高等学校计算机专业规划教材

C编译器剖析

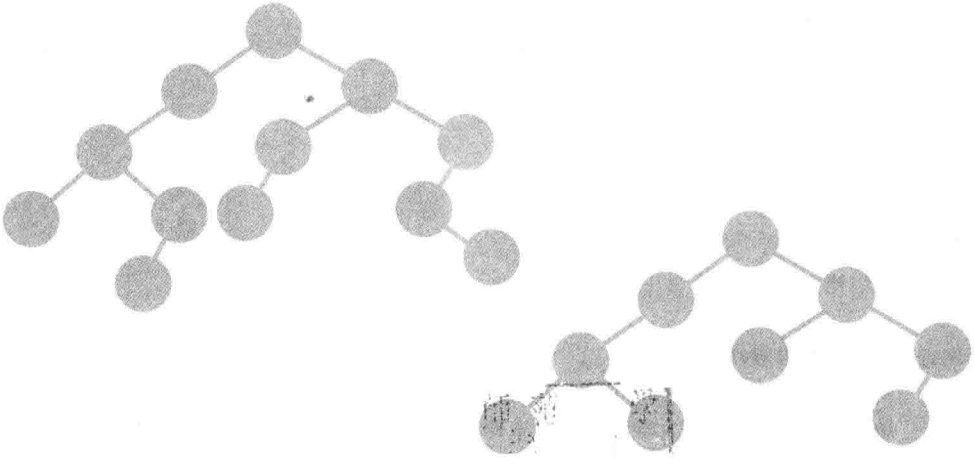
邹昌伟　编著

清华大学出版社

高等学校计算机专业规划教材

C编译器剖析

邹昌伟　编著



清华大学出版社
北京

内 容 简 介

“编译原理”课程是一门理论性与实践性非常强的课程，应遵循从具体到抽象的认知规律。本书以一个开源的C编译器(UCC)为案例，在源代码分析的过程中，展开对编译原理相关知识的学习和讨论。全书共分6章：第1章介绍文法和递归等知识点，并采用结合C语言学汇编的方式来讨论汇编代码；第2章讨论UCC编译器的词法分析、内存管理、符号表管理和类型系统等基本模块；第3章介绍UCC编译器的语法分析，采用的是手工打造分析器的技术路线；第4章介绍语义检查，通过本章的学习，有助于C程序员站在编译器的角度来深入理解C语言的语义规则；第5章分析UCC编译器的中间代码生成及优化；第6章介绍如何生成32位的x86汇编代码。

本书不仅是很好的编译原理和编译器设计教材，也可作为深入学习C程序设计的参考用书。

图书在版编目(CIP)数据

C编译器剖析/邹昌伟编著. —北京：清华大学出版社，2016
高等学校计算机专业规划教材
ISBN 978-7-302-42610-3

Ⅰ. ①C…　Ⅱ. ①邹…　Ⅲ. ①C语言－编译器－程序设计　Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2016)第005328号

责任编辑：龙启铭　战晓雷
封面设计：何凤霞
责任校对：梁　毅
责任印制：杨　艳

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦A座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954
印 装 者：北京鑫海金澳胶印有限公司
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：25.25　　**字　　数**：630千字
版　　次：2016年6月第1版　　**印　　次**：2016年6月第1次印刷
印　　数：1～2000
定　　价：49.00元

产品编号：066842-01

序

我是在大三的时候开始学习“编译原理”课程的，授课的是陈意云老师，使用的是陈老师自己编写的教材。这本教材理论知识非常丰富，每章后面有不少颇有难度的练习题。非常感谢陈老师讲的这门课程，我有限的编译原理理论知识基本上是从这门课程学的。但学完之后，总有一个感觉，即自己掌握的知识比较分散，不能完整地把它串起来。与国内其他高校差不多，这门课的大作业是一个玩具语言的编译器实现，之所以说是玩具语言，是因为语言本身很简单，缺乏很多实用的特性，无法作为实际的编程语言使用。当时就有一个模糊的想法，想自己写一个实用编程语言的编译器，这个编译器不能复杂，得适合在一个学期的时间中学习和掌握，但一直停留在想法这个阶段，也没想好实现哪个语言。

我在后来的学习和工作中一直和系统编程打交道，使用的基本上是C语言。对C语言的一些精巧特性也越来越熟悉，比如说“为什么函数标准的入栈顺序是从右到左”和“setjmp/longjmp是如何实现的”等。C语言的使用非常广泛，而且以精练著称，如果能写一个C编译器就很好了。开源的GCC非常复杂，基本上不适合学习。其间我完整地阅读了LCC的源代码，LCC功能完整，代码精简。但是LCC代码本身比较难懂，结构不清晰。我在这个时候就明确下来写一个适合学习的C编译器，这个编译器得有如下几个要点：

(1) 用C语言来实现C编译器，一是因为我最熟悉的是C语言，二是能实现自举(bootstrap)，这是对编译器很好的测试。

(2) 代码简洁易懂，结构清晰，适合学生学习和掌握。

(3) 实现ANSI C89标准。

(4) 一定要开源，因为我在学习和工作中受到开源社区的帮助很大。

(5) 编译器的核心难点和复杂度在后端优化，这是一个简单的用来学习基本原理的编译器，基本不涉及后端优化。

我从2007年开始在闲暇时间构思编写UCC，即Your C Compiler的缩写，断断续续用了一年半的时间完成。完成后我很开心，第一时间将其开源在sourceforge网站上，后来因为各种缘故，开源后一直没有维护。

现在非常高兴地看到邹老师以UCC编译器源码为基础，写了这本理论结合实践的书。这本书不是简单的源码剖析，它以编译原理的理论作为主线把整个知识串起来，辅以实际代码的讲解。我相信这种方式对于在校学生或者编译器爱好者非常有帮助。可能绝大部分程序员在职业生涯中不需要

去写一个完整的编译器，但是编译器的一些理论和算法在很多领域得到使用，掌握了编译原理的基础知识之后，会对代码的优化和效率有更好的理解，而且容易掌握一些比较难的语言特性，比如说 LISP 语言中的闭包和 continuation 等概念。

语言是对计算机硬件特性本身的一个抽象，汇编语言抽象的层次低，C 语言抽象的层次高。整个编译器的轴心就是如何把一个高阶的抽象逐渐变成一个低阶的抽象。这个过程不是一步到位的，是经过了词法分析、语法分析和语义分析等步骤逐渐完成的。每一步可以看作是把一个高阶的抽象语言变成一个稍低阶的抽象语言。带着这样一个主线去学习邹老师的这本书和 UCC 源代码，相信大家能很好地掌握编译原理。

小米公司 MIUI 首席架构师、UCC 编译器作者

汪文俊

前言

自从1999年在Turbo C 2.0下第一次用C语言写出Hello World以来，不知不觉在IT相关行当里混了近16年了。相信所有上机写过Hello World的人当初都会有这样的好奇心：由键盘敲进去的不过是一个个普通的英文字符，C编译器是如何神奇地发现哪一行出现了什么样的语法错误，更神奇的是编译连接后的可执行程序又是怎样在操作系统上运行起来的？这些问题也一直萦绕在我脑海里，挥之不去。大学时所学的“操作系统”和“编译原理”课程似乎对这些问题给出了答案，似乎又没有。很遗憾，那时也没有人告诉我，想比较彻底地搞明白这些问题，最好的办法是去读编译器或操作系统的源代码。因为“计算机科学与技术”这一学科虽然冠上了“科学与技术”之名，但实际上还是“技术”的成分要来得多些。在所有强调“技术”的工作中，“技术”通常只能来源于长期的动手实践。即便是开车倒库这样的操作也是个技术活，没有长期动手实践是倒不好车的。纸上得来终觉浅，绝知此事要躬行。而不少牛人在大学里上完“操作系统”和“编译原理”课程后就已脱颖而出，比如大牛Linus和Chris Lattner，前者大学时就建立了他的Linux操作系统王国，而后者也在研究生期间缔造了如今在业界如日中天的LLVM编译器。Apple公司已经不动声色地把公司的Object-C编译器从GCC转成了LLVM和Clang，各位的iPhone手机中运行的代码可能正是LLVM和Clang的产物呢。Apple公司新推出的Swift语言背后站着的仍然是Charis Lattner和LLVM。

能写出工业水准的操作系统和编译器的人绝对是大牛，而写出来的操作系统和编译器能被工业界普遍接受，则需要命运和机遇的垂青。大部分程序员注定是成为高级或低级的“码农”，成为软件生产流水线上的一颗螺丝钉。在生活和工作的压力面前，我们往往习惯于不做那么深入思考，只要代码能实现需求就可以，至于“为什么”的问题，人们往往有意或无意甚至被迫地将之忽略。就如儿时的梦想，只有在夜深人静时，想着年华已去，看着岁月无情，捧着那泛黄的旧照片时，才发现一直以来自己内心不常去的角落中始终留着那个梦想。

操作系统和编译器就如武侠小说中的“九阴真经”，没看过“九阴真经”的侠客也可以行走江湖，但看过并练成九阴真经的人最终才更有机会登上华山之巅。我们很幸运能生活在互联网时代，不必如20世纪七八十年代的欧美程序员那样为了一睹UNIX操作系统源代码的风采，私下偷偷复印那本

后来被称为旷世奇书的《莱昂氏 UNIX 源代码分析》。其实，真正旷世之作的是那本书中分析的 UNIX 操作系统和用来写 UNIX 的 C 语言。当然，我们更不必为了一睹“九阴真经”而在江湖中掀起血雨腥风。因为九阴真经就摆在我们的面前——GCC、LLVM 和 Linux 源代码就毫无保留地躺在那儿。冲动的我们一定曾经兴奋地下载了这些源代码，但面对这些动辄几百万行的源代码时，才发现自己是多么渺小和无力。其实再复杂的系统，其最初的原型往往也并不复杂，就如早期的 UNIX 操作系统和 C 编译器，而这些原型往往就是最精髓的部分，包括了最初的直觉和创意。与其面对几百万行的源代码感叹自己“too young, too simple”，不如选择一本浓缩的“九阴真经”。很幸运，只要我们打开 Google 搜索，就能找到这些原型版的操作系统和编译器。

如果要分析的是开源的、相对完整的 C 编译器源代码，而不是一个教学用的玩具，并且希望源代码的行数在一万行左右，那 Google 给我们的答案中就有这么两个，一个是 LCC 编译器，另一个是 UCC 编译器。如果我们还希望代码结构清晰，且函数名等能见名知义，能直接与 C 语言的文法吻合，不必去猜测这个函数名是哪几个单词的缩写，那 UCC 会更胜一筹。但很遗憾，UCC 的原作者汪文俊（wenjunw123@gmail.com）没有写关于 UCC 的书，他其实是最合适的人选。于是，UCC 就如一颗被遗落在角落的珍珠，渐渐地蒙上了尘埃。一万行左右的代码量，UCC 就能实现一个基本完整的 C 编译器，虽然其后端只面向 32 位的 x86 平台，也没有如 LCC 那样的可变目标——既可生成 MIPS，也可生成 SPARC 和 x86 代码。尽管偶有缺陷，纵然碧玉有瑕，即便不够知名，但只要把 UCC 这只麻雀解剖清楚，就能搞清 C 编译器是如何工作的，能站在编译器实现的角度来品味经 40 余年而历久弥新的 C 语言。C 和 C++ 语言的相关书籍中，最经典的有 *The C Programming Language*、*C++ Primer* 和 *The C++ programming language*，之所以经典，很大原因是这些作者本身就是 C 或 C++ 编译器的最早期实现者。

越俎代庖去写关于 UCC 编译器的书，希望这本书能帮助有兴趣读它的人基本搞明白 C 编译器，用好 C 语言这个简洁有力的利器。当然，正如一位在编译一线战斗的朋友所言，“UCC 和 LCC 还停留在编译前端和后端的初级阶段”，编译器的优化是现在工业界更加关注的焦点，UCC 和 LCC 在后端优化上都做得不够。不过，站在广大 C 程序员的角度来看，C 编译器的前端才是 C 编译器与程序员的接口，理解了 C 编译器前端的实现，就能更深刻地理解 C 语言的语法和语义；而 C 编译器后端其实是 C 编译器与 CPU 的接口，后端优化的目标往往是在保持语义的前提下生成速度更快的机器代码。我不是什么牛人，只希望这本书能帮有心人开启一扇看得见、够得着的编译大门，让 C 程序员能站在编译器实现的角度来看看自己朝夕相伴的 C 语言。全国每年有十万名以上的计算机相关专业学生毕业，“编译原理”课程在大多数毕业生中留下的印象就是很难及很理论化。本书的目的是想把“编译原理”课程具体化，所谓“伤其十指，不如断其一指”。如果你熟悉 C 语言，并且有基本的数据结构的知识，那就和我们一起开启 C 编译器剖析这趟旅程，学习编译原理和汇编语言相关的知识，期盼这趟行程结束后，不论你的下一站去何方，你都能觉得不虚此行。众里寻她千百度，蓦然回首，这就是 C 编译器。那我们就启程吧！

衷心感谢文俊兄在百忙中为本书写的序。从 UCC 编译器的源代码中，我受益颇多。在 UCC 源代码的阅读、分析和修改过程中所感受的快乐，像无形的指挥棒驱使着我把其

中的心得和体会写在博客 http://blog.csdn.net/sheisc 上。非常感谢清华大学出版社提供的出版机会，十分感谢本书的责任编辑龙启铭老师在本书出版过程中提供的帮助和指导。作为一名在教育一线工作的普通教师，我诚挚地希望拙作能传递这份快乐和收获，能为想深入理解 UCC 编译器的朋友们提供一点帮助。书中难免会有不足之处，敬请读者给予批评指正。

邹昌伟

目录

第1章 基础知识

1.1 语言、文法与递归

我们对语言这个概念再熟悉不过了，汉语和英语等都是自然语言。而C和C++语言是编程语言，若用更专业的术语，则这些编程语言可被称为“形式语言”(formal language)。formal这个单词的原意是“正式的”，非常准确地表达了形式语言和自然语言的区别。C和C++等编程语言之所以被称为形式语言，其原因在于这些语言的产生过程是先有正式的语法规则，之后才由这些语法规则产生语言。而汉语和英语则正好相反，原始人从树上刚走下来时，不可能先开会讨论一下语法规则，最原始的呼喊经数万年的演化和交融，自然而然地形成了汉语和英语等自然语言。若干年之后，广大语文老师们才聚在一起研究语法规则的问题。这种行为当然不够正式了。这也是为什么自然语言的识别会成为人工智能领域中一个比较麻烦的问题，而C语言的识别从一诞生就能得到很好解决的原因之一。

计算机相关学科的人会在大二时学习“离散数学”课程，其中一个很重要的概念就是集合，这个曾引起第三次数学危机的名词是个放之四海皆准的名词。我们要讨论的语言实际上也是一个集合。一般而言，有两个方法来表示集合：一个是列举法，另一个是描述法。对于个数有限的集合，可以用列举法一一列出；而对于无穷的集合，则多用描述法来表达。例如，$S=\{$张三，李四，王五$\}$构成了一个有限集合；而要表达偶数这样的无穷集合，则使用$E=\{x|x=2n,n$是非负整数$\}$这样的描述法。

既然要把C语言看成一个集合，那么每个合法的C源代码就是这个集合里的一个元素，合法的C源代码有无穷多个，那要用什么样的描述方法来表达C语言这个无穷集合？不妨先看看下面这个比C语言简单得多的语言：

$$T=\{a,a+a,a+a+a,\cdots\}$$

不难发现，上述集合T由若干个a相加构成，这是一个无穷集合，省略号“…”的意思是“这个集合的元素太多了，我们列举不下，你自个儿研究已列出的元素，然后去找规律吧”。这种“你猜你猜你猜猜”的描述方法显然不能让追求完美的我们感到满意。下面就来探索一下有没有更好的表达方法。先把集合T一分为二：

$$T=\{a,a+a,a+a+a,\cdots\}=\{a\}\cup\{a+a,a+a+a,\cdots\} \tag{1-1}$$

仔细观察其中的$\{a+a,a+a+a,\cdots\}$，可以发现它与T有点相似。如果把$\{a+a,a+a+a,\cdots\}$中每个元素的前缀$a+$移到大括号之前，就有

$$T = \{a\} \cup a + \{a, a+a, a+a+a, \cdots\} = \{a\} \cup a + T$$

这是一个对集合 T 的递归定义。到此，我们把语言、集合和递归关联到了一起。稍微整理一下，把这个无穷集合 T 写为

$$T \rightarrow a \mid a + T \tag{1-2}$$

式(1-2)表明，T 由 a 或者 $a+T$ 构成。该式实际上由两个式子构成，$T \rightarrow a$ 和 $T \rightarrow a+T$。也可分成多行来写，记为

T：

a

$a+T$

如果我们想到事物往往是有阴就有阳，有左就有右，有前就有后，就会尝试看看能不能做其他类似的变换。仔细观察式(1-1)的右部，还可以把 $+a$ 作为后缀提取到大括号之后，于是又有了

$$T = \{a\} \cup \{a, a+a, a+a+a, \cdots\} + a = \{a\} \cup T + a$$

稍作整理，可以得到下式：

$$T \rightarrow a \mid T + a \tag{1-3}$$

式(1-2)和式(1-3)的区别在于，前者的递归出现在 $a+T$ 右侧，而后者的递归定义出现在 $T+a$ 左侧，所以式(1-2)称为右递归，式(1-3)称为左递归。由这两个式子都能产生语言 T，即都可以生成以下集合中的各个元素：

$$\{a, a+a, a+a+a, \cdots\}$$

形如式(1-2)和式(1-3)这样的式子就称为产生式。有限的符号却精准地描述了无穷的集合，这非常符合老子在《道德经》所言的“一生二，二生三，三生万物”。让我们享受一下由产生式产生句子的过程。例如，要判断字符串 $a+a$ 是否是语言 T 中的一个合法句子，即判断字符串 $a+a$ 是否是集合 T 的一个元素，可以从 T 出发，做以下推导。式(1-4)和式(1-5)分别是以式(1-2)和式(1-3)作为产生式进行推导的结果。

$$\begin{aligned} T &\rightarrow a + T \\ &\rightarrow a + a \end{aligned} \tag{1-4}$$

$$\begin{aligned} T &\rightarrow T + a \\ &\rightarrow a + a \end{aligned} \tag{1-5}$$

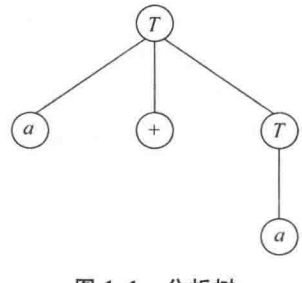


图 1.1 分析树

每一步的推导都是用某个产生式的右侧来替换左侧。图 1.1 就用一棵树来形象地表述了式(1-4)的推导过程，该树称为分析树(parsing tree)。可以通过这棵树来分析字符串 $a+a$ 是否为语言 T 的合法句子。对这棵树作进一步观察，可以发现，树中的有些结点可以向外生长，例如其中标为 T 的结点；而有些结点则不具备生长能力，例如其中标为 a 和 $+$ 的结点。因此，称式(1-2)中的 T 为非终结符，而 a 和 $+$ 为终结符。式(1-2)由两个产生式构成，即 $T \rightarrow a$ 和 $T \rightarrow a+T$，它们共同刻画了语言 T 的特征，是关于如何生成语言 T 的法则，所以称为文法(grammar)。当然，严格地说，文法由终结符、非终结符、产生式和开始符号构成。开始符号是要描述的集合对应的非终结符，此处即为 T。由图 1.1 可以形象地看到，开始符号为

树根 T，这是做推导起步的地方，也是语言中所有合法句子的始祖，所以称为开始符号。实际上，如果约定第一个产生式左侧的非终结符为文法的开始符号，则产生式就包含了文法的所有信息。以后的表述中，在不致混淆的情况下，不再区分文法和产生式。

如果任给一个字符串，我们当然不希望每次都人工来判断它是否为合法的句子。需要根据文法来写一个程序来做这些事情，这个程序称为语法分析器(syntax parser)，通常就简称为分析器(parser)。图 1.2 分别给出了为左递归和右递归文法编写语法分析函数的算法，其中的语法图作为一种中间过渡，由语法图可很直观地写出相应的分析函数，其基本的原则很简单：

(1) 对于文法中的非终结符，由于它代表的是一个集合，要判断某个字符串是否属于这个集合，则会构造一个与非终结符同名的函数来处理。在语法图中遇到非终结符时(对应语法图中的矩形)，例如图 1.2 中的 T，则直接调用函数 $T()$来进行语法分析。

(2) 对于文法中的终结符(对应语法图中的圆形)，例如图 1.2 中的 a 和＋，则直接调用函数 Expect 进行终结符的匹配。

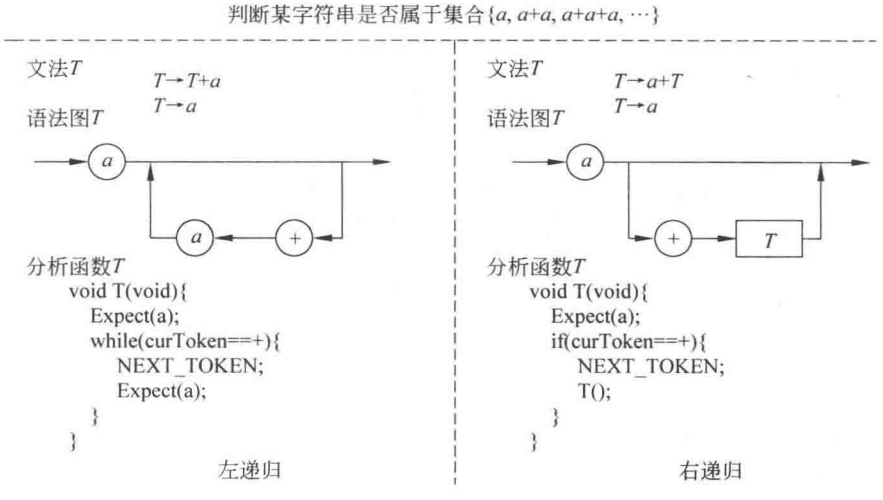


图 1.2　为非终结符 T 编写语法分析函数

图 1.2 中用于终结符匹配的函数 Expect()的算法如下，其中，NEXT_TOKEN 表示读取一个单词并保存到全局变量 curToken 中，TokenKind 表示单词的类别。

```
void Expect(TokenKind tk){
    if(curToken==tk){
        NEXT_TOKEN;
    }else{
        Error();
    }
}
```

需要注意的是，若加法运算是左结合的，对于 $a+a+a$ 来说，按照图 1.2 左侧的分析

函数，实际上用到的分析树如图 1.3 右侧所示，而图 1.3 左侧的分析树是在理论上做推导时用到的，真正上机实现时需要消除左递归。

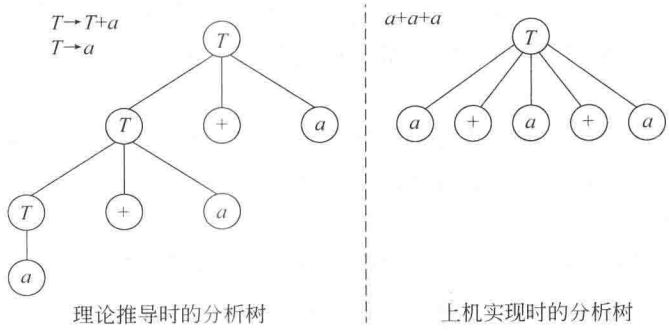


图 1.3　a＋a＋a 对应的分析树

在熟悉图 1.2 这样的套路后，就可不再画语法图，而是直接由文法来编写相应的语法分析函数。在 1.2 节，将会引入一个稍为复杂一点的文法，然后介绍如何基于这个文法来写一个语法分析器。

1.2　一个较复杂的文法

在 1.1 节中讨论了语言、集合、文法和递归之间的关系，在本节中，先给出一个较复杂的文法，该文法描述了一个包含语句、表达式和声明的程序，如图 1.4 所示。

```
Program→CompoundStatement
Statement→IfStatement|WhileStatement|CompoundStatement
|ExpressionStatement
IfStatement→if(expression) Statement
IfStatement→if(expression) Statement else Statement
WhileStatement→while(expression) Statement
CompoundStatement→{ StatementListopt}
StatementList→Statement|StatementList Statement
ExpressionStatement→id=Expression ;
ExpressionStatement→Declaration ;
Expression→AdditiveExpression
AdditiveExpression→MultiplicativeExpression
AdditiveExpression→AdditiveExpression+MultiplicativeExpression
AdditiveExpression→AdditiveExpression-MultiplicativeExpression
MultiplicativeExpression→PrimaryExpression
MultiplicativeExpression→MultiplicativeExpression * PrimaryExpression
MultiplicativeExpression→MultiplicativeExpression/PrimaryExpression
PrimaryExpression→id|num|(Expression)
Declaration→ int Declarator
```

图 1.4　Program 文法

```
Declarator→* Declarator |PostfixDeclarator
PostfixDeclarator→DirectDeclarator|PostfixDeclarator [num]
                  |PostfixDeclarator (void)
DirectDeclarator→id|(Declarator)
```

图 1.4 （续）

按照前面的约定，图 1.4 中的第一个产生式左侧的非终结符 Program 为文法的开始符号。下面看一个例子，就能初步明白这个看起来似乎很复杂的文法究竟刻画了一个什么样的语言。

```
{
    int (*f(void))[4];
    int (*(*fp2)(void))(void);
    if(c)
        a=f;
    else{
        b=k;
    }
    while(c){
        while(d){
            if(e){
                d=d-1;
            }
        }
        c=c-1;
    }
}
```

请从 http://download.csdn.net/detail/sheisc/8669715 下载 ucc162.3.tar.gz，解压后的目录 ucc\examples\sc 中包含的代码即根据图 1.4 构造出来的语法分析程序。在这个看似复杂的文法中，不少产生式与 1.1 节的产生式(1-2)及产生式(1-3)本质上是相同的。

例如，对以下几个从图 1.4 取出的产生式来说，如果把 AdditveExpression 视为 T，把 MultiplicativeExpression 当作 a，其形式上就与产生式(1-3)是一样的。如前文所述，产生式(1-3)是一个左递归的产生式。虽然由式(1-2)和式(1-3)生成的语言是相同的，但是左递归的产生式隐含了其运算符是左结合的，而右递归的产生式隐含了其运算符是右结合的。按习惯，加减乘除四则运算符都是左结合的，所以选取形如式(1-3)的产生式来表达加法运算。与 T 所表达的集合类似，AdditiveExpression 由若干个 MultiplicativeExpression 相加构成。换言之，要进行加法运算，需要先得到 MultiplicativeExpression。同理，MultiplicativeExpression 由若干个 PrimaryExpression 相乘构成，这意味着要进行乘法运算，就要先得到 PrimaryExpression。而 PrimaryExpression 则由标志符 id、数 num 和“加括号的表达式”构成。按这样的层次结构，以下产生式实际上隐含了加减乘除运算符之间的优先级。加、

减视为同一优先级，按结合性进行从左到右的结合；乘、除视为同一优先级，也采用左结合。用与此类似的方法，还可以引入更多的运算符，如 & 和 | 等。

```
AdditiveExpression→MultiplicativeExpression
AdditiveExpression→AdditiveExpression+MultiplicativeExpression
AdditiveExpression→AdditiveExpression-MultiplicativeExpression
MultiplicativeExpression→PrimaryExpression
MultiplicativeExpression →MultiplicativeExpression * PrimaryExpression
MultiplicativeExpression →MultiplicativeExpression/PrimaryExpression
PrimaryExpression→id|num|(Expression)
```

下面再来分析图 1.4 中与“语句”有关的产生式。由 Statement 的产生式可知，语句由 if 语句、while 语句、复合语句及表达式语句构成。而 if 语句以 if 开头，后面紧跟左括号，然后是表达式 expression，接着是右括号，之后又是一个语句 Statement。这里，再次在产生式中看到递归定义。

```
Statement→IfStatement|WhileStatement
          |CompoundStatement| ExpressionStatement
IfStatement→if(expression) Statement
IfStatement→if(expression) Statement else Statement
WhileStatement→while(expression) Statement
CompoundStatement→{ StatementList_opt }
StatementList→Statement|StatementList Statement
```

而 StatementList 对应的产生式实际上形如前面的产生式(1-3)。复合语句 CompoundStatement 中的 $StatementList_{opt}$ 表示 StatementList 是 Optional 的，即可有可无。接下来讨论与声明 Declaration 相关的产生式，在 C 语言中，通过声明表达了“C 程序员自定义的某个标志符是什么类型”的概念。

```
Declaration→ int Declarator
Declarator→ * Declarator |PostfixDeclarator
PostfixDeclarator→DirectDeclarator|PostfixDeclarator [num]
                  |PostfixDeclarator (void)
DirectDeclarator→id|(Declarator)
```

而 PostfixDeclarator 则再次与产生式(1-3)相似。其产生式实际上表明 PostfixDeclarator 由 DirectDeclarator 后面跟任意多个[num]或者(void) 构成。而 Declarator 对应的产生式则由若干个 * 后面再跟一个 PostfixDeclarator 组成。通过形如[num]的后缀声明了一个大小为 num 的数组；而通过形如(void)的后缀声明了一个无参的函数；而在声明中使用 *，实际上是声明了一个指针。

例如，对于字符串 int aa[30][50]，可以按前文画分析树的方法，由 Declaration 出发作推导，最终生成 int aa[30][50]，其中 30、50 皆为数 num，而 aa 为标志符 id。还可以由 Declaration 生成 int (* cc)[3][5]。按 C 的语法，aa 是数组，而 cc 是指向数组的指针。