AWK 程序设计语言

Alfred V.Aho Brian W.Kernighan Peter J.Weinberger

https://github.com/wuzhouhui/awk

2015 年 11 月 23 日

目录

第一章 快速入门 **1**

1.1 开始 1

AWK 程序的结构 2

运行 AWK 程序 3

错误 4

1.2 简单的输出 4

打印每一行 5

打印某些字段 5

NF, 字段的数量 5

计算和打印 5

打印行号 6

将文本放入输出中 6

1.3 更精美的输出 6

字段排列 7

输出排序 7

1.4 选择 8

通过比较进行选择 8

通过计算进行选择 8

通过文本内容选择 8

模式的组合 9

数据验证 10

BEGIN 与 END 10

1.5 用 AWK 计算 10

计数 11

计算总和与平均数 11

操作文本 11

字符串拼接 12

打印最后一行 12

内建函数 12

行, 单词与字符的计数 13

1.6 流程控制语句 13

If-Else 语句 13

While 语句 14
For 语句 14
1.7 数组 15
1.8 实用“一行”手册 16
1.9 接下来 18

第二章 Awk 语言 **19**
2.1 模式 20
BEGIN 与 END 21
将表达式用作模式 22
字符串匹配模式 23
正则表达式 25
复合模式 28
范围模式 29
模式总结 30
2.2 动作 30
表达式 31
流程控制语句 42
空语句 44
数组 45
2.3 用户自定义函数 47
2.4 输出 48
print 语句 49
输出分隔符 50
printf 语句 50
输出到文件 52
输出到管道 52
关闭文件与管道 53
2.5 输入 53
输入分隔符 54
多行记录 54
getline 函数 55
命令行变量赋值 56
命令行参数 56
2.6 与其他程序的交互 58
system 函数 58
用 AWK 制作 Shell 命令 58
2.7 小结 59

第三章 数据处理 **60**
3.1 数据转换与归约 60

列求和 60
计算百分比与分位数 62
带逗号的数 64
字段固定的输入 65
程序的交叉引用检查 66
格式化的输出 67
3.2 数据验证 69
对称的分隔符 70
密码文件检查 71
自动生成数据验证程序 72
AWK 的版本 72
3.3 打包与拆包 74
3.4 多行记录 75
由空行分隔的记录 75
处理多行记录 77
带有头部和尾部的记录 78
名字-值 78
3.5 小结 81

第四章 报表与数据库 **82**
4.1 报表生成 82
一个简单的报表 82
更复杂的报表 85
4.2 打包的查询与报表 92
套用信函 94
4.3 关系数据库系统 95
自然连接 96
`relfile` 99
q, 类 awk 查询语言 100
qawk, *q*-to-awk 翻译器 101
4.4 小结 104

第五章 文本处理 **105**
5.1 随机文本发生器 105
随机选择 105
废话生成器 106
随机语句 107
5.2 交互式的文本处理 110
技巧测试之运算 110
技巧测试之测验 111
5.3 文本处理 113

单词计数 . . . 113
文本格式化 . . . 114
维护手稿的交叉引用 . . . 114
制作 KWIC 索引 . . . 117
制作索引 . . . 118
5.4 小结 . . . 124

第六章 小型语言 **126**
6.1 汇编程序与解释程序 . . . 127
6.2 画图语言 . . . 130
6.3 **sort** 生成器 . . . 135
6.4 逆波兰式计算器 . . . 137
6.5 中缀计算器 . . . 140
6.6 递归下降语法分析 . . . 142
6.7 小结 . . . 148

第七章 算法实验 **150**
7.1 排序 . . . 150
插入排序 . . . 150
快速排序 . . . 157
堆排序 . . . 159
7.2 剖析 . . . 163
7.3 拓扑排序 . . . 166
广度优先拓扑排序 . . . 167
深度优先搜索 . . . 169
深度优先拓扑排序 . . . 170
7.4 Make: 文件更新程序 . . . 171
7.5 小结 . . . 175

第一章　快速入门

1

Awk 是一种使用方便且表现力很强的编程语言, 它可以应用在多种不同的计算与数据处理任务中. 这一章是一个简短的教程, 旨在让你尽可能快地写出自己的 awk 程序. 第 二 章对整个 awk 语言进行描述, 剩下的章节展示了在多种不同的领域中, 如何使用 awk 解决问题. 在书中出现的例子, 读者应该会感到非常有用, 有趣且具有指导作用.

1.1　开始

实用的 awk 程序通常都很短, 只有一两行. 假设你有一个文件, 叫作 `emp.data`, 这个文件包含有名字, 每小时工资 (以美元为单位), 工作时长, 每一行代表一个雇员的记录, 就像这样

```
Beth    4.00    0
Dan     3.75    0
Kathy   4.00    10
Mark    5.00    20
Mary    5.50    22
Susie   4.25    18
```

现在你想要打印每位雇员的名字以及他们的报酬 (每小时工资乘以工作时长), 而雇员的工作时长必须大于零. 这种类型的工作是 awk 的设计目标之一, 所以会很简单. 只要键入下面这一行即可:

```
awk '$3 > 0 { print $1, $2 * $3 }' emp.data
```

2

该行命令告诉操作系统运行 awk 程序, 被运行的程序用单引号包围起来, 从输入文件 `emp.data` 获取数据. 被单引号包围的部分是一个完整的 awk 程序. 它由一个单独的 **模式–动作** 语句 (*pattern-action statement*) 组成. 模式 `$3 > 0` 匹配每一行输入, 如果该行的第三列 (或者说 **字段** (*field*)) 大于零, 则动作

```
{ print $1, $2 * $3 }
```

就会为每一个匹配行打印第一个字段, 以及第二与第三个字段的乘积.

如果你想知道哪些员工在偷懒, 键入

```
awk '$3 == 0 { print $1 }' emp.data
```

模式 `$3 == 0` 匹配第三个字段为零的行, 动作

```
{ print $1 }
```

打印该行的第一个字段.

当你阅读这本书时, 请尝试运行并修改书中的程序. 由于大多数程序都很简短, 你可以快速理解 awk 的工作方式. 在 Unix 系统中, 可以这样运行上面提到的两个示例程序:

```
$ awk '$3 > 0 { print $1, $2 * $3 }' emp.data
Kathy 40
Mark 100
Mary 121
Susie 76.5
$ awk '$3 == 0 { print $1 }' emp.data
Beth
Dan
$
```

行首的字符 $ 是 Shell 的命令提示符, 在你的机器上或许会不一样.

AWK 程序的结构

现在让我们回退一步, 看一下到底发生了什么. 在上面的命令行中, 被单引号包围的部分是使用 awk 语言编写的程序. 本章的每一个 awk 程序都是由一个或多个 模式–动作 语句组成的序列:

pattern { *action* }
pattern { *action* }
...

awk 的基本操作是在由输入行组成的序列中, 陆续地扫描每一行, 搜索可以被模式 **匹配** (*match*) 的行. "匹配" 的精确含义依赖于问题中的模式, 比如, 对于 $3 > 0, 意味着 "条件为真". 3

每一个输入行轮流被每一个模式测试. 每匹配一个模式, 对应的动作 (可能包含多个步骤) 就会执行. 然后下一行被读取, 匹配重新开始. 这个过程会一起持续到所有的输入读取完毕.

上面的程序是模式与动作的典型例子.

```
$3 == 0 { print $1 }
```

是一个单一的 模式–动作 语句, 如果某行的第 3 个字段为 0, 那么它的第 1 个字段就会被打印出来.

在一个 模式–动作 语句中, 模式或动作可以省略其一, 但不能两者同时被省略. 如果一个模式没有动作, 例如

```
$3 == 0
```

会将每一个匹配行 (也就是条件判断为真的行) 打印出来. 这个程序将文件 emp.data 中第 3 个字段为 0 的两行打印出来:

```
Beth    4.00    0
Dan     3.75    0
```

如果只有动作而没有模式, 例如

```
{ print $1 }
```

对于每一个输入行, 动作 (在这个例子里是打印第 1 个字段) 都会被执行.

因为模式与动作都是可选的, 所以用花括号将动作包围起来, 以便区分两者.

运行 AWK 程序

运行一个 awk 程序有多种方式. 你可以键入下面这种形式的命令

awk '*program*' *input files*

这个命令对指定的输入文件的每一行, 执行 *program*. 例如你可以键入

```
awk '$3 == 0 { print $1 }' file1 file2
```

来打印文件 `file1` 与 `file2` 的每一行的第一个字段 (条件是该行的第 3 个字段为 0).

你也可以在命令行上省略输入文件, 只要键入

awk '*program*'

在这种情况下, awk 会将 *program* 应用到你接下来在终端输入的内容上面, 直到你键入一个文件结束标志 (Unix 系统是组合键 control-d). 下面是一个在 Unix 上运行的例子 4

```
$ awk '$3 == 0 { print $1 print $1 }'
Beth    4.00    0
Beth
Dan     3.75    0
Dan
Kathy   3.75    10
Kathy   3.75    0
Kathy
...
```

由 awk 打印的字符加粗显示.

这种行为对测试 awk 程序来说非常方便: 键入程序与数据, 检查程序的输出. 我们再次建议你运行并修改书中的程序.

注意到, 命令行中的程序被单引号包围. 这个规定可以防止程序中的字符 (例如 $) 被 shell 解释, 也可以让程序的长度多于一行.

当程序的长度比较短时 (只有几行), 这种安排会比较方便. 如果程序比较长, 更好的做法是将它们放在一个单独的文件中, 如果文件名是 `progfile` 的话, 运行时只要键入

awk -f progfile *optional list of files*

选项 `-f` 告诉 awk 从文件中提取程序. 在 `progfile` 出现的地方可以是任意的文件名.

错误

如果你在 awk 程序犯了一个错误, awk 会显示一个诊断信息. 例如, 如果你打错了一个花括号, 就像这样

```
awk '$3 == 0 [ print $1 }' emp.data
```

你将会收到一个这样的消息

```
awk: syntax error at source line 1
context is
        $3 == 0 >>> [ <<<
        extra }
        missing ]
awk: Bailing out at source line 1
```

“Syntax error” 意味着你犯了一个语法错误, 这个错误被发现的地方用 >>> <<< 标记. “Bailing out” 意味着无法恢复. 有时候, 你可以得到更多的关于错误的信息, 例如, 错误信息报告了一个不匹配的花括号或括号.

由于发生了语法错误, awk 不会尝试执行这个程序. 然而有些错误直到运行时才会检测到. 例如, 你的程序尝试用 0 作除数, 这时候 awk 会停止处理, 接着报告输入行的行号, 以及程序中尝试进行除法运算的代码所在的行号.

1.2　简单的输出

5

本章的余下部分包含了一系列简短并且典型的 awk 程序, 这些程序都是对文件 emp.data 进行处理. 我们会简单地介绍这些程序是怎么工作的, 但这些例子主要用于阐述有用的操作, 这些操作很容易用 awk 完成, 包括打印字段, 选择输入, 以及变换数据. 我们不会展现 awk 所能做的所有事情, 也不会对细节作过多的解释. 但是阅读完这一章, 你将有能力利用 awk 完成相当数量的工作, 而且会发现阅读后面的章节变得更加容易.

我们只将程序的主体显示出来, 而不是完整的命令行. 在每一种情况下, 程序或者可以被包围在一对单引号中, 作为 awk 命令的第一个参数来运行, 也可以将其放入一个文件中, 通过带有 -f 选项的 awk 命令来运行.

awk 的数据只有两种类型: 数值与由字符组成的字符串. 文件 emp.data 是很典型的待处理数据, 它既含有单词, 也包括数值, 且字段之间通过制表符或空格分隔.

awk 从它的输入中每次读取一行, 将行分解为一个个的字段 (默认将字段看作是非空白字符组成的序列). 当前输入行的第一个字段叫作 $1, 第二个是 $2, 依次类推. 一整行记为 $0. 每行的字段数有可能不一样.

通常情况下, 我们需要做的是打印每一行的部分或全部字段, 也可能会做一些计算. 这一节中的所有程序都是这种形式.

打印每一行

如果一个动作没有模式, 对于每一个输入行, 该动作都会被执行. 语句 print 会打印每一个当前输入行, 所以程序

```
{ print }
```

会将它所有的输入打印到标准输出. 因为 $0 表示一整行, 所以程序

```
{ print $0 }
```

完成同样的工作.

打印某些字段

在一个 print 语句中可以将多个条目打印到同一个输出行中. 打印每一个输入行的第 1 与第 3 个字段的程序是

```
{ print $1, $3 }
```

当 emp.data 作为输入时, 它会输出

```
Beth 0
Dan 0
Kathy 10
Mark 20
Mary 22
Susie 18
```

在 print 语句中由逗号分隔的表达式, 在输出时默认用一个空格符分隔. 由 print 打印的每一行都由一个换行符终止. 这些默认行为都可以修改, 我们将在第 二 章讨论如何修改.

NF, 字段的数量

有时候, 你必须总是通过 $1, $2 这样的形式引用字段, 但是任何表达式都可以出现在 $ 的后面, 用来指明一个字段的编号: 表达式被求值, 求出的值被当作字段的编号. Awk 计算当前输入行的字段数量, 并将它存储在一个内建的变量中, 这个变量叫作 NF. 因此程序

```
{ print NF, $1, $NF }
```

将会打印每一个输入行的字段数量, 第一个字段, 以及最后一个字段.

计算和打印

你也可以用字段的值进行计算, 并将计算得到的结果放在输出语句中. 程序

```
{ print $1, $2 * $3 }
```

是一个很典型的例子, 它会打印雇员的名字与报酬 (每小时工资乘以工作时长):

```
Beth 0
Dan 0
Kathy 40
Mark 100
Mary 121
Susie 76.5
```

我们待会儿就会展示如何将输出做得更好看.

打印行号

Awk 提供了另一个内建变量 NR, 这个变量计算到目前为止, 读取到的行的数量. 我们可以使用 NR 和 $0 为 emp.data 的每一行加上行号:

```
{ print NR, $0 }
```

输出就像这样:

```
1 Beth    4.00    0
2 Dan     3.75    0
3 Kathy   4.00    10
4 Mark    5.00    20
5 Mary    5.50    22
6 Susie   4.25    18
```

将文本放入输出中

你也可以把单词放在字段与算术表达式之间:

```
{ print "total pay for", $1, "is", $2 * $3 }
```

输出

```
total pay for Beth is 0
total pay for Dan is 0
total pay for Kathy is 40
total pay for Mark is 100
total pay for Mary is 121
total pay for Susie is 76.5
```

在 print 语句中, 被双引号包围的文本会和字段, 以及运算结果一起输出.

1.3　更精美的输出

print 用于简单快速的输出. 如果你想要格式化输出, 那么就需要使用 printf 语句. 正如我们将要在 2.4 节看到的那样, printf 几乎可以产生任何种类的输出, 但在这一节, 我们仅仅展现它的一小部分能力.

字段排列

printf 语句具有形式

printf(*format*, $value_1$, $value_2$, ... , $value_n$)

format 是一个字符串, 它包含按字面打印的文本, 中间散布着格式说明符, 格式说明符用于说明如何打印值. 一个格式说明符是一个 %, 后面跟着几个字符, 这些字符控制一个 *value* 的输出格式. 第一个格式说明符说明 $value_1$ 的输出格式, 第二个格式说明符说明 $value_2$ 的输出格式, 依次类推. 于是, 格式说明符的数量应该和被打印的 *value* 一样多.

这个程序使用 printf 打印每位雇员的报酬:

```
{ printf("total pay for %s is $%.2f\n", $1, $2 * $3) }
```

这个 printf 语句的格式字符串包含两个格式说明符. 第一个格式说明符 %s, 是说将第一个值 $1, 以字符串的形式打印; 第二个格式说明符 %.2f, 是说将第二个值 $2*$3, 按照数字格式打印, 且带有两位小数. 格式字符串的其他内容 (包括美元符) 按照字面值打印; 字符串末尾的 \n 表示换行符, 该符号使后面的输出从下一行开始. 当 emp.data 作为输入时, 这个程序输出: 8

```
total pay for Beth is $0.00
total pay for Dan is $0.00
total pay for Kathy is $40.00
total pay for Mark is $100.00
total pay for Mary is $121.00
total pay for Susie is $76.50
```

使用 printf 不会自动产生空格符或换行符; 你必须自己创建它们, 不要忘了 \n.

另外一个程序打印每位雇员的名字与报酬:

```
{ printf("%-8s $%6.2f\n", $1, $2 * $3) }
```

第一个格式说明符 %-8s, 将名字左对齐输出, 占用 8 个字符的宽度. 第二个格式说明符 %6.2f, 将报酬以带有两位小数的数字格式打印出来, 数字至少占用 6 个字符的宽度:

```
Beth     $  0.00
Dan      $  0.00
Kathy    $ 40.00
Mark     $100.00
Mary     $121.00
Susie    $ 76.50
```

更多的关于 printf 的例子会慢慢加以介绍, 而完整描述在 2.4 节.

输出排序

设想一下你想要为每一位雇员打印所有的数据, 包括他的报酬, 报酬按照升序排列. 最简单的办法是使用 awk 在每一位雇员的记录前加上报酬, 然后再通过一个排序程序进行排序, 在 Unix 中, 命令行[①]

[①] 原文中 sort 命令没有带上 -n 选项. — 译者注

```
awk '{ printf("%6.2f %s\n", $2 * $3, $0) }' emp.data | sort -n
```

将 awk 的输出通过管道传递给 sort 命令, 最后输出: 9

```
  0.00 Beth    4.00    0
  0.00 Dan     3.75    0
 40.00 Kathy   4.00    10
 76.50 Susie   4.25    18
100.00 Mark    5.00    20
121.00 Mary    5.50    22
```

1.4 选择

Awk 的模式非常擅长从输入中选择感兴趣的行, 以便进行进一步的处理. 因为一个没有动作的模式会将所有匹配的行打印出来, 所以许多 awk 程序仅含有单个模式. 这一节给出的的例子, 其模式具有很高的实用价值.

通过比较进行选择

这个程序使用一个比较模式来选择某些雇员的记录, 条件是他的每小时工资大于等于 $5.00, 也就是第二个字段大于等于 5:

```
$2 >= 5
```

它从 emp.data 中选择这些行:

```
Mark    5.00    20
Mary    5.50    22
```

通过计算进行选择

程序

```
$2 * $3 > 50 { printf("$%.2f for %s\n", $2 * $3, $1) }
```

打印那些报酬超过 $50 的雇员:

```
$100.00 for Mark
$121.00 for Mary
$76.50 for Susie
```

通过文本内容选择

除了数值选择, 你也可以选择那些包含特定单词或短语的输入行. 这个程序打印所有第一个字段是 Susie 的行:

```
$1 == "Susie"
```

操作符 == 测试相等性. 你也可以搜索含有任意字母, 单词或短语的文本, 通过一个叫做**正则表达式** (*regular expressions*) 的模式来完成. 这个程序打印所有包含 Susie 的行: 10

```
/Susie/
```

输出是

```
Susie   4.25    18
```

正则表达式可以用来指定非常精细的模式, 2.1 节包含了一个完整的讨论.

模式的组合

模式可以使用括号和逻辑运算符进行组合, 逻辑运算符包括 &&, ||, 和 !, 分别表示 AND, OR, 和 NOT. 程序

```
$2 >= 4 || $3 >= 20
```

打印那些 $2 至少为 4, 或者 $3 至少为 20 的行:

```
Beth    4.00    0
Kathy   4.00    10
Mark    5.00    20
Mary    5.50    22
Susie   4.25    18
```

两个条件都满足的行只输出一次. 将这个程序与下面这个程序作对比, 它包含两个模式:

```
$2 >= 4
$3 >= 20
```

如果某行对这两个条件都满足, 它会被打印两次

```
Beth    4.00    0
Kathy   4.00    10
Mark    5.00    20
Mark    5.00    20
Mary    5.50    22
Mary    5.50    22
Susie   4.25    18
```

注意程序

```
!($2 < 4 && $3 < 20)
```

打印的行不满足这样的条件: $2 小于 4, 并且 $3 也小于 20; 这个条件判断与上面的第一个等价, 虽然在可读性方面差了一点.

数据验证

真实的数据总是存在错误. 检查数据是否具有合理的值, 格式是否正确, 这种任务通常称作**数据验证**(*data validation*), 在这一方面 awk 是一款非常优秀的工具.

数据验证在本质上是否定: 不打印具有期望的属性的行, 而是打印可疑行. 接下来的程序使用比较模式, 将 5 条合理性测试应用到 emp.data 的每一行:

```
NF != 3   { print $0, "number of fields is not equal to 3" }
$2 < 3.35 { print $0, "rate is below minimum wage" }
$2 > 10   { print $0, "rate exceeds $10 per hour" }
$3 < 0    { print $0, "negative hours worked" }
$3 > 60   { print $0, "too many hours worked" }
```

如果数据没有错误, 就不会有输出.

BEGIN 与 END

特殊的模式 BEGIN 在第一个输入文件的第一行之前被匹配, END 在最后一个输入文件的最后一行被处理之后匹配. 这个程序使用 BEGIN 打印一个标题:

```
BEGIN { print "NAME     RATE    HOURS"; print "" }
      { print }
```

输出是

```
NAME     RATE    HOURS

Beth    4.00    0
Dan     3.75    0
Kathy   4.00    10
Mark    5.00    20
Mary    5.50    22
Susie   4.25    18
```

你可以在同一行放置多个语句, 语句之间用分号分开. 注意 print "" 打印一个空行, 它与一个单独的 print 并不相同, 后者打印当前行.

1.5　用 AWK 计算

一个动作就是一个语句序列, 语句之间用分号或换行符分开. 你已经见过只有一个单独的 print 语句的动作. 这一小节提供的例子所包含的语句可以用来进行简单的数学或字符串计算. 在这些语句里, 你不仅可以使用内建变量, 比如 NF, 还可以自己定义变量, 这些变量可以用来计算, 存储数据, 等等. 在 awk 中, 用户创建的变量不需要事先声明就可以使用.

计数

这个程序用一个变量 emp 计算工作时长超过 15 个小时的员工人数:

```
$3 > 15 { emp = emp + 1 }
END     { print emp, "employees worked more than 15 hours" }
```

对每一个第三个字段超过 15 的行, 变量 emp 的值就加 1. 用 emp.data 作输入数据, 这个程序输出:

```
3 employees worked more than 15 hours
```

当 awk 的变量作为数值使用时, 默认初始值为 0, 所以我们没必要初始化 emp.

计算总和与平均数

为了计算雇员的人数, 我们可以使用内建变量 NR, 它的值是到目前为止读取到的行数; 当所有输入都处理完毕时, 它的值就是读取到的行数.

```
END {print NR, "employees" }
```

输出是:

```
6 employees
```

这里有个程序利用 NR 来计算平均报酬:

```
{ pay = pay + $2 * $3 }
END { print NR, "employees"
      print "total pay is", pay
      print "average pay is", pay / NR
    }
```

第一个动作累加所有雇员的报酬. END 动作打印:

```
6 employees
total pay is 337.5
average pay is 56.25
```

很明显, printf 可以用来产生更加美观的输出. 这个程序有一个潜在的错误: 一种不常见的情况是 NR 的值为 0, 程序会尝试将 0 作除数, 此时 awk 就会产生一条错误消息.

操作文本

Awk 的长处之一是它可以非常方便地对字符串进行操作, 就像大多数其他语言操作数字那样方便. Awk 的变量除了可以存储数值, 还可以存储字符串. 这个程序搜索每小时工资最高的雇员:

```
$2 > maxrate { maxrate = $2; maxemp = $1 }
END { print "highest hourly rate:", maxrate, "for", maxemp }
```

它的输出是:

```
highest hourly rate: 5.50 for Mary
```

在这个程序里, 变量 maxrate 保存的是数值, 而 maxemp 保存的是字符串. (如果有多个雇员都拥有相同的最高每小时工资, 这个程序只会打印第一个人的名字.)

字符串拼接

可以通过组合旧字符串来生成一个新字符串; 这个操作叫作**拼接** (*concatenation*). 程序

```
    { names = names $1 " " }
END { print names }
```

将所有雇员的名字都收集到一个单独的字符串中, 每一次拼接都是把名字与一个空格符添加到变量 names 的值的末尾. names 在 END 动作中被打印出来:

```
Beth Dan Kathy Mark Mary Susie
```

在一个 awk 程序中, 字符串的拼接操作通过陆续写出字符串来完成. 对每一个输入行, 上面程序中的第一条语句将三个字符串连接在一起: names 早先的值, 第一个字段, 以及一个空格; 然后再将结果字符串赋值给 names. 于是, 当所有的输入行都读取完毕时, names 包含有一个由所有雇员名字组成的, 每个名字之间由空格分隔的字符串. 用来存储字符串的变量的初始值默认为空字符串 (也就是说该字符串不包含任何字符), 因此在这个程序里, names 不需要显式地初始化.

打印最后一行

虽然在 END 动作里, NR 的值被保留了下来, 但是 $0 却不会. 程序

```
    { last = $0 }
END { print last }
```

可以用来打印文件的最后一行:

```
Susie   4.25    18
```

内建函数

我们已经看到 awk 提供有内建变量, 这些变量可以用来维护经常需要用到的量, 比如字段的个数, 以及当前输入行的行号. 同样, awk 也提供用来计算其他值的内建函数. 求平方根, 取对数, 随机数, 除了这些数学函数, 还有其他用来操作文本的函数. 其中之一是 length, 它用来计算字符串中字符的个数. 例如, 这个程序计算每一个人的名字的长度:

```
{ print $1, length($1) }
```

程序运行结果是:

```
Beth 4
Dan 3
Kathy 5
```

```
Mark 4
Mary 4
Susie 5
```

行, 单词与字符的计数

这个程序使用 length, NF 与 NR 计算行, 单词与字符的数量, 为方便起见, 我们将每个字段都当成一个单词.

```
{ nc = nc + length($0) + 1
  nw = nw + NF
}
END { print NR, "lines,", nw, "words,", nc, "characters" }
```

文件 emp.data 含有

```
6 lines, 18 words, 77 characters
```

我们为每一个输入行末尾的换行符加 1, 这是因为 $0 不包含换行符.

1.6 流程控制语句

Awk 提供有用于决策的 if-else 语句, 以及循环语句, 所有的这些都来源于 C 语言. 它们只能用在动作 (Action) 里.

If-Else 语句

下面这个程序计算每小时工资多于 $6.00 的雇员的总报酬与平均报酬. 在计算平均数时, 它用到了 if 语句, 避免用 0 作除数.

```
$2 > 6 { n = n + 1; pay = pay + $2 * $3 }
END    { if (n > 0)
           print n, "employees, total pay is", pay,
                    "average pay is", pay/n
       else
           print "no employees are paid more than $6/hour"
       }
```

emp.data 的输出是:

```
no employees are paid more than $6/hour
```


在 if-else 语句里, if 后面的条件被求值, 如果条件为真, 第一个 print 语句执行, 否则是第二个 print 语句被执行. 注意到, 在逗号后面断行, 我们可以将一个长语句延续到下一行.

While 语句

一个 while 含有一个条件判断与一个循环体. 当条件为真时, 循环体执行. 下面这个程序展示了一笔钱在一个特定的利率下, 其价值如何随着投资时间的增长而增加, 价值计算的公式是 $value = amount(1 + rate)^{years}$.

```
# interest1 - compute compound interest
#   input:  amount  rate  years
#   output: compounded value at the end of each year

{   i = 1
    while (i <= $3) {
        printf("\t%.2f\n", $1 * (1 + $2) ^ i)
        i = i + 1
    }
}
```

while 后面被括号包围起来的表达式是条件判断; 循环体是跟在条件判断后面的, 被花括号包围起来的的两条语句. printf 格式控制字符串里的 \t 表示一个制表符; ^ 是指数运算符. 从井号 (#) 开始, 直到行末的文本是 **注释** (*comment*), 注释会被 awk 忽略, 但有助于其他人读懂程序.

你可以键入三个数, 看一下不同的本金, 利率和时间会产生怎样的结果. 举个例子, 这个交易展示了 $1000 在 6% 与 12% 的利率下, 在 5 年的时间里如何升值:

```
$ awk -f interest1
1000 .06 5
        1060.00
        1123.60
        1191.02
        1262.48
        1338.23
1000 .12 5
        1120.00
        1254.40
        1404.93
        1573.52
        1762.34
```

For 语句

16

大多数循环都包括初始化, 测试, 增值, 而 for 语句将这三者压缩成一行. 这里是前一个计算投资回报的程序, 不过这次用 for 循环:

```
# interest2 - compute compound interest
#   input:  amount  rate  years
```

```
#   output: compounded value at the end of each year

{   for (i = 1; i <= $3; i = i + 1)
        printf("\t%.2f\n", $1 * (1 + $2) ^ i)
}
```

初始化语句 `i = 1` 只执行一次. 接下来, 判断条件 `i <= $3` 是否成立; 如果测试结果为真, 循环体的 `printf` 语句被执行. 执行完循环体之后, 增值语句 `i = i + 1` 执行, 循环的下一次迭代从条件的另一次测试开始. 代码很紧凑, 因为循环体只有一条语句, 也就不再需要花括号.

1.7 数组

Awk 提供了数组, 用来存储一组相关的值. 虽然数组给予了 awk 非常可观的力量, 但是我们在这里只展示一个简单的例子. 下面这个程序按行逆序显示输入数据. 第一个动作将输入行放入数组 `line` 的下一个元素中; 也就是说, 第一行放入 `line[1]`, 第二行放入 `line[2]`, 依次类推. `END` 动作用一个 `while` 循环, 从数组的最后一个元素开始打印, 一直打印到第一个元素为止:

```
# reverse - print input in reverse order by line

    { line[NR] = $0 }  # remember each input line

END { i = NR           # print lines in reverse order
      while (i > 0) {
          print line[i]
          i = i - 1
      }
    }
```

用 `emp.data` 作输入, 输出是

```
Susie   4.25    18
Mary    5.50    22
Mark    5.00    20
Kathy   4.00    10
Dan     3.75    0
Beth    4.00    0
```

这是用 `for` 循环实现的等价的程序: 17

```
# reverse - print input in reverse order by line

    { line[NR] = $0 }  # remember each input line

END { for (i = NR; i > 0; i = i - 1)
```

```
        print line[i]
}
```

1.8　实用“一行”手册

虽然 awk 可以写出非常复杂的程序，但是许多具有实用价值的程序并不比我们目前为止看到的复杂多少. 这里有一些小程序集合，对你应该会有一些参考价值. 大多数是我们已经讨论过的程序的变形.

1. 输入行的总行数

```
END { print NR }
```

2. 打印第 10 行

```
NR == 10
```

3. 打印每一个输入行的最后一个字段

```
{ print $NF }
```

4. 打印最后一行的最后一个字段

```
{ field = $NF }
END { print field }
```

5. 打印字段数多于 4 个的输入行

```
NF > 4
```

6. 打印最后一个字段值大于 4 的输入行

```
$NF > 4
```

7. 打印所有输入行的字段数的总和

```
{ nf = nf + NF }
END { print nf }
```

8. 打印包含 Beth 的行的数量

```
/Beth/ { nlines = nlines + 1 }
END { print nlines }
```

9. 打印具有最大值的第一个字段，以及包含它的行 (假设 $1 总是 正的)

```
$1 > max { max = $1; maxline = $0 }
END { print max, maxline }
```

10. 打印至少包含一个字段的行

```
$NF > 0
```

11. 打印长度超过 80 个字符的行

```
length($0) > 80
```

12. 在每一行的前面加上它的字段数

```
{ print NF, $0 }
```

13. 打印每一行的第 1 与第 2 个字段, 但顺序相反

```
{ print $2, $1 }
```

14. 交换每一行的第 1 与第 2 个字段, 并打印该行

```
{ temp = $1; $1 = $2; $2 = temp; print }
```

15. 将每一行的第一个字段用行号代替

```
{ $1 = NR; print }
```

16. 打印删除了第 2 个字段后的行

```
{ $2 = ""; print }
```

17. 将每一行的字段按逆序打印

```
{ for (i = NF; i > 0; i = i - 1) printf("%s ", $i)
  printf("\n")
}
```

18. 打印每一行的所有字段值之和

```
{ sum = 0
  for (i = 1; i <= NF; i = i + 1) sum = sum + $i
  print sum
}
```

19. 将所有行的所有字段值累加起来

```
    { for (i = 1; i <= NF; i = i + 1) sum = sum + $i }
END { print sum }
```

20. 将每一行的每一个字段用它的绝对值替换

```
{ for (i = 1; i <= NF; i = i + 1) if ($i < 0) $i = -$i
  print
}
```

1.9　接下来

19

你已经见识过了 awk 的要点. 本章的每个程序都是由多个 模式–动作 语句组成的序列. Awk 用模式测试每一个输入行, 如果模式匹配, 对应的动作就会执行. 模式可以包括数值或字符串比较, 动作也可以包含计算和格式化输出. 除了可以自动从输入文件中读取数据, awk 还会将每一个输入行分割为字段. 它还提供了一系列的内建变量与函数, 当然你也可以自己定义. 有了这些特征的帮助, 许多实用的计算可以用非常简短的程序实现 — 如果使用其他语言实现同样的功能, 那么就要考虑到许多细节, 而 awk 程序可以隐式处理这些细节.

本书的剩下部分将会详细讨论这些基本概念. 由于有些程序会比本章给出的示例程序大一些, 我们强烈建议你尽可能开始自己写程序. 这会使你对 awk 更加熟悉, 也更容易理解稍大一些的程序. 更进一步讲, 没有什么能比一个简单的实验更能说明问题. 你还是应该浏览整本书; 每一个例子都传达出一些关于 awk 的东西, 可能是关于如何使用一个语言特性, 也可能是如何创建一个有趣的程序.

20

第二章　Awk 语言

21

这一章解释 —大部分都带有例子 —组成一个 awk 程序所需要的构造要素. 由于这次要描述的是整个语言, 所以材料会非常琐细, 我们建议读者只需要浏览一下即可, 当需要时再回来查阅细节.

最简单的 awk 程序是一个由多个 模式–动作 语句构成的序列:

pattern { *action* }
pattern { *action* }
...

在某些语句中, 模式可以不存在; 还有些语句, 动作及其包围它的花括号也可以不存在. 如果程序经过 awk 检查后没有发现语法错误, 它就会每次读取一个输入行, 对读取到的每一行, 按顺序检查每一个模式. 对每一个与当前行匹配的模式, 对应的动作就会执行. 一个缺失的模式匹配每一个输入行, 因此每一个不带有模式的动作对每一个输入行都会执行. 只含有模式而没有动作的语句, 会打印每一个匹配模式的输入行. 大部分情况下, 在这一章出现的术语 “输入行” 与 “记录” 被当作一对同义词. 在 2.5 节, 我们会讨论多行记录, 多行记录指的由多行数据组成的单个记录.

本章的第一节详细讨论模式. 第二节通过描述表达式, 赋值语句与流程控制语句, 展开对动作的讨论. 剩下的小节包括函数定义, 输出, 输入, 以及 awk 调用其他程序的方式. 大部分小节都包含对主要性质的总结.

输入文件 countries

作为本章许多 awk 程序的输入数据, 我们将使用文件 countries. 每一行都包括一个国家的名字, 面积 (以千平方英里为单位), 人口 (以百万为单位), 以及这个国家所在的大陆. 数据来源于 1984 年, 苏联被归到了亚洲. 在文件里, 四列数据用制表符分隔, 用一个空格分隔 North (South) 与 America.

文件 countries 包含下面几行: 22

```
USSR          8649    275     Asia
Canada        3852    25      North America
China         3705    1032    Asia
USA           3615    237     North America
Brazil        3286    134     South America
India         1267    746     Asia
Mexico        762     78      North America
France        211     55      Europe
Japan         144     120     Asia
```

```
Germany     96      61      Europe
England     94      56      Europe
```

在这一章的剩下部分里, 如果没有显式给出输入数据, 默认将 `countries` 作为输入.

程序格式

模式–动作 语句, 以及动作内的语句通常用换行符分隔, 但是若干条语句也可以出现在同一行, 只要它们之间用分号分开即可. 一个分号可以放在任何语句的末尾.

动作的左花括号必须与它的模式在同一行; 而剩下的部分, 包括右花括号, 则可以出现在下面几行.

空行会被忽略; 它们可以插入在语句之前或之后, 用来提高程序的可读性. 空格与制表符可以出现在运算符与操作数的周围, 同样也是为了提高可读性.

注释可以出现在任意一行的末尾. 一个注释以井号 (#) 开始, 以换行符结束, 正如

```
{ print $1, $3 }    # print country name and population
```

一条长语句可以分散成多行, 只要在断行处插入一个反斜杆即可:

```
{ print \
        $1,     # country name
        $2,     # area in thousands of square miles
        $3 }    # population in millions
```

正如这个例子所呈现的那样, 语句可以在逗号之后断行, 并且注释可以出现在断行的末尾.

在这本书里我们用到了若干种编程风格, 之所以这样做, 一方面是为了比较不同风格之间的差异, 另一方面是为了避免程序占用过多的行. 对于比较短小的程序 —- 就像本章中出现过的那些例子 —- 格式并不是非常重要, 但是一致性与可读性对于大程序的管理非常有帮助.

2.1 模式

23

模式控制着动作的执行: 当模式匹配时, 相应的动作便会执行. 这一小节描述模式的 6 种类型, 以及匹配它们的条件.

模式汇总

1. `BEGIN{` *statements*`}`

   在输入被读取之前, *statements* 执行一次.

2. `END{` *statements*`}`

   当所有输入被读取完毕之后, *statements* 执行一次.

3. *expression*`{` *statements*`}`

   每碰到一个使 *expression* 为真的输入行, *statements* 就执行. *expression* 为真指的是其值非零或非空.

4. */regular expression/* { *statements*}

   当碰到这样一个输入行时, *statements* 就执行: 输入行含有一段字符串, 而该字符串可以被 *regular expression* 匹配.

5. *compound pattern* { *statements*}

   一个复合模式将表达式用 &&(AND), ||(OR), !(NOT), 以及括号组合起来; 当 *compound pattern* 为真时, *statements* 执行.

6. $pattern_1$, $pattern_2$ { *statements*}

   一个范围模式匹配多个输入行, 这些输入行从匹配 $pattern_1$ 的行开始, 到匹配 $pattern_2$ 的行结束 (包括这两行), 对这其中的每一行执行 *statements*.

BEGIN 与 END 不与其他模式组合. 一个范围模式不能是其他模式的一部分. BEGIN 与 END 是唯一两个不能省略动作的模式.

BEGIN 与 END

BEGIN 与 END 这两个模式匹配任何输入行. 实际情况是, 当 awk 从输入读取数据之前, BEGIN 的语句开始执行; 当所有输入数据被读取完毕, END 的语句开始执行. 于是, BEGIN 与 END 分别提供了一种控制初始化与扫尾的方式. BEGIN 与 END 不能与其他模式作组合. 如果有多个 BEGIN, 与其关联的动作会按照它们在程序中出现的顺序执行, 这种行为对多个 END 同样适用. 我们通常将 BEGIN 放在程序开头, 将 END 放在程序末尾, 虽然这并不是强制的.

BEGIN 的一个常用用途是更改输入行被分割为字段的默认方式. 分割字符由一个内建变量 FS 控制. 默认情况下字段由空格或 (和) 制表符分割, 此时 FS 的值被设置为一个空格符. 将 FS 设置成一个非空格字符, 就会使该字符成为字段分割符. 24

下面这个程序在 BEGIN 的动作里将字段字割符设置为制表符 (\t), 并在输出之前打印标题. 第二个 printf 语句 (对每一个输入行它都会执行) 将输出格式化成一张表格, 使得每一列都刚好与标题的列表头对齐. END 打印总和. (变量与表达式在 2.2 节讨论.)

```
# print countries with column headers and totals

BEGIN { FS = "\t"   # make tab the field separator
        printf("%10s %6s %5s   %s\n\n",
              "COUNTRY", "AREA", "POP", "CONTINENT")
      }
      { printf("%10s %6d %5d   %s\n", $1, $2, $3, $4)
        area = area + $2
        pop = pop + $3
      }
END   { printf("\n%10s %6d %5d\n", "TOTAL", area, pop) }
```

如果将 countries 作为输入, 将出将是

```
   COUNTRY   AREA   POP   CONTINENT
```

```
   USSR   8649   275   Asia
 Canada   3852    25   North America
  China   3705  1032   Asia
    USA   3615   237   North America
 Brazil   3286   134   South America
  India   1267   746   Asia
 Mexico    762    78   North America
 France    211    55   Europe
  Japan    144   120   Asia
Germany     96    61   Europe
England     94    56   Europe

  TOTAL  25681  2819
```

将表达式用作模式

就像大多数程序设计语言一样, awk 拥有非常丰富的用来描述数值计算的表达式, 但是与许多语言不同的是, awk 还有用于描述字符串操作的表达式. 贯穿全书, *string* 都表示一个由 0 个或多个字符组成的序列. 这些字符串可以存储在变量中, 也可以以字符串常量的形式出现, 就像 `""` 或 `"Asia"`. 字符串 `""` 不包括任何字符, 叫做 **空字符串** (*null string*). 术语 **子字符串** (*substring*) 表示一个字符串内部的, 由 0 个或多个字符组成的连续序列. 对任意一个字符串, 空字符串都可以看作是该字符串第一个字符之前的, 长度为 0 的子字符串, 或者是一对相邻字符之间的子字符串, 又或者是最后一个字符之后的子字符串.

任意一个表达式都可以用作任意一个运算符的操作数. 如果一个表达式拥有一个数值形式的值, 而 25
运算符要求一个字符串值, 那么该数值会自动转换成字符串, 类似地, 当运算符要求一个数值时, 字符串被自动转换成数值.

任意一个表达式都可以当作模式来使用. 如果一个作为模式使用的表达式, 对当前输入行的求值结果不空或非零, 那么该模式就匹配该行. 典型的表达式模式是那些涉及到数值或字符串比较的表达式. 一个比较表达式包含 6 种关系运算符中的一种, 或者包含两种字符串匹配运算符中的一种: `~` 与 `!~` 在下一小节讨论. 关系运算符列在表 2.1 中.

如果模式是一个比较表达式, 就像 `NF > 10`, 当当前行使该条件满足时, 这个模式就算是匹配该输入行, 在这里条件满足指的是当前输入行的字段数大于 10. 如果模式是一个算术表达式, 就像 `NF`, 如果该表达式的值非零, 那么当前输入行被匹配. 如果模式是一个字符串表达式, 当表达式的字符串值非空时, 当前输入行被匹配.

在一个关系比较中, 如果两个操作数都是数值, 关系比较将会按照数值比较进行; 否则的话, 数值操作数会被转换成字符串, 再将操作数按字符串的形式进行比较. 两个字符串间的比较以字符为单位逐个相比, 字符间的先后顺序依赖于机器的字符集 (大多数情况下是 ASCII 字符集). 一个字符串 “小于” 另一个, 指的是它比另一个字符串更早出现, 例如 `"Canada" < "China"`, `"Asia" < "Asian"`.

模式

```
$3/$2 >= 0.5
```

表 2.1: 比较运算符

运算符	意义
<	小于
<=	小于或等于
==	等于
!=	不等于
>=	大于或等于
>	大于
~	匹配
!~	不匹配

选择的行, 其第 3 个字段除以第 2 个字段所得的商大于 0.5, 而

```
$0 >= "M"
```

26

选择那些以字母 M, N, O 等开头的输入行:

```
USSR        8649    275     Asia
USA 3615    237     North America
Mexico      762     78      North America
```

有时候一个比较运算符的类型不能单单靠表达式表现出来的语法形式来判断. 程序

```
$1 < $4
```

可以以数值的形式, 或者字符串的形式比较输入行的第 1 个与第 4 个字段. 在这里, 比较的类型取决于字段的值, 并且有可能每一行都有不同的情况出现. 文件 countries 的第 1 个与第 4 个字段总是字符串, 所以比较总是以字符串的形式进行; 输出是

```
Canada      3852    25      North America
Brazil      3286    134     South America
Mexico      762     78      North America
England     94      56      Europe
```

只有当两个字段都是数值时, 比较才会以数值的形式进行; 这种情况可以是

```
$2 < $3
```

2.2 节包含了一个更加完整的, 关于字符串, 数值与表达式的讨论.

字符串匹配模式

Awk 提供了一种称为 **正则表达式** (*regular expression*) 的表示法, 它可以用来指定和匹配一个字符串. 正则表达式在 Unix 中使用得非常普遍, 包括文本编辑器与 shell. 受限形式的正则表达式也出现在其他系统中, 在 MS-DOS 中可以用 “通配符” 指定一个文件名集合.

一个 **字符串匹配模式** (*string-matching pattern*) 测试一个字符串是否包含一段可以被正则表达式匹配的子字符串.

最简单的正则表达式是仅由数字与字母组成的字符串, 就像 `Asia`, 它匹配的就是它本身. 为了将一个正则表达式切换成一个模式, 只需要用一对斜杆包围起来即可:

```
/Asia/
```

这个模式匹配那些含有子字符串 `Asia` 的输入行, 例如 `Asia`, `Asian`, 或 `Pan-Asiatic`. 注意, 正则表达式中空格是有意义的: 字符串匹配模式 27

字符串匹配模式

1. */regexpr/*

   当当前输入行包含一段能够被 *regexpr* 匹配的子字符串时, 该模式被匹配.

2. *regular expression* ~ */regexpr/*

   如果 *expression* 的字符串值包含一段能够被 *regexpr* 匹配的子字符时, 该模式被匹配.

3. *expression* !~ */regexpr/*

   如果 *expression* 的字符串值不包含能够被 *regexpr* 匹配的子字符串, 该模式被匹配.

在 ~ 与 !~ 的语境中, 任意一个表达式都可以用来替换 */regexpr/*.

```
/ Asia /
```

只有当 `Asia` 被一对空格包围时才会匹配成功.

上面的模式是三种字符串匹配模式当中的一种. 它的形式是用一对斜杆将正则表达式包围起来:

```
/r/
```

如果某个输入行含有能被 *r* 匹配的子字符串, 则该行匹配成功.

剩下的两种字符串匹配模式使用到了显式的匹配运算符:

```
expression ~ /r/
expression !~ /r/
```

匹配运算符 ~ 的意思是 “被... 匹配”, !~ 的意思是 “不被... 匹配”. 当 *expression* 的字符串值包含一段能够被正则表达式 *r* 匹配的子字串时, 第一个模式被匹配; 当不存在这样的子字符串时, 第二个模式被匹配.

匹配运算符的左操作数经常是一个字段, 模式

```
$4 ~ /Asia/
```

匹配所有第 4 个字段包含 `Asia` 的输入行, 而

```
$4 !~ /Asia/
```

匹配所有第 4 个字段**不**包含 `Asia` 的输入行.

注意到, 字符串匹配模式

```
/Asia/
```

是

```
$0 ~ /Asia/
```

的简写形式.

正则表达式

28

正则表达式是一种用于指定和匹配字符串的表示法. 就像算术表达式一样, 一个正则表达式是一个基本表达式, 或者是多个子表达式通过运算符组合而成. 为了理解被一个正则表达式匹配的字符串, 我们需要先了解被子表达式匹配的字符串.

正则表达式

1. 正则表达式的元字符包括:

   `\ ^ $ . [ ] | ( ) * + ?`

2. 一个基本的表达式包括下面几种:

   一个不是元字符的字符, 例如 `A`, 这个正则表达式匹配的就是它本身.

   一个匹配特殊符号的转义字符: `\t` 匹配一个制表符 (见表 2.2).

   一个被引用的元字符, 例如 `\*`, 按字面意义匹配元字符.

   `^` 匹配一行的开始.

   `$` 匹配一行的结束.

   `.` 匹配任意一个字符.

   一个字符类: `[ABC]` 匹配字符 `A`, `B` 或 `C`.

   字符类可能包含缩写形式: `[A-Za-z]` 匹配单个字母.

   一个互补的字符类: `[^0-9]` 匹配任意一个字符, 但是除了数字.

3. 这些运算符将正则表达式组合起来:

   选择: `A|B` 匹配 `A` 或 `B`.

   拼接: `AB` 匹配后面紧跟着 `B` 的 `A`.

   闭包: `A*` 匹配 0 个或多个 `A`.

   正闭包: `A+` 匹配一个或多个 `A`.

   零或一: `A?` 匹配空字符串或 `A`.

   括号: (r) 匹配与 r 相同的字符串.

基本的正则表达式在上面的表格中列出. 字符

```
\ ^ $ . [ ] | ( ) * + ?
```

叫作 **元字符** (*metacharacter*), 之所以这样称呼是因为它们具有特殊的意义. 一个由单个非元字符构成的正则表达式匹配它自身. 于是, 一个字母或一个数字都算作是一个基本的正则表达式, 与自身相匹配. 为了在正则表达式中保留元字符的字面意义, 需要在字符的前面加上反斜杆. 于是, `\$` 匹配普通字符 `$`. 如果某个字符前面冠有 `\`, 我们就说该字符是被 **引用** (*quoted*) 的.

在一个正则表达式中, 一个未被引用的脱字符 `^` 表示一行的开始, 一个未被引用的美元符 `$` 匹配一行的结束, 一个未被引用的句点 `.` 匹配任意一个字符. 于是,

`^C`	匹配以字符 C 开始的字符串;
`C$`	匹配以字符 C 结束的字符串;
`^C$`	匹配只含有单个字符 C 的字符串;
`^.$`	匹配有且仅有一个字符的字符串;
`^...$`	匹配有且仅有 3 个字符的字符串;
`...`	匹配任意 3 个字符;
`\.$`	匹配以句点结束的字符串.

29

由一组被包围在方括号中的字符组成的正则表达式称为 **字符类** (*character class*); 这个表达式匹配字符类中的任意一个字符. 例如, `[AEIOU]` 匹配 A, E, I, O 或 U.

使用连字符的字符类可以表示一段字符范围. 紧跟在连字符左边的字符定义了范围的开始, 紧跟在连字符右边的字符定义了范围的结束. 于是, `[0-9]` 匹配任意一个数字, `[a-zA-Z][0-9]` 匹配一个后面紧跟着一个数字的字母. 如果左右两边都没有操作数, 那么字符类中的连字符就表示它本身, 所以 `[+-]` 与 `[-+]` 匹配一个 + 或 -. `[A-Za-z-]+` 匹配一个可能包含连字符的单词.

一个 **互补** (*complemented*) 的字符类在 `[` 之后以 `^` 开始. 这样一个类匹配任意一个不在类中的字符, "类中的字符" 指的是方括号内排在脱字符之后的那些字符. 于是, `[^0-9]` 匹配任意一个不是数字的字符; `[^a-zA-Z]` 匹配任意一个不是字母的字符.

`^[ABC]`	匹配以 A, B, 或 C 开始的字符串;
`^[^ABC]`	匹配以任意一个字符 (除了 A, B, 或 C) 开始的字符串;
`[^ABC]`	匹配任意一个字符, 除了 A, B, 或 C;
`^[^a-z]$`	匹配任意一个有且仅有一个字符的字符串, 且该字符不能是小写字母.

在一个字符类中, 所有的字符都具有它自身的字面意义, 除了引用字符 `\`, 互补字符类开头的 `^`, 以及两个字符间的 `-`. 于是, `[.]` 匹配一个句点, `^[^^]` 匹配不以脱字符开始的字符串.

可以使用括号来指定正则表达式中的各个成分如何组合. 有两种二元正则表达式运算符: 选择与拼接. 选择运算符 `|` 用来指定一个选择: 如果 r_1 与 r_2 是正则表达式, 那么 $r_1|r_2$ 所匹配的字符串, 或者与 r_1, 或者与 r_2 匹配.

Awk 不存在显式的拼接运算符. 如果 r_1 与 r_2 是正则表达式, 那么 $(r_1)(r_1)$ (在 (r_1) 与 (r_2) 之间没有空格) 所匹配的字符串具有形式 xy, 其中 x 被 r_1 匹配, y 被 r_2 匹配. r_1 或 r_2 两边的括号可以省略, 如果被括号包围的正则表达式不包含选择运算符的话. 正则表达式

```
(Asian|European|North American) (male|female) (black|blue)bird
```

一共匹配 12 种字符串, 从

```
Asian male blackbird
```

到 30

```
North American female bluebird
```

符号 *, + 与 ? 是一元运算符, 用来指定正则表达式的重复次数. 如果 *r* 是一个正则表达式, 那么 (*r*)* 所匹配的字符串含有零个或连续多个能被 *r* 匹配的子符串. *r*? 匹配的字符串, 要么是空字符串, 要么是能够被 *r* 匹配的字符串. 如果 *r* 是一个基本的正则表达式, 那么括号可以省略.

B*	匹配空字符串, 或 B, BB, 等等.
AB*C	匹配 AC, 或 ABC, ABBC, 等等.
AB+C	匹配 ABC, 或 ABBC, ABBBC, 等等.
AB?C	匹配 AC 或 ABC
[A-Z]+	匹配由一个或多个大写字母组成的字符串.
(AB)+C	匹配 ABC, ABABC, ABABABC, 等等.

在正则表达式中, 选择运算符 | 的优先级最低, 然后是拼接运算, 最后是重复运算符 *, +, 与 ?. 与算术表达式的规则一样, 优先级高的运算符优先处理. 这种规则经常使得括号被省略: ab|cd 等价于 (ab)|(cd), ^ab|cd*e$ 等价于 (^ab)|(c(d*)e$).

为了结束关于正则表达式的讨论, 这里列出了一些比较有用的字符串匹配模式的例子, 这些例子都带有使用了一元与二元运算符的正则表达式, 同时还描述了能够被该模式匹配的输入行. 回想一下, 如果当前输入行含有至少一个能够被 *r* 匹配的子字符串, 那么模式 /*r*/ 匹配成功.

```
^[0-9]+$/
```

匹配含有且只含有数字的输入行.

```
^[0-9][0-9][0-9]$/
```

输入行有且仅有 3 个数字.

```
^(\+|-)?[0-9]+\.?[0-9]*$/
```

十进制小数, 符号与小数部分是可选的.

```
^[+-]?[0-9]+[.]?[0-9]*$/
```

也是匹配十进制小数, 带有可选的符号与小数部分.

```
^[+-]?([0-9]+[.]?[0-9]*|[.][0-9]+)([eE][+-]?[0-9]+)?$/
```

浮点数, 符号与指数部分是可选的.

```
^[A-Za-z][A-Za-z0-9]*$/
```

一个字母, 后面再跟着任意多个字母或数字 (比如 awk 的变量名).

```
^[A-Za-z]$|^[A-Za-z][0-9]$/
```

一个字母, 又或者是一个后面跟着一个数字的字母 (比如 Basic 的变量名).

```
^[A-Za-z][0-9]?$/
```

同样是一个字母, 又或者是一个后面跟着一个数字的字母.

在第 3 个例子中, 为了匹配元字符+ 与 - 的字面值, 必须在它们的前面加上反斜线, 而在字符类中则不需要, 所以第 3 项与第 4 项的功能是等价的. 31

任意一个被一对斜线包围的正则表达式都可以作为匹配运算符的右操作数: 程序

```
$2 !~ /^[0-9]+$/
```

打印那些第 2 个字段不全是数字的行.

在正则表达式与字符串内部, awk 使用一个特定的字符序列 —**转义序列** (*escape sequences*)—来表示那些无法用其他方式表示的字符. 例如, \n 表示一个换行符, 它无法以其他方式出现在字符串或正则表达式中; \b 表示退格符; \t 表示制表符; 007 表示 ASCII 中的响铃符; \/ 表示一个斜杆. 转义序列在 awk 程序中才会有特殊的意义; 如果在数据中, 它们则是普通的字符. 完整的转义序列名单在表 2.2.

表 2.2: 转义序列

序列	意义
\b	退格
\f	换页
\n	换行
\r	回车
t	制表符
ddd	八进制数 *ddd*, *ddd* 含有 1 到 3 个数字, 每个数字的值在 0 到 7 之间
c	其他的字面意义上的 *c* (举例来说, \\ 表示反斜杆, \" 表示双引号)

表 2.3 总结了正则表达式, 以及它们所匹配的字符串. 运算符按优先级递增的顺序列出.

复合模式

一个复合模式是一个组合了其他模式的表达式, 通过括号, 逻辑运算符 ||(OR), &&(AND), !(NOT) 来进行组合. 如果表达式的值为真, 那么复合模式就匹配当前输入行. 下面这个程序使用 AND 运算符来选择那些第 4 个字段是 Asia 且第 3 个字段大于 500 的行: 32

```
$4 == "Asia" && $3 > 500
```

程序

```
$4 == "Asia" || $4 == "Europe"
```

使用 OR 运算符来选择那些第 4 个字段是 Asia 或 Europe 的行. 因为后面这个查询是一个针对字符串的测试, 所以该程序的另一种写法是用到了选择运算符的正则表达式:

```
$4 ~ /^(Asia|Europe)$/
```

(如果两个正则表达式匹配了同一个字符串, 我们就说这两个正则表达式是 **等价** (*equivalent*) 的. 测试一下你对前面所说的正则表达式的规则的理解程度, 下面这两个正则表达式等不等价: ^Asia|Europe$ 与 (Asia|Europe)$?)

如果在其他字段中没有出现 Asia 或 Europe, 那么上面的模式也可以写成

表 2.3: 正则表达式

表达式	匹配
c	非元字符 c
\c	转义序列或字面意义上的 c
^	字符串的开始
$	字符串的结束
.	任意一个字符
[$c_1c_2...$]	任意一个在 $c_1c_2...$ 中的字符.
[^$c_1c_2...$]	任意一个不在 $c_1c_2...$ 中的字符.
[c_1-$c_2...$]	任意一个在范围内的字符, 范围由 c_1 开始, 由 c_2 结束.
[^c_1-$c_2...$]	任意一个不在范围内的字符, 范围由 c_1 开始, 由 c_2 结束.
r_1|r_2	任意一个被 r_1 或 r_2 匹配的字符串.
(r_1)(r_2)	任意一个字串 xy, 其中 r_1 匹配 x, 而 r_2 匹配 y; 如果当中不含有选择运算符, 那么括号是可以省略的
(r)*	零个或连续多个能被 r 匹配的字符串.
(r)+	一个或连续多个能被 r 匹配的字符串.
(r)?	零个或一个能被 r 匹配的字符串. 在这里括号可以省略.
(r)	任意一个能被 r 匹配的字符串.

```
/Asia/ || /Europe/
```

或

```
/Asia|Europe/
```

运算符 || 优先级最低, 再往高是 &&, 最高的是 !. && 与 || 从左至右计算操作数的值, 一旦已经知道整个表达式的值, 计算便停止.

范围模式

一个范围模式由两个被逗号分开的模式组成, 正如

pat_1, pat_2

一个范围模式匹配多个输入行, 这些输入行从匹配 pat_1 的行开始, 到匹配 pat_2 的行结束, 包括这两行; pat_2 可以与 pat_1 匹配到同一行, 这时候模式的范围大小就退化到了一行. 作为一个例子, 模式

```
/Canada/, /USA/
```

匹配的行从包含 Canada 的行开始, 到包含 USA 的行结束.

一旦范围的第一个模式匹配到了某个输入行, 那么整个范围模式的匹配就开始了; 如果范围模式的第二个模式一直都没有匹配到某个输入行, 那么范围模式会一直匹配到输入结束:

33

```
/Europe/, /Africa/
```

输出

```
France  211     55      Europe
Japan   144     120     Asia
Germany 96      61      Europe
England 94      56      Europe
```

在下一个例子里, 变量 FNR 表示从当前输入文件中, 到目前为止读取到的行数, 变量 FILENAME 表示当前输入文件名; 它们两个都是内建变量. 于是, 程序

```
FNR == 1, FNR == 5 { print FILENAME ": " $0 }
```

打印每一个输入文件的前 5 行, 并在每一行的左边加上文件名. 这个程序也可以写成下面这种形式:

```
FNR <= 5 { print FILENAME ": " $0 }
```

一个范围模式不能是其他模式的一部分.

模式总结

表 2.4 总结了可以出现在 模式–动作 语句中的模式种类.

表 2.4: 模式

模式	例子	匹配
BEGIN	BEGIN	输入被读取之前
END	END	所有输入被读取完之后
expression	$3 < 100	第 3 个字段小于 100 的行
string-matching	/Asia/	含有 Asia 的行
compound	$3 < 100 && $4 == "Asia"	第 3 个字段小于 100 并且第 4 个字段含有 Asia 的行
range	NR==10, NR==20	输入的第 10 行到第 20 行.

2.2 动作

34

在一个 模式–动作 语句中, 模式决定动作什么时候执行. 有时候动作会非常简单: 一条单独的打印语句或赋值语句. 在有些时候, 动作有可能是多条语句, 语句之间用换行符或分号分开. 这一小节通过讨论表达式与流程控制语句来开始对动作的描述. 这节结束后, 将会讨论用户自定义函数与输入/输出语句.

动作

动作中的语句可以包括:

expression, 包括常量, 变量, 赋值, 函数调用等等.

```
print expression-list
printf(format, expression-list)
if (expression) statements
if (expression) statements else statements
while (expression) statements
for (expression; expression; expression) statements
for (expression in array) statements
do statements while (expression)
break
continue
```

表达式

我们从表达式开始讨论, 因为表达式是最简单的语句, 大多数其他语句都是由不同类型的表达式组合而成. 主表达式与其他表达式通过运算符组合在一起, 形成一个新的表达式. 主表达式是最原始的构造块: 它们包括常量, 变量, 数组引用, 函数调用, 以及各种内建变量, 例如字段的名字.

我们从常量与变量开始对表达式的讨论, 然后是运算符, 它们可以用来组合表达式. 这些表达式可以分成 5 种类别: 算术, 比较, 逻辑, 条件, 与赋值. 运算符之后, 讨论的是内建算术运算函数与字符串函数, 最后是数组.

常量 (*Constants*). Awk 中只有两种类型的常量: 字符串与数值. 将一个字符序列用一对双引号包围起来就创建了一个字符串常量, 正如 `"Asia"`, 或 `"hello, world"` 或 `""`. 字符串常量可以包含表 2.2 列出的转义序列. 35

一个数值常量可以是一个整数, 就像 `1127`, 或十进制小数, `3.14`, 或者是用科学计数法表示的数: `0.707E-1`. 同一个数的不同表示法都拥有相同的值: `1e6`, `1.00E6`, `10e5`, `0.1e7` 与 `1000000` 都表示同一个数. 所有的数都用浮点格式存储, 浮点数的精度依赖于机器.

变量 (*Variables*). 表达式可以包含若干种类型的变量: 用户定义的, 内建的, 或字段. 用户定义的变量名字由数字, 字母与下划线构成, 但是名字不能以数字开始. 所有的内建变量的名字都是大写字母.

每一个变量都有一个值, 这个值可以是字符串或数值, 或两者都是. 因为变量的类型不需要事先声明, 所以 awk 需要根据上下文环境推断出变量的类型. 当需要时, awk 可以把字符串转化为数值, 或反之. 例如, 在程序

```
$4 == "Asia" { print $1, 100 * $2 }
```

里, `$2` 会被转换成数值, 如果它原来不是数值的话, 同样的道理, 如果 `$1` 与 `$4` 原来不是字符串, 它们会被转换成字符串.

一个未初始化的变量的值是 `""` (空字符串) 与 0.

内建变量 (*Built-In Variables*). 表 2.5 列出了所有的内建变量. 其中一些我们已经见过了, 另一些会在本节或后面的章节里用到. 这些变量可以用在所有的表达式中, 而且可以被用户重置. 每当有一个

新的文件被读取, FILENAME 就会被重新赋值. 每当有一个新的记录被读进来, FNR, NF, NR 就会被重新赋值. 另外, 当 $0 发生改变, 或有新的字段被创建时, NF 就被重置. RLENGTH 与 RSTART 会随着每一次 match 的调用而改变.

表 2.5: 内建变量

变量	意义	默认值
ARGC	命令行参数的个数	-
ARGV	命令行参数数组	-
FILENAME	当前输入文件名	-
FNR	当前输入文件的记录个数	-
FS	控制着输入行的字段分割符	" "
NF	当前记录的字段个数	-
NR	到目前为止读的记录数量	-
OFMT	数值的输出格式	"%.6g"
OFS	输出字段分割符	" "
ORS	输出的记录的分割符	"\n"
RLENGTH	被函数 match 匹配的字符串的长度	-
RS	控制着输入行的记录分割符	"\n"
RSTART	被函数 match 匹配的字符串的开始	
SUBSEP	下标分割符	"\034"

字段变量 (*Field Variables*). 当前输入行的字段从 $1, $2, 一直到 $NF; $0 表示整行. 字段变量与其他变量相比没什么不同 — 它们也可以用在算术或字符串运算中, 也可以被赋值. 于是, 人们可以将 countries 的每一行的第 2 个字段除以 1000, 从而可以用百万平方英里 — 而不是千平方英里 — 来表示面积:

```
{ $2 = $2 / 1000; print }
```

也可以将一个新的字符串赋给字段:

```
BEGIN                    { FS = OFS = "\t" }
$4 == "North America" { $4 = "NA" }
$4 == "South America" { $4 = "SA" }
                         { print }
```

在这个程序里, BEGIN 动作重新设置 FS (FS 控制输入行的字段分割符) 与 OS (OS 控制输出的字段分割符) 为制表符. 第 4 行的 print 语句打印可能被修改过的 $0. 值得注意的是: 当 $0 被 修改或重新赋值 (同样的情况也可以发生在 $1, $2, 等等), NF 就会被重新计算; 同样的道理, 当 $1 — 或 $2 等 — 被修改了, $0 就会被重新构造, 构造的方式是使用 OFS 重新分割字段. 36

字段也可以通过表达式指定. 例如, $(NF-1) 表示当前输入行的倒数第 2 个字段. 表达式两边的括号不能省略: $NF-1 表示最后一个字段减 1 后的值.

如果字段变量引用到了不存在的字段, 例如 `$(NF+1)`, 那么它的值就是初始值 — 空字符串. 可以通过向一个字段变量赋值来创建它. 例如, 下面这个程序创建了第 5 个字段, 该字段包含的值是人口密度:

```
BEGIN { FS = OFS = "\t" }
{ $5 = 1000 * $3 / $2; print }
```

当需要时, 任何突然出现的字段都会被创建, 并且它们的初始值都是空值.

每一行的字段数都可以不同, 但是 awk 的具体实现通常将字段数上限设置为 100.

算术运算符 (*Arithmetic Operators*). Awk 提供了通常的 `+`, `-`, `*`, `/`, `%`, `^` 运算符. 运算符 `%` 计算余数: `x%y` 的值是 `x` 被 `y` 除的余数; 当 `x` 或 `y` 是负数时, `x%y` 的结果依赖于机器. `^` 是指数运算符: `x^y` 表示 x^y. 所有的算术运算都用的是浮点数.

比较运算符 (*Comparison Operators*). 比较表达式指的是那些含有关系运算符, 或含有正则表达式匹配运算符的表达式. 关系运算符包括 `<`, `<=`, `==` (相等), `!=` (不相等), `>=` 与 `>`. 37

表达式

1. 最基本的表达式包括:

   数值与字符串常量, 变量, 字段, 函数调用, 数组元素.

2. 可以把表达式组合起来的运算符包括:

   赋值运算符 `=  +=  -=  *=  /=  %=  ^=`

   条件表达式 `?:`

   逻辑运算符 `||` (OR), `&&` (AND), `!` (NOT)

   匹配运算符 `~` 和 `!~`

   关系运算符 `<  <=  ==  !=  >  >=`

   拼接运算符 (没有显式的拼接运算符)

   算术运算符 `+ - * / % ^`

   单目运算符 `+` 和 `-`

   自增与自减运算符 `++` 和 `--` (包括前缀与后缀)

   括号 (用于分组)

正则表达式的运算符包括 `~` (被匹配) 与 `!~` (不被匹配). 如果比较表达式的判断结果为真, 则它的值是 1, 否则为 0. 所以, 表达式

```
$4 ~ /Asia/
```

的值是 1, 如果当前输入行的第 4 个字段包含 `Asia` 的话; 反之, 如果不包含, 那么它的值就是 0.

逻辑运算符 (*Logical Operators*). 逻辑运算符将多个表达式组合成为逻辑表达式. 如果逻辑表达式的真值为真, 那么它的值就为 1; 如果为假, 值就为 0. 在对逻辑表达式求值时, 具有非零值或非空值的操作数被当作真; 相应的, 值为零或空的操作数被当作假. 操作数之间被 `&&` 或 `||` 分开, 求值是从左至右进行的, 当整个逻辑表达式的值可以确定时, 求值就停下来. 这意味着在表达式

$$expr_1 \ \&\& \ expr_2$$

中, 如果 $expr_1$ 的值为假, 那么 $expr_2$ 就不会被求值, 而在表达式

$$expr_3 \texttt{ || } expr_4$$

中, 如果 $expr_3$ 的值为真, $expr_4$ 就不会被求值.

在 && 与 || 之后可以插入换行符.

条件表达式 (*Conditional Expressions*). 一个条件表达式具有形式:

$$expr_1 \texttt{ ? } expr_2 \texttt{ : } expr_3$$

首先, $expr_1$ 被求值. 如果值为真, 也就是值非零或非空, 那么整个条件表达式的值就会是 $expr_2$ 的值; 否则, 如果 $expr_1$ 的值为假, 那么条件表达式的值就会是 $expr_3$. $expr_2$ 与 $expr_3$ 只有其中一个会被求值. 38

下面这个程序利用条件表达式打印 $1 的倒数, 如果 $1 的值为 0, 那就打印一条警告:

```
{ print ($1 != 0 ? 1/$1 : "$1 is zero, line " NR) }
```

赋值运算符 (*Assignment Operators*). 在赋值表达式中可以使用 7 种赋值运算符. 最简单的赋值表达式是

$$var = expr$$

在这个表达式中, *var* 是一个变量或字段的名字, 然后 *expr* 是一个任意的表达式. 例如, 为了计算总人口与亚洲的国家数量, 我们可以写

```
$4 == "Asia"    { pop = pop + $3; n = n + 1 }
END             { print "Total population of the", n,
                        "Asian countries is", pop, "million."
                }
```

将 countries 作为输入数据, 输出将是

```
Total population of the 4 Asian countries is 2173 million.
```

第一个动作含有两个赋值语句, 第一个累积人口, 第二个计算国家的数量. 变量没有被显式地初始化, 但程序仍然按照期望得那样运行, 这是因为每个变量都会默认初始化为空字符串 "" 或数值 0.

在下面这个程序里, 我们仍然利用了默认的初始化行为, 程序的功能是寻找人口最多的国家:

```
$3 > maxpop { maxpop = $3; country = $1 }
END         { print "country with largest population:",
                country, maxpop
            }
```

注意, 只有当至少有一行的 $3 是正数时, 程序才是正确的.

另外 6 个赋值运算符是 +=, -=, *=, /=, %=, 以及 ^=. 它们的意义都是类似的: $v\ op = e$ 等价于 $v = v\ op\ e$, 但是 v 只被求值一次. 赋值表达式:

```
pop = pop + $3
```

可以用 += 写成更加紧凑的形式:

```
pop += $3
```

紧凑版与冗长版具有相同的作用 — 左值加上右值, 并赋给左值 — 但是 += 更简短, 运行起来也更快. 再看另外一个例子:

```
{ $2 /= 1000; print }
```

这个程序将第 2 个字段除以 1000, 再打印整行.

一个赋值语句是一个表达式; 整个表达式的值是左边的变量的新值. 于是, 赋值语句可以放在任意表达式内部. 在复合赋值语句

```
FS = OFS = "\t"
```

中, 字段分割符与输出字段分割符都被设置为制表符. 赋值表达式在条件判断中也很常见, 例如:

```
if ((n = length($0)) > 0) ...
```

自增与自减 (*Increment and Decrement Operators*). 赋值语句:

```
n = n + 1
```

通常写成 ++n 或 n++, 这里用到了一元自增运算符, 作用是给变量加 1. 前缀形式 ++n 在传递 n 的值之前为 n 加 1; 后缀形式 n++ 在传递 n 的值之后, 为 n 加 1. 当 ++ 应用到赋值表达式时, 这会造成一些不一样的地方. 如果 n 的初始值为 1, 赋值语句 i = ++n 为 n 加 1, 并将新值 2 赋给 i, 而 i = n++ 同样为 n 加 1, 但将旧值 1 赋给 i. 如果仅仅是给 n 加 1, 那么 n++ 与 ++n 没什么区别. 前缀与后缀自减运算符 -- 给变量减 1, 除此之外与 ++ 相同.

内建算术函数 (*Built-In Arithmetic Functions*). 内建算术函数在表 2.6 中列出. 这些函数都可以作为主要表达式使用. 在表格中, x 与 y 都是任意的表达式.

表 2.6: 内建算术函数

函数	返回值
atan2(y,x)	y/x 的反正切值, 定义域在 $-\pi$ 到 π 之间
cos(x)	x 的余弦值, x 以弧度为单位
exp(x)	x 的指数函数, e^x
int(x)	x 的整数部分; 当 x 大于 0 时, 向 0 取整
log(x)	x 的自然对数 (以 e 为底)
rand()	返回一个随机数 r, $0 \le r < 1$
sin(x)	x 的正弦值, x 以弧度为单位.
sqrt(x)	x 的方根
srand(x)	x 是 rand() 新的随机数种子

可以用这些函数来得到一些有用的常量: atan2(0,-1) 得到 π, exp(1) 返回 e — 自然对数的底. 为了计算 x 的以 10 为底的对数, 我们可以用 log(x)/log(10) 来实现.

rand() 返回一个大于等于 0, 小于 1 的伪随机浮点数. 调用 srand(x) 可以使随机数生成器的开始点从 x 开始. 如果没有调用 srand, 每次程序运行时, rand 都从同一个值开始.

赋值语句

```
randint = int(n * rand()) + 1
```

将 randint 的值设置为 1 到 n 之间的一个整数, 包括 1 与 n. 这里我们用到函数 int 来丢弃返回值的小数部分. 赋值语句

```
x = int(x + 0.5)
```

将正数 x 四舍五入为最接近它的整数.

字符串运算符 (*String Operators*). Awk 中只有一种字符运算符 — 拼接. 拼接并没有显式的运算符, 通过陆续写出字符串常量, 变量, 数组元素, 函数返回值, 与其他表达式, 就可以创建一个字符串. 程序

```
{ print NR ":" $0 }
```

会将每一个输入行打印出来, 并在前面冠上行号, 以及一个冒号. 行号, 冒号与输入行之间没有空格. 数值 NR 会被自动转换成字符串 (如果必要的话, $0 也会做相同的转换); 然后三个字符串拼接在一起并打印出来.

作为正则表达式的字符串 (*Strings as Regular Expressions*). 到目前为止, 在所有的匹配表达式中, ~ 与 !~ 右边的操作数都是都是一个被斜杆包围起来的正则表达式. 实际上, 任意一个表达式都可以用作匹配运算符的右操作数. Awk 对这些表达式求值, 如果必要的话将这些值转换成字符串, 再将这些字符串解释成正则表达式. 例如, 程序

```
BEGIN { digits = "^[0-9]+$" }
$2 ~ digits
```

会将那些第 2 个字段有且仅有数字的行打印出来.

因为字符串可以被拼接起来, 于是一个正则表达式可以由多个子部分构成. 下面这个程序将那些具有有效浮点数的行打印出来:

```
BEGIN {
    sign = "[+-]?"
    decimal = "[0-9]+[.]?[0-9]*"
    fraction = "[.][0-9]+"
    exponent = "([eE]" sign "[0-9]+)?"
    number = "^" sign "(" decimal "|" fraction ")" exponent "$"
}
$0 ~ number
```

在一个匹配表达中, 一个被双引号包围的字符串, 可以与一个被斜杆包围的正则 达式互换着使用, 例如 "^[0-9]+$" 与 /^[0-9]+$/. 然而有一个例外, 如果被双引号包围的字符串想要匹配一个正则表达式元字符的字面值, 当写成被斜杆包围的形式时, 就要在反斜杆的前面再加一个反斜杆来保护它. 于是 41

```
$0 ~ /(\+|-)[0-9]+/
```

与

```
$0 ~ "(\\+|-)[0-9]+"
```

是等价的.

这种性质可能看起来非常晦涩难懂, 但是, 被双引号包围的字符串被 awk 解析时, 起保护作用的反斜线就会被移除, 这样想的话应该就容易理解多了. 如果一个正则表达式的元字符需要一个反斜杆来暂时移除它的特殊意义, 那么在字符串中, 这个反斜杆就需要一个额外的反斜杆来保住它自己. 如果一个匹配运算符的右操作数是一个变量或字段, 正如

```
x ~ $1
```

那么第 1 个字段中的反斜杆就不需要一层额外的保护, 因为在数据中反斜杆并没有特殊的意义.

测试你对正则表达式交互性的理解是很容易的: 程序

```
$1 ~ $2
```

可以让你输入一个字符串与一个正则表达式; 如果字符串匹配正则表达式的话, 它就会回射该行.

内建字符串函数 (*Built-In String Functions*). 表 2.7 列出了 awk 的内建字符串函数. 在这张表中, r 表示一个正则表达式 (或者是一个字符串, 或者是被一对斜杆包围了的). s 与 t 是字符串表达式, n 与 p 是整数.

表 2.7: 内建字符串函数

函数	描述
gsub(r,s)	将 $0 中所有出现的 r 替换为 s, 返回替换发生的次数.
gsub(r,s,t)	将字符串 t 中所有出现的 r 替换为 s, 返回替换发生的次数
index(s,t)	返回字符串 t 在 s 中第一次出现的位置, 如果 t 没有出现的话, 返回 0.
length(s)	返回 s 包含的字符个数
match(s,r)	测试 s 是否包含能被 r 匹配的子串, 返回子串的起始位置或 0; 设置 RSTART 与 RLENGTH
split(s,a)	用 FS 将 s 分割到数组 a 中, 返回字段的个数
split(s,a,fs)	用 fs 分割 s 到数组 a 中, 返回字段的个数
sprintf(*fmt*,*expr-list*)	根据格式字符串 *fmt* 返回格式化后的 *expr-list*
sub(r,s)	将 $0 的最左最长的, 能被 r 匹配的子字符串替换为 s, 返回替换发生的次数.
sub(r,s,t)	把 t 的最左最长的, 能被 r 匹配的子字符串替换为 s, 返回替换发生的次数.
substr(s,p)	返回 s 中从位置 p 开始的后缀.
substr(s,p,n)	返回 s 中从位置 p 开始的, 长度为 n 的子字符串.

函数 index(s,t) 返回 t 在 s 中第一次出现的位置; 如果 t 没有在 s 中出现, 那就返回 0. 字符串第一个字符的位置是 1:

```
index("banana", "an")
```

返回 2.

函数 match(*s*,*r*) 返回 *s* 中最左最长的, 能被 *r* 匹配的子串, 返回值是子串在 *s* 中的开始位置, 如果没有找到相匹配的子串, 那就返回 0. 这个函数还会将内建变量 RSTART 置为子串的起始位置, 将 RLENGTH 置为子串的长度.

函数 split(*s*,*a*,*fs*) 根据分割符 *fs* 将字符串 *s* 分割成一个个子串, 并存到数组 *a* 中. 这个函数会在数组之后讨论, 也就是在这一节的末尾. 42

sprintf(*format*,*expr*$_1$,*expr*$_2$,...,*expr*$_n$) 返回 (不打印) 一个字符串, 这个字符串包含格式化了的 *expr*$_1$, *expr*$_2$, ..., *expr*$_n$, 格式化的依据与 printf 的格式说明符相同, printf 的格式说明符就是 *format*. 于是, 语句

```
x = sprintf("%10s %6d", $1, $2)
```

把一个字符串赋值给 x, 这个字符串通过把 $1 格式化成 10 个字符宽度的字符串, 把 $2 格式化成至少 6 个字符宽度的十进制整数来生成. 2.4 节包含了一个完整的, 关于格式转换字符的描述.

自从 Unix 文件编辑器 ed 的替换命令出现之后, sub 与 gsub 就已经被模式化了. sub(*r*,*s*,*t*) 首先在目标字符串 *t* 中找到能被 *r* 匹配的最左最长子串, 再将这个子串替换为 *s*. 在 ed 中, "最左最长" 意味着先找到最左边的匹配, 然后尽可能的延长匹配的长度.

如果目标字符串是 banana, 那么 anan 就是正则表达式 (an)+ 的最左最长匹配. 相反, (an)* 的最左最长匹配是 b 左边的空字符串.

sub 返回替换发生的次数. sub(*r*,*s*) 等价于 gsub(*r*,*s*,$0).

gsub 是类似的, 但是它会连续地替换最左最长且不重叠的子串, 而不是只替换子串的第一次出现; 函数返回值仍然是替换发生的次数 (gsub 的 g 是 "global" 的缩写, 表示替换是全局的). 例如, 程序 43

```
{ gsub(/USA/, "United States"); print }
```

将输入行中的每一个 "USA" 替换为 "United States" (在这样的例子里, 当 $0 发生改变时, 字段与 NF 也会跟着发生改变). 程序 [①]

```
gsub(/ana/, "anda", "banana")
```

将 banana 变为 bandana, 匹配是不重叠的.

对一个由 sub 或 gsub 执行的替换来说, 字符 & 在 *s* 中任意一次出现都会被替换为被 *r* 匹配的子串, 于是

```
gsub(/a/, "aba", "banana")
```

将 banana 变为 babanabanaba; 同样的效果也可用

```
gsub(/a/, "&b&", "banana")
```

来完成. 在 & 的左边加上一个反斜杆就可以关闭它在替换字符串中的特殊意义.

substr(*s*,*p*) 返回 *s* 从位置 *p* 开始的后缀. 如果使用了 substr(*s*,*p*), 那么只会返回后缀的前 *n* 个字符; 如果后缀的长度小于 *n*, 那么就会返回整个后缀. 例如, 我们可以将国家名缩写为名字的前三个字母, 程序

```
{ $1 = substr($1, 1, 3); print $0 }
```

[①] 在 mawk-1.3.3 中, sub 与 gsub 的第 3 个参数 *t* 必须是变量. — 译者

输出

```
USS 8649 275 Asia
Can 3852 25 North America
Chi 3705 1032 Asia
USA 3615 237 North America
Bra 3286 134 South America
Ind 1267 746 Asia
Mex 762 78 North America
Fra 211 55 Europe
Jap 144 120 Asia
Ger 96 61 Europe
Eng 94 56 Europe
```

修改 $1 使得 awk 重新计算 $0, 于是字段之间的分隔符就变成了空格 (OFS 的默认值), 而不是制表符.

只要将字符串陆续写出来就可以将它们拼接在一起. 例如, 对于 countries, 程序

```
    { s = s substr($1, 1, 3) " " }
END { print s}
```

打印 44

```
USS Can Chi USA Bra Ind Mex Fra Jap Ger Eng
```

程序每一次只为 s 构造一小段. (如果你很在意 s 的末尾的空格, 在 END 中使用

```
print substr(s, 1, length(s)-1)
```

替换 print s)

数值还是字符串 (*Number or String*). 表达式的值可以自动从数值转换为字符串, 或反之, 具体取决于该值将用于什么运算. 算术表达式, 例如

```
pop + $3
```

这个表达式要求 pop 与 $3 都必须是数值, 如果它们原来不是数值的话, 就会被强制转换成数值. 类似的, 赋值语句

```
pop += $3
```

也要求 pop 与 $3 是数值. 字符串表达式, 就像

```
$1 $2
```

要求 $1 与 $2 必须是字符串才能进行拼接, 所以如果必要的话, 它们就会被强制转换成字符串.

同样的运算符有时既可以用在数值上, 也可以用在字符串, 对于这种情况有一些特殊的规则. 在赋值语句 $v = e$ 中, 赋值语句本身与变量 v 都会对表达式 e 的类型有所要求. 比较表达式, 就像

```
x == y
```

如果两个操作数都是数值, 那么比较就按照数值进行; 否则, 数值类型的操作数被强制转换成字符串, 然后再按字符串的方式进行比较.

现在让我们来查看一下这条规则具体应用时会产生什么影响, 语句

```
$1 == $2
```

涉及到字段. 在这个语句里, 比较的类型取决于这两个字段是否包含数值或字符串, 而这只有到程序运行时才可以知道; 对于不同的行, 比较的类型可能是不一样的. 当 awk 在运行状态下创建一个字段时, 会自动将它的类型设置为字符串; 另外, 如果字段包含一个机器可识别的数, 它也会给这个字段设置一个数值类型.

例如, 比较 `$1 == $2` 将会按照数值进行, 并且比较结果为真, 如果 `$1` 与 `$2` 的值是下面任意一种的话,

```
1    1.0    +1    1e0    0.1e+1    10E-1    001
```

45

这些值都是数值 1 的不同表示方法. 然而, 这个表达式也可以是字符串比较, 于是, 该表达式对下面每对值都会比较失败:

```
0           (null)
0.0         (null)
0           0a
1e500       1.0e500
```

在前三对值中, 第二个字段都不是一个有效的数值. 最后一对值也会按照字符串来比较, 这是因为如果将它们转换成数值的话, 其值大得无法在机器上表示.

输出语句

```
print $1
```

打印第 1 个字段的字符串值; 于是, 输出与输入是一样的.

未初始化的变量默认初始值为数值 0 或空字符串 `""`. 不存在的, 或显式为空的字段具有字符串值 `""`, 它们不是数值, 但是当强制转换为数值时, 将会是 0. 我们将在本节的末尾看到, 数组的下标是字符串.

有两种惯用语法可以将表达式从一种类型转换成另一种类型:

number `""`	将空字符串拼接到 *number* 可以将它强制转换成字符串;
string `+ 0`	给字符串加上零可以把它强制转换成数值.

于是, 为了强制让两个字段之间的比较按照字符串来进行, 我们可以把其中一个字段强制转换成字符串:

```
$1 + 0 == $2
```

为了让比较强制按照数值类型来进行, 我们需要将两个字段都转换成数值:

```
$1 + 0 == $2 + 0
```

无论字段包含什么内容, 这个方式总能奏效.

字符串的数值类型的值, 等于字符串中最长的看起来像数值的前缀的值. 于是

```
BEGIN { print "1E2"+0, "12E"+0, "E12"+0, "1X2Y3"+0 }
```

输出

```
100 12 0 1
```

数值的字符串形式需要根据 OFMT 转换后才会输出. 为了拼接, 比较, 与创建数组下标而需要把数值转换成字符串时, OFMT 也会影响转换的过程. OFMT 的默认值是 "%.6g". 于是

```
BEGIN { print 1E2 "", 12E-2 "", E12 "", 1.23456789 "" }
```

输出

```
100 0.12  1.23457
```

46

向 OFMT 赋予新值就可以修改它的默认值. 比如将 OFMT 改为 "%.2f", 那么在打印数值, 以及通过强制转换而得到的数值之间进行比较时, 小数点后将会保留两位小数.

表 2.8: 表达式运算符

操作	运算符	例子	例子的含义
赋值	= += -= *= /= %= ^=	x *= 2	x = x * 2
条件表达式	?:	x ? y : z	若 x 为真, 则 y, 否则 z
逻辑或	||	x || y	若 x 或 y 为真, 则为 1, 否则为 0
逻辑与	&&	x && y	若 x 与 y 都为真, 则为 1, 否则为 0
数组成员匹配	in	i in a	如果 a[i] 存在, 则为 1, 否则为 0
关系运算	< <= == != >= >	x == y	如果 x 等于 y, 则为 1, 否则为 0
拼接		"a" "bc"	"abc"; 不存在显式的拼接运算符
减法, 加法	+ -	x + y	x 与 y 的和
乘法, 除法, 取模	* / %	x % y	x 除以 y 的余数
单目加, 单目减	+ -	-x	x 的相反数
逻辑非	!	!$1	若 $1 为空或为 0, 则为 1, 否则为 0
自增, 自减	++ --	++x, x++	为 x 加 1
字段	$	$i+1	1 加上第 i 个字段的值
组合	( )	($i)++	给第 i 个字段的值加 1

运算符的总结 (*Summary of Operators*). 可以出现在表达式中的运算符全部列在了表 2.8 中. 将这些运算符应用到常量, 变量, 字段, 数组元素, 函数, 和其他表达式中, 就可以构造出一个表达式.

运算符按照优先级的升序排列. 优先级高的运算符优先求值; 举例来说, 乘法运算在加法运算之前求值. 所有的运算符都是左结合的, 除了赋值运算符, 条件运算符, 指数运算, 它们都是右结合的. 左结合性意味着相同优先级的运算符按照从左到右的顺序进行运算; 于是 3-2-1 是指 (3-2)-1, 而不是 3-(2-1).

因为没有显式的拼接运算符, 所以比较明智的做法是, 在拼接运算中, 将涉及到其他运算的 表达式 47
用括号括起来. 考虑下面这个程序

```
$1 < 0 { print "abs($1) = " -$1 }
```

跟在 print 后面的表达式看起来像是拼接, 实际上是一个减法运算. 程序

```
$1 < 0 { print "abs($1) = " (-$1) }
```

与

```
$1 < 0 { print "abs($1) =", -$1 }
```

都可以得到我们期望中的效果.

流程控制语句

Awk 提供花括号用于语句组合, if-else 用于决策, while, for, do 语句用于循环, 所有这些都来源于 C 语言.

一条单独的语句总是可以被替换为一个被花括号包围起来的语句列表, 列表中的语句用换行符或分号分开, 换行符可以出现在任何左花括号之后, 也可以出现在任何右花括号之后.

if-else 具有形式:

```
if (expression)
    statements1
else
    statements2
```

else $statements_2$ 是可选的. 右括号, $statements_1$, 和关键词 else 后面的换行符是可选的. 如果 else 与 $statements_1$ 出现在同一行, 并且 $statements_1$ 是一条单独的语句, 那么在 $statements_1$ 的末尾必须使用分号来终止语句.

在一个 if-else 语句中, 文本 *expression* 先被求值, 如果 *expression* 为真 (也就是非空或非零), 那么 $statements_1$ 就会执行. 如果 *expression* 为假 (也就是空或零), 如果 else $statements_2$ 存在, 那它就会执行.

为了避免歧义, 我们规定, else 与最近一个未匹配的 if 匹配. 举例来说, 语句

```
if (e1) if (e2) s=1; else s=2
```

中的 else 与第 2 个 if 匹配. (s=1 后面的分号是必须的, 因为 else 与 s=1 出现在同一行)

当条件为真时, while 就会重复执行一条语句:

流程控制语句

1. {*statements*}

   语句组

2. if (*expression*)*statements*

   如果 *expression* 为真, 执行 *statements*

3. if (*expression*) $statements_1$ else $statements_2$

   如果 *expression* 为真, 执行 $statements_1$, 否则执行 $statements_2$

4. `while (`*expression*`)` *statements*

   如果 *expression* 为真, 执行 *statements*; 然后重复前面的过程

5. `for (`*expression*$_1$`;`*expression*$_2$`;`*expression*$_3$ `)` *statements*

   等价于 *expression*$_1$`; while (`*expression*$_2$`) { `*statements*`; `*expression*$_3$`}`

6. `for (`*variable* `in` *array*`)` *statements*

   轮流地将 *variable* 设置为 *array* 的每一个下标, 并执行 *statements*

7. `do` *statements* `while (`*expression*`)`

   执行 *statements*; 如果 *expression* 为真就重复

8. `break`

   马上离开最内层的, 包围 `break` 的 `while`, `for` 或 `do`

9. `continue` 开始最内层的, 包围 `continue` 的 `while`, `for`, 或 `do` 的下一次循环

10. `next`

    开始输入主循环的下一次迭代

11. `exit`

12. `exit` *expression*

    马上执行 `END` 动作; 如果已经在 `END` 动作内, 那就退出程序. 将 *expression* 作为程序的退出状态返回.

48

```
while (expression)
    statements
```

右括号后面的换行符是可选的. 在这个循环语句里, *expression* 被求值; 如果它的值为真, *statements* 就会执行, 然后 *expression* 再被求值. 只要 *expression* 为真, 循环就会一直持续下去. 举例来说, 下面这个程序打印所有的输入字段, 每行一个:

```
{   i = 1
    while (i <= NF) {
        print $1
        i++
    }
}
```

当 `i` 到达 `NF+1` 时, 循环停止, 这也是循环结束时 `i` 的值.

`for` 语句是 `while` 的更加一般的形式:

```
for (expression1;expression2;expression3)
    statements
```

49

右括号后面的换行符是可选的. for 等效于

```
expression1
while (expression2) {
    statements
    expression3
}
```

所以

```
{ for (i = 1; i <= NF; i++)
    print $i
}
```

所做的工作, 与前面的用 while 循环遍历所有字段的例子是一样的. 在 for 语句中, 三个表达式都是可选的. 如果 *expression*$_2$ 被省略了, 那么条件就被当作是永真式, 于是 for(;;) 将会是无限循环.

for 语句的一个替代版本可用于遍历数组下标, 这在数组那一节讨论.

do 语句具有形式

```
do
    statements
while (expression)
```

关键字 do 与 *statements* 后面的换行符是可选的. 如果 while 与 *statements* 出现在了同一行, 并且 *statements* 是一条单独的语句, 那么就必须使用分号来终止 *statements*. do 循环执行 *statements* 一次, 然后只要 *expression* 为真, 就重复执行 *statements*. do 循环与 while , for 循环相比有很大的不同: 它的条件测试在循环体的底部, 而不是顶部, 所以循环体至少会执行一次.

有两种语句可以影响循环的运行: break 和 continue. break 会导致控制流马上从包围着它的循环内退出, 循环可以是 while, for , 或 do. continue 导致下一次迭代开始; 它使得执行流马上进入 while 与 do 的测试表达式, 或 for 的 *expression*$_3$.

next 与 exit 控制用于读取输入行的外层循环. next 使 awk 抓取下一个输入行, 然后从第一个模式–动作 语句开始匹配测试. 在 END 动作里, exit 会导致程序终止, 在其他动作里, exit 会使得程序表现得就好像所有输入都读完了; 不再有输入会被读取, 并执行 END 动作 (如果有 END 的话).

如果 exit 语句包含一个表达式

```
exit expression
```

50

它使得 awk 将 *expression* 的值作为程序的退出状态返回, 除非返回值被随后的错误或 exit 覆盖. 如果没有 *expression*, 退出状态就会是 0. 在某些操作系统中, 包括 Unix, 调用 awk 的程序可能会检查 awk 的退出状态.

空语句

单独一个分号表示一个空语句. 在下面这个程序里, for 的循环体是一个空语句.

```
BEGIN { FS = "\t" }
      { for (i = 1; i <= NF && $i != ""; i++)
          ;
        if (i <= NF)
            print
      }
```

这个程序打印所有的, 包含空字段的行.

数组

Awk 提供了一维数组, 用于存放字符串与数值. 数组与数组元素都不需要事先声明, 也不需要说明数组中有多少个元素. 就像变量一样, 当被提及时, 数组元素就会被创建, 数组元素的默认初始值为 0 或空字符串 "".

作为一个简单的例子, 语句

```
x[NR] = $0
```

将当前输入行赋值给数组 x 的元素 NR. 实际上, 将整个输入行读入到一个数组中, 然后再按照某种方便的顺序去处理它们, 这样做是很容易的, 虽然会有点慢. 举例来说, 下面这个程序是 1.7 节的逆序打印程序的变形:

```
    { x[NR] = $0 }
END { for (i = NR; i > 0; i--) print x[i] }
```

第一个动作仅仅是将每个输入行存放到数组 x 中, 使用行号作为下标; 真正的工作在 END 语句中完成.

Awk 的数组与大多数其他语言最大的不同点是, 数组元素的下标是字符串. 这个特性使得 awk 具有了一种能力, 这种能力类似于 SNOBOL4 表格的关联内存, 也由于这个原因, awk 的数组称为 关联数组 (*associative arrays*).

下面这个程序将 Asia 与 Europe 的人口数量累加到数组 pop 中. END 动作打印这两个洲的总人口.

```
/Asia/      { pop["Asia"] += $3 }
/Europe/    { pop["Europe"] += $3 }
END         { print "Asian population is",
                pop["Asia"], "million."
              print "European population is",
                pop["Europe"], "million."
            }
```

51

countries 作为输入, 输出是

```
Asian population is 2173 million.
European population is 172 million.
```

应该注意的是, 数组下标是字符串常量 "Asia" 与 "Europe". 如果我们将 pop["Asia"] 写成 pop[Asia], 后者使用变量 Asia 的值作为下标, 又因为变量是未初始化过的, 于是人口数量将会累加到 pop[""] 中.

这个例子其实并不需要关联数组, 因为这里只有两个元素, 并且都显式给出了名字. 假设我们的任务是计算每个大洲的总人口, 对于这种聚合问题, 使用关联数组是非常方便的做法. 数组下标可以是任意的表达式, 所以

```
pop[$4] += $3
```

使用当前输入行的第 4 个字段的字符串值来索引数组 pop, 并使用该项来累积第 3 个字段的值:

```
BEGIN   { FS = "\t" }
        { pop[$4] += $3 }
END     { for (name in pop)
            print name, pop[name]
        }
```

数组 pop 的下标是大洲的名字; 每一个元素的值是大洲的总人口. 无论大洲的数目有多少个, 这个程序都可以正常地工作; 文件 countries 的输出是

```
Europe 172
Asia 2173
South America 134
North America 340
```

上面的程序使用 for 语句的另一种形式来遍历数组下标:

for (*variable* in *array*)
    statements

这个循环将 *variable* 轮流地设置为数组的每一个下标, 并执行 *statements*. 下标的排列顺序依赖于实现. 如果 *statements* 往数组里加入了新元素, 那么结果是不可预知的.

为了判断某个特定的下标是否出现在数组中, 你可以这样写:

subscript in *A*

如果 *A*[*subscript*] 已经存在, 那么这个表达式 的值为 1, 否则为 0. 所以, 为了测试 Africa 是否是数组 pop 的一个下标, 你可以这样写: 52

```
if ("Africa" in pop) ...
```

这个条件判断可以在不产生副作用的情况下执行测试, 副作用指的是创建 pop["Africa"], 如果你写成下面这种样子, 就会有创建新元素的副作用:

```
if (pop["Africa"] != "") ...
```

delete 语句 (*The delete Statement*). 一个数组元素可以通过

delete *array*[*subscript*]

来删除. 例如, 下面这个循环删除数组 pop 中所有的元素:

```
for (i in pop)
    delete pop[i]
```

split 函数 (*The split Function*). 函数 split(*str*,*arr*,*fs*) 将字符串 *str* 切割成一个个字段, 并存储到数组 *arr* 中. 字段的个数作为函数的返回值返回. 第 3 个参数 *fs* 的字符串值作为字段的分割符. 如果第 3 个参数被忽略, 就使用 FS. 不管是哪一种情况, 分割字段的规则等同于输入行的字段分割 (见 2.5 节). 函数

```
split("7/4/76", arr, "/")
```

将字符串 7/4/76 分割成 3 个字段, 使用 / 作为分割符; 它将 7 存储在 arr["1"], 将 4 存储在 arr["2"], 将 76 存储在 arr["3"].

字符串是多功能的数组下标, 但是, 将数值类型的下标作为字符串来索引数组时, 有时候会出现有悖于常理的事情. 因为 1 的字符串值与 "1" 是相同的, 所以 arr[1] 与 arr["1"] 是同一个元素. 但是请注意, 01 的字符串值与 1 的字符串值并不相同, 并且字符串 10 排在字符串 2 之前.

多维数组 (*Multidimensional Arrays*). Awk 不直接支持多维数组, 但是它利用一维数组来近似模拟多维数组. 虽然你可以写出像 i, j 或 s, p, q, r 这样的多维数组下标, 实际上 awk 会将它们都拼接起来 (下标之间用一个分隔符分开), 合成一个单独的下标. 举例来说,

```
for (i = 1; i <= 10; i++)
    for (j = 1; j <= 10; j++)
        arr[i, j] = 0
```

创建了一个具有 100 个元素的数组, 下标具有形式 1,1, 1,2 等等. 在 awk 内部, 下标其实是以字符串 53
的形式存储的, 字符串具有形式 1 SUBSEP 1, 1 SUBSEP 2 等等. 内建变量 SUBSEP 用于分隔下标的各个构成成分, 它的默认值是 "\034", 而不是逗号, 之所以使用 "\034" 是因为这个字符不太可能出现在通常的文本中.

为了测试一个多维下标是否是某个数组的成员, 我们需要将下标用括号括起来, 例如

```
if ((i,j) in arr) ...
```

为了遍历一个这样的数组, 你可以这样写:

```
for (k in arr) ...
```

如果需要单独地访问下标的某个成分, 你可以使用 split(k, x, SUBSEP).

数组的元素不能再是数组.

2.3 用户自定义函数

除了内建函数, awk 还可以包含用户定义的函数. 定义函数的语句具有形式:

```
function  name(patameter-list) {
    statements
}
```

一个函数定义可以出现在任何 模式–动作 语句可以出现的地方. 于是, awk 程序的通常形式就变成了一系列的, 由换行符或分号分开的 模式–动作 语句与函数定义.

在一个函数定义中, 函数体左花括号后的换行符, 与右花括号前的换行符都是可选的, 参数列表由一系列的由逗号分开的变量名构成; 在函数体内部, 这些变量引用的是函数被调用时传递给它的参数.

函数体可能包含一个 `return` 语句, 用于将程序执行流返回至调用函数 (返回的时候可以带有一个值). 它具有形式

```
return expression
```

expression 与 `return` 语句都是可选的, 如果没有为 `return` 语句提供表达式, 或者最后一句执行的语句不是 `return`, 那么返回值就是未定义的.

举例来说, 这个函数计算参数的最大值:

```
function max(m, n) {
    return m > n ? m : n
}
```

变量 `m` 与 `n` 属于函数 `max`, 它们与程序中其他同名的变量没有任何关联. 54

用户定义的函数可以在任何 模式–动作 语句的任何表达式中使用, 也可以在出现在任何函数体内. 每一个对函数的使用都叫做一个 调用 (*call*). 如果一个用户定义的函数在函数体内调用了它自身, 我们就说这个函数是 递归 (*recursive*) 的.

举例来说, `max` 函数可以像这样调用:

```
{ print max($1, max($2, $3)) } # print maxinum of $1, $2, $3
function max(m, n) {
    return m > n ? m : n
}
```

调用函数时, 函数名与左括号之间不能留有空白.

一个带有参数 `$1` 的函数被调用时 (`$1` 只是一个普通的变量), 函数接收到的参数是变量的值的一份拷贝, 所以函数操作的是变量的拷贝, 而不是变量本身. 这意味着函数不会对函数体外部的变量的值产生影响. (用行话来说, 这样的变量叫做 “标量”, “按值传递” 给函数) 然而, 当数组作为函数的参数时, 函数接收到的参数就不是数组的拷贝, 所以函数可以改变数组的元素, 或往数组增添新的值 (这叫作 “按引用传递”). 函数名不能当作参数使用.

重申一遍, 在函数体内部, 参数是局部变量 — 它们只在函数执行时才存在, 而且它们与程序中其他同名的变量没有任何关联. 但是, 其他所有的变量都是全局的 (*all other variables are global*); 如果函数体内的某个变量没有出现在参数列表中, 那么整个程序范围内都可以访问该变量.

这意味着如果函数想要拥有私有的局部变量, 唯一的方法是将该变量包含在参数列表的末尾. 参数列表中没有实际参数对应的参数都将作为局部变量使用, 初始值为空值. 虽然这种设计不是非常优雅, 但至少为语言提供了必要的能力. 我们在参数与局部变量之间多放置几个空格, 以区分它们.

2.4 输出

`print` 与 `printf` 语句可以用来产生输出. `print` 用于产生简单的输出; `printf` 用于产生格式化的输出. 来自 `print` 与 `printf` 的输出可以被重定向到文件, 管道与终端. 这两个语句可以混合使用,

输出按照它们产生的顺序出现. 55

输出语句

1. `print`

   将 `$0` 打印到标准输出

2. `print` *expression*`,` *expression*`, ...`

   打印各个 *expression*, *expression* 之间由 `OFS` 分开, 由 `ORS` 终止

3. `print` *expression*`,`*expression*`, ... >` *filename*

   输出至文件 *filename*

4. `print` *expression*`,`*expression*`, ... >>` *filename*

   累加输出到文件 *filename*, 不覆盖之前的内容

5. `print` *expression*`,`*expression*`, ... |` *command*

   输出作为命令 *command* 标准输入

6. `printf(`*format*`,`*expression*`,`*expression*`, ...)`

7. `printf(`*format*`,`*expression*`,`*expression*`, ...) >` *filename*

8. `printf(`*format*`,`*expression*`,`*expression*`, ...) >>` *filename*

9. `printf(`*format*`,`*expression*`,`*expression*`, ...) |` *command*

   `printf` 类似于 `print`, 但是第 1 个参数规定了输出的格式

10. `close(`*filename*`), close(` *command*`)`

    断开 `print` 与 *filename* (或 *command*) 之间的连接

11. `system(`*command*`)`

    执行 *command*; 函数的返回值是 *command* 的退出状态

    `printf` 的参数列表不需要被包围在一对括号中. 但是如果 `print` 或 `printf` 的参数列表中, 有一个表达式含有关系运算符, 那么或者表达式, 或者参数列表, 必须用一对括号包围. 在非 Unix 系统上可能不提供管道与 `system`.

print 语句

`print` 语句具有形式:

`print` $expression_1$`,` $expression_2$`,...,`$expression_n$
`print (`$expression_1$`,` $expression_2$`,..., `$expression_n$`)`

两种形式都会把表达式的字符串值打印出现, 各个表达式的字符串值之间用输出字段分隔符分开, 最后跟着输出记录分隔符. 语句

```
print
```

是

```
print $0
```

的缩写形式. 为了打印空白行 (即只含有换行符的行), 可以这样写:

```
print ""
```

print 的第 2 种形式将参数列表包围在一对括号当中, 正如

```
print($1 ":", $2)
```

56

两种形式的 print 都可以产生相同的输出, 但是, 正如我们将会看到的那样, 当参数含有关系运算符时, 就必须使用括号.

输出分隔符

输出字段分割符与输出记录分隔符存储在内建变量 OFS 与 ORS 中. 初始情况下, OFS 与 ORS 分别被设置成一个空格符与一个换行符, 但它们的值可以在任何时候改变. 举例来说, 下面这个程序打印每一行的第 1 与第 2 个字段, 字段之间用分号分开, 在每一行的第 2 个字段之后输出两个换行符.

```
BEGIN   { OFS = ":"; ORS = "\n\n" }
        { print $1, $2 }
```

作为对比,

```
{ print $1 $2 }
```

打印第 1 个与第 2 个字段, 两个字段之间没有输出字段分隔符插入, 这是因为 $1 $2 表示两个字符串的拼接.

printf 语句

printf 用于产生格式化的输出. 它与 C 语言中的 printf 函数很像, 但是 awk 的 printf 不支持格式说明符 *. 与 print 一样, 它也有带括号与不带括号的两种形式:

printf *format*,*expression*$_1$,*expression*$_2$,..., *expression*$_n$
printf(*format*,*expression*$_1$,*expression*$_2$, ..., *expression*$_n$)

参数 *format* 总是必须的, 它是一个变量, 其字符串值含有字面文本与格式说明符, 字面文本会按照文本的字面值输出, 格式说明符规定了参数列表中的表达式将被如何格式化地输出, 表 2.9 列出了所有的格式说明符. 每一个格式说明符都以 % 开始, 以转换字符结束, 可能含有下面三种修饰符:

57

-	表达式在它的域内左对齐
width	为了达到规定的宽度, 必要时填充空格; 前导的 0 表示用零填充
.prec	字符串最大宽度, 或十进制数的小数部分的位数

表 2.10 包含有几个例子, 这些例子说明了格式说明符, 数据及其对应的输出. 由 printf 产生的输出不会在行末自动插入一个换行符, 除非你显式放置了一个换行符.

表 2.9: printf 格式控制字符

字符	表达式将被打印为
c	ASCII 字符
d	十进制整数
e	[-]d.dddddd E[+-]dd
f	[-]ddd.dddddd
g	按照 e 或 f 进行转换, 选择较短的那个, 无意义的零会被抑制
o	无符号八进制数
s	字符串
x	无符号十六进制数
%	打印一个百分号 %, 不会有参数被吸收

表 2.10: printf 格式说明符的示例

fmt	$1	printf(fmt, $1)
%c	97	a
%d	97.5	97
%5d	97.5	97
%e	97.5	9.750000e+01
%f	97.5	97.500000
%7.2f	97.5	97.50
%g	97.5	97.5
%.6g	97.5	97.5
%o	97	141
%06o	97	000141
%x	97	61
%s	January	|January|
%10s	January	| January|
%-10s	January	|January |
%.3s	January	|Jan|
%10.3s	January	| Jan|
%-10.3s	January	|Jan |
%%	January	%

输出到文件

重定向运算符 > 与 >> 用于将输出重定向到文件, 而不是原来的标准输出. 下面这个程序将所有输入行的第 1 个与第 3 个字段输出到两个文件中: bigpop, 如果第 3 个字段大于 100, 否则的话, 输出到 smallpop:

```
$3 > 100    { print $1, $3 > "bigpop" }
$3 <= 100   { print $1, $3 > "smallpop" }
```

注意, 文件名必须用双引号括起来; 如果没有双引号的话, bigpop 与 smallpop 将被当作未初始化的变量. 文件名也可以是表达式或变量: 58

```
{ print($1, $3) > ($3 > 100 ? "bigpop" : "smallpop") }
```

上面这个程序做的是同样的工作, 程序

```
{ print > $1 }
```

将所有的输入行输出到以第 1 个字段命名的文件中.

在 print 与 printf 语句中, 如果参数列表中的表达式包含有关系运算符, 那么, 表达式 (或者是参数列表) 需要用括号括起来. 这样做是为了避免由重定向运算符 > 带来的歧义. 在程序

```
{ print $1, $2 > $3 }
```

中, > 是重定向运算符, 而不是第 2 个表达式的一部分, 所以程序的功能是将第 1 个与第 2 个字段输出到以第 3 个字段命名的文件中. 如果你想要让 > 成为第 2 个表达式的一部分, 使用括号:

```
{ print $1, ($2 > $3) }
```

还有一点需要注意的是, 重定向运算符只会打开文件一次; 随后的 print 或 printf 语句将更多的数据添加到文件的末尾. 重定向运算符 > 在打开文件时先把文件内容清空, 然后再往里写数据. 如果用的是 >>, 那么文件原来的内容便会保留下来, 而把新的内容添加到原来内容的末尾.

输出到管道

在支持管道的系统中, 也可以把输出重定向到管道, 而不仅仅是文件. 语句

```
print | command
```

导致 print 的输出以管道的方式传递给 *command*

假设我们想要得到一张表格, 表格的每一行都是大洲的名字及其对应的人口数, 人口按照从大到小的顺序排列. 下面这个程序为每个大洲计算总人口, 人口数存储在数组 pop 中, END 动作打印大洲的名字及其对应的人口数, 并把打印的内容输送给 sort 命令[①]

```
# print continents and populations, sorted by population

BEGIN { FS = "\t" }
      { pop[$4] += $3 }
```

[①] 正确的写法应该是 sort -t'\t' -k1rn

```
END   { for (c in pop)
            printf("%15s\t%6d\n", c, pop[c]) | "sort -t'\t' +1rn"
      }
```

程序的输出是

```
          Asia   2173
        Europe    172
 North America    340
 South America    134
```

59

管道的另一个用法是把输出重定向至标准错误文件 (Unix 系统); 于是输出将会出现在用户的终端, 而不是标准输出, 有几种惯用语法可以把输出重定向至标准错误文件:

```
print message | "cat 1>&2"          # redirect cat to stderr
system("echo '" message "' 1>&2")   # redirect echo to stderr
print message > "/dev/tty"          # write directly on terminal
```

虽然我们的大多数例子都把字面字符串包围在双引号中, 其实命令行与文件名可以用任意的表达式指定. 在涉及到输出重定向的 print 语句中, 文件或管道通过它们的名字标识, 也就是说, 上面程序的管道是按字符串的字面值命名的

```
sort -t'\t' +1rn
```

通常来说, 在程序运行期间, 文件或管道只被打开一次. 如果文件或管道被显式地关闭, 而后面又使用到了, 那么它们就会被重新打开.

关闭文件与管道

语句 close(*expression*) 关闭一个文件或管道, 文件或管道由 *expression* 指定; *expression* 的字符串值必须与最初用于创建文件或管道的字符串值相同. 于是

```
close("sort -t'\t' +1rn")
```

关闭上面打开的排序管道.

在同一个程序中, 如果你写了一个文件, 而待会儿想要读取它, 那么就需要调用 close. 某一时刻, 同时处于打开状态的文件或管道数量最大值由实现定义.

2.5 输入

为 awk 提供输入数据有若干种方式. 最常见的是把输入数据放在一个文件中, 例如文件 data, 然后再键入

```
awk 'program' data
```

如果没有指定输入文件, awk 就从它的标准输入读取数据; 所以, 另一种常用的方法是把另一个程序的输出以管道的方式输送给 awk. 举例来说, 实用程序 egrep 从输入行中挑选具有指定正则表达式的行, 虽然 awk 也可以做同样的工作, 但是与前者相比就慢得多了. 我们可以输入命令

```
egrep 'Asia' countries | awk 'program'
```

60

egrep 挑出那些含有 Asia 的行, 再把这些行输送给 awk 做进一步的处理.

输入分隔符

内建变量 FS 的默认值是 " ", 也就是一个空格符. 当 FS 具有这个特定值时, 输入字段按照空格和 (或) 制表符分割, 前导的空格与制表符会被丢弃, 所以下面三行数据中, 其每一行的第 1 个字段都相同:

```
field1
  field1
    field1      field2
```

然而, 当 FS 是其他值时, 前导的空格与制表符不会被丢弃.

把一个字符串赋值给内建变量 FS 就可以改变字段分隔符. 如果字符串的长度多于一个字符, 那么它会被当成一个正则表达式. 当前输入行中, 与该正则表达式匹配的最左, 最长, 非空且不重叠的子字符串变成字段分隔符, 举例来说,

```
BEGIN { FS = ",[ \t]*|[ \t]+" }
```

如果某个子串由一个后面跟着空格或制表符的逗号组成, 或者没有逗号, 只有空格与制表符, 那么这个子串就是字段分隔符.

如果 FS 被设置成单个字符 (除了空格符), 那么这个字符就变成字段分隔符. 这个约定使得把正则表达式元字符当作字段分隔符来用, 变得很容易:

```
FS = "|"
```

把 | 变成字段分隔符, 但是有一些间接的方法需要注意, 就像

```
FS = "[ ]"
```

把字段分隔符设置为一个空格符.

多行记录

默认情况下, 记录之间由换行符分隔, 所以术语 “行” 与 “记录” 在通常情况下是等价的. 默认的记录分隔符可以通过向内建变量 RS 赋予新值来改变, 但是必须按照某种受限的方式来进行. 如果 RS 被设置成空值, 正如

```
BEGIN { RS = "" }
```

61

那么记录之间将由一个或多个的空行来分隔, 并且每个记录可以占据多行. 把 RS 重新设置成换行符 (RS = "\n") 就可以恢复原来的效果. 当记录由多行组成时, 无论 FS 是什么值, 换行符总是字段分隔符之一.

处理多行记录的通常方式是使用

```
BEGIN { RS = ""; FS = "\n" }
```

把记录分隔符设置成一个或多个空白行, 把字段分隔符设置成单个的换行符; 每一行都是一个单独的字段. 记录的长度是有限制的, 通常是 3000 个字符. 关于如何处理多行记录, 在第 三 章包含更多的讨论.

表 2.11: getline 函数

表达式	被设置的变量
getline	$0, NF, NR, FNR
getline *var*	*var*, NR, FNR
getline <*file*	$0, NF
getline *var* <*file*	*var*
cmd | getline	$0, NF
cmd | getline *var*	*var*

getline 函数

函数 getline 可以从当前输入行, 或文件, 或管道, 读取输入. getline 抓取下一个记录, 按照通常的方式把记录分割成一个个的字段. 它会设置 NF, NR, 和 FNR; 如果存在一个记录, 返回 1, 若遇到文件末尾, 返回 0, 发生错误时返回 -1 (例如打开文件失败).

表达式 getline x 读取下一条记录到变量 x 中, 并递增 NR 与 FNR, 不会对记录进行分割, 所以不会设置 NF.

表达式

```
getline <"file"
```

从文件 file 读取输入. 它不会对 NR 与 FNR 产生影响, 但是会执行字段分割, 并且设置 NF.

表达式

```
getline x <"file"
```

从 file 读取下一条记录, 存到变量 x 中. 记录不会被分割成字段, 变量 NF, NR, 与 FNR 都不会被修改.

表 2.11 总结了 getline 函数的所有形式. 每个表达式的值都是 getline 的返回值.

作为一个例子, 这个程序把它的输入复制到输出, 除了类似下面这样的行:

```
#include "filename"
```

这些行将被文件 filename 的内容所替代.

```
# include - replace #include "f" by contents of file f

/^#include/ {
    gsub(/"/, "", $2)
    while (getline x <$2 > 0)
        print x
    next
}
{ print }
```


也可以把其他命令的输出直接输送给 getline. 例如, 语句

```
while ("who" | getline)
    n++
```

执行 Unix 命令 who (只执行一次), 并将它的输出传递给 getline. who 命令的输出是所有已登录用户的用户名, while 循环的每一次迭代都从这个用户名列表里读取一行, 并递增变量 n, 当循环结束后, n 的值就是已登录用户的人数. 类似的, 表达式

```
"date" | getline d
```

把命令 date 的输出保存到变量 d 中, 于是 d 就被设置为当前日期. 同样, 在非 Unix 系统上可能不支持管道.

在所有的涉及到 getline 的情况中, 你必须注意由于文件无法访问而返回的错误. 写出这样的代码是有可能的

```
while (getline <"file") ...      # dangerous
```

如果文件不存在的话, 这将是个死循环, 因为对于不存在的文件, getline 返回 -1, 非零值都表示真. 安全的方式是

```
while (getline <"file" > 0) ...      # safe
```

这个循环只有在 getline 返回 1 的情况下才会执行.

命令行变量赋值

63

正如我们之前看到过的那样, awk 命令行具有多种形式:

```
awk 'program' f1 f2 ...
awk -f progfile f1 f2 ...
awk -Fsep 'program' f1 f2 ...
awk -Fsep -f progfile  f1 f2 ...
```

在上面的命令行中, f1, f2 等变量是命令行参数, 通常代表文件名. 如果一个文件名具有形式 *var=text*, 那这就表示赋值语句, 把 *text* 赋值给 *var*, 当这个参数被当作文件来访问时, 执行赋值动作. 这种类型的赋值语句允许程序在读文件之前或之后改变变量的值.

命令行参数

命令行参数可以通常 awk 的内建数组 ARGV 来访问, 内建变量 ARGC 的值是参数的个数再加 1. 对于命令行

```
awk -f progfile a v=1 b
```

ARGC 的值是 4, ARGV[0] 含有 awk, ARGV[1] 含有 a, ARGV[2] 含有 v=1, ARGV[3] 含有 b. ARGC 之所以会比参数的个数多 1, 是因为命令的名字 awk 也被当作参数之一, 存放在索引为 0 的位置, 就像 C 程序那样. 然而, 如果 awk 程序在命令行中出现, 那它就不会被当作参数, 对 -f *filename* 或任意的 -F 选项同样如此. 例如, 对于命令行

```
awk -F'\t' '$3 > 100' countries
```

ARGC 的值是 2, ARGV[1] 的值是 countries.

下面这个程序回射它的命令行参数:

```
# echo - print command-line arguments

BEGIN {
    for (i = 1; i < ARGC; i++)
        printf "%s ", ARGV[i]
    printf "\n"
}
```

请注意在 BEGIN 动作里所发生的一切: 因为没有其他的 模式–动作 语句, 所以参数不会被当作文件名, 也没有输入被读取.

另一个使用命令行参数的例子是 seq, 它可以产生一个整数序列:

```
# seq - print sequences of integers
# input:  arguments q, p q, or p q r;  q >= p; r > 0
# output: integers 1 to q, p to q, or p to q in steps of r

BEGIN {
    if (ARGC == 2)
        for (i = 1; i <= ARGV[1]; i++)
            print i
    else if (ARGC == 3)
        for (i = ARGV[1]; i <= ARGV[2]; i++)
            print i
    else if (ARGC == 4)
        for (i = ARGV[1]; i <= ARGV[2]; i += ARGV[3])
            print i
}
```

命令 64

```
awk -f seq 10
awk -f seq 1 10
awk -f seq 1 10 1
```

都是生成 1 到 10 这十个整数.

ARGV 里面的参数可以修改或添加, ARGC 可能会随之改变. 每当有一个输入文件结束, awk 把 ARGV 的下一个非空元素 (直到 ARGC - 1 的当前值) 看作是下一个输入文件名. 于是, 把 ARGV 的一个元素设置为空, 意味着 awk 不会把它看作是输入文件. 名字 "-" 可以当作标准输入来用.

2.6 与其他程序的交互

这一节描述几种 awk 程序与其他命令合作的方式. 主要针对 Unix 系统, 在非 Unix 系统上, 这里的例子可能无法工作或表现出不同的行为.

system 函数

内建函数 system(*expression*) 用于执行命令, 命令由 *expression* 给出, system 的返回值就是命令的退出状态.

例如, 我们可以重新实现 2.5 节提到的文件包含程序, 就像这样

```
$1 == "#include" { gsub(/"/, "", $2); system("cat " $2); next }
                 { print }
```

如果第一个字段是 #include, 那么双引号就会被移除, 然后 Unix 命令 cat 打印以第 2 个字段命名的文件. 其他输入行被原样复制.

用 AWK 制作 Shell 命令

65

到目前为止的所有例子, awk 程序都是写在一个文件中, 然后利用 -f 选项读取, 或者是出现在命令行中, 用一对单引号括起来, 就像:

```
awk '{ print $1 }' ...
```

Awk 用到了许多 Shell 也同样会用到的字符, 例如 $ 与 ", 把 awk 程序包围在一对单引号中可以确保 Shell 把程序原封不动地传递给 awk.

上面提到的两种执行 awk 程序的方法都要求用户自己打一些字. 为了降低打字的工作量, 我们想要把命令与程序都写到一个可执行文件中, 通过键入文件名来执行程序. 假设我们想要写一个命令 field1, 用于打印每个输入行的第一个字段, 其实这非常容易, 我们把

```
awk '{ print $1 }' $*
```

写到一个文件 field1 中, 为了让它成为一个可执行文件, 我们必须执行 Unix 命令:

```
chmod +x field1
```

现在, 如果我们想要打印某些文件的每行的第 1 个字段, 只要键入

```
field1 filenames ...
```

现在我们考虑实现一个更加通用的 field, 程序可以从每个输入行中打印任意组合的字段, 换句话说, 命令

```
field n1 n2 ... file1 file 2 ...
```

按照特定的顺序打印特定的字段. 但是我们如何获取 n_i 的值, 又如何区分 n_i 与文件名参数?

对于采用 Shell 编程的人来说, 解决上面提到的两个问题有多种方式, 如果使用 awk 的话, 最简单的方式是扫描内建数组 ARGV, 获取 n_i 的值之后, 把数组中对应的位置清零, 这样它们就不会被当成文件名:

66

```
# field - print named fields of each input line
#   usage:  field n n n ... file file file ...

awk '
BEGIN {
    for (i = 1; ARGV[i] ~ /^[0-9]+$/; i++) { # collect numbers
        fld[++nf] = ARGV[i]
        ARGV[i] = ""
    }
    if (i >= ARGC)   # no file names so force stdin
        ARGV[ARGC++] = "-"
}
{   for (i = 1; i <= nf; i++)
        printf("%s%s", $fld[i], i < nf ? " " : "\n")
}
' $*
```

这个程序可以处理标准输入, 也可以处理文件名参数列表, 并且字段的顺序与数量都可以是任意的.

2.7 小结

正如我刚开始说的那样, 这一章很长, 讨论了许多细节, 如果你是逐字逐词地读到这里来的话, 那确实是非常专注. 到后面你就会发现, 时不时的就需要回到这章再看一下某些小节, 有时候是为了精确地了解程序如何工作的, 有时候则是因为后面的章节出现了一些之前没有试过的构造方法.

和其他语言一样, awk 也需要通过经验与实践来学习, 所以我们鼓励你开始自己写程序. 程序不需要有多么的庞大或复杂 — 你可以通过仅仅几行代码, 来了解语言的某些特性如何工作, 或测试程序的某些关键点, 你也可以自己手动输入数据来查看程序的运行效果.

参考资料

The C Programming Language (Brian Kernighan 与 Dennis Ritchie 著, Prentice-Hall 1978 年出版) 对 C 程序设计语言进行了完整地描述. 关于如何使用 Unix 系统有大量的资料可供参考; *The Unix Programming Environment* (Brian Kernighan 与 Rob Pike 著, Prentice-Hall 1984 年出版) 包含一个扩展讨论, 讨论的是如何创建包含 awk 的 shell 程序.

第三章　数据处理

Awk 最初的设计目标是用于日常的数据处理, 例如信息查询, 数据验证, 以及数据转换与归约, 我们已经在第 一 章与第 二 章见到了关于这些的简单例子. 在这一章, 我们会按照相同的思路, 考虑一些更加复杂的任务, 大多数例子一次只处理一行, 但是最后一节讨论如何处理占据多行的输入记录.

Awk 程序通常按照增量模式开发: 先写上几行, 测试一下, 然后再添加几行, 再测试, 如此进行下去. 这本书里的大多数比较大的程序都是按照这种模式开发的.

也可以按照传统的方式来开发 awk 程序, 先拟好程序的主体框架, 然后查询手册, 但是, 通过修改已有的程序来达到我们自己想要的效果, 通常来说会更加容易. 于是, 这本书里的程序扮演了另一个角色: 通过例子来提供实用的编程模型.

3.1　数据转换与归约

Awk 的一个最常用的功能是把数据从一种形式转换成另一种形式, 通常情况下, 是把一种程序的输出格式, 转换成另一种程序要求的格式. 另一个常用的功能是从一个大数据集中提取相关的数据, 通常伴随着汇总信息的重新格式化与准备, 这一节包含了许多关于这些主题的例子.

列求和

我们已经看到了“两行” awk 程序的几种变体, 这些程序对单个字段上的所有数字求和. 下面这个程序执行的工作更加复杂, 但却是比较典型的数据归约任务. 每一个输入行都含有若干个字段, 每一个字段都包含数字, 程序的任务是计算每一列的和, 而不管该行有多少列.

```
# sum1 - print column sums
#   input:  rows of numbers
#   output: sum of each column
#     missing entries are treated as zeros

    { for (i = 1; i <= NF; i++)
          sum[i] += $i
      if (NF > maxfld)
          maxfld = NF
    }
END { for (i = 1; i <= maxfld; i++) {
          printf("%g", sum[i])
```

```
            if (i < maxfld)
                printf("\t")
            else
                printf("\n")
        }
    }
```

68

变量的自动初始化在这里显得非常方便, 因为 maxfld (到目前为止, 看到的字段数最大值) 自动从 0 开始, 随着程序的运行, 所有的项都被放入数组 sum 中, 虽然只有到程序运行结束, 我们才能知道数组中到底有多少项. 值得注意的是, 如果输入文件为空, 那么程序什么也不会打印出来.

程序不需要知道一行有多少个字段, 这对我们写程序来说就非常方便, 但是它不检查参与运算的项是否都是数值, 也不检查每行的字段数是否相同. 下面的程序做的是同样的事情, 但是它会检查每行的字段数是否与第一行的相同:

```
# sum2 - print column sums
#     check that each line has the same number of fields
#        as line one

NR==1 { nfld = NF }
      { for (i = 1; i <= NF; i++)
            sum[i] += $i
        if (NF != nfld)
            print "line " NR " has " NF " entries, not " nfld
      }
END   { for (i = 1; i <= nfld; i++)
            printf("%g%s", sum[i], i < nfld ? "\t" : "\n")
      }
```

我们还修正了位于 END 的输出代码, 这段代码显示了如何利用条件表达式, 使得在列与列之间插入一个制表符, 在最后一列之后插入一个换行符.

现在, 假设某些字段不是数值型, 所以它们不能被计算在内. 策略是新增一个数组 numcol, 用于跟踪数值型字段, 函数 isnum 用于检查某项是否是一个数值, 由于用到了函数, 所以测试只需要在一个地方完成, 这样做有助于将来对程序进行修改. 如果程序足够相信它的输入, 那么只需要查看第 1 行就够了, 我们仍然需要 nfld, 因为在 END 中, NF 的值是零.

69

```
# sum3 - print sums of numeric columns
#     input:  rows of integers and strings
#     output: sums of numeric columns
#       assumes every line has same layout

NR==1 { nfld = NF
        for (i = 1; i <= NF; i++)
            numcol[i] = isnum($i)
```

```
        }

        { for (i = 1; i <= NF; i++)
              if (numcol[i])
                  sum[i] += $i
        }

END     { for (i = 1; i <= nfld; i++) {
              if (numcol[i])
                  printf("%g", sum[i])
              else
                  printf("--")
              printf(i < nfld ? "\t" : "\n")
          }
        }

function isnum(n) { return n ~ /^[+-]?[0-9]+$/ }
```

函数 isnum 把数值定义成一个或多个数字, 可能有前导符号. 关于数值更加一般的定义可以在 2.1 节的正则表达式那里找到.

Exercise 3.1 修改程序 sum3: 忽略空行.

Exercise 3.2 为数值添加更加一般的正则表达式. 它会如何影响运行时间?

Exercise 3.3 如果把第 2 个 for 语句的 numcol 测试拿掉, 会产生什么影响?

Exercise 3.4 写一个程序, 这个程序读取一个 条目–数额 对列表, 对列表中的每一个条目, 累加它的数额; 在结束时, 打印条目以及它的总数额, 条目按照字母顺序排序.

计算百分比与分位数

假设我们不想知道每列的总和, 但是想知道每一列所占的百分比, 要完成这个工作就必须对数据遍历两遍. 如果只有一列是数值, 而且也没有太多的数据, 最简单的办法是在第一次遍历时, 把数值存储在一个数组中, 每二次遍历时计算百分比并把它打印出来:

```
# percent
#   input:  a column of nonnegative numbers
#   output: each number and its percentage of the total

    { x[NR] = $1; sum += $1 }

END { if (sum != 0)
```

```
            for (i = 1; i <= NR; i++)
                printf("%10.2f %5.1f\n", x[i], 100*x[i]/sum)
        }
```

70

虽然包含了稍微复杂一点的转换关系, 但是它可以用于许多事情, 例如调整学生的成绩, 使得成绩分布符合某种曲线. 一旦成绩计算完毕 (0 到 100 之间的数), 显示成一个直方图可能会比较有趣:

```
# histogram
#   input:  numbers between 0 and 100
#   output: histogram of deciles

    { x[int($1/10)]++ }

END { for (i = 0; i < 10; i++)
          printf(" %2d - %2d: %3d %s\n",
              10*i, 10*i+9, x[i], rep(x[i],"*"))
      printf("100:      %3d %s\n", x[10], rep(x[10],"*"))
    }

function rep(n,s,   t) {  # return string of n s's
    while (n-- > 0)
        t = t s
    return t
}
```

需要注意的是后缀递增运算符 `--` 如何控制 `while` 循环.

我们可以用随机生成的成绩来测试 `histogram`. 管道线上的第一个程序随机生成 200 个 0 到 100 的整数, 并把这些整数输送给 `histogram`

```
awk '
# generate random integers
BEGIN { for (i = 1; i <= 200; i++)
            print int(101*rand())
      }
' |
awk -f histogram
```

它的输出是

```
  0 -  9:  20 ********************
 10 - 19:  18 ******************
 20 - 29:  20 ********************
 30 - 39:  16 ****************
 40 - 49:  23 ***********************
```

```
 50 - 59:  17 *****************
 60 - 69:  22 **********************
 70 - 79:  20 ********************
 80 - 89:  20 ********************
 90 - 99:  22 **********************
100:        2 **
```

71

Exercise 3.5 根据比例决定星号的个数: 当数据过多时, 一行的长度不会超过屏幕的宽度.

Exercise 3.6 修改 histogram, 把输入分拣到指定数量的桶中, 根据目前为止看到的数据调整每个桶的范围.

带逗号的数

设想我们有一张包含了许多数的表, 每个数都有逗号与小数点, 就像 12,345.67, 因为第一个逗号会终止 awk 对数的解析, 所以它们不能直接相加, 必须首先把逗号移除:

```
# sumcomma - add up numbers containing commas

    { gsub(/,/, ""); sum += $0 }
END { print sum }
```

gsub(/,/, "") 把每一个逗号都替换成空字符串, 也就是删除逗号.

这个程序不检查逗号是否处于正确的位置, 也不在答案中打印逗号. 往数字中加入逗号只需要很少的工作量, 下一个程序就展示了这点, 它为数字加上逗号, 保留两位小数. 这个程序的结构是非常值得效仿的: 一个函数只负责添加逗号, 剩下的部分只管读取与打印, 一旦测试通过, 新的函数就可以被包含到最终的程序中.

基本思路是在一个循环中, 从小数点开始, 从左至右, 在适当的位置插入逗号, 每次迭代都把一个逗号插到最左边的三个数字的前面, 这三个数字后面跟着一个逗号或小数点, 而且每一个逗号的前面至少有一个数字. 算法使用递归来处理负数: 如果输入参数是负数, 那么函数 addcomma 使用正数来调用它自身, 返回时再加上负号.

```
# addcomma - put commas in numbers
#   input:  a number per line
#   output: the input number followed by
#      the number with commas and two decimal places

    { printf("%-12s %20s\n", $0, addcomma($0)) }

function addcomma(x,   num) {
    if (x < 0)
        return "-" addcomma(-x)
    num = sprintf("%.2f", x)   # num is dddddd.dd
```

```
        while (num ~ /[0-9][0-9][0-9][0-9]/)
            sub(/[0-9][0-9][0-9][,.]/, ",&", num)
        return num
    }
```

72

请注意 & 的用法, 通过文本替换, sub 在每三个数字的前面插入一个逗号.

这是某些测试数据的输出:

```
0                              0.00
-1                            -1.00
-12.34                       -12.34
12345                     12,345.00
-1234567.89          -1,234,567.89
-123.                       -123.00
-123456                 -123,456.00
```

Exercise 3.7 修复 sumcomma (带逗号的数字求和程序): 检查数字中的逗号是否处于正确位置.

字段固定的输入

对于那些出现在宽度固定的字段里的信息, 在直接使用它们之前, 通常需要某种形式的预处理. 有些程序 (例如电子表格) 在固定的列上面放置序号, 而不是给它们带上字段分隔符[①]. 如果序号太宽了, 这些列就会邻接在一起. 字段固定的数据最适合用 substr 处理, 它可以将任意组合的列挑选出来. 举例来说, 假设每行的前 6 个字符包含一个日期, 日期的形式是 mmddyy, 如果我们想让它们按照日期排序, 最简单的办法是先把日期转换成 yymmdd 的形式:

```
# date convert - convert mmddyy into yymmdd in $1

{ $1 = substr($1,5,2) substr($1,1,2) substr($1,3,2); print }
```

如果输入是按照月份排序的, 就像这样

```
013042 mary's birthday
032772 mark's birthday
052470 anniversary
061209 mother's birthday
110175 elizabeth's birthday
```

那么程序的输出是

```
420130 mary's birthday
720327 mark's birthday
700524 anniversary
090612 mother's birthday
751101 elizabeth's birthday
```

73

现在数据已经准备好, 可以按照年, 月, 日来排序了.

Exercise 3.8 将日期转换成某种形式, 这种形式允许你对日期进行算术运算, 例如计算某两个日期之间的天数.

程序的交叉引用检查

Awk 经常用于从其他程序的输出中提取信息, 有时候这些输出仅仅是一些同种类的行的集合, 在这种情况下, 使用字段分割操作或函数 substr 是非常方便且合适的. 然而, 有时候产生输出的程序本来就打算将输出表示成人类可读的形式, 对于这种情况, awk 需要把精心格式化过的输出重新还原成机器容易处理的形式, 只有这样, 才能从互不相关的数据中提取信息, 下面是一个简单的例子.

大型程序通常由多个源文件组成, 知道哪些文件定义了哪些函数, 以及这些函数在哪里被使用 — 可以带来许多方便之处 (有时候这是非常重要的). 为了完成这个任务, Unix 提供了命令 nm, nm 从一个目标文件集合中提取信息, 并打印一张精心格式化过的列表, 这张表包含了名字, 定义, 以及名字在哪里被使用到, nm 的典型输出是

```
file.o:
00000c80 T _addroot
00000b30 T _checkdev
00000a3c T _checkdupl
         U _chown
         U _client
         U _close
funmount.o:
00000000 T _funmount
         U cerror
```

只有一个字段的行 (比如 file.o) 是文件名, 有两个字段的行 (比如 U _close) 表示的是名字被使用到的地方, 有三个字段的行表示的是名字被定义的地方. T 表示这个定义是一个文本符号 (函数), U 表示这个名字是未定义的.

如果直接使用这些未加处理过的信息, 来判断某个文件定义了哪些名字, 又或者是某个符号都在哪里被用到 — 将会非常得麻烦, 因为每个符号都没有和它所在的文件名放在一起. 对于稍微大型的 C 程序来说, nm 的输出将会非常得长 —- awk 源代码由 9 个文件组成, 它的 nm 输出超过了 850 行. 一个仅仅 3 行的 awk 程序就可以把文件名加到每个符号的前面, 经过它的处理, 后续的程序仅通过一行就可以获取到有用的信息:

74

```
# nm.format - add filename to each nm output line

NF == 1 { file = $1 }
NF == 2 { print file, $1, $2 }
NF == 3 { print file, $2, $3 }
```

[①] 原文为 Some programs, such as spreadsheets, put out numbers in fixed columns, rather than with field separators.

把上面 nm 的输出作为 nm.format 的输入, 结果是

```
file.o: T _addroot
file.o: T _checkdev
file.o: T _checkdupl
file.o: U _chown
file.o: U _client
file.o: U _close
funmount.o: T _funmount
funmount.o: U cerror
```

现在, 如果有其他程序想要对输出作进一步的处理就容易多了.

上面的输出没有包括行号信息, 也没有指出某个名字在文件中被用到了多少次, 但是这些信息很容易通过文本编辑器或另一个 awk 程序来获取. 本小节的 awk 程序不依赖于目标文件的编程语言, 所以它比通常的交叉引用工具更加灵活, 也更加简短.

格式化的输出

接下来我们要用 awk 赚点钱, 或者是打印支票. 输入数据由多行组成, 每一行都包括支票编号, 金额, 帐款人, 字段之间用制表符分开, 输出是标准的支票格式: 8 行高, 第 2 行与第 3 行是支票编号与日期, 都向右缩进 45 个空格, 第 4 行是收款人, 占用 45 个字符宽的区域, 紧跟在它后面的是 3 个空格, 再后面是金额, 第 5 行是金额的大写形式, 其他行都是空白. 支票看起来就像这样:

```
                                          1026
                                          Aug 31, 2015
Pay to Mary R. Worth---------------------------------   $123.45
the sum of one hundred twenty three dollars and 45 cents exactly
```

这是打印支票的 awk 程序: 75

```
# prchecks - print formatted checks
#   input:  number \t amount \t payee
#   output: eight lines of text for preprinted check forms

BEGIN {
    FS = "\t"
    dashes = sp45 = sprintf("%45s", " ")
    gsub(/ /, "-", dashes)        # to protect the payee
    "date" | getline date         # get today's date
    split(date, d, " ")
```

```
    date = d[2] " " d[3] ", " d[6]
    initnum()    # set up tables for number conversion
}
NF != 3 || $2 >= 1000000 {        # illegal data
    printf("\nline %d illegal:\n%s\n\nVOID\nVOID\n\n\n", NR, $0)
    next                          # no check printed
}
{   printf("\n")                  # nothing on line 1
    printf("%s%s\n", sp45, $1)    # number, indented 45 spaces
    printf("%s%s\n", sp45, date)  # date, indented 45 spaces
    amt = sprintf("%.2f", $2)     # formatted amount
    printf("Pay to %45.45s   $%s\n", $3 dashes, amt)  # line 4
    printf("the sum of %s\n", numtowords(amt))        # line 5
    printf("\n\n\n")              # lines 6, 7 and 8
}

function numtowords(n,   cents, dols) { # n has 2 decimal places
    cents = substr(n, length(n)-1, 2)
    dols = substr(n, 1, length(n)-3)
    if (dols == 0)
        return "zero dollars and " cents " cents exactly"
    return intowords(dols) " dollars and " cents " cents exactly"
}

function intowords(n) {
    n = int(n)
    if (n >= 1000)
        return intowords(n/1000) " thousand " intowords(n%1000)
    if (n >= 100)
        return intowords(n/100) " hundred " intowords(n%100)
    if (n >= 20)
        return tens[int(n/10)] " " intowords(n%10)
    return nums[n]
}

function initnum() {
    split("one two three four five six seven eight nine " \
          "ten eleven twelve thirteen fourteen fifteen " \
          "sixteen seventeen eighteen nineteen", nums, " ")
    split("ten twenty thirty forty fifty sixty " \
```

```
            "seventy eighty ninety", tens, " ")
    }
```

程序中包含了几个比较有趣的部分. 首先, 要注意到我们在 BEGIN 中是如何利用 sprintf 来生成 76
空格字符串的, 并且通过替换将空格字符串转换成破折号. 还要注意的是, 在函数 initnum 中, 我们如何通过行的延续与字符串拼接来创建 split 的参数 — 这是很常见的编程技巧.

日期通过

```
"data" | getline data   # get today's date
```

从系统获取, 该行执行 date, 再把它的输出输送给 getline. 为了把

```
Wed Jun 17 13:39:36 EDT 1987
```

转换成

```
Jun 17, 1987
```

我们需要自己做一些处理工作. (在不支持管道的非 Unix 平台上, 该程序需要做些修改才能正确运行)

函数 numtowords 与 intowords 把数字转换成对应的单词, 转换过程非常直接 (虽然程序用了一半的代码来做这件事). intowords 是一个递归函数, 它调用自身来处理一个规模较小的子问题, 这是本章出现过的第 2 个递归函数, 在后面我们还会遇到更多这样的函数. 在很多情况下, 为了把一个大问题分解成相对容易解决的小问题, 递归都是一种非常有效的方法.

Exercise 3.9 利用前面提到过的程序 addcomma, 为金额加上逗号.

Exercise 3.10 对于负的, 或特别大的金额, 程序 prchecks 处理得并不是很好. 修改程序: 拒绝金额为负的打印请求, 同时能将数额特别巨大的金额分成两行打印出来.

Exercise 3.11 函数 numtowords 有时会在一行中连着打印两个空格, 还会打印出像 “one dollars” 这样有错误的句子, 你会如何消除这些瑕疵?

Exercise 3.12 修改 prchecks: 在适当的地方, 为金额的大写形式加上连字符, 比如 “twenty-one dollars”.

3.2　数据验证

Awk 的另一个常用功能是数据验证: 确保数据是合法的, 或至少合理的. 本节包含了几个用于验证输入有效性的小程序, 例如, 考虑上一节出现的列求和程序, 有没有这样一种情况: 在本应是数值的字段上出现了非数值的量 (或反之)? 下面这个程序与列求和程序非常相似, 但没有求和操作: 77

```
# colcheck - check consistency of columns
#   input:  rows of numbers and strings
#   output: lines whose format differs from first line

NR == 1 {
```

```
        nfld = NF
        for (i = 1; i <= NF; i++)
           type[i] = isnum($i)
    }
    {   if (NF != nfld)
           printf("line %d has %d fields instead of %d\n",
              NR, NF, nfld)
        for (i = 1; i <= NF; i++)
           if (isnum($i) != type[i])
              printf("field %d in line %d differs from line 1\n",
                 i, NR)
    }

    function isnum(n) { return n ~ /^[+-]?[0-9]+$/ }
```

同样, 我们把数值看成是仅由数字构成的序列, 可能有前导符号, 如果想让这个判断更加完整, 请参考 2.1 节关于正则表达式的讨论.

对称的分隔符

本书有一个机器可读的版本, 在该版本中, 每一个程序都由一种特殊行开始, 这个特殊行以 .P1 打头, 同样, 程序以一种特殊行结束, 该行以 .P2 打头. 这些特殊行叫作“文本格式化”命令, 有了这些命令的帮助, 在排版书籍的时候, 程序可以以一种容易识别的字体显示出来. 因为程序之间不能互相嵌套, 所以这些文本格式化命令必须按照交替序列, 轮流出现:

```
.P1 .P2 .P1 .P2 ... .P1 .P2
```

如果其中一个被漏掉了, 那么排版软件的输出将会是完全混乱的. 为了确保书籍被正确得排版, 我们写了这个小程序, 它可以用于检查分隔符是否按照正确的顺序出现, 程序虽小, 但却是检查程序的典型代表:

```
# p12check - check input for alternating .P1/.P2 delimiters

/^\.P1/ { if (p != 0)
               print ".P1 after .P1, line", NR
          p = 1
        }
/^\.P2/ { if (p != 1)
               print ".P2 with no preceding .P1, line", NR
          p = 0
        }
END     { if (p != 0) print "missing .P2 at end" }
```

如果分隔符按照正确的顺序出现, 那么变量 p 就会按照 0 1 0 1 0 ... 1 0 的规律变化, 否则, 一条错误消息被打印出来, 消息含有发生错误时, 当前输入行所在的行号.

Exercise 3.13 如何修改这个程序, 使得它可以处理具有多种分隔符的文本?

密码文件检查

Unix 系统中的密码文件含有授权用户的用户名及其相关信息, 密码文件的每一行都由 7 个字段组成:

```
root:qyxRi2uhuVjrg:0:2::/:
bwk:1L./v6iblzzNE:9:1:Brian Kernighan:/usr/bwk:
ava:otxs1oTVoyvMQ:15:1:Al Aho:/usr/ava:
uucp:xutIBs2hKtcls:48:1:uucp daemon:/usr/lib/uucp:uucico
pjw:xNqY//GDc8FFg:170:2:Peter Weinberger:/usr/pjw:
mark:j0z1fuQmqIvdE:374:1:Mark Kernighan:/usr/bwk/mark:
...
```

第 1 个字段是用户的登录名, 只能由字母或数字组成. 第 2 个字段是加密后的登录密码, 如果密码是空的, 那么任何人都可以利用这个用户名来登录系统, 如果这个字段非空, 那么只有知道密码的用户才能成功登录. 第 3 与第 4 个字段是数字, 第 6 个字段以 / 开始. 下面这个程序打印的行不符合前面描述的结构, 顺带打印它们的行号, 及一条恰当的诊断消息, 每个晚上都让这个程序运行一遍可以让系统更加健康, 远离攻击.

```
# passwd - check password file

BEGIN {
    FS = ":" }
NF != 7 {
    printf("line %d, does not have 7 fields: %s\n", NR, $0) }
$1 ~ /[^A-Za-z0-9]/ {
    printf("line %d, nonalphanumeric user id: %s\n", NR, $0) }
$2 == "" {
    printf("line %d, no password: %s\n", NR, $0) }
$3 ~ /[^0-9]/ {
    printf("line %d, nonnumeric user id: %s\n", NR, $0) }
$4 ~ /[^0-9]/ {
    printf("line %d, nonnumeric group id: %s\n", NR, $0) }
$6 !~ /^\// {
    printf("line %d, invalid login directory: %s\n", NR, $0) }
```

这是增量开发程序的好例子: 每当有人认为需要添加新的检查条件时, 只需要往程序中添加即可, 其他部分保持不动, 于是程序会越来越完善.

自动生成数据验证程序

79

密码文件检查程序由我们手工编写而成, 不过更有趣的方式是把条件与消息集合自动转化成检查程序. 下面这个集合含有几个错误条件及其对应的提示信息, 这些错误条件取自上一个程序. 如果某个输入行满足错误条件, 对应的提示信息就会被打印出来.

```
NF != 7                 does not have 7 fields
$1 ~ /[^A-Za-z0-9]/     nonalphanumeric user id
$2 == ""                no password
```

下面这个程序把 条件-消息 对转化成检查程序:

```
# checkgen - generate data-checking program
#     input:  expressions of the form: pattern tabs message
#     output: program to print message when pattern matches

BEGIN { FS = "\t+" }
{ printf("%s {\n\tprintf(\"line %%d, %s: %%s\\n\",NR,$0) }\n",
      $1, $2)
}
```

程序的输出是一系列的条件与打印消息的动作:

```
NF != 7 {
    printf("line %d, does not have 7 fields: %s\n",NR,$0) }
$1 ~ /[^A-Za-z0-9]/ {
    printf("line %d, nonalphanumeric user id: %s\n",NR,$0) }
$2 == "" {
    printf("line %d, no password: %s\n",NR,$0) }
```

检查程序运行时, 如果当前输入行使得条件为真, 那么程序就会打印出一条消息, 消息含有当前输入行的行号, 错误消息, 及当前输入行的内容. 需要注意的是, 在程序 checkgen 中, printf 格式字符串的某些特殊字符需要用双引号括起来, 只有这样才能生成有效的程序. 举例来说, 为了输出一个 %, 必须将它写成 %%; 为了输出 \n, 必须写成 \\n.

用一个 awk 程序来生成另一个 awk 程序是一个应用很广泛的技巧 (不限于 awk 语言), 在本书后面的章节里我们还会看到更多这样的例子.

Exercise 3.14 增强 checkgen 的功能, 使得我们可以原封不动地向程序传递一段代码, 例如创建一个 BEGIN 来设置字段分隔符.

AWK 的版本

Awk 可以用来检验程序, 也可以用来组织程序测试. 本小节包含的程序有几分近亲相奸的味道: 用 awk 程序检查 awk 程序.

Awk 的新版可能包含更多的内建变量与内建函数, 而老程序有可能不小心用到了 这些名字, 例如,

80

老程序用 sub 命名一个变量, 而在新版 awk 中, sub 是一个内建函数. 下面的程序可以用来检测这种错误:

```
# compat - check if awk program uses new built-in names

BEGIN { asplit("close system atan2 sin cos rand srand " \
               "match sub gsub", fcns)
        asplit("ARGC ARGV FNR RSTART RLENGTH SUBSEP", vars)
        asplit("do delete function return", keys)
      }

      { line = $0 }

/"/   { gsub(/"([^"]|\\")*"/, "", line) }     # remove strings,
/\//  { gsub(/\/([^\/]|\\\/)+\//, "", line) } # reg exprs,
/#/   { sub(/#.*/, "", line) }                # and comments

      { n = split(line, x, "[^A-Za-z0-9_]+")  # into words
        for (i = 1; i <= n; i++) {
            if (x[i] in fcns)
                warn(x[i] " is now a built-in function")
            if (x[i] in vars)
                warn(x[i] " is now a built-in variable")
            if (x[i] in keys)
                warn(x[i] " is now a keyword")
        }
      }

function asplit(str, arr) {  # make an assoc array from str
    n = split(str, temp)
    for (i = 1; i <= n; i++)
        arr[temp[i]]++
    return n
}

function warn(s) {
    sub(/^[ \t]*/, "")
    printf("file %s, line %d: %s\n\t%s\n", FILENAME, FNR, s, $0)
}
```

程序中真正复杂的地方在于替换语句: 在一个输入行被检查之前, 语句试图从输入行中移除被双引号包围的字符串, 正则表达式, 以及注释. 替换语句并不是非常完善, 所以有些行可能没有得到正确的处

理.

第 1 个 split 函数的第 3 个参数被解释成一个正则表达式, 输入行中, 被该表达式匹配的最左最长子字符串成为字段分隔符. split 把不含字母或数字的字符串当作分隔符, 将输入行分割成一个个子字符串, 每个子字符串仅含有字母或数字. 这个分割操作把所有的运算符或标点符号一次性移除. 81

函数 asplit 类似于 split, 但前者创建一个数组, 数组的下标是字符串内的单词, 然后就可以测试新来的单词是否在数组内.

如果把文件 compat 作为输入, 输出是:

```
file tmp.awk, line 12: gsub is now a built-in function
    /\//  { gsub(/\/([^\/]|\\\/)+\//, "", line) } # reg exprs,
file tmp.awk, line 13: sub is now a built-in function
    /#/   { sub(/#.*/, "", line) }                # and comments
file tmp.awk, line 26: function is now a keyword
    function asplit(str, arr) {  # make an assoc array from str
file tmp.awk, line 30: return is now a keyword
    return n
file tmp.awk, line 33: function is now a keyword
    function warn(s) {
file tmp.awk, line 34: sub is now a built-in function
    sub(/^[ \t]*/, "")
file tmp.awk, line 35: FNR is now a built-in variable
    printf("file %s, line %d: %s\n\t%s\n", FILENAME, FNR, s, $0)
```

Exercise 3.15 重写 compat, 不用 asplit, 而是用正则表达式来识别关键词, 内建函数等. 比较两个版本的复杂度与速度.

Exercise 3.16 因为 awk 的变量不需要事先声明, 所以如果你不小心把变量名写错了, awk 并不会检测到该错误. 写一个程序, 这个程序搜索文件中只出现一次的名字. 为了让这个程序更具实用价值, 你可能需要对函数的定义及其用到的变量花点心思.

3.3　打包与拆包

在讨论多行记录之前, 让我们先考虑一种特殊的情况: 如何将多个 ASCII 文件打包 (bundle) 成一个文件, 在打包完之后, 还可以将它们还原 (unbundle) 成原来的文件. 这一节包含的两个程序分别用来完成这两件事情, 我们可以用它们将多个小文件打包成一个文件, 从而节省磁盘空间或方便邮寄.

程序 bundle 非常简短, 简短到你可以直接在命令行上输入, 它所做的工作仅仅是为每一行加上文件名前缀, 文件名可以通过内建变量 FILENAME 得到.

```
# bundle - combine multiple files into one
{ print FILENAME, $0 }
```

对应的 unbundle 程序只是稍微需要花点心思: 82

```
# unbundle - unpack a bundle into separate files

$1 != prev { close(prev); prev = $1 }
           { print substr($0, index($0, " ") + 1) >$1 }
```

如果遇到一个新文件, 则关闭之前打开的文件, 如果文件不是很多 (小于同时处于打开状态的文件数的最大值), 那么这一行可以省略.

实现 bundle 与 unbundle 的方法还有很多种, 但是这里介绍的方法是最简单的, 而且对于比较短的文件, 空间效率也比较高. 另一种组织方式是在每一个文件之前, 添加一行带有文件名的, 容易识别的行, 这样的话, 文件名只需要出现一次.

Exercise 3.17 比较不同版本 bundle 与 unbundle 的时间效率和空间效率, 这些不同的版本用到了不同的头部信息与尾部信息, 对程序的性能与复杂性之间的折衷进行评价.

3.4　多行记录

到目前为止遇到的记录都是由单行组成的, 然而, 还有大量的数据, 其每一条记录都由多行组成, 比如地址薄

```
Adam Smith
1234 Wall St., Apt. 5C
New York, NY 10021
212 555-4321
```

或参考文献

```
Donald E. Knuth
The Art of Computer Programming
Volume 2: Seminumerical Algorithms, Second Edition
Addison-Wesley, Reading, Mass.
1981
```

或个人笔记

```
Chateau Lafite Rothschild 1947
12 bottles @ 12.95
```

如果大小合适, 结构也很规范, 那么创建并维护这些信息相对来说还是比较容易的, 在效果上, 每一条记录都等价于一张索引卡片. 与单行数据相比, 使用 awk 处理多行数据所付出的工作量只是稍微多了一点. 我们将展示几种处理多行数据的方法.

由空行分隔的记录

假设我们有一本地址薄, 其每一条记录的前面 4 行分别是名字, 街道地址, 城市和州, 在这 4 行之后, 可能包含一行额外的信息, 记录之间由一行空白行分开:

```
Adam Smith
1234 Wall St., Apt. 5C
New York, NY 10021
212 555-4321

David W. Copperfield
221 Dickens Lane
Monterey, CA 93940
408 555-0041
work phone 408 555-6532
Mary, birthday January 30

Canadian Consulate
555 Fifth Ave
New York, NY
212 586-2400
```

如果记录是由空白行分隔的, 那么它们可以被直接处理: 若记录分隔符 RS 被设置成空值 (RS=""), 则每一个行块都被当成一个记录, 于是

```
BEGIN { RS = "" }
/New York/
```

打印所有的, 含有 New York 的记录, 而不管这个记录有多少行:

```
Adam Smith
1234 Wall St., Apt. 5C
New York, NY 10021
212 555-4321
Canadian Consulate
555 Fifth Ave
New York, NY
212 586-2400
```

如果记录按照这种方式打印出来, 则输出记录之间是不会有空白行的, 输入格式并不会被保留下来. 为了解决这个问题, 最简单的办法是把输出记录分隔符 ORS 设置成 \n\n:

```
BEGIN { RS = ""; ORS = "\n\n" }
/New York/
```

假设我们想要输出 Smith's 的全名和他的电话号码 (也就是第 1 行以 Smith 结尾的记录的第 1 行与第 4 行), 如果每一行都表示一条记录, 那么就比较容易, 只要把 FS 设置成 \n 即可: 84

```
BEGIN           { RS = ""; FS = "\n" }
$1 ~ /Smith$/   { print $1, $4 } # name, phone
```

程序的输出是

```
Adam Smith 212 555-4321
```

前面提过, 不管 FS 的值是什么, 换行符总是多行记录的字段分隔符之一. 如果 RS 被设置成 "", 则默认的字段分隔符就是空格符, 制表符, 以及换行符; 如果 FS 是 \n, 则换行符就是唯一的字段分隔符.

处理多行记录

如果已经有一个程序可以以行为单位对输入进行处理, 那么我们只需要再写 2 个 awk 程序, 就可以把原来的程序应用到多行记录上. 第 1 个程序把多行记录组合成单行记录, 然后再由已存在的程序进行处理, 最后, 第 2 个程序再把输出转换成多行格式. (我们假设行的长度不会超过 awk 的上限)

为了使过程更加具体, 现在让我们用 Unix 命令 sort 对地址薄进行排序, 下面的程序 pipeline 按照姓氏对输入进行排序:

```
# pipeline to sort address list by last names

awk '
BEGIN { RS = ""; FS = "\n" }
      { printf("%s!!#", x[split($1, x, " ")])
        for (i = 1; i <= NF; i++)
            printf("%s%s", $i, i < NF ? "!!#" : "\n")
      }
' |
sort |
awk '
BEGIN { FS = "!!#" }
      { for (i = 2; i <= NF; i++)
            printf("%s\n", $i)
        printf("\n")
      }
'
```

第 1 个程序中, 函数 split($1, x, " ") 把每个记录的第 1 行切割并保存到数组 x 中, 返回元素的个数, 于是, 姓氏保存在元素 x[split($1, x, " ")] 中 (前提是记录的第 1 行的最后一个单词确实是姓氏). 对每一条多行记录, 第 1 个程序都会创建一个单行记录, 记录包括姓氏, 后面跟着字符串 !!#, 再后面是原多行记录的各个字段 (字段之间也是通过字符串 !!# 分隔). 只要是输入数据中没有出现的, 并且在排序时可以排在输入数据之前的字符串, 都可以用来代替 !!#. sort 之后的程序通过分隔符 !!# 识别原来的字段, 并重构出多行记录.

Exercise 3.18 修改第 1 个 awk 程序: 检查输入数据中是否包含魔术字符串 !!#.

带有头部和尾部的记录

有时候, 记录通过一个头部信息与一个尾部信息来识别, 而不是字段分隔符. 考虑一个简单的例子, 仍然是地址薄, 不过每个记录都带有一个头部信息, 该信息指出了记录的某些特征 (比如职业), 跟在头部后面的是名字, 每条记录 (除了最后一条) 都由一个尾部结束, 尾部由一个空白行组成:

```
accountant
Adam Smith
1234 Wall St., Apt. 5C
New York, NY 10021

doctor - ophthalmologist
Dr. Will Seymour
798 Maple Blvd.
Berkeley Heights, NJ 07922

lawyer
David W. Copperfield
221 Dickens Lane
Monterey, CA 93940

doctor - pediatrician
Dr. Susan Mark
600 Mountain Avenue
Murray Hill, NJ 07974
```

为了打印所有医生的记录, 范围模式是最简单的办法:

```
/^doctor/, /^$/
```

范围模式匹配以 doctor 开始, 以空白行结束的记录 (/^$/ 匹配一个空白行).

为了从输出中移除掉头部信息, 我们可以用

```
/^doctor/ { p = 1; next }
p == 1
/^$/      { p = 0; next }
```

这个程序使用了一个变量来控制行的打印, 如果当前输入行包含有期望的头部信息, 则 p 被设置为 1, 随后的尾部信息将 p 重置为 0 (也就是 p 的初始值). 因为仅当 p 为 1 时才会把当前输入行打印出来, 所以程序只打印记录的主体部分与尾部, 而选择其他输出组合反而比较简单.

名字-值

86

在某些应用中, 数据可能包含更复杂的结构[①], 例如, 地址可能含有国家名称, 也可能不包括街道地址.

[①] 原文为 In some applications data may have more structure than can be captured by a sequence of unformatted lines.

处理结构化数据的一种方法是为记录的每一个字段加上一个名字或关键词, 例如, 我们有可能如此组织一本支票薄:

```
check 1021
to Champagne Unlimited
amount 123.10
date 1/1/87

deposit
amount 500.00
date 1/1/87

check 1022
date 1/2/87
amount 45.10
to Getwell Drug Store
tax medical

check 1023
amount 125.00
to International Travel
date 1/3/87

amount 50.00
to Carnegie Hall
date 1/3/87
check 1024
tax charitable contribution

to American Express
check 1025
amount 75.75
date 1/5/87
```

我们仍然使用多行记录, 记录之间用一个空白行分隔, 但是在记录内部, 每一个数据都是自描述的: 每一个字段都由一个条目名称, 一个制表符, 及信息组成. 这意味着不同的记录可以包含不同的字段, 即使是类似的字段, 其排列顺序也可以不一样.

处理这种数据的方法之一是把它们都当作单行数据, 但是要注意空白行被当作分隔符. 每一行都指出了字段的名称及其所对应的值, 但是它们之间并没有以其他方式相联系[①]. 比如说, 如果我们想要计算存款与支票的总额, 只需要扫描存款项与支票项即可:

87

```
# check1 - print total deposits and checks
```

[①] 原文为 Each line identifies the value it corresponds to, but they are not otherwise connected.

```
/^check/   { ck = 1; next }
/^deposit/ { dep = 1; next }
/^amount/  { amt = $2; next }
/^$/       { addup() }

END        { addup()
             printf("deposits $%.2f, checks $%.2f\n",
                 deposits, checks)
           }

function addup() {
    if (ck)
        checks += amt
    else if (dep)
        deposits += amt
    ck = dep = amt = 0
}
```

输出是

```
deposits $500.00, checks $418.95
```

程序非常简单,只要输入数据格式正确,不管记录中的条目以何种顺序出现,程序都能正确得工作.但是程序也很脆弱,它需要非常认真地初始化,以及对文件结束标志的处理.我们还有另一种方案可供选择,那就是一次读取一条记录,当需要时再对记录的条目进行挑选.下面的程序也是对存款与支票进行求和,但它使用了一个函数,这个函数提取具有指定名字的条目的值:

```
# check2 - print total deposits and checks

BEGIN           { RS = ""; FS = "\n" }
/(^|\n)deposit/ { deposits += field("amount"); next }
/(^|\n)check/   { checks += field("amount"); next }
END             { printf("deposits $%.2f, checks $%.2f\n",
                       deposits, checks)
                }

function field(name,   i,f) {
    for (i = 1; i <= NF; i++) {
        split($i, f, "\t")
        if (f[1] == name)
            return f[2]
    }
```

```
        printf("error: no field %s in record\n%s\n", name, $0)
    }
```

函数 field(s) 在当前记录中搜索名字是 s 的条目, 如果找到, 就把该项的值返回.

第 3 种方案是把记录的每一个字段都分割到一个关联数组中, 然后再对值进行访问. 下面将要介绍的程序以一种更加紧凑的方式打印支票信息: 88

```
  1/1/87  1021  $123.10  Champagne Unlimited
  1/2/87  1022   $45.10  Getwell Drug Store
  1/3/87  1023  $125.00  International Travel
  1/3/87  1024   $50.00  Carnegie Hall
  1/5/87  1025   $75.75  American Express
```

程序的代码是

```
# check3 - print check information

BEGIN { RS = ""; FS = "\n" }
/(^|\n)check/ {
    for (i = 1; i <= NF; i++) {
        split($i, f, "\t")
        val[f[1]] = f[2]
    }
    printf("%8s %5d %8s  %s\n",
        val["date"],
        val["check"],
        sprintf("$%.2f", val["amount"]),
        val["to"])
    for (i in val)
        delete val[i]
}
```

利用 sprintf, 我们在总额的前面加上了美元符, 然后 printf 再对字符串进行右对齐后输出.

Exercise 3.19 写一个命令 lookup x y, 该命令从已知的文件中打印所有符合条件的多行记录, 条件是记录含有名字为 x 且值为 y 的项.

3.5 小结

在这一章, 我们展示了多种不同种类的数据处理程序, 它们包括: 从地址薄中获取信息, 从数值数据中计算简单的统计信息, 检查程序或数据的有效性, 等等. 使用 awk 完成这些工作非常简单, 其中原因是多方面的: 模式–动作 模型非常适合这种类型的工作, 可调整的字段与记录分隔符可以适应不同格式与形状的输入数据, 关联数组无论是存储数值, 还是字符串都非常方便, 像 split 与 substr 这样的函数擅长于文本数据的挑选, 而 printf 则是一个灵活的格式化工具. 在下面的章节里, 我们将会看到这些功能的更进一步的应用.

第四章　报表与数据库

89

这章展示如何使用 awk 从文件中提取信息, 并生成报表, 我们把重点放在表格数据, 但是同样的技术也可以用在更加复杂的输入格式上. 本章的主题是开发一个可以与其他程序配合使用的程序. 我们将会看到大量的工作中会经常遇到的数据处理问题, 这些问题很难一步解决, 但是如果多次遍历数据, 就相对比较容易一些.

本章的第一部分讨论如何扫描单个文件来生成报表, 虽然控制报表的最终格式的确需要花点心思, 但是其实扫描步骤也是挺复杂的. 第二部分讨论如何从多个相关的文件中收集数据, 我们考虑用一种比较一般的方法来解决这个问题, 基本思想是把文件组看成是关系数据库, 这样做的好外是字段可以用名字来标识, 而不是数字.

4.1　报表生成

Awk 可以从文件中挑选数据, 并将挑选到的数据格式化成报表. 我们将使用一个三步骤过程来生成报表: 准备, 排序, 格式化. 准备步骤包括选择数据, 可能的话还会对数据进行一些运算, 进而得到期望的信息; 如果我们想让数据按照某种特定的顺序排列, 就必须使用排序步骤, 排序操作可以通过将准备阶段的输出输送给系统的排序命令来完成; 格式化操作由第 2 个 awk 程序来完成, 它根据已排序的数据生成报表. 为了详细说明, 在这一节我们利用第 二 章的文件 countries 来生成几张报表.

一个简单的报表

假设我们想要一张报表, 这张表包含了每个国家的人口, 面积, 及人口密度. 我们还希望国家按照所在的大洲进行分组, 大洲按照字母顺序排列, 大洲相同的国家按照人口密度的降序排列, 就像这样: 90

CONTINENT	COUNTRY	POPULATION	AREA	POP. DEN.
Asia	Japan	120	144	833.3
Asia	India	746	1267	588.8
Asia	China	1032	3705	278.5
Asia	USSR	275	8649	31.8
Europe	Germany	61	96	635.4
Europe	England	56	94	595.7
Europe	France	55	211	260.7
North America	Mexico	78	762	102.4
North America	USA	237	3615	65.6
North America	Canada	25	3852	6.5

```
South America   Brazil          134        3286        40.8
```

生成报表的前两个阶段由程序 prep1 完成, 当文件 countries 作为输入时, prep1 提取并计算相关的信息, 并对其进行排序:

```
# prep1 - prepare countries by continent and pop. den.

BEGIN { FS = "\t" }
      { printf("%s:%s:%d:%d:%.1f\n",
            $4, $1, $3, $2, 1000*$3/$2) | "sort -t: +0 -1 +4rn"
      }
```

输出是一系列的行, 每行都包括 5 个字段, 用冒号分隔, 从左至右, 字段依次表示大洲, 国家, 人口, 面积, 以及人口密度:

```
Asia:Japan:120:144:833.3
Asia:India:746:1267:588.8
Asia:China:1032:3705:278.5
Asia:USSR:275:8649:31.8
Europe:Germany:61:96:635.4
Europe:England:56:94:595.7
Europe:France:55:211:260.7
North America:Mexico:78:762:102.4
North America:USA:237:3615:65.6
North America:Canada:25:3852:6.5
South America:Brazil:134:3286:40.8
```

程序 prep1 把输出直接输送给 sort 命令, 参数 -t 告诉 sort 把冒号作为字段分隔符, +0 -1 表示把第 1 个字段作为排序的主键, 参数 +4rn 表示把第 5 个字段作为次要主键, 按照数值的逆序进行排序. (在 6.3 节, 我们将展示一个生成排序的程序, 这个程序可以从单词描述中生成排序所需的参数列表)

如果你的系统不支持管道, 那就把 sort 命令删除, 使用 print > *file* 直接输出到文件中, 然后再利用单独的步骤对文件排序, 这个方法适用于本章的所有例子.

现在我们已经完成了三个步骤中的前两个: 准备与排序, 现在所要做的是把数据格式化成我们想要的报表格式, 程序 form1 做的正是这个工作:

91

```
# form1 - format countries data by continent, pop. den.

BEGIN { FS = ":"
        printf("%-15s %-10s %10s %7s %12s\n",
            "CONTINENT", "COUNTRY", "POPULATION",
            "AREA", "POP. DEN.")
      }
      { printf("%-15s %-10s %7d %10d %10.1f\n",
            $1, $2, $3, $4, $5)
      }
```

期望中的报表可以通过键入

```
awk -f prep1 countries | awk -f form1
```

来得到.

prep1 中 sort 的参数非常古怪, 我们可以通过格式化输出, 使得 sort 不再需要任何参数, 然后再让格式化程序对行重新格式化即可. 默认情况下, sort 对输入数据按照字母顺序进行排列, 但是在最终的报表中, 输出首先按照大洲的字母顺序排列, 然后再按人口密度的逆序排列. 为了避免让 sort 带上参数, 准备程序可以预先在每一行的开始处放置一个分量, 分量的大小依赖于大洲的字母顺序与人口密度, 使得按照这个量进行排序时, 排序结果是正确的. 分量的一种可取的表示方法是大洲的名字, 后面再跟着人口密度的倒数, 见程序 prep2:

```
# prep2 - prepare countries by continent, inverse pop. den.

BEGIN { FS = "\t"}
      { den = 1000*$3/$2
        printf("%-15s:%12.8f:%s:%d:%d:%.1f\n",
            $4, 1/den, $1, $3, $2, den) | "sort"
      }
```

当 countries 作为输入时, prep2 的输出是:

```
Asia           :  0.00120000:Japan:120:144:833.3
Asia           :  0.00169839:India:746:1267:588.8
Asia           :  0.00359012:China:1032:3705:278.5
Asia           :  0.03145091:USSR:275:8649:31.8
Europe         :  0.00157377:Germany:61:96:635.4
Europe         :  0.00167857:England:56:94:595.7
Europe         :  0.00383636:France:55:211:260.7
North America  :  0.00976923:Mexico:78:762:102.4
North America  :  0.01525316:USA:237:3615:65.6
North America  :  0.15408000:Canada:25:3852:6.5
South America  :  0.02452239:Brazil:134:3286:40.8
```

格式 %-15s 对大洲名来说已经足够宽了, %12.8f 对人口密度的倒数来说, 覆盖范围也已足够. 最终的格式化程序类似于 form1, 但是忽略了第 2 个字段. 为了简化排序程序的选项而特意制造一个排序键, 这种技巧非常常见, 我们会在第 五 章的索引程序中再次用到. 92

如果我们想要一个更加精美的输出, 其只打印大洲名字一次, 那么我们可以使用程序 form2:

```
# form2 - format countries by continent, pop. den.

BEGIN { FS = ":"
        printf("%-15s %-10s %10s %7s %12s\n",
            "CONTINENT", "COUNTRY", "POPULATION",
            "AREA", "POP. DEN.")
```

```
}
{ if ($1 != prev) {
      print ""
      prev = $1
  } else
      $1 = ""
  printf("%-15s %-10s %7d %10d %10.1f\n",
      $1, $2, $3, $4, $5)
}
```

执行程序的命令行是

```
awk -f prep1 countries | awk -f form2
```

程序的输出是

```
CONTINENT       COUNTRY    POPULATION    AREA    POP. DEN.

Asia            Japan          120        144      833.3
                India          746       1267      588.8
                China         1032       3705      278.5
                USSR           275       8649       31.8

Europe          Germany         61         96      635.4
                England         56         94      595.7
                France          55        211      260.7

North America   Mexico          78        762      102.4
                USA            237       3615       65.6
                Canada          25       3852        6.5

South America   Brazil         134       3286       40.8
```

格式化程序 form2 是一个 “control-break” 程序, 变量 prev 跟踪大洲的名字, 只有当大洲名字变化时才会打印出来. 在下一节, 我们将会看到更复杂的 “control-break” 程序.

更复杂的报表

典型的商业报表比我们现在看到的具有更多的内容 (至少在形式上), 为了详细说明, 假设我们需要为每一个大洲作一个汇总, 以及计算每一个国家占总人口与总面积的比重, 我们需要新增一个标题, 以及更多的列表头:

93

```
Report No. 3       POPULATION, AREA, POPULATION DENSITY         January 1, 1988

 CONTINENT      COUNTRY        POPULATION              AREA           POP. DEN.
```

		Millions of People	Pct. of Total	Thousands of Sq. Mi.	Pct. of Total	People per Sq. Mi.
Asia	Japan	120	4.3	144	0.6	833.3
	India	746	26.5	1267	4.9	588.8
	China	1032	36.6	3705	14.4	278.5
	USSR	275	9.8	8649	33.7	31.8
TOTAL for Asia		2173	77.1	13765	53.6	
Europe	Germany	61	2.2	96	0.4	635.4
	England	56	2.0	94	0.4	595.7
	France	55	2.0	211	0.8	260.7
TOTAL for Europe		172	6.1	401	1.6	
North America	Mexico	78	2.8	762	3.0	102.4
	USA	237	8.4	3615	14.1	65.6
	Canada	25	0.9	3852	15.0	6.5
TOTAL for North America		340	12.1	8229	32.0	
South America	Brazil	134	4.8	3286	12.8	40.8
TOTAL for South America		134	4.8	3286	12.8	
GRAND TOTAL		2819	100.0	25681	100.0	

我们仍然可以使用 准备-排序-格式化 三步骤策略来生成这张报表, prep3 从文件 countries 中准备并排序必要的信息:

```
# prep3 - prepare countries data for form3

BEGIN  { FS = "\t" }
pass == 1 {
    area[$4] += $2
    areatot += $2
    pop[$4] += $3
    poptot += $3
}
pass == 2 {
    den = 1000*$3/$2
    printf("%s:%s:%s:%f:%d:%f:%f:%d:%d\n",
        $4, $1, $3, 100*$3/poptot, $2, 100*$2/areatot,
```

```
            den, pop[$4], area[$4]) | "sort -t: +0 -1 +6rn"
    }
```

这个程序需要遍历输入数据两次, 第一次遍历累加每个大洲的面积与人口数, 并分别保存到数组 area 与 pop 中, 同时计算总面积与总人口数, 分别保存在变量 areatot 与 poptot 中. 第二次遍历对每个国家的统计结果进行格式化, 并输送给 sort. 两次遍历通过变量 pass 控制, 其值可以通过命令行设置: 94

```
awk -f prep3 pass=1 countries pass=2 countries
```

prep3 产生的输出, 其每一行都由 9 个字段组成, 字段之间用冒号分隔, 这些字段包括:

大洲
国家
国家人口
人口所占的比重
国土面积
面积所占的比重
人口密度
该国所在的大洲的总人口
该国所在的大洲的面积

注意 sort 的命令行参数, 该命令行参数使得排序后的记录先按照第 1 个字段的字母顺序排列, 再按第 7 个字段的数值形式的逆序排列.

键入下面的命令行, 就可以生成前面那张精美的报表 Report No. 3:

```
awk -f prep3 pass=1 countries pass=2 countries | awk -f form3
```

其中, 程序 form3 的源代码是

```
# form3 - format countries report number 3

BEGIN  {
    FS = ":"; date = "January 1, 1988"
    hfmt = "%36s %8s %12s %7s %12s\n"
    tfmt = "%33s %10s %10s %9s\n"
    TOTfmt = "   TOTAL for %-13s%7d%11.1f%11d%10.1f\n"
    printf("%-18s %-40s %19s\n\n", "Report No. 3",
      "POPULATION, AREA, POPULATION DENSITY", date)
    printf(" %-14s %-14s %-23s %-14s %-11s\n\n",
      "CONTINENT", "COUNTRY", "POPULATION", "AREA", "POP. DEN.")
    printf(hfmt, "Millions ", "Pct. of", "Thousands ",
                 "Pct. of", "People per")
    printf(hfmt, "of People", "Total ", "of Sq. Mi.",
                 "Total ", "Sq. Mi. ")
    printf(hfmt, "---------", "-------", "----------",
```

```
                    "-------", "----------")
}
{   if ($1 != prev) { # new continent
        if (NR > 1)
            totalprint()
        prev = $1     # first entry for continent
        poptot = $8;  poppct = $4
        areatot = $9; areapct = $6
    } else {          # next entry for continent
        $1 = ""
        poppct += $4; areapct += $6
    }
    printf(" %-15s%-10s %6d %10.1f %10d %9.1f %10.1f\n",
        $1, $2, $3, $4, $5, $6, $7)
    gpop += $3;  gpoppct += $4
    garea += $5; gareapct += $6
}

END {
    totalprint()
    printf(" GRAND TOTAL %20d %10.1f %10d %9.1f\n",
        gpop, gpoppct, garea, gareapct)
    printf(tfmt, "=====", "======", "=====", "======")
}

function totalprint() {     # print totals for previous continent
    printf(tfmt, "----", "-----", "-----", "-----")
    printf(TOTfmt, prev, poptot, poppct, areatot, areapct)
    printf(tfmt, "====", "=====", "=====", "=====")
}
```


除了格式化, form3 累加并打印每个大洲的汇总信息. 除此之外, 它还会累加总人口, 总人口比重, 总面积, 总面积比重, 这些信息在 END 中被打印出来.

form3 在打印完每个大洲的汇总信息之后, 再打印合计信息, 但是一般情况下, 除非读取到一个新的大洲, 否则它不会知道是否已经处理完所有的条目, 解决这种“我们已经走得太远了 (We've gone too far)”问题是 control-break 编程的经典例子. 解决办法是在打印之前检查每一个输入行, 判断是否需要为前一个数据组生成合计信息, 同样的检查也出现在了 END 中, 所以计算工作最好用一个单独的函数来完成. 如果层次只有一层, 那么 control-break 是一种非常简单且有效的方法, 但是当层次加深时, 事情就会变得很糟.

正如上面的程序所呈现得那样, 复杂的格式化工作可以通过多个 awk 程序的组合来完成, 但是为了打印出适当的行, 我们必须精心地计算字符并编写 printf 语句, 这种工作其实非常乏味, 尤其是其中的

某些部分需要进行修改时.

一种可选的方案是让某个程序去计算量的大小, 然后再根据你的需求进行调整[①]. 为打印程序写一个 awk 程序, 该程序对简单的表格进行格式化 — 这是一种可行的做法, 我们待会儿再回来讨论. 因为用的是 Unix 系统与排版程序, 所以我们可以用一些已经存在的工具 — tbl 程序可以对表格进行格式化. 程序 form4 非常类似于 form3, 但是它没有用于控制列宽度的魔数, 相反, 它生成了一些 tbl 命令与表格数据, 不同列的数据之间用制表符分隔, 剩下的工作由 tbl 完成. (如果你对 tbl 不太熟悉, 大可不必理会这些细节)

96

```
# form4 - format countries data for tbl input

BEGIN  {
    FS = ":"; OFS = "\t"; date = "January 1, 1988"
    print ".TS\ncenter;"
    print "l c s s s r s\nl\nl l c s c s c\nl l c c c c c."
    printf("%s\t%s\t%s\n\n", "Report No. 3",
        "POPULATION, AREA, POPULATION DENSITY", date)
    print "CONTINENT", "COUNTRY", "POPULATION",
          "AREA", "POP. DEN."
    print "", "", "Millions", "Pct. of", "Thousands",
          "Pct. of", "People per"
    print "", "", "of People", "Total", "of Sq. Mi.",
          "Total", "Sq. Mi."
    print "\t\t_\t_\t_\t_\t_"
    print ".T&\nl l n n n n n."
}

{    if ($1 != prev) {  # new continent
        if (NR > 1)
            totalprint()
        prev = $1
        poptot = $8;  poppct = $4
        areatot = $9; areapct = $6
    } else {            # next entry for current continent
        $1 = ""
        poppct += $4; areapct += $6
    }
    printf("%s\t%s\t%d\t%.1f\t%d\t%.1f\t%.1f\n",
        $1, $2, $3, $4, $5, $6, $7)
    gpop += $3;  gpoppct += $4
    garea += $5; gareapct += $6
```

[①] 原文为 An alternative is to let a program compute how big things are, then do the positioning for you.

```
}

END {
    totalprint()
    print ".T&\nl s n n n n n."
    printf("GRAND TOTAL\t%d\t%.1f\t%d\t%.1f\n",
        gpop, gpoppct, garea, gareapct)
    print "", "=", "=", "=", "=", "="
    print ".TE"
}

function totalprint() {    # print totals for previous continent
    print ".T&\nl s n n n n n."
    print "", "_", "_", "_", "_", "_"
    printf("   TOTAL for %s\t%d\t%.1f\t%d\t%.1f\n",
        prev, poptot, poppct, areatot, areapct)
    print "", "=", "=", "=", "=", "="
    print ".T&\nl l n n n n n."
}
```

97

如果把 form4 的输出输送给 tbl, 输出的表格是:

Report No. 3　POPULATION, AREA, POPULATION DENSITY　January 1, 1988

CONTINENT	COUNTRY	POPULATION		AREA		POP. DEN.
		Millions of People	Pct. of Total	Thousands of Sq. Mi.	Pct. of Total	People per Sq. Mi.
Asia	Japan	120	4.3	144	0.6	833.3
	India	746	26.5	1267	4.9	588.8
	China	1032	36.6	3705	14.4	278.5
	USSR	275	9.8	8649	33.7	31.8
TOTAL for Asia		2173	77.1	13765	53.6	
Europe	Germany	61	2.2	96	0.4	635.4
	England	56	2.0	94	0.4	595.7
	France	55	2.0	211	0.8	260.7
TOTAL for Europe		172	6.1	401	1.6	
North America	Mexico	78	2.8	762	3.0	102.4
	USA	237	8.4	3615	14.1	65.6
	Canada	25	0.9	3852	15.0	6.5
TOTAL for North America		340	12.1	8229	32.0	
South America	Brazil	134	4.8	3286	12.8	40.8
TOTAL for South America		134	4.8	3286	12.8	
GRAND TOTAL		2819	100.0	25681	100.0	

如果可能的话，我们建议构造一个程序来格式化表格，实现一个像 tbl 这样复杂的程序的确很有野心，但不妨让我们从一个小程序开始：这个程序以左对齐的方式在列中打印条目，如果是数值，则右对齐，并且在同一列中，每一行的列宽度都是相同的（都等于该列宽度的最大值）. 如果给定头部与输入文件 countries，程序的输出是[1]：

```
COUNTRY    AREA   POPULATION   CONTINENT
USSR       8649        275     Asia
Canada     3852         25     North America
China      3705       1032     Asia
USA        3615        237     North America
Brazil     3286        134     South America
India      1267        746     Asia
Mexico      762         78     North America
France      211         55     Europe
Japan       144        120     Asia
Germany      96         61     Europe
England      94         56     Europe
```

程序的实现代码非常紧凑：

98

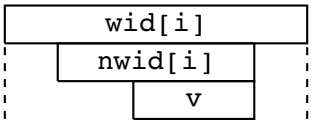
```
# table - simple table formatter

BEGIN {
    FS = "\t"; blanks = sprintf("%100s", " ")
    number = "^[+-]?([0-9]+[.]?[0-9]*|[.][0-9]+)$"
}

{   row[NR] = $0
    for (i = 1; i <= NF; i++) {
        if ($i ~ number)
            nwid[i] = max(nwid[i], length($i))
        wid[i] = max(wid[i], length($i))
    }
}

END {
    for (r = 1; r <= NR; r++) {
        n = split(row[r], d)
        for (i = 1; i <= n; i++) {
            sep = (i < n) ? "   " : "\n"
            if (d[i] ~ number)
```

[1] countries 原来没有头部，需要手工加上 —— 译者注

```
                printf("%" wid[i] "s%s", numjust(i,d[i]), sep)
            else
                printf("%-" wid[i] "s%s", d[i], sep)
        }
    }
}

function max(x, y) { return (x > y) ? x : y }

function numjust(n, s) {   # position s in field n
    return s substr(blanks, 1, int((wid[n]-nwid[n])/2))
}
```

第一次遍历记录数据与每列的最大宽度, 第二次遍历 (位于 END) 在适当的位置打印每一项. 对字母项进行左对齐操作起来比较容易: 我们使用 wid[i] (第 i 列的最大宽度) 为 printf 构造格式字符串, 比如说, 如果列的最大宽度是 10, 则第 i 列的格式字符串就是 %-10s (假设该列是字母项).

如果是数字项, 则要多做点工作: 第 i 列的条目 v 需要右对齐, 就像这样:



v 左边的空格数是 (wid[i]-nwid[i])/2, 所以 numjust 会在 v 的末尾拼接这么多的空格, 然后再按照 %10s 的格式输出 (假设该列的最大宽度是 10). 99

Exercise 4.1 修改 form3 与 form4: 从别处获取日期, 而不是将日期硬编码到代码中.

Exercise 4.2 由于四舍五入, 由 form3 与 form4 打印的项并不总是等于对应列的小计, 你会如何修正这个问题?

Exercise 4.3 表格格式化程序假定所有数字的小数部分的位数都是相同的, 修改它, 使得即使这个假定不成立, 程序也可以正确地工作.

Exercise 4.4 增强 table 的功能, 增强后的 table 允许输入数据中出现一个格式说明行序列, 这个序列说明了每一列如何格式化随后的数据. [1] (tbl 就是这样控制输出格式的)

4.2　打包的查询与报表

如果某个查询经常被访问, 比较方便的做法是把它打包到一个命令中, 这样在下次执行的时候, 就可以少打点字. 假设我们想要查询某个国家的人口, 面积, 以及人口密度, 比如说 Canada, 则可以键入 (假定用的是类 Unix 的 shell):

[1] 原文为 Enhance table to permit a sequence of specification lines that tell how the subsequent data is to be formatted in each column.

```
awk '
BEGIN { FS = "\t" }
$1 ~ /Canada/ {
    printf("%s:\n", $1)
    printf("\t%d million people\n", $3)
    printf("\t%.3f million sq. mi.\n", $2/1000)
    printf("\t%.1f people per sq. mi.\n", 1000*$3/$2)
}
' countries
```

输出是

```
Canada:
    25 million people
    3.852 million sq. mi.
    6.5 people per sq. mi.
```

现在, 如果我们想要查询其他国家的信息, 在执行每次查询前都需要修改国家的名称, 但是更方便的做法是把程序放入一个可执行文件中, 比如就叫 info, 查询时只要输入

```
info Canada
info USA
...
```

我们可以利用 2.6 节的技术, 把一个国家的名称传递给程序, 或者, 我们也可以利用 shell, 把国家的名称放到适当的位置上:

100

```
awk '
# info - print information about country
#    usage: info country-name

BEGIN { FS = "\t" }
$1 ~ /'$1'/ {
    printf("%s:\n", $1)
    printf("\t%d million people\n", $3)
    printf("\t%.3f million sq. mi.\n", $2/1000)
    printf("\t%.1f people per sq. mi.\n", 1000*$3/$2)
}
' countries
```

程序开头的第二行

```
$1 ~ /'$1'/
```

第一个 $1 指的是当前输入行的第一个字段, 而第二个 $1 (被单引号包围着的) 指的是国家名称参数, 也就是执行 shell 命令 info 时的第一个参数. 第二个 $1 只对 shell 可见, 当命令被执行时, 它被 info 后

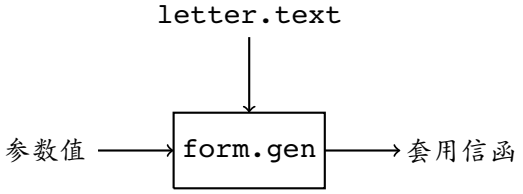
面的字符串所替代. 具体过程是: shell 通过拼接三个字符串来组合成 awk 程序, 这三个字符串其中两个是被一对单引号包围起来的多行文本, 另外一个是 $1, 即 info 的参数. 需要注意的是, 可以把任意一个正则表达式传递给 info, 尤其是, 我们可以只给出国家名称的一部分, 或者一次指定多个国家, 也可以查询到相关的国家信息, 比如

```
info 'Can|USA'
```

Exercise 4.5 修改 info: 参数通过 ARGV 传递进来, 而不是由 shell 进行替换.

套用信函

Awk 也可以用于生成套用信函, 生成时只需要用参数替换掉套用信函中的文本即可:



套用信函的文本内容存放在文件 letter.text 中, 文本中包含了许多参数, 只需要通过参数替换, 就可以生成不同内容的信函[①]. 例如, 下面的文本使用了参数 #1 到 #4, 这些参数分别表示国家名称, 人口, 面积, 以及人口密度:

101

```
Subject: Demographic Information About #1
From: AWK Demographics, Inc.

In response to your request for information about #1,
our latest research has revealed that its population is #2
million people and its area is #3 million square miles.
This gives #1 a population density of #4 people per
square mile.
```

如果输入参数是

```
Canada:25:3.852:6.5
```

输出的的套用信函是:

```
Subject: Demographic Information About Canada
From: AWK Demographics, Inc.

In response to your request for information about Canada,
our latest research has revealed that its population is 25
million people and its area is 3.852 million square miles.
This gives Canada a population density of 6.5 people per
square mile.
```

[①] 原文为 The text contains parameters that will be replaced by a set of parameter values for each form letter that is generated.

程序 form.gen 是套用信函生成器:

```
# form.gen - generate form letters
#   input:  prototype file letter.text; data lines
#   output: one form letter per data line

BEGIN {
    FS = ":"
    while (getline <"letter.text" > 0) # read form letter
    form[++n] = $0
}

{   for (i = 1; i <= n; i++) { # read data lines
        temp = form[i]         # each line generates a letter
        for (j = 1; j <= NF; j++)
            gsub("#" j, $j, temp)
        print temp
    }
}
```

form.gen 的 BEGIN 从文件 letter.text 中读取套用信函的文本, 并存放到数组 form 中, 剩下的工作是读取输入参数, 调用 gsub 将文本中的 #n 替换成对应的参数, 被修改的只是存储在数组 form 中的信函副本, 文件 letter.text 的内容并没有发生改变. 注意程序是如何通过字符串拼接来生成 gsub 的第一个参数的.

102

4.3　关系数据库系统

在这一节, 我们将会描述一个简单的关系型数据库系统, 这个系统的核心包括一个类 awk 查询语言 q, 一个数据字典 relfile, 以及一个查询处理程序 qawk, qawk 的作用是把 q 描述的查询翻译成 awk 程序.

这个系统通过以下三种方式将 awk 扩展成一个数据库语言:

- 字段通过名字来引用, 而不是数字.
- 数据库可以分散在多个文件中, 而不是限定在一个文件上.
- 可以交互性地生成一个查询序列.

使用名字 (而非数字) 引用字段的优势是显而易见的 — $area 看起来比 $2 更加自然, 但是, 把一个数据库分散存储在若干个文件中的优点就没那么容易看出来. 多个文件组成的数据库维护起来更容易, 这主要是因为编辑一个只含有少量字段的文件, 要比编辑一个包含全部字段的文件容易得多. 另外, 利用本节开发的数据库系统, 我们就有可能在不改变数据库访问程序的情况下, 重构数据库. 最后, 对于简单的查询来说, 访问一个小文件要比访问一个大文件高效得多. 另一方面, 当我们往数据库添加数据时, 必须小心修改所有的相关文件, 否则会带来一致性问题.

根据前面的讨论[1], 我们的数据库由一个单独的文件 countries 组成, 文件的每一行都包括四个字段, 字段的名字分别是 country, area, population, 以及 continent. 现在我们打算在数据库中新增一个文件 capitals, 文件的每一行都由两个字段组成 — 国家及其首都:

```
USSR    Moscow
Canada  Ottawa
China   Beijing
USA     Washington
Brazil  Brasilia
India   New Delhi
Mexico  Mexico City
France  Paris
Japan   Tokyo
Germany Bonn
England London
```

与 countries 一样, 字段之间用制表符分开.

从这两个文件出发, 如果我们想要打印位于亚洲的国家, 及其人口与首都, 那就必须扫描这两个文件, 再拼凑出最终的结果. 举例来说, 如果输入数据不是很多, 下面的程序就可以完成这件工作: 103

```
awk ' BEGIN { FS = "\t" }
      FILENAME == "capitals" {
          cap[$1] = $2
      }
      FILENAME == "countries" && $4 == "Asia" {
          print $1, $3, cap[$1]
      }
' capitals countries
```

如果我们只需要输入

```
$continent ~ /Asia/ { print $country, $population, $capital }
```

就可以查询到亚洲国家的相关信息, 那将会非常的方便. 执行这条命令时, 有一个程序会计算出字段的位置, 并将它们组成在一起, 这也是我们在 q 中短语化查询的方式, 接下来马上就会讨论到 q.

自然连接

现在是时候讨论一些术语了. 在关系型数据库中, 一个文件被称为一张 **表** (*table*) 或一个 **关系** (*relation*), 把列叫作 **属性** (*attribute*), 所以我们可以认为表格 capitals 具有属性 country 与 capital.

一个 **自然连接** (*natural join*) (或简称为 **连接**) 指的是一个运算, 它将两张表以公共属性为基础组合成一张表, 这张表包含了两张表所有的属性, 但是重复的属性会被移除. 如果我们把 countries 与 capitals 这两张表作自然连接, 我们就会得到一张新表 (姑且称为 cc), 它的属性包括:

[1] 原文为 Up to this point

```
country, area, population, continent, capital
```

对于每一个同时出现在两张表中的国家来说, 我们都可以在 cc 中找到一行记录, 它包含了国家的名称, 后面跟着面积, 人口, 所在大洲, 以及国家的首都:

```
Brazil   3286    134     South America Brasilia
Canada   3852    25      North America Ottawa
China    3705    1032    Asia                Beijing
England  94      56      Europe              London
France   211     55      Europe              Paris
Germany  96      61      Europe              Bonn
India    1267    746     Asia                New Delhi
Japan    144     120     Asia                Tokyo
Mexico   762     78      North America Mexico City
USA      3615    237     North America Washington
USSR     8649    275     Asia                Moscow
```

我们实现连接运算的方法是根据它们的公共属性进行排序, 如果两张表的某一行的公共属性值相同, 那就把它们合并起来, 就像上面这张表显示的那样. 为了处理涉及到多张表的查询, 我们会先将表格作连接, 然后再对连接后的表作查询, 也就是说, 如果有必要的话, 我们会创建一个临时文件. 为了响应查询: 104

```
$continent ~ /Asia/ { print $country, $population, $capital }
```

我们将表 countries 与表 capitals 作连接, 再将查询应用到连接结果上, 通常来说, 完成这些操作最关键的地方在于将哪些表作连接.

实际的连接操作可以通过 Unix 命令 join 来完成, 但是如果你的系统上没有该命令, 可以用这里提供的一个用 awk 实现的简易版本. 它将两个文件按照各自的第一个字段进行连接, 比如下面两张表

ATT1	ATT2	ATT3
A	w	p
B	x	q
B	y	r
C	z	s

ATT1	ATT4
A	1
A	2
B	3

作连接后得到

ATT1	ATT2	ATT3	ATT4
A	w	p	1
A	w	p	2
B	x	q	3
B	y	r	3

join 不假定输入表格是等长的, 但是它会认为表格是有序的, 互相匹配的输入字段的每一种组合都会在输出中占据一行.

```
# join - join file1 file2 on first field
#   input:  two sorted files, tab-separated fields
#   output: natural join of lines with common first field

BEGIN {
    OFS = sep = "\t"
    file2 = ARGV[2]
    ARGV[2] = ""  # read file1 implicitly, file2 explicitly
    eofstat = 1   # end of file status for file2
    if ((ng = getgroup()) <= 0)
        exit      # file2 is empty
}

{   while (prefix($0) > prefix(gp[1]))
        if ((ng = getgroup()) <= 0)
            exit  # file2 exhausted
    if (prefix($0) == prefix(gp[1]))  # 1st attributes in file1
        for (i = 1; i <= ng; i++)     #     and file2 match
            print $0, suffix(gp[i])   # print joined line
}

function getgroup() { # put equal prefix group into gp[1..ng]
    if (getone(file2, gp, 1) <= 0)    # end of file
        return 0
    for (ng = 2; getone(file2, gp, ng) > 0; ng++)
        if (prefix(gp[ng]) != prefix(gp[1])) {
            unget(gp[ng])    # went too far
            return ng-1
        }
    return ng-1
}

function getone(f, gp, n) {  # get next line in gp[n]
    if (eofstat <= 0) # eof or error has occurred
        return 0
    if (ungot) {      # return lookahead line if it exists
        gp[n] = ungotline
        ungot = 0
        return 1
    }
```

```
        return eofstat = (getline gp[n] <f)
    }

    function unget(s)  { ungotline = s; ungot = 1 }
    function prefix(s) { return substr(s, 1, index(s, sep) - 1) }
    function suffix(s) { return substr(s, index(s, sep) + 1) }
```

105

执行 join 时, 需要向它传递两个参数, 它们都表示输入文件名. 第一个属性值相同的行组成一个行组, 它们从第二个文件中读出, 如果第一个文件的某一行的前缀与某些行组的公共属性值相同, 那么行组中的每一行都会出现在输出中.

函数 getgroup 把下一组前缀相同的行放入数组 gp 中, 它调用 getone 来获取每一行, 如果发现获取到的行不属于本组, 就调用 unget 将它放回. 我们把提取第一个属性值的代码局限在函数 prefix 中, 这样以后改起来也比较方便.

你应该注意一下函数 getone 与 unget 如何撤回一个输入行. 在读取新行之前, getone 检查是否已经有一行被读取并由 unget 存放在某个变量中, 如果是, 则返回该行. 撤回是解决过早遭遇问题 (在一次操作中, 读取了过多的输入[①]) 的方法之一. 在本章早些时候出现的 control-break 程序中, 我们把处理操作延迟了, 而在这里, 通过一对函数, 我们假定不会看到额外的输入.

Exercise 4.6 本节实现的 join 不会进行错误检查, 也不会检查文件是否是有序的. 修复这些问题, 在修复之后, 程序会变得多大?

Exercise 4.7 实现 join 的另一个版本, 它将一个文件整个读入内存, 然后再执行连接操作. 与原来的版本相比, 哪个更简单?

Exercise 4.8 修改 join: 它可以按照输入文件的任意一个字段或字段组来进行连接, 并可以按照任意的顺序, 有选择地输出某些字段.

relfile

106

为了回答关于分散在多张表中的数据库的问题, 我们必须知道每张表中包含什么内容, 我们把这些信息存储在名为 relfile 的文件中 ("rel" 指的是 "relation"). 文件 relfile 包含了数据库中每张表的名字, 属性, 如果表格不存在, 那么文件还包含了构造表格的规则. 文件 relfile 的内容是一系列的表格描述符, 表格描述符具有形式:

tablename
    attribute
    attribute
        ...
    !*command*
        ...

[①] 原文为 reading one too many inputs.

tablename 与 *attribute* 都是字符串, 在 *tablename* 之后是该表包含的属性名列表, 每一个属性名前面都有一个空格或制表符, 属性名之后是可选的命令序列, 命令以感叹号开始, 它说明了如何构造这张表格. 如果一张表不含构造命令, 则说明已经有一个文件以该表的名字命名, 且包含了该表的数据, 这样的表叫做 **基表** (*base table*), 数据被插入到基表中, 并在基表中更新. 在文件 `relfile` 中, 如果某张表在属性名之后出现了构造命令, 则称这张表是 **导出表** (*derived table*), 只有在必要的时候才会构造导出表.

我们使用下面的 `relfile` 来表示扩展了的国家数据库:

```
countries:
    country
    area
    population
    continent
capitals:
    country
    capital
cc:
    country
    area
    population
    continent
    capital
    !sort countries >temp.countries
    !sort capitals >temp.capitals
    !join temp.countries temp.capitals >cc
```

这个文件说明数据库中有两张基表 — `countries` 与 `capitals`, 一张导出表 `cc`, 导出表 `cc` 通过把基表排序并存放在临时文件中, 再对临时文件执行连接操作来生成, 也就是说, `cc` 通过执行

107

```
sort countries >temp.countries
sort capitals >temp.capitals
join temp.countries temp.capitals >cc
```

来生成.

一个 `relfile` 经常包含一张 **通用关系表** (*universal relation*), 它是一张包含了所有属性的表, 是 `relfile` 的最后一张表, 这种做法保证至少有一张表包含了属性的所有可能的组合, 表格 `cc` 就是数据库 countries-capitals 的通用关系表.

一个复杂数据库的优秀设计必须考虑到该数据库可能收到的查询种类, 属性间存在的依赖关系, 但是对于一个小数据库来说, *q* 已经足够快了. 因为数据库的表比较少, 所以很难展现出 `relfile` 设计的精妙之外.

q, 类 awk 查询语言

我们的数据库查询语言 *q* 由单独一行的 awk 程序组成, 但字段名被属性名替代. 查询处理程序 `qawk` 按照下面的步骤来响应一个查询:

1. 判断该查询所包含的属性集;
2. 从 relfile 的第一行开始, 搜索第一张包含了查询中全部属性的表, 如果该表是基表, 则用它作为查询的输入, 如果是导出表, 则构造出该表, 再用它作为查询的输入. (这意味着查询中可能出现的属性组合一定也出现在 relfile 的基表或导出表中);
3. 通过把符号型的字段引用替换成对应的数值型字段引用, 将 q 查询转换成等价的 awk 程序, 这个程序接下来会把步骤 2 决定的表作为输入数据.

q 查询:

```
$continent ~ /Asia/ { print $country, $population }
```

提到了属性 continent, country 与 population, 它们都是第一张表 countries 的属性子集. 查询处理程序把该查询转换成程序

```
$4 ~ /Asia/ { print $1, $3 }
```

并把文件 countries 作为输入数据.

q 查询

```
{ print $country, $population, $capital }
```

包含属性 country, population 与 capital, 它们都是导出表 cc 的属性子集. 于是查询处理程序利用 relfile 列出的构造命令构造出表格 cc, 并将该查询转换成程序

```
{ print $1, $3, $5 }
```

108

程序把刚构造而成的 cc 作为输入数据.

虽然我们用的是“查询”这个词来描述 qawk, 但实际上它也可以用来作计算, 下面这个查询打印面积的平均值:

```
{ area += $area }; END { print area/NR }
```

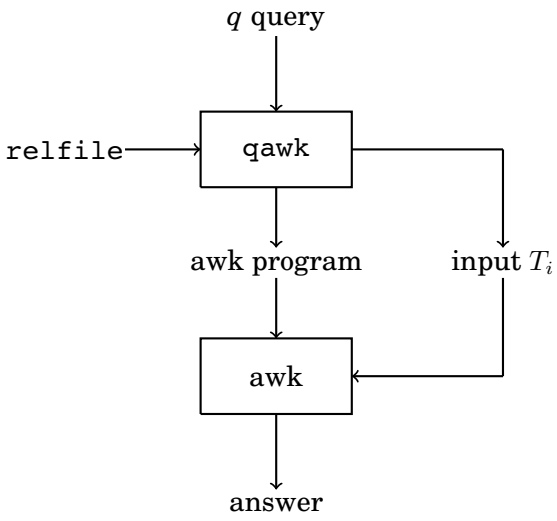
qawk, q-to-awk 翻译器

我们用 qawk 的实现来结束本章, 它把 q 查询转换成等价的 awk 程序.

首先, qawk 读取文件 relfile, 将表名收集到数组 relname 中. 为了构造第 i 张表, 程序把必要的构造命令收集到数组 array 中, 从 cmd[i, 1] 开始存放. 它还把每张表的属性收集到 2 维数组 attr 中, 元素 attr[i,a] 存放的是第 i 张表中, 名为 a 的属性的索引.

然后, qawk 读取一个查询, 判断它会用到哪些属性, 查询中的属性都是形式为 $\$name$ 的字符串. 利用函数 subset, 就可以找到第一张包含了查询中全部属性的表 T_i. 程序通过把属性的索引替换成原始形式来生成 awk 程序, 并执行必要的命令来生成表格 T_i, 再执行新生成的 awk 程序, 把表格 T_i 作为输入数据.

对每一个查询, 第二个步骤都要重复一次, 下面这个流程图展示了 qawk 的框架:



109

这是 qawk 的源代码:

```
# qawk - awk relational database query processor

BEGIN { readrel("relfile") }
/./   { doquery($0) }

function readrel(f) {
    while (getline <f > 0)   # parse relfile
        if ($0 ~ /^[A-Za-z]+ *:/) {     # name:
            gsub(/[^A-Za-z]+/, "", $0)  # remove all but name
            relname[++nrel] = $0
        } else if ($0 ~ /^[ \t]*!/)     # !command...
            cmd[nrel, ++ncmd[nrel]] = substr($0,index($0,"!")+1)
        else if ($0 ~ /^[ \t]*[A-Za-z]+[ \t]*$/)  # attribute
            attr[nrel, $1] = ++nattr[nrel]
        else if ($0 !~ /^[ \t]*$/)      # not white space
            print "bad line in relfile:", $0
}
function doquery(s,   i,j) {
    for (i in qattr)  # clean up for next query
        delete qattr[i]
    query = s    # put $names in query into qattr, without $
    while (match(s, /\$[A-Za-z]+/)) {
        qattr[substr(s, RSTART+1, RLENGTH-1)] = 1
        s = substr(s, RSTART+RLENGTH+1)
    }
    for (i = 1; i <= nrel && !subset(qattr, attr, i); )
        i++
```

```
        if (i > nrel)     # didn't find a table with all attributes
            missing(qattr)
        else {            # table i contains attributes in query
            for (j in qattr)   # create awk program
                gsub("\\$" j, "$" attr[i,j], query)
            for (j = 1; j <= ncmd[i]; j++)  # create table i
                if (system(cmd[i, j]) != 0) {
                    print "command failed, query skipped\n", cmd[i,j]
                    return
                }
            awkcmd = sprintf("awk -F'\t' '%s' %s", query, relname[i])
            printf("query: %s\n", awkcmd)   # for debugging
            system(awkcmd)
        }
    }
    function subset(q, a, r,   i) {  # is q a subset of a[r]?
        for (i in q)
            if (!((r,i) in a))
                return 0
        return 1
    }
    function missing(x,     i) {
        print "no table contains all of the following attributes:"
        for (i in x)
            print i
    }
```

110

Exercise 4.9 如果你的系统不支持 awk 的 system 函数, 请修改 qawk, 修改后的 qawk 把命令序列写入一个或多个文件中, 这些文件可以被单独地执行.

Exercise 4.10 构造导出表的时候, qawk 为每一个命令调用一次 system, 修改 qawk, 修改后的 qawk 把所有的构造表格的命令都收集到一个字符串中, 这样就可以只需要调用一次 system.

Exercise 4.11 修改 qawk, 使得它可以检查即将作为输入的导出表是否已经存在. 如果导出表已经存在, 并且基表在导出表被构造出来之后未曾修改过, 那我们就可以直接使用该导出表, 而不用重新构造. 可以参考第 七 章的程序 make.

Exercise 4.12 提供一种输入并编辑多行查询的方式. 只要对 qawk 作些许修改, 就可以把多行查询收集起来. 编辑的一种可选方案是调用你喜欢的编辑器, 另一种方式是用 awk 写一个简单的编辑器.[①]

[①] 原文为 Provide a way to enter and edit multiline queries. Multiline queries can be collected with minimal changes to qawk. One possibility for editing is a way to invoke your favorite text editor; another is to write a very simple editor in awk itself.

4.4　小结

在这一章, 我们主要学习了如何使用 awk, 按照结构化的方式来访问和打印信息, 与第 三 章的 *ad hoc* 使用方式形成对照. [①]

为了生成表格, "分而治之" 通常是最好的方法: 在一个程序中准备数据, 如果必要的话就对数据排序, 然后用第二个程序对数据进行格式化. Control break 可以通过向前看, 或者撤回输入 (通常是比较优雅的方式) 来处理. (有时候也可以通过管道来完成这些工作, 虽然我们在这章没有用到.) 为了处理格式的细节, 相对于手工计算字符个数, 一种更好的替代方案是用某些程序来完成机械化的部分.

虽然 awk 并不是生产数据库的工具, 但是对于小型的个人数据库来说, 它是一个非常合理的选择, 而且在阐述某些数据库的基本概念方面, 它表现得也非常优秀, 为了论证这两点, 我们开发了 `qawk`.

参考资料

关于数据库有着非常丰富的优秀书籍, 你可以试一下 J. D. Ullman 的 *Principles of Database Systems* (Computer Science Press, 1986).

[①] 原文为 in contrast to the more typical *ad hoc* uses of Chapter 3

第五章　文本处理

111

本章的程序都指向一个共同主题: 文本处理. 示例程序涵盖的范围包括随机单词与句子的生成 (生成的句子可以和用户进行有限的对话), 以及文本处理. 大多数示例程序都很简单, 它们只是起说明作用, 但是, 其中一些文件准备程序的确拥有实际用途.

5.1　随机文本发生器

生成随机数据的程序有多种用途, 这种程序可以用内建函数 rand 创建, 每次调用该函数都会返回一个伪随机数. rand 每次都使用同一个种子数来生成随机数, 所以, 如果你想要得到一个不同的随机数序列, 就必须调用一次 srand(), 它根据当前时间计算出一个种子数, 并用该种子数初始化 rand.

随机选择

每次调用 rand 都会返回一个大于等于 0, 小于 1 的浮点数, 但是一般来说, 更通常的需求是返回一个 1 到 n 之间的随机整数, 我们可以用 rand 来实现:

```
# randint - return random integer x, 1 <= x <= n

function randint(n) {
    return int(n * rand()) + 1
}
```

randint(n) 按比例调整 rand 的返回值, 调整后的值大于等于 0 并且小于 n, 将小数部分截去可以得到 0 至 n-1 的整数, 然后再加 1, 就是 1 到 n 之间的整数.

我们可以用 randint 随机选择一个字母:

112

```
# randlet - generate random lower-case letter

function randlet() {
    return substr("abcdefghijklmnopqrstuvwxyz", randint(26), 1)
}
```

利用 randint, 我们可以从 n 项的数组中随机选择一个元素:

```
print x[randint(n)]
```

一个更有趣的问题是如何从数组中随机选择几项, 并且被选中的项必须按照原来的顺序排列. 举例来说, 如果数组 x 按照升序排列, 则被选中的元素也要按照升序排列.

函数 choose 从数组 A 的前 n 中随机选择 k 个元素, 并按照原来的顺序打印出来:

```
# choose - print in order k random elements from A[1]..A[n]

function choose(A, k, n,    i) {
    for (i = 1; n > 0; i++)
        if (rand() < k/n--) {
            print A[i]
            k--
        }
}
```

在函数内, k 是还需要打印的项的数目, n 是数组中等待检验的元素个数. 打印第 i 个元素的条件是 rand() < k/n, 每输出一个元素, k 就递减一次, 每测试一次 rand() < k/n, n 就递减一次.

Exercise 5.1 测试 rand 的输出是不是真的随机数.

Exercise 5.2 写一个程序, 该程序生成 1 到 n 之间 k 个不同的随机整数, 程序的时间复杂度与 k 成正比.

Exercise 5.3 写一个程序, 该程序生成随机的桥牌手 (bridge hands).

废话生成器

我们的下一个例子是废话生成器 (cliche generator), 它根据已有的废话重新创建一个新的出来. 输入是一个句子集合:

```
A rolling stone:gathers no moss.
History:repeats itself.
He who lives by the sword:shall die by the sword.
A jack of all trades:is master of none.
Nature:abhors a vacuum.
Every man:has a price.
All's well that:ends well.
```

冒号将主语和谓语分开. 废话生成器随机选择一个主语与另一个谓语作组合, 如果运气好的话, 可能会产 113
生很有意思的格言警句:

```
A rolling stone repeats itself.
History abhors a vacuum.
nature repeats itself.
All's well that gathers no moss.
He who lives by the sword has a price.
```

实现代码非常简单:

```
# cliche - generate an endless stream of cliches
#     input:  lines of form subject:predicate
#     output: lines of random subject and random predicate

BEGIN { FS = ":" }
      { x[NR] = $1; y[NR] = $2 }
END   { for (;;) print x[randint(NR)], y[randint(NR)] }

function randint(n) { return int(n * rand()) + 1 }
```

请注意, 程序中的死循环是有意为之.

随机语句

上下文无关语法 (*context-free grammar*) 指的是一组规则, 这组规则定义了如何生成或分析一个语句集合. 每一条规则 (称为 **产生式** (*production*)) 都具有形式:

$$A \longrightarrow B\ C\ D\ ...$$

该产生式的意思是每一个 A 都可以被 "重写" 为 $B\ C\ D\ ....$ 产生式左边的符号 (A) 称为 **非终结符** (*nonterminal*), 它可以被进一步地扩展. 产生式右边的符号可以是非终结符 (可以是多个 A) 或 **终结符** (*terminal*), 终结符指的是不能被扩展的符号. 多个产生式可以共享同一个非终结符, 终结符与非终结符也可以在产生式的右边出现多次.

我们将在第 六 章展示 awk 的部分语法规则, 并利用该规则开发一个语法分析器, 用来分析 awk 程序. 然而在这一章, 我们感兴趣的是规则的生成, 而不是分析. 举例来说, 类似 "the boy walks slowly" 和 "the girl runs very very quickly" 这样的句子可以用下面的语法来描述:

```
Sentence -> Nounphrase Verbphrase
Nounphrase -> the boy
Nounphrase -> the girl
Verbphrase -> Verb Modlist Adverb
Verb -> runs
Verb -> walks
Modlist -> very Modlist
Adverb -> quickly
Adverb -> slowly
```

114

如下所示, 产生式为非终结符生成语句. 假设 Sentence 是起始非终结符, 那么选择一条以该符号作为左部的产生式:

```
Sentence -> Nounphrase Verbphrase
```

接下来, 从右部选择一个非终结符, 比如说 Nounphrase, 然后用以 Nounphrase 作为左部的产生式替换掉 Nounphrase:

```
Sentence -> Nounphrase Verbphrase
         -> the boy Verbphrase
```

不断进行下去, 直到所有的非终结符都被替换掉为止:

```
Sentence -> Nounphrase Verbphrase
         -> the boy Verbphrase
         -> the bov Verb Modlist Adverb
         -> the boy walks very Modlist Adverb
         -> the boy walks very Adverb
         -> the boy walks very quickly
```

Sentence 的最终展开结果是一个句子. 非终结符的推导过程与我们在初级学校所学到的语句图 (sentence-diagram) 刚好相反: 我们现在是把动词短语拆分成动词与副词, 而不是把动词与副词组合成动词短语.

Modlist 的产生式比较有趣. 一条规则是说用 very Modlist 替换 Modlist, 每次使用这条规则都会使句子变长. 幸运地是, 只要运用另一条产生式规则 (该规则用空字符串替换掉 Modlist) 就可以终止潜在的无限循环.

我们现在打算开发一个程序, 该程序根据语法生成语句, 每次生成都从一个指定的非终结符开始. 程序从文件中读取语法规则, 记录下每一个左部出现的次数, 左部所拥有的右部的个数, 以及它们各自的组成成分. 然后, 每输入一个非终结符, 就会为该非终结符生成一个随机语句.

程序使用三个数组来存放语法规则: lhs[A] 给出了非终结符 A 的产生式个数, rhscnt[A, i] 存放的是 A 的第 i 条产生式右部的符号个数, rhslist[A, i, j] 存放的是 A 的第 i 条产生式右部的第 j 个符号. 对于前面提到的语法规则, 三个数组的内容分别是:

lhs:

Sentence	1
Nounphrase	2
Verbphrase	1
etc.	

rhscnt:

Sentence,1	2
Nounphrase,1	2
Nounphrase,2	2
Verbphrase,1	3
etc.	

rhslist:

Sentence,1,1	Nounphrase
Sentence,1,2	Verbphrase
Nounphrase,1,1	the
Nounphrase,1,2	boy
Nounphrase,2,1	the
Nounphrase,2,2	girl
Verbphrase,1,1	Verb
Verbphrase,1,2	Modlist
Verbphrase,1,3	Adverb
etc.	

115

程序的源代码是:

```
# sentgen - random sentence generator
#   input:  grammar file; sequence of nonterminals
#   output: a random sentence for each nonterminal

BEGIN {  # read rules from grammar file
    while (getline < "grammar" > 0)
        if ($2 == "->") {
```

```
            i = ++lhs[$1]              # count lhs
            rhscnt[$1, i] = NF-2       # how many in rhs
            for (j = 3; j <= NF; j++)  # record them
               rhslist[$1, i, j-2] = $j
        } else
            print "illegal production: " $0
}

{   if ($1 in lhs) {  # nonterminal to expand
        gen($1)
        printf("\n")
    } else
        print "unknown nonterminal: " $0
}

function gen(sym,    i, j) {
    if (sym in lhs) {       # a nonterminal
        i = int(lhs[sym] * rand()) + 1   # random production
        for (j = 1; j <= rhscnt[sym, i]; j++) # expand rhs's
            gen(rhslist[sym, i, j])
    } else
        printf("%s ", sym)
}
```

函数 `gen("A")` 为非终结符 A 生成一条语句. 如果前一次的扩展引入了非终结符, 那么函数通过递归调用自身来展开. 请记住, 所有被递归函数用到的临时变量都需要出现在函数的参数列表中, 否则的话就是全局变量, 此时程序就无法正确地工作. 116

我们把右部个数及其组成成分分别存放在两个数组中, 实际上, 不使用下标来编码字段也是可以的, 但并非通过其他语言中的记录或结构体来实现. 举例来说, 数组 `rhscnt[i,j]` 可以是 `rhslist` 的元素, 表示成 `rhslist[i,j,"cnt"]`.[①]

Exercise 5.4 写一套语法规则, 该规则能够生成关于某学科的, 听起来貌似合理的文本 — 学科可以是商业, 政治, 或计算机.[②]

Exercise 5.5 在某些语法规则下, 语句生成程序很有可能落入到这样一种境地: 推导过程越来越长, 却没有停下来的迹象, 添加一条机制, 使得程序可以限制推导过程的长度.

Exercise 5.6 给语法规则加上权重, 使得对同一个非终结符来说, 它的各个展开规则被选中的概率是不同的.

[①] 原文为 We chose to use separate arrays for the right-hand-side counts and components, but it is possible instead to use subscripts to encode different fields, rather like records or structures in other languages. For example, the array `rhscnt[i,j]` could be part of `rhslist`, as `rhslist[i,j,"cnt"]`.

[②] 原文为 Write a grammar for generating plausible-sounding text from a field that appeals to you – business, politics, and computing are all good possibilities.

Exercise 5.7 实现一个非递归的语句生成程序.

5.2　交互式的文本处理

使用 awk 很容易就可以写出一个交互式程序, 我们将通过两个示例程序来阐明基本思路. 第一个程序测试运算能力, 第二个程序测试某一特定领域的相关知识.

技巧测试之运算

下面的程序 arith (最适用于幼儿) 显示一系列加法运算问题, 比如

```
7 + 9 = ?
```

在每个问题之后, 用户输入问题的答案. 如果回答是正确的, 用户就会收到一句赞美, 然后显示下一个问题; 如果回答是错误的, 那么程序会再次请求输入答案; 如果用户没有输入答案, 那么在显示下一个问题之前, 程序输出正确答案.

调用程序的命令行有两种形式:

```
awk -f arith
awk -f arith n
```

如果在 arith 之后有一个参数, 程序将使用该参数限制每个问题的数的最大值. 该参数被读取之后, ARGV[1] 被设置为 "-", 于是程序从标准输入读取用户的回答. 如果没有指定参数, 那么数的最大值就是 10.

117

```
# arith - addition drill
#   usage:  awk -f arith [ optional problem size ]
#   output: queries of the form "i + j = ?"

BEGIN {
    maxnum = ARGC > 1 ? ARGV[1] : 10   # default size is 10
    ARGV[1] = "-"  # read standard input subsequently
    srand()        # reset rand from time of day
    do {
        n1 = randint(maxnum)
        n2 = randint(maxnum)
        printf("%g + %g = ? ", n1, n2)
        while ((input = getline) > 0)
            if ($0 == n1 + n2) {
                print "Right!"
                break
            } else if ($0 == "") {
                print n1 + n2
                break
```

```
            } else
                printf("wrong, try again: ")
        } while (input > 0)
    }

    function randint(n) { return int(rand()*n)+1 }
```

Exercise 5.8 除了加法外, 再新增几种数学运算. 另外, 如果用户的回答是错误的, 显示一条提示信息.

技巧测试之测验

我们的第二个例子是程序 quiz, quiz 从题库中抽取特定的文件, 并用文件中的问题向用户提问. 例如, 我们可以测试用户对化学元素的了解程度. 假设化学元素的题库文件是 quiz.elems, 文件包含了化学元素的符号, 原子序数, 以及元素的全称, 字段之间用冒号分开. 文件的第一行比较特殊, 它标明了各个字段的意义:

```
symbol:number:name|element
H:1:Hydrogen
He:2:Helium
Li:3:Lithium
Be:4:Beryllium
B:5:Boron
C:6:Carbon
N:7:Nitrogen
O:8:Oxygen
F:9:Fluorine
Ne:10:Neon
Na:11:Sodium|Natrium
...
```

程序根据第一行来判断哪个字段是问题, 哪个字段是正确答案, 然后把文件剩下的 部分读取到一个数组 118
中, 通过该数组, 程序就可以随机地选择问题并检查回答的正确性. 输入命令行

```
awk -f quiz quiz.elems name symbol
```

之后, 我们将会得到类似下面的对话:

```
Beryllium? B
wrong, try again: Be
Right!
Fluorine?
...
```

注意, 我们可以通过正则表达式来处理备选答案 (例如 sodium 或 natrium).

```
# quiz - present a quiz
#   usage: awk -f quiz topicfile question-subj answer-subj

BEGIN {
    FS = ":"
    if (ARGC != 4)
        error("usage: awk -f quiz topicfile question answer")
    if (getline <ARGV[1] < 0)    # 1st line is subj:subj:...
        error("no such quiz as " ARGV[1])
    for (q = 1; q <= NF; q++)
        if ($q ~ ARGV[2])
            break
    for (a = 1; a <= NF; a++)
        if ($a ~ ARGV[3])
            break
    if (q > NF || a > NF || q == a)
        error("valid subjects are " $0)
    while (getline <ARGV[1] > 0) # load the quiz
        qa[++nq] = $0
    ARGC = 2; ARGV[1] = "-"      # now read standard input
    srand()
    do {
        split(qa[int(rand()*nq + 1)], x)
        printf("%s? ", x[q])
        while ((input = getline) > 0)
            if ($0 ~ "^(" x[a] ")$") {
                print "Right!"
                break
            } else if ($0 == "") {
                print x[a]
                break
            } else
                printf("wrong, try again: ")
    } while (input > 0)
}

function error(s) { printf("error: %s\n", s); exit }
```

119

为了辨别出正确答案, 我们用 ^ 和 $ 包围正则表达式, 否则的话, 只要用户的回答中含有与标准答案匹配的子字符串, 那么该回答就会被认为是正确的 (于是, Ne, Na 与 N 都会被当作标准答案 N).

Exercise 5.9 修改 quiz: 对于同一道问题最多输出一次.

5.3　文本处理

由于强大的字符串处理能力, 对于涉及到文本处理与文件准备的工作来说, awk 是一个非常有用的工具. 作为示例, 这一节包含的程序可以用于单词计数, 文本格式化, 交叉索引维护, 制作 KWIC 索引, 以及索引准备工作.

单词计数

在第 一 章, 我们展示了一个用于计算某个文件的行数, 单词数与字符数的程序, 在该程序中, 单词被定义为由多个非空白字符组成的字符序列. 与此相关的问题是计算某个文档中, 每个单词的出现次数, 解决这个问题的一种思路是把文档中的单词分解出来, 对它们排序, 这样相同的单词就会紧挨在一起, 最后再用 control-break 程序计算每个单词的出现次数.

另一种思路 (与 awk 非常契合) 是分解出每一个单词, 把单词的出现次数记录在关联数组中. 为了更好地完成这个任务, 我们必须搞清楚一个单词究竟是由什么组成的. 在下面的程序里, 单词是一个移除了标点符号的字段, 于是, "word", "word;" 以及 "(word)" 都看作是单词 word. END 降序输出每个单词的出现次数.

```
# wordfreq - print number of occurrences of each word
#   input:  text
#   output: number-word pairs sorted by number

    { gsub(/[.,:;!?(){}]/, "")    # remove punctuation
      for (i = 1; i <= NF; i++)
          count[$i]++
    }
END { for (w in count)
          print count[w], w | "sort -rn"
    }
```

本章草稿出现最多的十个单词是:

```
312 the     152 a       126 of      121 is      110 to
92 and      72 in       71 The      59 at       54 that
```

Exercise 5.10 修改单词计数程序: 不区分单词大小写, 于是 The 与 the 被当作是同一个单词.

Exercise 5.11 写一个程序, 该程序计算某个文档中句子的个数及每个句子的长度.

120

Exercise 5.12 写一个 control-break 程序, 用来计算单词的个数. 与 wordfreq 相比, 它的性能表现如何?

文本格式化

程序 fmt 把它的输入格式化成每行至多 60 个字符, 基本思路是通过移动单词, 尽可能地把每一行都塞满. 空行表示分段, 除此之外没有其他控制指令. 如果某些文本在创建时没有考虑到每行的长度, 就可以使用 fmt 对它们进行格式化.

```
# fmt - format
#    input:  text
#    output: text formatted into lines of <= 60 characters

/./  { for (i = 1; i <= NF; i++) addword($i) }
/^$/ { printline(); print "" }
END  { printline() }

function addword(w) {
    if (length(line) + length(w) > 60)
        printline()
    line = line " " w
}

function printline() {
    if (length(line) > 0) {
        print substr(line, 2)   # removes leading blank
        line = ""
    }
}
```

Exercise 5.13 修改 fmt: 对齐输出文本的右边空白.

Exercise 5.14 增强 fmt 的功能, 使得它可以通过识别文档中可能的标题, 列表等信息, 推断出文档的正确格式. 这次不是直接对文档进行格式化, 而是生成排版程序 (例如 troff, TEX 等) 的格式化命令.

维护手稿的交叉引用

文件准备的一个常见问题是为条目创建一个一致的名字或数字集合, 条目可以是文献引用, 图表, 示例等.[①] 某些文本格式化程序可以帮助我们完成这件工作, 但是大多数情况下还是需要自己来完成. 我们的下一个例子是对交叉引用进行编号, 该技术对于技术文档编写来说特别有用.

编写文档时, 作者为不同的条目创建并使用不同的符号名, 这些条目之后会被交叉引用. 因为名字是符号化的, 所以条目可以被添加, 删除, 重新编排, 而不用修改已存在的名字. 本节所提出的交叉引用技

[①] 原文为 A common problem in document preparation is creating a consistent set of names or numbers for items like bibliographic citations, figures, tables, examples, and so on.

术是使用两个程序共同创建一个文本, 文本中的符号名被适当的数字所替代[①]. 这里有一个示例文档, 文档中包含了两个文献引用和一张图的符号名: 121

```
.#Fig _quotes_
Figure _quotes_ gives two brief quotations from famous books.

                        Figure _quotes_:

.#Bib _alice_
  "... `and what is the use of a book,' thought Alice,
  `without pictures or conversations?'" [_alice_]

.#Bib _huck_
  "... if I'd a knowed what a trouble it was to make a book
  I wouldn't a tackled it and ain't agoing to no more." [_huck_]


[_alice_] Carroll, L., Alice's Adventures in Wonderland,
    Macmillan, 1865.
[_huck_] Twain, M., Adventures of Huckleberry Finn,
    Webster & Co., 1885.
```

每一个定义符号名的行都具有形式:

```
.#Category _SymbolicName_
```

这样的定义可以出现在文档中的任何地方, 只要作者愿意, 他可以定义多种不同的 Category. 在整篇文档中, 某一条目总是通过它的符号名来引用. 我们规定符号名以下划线开始并结尾, 不过你也可以使用其他任意的名字, 前提是你可以从文本中把它们分离出来 (条目的名字不可以相同, 即使它们在不同的类别中). 名字 .#Fig 与 .#Big 以句点开始, 这样的话即使交叉引用未被解析, 格式化程序 troff 也可以忽略它们 (对于不同的格式化程序, 我们可能需要作不同的约定)

转换程序创建一份新版本的文档, 在新版本中, 定义被删除, 并且每一个符号名都被一个数字所替代. 在每一个类别中, 数字从 1 开始, 并按照原始文档中该类别定义出现的顺序而递增.

把文档输送给两个程序就可以完成上面所说的转换过程. 这里体现出的工作划分思想是强大的通用编程技巧的又一个实例: 第一个程序创建第二个程序, 并让第二个程序完成剩下的部分. 在这个案例中, 第一个程序 xref 扫描原始文档并创建第二个程序 xref.temp, 实际的转换过程将由 xref.temp 来完成. 假设手稿的原版是 document, 只需要键入

```
awk -f xref document > xref.temp
awk -f xref.temp document
```

就可以得到带有数字形式引用的文档. 第二个程序的输出可以被重定向到打印机或文本格式化程序. 上面所提到的示例文档的转换结果是: 122

[①] 原文为 Two programs create the version in which the symbolic names are replaced by suitable numbers. 原文显得有点突兀: 不经过渡就直接提到 “Two programs ...”

```
Figure 1 gives two brief quotations from famous books.

                        Figure 1:

  "... `and what is the use of a book,' thought Alice,
  `without pictures or conversations?'" [1]

  "... if I'd a knowed what a trouble it was to make a book
  I wouldn't a tackled it and ain't agoing to no more." [2]


[1] Carroll, L., Alice's Adventures in Wonderland,
    Macmillan, 1865.
[2] Twain, M., Adventures of Huckleberry Finn,
    Webster & Co., 1885.
```

程序 xref 在文档中搜索以 .# 开始的行, 对该行的每一次出现, 程序都会递增数组 count 中与该类别对应的元素的值, 然后打印一条 gsub 语句.

```
# xref - create numeric values for symbolic names
#    input:  text with definitions for symbolic names
#    output: awk program to replace symbolic names by numbers

/^\.#/ { printf("{ gsub(/%s/, \"%d\") }\n", $2, ++count[$1]) }
END    { printf("!/^[.]#/\n") }
```

对于文件 document, xref 输出的是第二个程序 xref.temp:

```
{ gsub(/_quotes_/, "1") }
{ gsub(/_alice_/, "1") }
{ gsub(/_huck_/, "2") }
!/^[.]#/
```

gsub 把符号名全局性地替换成数字, 最后一条语句忽略以 .# 开始的行, 从而删除掉符号名定义.

Exercise 5.15 如果遗漏了符号名末尾的下划线, 会发生什么事?

Exercise 5.16 修改 xref: 可以侦测到某个符号名的多次定义

Exercise 5.17 修改 xref, 使得它可以生成你所喜爱的文本编辑器或流式编辑器 (比如 sed) 的编辑命令, 而非 awk 命令. 这会对编辑器的性能产生什么影响?

Exercise 5.18 你有没有办法让 xref 只需要对输入数据遍历一次? “遍历一次” 对定义的放置位置而言, 隐含着什么限制条件?

制作 KWIC 索引

一个 KWIC (Keyword-In-Context) 索引指的是一种显示了其所在行的上下文内的每一个单词的索引[①]. 在本质上, 它所提供的信息等价于重要语汇索引 (concordance), 虽然形式上有点 不同. 考虑下面三个句子: 123

```
All's well that ends well.
Nature abhors a vacuum.
Every man has a price.
```

这三个句子的 KWIC 索引是[②]:

```
                    Nature  abhors a vacuum.
                            All's well that ends well.
              Every man has  a price.
              Nature abhors  a vacuum.
          All's well that  ends well.
                           Every man has a price.
                Every man  has a price.
                    Every  man has a price.
                           Nature abhors a vacuum.
          Every man has a  price.
               All's well  that ends well.
          Nature abhors a  vacuum.
    All's well that ends  well.
                    All's  well that ends well.
```

在软件工程领域, 关于如何构造 KWIC 索引有一段很有趣的历史. 该问题由 Parnas 在 1972 年提出, 当时是把它当作一个设计习题, 他提供了一个基于单个程序的解决方案. Unix 命令 `ptx` 用于构造 KWIC 索引, 它的方法与 Parnas 相比非常相似, 大约只有 500 行 C 代码.

Unix 管道提供了一个三步骤的解决方案: 第一个程序生成每一个输入行的旋转, 于是, 行内的每一个单词轮流移动到其所在行的行首; 第二个程序对它们进行排序; 最后一个程序把它们恢复到旋转前的样子. *Software Tools* 也提供了一个构造 KWIC 索引的程序, 该程序就是以本段所提出的方案为基础实现的, 除了排序, 大约包含 70 行 Ratfor (一种结构化的 Fortran 衍生语言) 代码.

使用 awk 的话会更加方便, 只需要两个简短的 awk 程序, 程序之间再放置一个 `sort` 命令即可:

```
awk '
# kwic - generate kwic index

{   print $0
    for (i = length($0); i > 0; i--) # compute length only once
        if (substr($0,i,1) == " ")
```

[①] 原文为 A Keyword-In-Context or KWIC index is an index that shows each word in the context of the line it is found in.

[②] 这个索引是在我的系统上运行 `kwic` 得到的, 某几行的出现顺序与英文原版不太一致, 我也不知道是什么原因 (我甚至没搞懂 KWIC 究竟是什么) — 译者注

```
            # prefix space suffix ==> suffix tab prefix
            print substr($0,i+1) "\t" substr($0,1,i-1)
} ' |
sort -f |
awk '
BEGIN { FS = "\t"; WID = 30 }
      { printf("%" WID "s  %s\n", substr($2,length($2)-WID+1),
            substr($1,1,WID))
} '
```

第一个程序首先打印每个输入行的副本, 然后, 为输入行内的每一个空格打印一行输出, 输出行由三部分构成: 当前输入行空格后的内容, 制表符, 当前输入行空格前的内容.

124

所有的输出行再以管道的方式输送给 Unix 命令 `sort -f`, 选项 `-f` 表示 “合并 (folding)” 大小写, 比如, `Jack` 与 `jack` 将会紧挨在一起出现.

第二个 awk 程序对 `sort` 的输出进行重构与格式化. 它首先打印当前输入行制表符后面的内容, 再是一个制表符, 最后是当前输入行制表符前面的内容.

Exercise 5.19 为 `kwic` 添加一个 “停止列表 (stop list)”, “停止列表” 指的是不能看作关键词的单词集合, 比如 “a”, “the”.

Exercise 5.20 修改 `kwic`, 使得它可以显示尽可能多的行内数据, 方法是在末尾换行, 而非截断.[①]

Exercise 5.21 编写一个创建重要语汇索引 (concordance) 的程序: 对每一个重要的单词, 显示所有的出现了该单词的句子或短语.

制作索引

编写大型文档 (例如书籍或手册) 时, 通常情况下需要制作一份索引. 这个任务由两部分组成: 第一部分是决定为哪些术语制作索引, 想把这部分工作做好需要多动点脑子, 而且它无法通过机械化步骤完成; 另一部分则完成是机械化的: 根据索引字和页码列表, 生成按字母排序且精心格式化过的索引. 本书的最后几页就是由索引组成的.

在本节的剩下部分里, 我们将使用 awk 和 `sort` 构造一个索引器的核心部分 (本书所用的索引构造程序只比它稍微复杂了一点). 基本思路类似于 KWIC 索引程序: 分而治之. 任务被细分成一系列小任务, 每个小任务只需要一行排序命令或一个简短的 awk 程序就可以完成. 由于每一个小任务都很简单且独立, 所以很容易通过扩充或修改它们, 来满足更加复杂的索引需求.

这些程序包含了许多与 `troff` 相关的细节 (本书就是用 `troff` 排版的), 如果用的是其他排版程序 (比如 TEX 或 Scribe), 那么这些细节都会发生变化, 不过程序的基本结构是一样的.

我们通过在文本中插入格式化命令来为书籍制作索引. 当 `troff` 扫描文本时, 根据命令把索引字与页码搜集到一个文件中. 文件的内容类似于下面的文本, 这些文本是索引准备程序进一步加工处理的原材料 (索引字与页码之间用一个制表符分开):

125

[①] 原文为 Fix `kwic` to show as much as possible of lines, by wrapping around at the ends rather than truncating.

```
[FS] variable       35
[FS] variable       36
arithmetic operators        36
coercion rules      44
string comparison   44
numeric comparison  44
arithmetic operators        44
coercion~to number  45
coercion~to string  45
[if]-[else] statement       47
control-flow statements     48
[FS] variable       52
...
```

我们的目标是对于索引字, 比如

```
string comparison       44
```

在最终的索引中会以两种形式呈现:

```
string comparison 44
comparison, string 44
```

索引字通常按照术语中出现的每一个空格进行分割并旋转, 波浪号 ~ 用于阻止分割:

```
coercion~to number      45
```

将不会对 “to” 进行索引.

有几个细节需要注意. 由于我们用的是 troff, 它有一些字号与字体设置命令, 所以在排序时需要识别并忽略它们. 另外, 由于字体需要经常改变, 所以我们用命令 `[...]` 表示在索引中用等宽字体显示被方括号包围的文本, 例如

```
[if]-[else] statement
```

将会被打印成

> if-else statement

制作索引的过程由六个命令共同完成:

ix.sort1	先按索引字, 再按页码对输入进行排序
ix.collapse	合并同一个术语的页码
ix.rotate	生成索引字的旋转
ix.genkey	为了强制按照正确的顺序进行排序, 生成一个排序键
ix.sort2	按照排序键进行排序
ix.format	生成最终的输出

这些命令逐渐地往最终的索引中添加 索引字-页码 对. 接下来, 我们按照顺序对这些程序进行分析.

第一个排序命令把 索引字-页码 对当作输入数据, 把相同的术语放在一起, 并按照页码排序:[1] 126

```
# ix.sort1 - sort by index term, then by page number
#     input/output: lines of the form string tab number
#     sort by string, then by number; discard duplicates
```

```
sort -t 'tab' +0 -1 +1n -2 -u
```

解释一下 sort 的命令行参数: -t'*tab*' 表示使用制表符作为字段分隔符; +0 -1 表示第一个排序键是第一个字段, 结果是按照字母排序; +1n -2 表示第二个排序键是第二个字段, 结果是按照数值排序; -u 表示丢弃重复条目. (在第 六 章我们将展示一个排序生成程序, 它可以根据你的需要构建排序命令的参数) 如果以上面的数据作为输入, 则 ix.sort1 的输出是:

```
arithmetic operators        36
arithmetic operators        44
coercion rules      44
coercion~to number  45
coercion~to string  45
control-flow statements     48
[FS] variable       35
[FS] variable       36
[FS] variable       52
[if]-[else] statement       47
numeric comparison  44
string comparison   44
```

这个输出成为下一个程序 ix.collapse 的输入, 它把同一个术语的页码都放在同一行, 该程序是通常的 control-break 程序的变形.

```
# ix.collapse - combine number lists for identical terms
#   input:  string tab num \n string tab num ...
#   output: string tab num num ...

BEGIN { FS = OFS = "\t" }
$1 != prev {
    if (NR > 1)
        printf("\n")
    prev = $1
    printf("%s\t%s", $1, $2)
    next
}
    { printf(" %s", $2) }
```

[1] 在 Linux 中你可以写成 sort -t $'\t' +0 -1 +1n -2 -u

```
    END { if (NR > 1) printf("\n") }
```

ix.collapse 的输出是 127

```
    arithmetic operators        36 44
    coercion rules      44
    coercion~to number  45
    coercion~to string  45
    control-flow statements     48
    [FS] variable       35 36 52
    [if]-[else] statement       47
    numeric comparison  44
    string comparison   44
```

下一个程序 ix.rotate 为索引字生成旋转,例如根据“string comparison”生成“comparison, string”. 旋转操作与 KWIC 索引制作过程中出现的旋转大致相同, 虽然我们用了不同的方法来编写. 注意 for 循环中的赋值表达式.

```
    # ix.rotate - generate rotations of index terms
    #   input:  string tab num num ...
    #   output: rotations of string tab num num ...

    BEGIN { FS = OFS = "\t" }
    {   print $1, $2    # unrotated form
        for (i = 1; (j = index(substr($1, i+1), " ")) > 0; ) {
            i += j      # find each blank, rotate around it
            printf("%s, %s\t%s\n",
                substr($1, i+1), substr($1, 1, i-1), $2)
        }
    }
```

ix.rotate 的部分输出内容是

```
    arithmetic operators        36 44
    operators, arithmetic       36 44
    coercion rules      44
    rules, coercion     44
    coercion~to number  45
    number, coercion~to 45
    coercion~to string  45
    string, coercion~to 45
    control-flow statements     48
    statements, control-flow    48
    ...
```

下一个步骤是对旋转后的索引字排序. 如果直接对它们进行排序, 由于文本内仍然嵌入有格式化命令 (比如 `[...]`), 这些命令会对排序结果造成干扰. 我们的解决办法是为每一行加上一个前缀 (排序键), 这个前缀确保排序结果是正确的, 在后面的步骤中会把这些前缀移除. 程序 `ix.genkey` 通过移除 `troff` 的字号与字体设置命令来构造前缀, `troff` 的字号与字体设置命令类似于 `\s+`*n*, `\s-`*n*, `\f`*x*, 或 `\f(`*xx*. `ix.genkey` 还会把排序键中的波浪号替换成空格, 移除任意非字母数字字符 (除了空格). [1] 128

```
# ix.genkey - generate sort key to force ordering
#   input:  string tab num num ...
#   output: sort key tab string tab num num ...

BEGIN { FS = OFS = "\t" }

{   gsub(/~/, " ", $1)       # tildes now become blanks
    key = $1
    # remove troff size and font change commands from key
    gsub(/\\f.|\\f\(..|\\s[-+][0-9]/, "", key)
    # keep blanks, letters, digits only
    gsub(/[^a-zA-Z0-9 ]+/, "", key)
    if (key ~ /^[^a-zA-Z]/)  # force nonalpha to sort first
        key = " " key        # by prefixing a blank
    print key, $1, $2
}
```

输出是

```
arithmetic operators        arithmetic operators    36 44
operators arithmetic        operators, arithmetic   36 44
coercion rules      coercion rules  44
rules coercion      rules, coercion 44
coercion to number  coercion to number      45
number coercion to  number, coercion to     45
coercion to string  coercion to string      45
string coercion to  string, coercion to     45
controlflow statements      control-flow statements 48
...
```

前面几行应该能够阐明排序键与实际数据之间的区别.[2]

第二个排序命令对输入按照字母顺序排序, 同之前的一样, 选项 `-f` 表示合并大小写字母, `-d` 表示按照字典序排序.

```
# ix.sort2 - sort by sort key
#     input/output: sort-key tab string tab num num ...
```

[1] 从前面的输出中可以看到, 对于现代 `sort` 命令来说, 这个步骤是多余的 — 译者注

[2] 原文为 The first few lines should clarify the distinction between the sort key and the actual data.

```
sort -f -d
```

程序的输出是索引最终的排列顺序:

```
arithmetic operators        arithmetic operators    36 44
coercion rules      coercion rules  44
coercion to number  coercion to number      45
coercion to string  coercion to string      45
comparison numeric  comparison, numeric     44
comparison string   comparison, string      44
controlflow statements      control-flow statements 48
FS variable [FS] variable   35 36 52
ifelse statement    [if]-[else] statement   47
number coercion to  number, coercion to     45
...
```

129

最后一步是用程序 ix.format 移除排序键, 把 [...] 扩展成 troff 的字体设置命令, 并在每个术语之前加上一个格式化命令 .XX, 格式化程序可以利用这个命令控制文本的大小, 位置等. (实际产生的命令只对 troff 有意义, 你大可忽略这些细节)

```
# ix.format - remove key, restore size and font commands
#   input:  sort key tab string tab num num ...
#   output: troff format, ready to print

BEGIN { FS = "\t" }

{   gsub(/ /, ", ", $3)        # commas between page numbers
    gsub(/\[/, "\\f(CW", $2)   # set constant-width font
    gsub(/\]/, "\\fP", $2)     # restore previous font
    print ".XX"                # user-definable command
    printf("%s  %s\n", $2, $3) # actual index entry
}
```

输出的部分内容是

```
.XX
arithmetic operators  36, 44
.XX
coercion rules  44
.XX
coercion to number  45
...
```

概括起来, 索引构造过程由六个命令的流水线组成

```
sh ix.sort1 |
awk -f ix.collapse |
awk -f ix.rotate |
awk -f ix.genkey |
sh ix.sort2 |
awk -f ix.format
```

如果把本节开头所展示的 索引字-页码 对集合作为输入数据, 再对输出进行格式化, 则最终的排版结果是

arithmetic operators 36, 44
coercion rules 44
coercion to number 45
coercion to string 45
comparison, numeric 44
comparison, string 44
control-flow statements 48
`FS` variable 35, 36, 52
`if-else` statement 47
number, coercion to 45
numeric comparison 44

130

可以对这些程序进行一些增强或变形, 针对这点, 习题给出了几点比较有用的建议. 不过最重要的是, 把一个大型任务分解成若干个小任务可以使整个工作完成起来更加简单容易, 也更容易适应需求的变化.

Exercise 5.22 修改或增强索引程序的功能: 它可以提供分层的索引 (*See* 与 *See also*) 与罗马数字形式的页码.

Exercise 5.23 允许索引字中出现字面意义上的 `[`, `]`, `~` 与 `%`

Exercise 5.24 构造一个准备单词与短语列表的工具程序, 利用该程序来自动创建索引. 在促成索引字与主题方面, `wordfreq` 所生成的单词频率列表表现如何?[①]

5.4 小结

Awk 处理文本就像 C 或 Pascal 处理数字那样轻松 — 自动管理存储, 内建运算符与内建函数提供了许多必需的功能. 结果是 awk 擅长于原型构造, 有时候即使是对产品化的使用来说, 原型也已足够[②]. 索引程序就是一个很好的例子 — 我们使用该程序的另一个版本为本书制作索引.

[①] Attack the problem of creating an index automatically by building tools that prepare lists of words, phrases, etc. How well does the list of word frequencies produced by `wordfreq` suggest index terms or topics?

[②] 原文为 As a result, awk is usually good for prototyping, and sometimes it is quite adequate for production use.

参考资料

我们的测试程序以 Unix 的测试程序为原型构造而来, 后者最初由 Doug McIlroy 开发而成. 废话生成器这个主意来自于 Ron Hardin. Parnas 的一篇关于 KWIC 索引的论文 “On the criteria to be used in decomposing systems into modules” 载于 *Communications of the ACM*, 1972 年 12 月. Jon Bentley 提供了 KWIC 索引程序的早期版本, 详细内容在 *Programming Pearls*, *Communications of the ACM*, 1985 年 6 月. 维护交叉引用的程序基于 Aho 与 Sethi 的工作, “Maintaining Cross-Reference in Manuscripts”, CSTR 129, AT&T Bell Laboratories, Murray Hill, NJ (1986). 构造索引的程序源自 Bentley 与 Kernighan 的工作, “Tools for Printing Indexes”, CSTR 130, AT&T Bell Laboratories, Murray Hill, NJ (1986).

第六章　小型语言

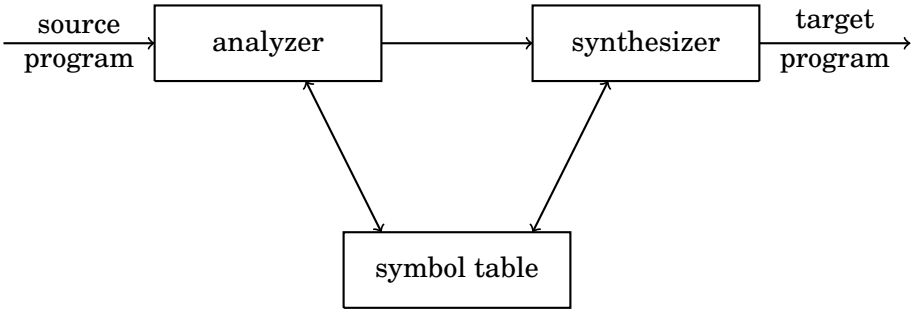
131

人们经常使用 awk 开发“小型语言”的翻译器 (“小型语言”指的特定于某些应用领域的专用编程语言), 开发翻译器的原因主要有三点. 首先, 它可以帮助你了解语言处理程序的工作流程. 本章的第一个例子是一个汇编程序, 虽然只有 20 来行, 但已经包含了汇编过程的核心要素, 为了执行汇编程序, 我们还要开发一个解释程序. 汇编程序与解释程序反映了早期阶段汇编语言与机器架构的关系. 其他例子还包括一个后缀计算器, 和 awk 子集的递归下降分析器.

第二个原因是在实际工作中, 为了实现一个专用的编程语言, 通常需要投入大量的精力与财力, 不过在这之前, 我们有必要测试一下新语言的语法和语义. 作为示例, 本章讨论了一个画图语言和一个参数设置语言, 后者用于设置排序命令的参数.

最后一点是希望编程语言能够在实际的工作发挥作用, 就比如说本章所开发的计算器.

语言处理程序围绕下面这个模型构造而成:



分析器 (analyzer) 系语言处理程序的前端, 它负责读取源程序 (source program) 并将其切分成一个个词法单元, 词法单元包括运算符, 操作数等. 分析器对源程序进行语法检查, 如果源程序含有语法错误, 它就会打印一条错误消息. 最后, 分析器把源程序转换成某种中间形式, 并传递给后端 (合成器), 合成器 (synthesizer) 再根据中间形式生成目标程序 (target program). 合成器在生成目标程序的过程中需要和符号表 (symbol table) 通信, 而符号表中的内容由分析器收集而来. 虽然我们把语言的处理过程描述成多个界限分明的不同步骤, 但实际上, 这些界限通常很模糊, 而且有些步骤有可能被合并成一个.

132

利用 awk 为实验性语言构造处理程序非常方便, 这是因为它支持许多和语言翻译相关的操作. 对源程序的分析可以通过字段分割与正则表达式来完成, 用关联数组管理符号表, 用 `printf` 生成目标代码.

关于上面提到的几点, 我们将通过开发几个翻译器来进一步说明. 在保证足够说明问题的前提下, 将尽量保持程序的简短, 而把润色与优化留到习题中.

6.1　汇编程序与解释程序

我们的第一个例子是虚拟计算机的汇编程序, 虚拟计算机这个概念经常出现在计算机体系结构或系统编程的基础课程中. 虚拟计算机有一个累加器, 十条指令, 按字编址且大小为 1000 字的内存, 我们假设内存的一个“字”可以保存 5 个十进制数位, 如果某个字存放的是一条指令, 那么前两个数位表示操作码, 最后三个数位表示内存地址. 所有的汇编语言指令在表 6.1 中列出.

表 6.1: 汇编语言指令集

操作码	指令	意义
01	get	从输入读取一个数, 并存放到累加器中
02	put	把累加器的值写到输出
03	ld M	把地址为 M 的内存单元的值读取到累加器中
04	st M	把累加器的值存放到地址为 M 的内存单元中
05	add M	把地址为 M 的内存单元的值与累加器的值相加, 再把结果存放到累加器中
06	sub M	把地址为 M 的内存单元的值与累加器的值相减, 再把结果存放到累加器中
07	jpos M	如果累加器的值为正, 则跳转到内存地址 M
08	jz M	如果累加器的值为零, 则跳转到内存地址 M
09	j M	跳转到内存地址 M
10	halt	停止执行
	const C	伪操作符, 用于定义一个常量 C

汇编语言程序由语句序列组成, 每一条语句都包括三个部分: 标号, 操作符, 操作数, 任意一个部分都可以省略, 标号如果存在, 则必须是所在行的第一个字段. 程序可以包含 awk 形式的注释. 这里有一个简单的汇编语言程序, 它的功能是输出多个整数的和, 0 表示输入结束.

133

```
# print sum of input numbers (terminated by zero)

      ld    zero   # initialize sum to zero
      st    sum
loop  get          # read a number
      jz    done   # no more input if number is zero
      add   sum    # add in accumulated sum
      st    sum    # store new value back in sum
      j     loop   # go back and read another number

done  ld    sum    # print sum
      put
      halt

zero  const 0
```

```
sum  const
```

对应的目标程序由整数序列组成, 这些整数其实就是程序的机器码形式, 当目标程序运行时, CPU 从内存中读取指令, 译码并执行. 上面程序的机器码是:

```
 0: 03010           ld      zero    # initialize sum to zero
 1: 04011           st      sum
 2: 01000      loop get             # read a number
 3: 08007           jz      done    # no more input if number is zero
 4: 05011           add     sum     # add in accumulated sum
 5: 04011           st      sum     # store new value back in sum
 6: 09002           j       loop    # go back and read another number
 7: 03011      done ld      sum     # print sum
 8: 02000           put
 9: 10000           halt
10: 00000      zero const 0
11: 00000      sum  const
```

第一个字段是内存地址, 第二个字段是编码后的指令. 内存地址 0 存放的是汇编语言程序的第一条指令: `ld zero`.

汇编程序对源程序进行汇编时需要遍历两次. 第一次遍历利用字段分割操作对源程序进行词法与语法检查: 读取汇编语言源程序, 忽略注释, 为每一个标号分配内存地址, 把操作符与操作数的中间表示形式写到一个临时文件中. 第二次遍历读取临时文件, 根据第一次遍历时计算的结果, 把符号化的操作数转换成内存地址, 对操作符与操作数进行编码, 把最终的机器语言程序保存到数组 mem 中.

我们将会开发一个解释器来完成另一半的工作, 解释器可以用来模拟计算机执行机器语言程序时所表现出的行为. 解释器循环地从 mem 中读取指令, 把指令译码成操作符与操作数, 再模拟指令的执行. 变量 pc 用来模拟程序计数器 (program counter). 134

```
# asm - assembler and interpreter for simple computer
#   usage: awk -f asm program-file data-files...

BEGIN {
    srcfile = ARGV[1]
    ARGV[1] = ""  # remaining files are data
    tempfile = "asm.temp"
    n = split("const get put ld st add sub jpos jz j halt", x)
    for (i = 1; i <= n; i++)   # create table of op codes
        op[x[i]] = i-1

# ASSEMBLER PASS 1
    FS = "[ \t]+"
    while (getline <srcfile > 0) {
        sub(/#.*/, "")         # strip comments
```

```
            symtab[$1] = nextmem   # remember label location
            if ($2 != "") {        # save op, addr if present
                print $2 "\t" $3 >tempfile
                nextmem++
            }
        }
        close(tempfile)

    # ASSEMBLER PASS 2
        nextmem = 0
        while (getline <tempfile > 0) {
            if ($2 !~ /^[0-9]*$/)  # if symbolic addr,
                $2 = symtab[$2]    # replace by numeric value
            mem[nextmem++] = 1000 * op[$1] + $2  # pack into word
        }

    # INTERPRETER
        for (pc = 0; pc >= 0; ) {
            addr = mem[pc] % 1000
            code = int(mem[pc++] / 1000)
            if      (code == op["get"])  { getline acc }
            else if (code == op["put"])  { print acc }
            else if (code == op["st"])   { mem[addr] = acc }
            else if (code == op["ld"])   { acc  = mem[addr] }
            else if (code == op["add"])  { acc += mem[addr] }
            else if (code == op["sub"])  { acc -= mem[addr] }
            else if (code == op["jpos"]) { if (acc >  0) pc = addr }
            else if (code == op["jz"])   { if (acc == 0) pc = addr }
            else if (code == op["j"])    { pc = addr }
            else if (code == op["halt"]) { pc = -1 }
            else                         { pc = -1 }
        }
    }
```

数组 symtab 记录标号的内存地址, 如果当前输入行没有标号, 那么内存地址赋值给 symtab[""].

标号是汇编语句的第一个字段, 操作符前面有一个空格符. 第一次遍历前, 把 FS 设置成 [\t]+, 于 135
是字段分隔符变成由多个空格符和制表符组成的序列. 比较特殊的是, 前导空格也被当作字段分隔符, 所以 $1 总是标号, 而 $2 总是操作符.

因为伪操作符 const 的操作码是 0, 所以在第二次遍历时, 语句

```
mem[nextmem++] = 1000 * op[$1] + $2  # pack into word
```

可以同时用来存放常数与指令.

Exercise 6.1 修改 asm, 打印程序与内存的内容, 就像上面显示的那样.

Exercise 6.2 增强解释器的功能, 打印指令的执行轨迹.

Exercise 6.3 适当扩大汇编语言的规模, 比如添加错误处理代码与其他条件判断指令. 为了方便用户使用, 你会怎么处理立即数, 比如 `add = 1` (如果不支持立即数, 就必须要求用户自己创建一个名为 one 的内存单元)?

Exercise 6.4 写一个反汇编程序, 把内存中的内容转换成对应的汇编语言.

Exercise 6.5 查看一台真实的机器 (比如 Apple-II 和 Commodore 的 6502 芯片, 或 IBM PC 及其兼容机的 8086 芯片族), 尝试为它的汇编语言子集写一个汇编程序.

6.2　画图语言

利用字段分割操作, 很容易就可以对我们自己定义的汇编语言作词法和语法分析, 这种简易性对一些高级语言来说同样成立. 我们的下一个例子是 `graph` 的语言处理程序, `graph` 是一种用来画数据坐标图的原型语言. 输入是一张图的规范说明, 规范说明的每一行都表示一个数据点, 或坐标轴的标签信息. 数据点有两种表示形式: 一对 *x*-*y*, 或者只有一个 *y*, 此时默认 *x* 是从 1 开始的递增序列, 即 1, 2, 3, 等等. 两种形式中, 数据值的后面都可以跟随一个可选的非数字字符, 作为数据点的绘图字符 (默认是星号 *). 标签信息由一个关键词和多个参数值组成:

`label` *caption*
`range` *xmin ymin xmax ymax*
`left ticks` t_1 t_2 ...
`bottom ticks` t_1 t_2 ...
`height` *number*
`width` *number*

这些行的出现顺序是任意的, 任意一行都可以省略, 也不需要指定数据的值的范围.

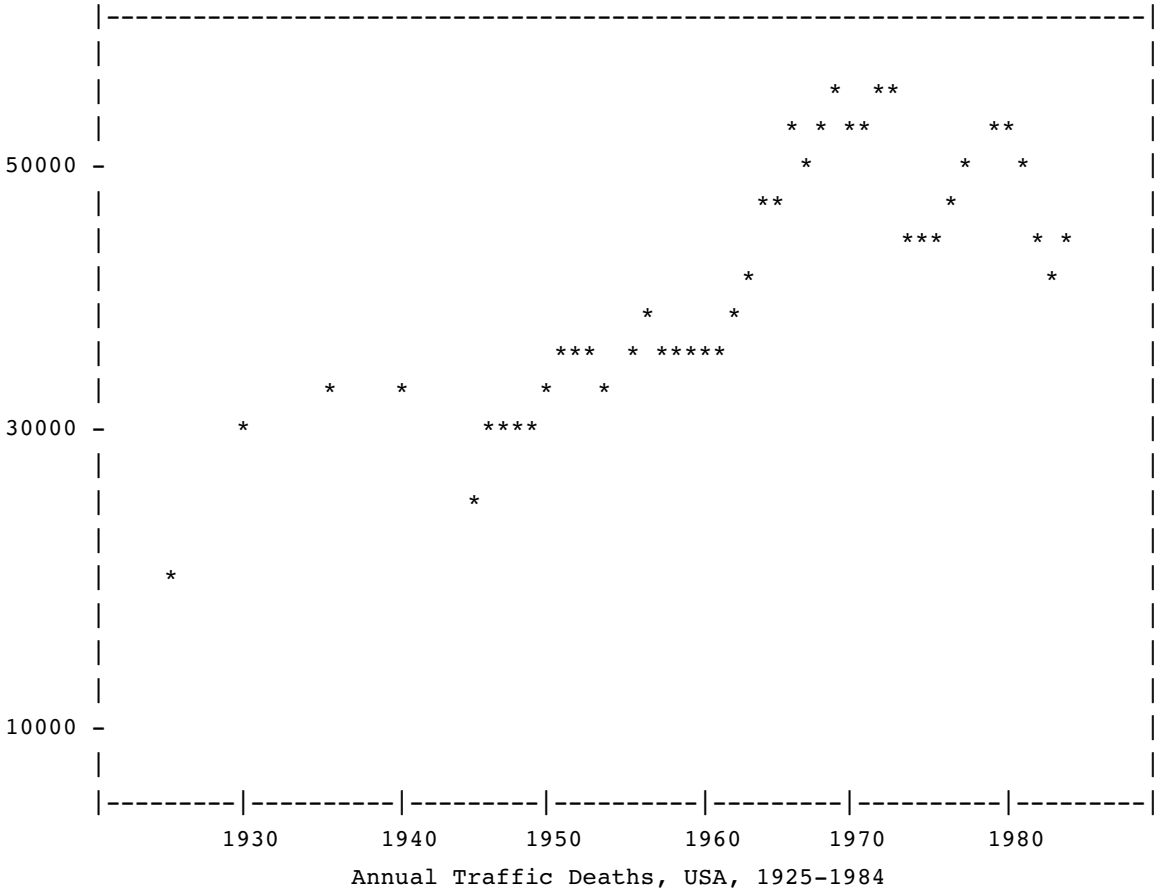
处理程序按比例调整数据点的大小, 并生成绘图命令. 为了使讨论更加具体, 我们把它们打印到 24×80 的字符数组中, 但是, 如果是为某些图像设备生成绘图命令, 实现起来其实也很容易. 例如, 输入数据:

136

```
label Annual Traffic Deaths, USA, 1925-1984
range 1920 5000 1990 60000
left ticks 10000 30000 50000
bottom ticks 1930 1940 1950 1960 1970 1980

1925 21800
1930 31050
1935 36369
...
```

```
1981 51500
1982 46000
1983 44600
1984 46200
```

的输出是[1]:



graph 的处理程序分为两个阶段. 主循环读取并解析图的规范说明, 使用模式辨认不同类型的语句. 图的中间表示形式存放在若干个数组与变量中, 如果必要的话, END 据此计算值的范围, 然后开始绘制边框, 刻度, 标签和数据点. 输出操作被分散在若干个函数中, 这样的话, 即使以后要为特定的设备对代码进行修改, 也可以把修改局限在小范围内.

到目前为止, 这是我们看过的最长的 awk 程序, 大约有 100 行, 其实它是本书第二长的程序. 不过不要担心, 程序把一个大任务分成若干个小步骤来完成, 所以每个部分都很简短. 137

```
# graph - processor for a graph-drawing language
#   input:  data and specification of a graph
#   output: data plotted in specified area

BEGIN {                 # set frame dimensions...
    ht = 24; wid = 80   # height and width
```

[1] 英文原版没有给出完整的统计数据, 下面这张表的统计数据来自 https://en.wikipedia.org/wiki/List_of_motor_vehicle_deaths_in_U.S._by_year — 译者注

```
    ox = 6; oy = 2     # offset for x and y axes
    number = "^[-+]?([0-9]+[.]?[0-9]*|[.][0-9]+)" \
                            "([eE][-+]?[0-9]+)?$"
}
$1 == "label" {                         # for bottom
    sub(/^ *label */, "")
    botlab = $0
    next
}
$1 == "bottom" && $2 == "ticks" {     # ticks for x-axis
    for (i = 3; i <= NF; i++) bticks[++nb] = $i
    next
}
$1 == "left" && $2 == "ticks" {       # ticks for y-axis
    for (i = 3; i <= NF; i++) lticks[++nl] = $i
    next
}
$1 == "range" {                         # xmin ymin xmax ymax
    xmin = $2; ymin = $3; xmax = $4; ymax = $5
    next
}
$1 == "height" { ht = $2; next }
$1 == "width"  { wid = $2; next }
$1 ~ number && $2 ~ number {          # pair of numbers
    nd++    # count number of data points
    x[nd] = $1; y[nd] = $2
    ch[nd] = $3    # optional plotting character
    next
}
$1 ~ number && $2 !~ number {         # single number
    nd++    # count number of data points
    x[nd] = nd; y[nd] = $1; ch[nd] = $2
    next
}
END {    # draw graph
    if (xmin == "") {         # no range was given
        xmin = xmax = x[1]    # so compute it
        ymin = ymax = y[1]
        for (i = 2; i <= nd; i++) {
            if (x[i] < xmin) xmin = x[i]
```

```
            if (x[i] > xmax) xmax = x[i]
            if (y[i] < ymin) ymin = y[i]
            if (y[i] > ymax) ymax = y[i]
        }
    }
    frame(); ticks(); label(); data(); draw()
}
```

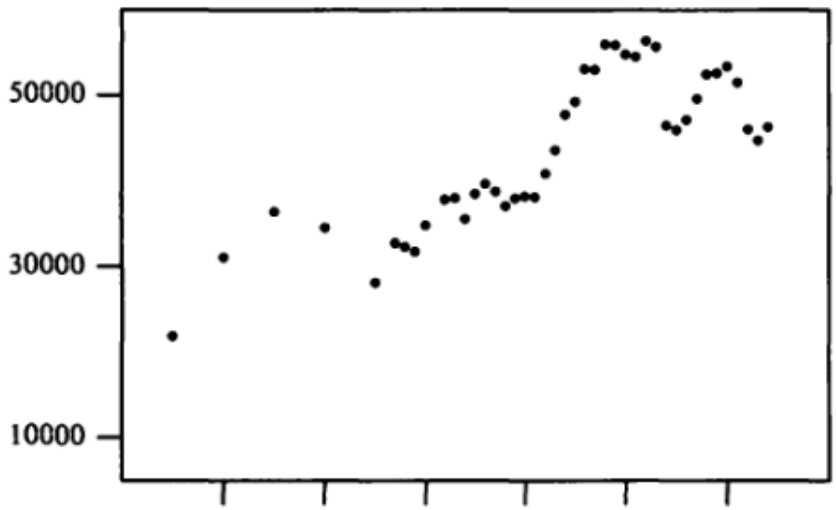
138

```
function frame() {        # create frame for graph
    for (i = ox; i < wid; i++) plot(i, oy, "-")     # bottom
    for (i = ox; i < wid; i++) plot(i, ht-1, "-")   # top
    for (i = oy; i < ht; i++) plot(ox, i, "|")      # left
    for (i = oy; i < ht; i++) plot(wid-1, i, "|")   # right
}
function ticks(    i) {   # create tick marks for both axes
    for (i = 1; i <= nb; i++) {
        plot(xscale(bticks[i]), oy, "|")
        splot(xscale(bticks[i])-1, 1, bticks[i])
    }
    for (i = 1; i <= nl; i++) {
        plot(ox, yscale(lticks[i]), "-")
        splot(0, yscale(lticks[i]), lticks[i])
    }
}
function label() {        # center label under x-axis
    splot(int((wid + ox - length(botlab))/2), 0, botlab)
}
function data(    i) {    # create data points
    for (i = 1; i <= nd; i++)
        plot(xscale(x[i]),yscale(y[i]),ch[i]=="" ? "*" : ch[i])
}
function draw(    i, j) { # print graph from array
    for (i = ht-1; i >= 0; i--) {
        for (j = 0; j < wid; j++)
            printf((j,i) in array ? array[j,i] : " ")
        printf("\n")
    }
}
function xscale(x) {      # scale x-value
    return int((x-xmin)/(xmax-xmin) * (wid-1-ox) + ox + 0.5)
}
```

```
function yscale(y) {      # scale y-value
    return int((y-ymin)/(ymax-ymin) * (ht-1-oy) + oy + 0.5)
}
function plot(x, y, c) {  # put character c in array
    array[x,y] = c
}
function splot(x, y, s,    i, n) { # put string s in array
    n = length(s)
    for (i = 0; i < n; i++)
        array[x+i, y] = substr(s, i+1, 1)
}
```

Awk 很适合处理面向模式的 graph 语言, 之所以这样说是因为它的规范说明语句是带有值的关键词. 使用这种风格来设计语言是一个很好的开始: 既方便用户使用, 也方便处理.

graph 是坐标绘图语言 grap 的简化版本, 而 grap 是绘图语言 pic 的预处理器. 当同时使用 grap, pic 与 troff 时, 同样的数据与几乎相同的描述可以生成下面这张图片: 139



Annual Traffic Deaths, USA, 1925-1984

Awk 擅长于设计与测验小型语言, 如果设计被证明是合理的, 那么就可以用更高效的语言 (比如 C) 来实现一个产品化的版本. 在某些情况下, 即使是一个原型化的版本也可以用在实际生产中. 典型的例子包括糖衣或者对已有工具的定制.[①]

一个特定的实例是定制图表的准备, 在这个实例中, 我们把一门简单的语言, 用 awk 程序转换成 grap 命令. 其他例子还包括散点矩阵 (scatter-plot matrices), 点阵图 (dotchart, 直方图的一种), 箱线图 (boxplot, 显示了某个观测集的期望值 (mean) , 四分位数 (quartile) 和极值 (extreme)) 以及饼图.

Exercise 6.6 修改 graph: 对图进行转置, 即 x 轴在垂直方向上, 而 y 轴在水平方向上, 而且可以指定是否以对数的方式显示 x 坐标或 y 坐标.

[①] 原文为 These situations typically involve sugar-coating or specializing an existing tool.

Exercise 6.7 为 graph 添加一条命令, 这条命令可以让 graph 从文件中读取数据.

Exercise 6.8 自动计算刻度的位置. (即使最终的实现要用到另一种语言, 但 awk 仍然很适合于对算法进行测验)

Exercise 6.9 如果你的系统提供了图形库, 修改 graph, 为图形库生成命令. (这是一个糖衣的例子[1])

140

6.3 sort 生成器

如果你知道如何使用 Unix 命令 sort 的话, 那么它将会是一个非常强大的工具. 但是要记住它所有的选项是非常困难的, 而且对于“字段编号从零开始”这个事实, 你可能会觉得不太习惯. 作为小型语言的另一个训练, 我们将开发一个 sortgen 程序, 它根据由自然语言描述的规范说明来生成排序命令. sortgen 生成排序命令但并不执行 — 这个工作留给用户, 因为用户在执行之前可能想要检查一下生成的命令.

sortgen 的输入是一连串的单词或短语, 它们描述了有关排序的选项, 比如字段分隔符, 排序键, 比较的方向或类型. 程序的目标是在尽可能宽松的语法规则下, 解决比较常见的情况, 比如输入

```
descending numeric order
```

的输出是

```
sort -rn
```

更复杂的一个输入是

```
field separator is :
primary key is field 1
    increasing alphabetic
secondary key is field 5
    reverse numeric
```

sortgen 产出的 sort 命令及其选项与第 四 章的第一个 sort 命令等价:

```
sort -t':' +0 -1 +4rn -5
```

sortgen 的核心是一套规则集, 该规则集把单词与短语描述的排序选项翻译成对应的 sort 命令行选项. 规则集通过 模式–动作 语句实现, 模式用于匹配选项的描述字符串, 而动作则计算相应的命令行选项. 比如, 只要选项描述字符串中出现了 “unique” 或 “discard identical”, 就把它们当作选项 -u, 这个选项的功能是忽略重复条目. 类似的, 字段分隔符假设是制表符, 可是如果描述中出现了 “separate” 这样的字眼, 那就把字段分隔符设置成对应的单个字符.

最难的部分是多个排序键的处理, 每一个排序键都可以跨越多个字段. 识别排序键的关键词是 “key”. 如果描述中出现了这个词, 那么就把一个或两个互相隔离的数字收集起来当作字段编号. “key” 的每一次出现都会为新的排序键收集选项. 排序键私有的选项包括空格抑制 (-b), 字典序 (-d), 忽略大小写 (-f), 数字序 (-n), 和逆序 (-r).

141

[1] This is an example of sugar-coating

```
# sortgen - generate sort command
#   input:  sequence of lines describing sorting options
#   output: Unix sort command with appropriate arguments

BEGIN { key = 0 }

/no |not |n't / { print "error: can't do negatives:", $0; ok = 1 }

# rules for global options
        { ok = 0 }
/uniq|discard.*(iden|dupl)/  { uniq = " -u"; ok = 1 }
/separ.*tab|tab.*sep/        { sep = "t'\t'"; ok = 1 }
/separ/ { for (i = 1; i <= NF; i++)
              if (length($i) == 1)
                  sep = "t'" $i "'"
          ok = 1
        }
/key/   { key++; dokey(); ok = 1 } # new key; must come in order

# rules for each key

/dict/                            { dict[key] = "d"; ok = 1 }
/ignore.*(space|blank)/           { blank[key] = "b"; ok = 1 }
/fold|case/                       { fold[key] = "f"; ok = 1 }
/num/                             { num[key] = "n"; ok = 1 }
/rev|descend|decreas|down|oppos/  { rev[key] = "r"; ok = 1 }
/forward|ascend|increas|up|alpha/ { next }    # this is sort's default
!ok   { print "error: can't understand:", $0 }

END {                         # print flags for each key
    cmd = "sort" uniq
    flag = dict[0] blank[0] fold[0] rev[0] num[0] sep
    if (flag) cmd = cmd " -" flag
    for (i = 1; i <= key; i++)
        if (pos[i] != "") {
            flag = pos[i] dict[i] blank[i] fold[i] rev[i] num[i]
            if (flag) cmd = cmd " +" flag
            if (pos2[i]) cmd = cmd " -" pos2[i]
        }
    print cmd
```

```
}

function dokey(    i) {      # determine position of key
    for (i = 1; i <= NF; i++)
        if ($i ~ /^[0-9]+$/) {
            pos[key] = $i - 1    # sort uses 0-origin
            break
        }
    for (i++; i <= NF; i++)
        if ($i ~ /^[0-9]+$/) {
            pos2[key] = $i
            break
        }
    if (pos[key] == "")
        printf("error: invalid key specification: %s\n", $0)
    if (pos2[key] == "")
        pos2[key] = pos[key] + 1
}
```

142

为了避免处理类似于 “don’t discard duplicates” 或 “no numeric data” 这样的输入, sortgen 会拒绝含有否定短语的行, 接下来的规则处理全局选项, 然后是针对当前排序键的选项, 如果程序无法理解某一行的意思, 就会向用户打印一条消息.

这个程序虽然不够聪明, 但是, 如果用户想要用它来获取正确的答案, 而不是诱导它犯错, 那么 sortgen 仍然是一个很有用的工具.

Exercise 6.10 实现一个新版的 sortgen, 它提供了访问 sort 全部功能的方法, 要求程序能够检测出不一致的请求, 比如同时按照字典序与数值对数据进行排序.

Exercise 6.11 在对输入语言规范性不作过多要求的前提下, 如何提高 sortgen 的精确度.

Exercise 6.12 写一个程序, 把一个排序命令翻译成对应的英文句子, 然后再把句子作为 sortgen 的输入.

6.4 逆波兰式计算器

假设我们现在想要一个计算器, 用来结算支票或计算数学表达式的值. Awk 本身很适合处理这种计算工作, 但是每次程序发生变化时, 都必须重新运行. 我们需要的是一个随着表达式的输入, 可以自动读取并运算的计算器程序.

为了避免开发语法解析器, 我们要求用户按照逆波兰表示法输入表达式. (“逆” 指的是运算符跟在操作数的后面, “波兰” 是因为这种表示法是波兰数学家 Jan Lukasiewicz 发明的). 中缀表达式

```
(1 + 2) * (3 - 4) / 5
```

写成逆波兰表示法变成

```
1 2 + 3 4 - * 5 /
```

逆波兰式不需要括号 — 如果提前知道运算符的操作数个数, 那么表达式就不会有歧义. 逆波兰式非常适合用栈来处理, 由于这个特点, 许多编程语言 (包括 Forth 和 Postscript) 和口袋计算器都把它作为基本表示法来使用.

我们的第一个计算器只能用来计算采用逆波兰式表示法表示的数学表达式, 所有的运算符与操作数都用空格分开. 如果某个字段是操作数, 就把它入栈; 如果是运算符, 就对栈顶的操作数执行对应的运算. 每当一个输入行结束时, 打印栈顶元素的值, 并把它弹出.

143

```
# calc1 - reverse-Polish calculator, version 1
#   input:  arithmetic expressions in reverse Polish
#   output: values of expressions

{   for (i = 1; i <= NF; i++)
        if ($i ~ /^[+-]?([0-9]+[.]?[0-9]*|[.][0-9]+)$/) {
            stack[++top] = $i
        } else if ($i == "+" && top > 1) {
            stack[top-1] += stack[top]; top--
        } else if ($i == "-" && top > 1) {
            stack[top-1] -= stack[top]; top--
        } else if ($i == "*" && top > 1) {
            stack[top-1] *= stack[top]; top--
        } else if ($i == "/" && top > 1) {
            stack[top-1] /= stack[top]; top--
        } else if ($i == "^" && top > 1) {
            stack[top-1] ^= stack[top]; top--
        } else {
            printf("error: cannot evaluate %s\n", $i)
            top = 0
            next
        }
    if (top == 1)
        printf("\t%.8g\n", stack[top--])
    else if (top > 1) {
        printf("error: too many operands\n")
        top = 0
    }
}
```

对于输入

```
1 2 + 3 4 - * 5 /
```

calc1 给出的答案是 -0.6.

我们的第二个逆波兰式计算器支持用户自定义变量, 并且可以调用数学函数. 变量名由字母开始, 后跟字母或数字, 语句 *var*= 表示把栈顶元素弹出, 并赋值给变量 *var*. 如果输入行以赋值语句结束, 则不会打印出任何值. 一个典型的交互过程看起来就像这样 (由程序打印的内容向右缩进):

```
0 -1 stan2 pi=
pi
    3.1415927
355 113 / x= x
    3.1415929
x pi /
    1.0000001
2 sqrt
    1.4142136
```

直接对前一个程序进行扩展, 就可以得到新版的计算器: 144

```
# calc2 - reverse-Polish calculator, version 2
#     input:  expressions in reverse Polish
#     output: value of each expression

{ for (i = 1; i <= NF; i++)
      if ($i ~ /^[+-]?([0-9]+[.]?[0-9]*|[.][0-9]+)$/) {
          stack[++top] = $i
      } else if ($i == "+" && top > 1) {
          stack[top-1] += stack[top]; top--
      } else if ($i == "-" && top > 1) {
          stack[top-1] -= stack[top]; top--
      } else if ($i == "*" && top > 1) {
          stack[top-1] *= stack[top]; top--
      } else if ($i == "/" && top > 1) {
          stack[top-1] /= stack[top]; top--
      } else if ($i == "^" && top > 1) {
          stack[top-1] ^= stack[top]; top--
      } else if ($i == "sin" && top > 0) {
          stack[top] = sin(stack[top])
      } else if ($i == "cos" && top > 0) {
          stack[top] = cos(stack[top])
      } else if ($i == "atan2" && top > 1) {
          stack[top-1] = atan2(stack[top-1],stack[top]); top--
      } else if ($i == "log" && top > 0) {
          stack[top] = log(stack[top])
```

```
        } else if ($i == "exp" && top > 0) {
            stack[top] = exp(stack[top])
        } else if ($i == "sqrt" && top > 0) {
            stack[top] = sqrt(stack[top])
        } else if ($i == "int" && top > 0) {
            stack[top] = int(stack[top])
        } else if ($i in vars) {
            stack[++top] = vars[$i]
        } else if ($i ~ /^[a-zA-Z][a-zA-Z0-9]*=$/ && top > 0) {
            vars[substr($i, 1, length($i)-1)] = stack[top--]
        } else {
            printf("error: cannot evaluate %s\n", $i)
            top = 0
            next
        }
    if (top == 1 && $NF !~ /\=$/)
        printf("\t%.8g\n", stack[top--])
    else if (top > 1) {
        printf("error: too many operands\n")
        top = 0
    }
}
```

Exercise 6.13 为 calc2 的标准值 (例如 π 与 e) 添加内建变量. 添加一个内建变量, 用来表示最后一个输入行的运算结果. 添加两个栈操作运算符, 分别用来完成栈顶元素的复制, 与交换栈顶的两个元素.

145

6.5　中缀计算器

到目前为止, 本章所介绍的语言都拥有易于分析的语法规则. 然而, 大多数高级语言拥有大量不同优先级的运算符, 嵌套结构 (比如括号与 if-then-else 语句), 以及其他复杂的构造, 所有的这些都要求更强大的解析技术, 而不仅仅是字段分割与正则表达式匹配. 使用 awk 处理高级语言是可能的, 只要有一个完全成熟的解析程序即可, 而解析程序可以用任何一种语言来写[①]. 本节将开发一个对数学表达式进行求值的程序, 表达式使用人们通常所熟悉的中缀表示法表示. 以该程序为先导, 我们会在下一节开发一个更加复杂的解析程序.

带有运算符 +, –, * 和 / 的数学表达式语法, 可以用我们在第 五 章讲过的方法来描述:

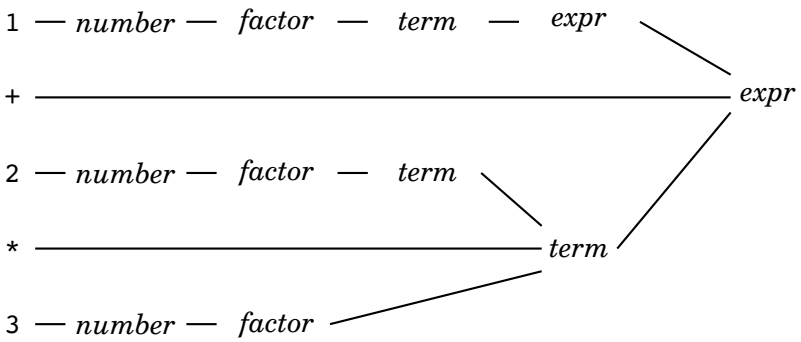
expr → *term*
expr + *term*
expr – *term*
term → *factor*

[①] 原文为 It is possible to process such languages in awk by writing a full-fledged parser, as one would in any language

$$
\begin{array}{rcl}
 & & \mathit{term} * \mathit{factor} \\
 & & \mathit{term} \ / \ \mathit{factor} \\
\mathit{factor} & \rightarrow & \mathit{number} \\
 & & (\ \mathit{expr}\ )
\end{array}
$$

该语法规则不仅描述了数学表达式的有效形式, 还规定了运算符的优先级和结合性. 例如, 一个 *expr* 是多个 *term* 的和或差, 而一个 *term* 由多个 *factor* 组成, 这确保了乘除在加减之前完成.

我们可以把解析过程画成一张语句图来帮助理解, 这与第 五 章所描述的生成过程刚好相反. 比如, 表达式 `1 + 2 * 3` 的解析过程是:



为了开发中缀计算器, 我们需要一个表达式解析程序. 只需要多花点精力, 就可以利用语法规则来构造解析器, 除此之外, 它还可以用来组织程序. 为每一个非终结符写一个函数: 程序使用函数 `expr` 处理由加号或负号分隔的 *term*, 使用函数 `term` 处理由乘号或除号分隔的 *factor*, 最后, 使用函数 `factor` 来识别数字和被括号包围起来的 *expr*.

在下面的程序里, 每一个输入行都被当作一个单独的表达式, 表达式被求值并打印出来. 我们仍然要求所有的运算符, 操作数和括号都用空格分开. 变量 `f` 指向下一个待检验的字段 (运算符或操作数). 146

```
# calc3 - infix calculator

NF > 0 {
    f = 1
    e = expr()
    if (f <= NF) printf("error at %s\n", $f)
    else printf("\t%.8g\n", e)
}

function expr(  e) {        # term | term [+-] term
    e = term()
    while ($f == "+" || $f == "-")
        e = $(f++) == "+" ? e + term() : e - term()
    return e
}

function term(  e) {        # factor | factor [*/] factor
```

```
        e = factor()
        while ($f == "*" || $f == "/")
            e = $(f++) == "*" ? e * factor() : e / factor()
        return e
    }

    function factor(  e) {      # number | (expr)
        if ($f ~ /^[+-]?([0-9]+[.]?[0-9]*|[.][0-9]+)$/) {
            return $(f++)
        } else if ($f == "(") {
            f++
            e = expr()
            if ($(f++) != ")")
                printf("error: missing ) at %s\n", $f)
            return e
        } else {
            printf("error: expected number or ( at %s\n", $f)
            return 0
        }
    }
```

表达式 `$(f++)` 先取 `$f` 的值, 然后再递增 `f`, 请注意它与 `$f++` 的区别, 后者是递增 `$f` 的值.

Exercise 6.14 构造一组测试集, 对 `calc3` 进行详尽地测试.

Exercise 6.15 为中缀计算器添加指数运算, 内建函数和变量. 与逆波兰式计算器的实现作比较.

Exercise 6.16 加强 `calc3` 的错误处理功能.

147

6.6 递归下降语法分析

在这一节, 我们为 awk 的子集开发一个递归下降翻译器, 开发语言还是用 awk. 其中处理算术表达式的部分和我们在上一节里展示的程序, 在本质上是一样的. 为了增加真实性, 翻译器将会生成 C 程序, 原来代码中的 awk 运算符用函数调用来替换. 这个程序在一定程度上说明了语法制导翻译的原理, 以及生成 C 版本 awk 程序的一种方式, C 程序运行起来更快, 也更容易扩展. 基本思路是用函数调用替换掉每一个算术运算符, 比如, `x=y` 变成 `assign(x,y)`, `x+y` 变成 `eval("+",x,y)`. 主输入循环用 `while` 来表示, 每次循环都会调用 `getrec` 来读取输入行, 并把输入行分割成字段. 于是, 由

```
BEGIN   { x = 0; y = 1 }

$1 > x  { if (x == y+1) {
              x = 1
```

```
                y = x * 2
            } else
                print x, z[x]
        }

NR > 1  { print $1 }

END     { print NR }
```

得到的 C 代码是

```
assign(x, num((float)0));
assign(y, num((float)1));
while (getrec()) {
        if (eval(">", field(num((float)1)), x)) {
                if (eval("==", x, eval("+", y, num((float)1)))) {
                        assign(x, num((float)1));
                        assign(y, eval("*", x, num((float)2)));
                } else {
                        print(x, array(z, x));
                }
        }
        if (eval(">", NR, num((float)1))) {
                print(field(num((float)1)));
        }
}
print(NR);
```

如果你要为某个语言处理程序设计前端, 一个比较好的开始方式是为输入语言写一套语法规则. 利用 5.1 节介绍的表示法, 我们可以这样描述 awk 子集的语法规则: 148

program → *opt-begin pa-stats opt-end*
opt-begin → `BEGIN` *statlist* | `""`
opt-end → `END` *statlist* | `""`
pa-stats → *statlist* | *pattern* | *pattern statlist*
pattern → *expr*
statlist → `{` *stats* `}`
stats → *stat stats* | `""`
stat → `print` *exprlist* |
`if (` *expr* `)` *stat opt-else* |
`while (` *expr* `)` *stat* |
statlist |
ident = *expr*

opt-else → `else` *stat* | `""`
exprlist → *expr* | *expr* `,` *exprlist*
expr → *number* | *ident* | `$` *expr* | `(` *expr* `)` |
expr `<` *expr* | *expr* `<=` | ... | *expr* `>` *expr* |
expr `+` *expr* | *expr* `-` *expr* |
expr `*` *expr* | *expr* `/` *expr* | *expr* `%` *expr*
ident → *name* | *name*`[`*expr*`]` | *name*`(` *exprlist* `)`

记号 "" 表示空字符串, 而 | 表示选择.

递归下降语法分析器的关键部分在于它的一整套递归分析函数, 它们负责识别由非终结符生成的字符串. 每一个函数都按照产生式规则调用其他函数, 一直分析到终结符为止, 到这时, 输入中所含的词法单元都被读取出来并加以分类. 由于该方法的自顶向下与递归这两个特性, 所以被称为 "递归下降语法分析".

解析函数的结构紧紧依赖于语言的语法结构. 例如, 函数 program 搜索的语句, 开头可能有 BEGIN 动作, 后面跟着一个 模式–动作 语句列表, 最后面可能还有 END 动作.

在我们的递归下降语法分析器中, 词法分析由子例程 advance 完成, 它搜索下一个词法单元, 并把它保存到变量 tok 中. 每次识别到 *stat* 时, 就产生一个输出, 下层函数返回的字符串会拼接成一个更长的字符串. 为了使输出更具有可读性, 程序会尝试在适当的地方加上制表符, 嵌套的层次在变量 nt 中维护.

程序并不完整 — 它无法识别出 awk 的全部语法, 也无法生成 awk 子集所需的所有 C 代码, 而且程序的健壮性还有待加强. 但是作为一个示例, 它足够说明整个过程是如何进行的. 虽然分析的只是真实语言的子集, 但是这个子集处理起来并不简单, 通过对该子集的分析, 我们可以看到递归下降翻译器的基本结构.

149

```
# awk.parser - recursive-descent translator for part of awk
#   input:  awk program (very restricted subset)
#   output: C code to implement the awk program

BEGIN { program() }

function advance() {      # lexical analyzer; returns next token
    if (tok == "(eof)") return "(eof)"
    while (length(line) == 0)
        if (getline line == 0)
            return tok = "(eof)"
    sub(/^[ \t]+/, "", line)   # remove white space
    if (match(line, /^[A-Za-z_][A-Za-z_0-9]*/) ||    # identifier
        match(line, /^-?([0-9]+\.?[0-9]*|\.[0-9]+)/) ||  # number
        match(line, /^(<|<=|==|!=|>=|>)/) ||         # relational
        match(line, /^./)) {                    # everything else
            tok = substr(line, 1, RLENGTH)
            line = substr(line, RLENGTH+1)
```

```
            return tok
        }
    error("line " NR " incomprehensible at " line)
}
function gen(s) {     # print s with nt leading tabs
    printf("%s%s\n", substr("\t\t\t\t\t\t\t\t\t\t", 1, nt), s)
}
function eat(s) {     # read next token if s == tok
    if (tok != s) error("line " NF ": saw " tok ", expected " s)
    advance()
}
function nl() {       # absorb newlines and semicolons
    while (tok == "\n" || tok == ";")
        advance()
}
function error(s) { print "Error: " s | "cat 1>&2"; exit 1 }

function program() {
    advance()
    if (tok == "BEGIN") { eat("BEGIN"); statlist() }
    pastats()
    if (tok == "END") { eat("END"); statlist() }
    if (tok != "(eof)") error("program continues after END")
}
function pastats() {
    gen("while (getrec()) {"); nt++
    while (tok != "END" && tok != "(eof)") pastat()
    nt--; gen("}")
}
function pastat() {   # pattern-action statement
    if (tok == "{")       # action only
        statlist()
    else {                # pattern-action
        gen("if (" pattern() ") {"); nt++
        if (tok == "{") statlist()
        else              # default action is print $0
            gen("print(field(0));")
        nt--; gen("}")
    }
}
```

```
function pattern() { return expr() }

function statlist() {
    eat("{"); nl(); while (tok != "}") stat(); eat("}"); nl()
}

function stat() {
    if (tok == "print") { eat("print"); gen("print(" exprlist() ");") }
    else if (tok == "if") ifstat()
    else if (tok == "while") whilestat()
    else if (tok == "{") statlist()
    else gen(simplestat() ";")
    nl()
}

function ifstat() {
    eat("if"); eat("("); gen("if (" expr() ") {"); eat(")"); nl(); nt++
    stat()
    if (tok == "else") {      # optional else
        eat("else")
        nl(); nt--; gen("} else {"); nt++
        stat()
    }
    nt--; gen("}")
}

function whilestat() {
    eat("while"); eat("("); gen("while (" expr() ") {"); eat(")"); nl()
    nt++; stat(); nt--; gen("}")
}

function simplestat(   lhs) { # ident = expr | name(exprlist)
    lhs = ident()
    if (tok == "=") {
        eat("=")
        return "assign(" lhs ", " expr() ")"
    } else return lhs
}

function exprlist(    n, e) { # expr , expr , ...
```

```
        e = expr()        # has to be at least one
        for (n = 1; tok == ","; n++) {
            advance()
            e = e ", " expr()
        }
        return e
    }

    function expr(e) {            # rel | rel relop rel
        e = rel()
        while (tok ~ /<|<=|==|!=|>=|>/) {
            op = tok
            advance()
            e = sprintf("eval(\"%s\", %s, %s)", op, e, rel())
        }
        return e
    }

    function rel(op, e) {         # term | term [+-] term
        e = term()
        while (tok == "+" || tok == "-") {
            op = tok
            advance()
            e = sprintf("eval(\"%s\", %s, %s)", op, e, term())
        }
        return e
    }

    function term(op, e) {        # fact | fact [*/%] fact
        e = fact()
        while (tok == "*" || tok == "/" || tok == "%") {
            op = tok
            advance()
            e = sprintf("eval(\"%s\", %s, %s)", op, e, fact())
        }
        return e
    }

    function fact(  e) {          # (expr) | $fact | ident | number
        if (tok == "(") {
            eat("("); e = expr(); eat(")")
```

151

```
            return "(" e ")"
        } else if (tok == "$") {
            eat("$")
            return "field(" fact() ")"
        } else if (tok ~ /^[A-Za-z][A-Za-z0-9]*/) {
            return ident()
        } else if (tok ~ /^-?([0-9]+\.?[0-9]*|\.[0-9]+)/) {
            e = tok
            advance()
            return "num((float)" e ")"
        } else
            error("unexpected " tok " at line " NR)
    }

    function ident(  id, e) {     # name | name[expr] | name(exprlist)
        if (!match(tok, /^[A-Za-z_][A-Za-z_0-9]*/))
            error("unexpected " tok " at line " NR)
        id = tok
        advance()
        if (tok == "[") {         # array
            eat("["); e = expr(); eat("]")
            return "array(" id ", " e ")"
        } else if (tok == "(") {  # function call
            eat("(")
            if (tok != ")") {
                e = exprlist()
                eat(")")
            } else eat(")")
            return id "(" e ")"   # calls are statements
        } else
            return id             # variable
    }
```

152

6.7　小结

构造小型语言通常是一件非常多产的编程工作. 如果词法和语法分析可以通过字段分割与正则表达式来完成, 那么使用 awk 就很方便, 符号表可以用关联数组来存放, 模式–动作 结构与面向模式的编程语言非常般配.

通常来说, 如果缺乏经验, 那么为新领域的新语言作设计决策是一件很困难的工作. 不过, 利用 awk 很容易就可以构造出语言原型并加以试验. 在投入大量的精力与财力进行正式开发之前, 原型可以帮助

人们发现原有设计的问题, 并加以修改. 一旦创建成功, 把原型转化成产品的过程就相对来说比较直接, 转化过程可以用编译器构造工具 (比如 `lex` 和 `yacc`) 来完成, 或者是编译型编程语言, 比如 C.

参考资料

汇编程序与解释程序来源于 Jon Bentley 和 John Dallen, 当时是为了教授软件工程专业课而开发了这两个程序, 具体描述载于 “Exercises in software design”, *IEEE Transactions on Software Engineering*, 1987 年.

关于排版制图语言 `grap` 的相关信息可以在 Jon Bentley 和 Brian Kernighan 写的一篇文章中找到, 文章登在 *Communications of the ACM*, 1986 年 8 月. 这期发行还刊登了 Bentley 写的一篇文章, 题目是 “Little Languages”, 登在 *Programming Pearls* 栏目.

更多的关于如何构造递归下降翻译器的讨论, 可以参考 *Compilers: Principles, Techniques, and Tools* (Aho, Sethi 和 Ullman 著, Addison-Wesley 1986 年出版) 的第 2 章.

第七章　算法实验

153

一般而言, 理解事物如何工作的最好方式就是自己动手做一些小实验, 算法学习就是一个典型的例子: 编写实际代码有助于弄清楚那些容易被伪码掩盖的问题. 不仅如此, 最终得到的程序是可运行的, 通过观察运行结果, 就可以知道算法的的正确性, 而这是伪码所无法办到的.

Awk 很适合做这种测试工作. 如果某个程序使用 awk 编写, 那我们就可以把精力集中在算法上, 而不是语言本身. 如果某个算法最终要应用到某个大型程序中, 那么先让算法能够单独地运行起来可能会更有效率. 当需要为某个算法进行调试, 测试与性能评价时, 通常需要构造一些脚手架程序, 在这一方面, awk 是一款优秀的脚手架构造工具, 它并不关心算法本身是用什么语言实现的.

这一章讨论算法实验. 前半章描述三种排序算法, 这三种算法常常是算法课首先要介绍的内容, 我们将使用 awk 程序对这些算法进行测试, 性能度量和刻画. 后半章展示几种拓扑排序算法, 实现 Unix 的文件更新实用程序 make.

7.1　排序

这一小节讨论三种著名并且很有用的算法: 插入排序, 快速排序, 以及堆排序. 插入排序非常简单, 但是只有在元素很少的情况下效率才足够高; 快速排序是最好的通用排序算法之一; 堆排序可以保证即使在最坏的情况下, 也可以拥有较高的效率. 我们对每一种算法都进行介绍, 并加以实现, 然后再用测试例程对它们进行测试, 最后评价性能.

插入排序

基本概念. 插入排序的过程类似于给一堆卡片排序: 每次从卡片堆里拿出一张, 把它插入到手上拿着的牌的合适位置. [1]

实现. 下面的代码使用插入排序对数组 A[1], ...,A[n] 进行升序排列. 第一个动作把输入数据读取到一个数组中, END 动作调用函数 isort 对数组进行排序, 最后输出排序结果:

154

```
# insertion sort
    { A[NR] = $0 }
END { isort(A, NR)
      for (i = 1; i <= NR; i++)
          print A[i]
    }
```

[1] 原文为 Insertion sort is similar to the method of sorting a sequence of cards by picking up the cards one at a time and inserting each card into its proper position in the hand

```
# isort - sort A[1..n] by insertion
function isort(A, n,    i, j, t) {
    for (i = 2; i <= n; i++)
        for (j = i; j > 1 && A[j-1] > A[j]; j--) {
            # swap A[j-1] and A[j]
            t = A[j-1]; A[j-1] = A[j]; A[j] = t
        }
}
```

isort 函数内的外层循环在每次迭代开始时，数组 A 的元素 1 至元素 i-1 就已经处于有序状态. 内层循环每次迭代都把当前处于第 i 个位置上的元素向前移动，跳过所有比它大的元素. 当外层循环结束时，所有的 n 个元素都处于有序状态.

数值或字符串都可以用这个程序进行排序. 但是当输入数据同时含有数值与字符串时，就要小心一点 — 由于强制类型转换，比较结果可能会让你感到惊讶.

如果数组 A 含有以下 8 个整数:

```
8 1 6 3 5 2 4 7
```

那么排序的过程如下所示:

```
8|1 6 3 5 2 4 7
1 8|6 3 5 2 4 7
1 6 8|3 5 2 4 7
1 3 6 8|5 2 4 7
1 3 5 6 8|2 4 7
1 2 3 5 6 8|4 7
1 2 3 4 5 6 8|7
1 2 3 4 5 6 7 8|
```

竖线符把数组的已排序部分和未排序部分分开.

测试. 应该如何测试 isort? 我们可以每次输入一点数据，并查看排序结果，当然，这样做并没有错，可是对于任意规模的程序来说，这种方法不能做到详尽的测试. 第二种方案是自动生成大量的随机数集合，把这些集合作为 isort 的输入数据. 这的确是一个不错的办法，但是还可以做得更好: 为了测试程序的薄弱环节，我们还需要构造一些特殊的测试用例，用来测试边界与异常情况. 对排序来说，典型的边界与异常情况包括: 155

- 序列长度为 0
- 序列长度为 1
- 序列包含 n 个随机数
- 序列包含 n 个已排序的数
- 序列包含 n 个逆序排列的数
- 序列包含 n 个相同的数

本章的目标之一是展示如何使用 awk 来帮助测试和评价程序. 为了说明, 我们现在要对排序例程的测试与运行结果评价过程进行自动化.

主要有两种办法来实现测试与评价过程的自动化, 每一种都有它各自的优点. 第一种称为 “批处理模式”: 编写一个程序来运行事先计划好的测试集, 并运用上面提到排序算法. 下面的程序可以生成测试数据并检查测试结果. 除了 isort, 还有其他几个函数, 它们用来生成不同类型的数组, 以及检查排序后的数组是否是有序的.

```
# batch test of sorting routines

BEGIN {
    print "    0 elements"
    isort(A, 0); check(A, 0)
    print "    1 element"
    genid(A, 1); isort(A, 1); check(A, 1)

    n = 10
    print "    " n " random integers"
    genrand(A, n); isort(A, n); check(A, n)

    print "    " n " sorted integers"
    gensort(A, n); isort(A, n); check(A, n)

    print "    " n " reverse-sorted integers"
    genrev(A, n); isort(A, n); check(A, n)

    print "    " n " identical integers"
    genid(A, n); isort(A, n); check(A, n)
}

function isort(A,n,     i,j,t) {
    for (i = 2; i <= n; i++)
        for (j = i; j > 1 && A[j-1] > A[j]; j--) {
            # swap A[j-1] and A[j]
            t = A[j-1]; A[j-1] = A[j]; A[j] = t
        }
}
```

156

```
# test-generation and sorting routines...

function check(A,n,   i) {
```

```
    for (i = 1; i < n; i++)
        if (A[i] > A[i+1])
            printf("array is not sorted, element %d\n", i)
}

function genrand(A,n,  i) { # put n random integers in A
    for (i = 1; i <= n; i++)
        A[i] = int(n*rand())
}

function gensort(A,n,  i) { # put n sorted integers in A
    for (i = 1; i <= n; i++)
        A[i] = i
}

function genrev(A,n,  i) {  # put n reverse-sorted integers
    for (i = 1; i <= n; i++)  # in A
        A[i] = n+1-i
}

function genid(A,n,  i) {   # put n identical integers in A
    for (i = 1; i <= n; i++)
        A[i] = 1
}
```

第二种方法相对来说没那么方便, 但很适合用 awk 来处理. 基本思想是构建一个框架程序, 利用该框架可以很容易以交互性的方式来完成测试. 交互式方案是批处理模式的一个很好的补充, 特别是当待测试的算法没有排序那么容易理解时. 交互式处理模式在调试程序时也很方便.

特别地, 我们将要设计的程序, 在效果上等价于一个专门用于构造测试数据与操作的微型编程语言, 因为语言并不需要做太多的工作, 也不必处理大量用户的情况, 所以不用设计得多么复杂. 如果必要的话, 我们甚至可以丢掉写了一半的代码, 重新开始. 我们的语言提供了自动生成数组的功能, 如果再继续往下看的话, 就会发现它还可以指定待运用的排序算法. 我们省略了排序和数据生成子程序, 它们和前一个示例相同.

程序的基本组织是一系列的正则表达式, 它们负责扫描输入数据, 判断数据类型和使用的排序算法. 如果某个输入数据不与任何一个模式相匹配, 程序就会输出一条错误消息, 并演示正确的使用方法. 如果仅仅说明输入数据有错, 可能没多大帮助.

157

```
# interactive test framework for sort routines

/^[0-9]+.*rand/ { n = $1; genrand(A, n); dump(A, n); next }
/^[0-9]+.*id/   { n = $1; genid(A, n); dump(A, n); next }
/^[0-9]+.*sort/ { n = $1; gensort(A, n); dump(A, n); next }
```

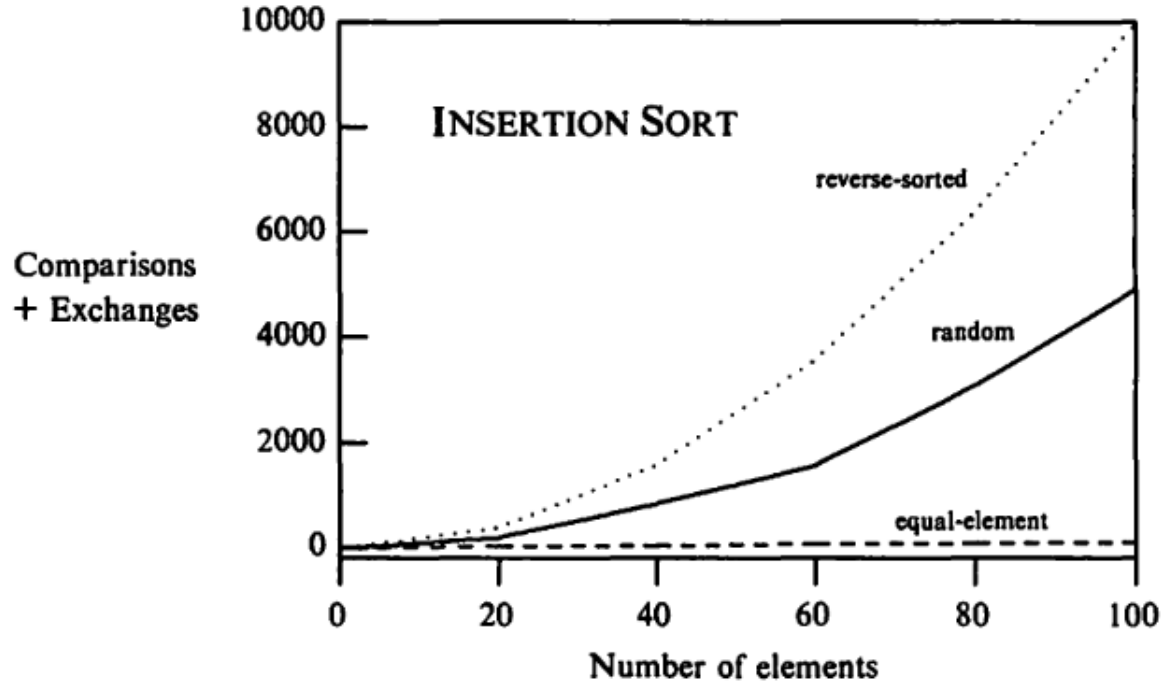
```
/^[0-9]+.*rev/  { n = $1; genrev(A, n); dump(A, n); next }
/^data/ {   # use data right from this line
        for (i = 2; i <= NF; i++)
                A[i-1] = $i
        n = NF - 1
        next
}
/q.*sort/ { qsort(A, 1, n); check(A, n); dump(A, n); next }
/h.*sort/ { hsort(A, n); check(A, n); dump(A, n); next }
/i.*sort/ { isort(A, n); check(A, n); dump(A, n); next }
/./  { print "data ... | N [rand|id|sort|rev]; [qhi]sort" }

function dump(A, n) {    # print A[1]..A[n]
        for (i = 1; i <= n; i++)
                printf(" %s", A[i])
        printf("\n")
}

# test-generation and sorting routines ...
...
```

正则表达式提供了一种非常宽松的输入语法: 比如说, 只要任何一个短语和 “quicksort” 稍微有点接近, 就选择快速排序算法. 我们也可以直接手工输入数据, 而不是自动生成, 这个功能允许我们既可以基于文本, 也可以基于数字对算法进行测试. 为了说明, 上面程序的一个输出是:

```
10 random
 9 8 4 6 7 2 4 0 4 0
isort
 0 0 2 4 4 4 6 7 8 9
10 reverse
 10 9 8 7 6 5 4 3 2 1
qsort
 1 2 3 4 5 6 7 8 9 10
data now is the time for all good men
hsort
 all for good is men now the time
```

性能. `isort` 所执行的操作次数取决于 n 的值, 即待排序的元素个数, 以及它们原来的排列顺序. 插入排序是 **平方**级算法, 也就是说, 在最坏的情况下, 算法的运行时间增长速度与元素个数的平方成正比. 这意味着如果元素个数变成原来的 2 倍, 那么运行时间将会是原来的 4 倍. 如果待排序元素碰巧就处于一种基本有序的状态, 那么程序的工作量就会少很多, 于是运行时间将会按照线性增长, 线性增长指的是与元素的个数成正比.

下面这副图显示了 isort 面对三种类型的数据时的性能变化情况, 这三种类型分别是: 逆序序列, 随机序列, 以及同一个元素组成的序列. 我们计算了比较和交换的次数, 对于一个排序过程来说, 这是两个很客观的指标. 正如你所看到的那样, 逆序序列拥有最差的性能, 随机序列居中, 而同一元素序列表现出了最佳的性能. 有序序列的性能表现 (在图中没有显示出来). 和同一元素序列非常接近



总得来说, 插入排序适用于元素个数较少的情况, 当元素个数过多时, 该算法的性能就会快速下降, 除非输入数据基本有序.

通过为每个排序函数添加两个计数器, 我们就可以为上面的图, 以及本章中的其他图生成所需要的数据. 其中一个计算比较的次数, 另一个计算交换的次数. 这是带有计数功能的 isort 函数:

```
function isort(A,n,     i,j,t) {  # insertion sort
    for (i = 2; i <= n; i++)      # with counters
        for (j = i; j > 1 && ++comp &&
          A[j-1] > A[j] && ++exch; j--) {
            # swap A[j-1] and A[j]
            t = A[j-1]; A[j-1] = A[j]; A[j] = t
        }
}
```

计数操作都放在内层 for 循环的条件判断部分完成. 由 && 连接的条件判断, 按照从左到右的顺序进行求值, 直到某一项为假. 表达式 ++comp 总是为真 (这里必须使用前缀形式的自增运算符), 于是, 数组中的元素每比较一次, comp 的值就加 1, 递增操作在比较之前完成. 当且仅当某两个元素被交换时, exch 的值才会加 1.

下面的程序用于组织测试, 以及为坐标图准备数据. 同样, 它的功能相当于一个微型编程语言, 可以灵活地指定参数.

```
# test framework for sort performance evaluation
#   input:  lines with sort name, type of data, sizes...
#   output: name, type, size, comparisons, exchanges, c+e

{   for (i = 3; i <= NF; i++)
        test($1, $2, $i)
}

function test(sort, data, n) {
    comp = exch = 0
    if (data ~ /rand/)
        genrand(A, n)
    else if (data ~ /id/)
        genid(A, n)
    else if (data ~ /rev/)
        genrev(A, n)
    else
        print "illegal type of data in", $0
    if (sort ~ /q.*sort/)
        qsort(A, 1, n)
    else if (sort ~ /h.*sort/)
        hsort(A, n)
    else if (sort ~ /i.*sort/)
        isort(A, n)
    else print
        "illegal type of sort in", $0
    print sort, data, n, comp, exch, comp+exch
}

# test-generation and sorting routines ...
```

输入数据是由多行组成的序列, 类似于

```
isort random 0 20 40 60 80 100
isort ident 0 20 40 60 80 100
```

输出数据的每一行都包括排序算法名称, 测试集类型, 测试集大小, 以及操作执行的次数. 输出数据被输送给绘图程序 grap, 它是我们在第 六 章讨论的绘图程序的原始版本.

Exercise 7.1 实际上, 函数 check 并不是一个很强大的测试, 什么样的错误会使它检测失败? 你会如何改进?

Exercise 7.2 我们的大多数测试都基于整数排序, 面对其他类型的数据时, isort 的表现会如何? 为了处理更一般的数据, 你会如何改进测试框架程序?

Exercise 7.3 我们默认每一种基本操作都需要相同的时间, 也就是说, 访问一个元素, 比较两个元素的值, 加法, 赋值等操作所消耗的时间是相同的. 对于 awk 程序来说这个假设是否合理? 写一个处理大量数据的程序, 来验证你的想法.

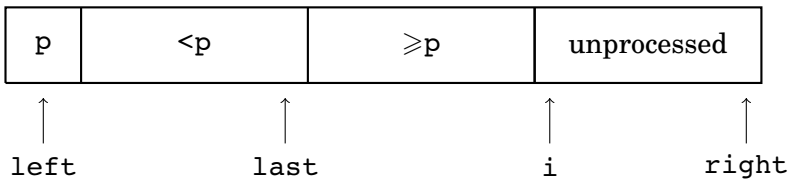
160

快速排序

基本概念. 快速排序是最高效的通用排序算法之一, 由 C. A. R. Hoare 在六十年代早期提出. 为了对一个元素序列进行排序, 快速排序算法把序列划分成两个子序列, 然后再递归地对子序列进行排序. 在划分阶段, 算法从序列中选取一个数作为划分点, 把剩下的元素分成两组: 小于划分元素的分在一个组中, 而大于或等于划分元素的分在另一个组中, 然后再对这两组递归调用快速排序. 如果一个序列所含的元素个数小于 2, 则可认为它已经是有序的了, 于是算法什么也不做就返回.

实现. 实现快速排序有多种方式, 这些方式的不同点就在于划分阶段. 我们所选择的实现方式比较容易理解, 虽然不是最快的. 因为算法是递归的, 所以我们会在划分操作作用在子数组 A[left], A[left+1], ..., A[right] 上时, 对划分进行描述.

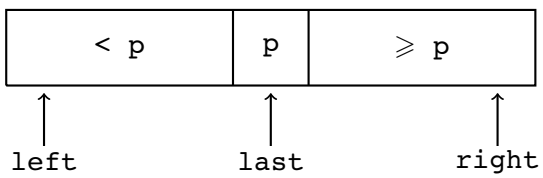
首先需要选择一个划分元素: 从范围 [left,right] 中随机选取一个数 r 作为划分点, 任何一个元素都可以作为划分点, 但是如果序列已经具有了某种程度上的有序, 那么随机选择的效果会更好. 位置 r 上的元素 p 现在成了划分元素. 交换 A[left] 与 A[r], 在划分的过程中, 数组始终把元素 p 放在 A[left], A[left] 的后面是比 p 小的元素, 再接下来是大于或等于 p 的元素, 最后是到目前为止未处理的元素:



下标 last 指向最后一个比 p 小的元素, 下标 i 指向下一个未处理的元素. 初始化时, last 等于 left, i 等于 left+1.

在划分过程的循环中, 我们比较元素 A[i] 与 p. 如果 A[i] ⩾ p, 只需要递增 i 即可; 如果 A[i] < p, 此时需要递增 last, 并交换 A[last] 与 A[i], 最后再递增 i. 按照这种方式, 一旦所有的元素都处理完毕, 我们需要交换 A[left] 与 A[last]. 到这里, 我们已经完成了一次划分, 此时的数组看起来就像这样:

161



现在, 我们对左边的子数组与右边的子组执行同样的操作.

假设我们对下列 8 个元素进行快速排序:

```
8 1 6 3 5 2 4 7
```

第一步我们可能选择 4 作为划分元素, 接下来, 划分操作会把数组重新排列成

```
2 1 3|4|5 6 8 7
```

然后再递归地对子数组 2 1 3 与 5 6 8 7 进行快速排序, 当子数组的元素个数少于 2 时, 递归过程就会停止.

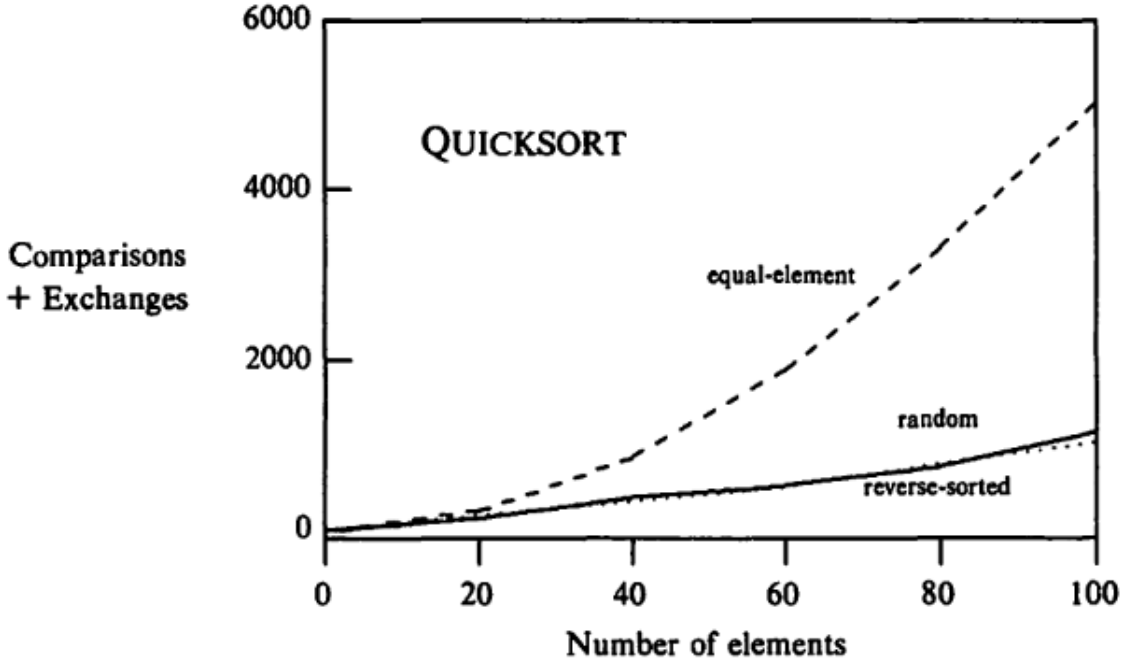
下面程序所包含的函数 qsort 实现了快速排序算法, 我们可以用插入排序的测试例程对该程序进行测试.

```
# quicksort

    { A[NR] = $0 }

END { qsort(A, 1, NR)
      for (i = 1; i <= NR; i++)
          print A[i]
    }

# qsort - sort A[left..right] by quicksort

function qsort(A,left,right,   i,last) {
     if (left >= right)  # do nothing if array contains
         return          # less than two elements
     swap(A, left, left + int((right-left+1)*rand()))
     last = left   # A[left] is now partition element
     for (i = left+1; i <= right; i++)
         if (A[i] < A[left])
             swap(A, ++last, i)
     swap(A, left, last)
     qsort(A, left, last-1)
     qsort(A, last+1, right)
}

function swap(A,i,j,   t) {
     t = A[i]; A[i] = A[j]; A[j] = t
}
```

性能. qsort 所执行的操作次数取决于数组划分时的均匀程度. 如果数组划分每次都很平均, 那么程序的运行时间与 $n \log n$ 成正比. 于是, 如果数据规模变为原来的两倍, 那么程序的运行时间只会在原

来两倍的基础上再稍微多出一点.

在最坏的情况下, 划分操作的结果会出现其中一个子数组长度为 0 的情况, 比如, 当所有元素都相同时, 就会出现这种情况. 这时候, 快速排序的时间复杂度就会退化到二次方. 幸运的是, 对于随机数据来说, 不会出现这种极不均匀的划分. 下面这张图显示了快速排序面对三种类型的输入数据时的性能表现 (在测试插入排序时, 也用到了这三种类型的数据). 正如你所看到的那样, 对于同元素序列来说, 操作次数的增长速度要比另外两种类型的增长速度快得多.



Exercise 7.4 为 `qsort` 添加计数语句, 计算比较操作和交换操作的执行次数. 你得到的结果是否和我们的类似?

Exercise 7.5 记录程序的运行时间, 而不是操作次数. 运行时间的统计图是否和操作次数的相同? 用大一点的例子作测试, 看看其统计图还是不是一样的.

堆排序

基本概念. **优先级队列** (*priority queue*) 是一种数据结构, 用于存储和检索元素. 它有两种基本操作: 往队列中插入一个新元素, 以及从队列中提取最大的元素. 这表明优先级队列可以用来排序: 首先把所有的元素插入到队列中, 然后每次抽取一个元素. 因为每次移除的都是最大的元素, 所以元素是以降序地方式从队列中抽取出来. 这种排序方法叫作堆排序, 由 J. W. J. Williams 和 R. W. Floyd 在 60 年代早期提出.

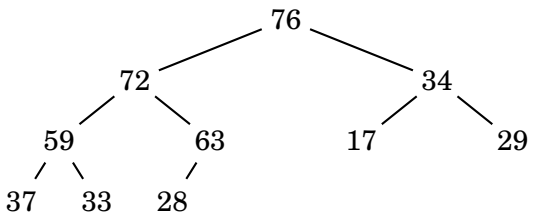
163

堆排序使用一种称为 **堆** (*heap*) 的数据结构来维护优先级队列, 我们可以把堆想像成一棵二叉树, 但是带有两条额外的性质:

1. 树是高度平衡的: 叶子结点最多只在两个不同的层次上出现, 另外, 位于最底层 (距离根结点最远的层次) 的叶子尽量靠左排列;

2. 树是部分有序的: 每个结点的值大于或等于它的孩子结点.

这是带有 10 个元素的堆:



堆具有两条非常重要的性质. 第一条性质是, 如果有 n 个结点, 那么所有的从根结点到叶子结点的路径长度都不会大于 $\log_2 n$. 第二条性质是, 具有最大值的元素总是在根结点 (这个位置称为 "堆顶").

如果我们用一个数组 A 来模拟堆, 那么就不需要构造显式的二叉树. 树中的结点按照宽度优先遍历的顺序出现在数组中, 也就是说, 根结点在 A[1], 它的左孩子与右孩子分别在 A[2] 和 A[3]. 总的来说, 如果某个结点在 A[i], 那么它的孩子就在 A[2i] 和 A[2i+1], 如果只有一个孩子的话, 那就在 A[2i]. 于是, 上面那棵树对应的数组 A 就是:

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]
76	72	34	59	63	17	29	37	33	28

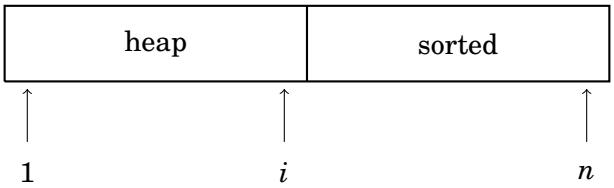
堆的部分有序性质指的是元素 A[i] 大于或等于它的孩子结点 A[2i] 与 A[2i+1], 如果只有一个孩子的话, 就那大于或等于 A[2i]. 如果某个数组满足这个条件, 我们就说这个数组具有 "堆属性".

实现 (*Implementation*). 堆排序由两个阶段组成: 构造堆, 以及按顺序从堆中提取元素. 两个阶段都要调用函数 heapify(A,i,j), 使得子数组 A[i], A[i+1], ..., A[j] 具有堆属性 (假设 A[i+1], ..., A[j] 已经具有了堆属性). heapify 的基本操作是比较 A[i] 与它的孩子, 如果 A[i] 没有孩子, 或者比它的孩子大, 那么函数就直接返回; 否则, 交换 A[i] 和它的最大孩子, 然后再对孩子重复这个基本操作.

在第一个阶段, 堆排序通过调用 heapify(A,i,n) (i 从 n/2 递减到 1), 把数组转化成一个堆. 164

第 2 个阶段开始时, i 被赋值为 n, 然后重复执行以下三个步骤: 首先, 把堆的最大元素 A[1] 与 A[i] 作交换, A[i] 是堆中最靠右的元素. 然后, 堆的元素个数减 1 (即 i 减 1). 这两个步骤相当于从堆中移除最大元素. 注意到, 这样做的结果是数组中最后的 n−i+1 个元素处于有序状态. 最后, 对数组 A 的前 i−1 个元素调用 heapify(A,1,i−1).

这三个步骤一直重复到堆中只剩下一个元素为止, 而这个元素其实就是序列中的最小值. 由于数组中剩下的元素按照升序排列, 所以当操作结束时, 序列就是有序的了. 在操作过程中, 数组看起来就像这样:



数组中从 1 到 i 的元素具有堆属性, 从 i+1 到 n 是数组中最大的 n−i 个元素, 按照升序排列. 开始时, i=n, 因此数组中没有已排序的部分.

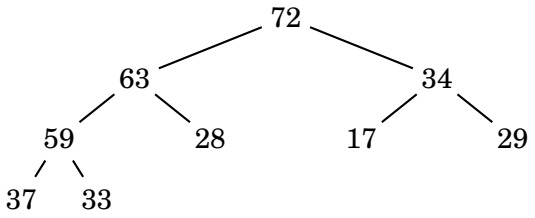
考虑上面展示的元素所组成的数组, 该数组已经具有堆属性. 在第二个阶段的第一个步骤, 我们交换元素 76 和 28:

```
28 72 34 59 63 17 29 37 33 | 76
```

在第 2 个步骤, 我们把堆的大小递减到 9. 然后在第 3 个步骤中, 通过一系列的交换操作, 把 28 移动到合适的位置, 从而维持住前 9 个元素的堆属性:

```
72 63 34 59 28 17 29 37 33 | 76
```

我们可以把这个过程可视化: 让元素 28 沿着二叉树的路径, 从根结点开始, 朝着叶子结点方向, 逐渐向下渗透, 直到到达这样一个的结点: 它的孩子小于或等于 28:



在下一次迭代中, 第一个步骤交换元素 72 和 33:

```
33 63 34 59 28 17 29 37 | 72 76
```

第二个步骤把 i 递减到 8, 第三次迭代把 33 移动到合适的位置: 165

```
63 59 34 37 28 17 29 33 | 72 76
```

下一次迭代以交换 63 与 33 作为开始, 迭代结束时, 序列变成:

```
59 37 34 33 28 17 29 | 63 72 76
```

当数组处于有序状态时, 迭代过程结束.

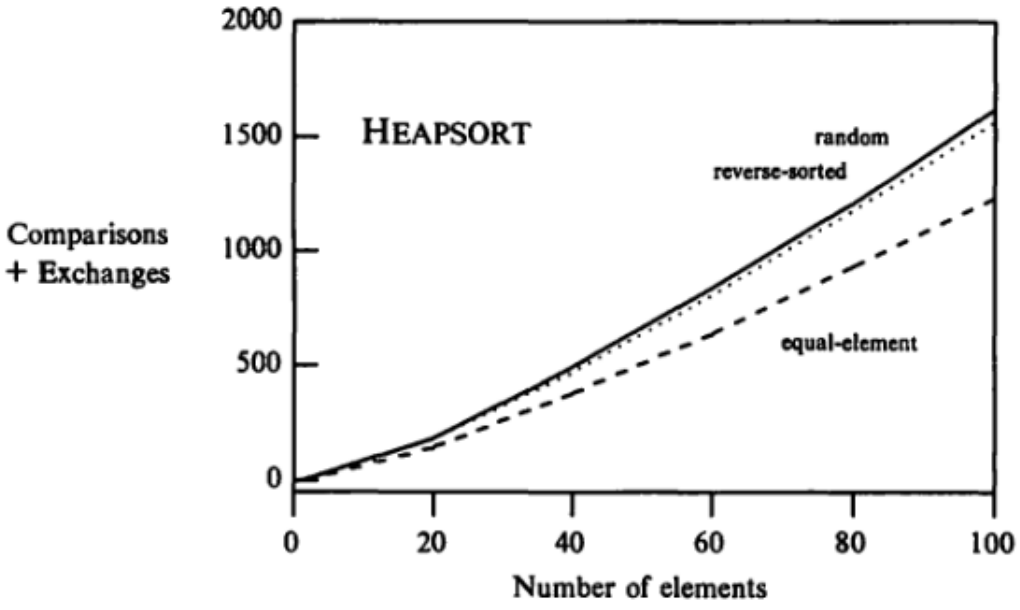
下面的程序对输入数据按照升序进行排列, 使用的正是上面提到的过程. 我们把大部分的, 由单独一条表达式组成的语句用花括号包围起来, 至于这样做的原因将会在下一节讨论剖析时说明.

```
# heapsort

    { A[NR] = $0 }

END { hsort(A, NR)
      for (i = 1; i <= NR; i++)
          { print A[i] }
    }

function hsort(A,n,  i) {
    for (i = int(n/2); i >= 1; i--)  # phase 1
         { heapify(A, i, n) }
```

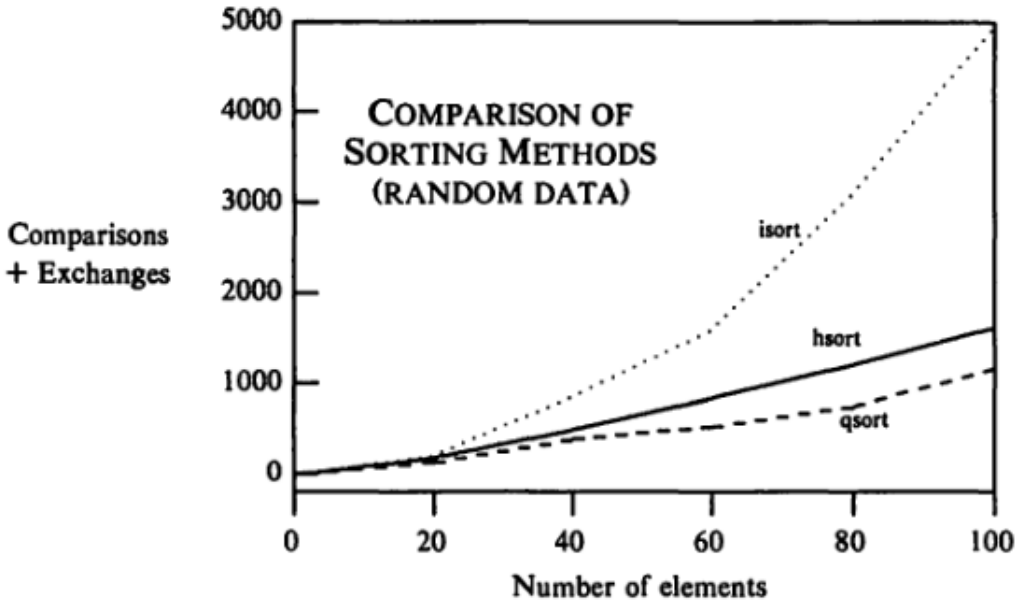
```
    for (i = n; i > 1; i--) {          # phase 2
         { swap(A, 1, i) }
         { heapify(A, 1, i-1) }
    }
}
function heapify(A,left,right,   p,c) {
    for (p = left; (c = 2*p) <= right; p = c) {
        if (c < right && A[c+1] > A[c])
            { c++ }
        if (A[p] < A[c])
            { swap(A, c, p) }
    }
}
function swap(A,i,j,   t) {
    t = A[i]; A[i] = A[j]; A[j] = t
}
```

性能. hsort 总的操作次数正比于 $n \log n$, 即使是在最坏的情况下也是如此. 同样, 我们用三种类型的元素序列对 hsort 进行性能测试, 下面这张图显示了测试结果, 可以看到, hsort 面对相同元素序列时, 性能优于快速排序.

166



接下来的这张图显示了本节所讨论的三种排序算法, 在面对随机输入数据时的性能表现.



前面曾经说过, `isort` 对随机数据进行排序的时间复杂度是二次的, 而 `hsort` 与 `qsort` 的复杂度却是 $n \log n$. 这张图清楚地展示了一个优秀算法的重要性: 随着数据规模的增加, 二次方程序的性能与 $n \log n$ 程序性能之间的差距急剧加大.

167

Exercise 7.6 `check` 总是发现 `isort` 的输出是有序的. 对于 `qsort` 与 `hsort` 是否也成立? 如果输入数据仅仅由数值组成, 或者是一些看起来不太像数值的字符串, 那么是否还成立?

7.2　剖析

在上一节中, 为了评价排序程序的性能, 我们计算了特定操作的执行次数. 评价程序性能的另一个有效方法是对程序进行剖析[①], 也就是计算每条语句的执行次数. 许多编程环境都会提供一个称为 **剖析器** (*profiler*) 的工具程序, 剖析器可以打印某个程序中每条语句的执行次数.

我们并没有为 awk 开发一个剖析器, 但是在这一节, 我们将展示如何使用两个简短的程序, 来近似地模拟剖析器的功能. 第一个程序 `makeprof` 通过在代码中插入计数与打印语句, 为 awk 程序构造剖析版本. 当剖析版的程序运行时, 它会计算每条语句的执行次数, 并把计数结果写到文件 `prof.cnts` 中. 第二个程序 `printprof` 从 `prof.cnts` 中获取语句的执行次数, 并添加到原版程序中.

为了简单起见, 当程序运行时, 我们只对 "可执行的" 的左花括号进行计数. 通常来说, 这种计数安排已经足够反映问题了, 因为每一个动作, 以及每一块组合语句都被一对花括号包围起来. 任何一条语句都可以用一对花括号包围起来, 所以, 通过为语句加花括号, 我们可以按照自己的需要来控制计数的精度.

程序 `makeprof` 把一个普通的 awk 程序转换成一个剖析版程序. 它在每个输入行的左花括号的右边插入计数语句, 计数语句具有形式:

```
_LBcnt[i]++;
```

除此之外, 它还会添加一个 `END` 动作, 用于输出计数结果到文件 `prof.cnts` 中, 每行一个.

[①] "profiling" 这个词很难翻译, 我就把它译作 "部析" 得了. — 译者注

```
# makeprof - prepare profiling version of an awk program
#   usage:  awk -f makeprof awkprog >awkprog.p
#   running awk -f awkprog.p data creates a
#       file prof.cnts of statement counts for awkprog

    { if ($0 ~ /{/) sub(/{/, "{ _LBcnt[" ++_numLB "]++; ")
      print
    }

END { printf("END { for (i = 1; i <= %d; i++)\n", _numLB)
      printf("\t\t print _LBcnt[i] > \"prof.cnts\"\n}\n")
    }
```

给定一些输入数据, 运行完某个程序的剖析版之后, 我们可以用 printprof, 把 protf.cnts 中的语句计数结果附加到原始程序中: 168

```
# printprof - print profiling counts
#     usage:  awk -f printprof awkprog
#     prints awkprog with statement counts from prof.cnts

BEGIN { while (getline < "prof.cnts" > 0) cnt[++i] = $1 }
/{/   { printf("%5d", cnt[++j]) }
      { printf("\t%s\n", $0) }
```

作为示例, 考虑为 7.1 节末尾的程序 heapsort 作剖析. 为了构造这个程序的剖析版, 键入并执行命令

```
awk -f makeprof heapsort > heapsort.p
```

构造出的剖析版 heapsort.p 看起来就像:

```
# heapsort

    { _LBcnt[1]++;  A[NR] = $0 }

END { _LBcnt[2]++;  hsort(A, NR)
      for (i = 1; i <= NR; i++)
          { _LBcnt[3]++;  print A[i] }
    }

function hsort(A,n,  i) { _LBcnt[4]++;
    for (i = int(n/2); i >= 1; i--)  # phase 1
         { _LBcnt[5]++;  heapify(A, i, n) }
    for (i = n; i > 1; i--) { _LBcnt[6]++;         # phase 2
```

```
            { _LBcnt[7]++;  swap(A, 1, i) }
            { _LBcnt[8]++;  heapify(A, 1, i-1) }
        }
    }
    function heapify(A,left,right,   p,c) { _LBcnt[9]++;
        for (p = left; (c = 2*p) <= right; p = c) { _LBcnt[10]++;
            if (c < right && A[c+1] > A[c])
                { _LBcnt[11]++;  c++ }
            if (A[p] < A[c])
                { _LBcnt[12]++;  swap(A, c, p) }
        }
    }
    function swap(A,i,j,   t) { _LBcnt[13]++;
        t = A[i]; A[i] = A[j]; A[j] = t
    }
    END { for (i = 1; i <= 13; i++)
                    print _LBcnt[i] > "prof.cnts"
    }
```

正如你所看到的那样, 原版程序中插入了 13 条计数语句, 还额外添加了一个 END 动作, 该动作把计数结果输出到文件 prof.cnts. 多个 END 动作等价于按照它们出现的顺序, 组合到一个单独的 END 中. 169

现在, 假设把 100 个随机数作为输入数据, 运行 heapsort.p, 程序运行结束后, 键入并执行命令

```
awk -f printprof heapsort
```

就可以得到带有语句执行次数的原版程序 (每条语句的执行次数仅对本次运行有效). 命令的执行结果是:

```
        # heapsort

  100       { A[NR] = $0 }

    1   END { hsort(A, NR)
              for (i = 1; i <= NR; i++)
  100             { print A[i] }
          }

    1   function hsort(A,n,  i) {
          for (i = int(n/2); i >= 1; i--)  # phase 1
   50           { heapify(A, i, n) }
   99       for (i = n; i > 1; i--) {        # phase 2
   99           { swap(A, 1, i) }
   99           { heapify(A, 1, i-1) }
```

```
            }
        }
  149   function heapify(A,left,right,   p,c) {
  526       for (p = left; (c = 2*p) <= right; p = c) {
                if (c < right && A[c+1] > A[c])
  228               { c++ }
                if (A[p] < A[c])
  473               { swap(A, c, p) }
            }
        }
  572   function swap(A,i,j,   t) {
            t = A[i]; A[i] = A[j]; A[j] = t
        }
```

我们实现的剖析器最重要的特点是简单, 但这同时也是它最大的缺点. 程序 makeprof 盲目地在每一行的第一个左花括号的右边插入计数语句, 一个更好的做法是忽略字符串常量, 正则表达式和注释中的左花括号. 除了语句的执行次数, 最好还要报告语句的执行时间, 但是这里介绍的方法还无法做到这一点.

Exercise 7.7 修改剖析器, 使得程序不会在字符串常量, 正则表达式或者注释中插入计数语句. 你实现的剖析器是否允许对自身进行剖析?

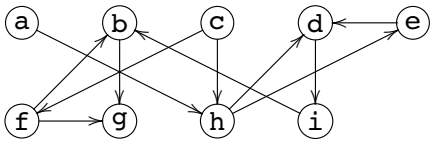
Exercise 7.8 如果 END 动作中包含 exit 语句, 那么剖析器就无法正常地工作, 请问这是为什么? 修复这个问题.

170

7.3　拓扑排序

在工程建筑项目中, 有些工作必须在其他工作开始前完成. 为了方便说明, 我们通常会把这些工作都列出来, 这样就可以很容易地看出哪些工作必须在其他工作完成后才能开始. 在一个程序库中, 程序 a 可能会调用程序 h, h 也有可能会调用程序 d 和 e, 等等. 我们可能希望对程序进行排序, 使得程序在它调用的所有程序之前出现 (Unix 命令 lorder 可以完成这个工作). 这些, 以及其他类似的问题都是 **拓扑排序** (*topological sorting*) 问题的实例: 找到一个顺序, 该顺序满足一个约束条件集合, 集合中的每个条件都具有形式 "x 必须在 y 之前出现". 由约束条件集合所表示的偏序得到该集合上的一个线序, 这个操作叫作拓扑排序.

约束条件可以用一张图来表示, 图中的每个结点都用名字标记, 如果结点 x 必须先于 y 出现, 那么就有一条边, 从 x 指向 y. 下面这张图就是一个例子:



如果存在一条边, 从 x 指向 y, 那么 x 就是 y 的 **前驱** (*predecessor*), y 是 x 的 **后继** (*successor*). 假设约束条件用 前驱-后继 对来表示, 其中每个输入行的 x 和 y 表示一条从结点 x 指向 y 的边, 上面那张图的文本表示形式是:

```
a   h
b   g
c   f
c   h
d   i
e   d
f   b
f   g
h   d
h   e
i   b
```

如果存在一条从 x 指向 y 的边, 那么在输出中 x 必须先于 y 出现. 如果把上面的数据作为输入, 那么可能的一个输出是:

```
a c f h e d i b g
```

除此之外, 还有其他线序关系也包含了上图所描述的偏序, 另一个可能的线序是:

```
c a h e d i f b g
```

拓扑排序的目标是对图中的结点进行排序, 使得所有前驱都在它们的后继之前出现. 当且仅当图中 171
不含有 **环** (*cycle*, 是由多条边组成的序列, 当结点沿着这个序列前进时, 最终会回到结点的原来位置) 时, 这样的顺序才存在.

广度优先拓扑排序

有多种算法可以用于图的拓扑排序, 其中最简单的算法可能是 — 在每次迭代时, 从图中移除没有前驱的结点. 如果所有的结点都按照这种方式从图中移除出来, 那么图的拓扑排序结果就是结点按照移除顺序所组成的序列. 在上面的图中, 我们可以从移除结点 a 及以它为起点的边开始, 然后移除结点 c, 再接着是结点 f 或 h, 依此类推.

我们的实现使用了一种先进先出的数据结构 — **队列** (*queue*) — 来确定不含有前驱的结点的处理顺序, 以一种 "广度优先" 的方式进行. 当所有的结点都被读取完毕后, 一个循环体计算结点的数量, 并把所有的, 没有前驱的结点插入到队列中. 第二个循环体移除队列的第一个结点, 打印该结点的名字, 然后递减它的每一个后继结点的前驱结点个数. 如果某一个后继结点的前驱结点个数变为 0, 那就把这个后继结点插入到队尾. 当队列变成空, 并且所有的结点都被访问过, 这时候拓扑排序就完成了. 但是, 如果有某些结点从未被插入到队列中, 那就说明这些结点在一个环中, 因此无法对该图进行拓扑排序. 如果图中不存在环, 那么被打印出来的结点序列就是一个拓扑排序.

tsort 的前三条语句从输入中读取 前驱-后继对, 并构造一个后继结点列表, 就像:

node	pcnt	scnt	slist
a	0	1	h
b	2	1	g
c	0	2	f, h
d	2	1	i
e	1	1	d
f	1	2	b, g
g	2	0	
h	2	2	d, e
i	1	1	b

数组 pcnt 和 scnt 记录每个结点的前驱结点与后继结点个数, slist[x,i] 给出了结点 x 的第 i 个后继结点的名字. 如果某个元素原来不在数组 pcnt 中, 那么程序的第一行就会为它创建一个元素. 172

```
# tsort - topological sort of a graph
#   input:  predecessor-successor pairs
#   output: linear order, predecessors first

    { if (!($1 in pcnt))
          pcnt[$1] = 0           # put $1 in pcnt
      pcnt[$2]++                 # count predecessors of $2
      slist[$1, ++scnt[$1]] = $2 # add $2 to successors of $1
    }
END { for (node in pcnt) {
          nodecnt++
          if (pcnt[node] == 0)   # if it has no predecessors
              q[++back] = node   # queue node
      }
      for (front = 1; front <= back; front++) {
          printf(" %s", node = q[front])
          for (i = 1; i <= scnt[node]; i++)
              if (--pcnt[slist[node, i]] == 0)
                  # queue s if it has no more predecessors
                  q[++back] = slist[node, i]
      }
      if (back != nodecnt)
          print "\nerror: input contains a cycle"
      printf("\n")
    }
```

在 awk 中实现队列非常简单: 只需要一个拥有两个下标的数组, 一个指向队头, 一个指向队尾.

Exercise 7.9 修改 tsort, 使得它可以处理图中孤立的结点.

深度优先搜索

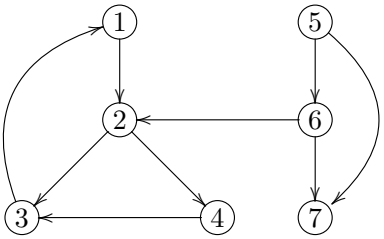
为了说明深度优先搜索技术, 我们将再构造一个拓扑排序程序. 深度优先搜索可以用来解决与图相关的许多问题, 包括 Unix 实用程序 make 中蕴含的图问题. 深度优先搜索是另一种遍历图结点的系统化方法, 即使是在含有环的图中. 在最简单的形式中, 深度优先搜索只是一个递归过程:

```
dfs(node):
    mark node visited
    for all unvisited successors s of node do
        dfs(s)
```

之所以称为 "深度优先", 是因为搜索从一个结点开始, 然后访问该结点的某个未被访问过的后继结点, 再接着访问该后继结点的某个未被访问过的后继结点, 依此类推, 基本原则是尽可能快地向图的纵向深入. 如果某个结点的所有后继结点都被访问过了, 搜索过程就回退到前驱结点, 然后再按照深度优先的方法, 搜索前驱结点的另一个未被访问过的后继结点.

173

考虑下面的这张图. 如果遍历从结点 1 开始, 那么深度优先搜索将会陆续访问结点 1, 2, 3 和 4. 这时候, 如果从另一个未访问过的结点开始, 比如 5, 那么接下来将会访问结点 5, 6 和 7. 如果遍历从不同的结点开始, 就会得到不同的结点访问序列.



深度优先搜索可以用来判断图中是否含有环. 如果有一条边指向的是之前访问过的祖先结点, 比如 (3, 1), 那么这条边就叫作**回边** (*back edge*). 因为存在回边等同于存在环, 所以为了判断图中是否含有环, 我们只需要搜索回边就够了. 下面的函数判断某个图中是否存在结点 node 可到达的环, 图用后继结点列表来表示, 这个数据结构我们已经在 tsort 中见过.

```
# dfs - depth-first search for cycles

function dfs(node,     i, s) {
    visited[node] = 1
    for (i = 1; i <= scnt[node]; i++)
        if (visited[s = slist[node, i]] == 0)
            dfs(s)
        else if (visited[s] == 1)
            print "cycle with back edge (" node ", " s ")"
    visited[node] = 2
}
```

这个函数使用数组 visited 来判断某个结点是否被访问过. 开始时, visited[x] 等于 0. 在第一次进入结点 x 时, dfs 把 visited[x] 设置为 1, 最后一次离开结点 x 时, 把 visited[x] 设置为 2. 在

遍历过程中, dfs 利用 visited 来判断结点 y 是否是当前结点的祖先 (以及它之前是否被访问过). 如果 y 是当前结点的祖先, 那么 visited[y] 等于 1, 如果 y 之前被访问过, 那么 visited[y] 的值就等于 2.[①]

深度优先拓扑排序

函数 dfs 很容易就可以修改为结点排序例程. 从某结点开始的搜索过程一旦结束, 如果在此时打印该结点的名字, 那么所打印出来的结点序列刚好就是逆序的拓扑排序 (前提是图中不含有环). 给定一个前驱-后续 对序列作为输入数据, 程序 rtsort 输出图的逆序拓扑排序, 它对每一个没有前驱的结点应用深度优先搜索, 程序中使用的数据结构与 tsort 中的相同.

174

```
# rtsort - reverse topological sort
#   input:  predecessor-successor pairs
#   output: linear order, successors first

    { if (!($1 in pcnt))
          pcnt[$1] = 0           # put $1 in pcnt
      pcnt[$2]++                 # count predecessors of $2
      slist[$1, ++scnt[$1]] = $2 # add $2 to successors of $1
    }

END { for (node in pcnt) {
          nodecnt++
          if (pcnt[node] == 0)
              rtsort(node)
      }
      if (pncnt != nodecnt)
          print "error: input contains a cycle"
      printf("\n")
    }

function rtsort(node,     i, s) {
    visited[node] = 1
    for (i = 1; i <= scnt[node]; i++)
        if (visited[s = slist[node, i]] == 0)
            rtsort(s)
        else if (visited[s] == 1)
            printf("error: nodes %s and %s are in a cycle\n",
                s, node)
    visited[node] = 2
    printf(" %s", node)
```

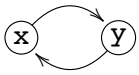
[①] TODO 需要仔细检查这段话 — 译者注

```
        pncnt++     # count nodes printed
    }
```

如果把本节开头给出的 前驱-后继 对作为输入数据, 这个程序将会输出:

```
g b i d e h a f c
```

需要注意的是, 对图中的某些环来说, 这个程序可以通过查找回边来显式地探测, 而对另一些环的探测则是隐式地, 对于后一种情况, 程序将无法打印出全部的结点, 就像这幅图:



Exercise 7.10 修改 rtsort, 按照通常的顺序打印结点, 也就是先打印前驱结点. 你是否可以在不修改 rtsort 的前提下达到同样的效果?

175

7.4 Make: 文件更新程序

大型程序中包含的声明与子例程通常分布在多个文件中, 通过一系列复杂的处理步骤来生成最终的程序. 一篇复杂的文档 (比如本章) 可能由许多图表和程序组成, 这些图表和程序代码存放在不同的文件中, 而且程序会被运行和测试, 最终将通过一系列互相依赖的操作来生成可打印的文档. 如果有一个自动更新工具能够在消耗最小的人力物力的前提下, 完成这些处理工作, 那么这个工具的价值将是无法衡量的. 本节将仿照 Unix 命令 make, 开发一个具备基本功能的更新程序, 基本算法是上节讨论过的深度优先搜索.

为了使用更新程序, 用户必须显式地描述出系统的组成成分, 各个成分之间的依赖关系, 以及构造它们的命令. 我们假设依赖关系和构造命令存放在一个文件中, 文件名是 makefile, 里面包含了一系列规则, 每条规则都具有形式:

name: $t_1\ t_2\ ...\ t_n$
　　command

规则的第一行是依赖关系, 意思是程序或文件 *name* 依赖于目标 $t_1, t_2, ..., t_n$, 其中 t_i 是文件名或另一个 *name*. 依赖关系的下面是一行或多行 *command*, 这些 *command* 是用来生成 *name* 的的命令. 下面显示的是某个小程序的 makefile 文件, 程序由两个 C 文件和一个 yacc 语法文件组成 (yacc 是一个典型的用于程序开发的应用程序):

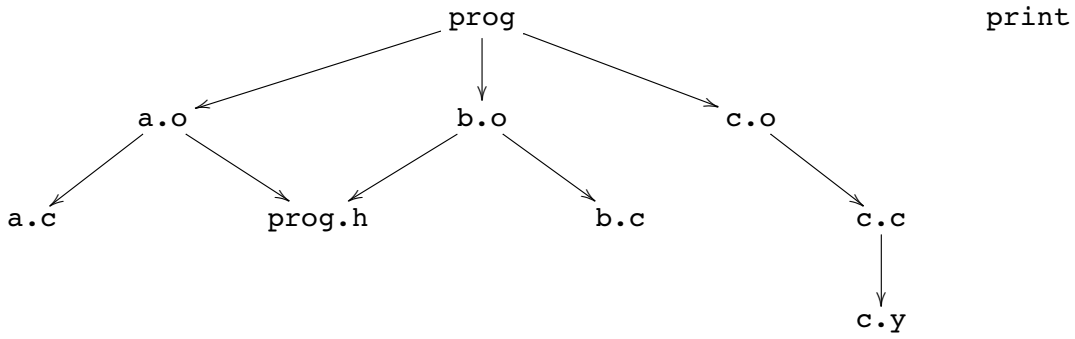
```
prog:       a.o b.o c.o
            cc a.o b.o c.o -ly -o prog
a.o:        prog.h a.c
            cc -c prog.h a.c
b.o:        prog.h b.c
            cc -c prog.h b.c
c.o:        c.c
            cc -c c.c
```

```
c.c:        c.y
            yacc c.y
            mv y.tab.c c.c
print:
            pr prog.h a.c b.c c.y
```

第 1 行表示 prog 依赖于目标文件 a.o, b.o 和 c.o. 第 2 行是说使用 C 编译器 cc 链接 a.o, b.o, c.o 和一个库文件来生成 prog. 下一条规则 (第 3 行) 表示 a.o 依赖于目标 prog.h 和 a.c, 通过编译这 2 个文件来生成 a.o; b.o 是类似的. 文件 c.o 依赖于 c.c, 而后者依赖于 c.y, c.y 将会被 yacc 处理. 最后, 名字 print 不依赖于任何目标, 按照惯例, 对于没有目标的名字, make 总是会执行后面的命令, 在这里是用命令 pr 打印所有的源代码文件.

176

makefile 中的依赖关系可以用一张图来表示, 如果依赖性规则左边的 x 依赖于右边的目标 y, 那么在图中就有一条边从结点 x 指向结点 y. 如果某条规则没有目标, 就在图中创建一个没有后继的结点, 并用规则左边的名字来命名. 上文的 makefile 对应的依赖图是:



如果在 x 的最后一次修改之后, y 才会被修改, 那么, 我们就认为 x 比 y **老** (*older*). 为了跟踪年龄, 对每一个 x 都有一个对应的整数 age[x], 后者表示自从 x 上一次被修改之后, 已经过了多长时间. 年龄值越大, 说明文件越老: 如果 age[x] $\geqslant$ age[y], 则 x 比 y 老.

如果存在一条依赖关系:

```
n: a b c
```

那么在更新 n 之前, 需要先更新 a, b 和 c, 而在更新后面 3 个时, 也需要按照依赖关系来进行. 如果 makefile 中的某个目标既不是一个名字, 也不是某个已存在的文件, 程序就报错并退出. 否则的话, 检查目标的年龄, 如果至少有一个目标的年龄比 n 的年龄小 (也就是说, n 比某些依赖还老), 我们就执行依赖关系下面的命令. 命令执行完毕后, 重新计算所有对象的年龄. 如果有一条规则是:

```
print:
            pr prog.h a.c b.c c.y
```

这是一条没有目标的规则, 对于这些规则, 我们总是执行和该规则相关的命令, 并更新对象的年龄.

程序 make 接收一个 *name* 作为输入参数, 并采用下面的算法来更新 *name*:

1. 在 makefile 中搜索 *name* 对应的规则, 并递归地更新依赖关系右边的目标 $t_1, t_2, ..., t_n$, 如果某个 t_i 不是一个名字, 或者文件 t_i 不存在, 程序就会中止更新过程.

2. 在更新完所有的 t_i 之后, 如果当前的 *name* 比某个 t_i 老, 又或者是 *name* 没有目标, 程序就会执行依赖关系下面的命令.

177

在本质上, 本节所使用的方法和上一节的方法是一样的, make 根据 makefile 中的依赖关系来构造依赖图, 它使用 Unix 命令

```
ls -t
```

对文件进行排序 (越新的文件越靠前), 排序的依据是文件最后的修改时间. 文件名将作为数组 age 的索引, 而数组元素的值则是文件在排序序列中的名次, 最老的文件拥有最大值的名次. 如果规则中的名字不是当前目录中的文件名, make 就把它的时间设置成一个很大的值.

最后, make 使用上一节讲过的深度优先搜索来遍历依赖图. 在节点 n, make 遍历 n 的后继节点, 如果存在某个后继节点, 它的年龄比 n 的年龄小, make 就执行规则中的命令, 并计算出一套新的年龄. 如果 make 发现某个名字的依赖关系存在环, 程序就会报错并中止更新过程.

为了说明 make 如何工作, 假设我们现在是第一次键入并执行命令

```
make prog
```

make 将会执行以下命令序列:

```
cc -c prog.h a.c
cc -c prog.h b.c
yacc c.y
mv y.tab.c c.c
cc -c c.c
cc a.o b.o c.o -ly -o prog
```

如果我们对 b.c 作了修改, 然后再次键入

```
make prog
```

make 只会执行

```
cc -c prog.h b.c
cc a.o b.o c.o -ly -o prog
```

因为在上一次创建 prog 之后, 其他文件并没有被修改过, 所以 make 并不会对它们进行处理. 最后, 如果我们再次执行

```
make prog
```

make 将会输出

```
prog is up to date
```

因为文件没有发生变化, 所以什么也不需要做.

178

```
# make - maintain dependencies
```

```
BEGIN {
    while (getline <"makefile" > 0)
        if ($0 ~ /^[A-Za-z]/) {  #  $1: $2 $3 ...
            sub(/:/, "")
            if (++names[nm = $1] > 1)
                error(nm " is multiply defined")
            for (i = 2; i <= NF; i++) # remember targets
                slist[nm, ++scnt[nm]] = $i
        } else if ($0 ~ /^\t/)        # remember cmd for
            cmd[nm] = cmd[nm] $0 "\n" #   current name
        else if (NF > 0)
            error("illegal line in makefile: " $0)
    ages()      # compute initial ages
    if (ARGV[1] in names) {
        if (update(ARGV[1]) == 0)
            print ARGV[1] " is up to date"
    } else
        error(ARGV[1] " is not in makefile")
}

function ages(      f,n,t) {
    for (t = 1; ("ls -t" | getline f) > 0; t++)
        age[f] = t   # all existing files get an age
    close("ls -t")
    for (n in names)
        if (!(n in age))   # if n has not been created
            age[n] = 9999  # make n really old
}
function update(n,   changed,i,s) {
    if (!(n in age)) error(n " does not exist")
    if (!(n in names)) return 0
    changed = 0
    visited[n] = 1
    for (i = 1; i <= scnt[n]; i++) {
        if (visited[s = slist[n, i]] == 0) update(s)
        else if (visited[s] == 1)
            error(s " and " n " are circularly defined")
        if (age[s] <= age[n]) changed++
    }
    visited[n] = 2
```

```
        if (changed || scnt[n] == 0) {
            printf("%s", cmd[n])
            system(cmd[n])  # execute cmd associated with n
            ages()          # recompute all ages
            age[n] = 0      # make n very new
            return 1
        }
        return 0
    }
    function error(s) { print "error: " s; exit }
```

179

Exercise 7.11 在示例中, 函数 ages 会执行多少次?

Exercise 7.12 添加一些参数或宏替换功能, 使得我们可以很容易地对规则进行修改.

Exercise 7.13 为常见的更新操作添加隐式规则, 例如, 使用 cc 编译 .c 文件来生成 .o 文件. 你将会用什么方法来表示隐式规则, 使得用户可以对这些规则进行修改?

7.5　小结

这一章更多地是在讨论基本的算法知识, 而不是 awk, 然而, 算法的确是一门非常有用的知识, 而且我们也希望读者能够学习到如何使用 awk 来支持其他程序的实验工作.

我们还讨论了脚手架程序. 通常来说, 和执行一个单独的测试比起来, 写一个小程序来生成和控制测试 (或调试) 并不会多花很多时间, 但是脚手架程序可以被重复地使用, 因此可以把工作做得更加彻底.

最后一点则比较普通, 我们已经在前面说过很多遍了. Awk 经常用于从某个程序的输出中提取数据, 并转换成另一种格式, 例如, 读者已经见识过我们如何把排序的测试结果转换成 grap 的输入, 以及如何把计数语句插入到原始程序中.

参考资料

我们的快速排序, 堆排序和拓扑排序程序来自于 Jon Bentley, 他同时也是脚手架和剖析程序的灵感来源, 关于这些, 读者可以进一步阅读 *Communications of the ACM* 6 月和 7 月的 *Programming Pearls* 专栏. 关于排序和搜索算法的扩展讨论及分析, 请阅读 *The Art of Computer Programming* 第 3 卷: 搜索与排序 (D. E. Knuth 著, Addison-Wesley 1973 年出版), 或者是 *The Design and Analysis of Computer Algorithms* (Aho, Hopcroft 和 Ullman 著, Addison-Wesley 1974 年出版).

Unix 程序 make 最早由 Stu Feldman 开发, 关于该程序的首篇文章载于 1979 年 4 月的 *Software — Practice and Experience*. 更多的关于 make 的讨论, 请参考 W. Miller 和 E. Myers 撰写的 "Side-effect in Automatic File Updating", 载于 *Software — Practice and Experience*, 1986 年 9 月.