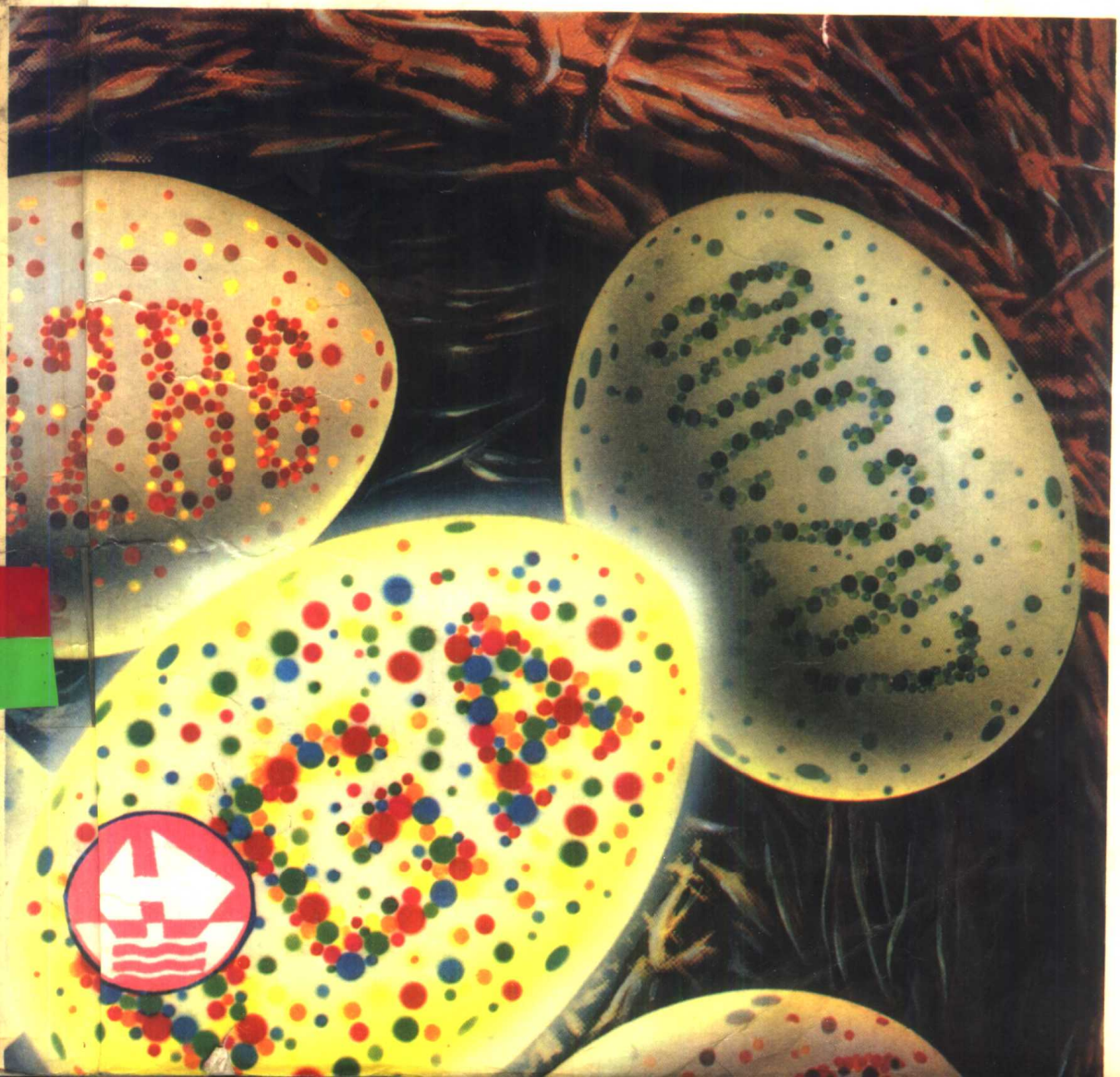
程序设计高级教程

编译原理与实践

亚文　张勇　建平　编
顾铁成　王岩冰　审校



中国科学院希望高级电脑技术公司

程序设计高级教程

编译原理与实践

亚文　张勇　建平　编
顾铁成　王岩冰　审校

- 高级程序设计人员必读
- 详尽而深入浅出的理论阐述
- 二万余行的编译器模型实现

中国科学院希望高级电脑技术公司

■北京市新闻出版局
准印证号：3188—90188
■订购单位：北京8721信箱资料部
■邮　　码：100080
■电　　话：2562329
■乘　　车：320、332、302路车至海淀黄庄下车
■办公地点：希望公司大楼101房间

前 言

编译器（又称编译程序）是计算机软件的一个极为重要的组成部分，本书较为系统地介绍了这方面的知识。第一章介绍了一般的概念；第二章介绍编译器的输入与词法分析；第三章介绍上下文无关文法的内容；第四、五章讲述了自顶向下与自底向上分析技术；第六章的内容主要涉及代码生成，第七章讨论了优化策略。附录给出了一些相关的内容及几个实用工具—LEX、Curses、LLama和Occs。

本书浅显易懂，理论与实践紧密联系。本书以一个 *C* 语言的编译器模型实现贯穿始终，所给出的 *C* 编译程序简明、易读，是一个理想的编译程序教学模型，也可略加改造而成为一个实用的产品。

本书既可供高等院校有关专业学生、研究生和教师作为教学参考书，也是广大科技工作者的一本理想的参考书。本书还适合广大计算机用户学习、了解编译程序时使用。

为配合本书的使用，书中所给的 *C* 语言编译器模型已在 IBM-PC 微机上用 Microsoft C 5.0 版实现。

衷心感谢中科院希望电脑公司资料部经理秦人华、杨淑欣为本书出版所给予的支持。

目　录

绪论

本书以编译器设计为主题，并直接面向程序设计者．该书的基本出发点是，要想设计一个编译器，最好的途径是深入分析一个编译器；要想理解某种理论，最好的方法是构造一个使用这种理论的工具．因此，本书列出了许多实用代码，以阐明如何应用其中提到的理论．在描述理论时，尽量避免使用数学术语，而采用自然语言的方式，并应用程序代码解释各种处理过程．对于不易弄清楚的理论，可参阅实现该理论的源代码．当然，书中所列的代码并不是实现相应理论的唯一方法，但是，在理解理论时参阅源代码可能会深受裨益；而且，读者可以根据这些概念重新构造源代码．

这样组织本书的缺陷在于，可能其中含有大量细节性的内容．但是，对于实际构造一个编译器而言，这部分细节极为重要，而且它正是与编译器设计有关的其它书目中所没有的．同样，多数细节针对一般性的程序设计，而不是针对某个特定的编译器．但是，设计编译器时所应注意的另外一个方面是，如何集成大型的复杂程序，而不是仅仅考虑与编译器直接相关的各个组成部分．如果读者无意阅读细节部分，跳过源代码章节即可．

一般情况下，本书只注重代码的可读性，而不考虑其执行效率．亦即，既然本书的宗旨是讨论如何设计编译器，所以，在考虑可理解性的基础上再考虑代码的执行效率似乎比较合适．例如，在讨论 LEX 时，使用 Thompson 构造方法设计 DFA 而不使用 Aho 方法，因为相对而言，Thompson 构造方法更易于理解(同时，它也引导出某些关键概念[例如闭包]，而且显得很自然)．计算 LALR(1)超前查看符号的算法也没有达到它应有的效率，当然，读者也可以自行设计出更有效的算法．

事实上，本书深入讨论了三种编译器生成工具的源代码，同时构造了完整的 C 语言编译器．(以 UNIX 中的 lex 实用程序为模型的词法分析程序生成器，两个与 Yacc 类似的编译程序生成器) 所以，与其它有关书籍不同——本书的重点是指导读者如何编制实际的编译器——即侧重于编译器工程．另外，本书涉及许多理论，内容集中而且面向实际应用．尽管书中提到了完整的编译程序，有助于读者理解各种理论，但这些细节问题并不是本书的重点所在．重要的是，如何在用户程序中应用这些处理方法．

设计实用程序是为了辅助读者学习编译原理．例如 Lama 和 occs (两个编译程序生成器) 可以创建一个面向窗口的交互式调试环境，其中嵌入了一个"可见的语法分析器"，以监视语法分析过程．(一个窗口显示状态栈和值栈，其它窗口显示输入、输出以及语法分析进程．) 用户可以观察语法分析栈的变化，监视继承属性和综合属性，设置条件断点 (根据输入符号、产生式归约、栈顶符号等等)，如有必要，还可以将整个语法分析过程输出到文件中．监测自底向上的语法分析过程极有助于理解语法分析步骤，并且为了追求好的效果，本书的语法分析器常常被设计成透明的形式．

第六章列出的 C 编译器实现了与 ANSI 兼容的 C 子集——其中滤掉了浮点数处理，因为它使代码量急剧增加．本书尽量涉及所有难以实现的细节问题，这正是本书有别于其它书籍的特色之一．例如，它支持包括结构和其它复杂说明在内的完整的 C 语法说明．类似地，它也涉及到嵌套块说明和更复杂的嵌套控制结构说明，例如开关语句等．

所有源代码都是 ANSI C，且与 UNIX 兼容．例如，窗口管理是在 IBM-PC 机上模拟标准 UNIX 窗口管理包而成的，其中还提供了完整的仿真程序的源代码．所有软件均适宜于在 UNIX、MS-DOS 和 Macintosh 环境中运行．

全书都假定读者将要阅读源代码以及源代码中的注释行．对于代码本身已有详尽的实现细节，正文部分将不再另加阐述．但是，对于程序中意思含混的部分，本书作了一些说明．

预备知识

阅读该书之前，读者应当了解 ANSI C 和一般的程序设计知识．除了 C 语言本身之外，还要熟悉 ANSI 标准中描述的标准库函数．

全书采用结构化程序设计技术，并遵从数据抽象原则，但书中没有具体讲述这些技术以及使用这些技术的原因．正文部分全面地解释了复杂的数据结构，但没有解释对于一些基本数据结构的先行知识，例如堆栈

和二叉树等．同样，了解集合论和图论的基本知识也将有所帮助．最后，读者可以预先熟悉汇编语言概念(例如子程序调用过程)，但没有必要对某种汇编语言作更深入的了解，因为本书采用的是C语言子集，而不是汇编语言．

尽管读者有必要了解C语言，但对UNIX或MS-DOS操作系统的了解则不是必须的．仅在构造UNIX工具等方面，本书才是面向UNIX操作系统的．顺便指出，这里深入讨论了基于MS-DOS的实现细节及其兼容性，源代码本身也尽量与MS-DOS兼容．

组织

读者可以用两种方式使用本书．如果对理论部分不感兴趣，而只希望构造一个实实在在的编译器，则可以浏览第一章中对一般编译器设计过程的概述，参阅附录D和附录E中对编译器构造工具（LEX和occs)的使用说明，查看第六章中对代码生成技术的讨论．在阅读完这些章节的内容之后，读者就可以立即着手编写编译器了，而在稍后再阅读理论部分．

最好是顺序阅读本书，但没有必要阅读每一章中的每一小节——如果读者无意阅读某些细节部分，则可以直接跳过描述程序内部结构的章节．当然，我们郑重建议阅读源代码，以及附录E和附录D，然后再阅读描述源代码功能的正文．

附录A汇集了程序内部使用的各种支持函数，这些函数涉及到集合处理和窗口管理等方面．这样组织的目的是为了让读者的思路不致被各种函数声明所打断．附录A中的函数贯穿于全书，所以在阅读源代码之前最好阅读并熟悉它们一至少应熟悉其调用形式．

本书的主要组织特点是理论部分与实现部分是分开来的．所有的理论部分都被组织在每章的前面，以便读者在对实现部分不感兴趣时可以轻易地跨越之．但有些理论概念与实际代码融合在一起，因为解释源代码可以阐明一些理论概念．如前所述，在这种情况下，理论部分被分解成若干个小节．一般而言，在讨论完理论之后，紧接着就解释实现此理论的源代码．这样，只要读者愿意，就可以毫不费力地跳过与实现相关的内容．

这里没有提及与编译器有关的内容．除了第七章的概述之外，本书没有谈及代码优化问题——第七章讨论了各种优化方法会怎样修改代码，但除了一些简单的例子之外，它没有讨论优化机制本身．同样，本书只讨论了最常用的语法分析策略（例如，没有提到算符优先语法分析)．

源代码及其兼容性

本书中的所有代码均是用ANSI C写成的．大多数情况下，在编译之前适当修改一些#define语句即可将代码的MS-DOS版本转换成UNIX版本．源代码盘上含有一个小型的UNIX予处理器，后者负责处理转换中的其它细节性问题；而且，予处理器的输出结果可以直接为UNIX CC所接受．本书假定UNIX编译器支持在大多数UNIX系统中实现的ANSI功能（例如结构赋值等)．

如果希望直接应用这些代码（不经过UNIX予处理过程)，则必须有支持函数原型的与ANSI兼容的编译器．此外:

- <stdarg.h>用于可变参数表．
- 允许在予处理器伪指令中的#左右附加空格．
- 可以为结构赋值．
- 支持unsigned char类型．
- 可使用函数原型．
- isdigit() 等无副作用．
- 可使用串连接操作．
- 允许标识符名长达16个字符．

与严格的ANSI唯一不同的是名长度．从理论上讲，ANSI允许的外部符号名最多可有6个字符，但此处假定它支持16个字符长的符号名．本书也采用以前的Kernighan & Ritchie风格的子程序声明形式:

```
lorenzo (arg1,arg2)
char *arg1;
```

```
double * arg2;
```

而不是:

```
lorenzo(char * arg1,double * arg2)
```

之所以如此，是因为大多数编译器仍不支持新的语法，而旧语法也仍然有效.

本书有意识地避免使用 Microsoft 编译器的特殊功能：没有使用 far 关键字和 huge 指针，这样，代码模式更为紧凑；但使用这些关键字可以提高代码的运行效率. 同样地，本书没有使用 register 变量，因为 Microsoft 编译器能更有效地给寄存器赋值.

遗憾的是，8086 毕竟是一种主要的微处理器，源代码中也有部分代码与 8086 芯片相关. 这部分代码被分离出来，并组织于宏定义中. 这样，修改这些宏并重新编译源代码，就可以方便地使之适用于不同的环境. 书中的确讨论了与 8086 相关的一些细节问题，如果读者对此不感兴趣，尽可跳过这部分内容.

第一章　基本概念

本章引入编译器设计的基本概念，讨论编译器的内部组织形式，介绍正则文法和语法分析树；同时针对递归表达式构造一个小型的编译器．但在阅读本章的内容之前，值得提醒读者注意的是，本章及全书都是按一定的顺序进行组织的．一旦读者熟悉了通用编译器的结构．那么，编译器就只不过是一种并不特别难理解的程序了．主要问题不是编译器的每一部分都难以理解；而是其组成成份太多—而且，只有在弄懂了绝大多数成份之后才能体会到每种成份的含义．所以，建议读者不必弄清各种成份之间如何连接，而直接顺序阅读．可以肯定，读者终将会到达一个"突破点"，此时，对整个系统的理解就豁然开朗了．

1.1　编译器的组成成份

编译器是一种复杂的程序，所以，它通常被分解成几个不同的部分，称作"遍"，各部分之间通过临时文件相关联．但是，遍本身只是编译过程的一部分．除了编译之外，由源代码文件生成可执行文件的过程还包括许多步骤（予处理，汇编，连接等）．事实上，某些操作系统（如 Microsoft OS／2）在真正装入程序运行时才生成最终的可执行文件．这种情形在象 UNIX cc 或 Microsoft C 的 cl 等驱动程序中表现得尤其明显，这两种驱动程序对用户封闭了其编译的大部分过程．这些驱动程序作为执行文件，控制着组成编译器的各种不同成份，而各种成份的使用，对用户是透明的．为了本书的目的起见，我们将编译器定义成能将一种语言编译成另一种语言的一个或一组程序—当然，是将高级计算机语言的源代码编译成汇编语言．汇编器、连接器等都不作为编译器的组成部分．

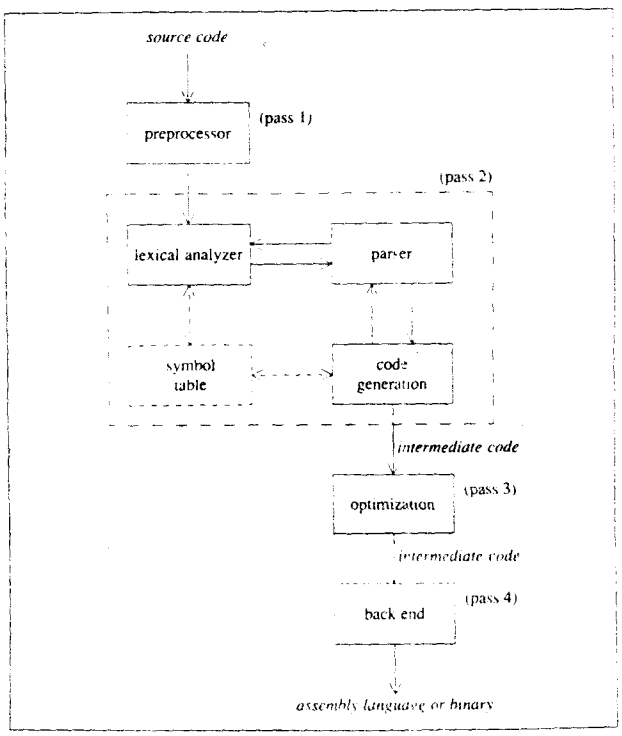


图 1.1　典型的四遍扫描编译器的结构

图 1.1 描述了一种典型的四遍扫描的编译器结构．第一遍是予处理器，一般情况下，予处理器完成宏替换，从源代码中滤掉注释，处理各种其他任务，从而减轻编译器的负担．第二遍扫描是编译器的核心．它由词

法分析器、语法分析器和代码生成器组成，并将源代码翻译成类似于汇编语言的中间语言．第三遍是优化器，它可以提高已生成的中间代码的质量．第四遍是善后处理，将经过优化的代码翻译成汇编语言或某种形式的可执行二进制码．当然，这种结构还有许多变种．许多编译器没有予处理器；还有一些编译器在第二遍扫描中产生汇编语言代码，并且直接优化汇编代码，不需要经过第四遍扫描；也有一些编译器直接产生二进制指令，不经过象汇编语言一样的中间 ASCII 语言．

本书集中描述了上述模型中的第二遍扫描．也介绍了一些比高级语言中的扫描更为复杂的操作，它们共享着数据结构(例如符号表)．

1.1.1 词法分析器

编译过程中的每一阶段都完成一件独立的任务，通常一遍扫描由许多阶段组成．编译器的词法分析阶段(常称为扫描器)可将输入转换成编译器的其它部分更便于使用的形式．词法分析器把输入流看作是称为标记(token)的基本语言元素的集合．亦即，一个标记指一个独立不可分的词法单元．在 C 语言中，关键字 while 或 for 等是标记(不能写作 wh ile)，符号 > 、 > = 、 > > 和 > > = 等是标记，名字和数值是标记，如此类推．组成标记的原串被称为标记串(lexeme)．注意，在标记串与标记之间没有一一对应关系．例如，与 name 标记或 number 标记相关的标记串可能有许多任务；而 while 标记通常只匹配单个标记串．对于叠用的标记(如 > 、 > = 、 > > 及 > > = 等)，这种情形更为复杂．通常情况下，词法分析器识别出与最长标记串相匹配的标记—许多语言本身的含义亦是如此．假定输入为 > >，那么将识别出一个 shift(移位)标记，而不是两个 greater_than(大于)标记．

词法分析器将标记串转换成标记．通常情况下，标记的内部表达形式是独立的整数或另一种可数类型．往往同时使用这两种形式—即标记本身及标记串，示例中的标记串用以区别各种不同的 name 或 number 标记．

影响整个编译器结构的早期设计策略之一是对于标记集的选择．可以为每个输入符号创建一个标记，也可以将许多输入符号合并成同一个标记一例如，可将 > 、 > = 、 > > 及 > > = 看作是四个标记，也可以看作是单一的 Comparison-operator(比较算符)标记—其标记串用以消除标记的二义性．前者有时可使得代码的生成过程更为简单．但是，如果标记太多，语法分析器就会比实际需要的规模更大，也更难以书写．对于标记的选择，并没有一种快速而准确的规则；但在阅读了本书之后，读者可能会理清设计思路，作出明智的选择．通常可将具有同一优先级和相关性的算符组织在一起，将类型声明关键字(如 int 和 char)组织在一起，如此类推．

词法分析器通常是一个自包含单元，通过极少数(通常是一或两个)子程序和全程变量与编译器的其它部分进行接口．每当需要一个新的标记时，语法分析器就调用一次词法分析器，而后者负责返回标记及与其相关的标记串．由于对语法分析器而言，实际的输入机制是隐含的，所以，可以在不影响编译器中其它部分的情况下修改或更换词法分析器．

1.1.2 语法分析器

编译器是一种语言翻译器—它将高级语言如 C 等翻译成低级语言，如 8086 汇编语言．因而，该过程的许多理论方面都借鉴了语言学中的作法．其中之一就是语法分析思想．分析一个英文句子的语法就是将其分解成各种语构成份，以便从文法上对其加以分析．例如，句子

Jane sees Spot run

可分解成主语(Jane)和谓语(Sees Spot run)．然后，谓语又分解成动词(sees)、直接宾语(Spot)和影响直接宾语的辅助语(run)．图 1.2 描绘了该句子，它用一种方便的语句图形式进行表示，读者在英语课中可能学过．

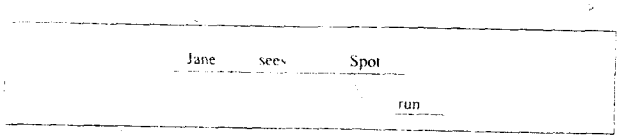


图 1.2. 句子 Jane Sees Spot Run 的语句图

编译器在语法分析阶段完成类似的过程(将语子分解成其语构成份),但它通常是以树形结构而不是以语句图的形式来表示被分析的语句。(此时,语句指整个程序。)

该语句图本身描绘了语句各部分之间的联系,所以,通常称这类图形为语法图(如果为树形结构,则称为语法树)。但是可以扩充这种语法树,使之反映出文法结构及语法结构。后一种表示方法称为语法分析树。图 1.3 描绘了前面句子的语法分析树。图 1.4 描绘了表达式 A×B+C×B 的语法分析树。这里采用了树型结构,主要是因为它易于用计算机程序表达;而语句图则不然。

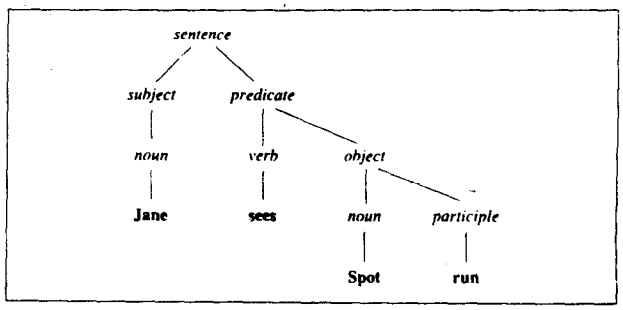


图 1.3.语句 Jane Sees Spot Run 的语法分析树

顺便指出,语句本身也是一个术语,其含义与其在英文中的含义相同。它是标记的集合,遵从一定的文法结构。就编译器而言,语句通常指整个计算机程序。在象 Pascal 等语言中,这种对应关系非常明显,Pascal 语言本身反映出英语标点及文法。正如英语句子一样,Pascal 程序也用句号作为结束标记。类似地,在英语句子中,逗号用以分隔两个独立的语句,而在 Pascal 中,逗号用以分融两个完整的过程。

总之,语法分析器指一组子程序,它将标记流转换成一颗语法分析树,而语法分析树是对被分析语句的结构化表示。从另一个角度来看,语法分析树用一种分层形式表示语句,从对语句的常规描术(树根)一直深入到树叶处的特定的被分析语句(实际的标记)。一些编译器创建物理上的语法分析树,它由结构、指针等组成,但大多数编译器隐式地描述语法分析树。其它语法分析方法只对树中的位置进行跟踪,而并不创建物理上的分析树(以下将紧接着讨论这种方法的工作过程)。语法分析树本身是一非常有用的概念,借助它有利于澄清语法分析的操作过程。

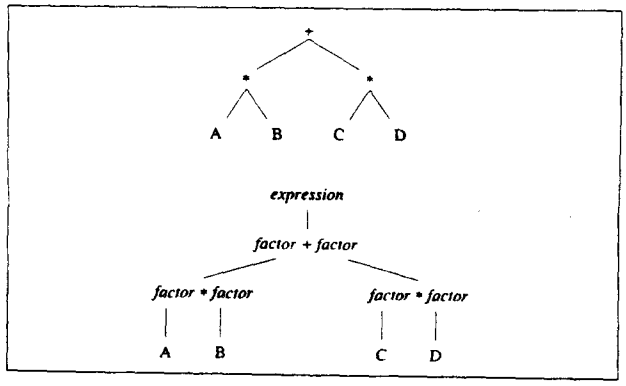


图 1.4. A×B+C×D 的语法及语法分析树

1.1.3 目标代码生成器

编译器的最后一部分是代码生成器。将代码生成阶段从语法分析过程中独立出来有时会引起误解,因为大多数编译器在进行语法分析时就开始产生代码。亦即,代码可以由分析输入流的同一子程序所产生。但是,语法分析器可以为整个输入文件创建一颗语法分析树,然后,用独立的代码生成器将此语法分析树转换成代码,有一些编译器采用的就是这种方法。第三种可能的选择是,用语法分析器为输入流创建一种中间语言表

示形式，再在优化扫描过程中重新创建一颗语法树．相对于线性指令流而言，某些优化过程更易于施用于语法树．最后，代码生成扫描过程将经过优化的语法树转换成代码．

尽管编译器可以直接生成目标码，但它常将代码生成延迟到由另一个程序来做．不是直接产生机器语言指令，而是创建一种中间语言程序，再由编译器的第四遍扫描过程翻译成机器语言．可以将中间语言看成是一种为完成特定任务(如优化)而设计的超级汇编语言．正如读者所能期望的一样，使用中间语言有许多形式，分别用于不同的应用中．

书写采用中间语言的编译器有利也有弊．其主要缺点是降低了速度．直接根据标记产生二进制目标代码的语法分析过程是很快的，而处理中间代码这一额外步骤常常会使编译时间增倍．但使用中间语言所带来的好处通常会抵消速度上的损失．这表现在其优化和灵活性上．一些优化措施，例如简单常量的求值—在编译而不是在运行时对常量表达式进行求值—可以在语法分析阶段完成．但对于大多数优化过程，既便语法分析器可以完成，也显得相当困难．所以，带有优化程序的编译器中，语法分析器只输入中间语言，以便于第二次优化扫描时对其进行处理．

中间语言也为用户增添了灵活性．通过提供独立的善后程序，可以使用词法／语法分析器为各种不同的机型产生目标代码，善后程序的作用是将通用的中间语言转换成针对特定机型的汇编语言．同样地，可以书写各种高级语言和语法分析器，但输入同一种中间语言代码．这样，各种语言的编译器可以共享同一种优化器和善后程序．

中间语言的最后一个用途是用于增量式编译器或解释程序中．这些程序直接执行中间代码，并不将它先译为二进制码，从而省略了用于汇编和链接实际程序的时间，缩短了开发周期．解释程序也可以改进用户的调试环境，因为它可以在运行时检查诸如数组越界等错误．

第六章中介绍的编译器使用了中间语言作为输出代码．该章还深入讨论了该语言的特点，但这里有必要指出几点，因为本书不规范地采用该语言作为代码生成样例．简单地讲，该中间语言是C语言的子集，其中大多数指令直接翻译成一种典型机型上的少数汇编语言指令(通常是一或两条)．例如，表达式 x=a+b×c+d 被翻译成如下形式：

```
t0 = -a;
t1 = -b;
t1 *= c;
t0 += t1;
t0 += -d;
```

上述代码中的 t0 和 t1 是编译器申请用来保存表达式部分求值结果的临时变量．通常称这些临时变量为匿名中间单元(常简称为中间单元)，以下将深入讨论这一点．编译器将下划线添加到声明过的变量的变量名前，以便将它们与编译器本身产生的变量如 t0、t1 区分开来(变量名本身没有下划线)．

由于中间语言与C语言如此相似，所以此处不经过正式的语言定义就加以使用．但请记住，中间语言不是C语言(编译器不会将高效的C程序翻译成低劣的C程序)—它只是一种汇编语言，且其语法与C语言语法类似．

1.2 计算机语言的表示

某些设计上的抽象化有利于构造代码，在这一点上，编译器与其它程序很相似．流程图、Warnier-Orr 图和结构图就是抽象设计的例子．在编译器应用上，最好的抽象莫过于用一种反映编译器本身结构的方式去描述被编译的语言．

1.2.1 文法和语法分析树

以一种形式化的方式描述程序设计语言的最通用的方法就是从语言学中借鉴而来的．该方法即正则文法，首先由 M．I．T Noam Chomsky 提出来，并由 J.M.Backus 用于计算机程序并实现第一个 FORTRAN 编译器．

正则文法常用修改的 Backus-Naur 范式(亦称 Backus-Normal 范式，BKS 范式)来表示，BKS 范式简称为 BNF．严格的 BNF 表示方式以一序列标记开头，这些标记称为终结符；还有一序列定义，称为非终结符．这些定义构成了某种体系，其中，语言中的每种合法结构均可予以表达．算子::= 的含义是"被定义成"或"转

向″．例如，下列 BNF 规则可以作为英语句子文法的开头：

Sentence ::= subject predicate

sentence 被定义成 Subject 后跟 predicate. 也可以理解成″语句即是主语后跟谓语．″每类规则都被称为一个产生式(production)．::= 左边的非终结符称为左部符号(left－hand side)，::= 右边的符号称为产生式的右部符号(right－hand side)．在本书所用的文法中，产生式的左部通常由单个的非终结符组成，且在右部使用过的任何非终结符也必须在左部出现．不在左部出现的任何符号，例如输入语言中的标记等，都是终结符．

实际的文法总是有不断深入的定义，直到定义了所有终结符为止．例如，上述文法可以继续书写为下：

subject ::= noun

noun ::= JANE

其中，JANE 是终结符(与输入串″Jane″相匹配的标记)．

通常对严格的 BNF 加以修改，以使该文法更容易书写，本书将采用改进的 BNF．修改之一是引入了 OR 算子，用竖线(|) 表示．例如，

noun ::= JANE

noun ::= DICK

noun ::= SPOT

可以表示如下：

noun ::= DICK|JANE|SPOT

类似地，通常用→代替::= ：

noun→DICK|JUNE

本书的大多数场合都使用→．

另外还有一个很重要的概念．文法必须尽可能灵活，获得灵活性的途径之一就是应用某些可选规则．如规则

article → THE

表明 THE 是 article；而读者可以使用以下产生式：

object → article noun

在英语中，object 被表示成 article 后跟 noun．上面的规则要求组成 object 的所有 noun 的前面都必须有 article．但如果希望 articl 是可选的，这时又该如何处理呢？要完成这一点，可以将 article 定义成名词″the″或空串．下列规则表达了这一含义：

article →THE | ε

ε(念作″epsilon″) 表示空串．如果标记 THE 出现于输入串中，则规则

article →THE

得到应用．否则，article 匹配空串，规则

article → ε

得到应用．所以，语法分析器通过检查下一个输入符号，从而确定使用两个产生式中的哪一个．

识别有限的英语语句集的文法如下：

sentence －> subjext predicate
subject －> noun
predicate －> verb object
object －> noun opt participle
opt participle －> participle | ε
noun －> SPOT | JANE | DICK
participle －> RUN
verb －> SEES

使用该文法，通过一序列置换可以识别出输入的英语句子．过程如下：

(1) 以文法的顶部符号，即目标符号作为开始．

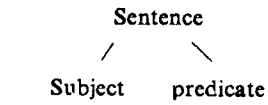
(2) 用其右部之一代替该符号．

(3) 继续置换非终结符，通常是用产生式右部置换最左边的非终结符，直到没有非终结符可以置换为止．

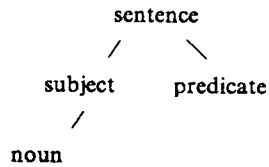
例如，该文法可用来识别句子″Jane sees Spot run″，其识别过程如下：

sentence	施用sentence→subject predicte得到:
subject predicate	施用 subject→noun 得到:
noun predicate	施用 noun→JANE 得到:
JANE predicate	施用 predicate→verb object 得到:
JANE verb object	施用 predicate→verb object 得到:
JANE verb object	施用 verb→SEES 得到:
JANE SEES object	施用 object→noun op—participle 得到:
JANE SEES noun opt—participle	施用 noun→spot 得到:
JANE SEES SPOT opt—participle	施用 opt—participle→partiple得到:
JANE SEES SPOT participle	施用 participle→RUN 得到:
JANESEES SPOT RUN	完毕—没有可置换的非终结符

这些置换可用于建造语法分析树．例如，用 subject predicate 置换 sedten 的过程可以用树型结构表示如下:



用 noun 置换 subject，即第二步则将上述树形结构修改成:



如此类推．图 1.5 绘出了整棵语法分析树的生长过程．

从语法分析树中不难发现术语终结符和非终结符的出处．终结符通常指树叶(位于枝的末端)，而非终结符通常指中间节点．

1.2.2 表达式文法

表 1.1 描绘了识别一或多条语句的文法，每条语句是一个算术表达式后跟一个分号．Statements 由一序列以分号作为界符的表达式组成，而表达式又由以星号(乘法运算)或加号(加法运算)分开的数串组成．

注意，该文法是递归的．例如，在第 2 个产生式中，其左部和右部均出现了 statements．在第 8 个产生式中也存在着第三类递归．由于第 8 个产生式中包含有 expression，而得到 expression 的唯一方法就是通过产生式 3，其中 expression 出现于其左部．如果在文法中作一序列替换，那么后一种递规就更为清晰了．可以用产生式 6 的右步替换产生式 4 中的 term，即

expression → factor

再用产生式 8 的右部替换 factor:

expression → (expression)

由于文法本身是递归的，所以可想而知，在对文法进行语法分时也可以采用递归—以下将讨论如何作到这一点．从结构化的角度来看，递归也是很重要的—正是由于递归，才使得有限的文法可以识别无限数目的语句．

上述文法的功能很强，而且也很直观—其结构直接反映了表达式的结合方式．但其中存在着一个重大问题．许多产生式右部的最左符号与出现在其左部的符号相同．例如在产生式 3 中，expression 既出现在左部，同时又出现在右部的最左端．这种情形称为左递归，而某些语法分析器(例如后面将要讨论的递归下降语法分析器)却不能处理左递归的产生式．它们只是无限循环，重复地用整个右部去替换右部的最左符号．

为了理解这一问题，可以考虑语法分析器在需要替换一个有多个右部的非终结符时如何施用一条产生式．产生式 7 和 8 就是一个简单明了的例子．在需要扩展 factor 时，语法分析器可以查看下一个输入符号，从而决定选择哪条产生式．如果下一个符号是 number(数)，那么语法分析器施用规则 7，并用 number 替换 factor．如果下一个输入符号是一个开括号，那么语法分析器将选择产生式 8．而对于产生式 5 和产生式 6 的