
2. 文法中每一个非终结符 A 的各个产生式的候选首符集两两不相交。
3. 对文法中的每个非终结符 A，若它存在某个候选首符集包含 ϵ，则

$$\text{FIRST(A)} \cap \text{FOLLOW(A)} = \phi$$

如果一个文法 G 满足以上条件，则称该文法 G 为 LL(1) 文法。LL(1) 中第一个 L 表示从左到右扫描输入串，第二个 L 表示最左推导，1 表示分析时每一步只需向前查看一个符号。

对于一个 LL(1) 文法，可以对其输入串进行有效的无回溯的自上而下分析。假设要用非终结符 A 进行匹配，面临的输入符号为 a,A 的所有产生式为

$$A \to \alpha_1 | \alpha_2 | \cdots | \alpha_n$$

- 若 $a \in \text{FIRST}(\alpha_1)$，则指派 α_i 去执行匹配任务。
- 若 a 不属于任何一个候选首符集，则：
  - 若 ϵ 属于某个 $\text{FIRST}(\alpha_i)$，且 $a \in \text{FOLLOW(A)}$，则让 A 与 ϵ 自动匹配；
  - 否则，a 的出现是一种语法错误。

4.4 总结与题型

4.4.1 FIRST 集, FOLLOW 集, SELECT 集的计算

FIRST 集

$FIRST(X)$ 定义: 可以从 X 推导出的所有串首终结符构成的集合，如果能推出空集 (ϵ), 那么 $\epsilon \in FIRST(X)$。

$FIRST(X)$ 计算的三种情况:

- 首字符是终结符 $(E \to \alpha)$: 直接加入对应的 $FIRST$ 集中: $FIRST(E) = \alpha$。
- 首字符不是终结符 $(E \to A\alpha)$: 对应非终结符的 $FIRST$ 集加入到要求的 $FIRST$ 集中: $FIRST(E) = FIRST(A)$。
- 推导出空串 $(E \to \epsilon)$: 将空串加入到对应的 $FIRST$ 集中: $FIRST(E) = \epsilon$。

计算大致可以分两个步骤，第一步找能找全的非终结符和空串; 第二部将其他 FIRST 集的内容填入。

举个例子 (计算符号的 FIRST 集):

表 6 FIRST 集计算

产生式	第一步	第二步
$E \to TE'$	$FIRST(E) = FIRST(T)$	$FIRST(E) = \{(, id\}$
$E' \to +TE'\|\epsilon$	$FIRST(E') = \{+, \epsilon\}$	$FIRST(E') = \{+, \epsilon\}$
$T \to FT'$	$FIRST(T) = FIRST(F)$	$FIRST(T) = \{(, id\}$
$T' \to *FT'\|\epsilon$	$FIRST(T') = \{*, \epsilon\}$	$FIRST(T') = \{*, \epsilon\}$
$F \to (E)\|id$	$FIRST(F) = \{(, id\}$	$FIRST(F) = \{(, id\}$

在计算符号的 $FIRST$ 集的基础上，还可以计算串的 $FIRST$ 集，例如计算 $X_1X_2 \dots X_n$ 的 FIRST 集:

- 加入 $FIRST(X_1)$ 中的所有非 ϵ 符号
- 如果 ϵ 在 $FIRST(X_1)$ 中，则加入 $FIRST(X_2)$ 中的所有非 ϵ 符号
- 以此类推
- 如果直至 ϵ 在 $FIRST(X_n)$, 则将 ϵ 加入串的 $FIRST$ 集中

FOLLOW 集

$FOLLOW(X)$ 集定义: 所有句型中出现在紧接 X 之后的终结符, 若 X 是某个句型的最右符号，则加上结束符 $\#$。

计算 $FOLLOW$ 集首先需要计算 $FIRST$ 集，然后对每个产生式从左到右进行如下分析[29]:

- 如果是最右符号，加入结束符 $\#$。
- 将紧挨着的下一个非终结符的 $FIRST$ 集中的非终结符 (不包含 ϵ) 加入到前一个非终结符的 $FOLLOW$ 集中。
- 如果某个非终结符的 $FIRST$ 集中包含 ϵ，则需屏蔽该非终结符进行分析。
- 表达式右边的最后一个非终结符的 $FOLLOW$ 集包含表达式左边最后一个非终结符的 $FOLLOW$ 集。
- 如果非终结符后面跟着终结符，则将该终结符加入到非终结符的 $FOLLOW$ 集中。
- 不断迭代，直至所有非终结符的 $FOLLOW$ 集不再更新。

表 7 FOLLOW 集计算

产生式	$FIRST$ 集	$FOLLOW$ 集
$E \to TE'$	$FIRST(E) = \{(, id\}$	$FOLLOW(E) = \{\#,)\}$
$E' \to +TE'\|\epsilon$	$FIRST(E') = \{+, \epsilon\}$	$FOLLOW(E') = \{\#,)\}$
$T \to FT'$	$FIRST(T) = \{(, id\}$	$FOLLOW(T) = \{+, \#,)\}$
$T' \to *FT'\|\epsilon$	$FIRST(T') = \{*, \epsilon\}$	$FOLLOW(T') = \{+, \#,)\}$
$F \to (E)\|id$	$FIRST(F) = \{(, id\}$	$FOLLOW(F) = \{*, +, \#,)\}$

[29]这里有更详细的讲解: https://www.bilibili.com/video/BV1zW411t7YE?p=22

SELECT 集

$SELECT$ 集计算的三种情况:

- 产生式右部首字符是终结符: $SELECT$ 集是对应终结符。
- 产生式右部首字符是非终结符: $SELECT$ 集是对应非终结符的 $FIRST$ 集中的非 ϵ 的终结符。
- 产生式右部首字符是 ϵ: $SELECT$ 集是左部的 $FOLLOW$ 集。

表 8 SELECT 集计算

产生式	$SELECT$ 集
$(1)E \to TE'$	$SELECT(1) = \{(, id\}$
$(2)E' \to +TE'$	$SELECT(2) = \{+\}$
$(3)E' \to \epsilon$	$SELECT(3) = \{\#,)\}$
$(4)T \to FT'$	$SELECT(4) = \{(, id)\}$
$(5)T' \to *FT'$	$SELECT(5) = \{*\}$
$(6)T' \to \epsilon$	$SELECT(6) = \{+,), \#\}$
$(7)F \to (E)$	$SELECT(7) = \{(\}$
$(8)F \to id$	$SELECT(8) = \{\#\}$

判断是否为 LL(1) 文法

判断是否为 LL(1) 文法只需要看左部相同的产生式的 $SELECT$ 集是否相交。不相交，则是 LL(1) 文法。

预测分析表

根据 $SEELCT$ 集可以构建预测分析表，看表怎么画就知道这是啥了:

表 9 预测分析表

非终结符	输入符号					
	id	+	*	(	)	#
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \#$	$E' \to \#$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \#$	$T' \to \#$
F	$F \to id$			$F \to (E)$		