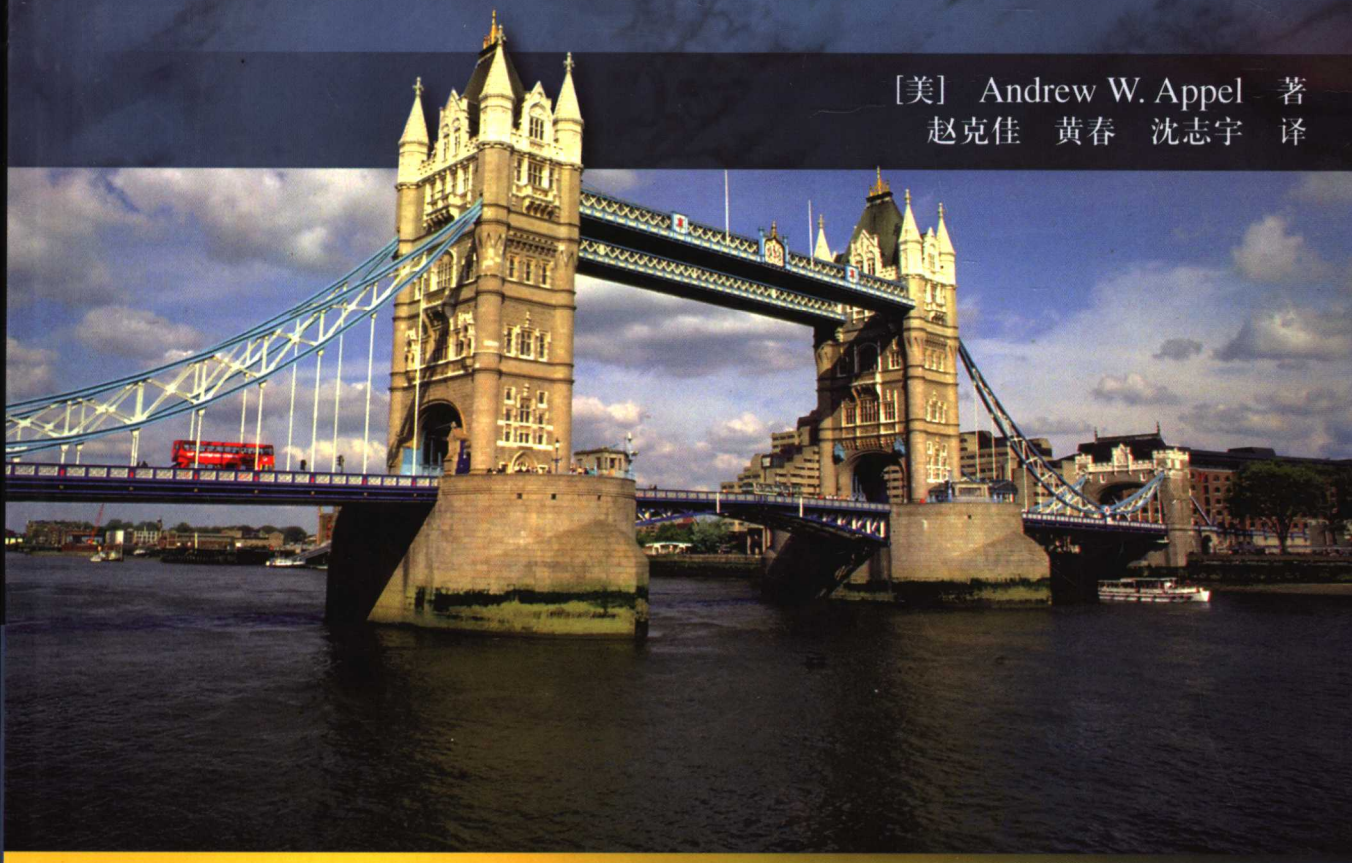
TURING 图灵计算机科学丛书

现代编译原理
C语言描述

Modern Compiler Implementation in C



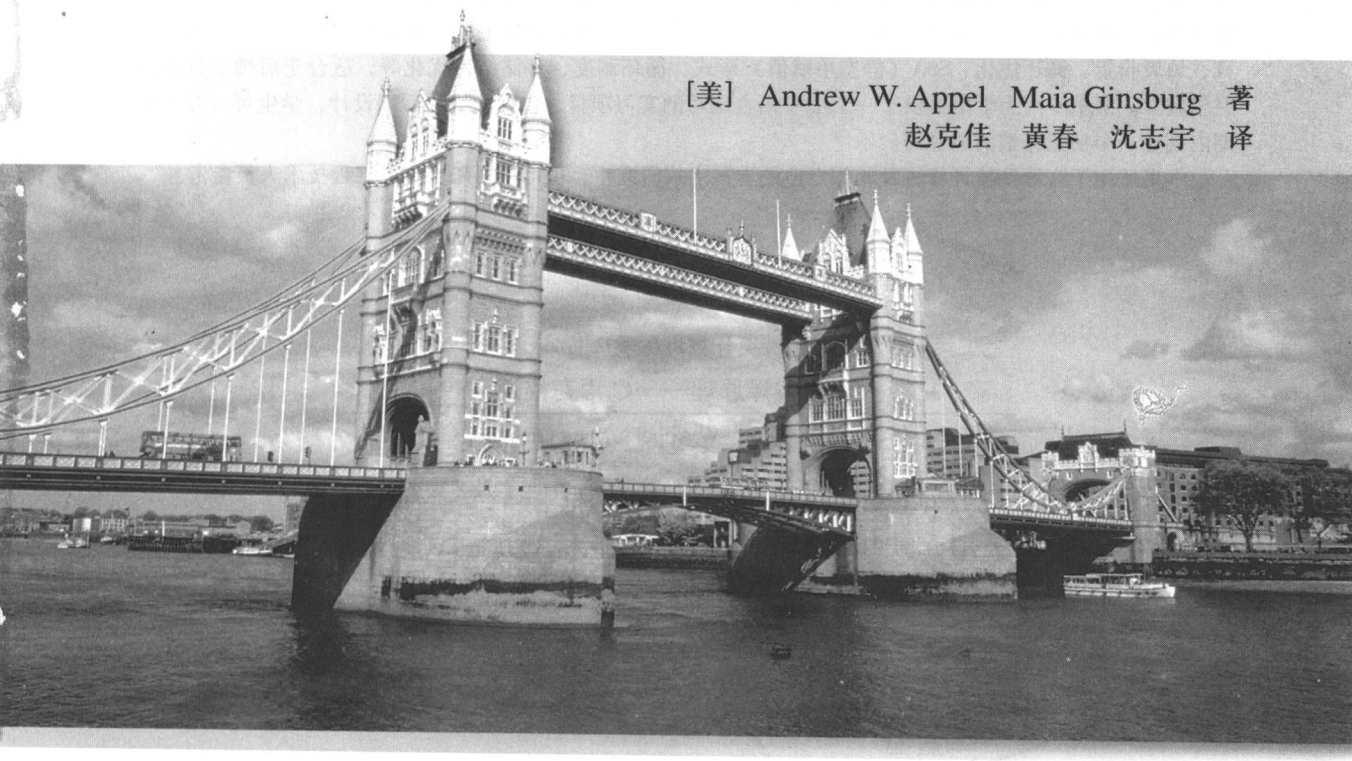
[美] Andrew W. Appel 著
赵克佳 黄春 沈志宇 译

人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵计算机科学丛书

现代编译原理
C语言描述

Modern Compiler Implementation in C



[美] Andrew W. Appel Maia Ginsburg 著
赵克佳 黄春 沈志宇 译

人民邮电出版社
POSTS & TELECOM PRESS

图书在版编目（CIP）数据

现代编译原理：C 语言描述 /（美）安佩尔，（美）金斯伯格著；赵克佳，黄春，沈志宇译. —北京：人民邮电出版社，2006.4
（图灵计算机科学丛书）

ISBN 7-115-14552-0

Ⅰ. 现… Ⅱ. ①安…②金…③赵…④黄…⑤沈… Ⅲ. ①编译程序—程序设计②C 语言—程序设计 Ⅳ. TP314

中国版本图书馆 CIP 数据核字（2006）第 013518 号

内 容 提 要

本书全面讲述了现代编译器的各个组成部分，包括词法分析、语法分析、抽象语法、语义检查、中间代码表示、指令选择、数据流分析、寄存器分配以及运行时系统等。全书分成两部分，第一部分是编译的基础知识，适用于第一门编译原理课程（一个学期）；第二部分是高级主题，包括面向对象语言和函数语言、垃圾收集、循环优化、SSA（静态单赋值）形式、循环调度、存储结构优化等，适合于后续课程或研究生教学。书中专门为学生提供了一个用 C 语言编写的实习项目，包括前端和后端设计，学生可以在一学期内创建一个功能完整的编译器。

本书适用于高等院校计算机及相关专业的本科生或研究生，也可供科研人员或工程技术人员参考。

图灵计算机科学丛书

现代编译原理——C 语言描述

◆ 著 [美]Andrew W.Appel Maia Ginsburg
译 赵克佳 黄 春 沈志宇
责任编辑 杨海玲

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
邮编 100061 电子函件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京顺义振华印刷厂印刷
新华书店总店北京发行所经销

◆ 开本：787×1092 1/16
印张：25
字数：665 千字 2006 年 4 月第 1 版
印数：1－5 000 册 2006 年 4 月北京第 1 次印刷
著作权合同登记号 图字：01-2005-4375 号

ISBN 7-115-14552-0/TP・5271

定价：45.00 元

读者服务热线：(010)88593802 印装质量热线：(010)67129223

前　言

近十余年来，编译器的构建方法出现了一些新的变化。一些新的程序设计语言已经得到应用，例如，具有动态方法的面向对象语言、具有嵌套作用域和一阶函数闭包（first-class function closure）的函数式语言等，这些语言中有许多都需要垃圾收集技术的支持。另一方面，新的计算机都有较大的寄存器集合，且存储器访问成为了影响性能的主要因素，这类机器在具有指令调度能力，并能对指令和数据高速缓存（cache）进行局部性优化的编译器辅助下，常常能运行得更快。

本书可作为一到两个学期编译课程的教材。学生将看到一个编译器不同部分中隐含的理论，学习到将这些理论付诸实现时使用的程序设计技术和以模块化方式实现该编译器时使用的接口。为了清晰具体地给出这些接口和程序设计的例子，我使用 C 语言来编写它们。本书还有使用Java和 ML 语言的另外两个版本。

实现项目。我在书中概述了一个“学生项目编译器”，它相当简单，而且其安排方式也便于说明现在常用的一些重要技术。这些技术包括避免语法和语义相互纠缠的抽象语法树、独立于寄存器分配的指令选择、能使编译器前期阶段有更多灵活性的复写传播，以及防止从属于特定目标机的方法。与其他许多教材中的“学生编译器”不同，本书中采用的编译器有一个简单而完整的后端，它允许在指令选择之后进行寄存器分配。

本书第一部分中，每一章都有一个与编译器的一个模块对应的程序设计习题。在 `http://www.cs.princeton.edu/~appel/modern/c` 中可找到对这些习题有帮助的一些软件。

习题。每一章都有一些书面习题；标有一个星号的习题有点挑战性，标有两个星号的习题较难但仍可解决，偶尔出现的、标有三个星号的习题是一些尚未找到解决方法的问题。

授课顺序。图 0-1 展示了各章相互之间的依赖关系。

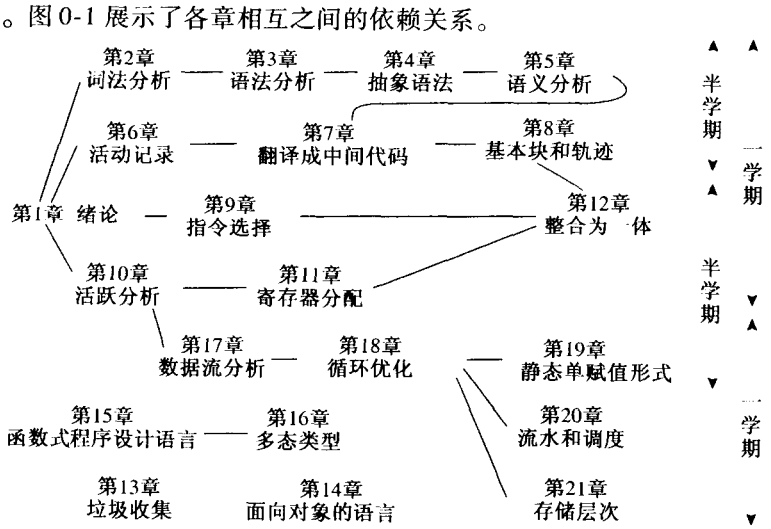


图 0-1　内容结构图

- 一学期的课程可包含第一部分的所有章节（第 1 ~ 12 章），同时让学生实现项目编译器（多半按项目组的方式进行）。另外，授课内容中还可以包含从第二部分中选择的一些主题。
- 高级课程或研究生课程可包含第二部分的内容，以及另外一些来自最新文献的主题。第二部分中有许多章节与第一部分无关，因此，对于那些在初始课程中使用不同教材的学生而言，仍然可以给他们讲授高级课程。
- 若按两个半个学期来安排教学，则前半学期可包含第 1 ~ 8 章，后半学期包括第 9 ~ 12 章和第二部分的某些章。

致谢。对于本书，许多人给我提出了富有建设性的意见，或在其他方面给我提供了帮助。我要感谢这些人，他们是 Leonor Abraido-Fandino、Scott Ananian、Stephen Bailey、Max Hailperin、David Hanson、Jeffrey Hsu、David MacQueen、Torben Mogensen、Doug Morgan、Robert Netzer、Elma Lee Noah、Mikael Petterson、Todd Proebsting、Anne Rogers、Barbara Ryder、Amr Sabry、Mooly Sagiv、Zhong Shao、Mary Lou Soffa、Andrew Tolmach、Kwangkeun Yi 和 Kenneth Zadeck。

目　录

第一部分　编译基本原理

第二部分　高级主题

第一部分　编译基本原理

第1章　绪　　论

编译器（compiler）：原指一种将各个子程序装配组合到一起的程序［连接-装配器］。当1954年出现了（确切地说是误用了）复合术语“代数编译器”（algebraic compiler）之后，这个术语的意思变成了现在的含义。

Bauer 和 Eickel［1975］

本书讲述将程序设计语言转换成可执行代码时使用的技术、数据结构和算法。现代编译器常常由多个阶段组成，每一阶段处理不同的抽象“语言”。本书的章节按照编译器的结构来组织，每一章循序渐进地论及编译器的一个阶段。

为了阐明编译真实的程序设计语言时遇到的问题，本书以 Tiger 语言为例来说明如何编译一种语言。Tiger 语言是一种类 Algol 的语言，它有嵌套的作用域和在堆中分配存储空间的记录，它虽简单却并不平凡。每一章的程序设计练习都要求实现相应的编译阶段；如果学生实现了本书第一部分讲述的所有阶段，他便能够得到一个可以运行的编译器。不难将 Tiger 修改成函数式的或面向对象的（或同时满足两者的）语言，第二部分中的习题说明了如何进行这种修改。第二部分的其他章节讨论了有关程序优化的高级技术。附录 A 描述了 Tiger 语言。

编译器各模块之间的接口几乎和模块内部的算法同等重要。为了具体描述这些接口，较好的做法是用真正的程序设计语言来编写它们，本书使用的是 C 语言。 3

1.1　模块与接口

对于任何大型软件系统，如果设计者注意到了该系统的基本抽象和接口，对这个系统的理解和实现就要容易得多。图 1-1 展示了一个典型的编译器的各个阶段，每个阶段由一至多个软件模块来实现。

将编译器分解成这样的多个阶段是为了能够重用它的各种构件。例如，当要改变此编译器所生成的机器语言的目标机时，只要改变栈帧布局（Frame Layout）模块和指令选择（Instruction Selection）模块就够了。当要改变被编译的源语言时，则至多只需改变翻译（Translate）模块之前的模块就可以了，该编译器也可以在抽象语法（Abstract Syntax）接口处与面向语言的语法编辑器相连。

每个学生都不应错过反复多次“思考-实现-重新设计”，从而达到正确的抽象这样一种学习经历。但是，想要学生在一个学期内实现一个编译器是不现实的。因此，我在书中给出了一个项目框架，其中的模块和接口都经过深思熟虑，而且尽可能地使之既精巧又通用。 4

抽象语法（Abstract Syntax）、IR 树（IR Tree）和汇编（Assem）之类的接口是数据结构的形式，例如语法分析动作阶段建立抽象语法数据结构，并将它传递给语义分析阶段。另一些接

口是抽象数据类型：翻译接口是一组可由语义分析阶段调用的函数；单词符号（Token）接口是函数形式，分析器通过调用它而得到输入程序中的下一个单词符号。

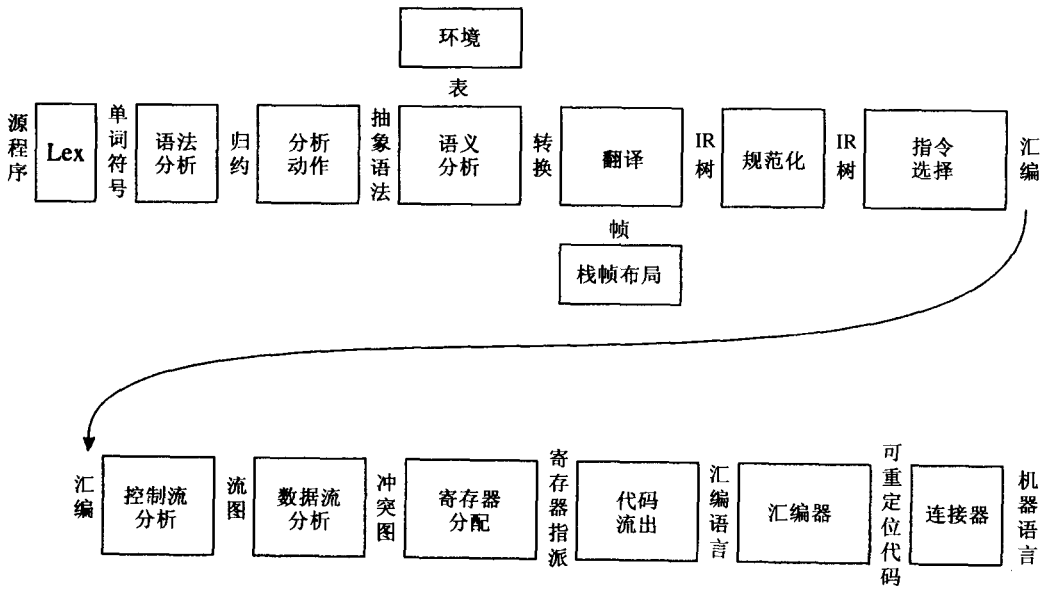


图 1-1　编译器的各个阶段以及它们之间的接口

各个阶段的描述

第一部分的每一章各描述编译器的一个阶段，具体如表 1-1 所示。

表 1-1　编译器的各阶段

章号	阶段	描　述
2	词法分析	将源文件分解为一个个独立的单词符号
3	语法分析	分析程序的短语结构
4	语义动作	建立每个短语对应的抽象语法树
5	语义分析	确定每个短语的含义，建立变量和其声明的关联，检查表达式的类型，翻译每个短语
6	栈帧布局	按机器要求的方式将变量、函数参数等分配于活跃记录（即栈帧）内
7	翻译	生成中间表示树（IR 树），这是一种与任何特定程序设计语言和目标机体系结构无关的表示
8	规范化	提取表达式中的副作用，整理条件分支，以方便下一阶段的处理
9	指令选择	将 IR 树结点组合成与目标机指令相对应的块
10	控制流分析	分析指令的顺序并建立控制流图，此图表示程序执行时可能流经的所有控制流
10	数据流分析	收集程序变量的数据流信息，例如，活跃分析（liveness analysis）计算每一个变量仍需使用其值的地点（即它的活跃点）
11	寄存器分配	为程序中的每一个变量和临时数据选择一个寄存器，不在同一点活跃的两个变量可以共享同一个寄存器
12	代码流出	用机器寄存器替代每一条机器指令中出现的临时变量名

这种模块化设计是很多真实编译器的典型设计。但是，也有一些编译器把语法分析、语义分析、翻译和规范化合并成一个阶段，还有一些编译器将指令选择安排在更后一些的位置，并且将它与代码流出合并在一起。简单的编译器通常没有专门的控制流分析、数据流分析和寄存器分配。

我在设计本书的编译器时尽可能地进行了简化，但并不意味着它是一个简单的编译器。具体而言，虽然为简化设计而去掉了一些细枝末节，但该编译器的结构仍然可以允许增加更多的优化或语义而不会违背现存的接口。

1.2　工具和软件

现代编译器中使用的两种最有用的抽象是*上下文无关文法*（context-free grammar）和*正则表达式*（regular expression）。上下文无关文法用于语法分析，正则表达式用于词法分析。为了更好地利用这两种抽象，较好的做法是借助一些专门的工具，例如 Yacc（它将文法转换成语法分析器）和 Lex（它将一个说明性的规范转换成一个词法分析器）。

本书的程序设计项目可借助 Lex（或更为现代的 Flex）和 Yacc（或更为现代的 Bison），用任何 ANSI 标准 C 编译器来编译，这些工具中有些可免费从因特网上得到，更多的信息可参看网页：`http://www.cs.princeton.edu/~appel/modern/c/`。

5

Tiger 编译器中某些模块的源代码、某些程序设计习题的框架源代码和支持代码、Tiger 程序的例子以及其他一些有用的文件都可以从该网址中找到。本书的程序设计习题中，当提及到特定子目录或文件所在的某个目录时，指的是目录 `$TIGER/`。

6

1.3　树语言的数据结构

编译器中使用的许多重要数据结构都是被编译程序的*中间表示*。这些表示常常采用树的形式，树的结点有若干种类型，每一种类型都有一些不同的属性。这种树可以作为图 1-1 所示的许多阶段的接口。

树表示可以用文法来描述，就像程序设计语言一样。为了介绍有关概念，我将给出一种简单的程序设计语言，该语言有语句和表达式，但是没有循环或 if 语句［这种语言称为*直线式程序*（straight-line program）语言］。

该语言的语法给出在文法 1-1 中。

文法 1-1　直线式程序设计语言

Stm → *Stm* ; *Stm*	(CompoundStm)	*ExpList* → *Exp* , *ExpList*	(PairExpList)
Stm → `id` := *Exp*	(AssignStm)	*ExpList* → *Exp*	(LastExpList)
Stm → `print` (*ExpList*)	(PrintStm)	*Binop* → +	(Plus)
Exp → `id`	(IdExp)	*Binop* → −	(Minus)
Exp → `num`	(NumExp)	*Binop* → ×	(Times)
Exp → *Exp Binop Exp*	(OpExp)	*Binop* → /	(Div)
Exp → (*Stm* , *Exp*)	(EseqExp)		

这个语言的非形式语义如下。每一个 *Stm* 是一个语句，每一个 *Exp* 是一个表达式。$s_1;s_2$ 表示先执行语句 s_1，再执行语句 s_2。$i:=e$ 表示先计算表达式 e 的值，然后把计算结果赋给变量 i。

print(e_1,e_2,…,e_n)表示从左到右输出所有表达式的值，这些值之间用空格分开并以换行符结束。

标识符表达式，例如 i，表示变量 i 的当前内容。数按命名它的整数计值。操作符表达式 e_1 op e_2表示先计算 e_1，再计算 e_2，然后按给定的二元操作符计算表达式结果。表达式序列(s,e)的行为类似于 C 语言中的逗号操作符，在计算表达式 e（并返回其结果）之前先计算语句 s 的副作用。

7

例如，执行下面这段程序：

```
a := 5+3; b := (print(a, a-1), 10*a); print(b)
```

将打印出：

```
8 7
80
```

那么，这段程序在编译器内部是如何表示的呢？一种表示是源代码形式，即程序员所编写的字符，但这种表示不易处理。较为方便的表示是树数据结构，每一条语句（*Stm*）和每一个表达式（*Exp*）都有一个树结点。图 1-2 给出了这个程序的树表示，其中结点都用文法 1-1 中产生式的标识加以标记，并且每个结点的子结点数量与相应文法产生式右边的符号个数相同。

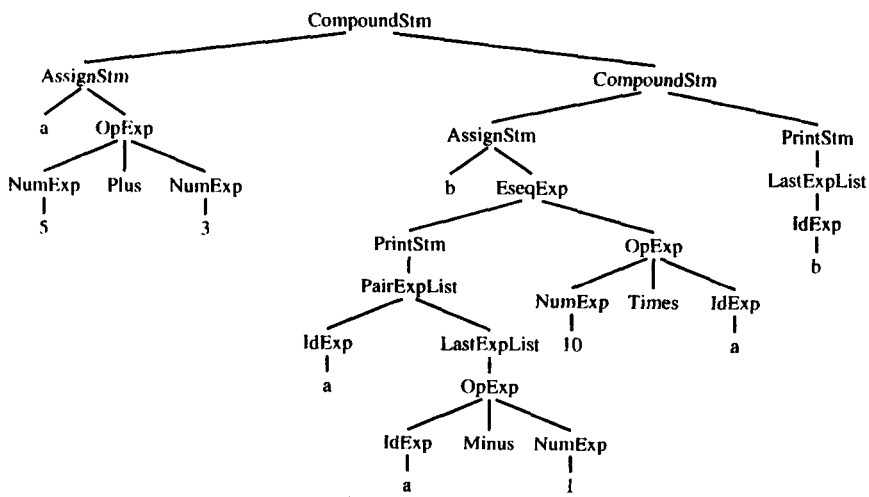


图 1-2　直线式程序的树表示

我们可以将这个文法直接翻译成数据结构定义，如程序 1-1 所示。每个文法符号对应于这些数据结构中的一个 typedef：

8

文法	typedef
Stm	A_stm
Exp	A_exp
ExpList	A_expList
id	string
num	int

程序 1-1 直线式程序的表示

```
typedef char *string;
typedef struct A_stm_ *A_stm;
typedef struct A_exp_ *A_exp;
typedef struct A_expList_ *A_expList;
typedef enum {A_plus,A_minus,A_times,A_div} A_binop;

struct A_stm_ {enum {A_compoundStm, A_assignStm, A_printStm} kind;
               union {struct {A_stm stm1, stm2;} compound;
                      struct {string id; A_exp exp;} assign;
                      struct {A_expList exps;} print;
                     } u;
              };
A_stm A_CompoundStm(A_stm stm1, A_stm stm2);
A_stm A_AssignStm(string id, A_exp exp);
A_stm A_PrintStm(A_expList exps);

struct A_exp_ {enum {A_idExp, A_numExp, A_opExp, A_eseqExp} kind;
               union {string id;
                      int num;
                      struct {A_exp left; A_binop oper; A_exp right;} op;
                      struct {A_stm stm; A_exp exp;} eseq;
                     } u;
              };
A_exp A_IdExp(string id);
A_exp A_NumExp(int num);
A_exp A_OpExp(A_exp left, A_binop oper, A_exp right);
A_exp A_EseqExp(A_stm stm, A_exp exp);

struct A_expList_ {enum {A_pairExpList, A_lastExpList} kind;
                   union {struct {A_exp head; A_expList tail;} pair;
                          A_exp last;
                         } u;
                  };
```

每一项文法规则都有一个构造器（constructor），隶属于规则左部符号的联合（union）。这些构造器的名字在文法 1-1 各项右部的括号内。

每一项文法规则有若干右部成分，这些成分都必须用数据结构来表示。例如，CompoundStm 的右部有两个 Stm；AssignStm 有一个标识符和一个表达式，等等。表示每一个文法符号的结构（struct）都含有一个联合（union）和一个 kind 域，前者用于存放可选的成分值，后者用于指明选用这个联合中的哪一个成员。

对于每一种选择（CompoundStm、AssignStm 等），我们创建一个构造函数（constructor function），它用 malloc 为此数据结构分配空间并对其进行初始化。程序 1-1 只给出了这些函数的原型；A_CompoundStm 可这样定义：

```
A_stm A_CompoundStm(A_stm stm1, A_stm stm2) {
  A_stm s = checked_malloc(sizeof(*s));
  s->kind = A_compoundStm;
  s->u.compound.stm1=stm1; s->u.compound.stm2=stm2;
  return s;
}
```

二元操作符（Binop）的情形要简单些。尽管我们也可以为 Binop 创建一个结构（此结构中的联合的成员分别表示 Plus、Minus、Times、Div），但这样做是多余的，因为这些成员并不存放数

据。我们为它定义的是一个枚举类型 A_binop。

程序设计风格。在用 C 表示树数据结构时，我们遵循以下一些约定：

(1) 树都用文法来描述。

(2) 一棵树用一至多个 typedef 来描述，每个 typedef 对应文法中的一个符号。

(3) 每个 typedef 定义一个指向相应 struct 的指针。这个 struct 的名字以下划线结束，它除了在 typedef 的声明中和该结构定义本身中出现外，决不会在其他地方使用。

(4) 每个 struct 有一个 kind 成员和一个 u 成员。kind 是一个指明不同选择的枚举量，
9 每个枚举值对应一个可选的文法规则；u 是一个联合。

(5) 如果一个规则的右部有多个非平凡的（即携带有值的）符号（例如，规则 Compound-Stm），则它的 union 有一个本身也是结构的成员给出组成它的这些值（例如，A_stm_联合中的成员 compound）。

(6) 如果一个规则的右部只有一个非平凡的符号，则它的 union 有一个就是其值的成员（例如，A_exp 联合中的成员 num）。

(7) 每个类有一个对所有成员进行初始化的构造函数。除了在这些构造函数中，其他地方
10 决不会直接调用 malloc 函数。

(8) 每一个模块（头文件）有一个唯一标识该模块的前缀（例如，程序 1-5 中的 A_）。

(9) 类型定义名（位于前缀之后的）应当用小写字母开头，构造函数名（位于前缀之后）用大写字母开头，联合的成员（它们没有前缀）用小写字母开头。

C 程序的模块化规则。编译器是一个很大的程序，仔细地设计模块和接口能避免混乱。在用 C 编写一个编译器时，我们将使用如下一些规则：

(1) 编译器的每个阶段或者模块都应归入各自的“.c”文件，且该文件有对应的“.h”文件。

(2) 每个模块应有该模块唯一的一个前缀。由此模块导出的所有全局名字（结构和联合的成员名字不是全局的）都应以此前缀打头。这样，文件的阅读者就不必通过到文件之外去查找来确定一个名字的来源。

(3) 所有函数都应有函数原型；如果使用了没有原型的函数，C 编译器将给出警告信息。

(4) 我们将在每一个文件中用#include"util.h"包含 util.h：

```
/* util.h */
#include <assert.h>

typedef char *string;
string String(char *);

typedef char bool;
#define TRUE 1
#define FALSE 0

void *checked_malloc(int);
```

包含 assert.h 是为了鼓励 C 程序员多使用断言。

(5) string 类型表示分配在堆中的字符串，这种字符串在初次创建之后便不会再改变。函数 String 从 C 风格的字符指针来创建一个分配在堆中的字符串 string（类似于标准 C 库函数 strdup）。那些以 string 作为参数的函数都假定这些参数的内容决不会改变。

(6) C 的 malloc 函数在无存储空间可分配时返回 NULL，Tiger 编译器没有复杂的存储管

理来处理这种情形。相反，它从不直接调用malloc，而是只调用我们自己的函数check-ed_malloc，这个函数保证不会返回NULL：

11

```
void *checked_malloc(int len) {
 void *p = malloc(len);
 assert(p);
 return p;
}
```

（7）我们也决不调用free。当然，达到软件成品级质量的编译器必须释放无用数据以避免浪费存储空间。做到这一点最好的方法是使用如第13章介绍的那种自动的垃圾收集器（见第209页保守收集）。没有垃圾收集器的支持，当结构p即将变成不可访问的时，程序员必须特别小心地调用free(p)——既不能太迟，太迟了指针p将丢失；也不可太早，太早了会释放仍然有用的数据（然后被改写）。为了能够使我们的精力更集中于编译技术而不是存储释放技术，我们可以简单地不做任何释放动作。

程序设计：直线式程序解释器

为直线程序设计语言实现一个简单的程序分析器和解释器。这个练习可作为对*环境*（即符号表，它将变量名映射到这些变量相关的信息）、*抽象语法*（表示程序的短语结构的数据结构）、*树数据结构的递归性*（它对于编译器中很多部分都是非常有用的）以及无赋值语句的*函数式风格*程序设计的入门。

这个练习也可以作为C程序设计的热身。熟悉其他语言但对C语言陌生的程序员应该也能完成这个习题，只是需要有关C语言的辅助资料（如教材）的帮助。

需要进行解释的程序已经被分析为抽象语法，这种抽象语法如程序1-5中的数据类型所示。

但是，我们并不希望涉及该语言的具体分析细节，因此，我们利用了相应数据的构造器来编写该程序：

```
A_stm prog =
A_CompoundStm(A_AssignStm("a",
                A_OpExp(A_NumExp(5), A_plus, A_NumExp(3))),
 A_CompoundStm(A_AssignStm("b",
     A_EseqExp(A_PrintStm(A_PairExpList(A_IdExp("a"),
                A_LastExpList(A_OpExp(A_IdExp("a"), A_minus,
                                      A_NumExp(1))))),
            A_OpExp(A_NumExp(10), A_times, A_IdExp("a")))),
  A_PrintStm(A_LastExpList(A_IdExp("b")))));
```

12

在目录$TIGER/chap1中可以找到包含树的数据类型声明的文件以及这个样板程序。

编写没有副作用（即更新变量和数据结构的赋值语句）的解释器是理解*指称语义*（denotational semantic）和*属性文法*（attribute grammar）的好方法，后两者都是描述程序设计语言做什么的方法。对编写编译器而言，它也常常是很有用的技术，因为编译器也需要知道程序设计语言做的是什么。

因此，在实现这些程序时，除了初始化，决不要给任何变量或结构成员赋予新值。对于局部变量，要使用带初始化的声明形式（例如，int i = j + 3;），并且对于每一种结构（struct），要类似于第5页例子中的A_CompoundStm那样，用一个构造函数来分配它们并给它的各个成员赋初值。