D

Basic Programming Concepts

This appendix attempts to define some of the basic concepts and terms that are used throughout the rest of this book. As this book is specifically about *awk*, and not about computer programming in general, the coverage here is by necessity fairly cursory and simplistic. (If you need more background, there are many other introductory texts that you should refer to instead.)

What a Program Does

At the most basic level, the job of a program is to process some input data and produce results. This is shown graphically in Figure D-1.

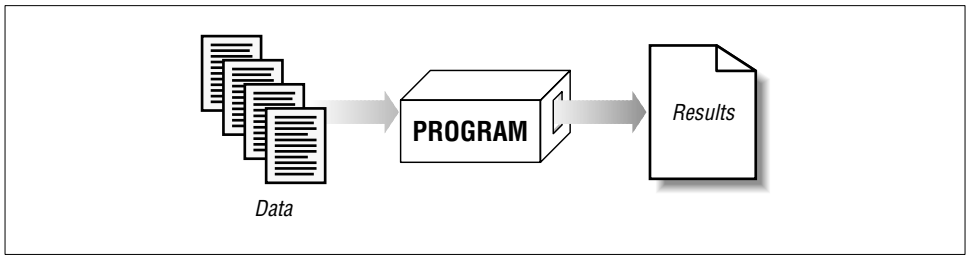

Figure D-1. The basic job of a program

The "program" in the figure can be either a compiled program* (such as *ls*), or it may be *interpreted*. In the latter case, a machine-executable program such as *awk* reads your program, and then uses the instructions in your program to process the data.

* Compiled programs are typically written in lower-level languages such as C, C++, Fortran, or Ada, and then translated, or *compiled*, into a form that the computer can execute directly.

When you write a program, it usually consists of the following, very basic set of steps, as shown in Figure D-2:

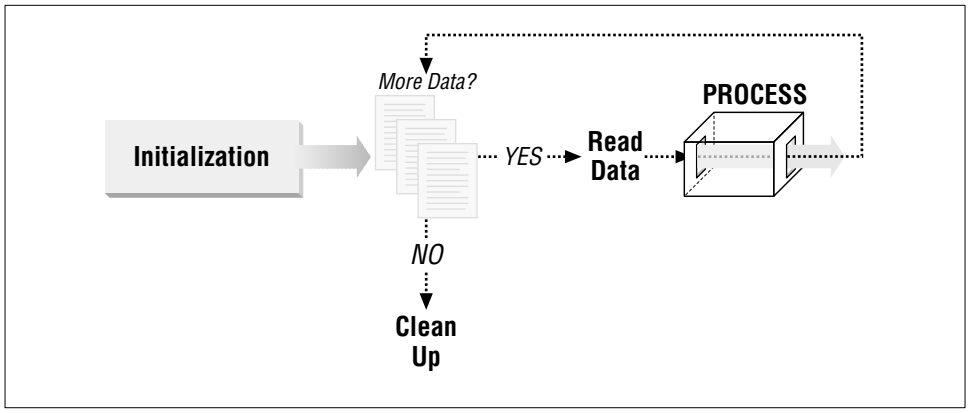

Figure D-2. The basic outline of a program

Initialization

These are the things you do before actually starting to process data, such as checking arguments, initializing any data you need to work with, and so on. This step corresponds to *awk*'s `BEGIN` rule (see the section "The BEGIN and END Special Patterns" in Chapter 6, *Patterns, Actions, and Variables*).

If you were baking a cake, this might consist of laying out all the mixing bowls and the baking pan, and making sure you have all the ingredients that you need.

Processing

This is where the actual work is done. Your program reads data, one logical chunk at a time, and processes it as appropriate.

In most programming languages, you have to manually manage the reading of data, checking to see if there is more each time you read a chunk. *awk*'s pattern-action paradigm (see Chapter 1, *Getting Started with awk*) handles the mechanics of this for you.

In baking a cake, the processing corresponds to the actual labor: breaking eggs, mixing the flour, water, and other ingredients, and then putting the cake into the oven.

Clean Up

Once you've processed all the data, you may have things you need to do before exiting. This step corresponds to *awk*'s `END` rule (see the section "The BEGIN and END Special Patterns" in Chapter 6).

> After the cake comes out of the oven, you still have to wrap it in plastic wrap to keep anyone from tasting it, as well as wash the mixing bowls and utensils.

An *algorithm* is a detailed set of instructions necessary to accomplish a task, or process data. It is much the same as a recipe for baking a cake. Programs implement algorithms. Often, it is up to you to design the algorithm and implement it, simultaneously.

The "logical chunks" we talked about previously are called *records*, similar to the records a company keeps on employees, a school keeps for students, or a doctor keeps for patients. Each record has many component parts, such as first and last names, date of birth, address, and so on. The component parts are referred to as the *fields* of the record.

The act of reading data is termed *input*, and that of generating results, not too surprisingly, is termed *output*. They are often referred to together as "input/output," and even more often, as "I/O" for short. (You will also see "input" and "output" used as verbs.)

awk manages the reading of data for you, as well as the breaking it up into records and fields. Your program's job is to tell *awk* what to with the data. You do this by describing *patterns* in the data to look for, and *actions* to execute when those patterns are seen. This *data-driven* nature of *awk* programs usually makes them both easier to write and easier to read.

Data Values in a Computer

In a program, you keep track of information and values in things called *variables*. A variable is just a name for a given value, such as `first_name`, `last_name`, `address`, and so on. *awk* has several predefined variables, and it has special names to refer to the current input record and the fields of the record. You may also group multiple associated values under one name, as an array.

Data, particularly in *awk*, consists of either numeric values, such as 42 or 3.1415927, or string values. String values are essentially anything that's not a number, such as a name. Strings are sometimes referred to as *character data*, since they store the individual characters that comprise them. Individual variables, as well as numeric and string variables, are referred to as *scalar* values. Groups of values, such as arrays, are not scalars.

Within computers, there are two kinds of numeric values: *integers* and *floating-point*. In school, integer values were referred to as "whole" numbers—that is, numbers without any fractional part, such as 1, 42, or −17. The advantage to integer numbers is that they represent values exactly. The disadvantage is that their range is limited. On most modern systems, this range is −2,147,483,648 to

2,147,483,647.

Integer values come in two flavors: *signed* and *unsigned*. Signed values may be negative or positive, with the range of values just described. Unsigned values are always positive. On most modern systems, the range is from 0 to 4,294,967,295.

Floating-point numbers represent what are called "real" numbers; i.e., those that do have a fractional part, such as 3.1415927. The advantage to floating-point numbers is that they can represent a much larger range of values. The disadvantage is that there are numbers that they cannot represent exactly. *awk* uses *double-precision* floating-point numbers, which can hold more digits than *single-precision* floating-point numbers. Floating-point issues are discussed more fully in the section "Floating-Point Number Caveats" later in this appendix.

At the very lowest level, computers store values as groups of binary digits, or *bits*. Modern computers group bits into groups of eight, called *bytes*. Advanced applications sometimes have to manipulate bits directly, and *gawk* provides functions for doing so.

While you are probably used to the idea of a number without a value (i.e., zero), it takes a bit more getting used to the idea of zero-length character data. Nevertheless, such a thing exists. It is called the *null string*. The null string is character data that has no value. In other words, it is empty. It is written in *awk* programs like this: `""`.

Humans are used to working in decimal; i.e., base 10. In base 10, numbers go from 0 to 9, and then "roll over" into the next column. (Remember grade school? 42 is 4 times 10 plus 2.)

There are other number bases though. Computers commonly use base 2 or *binary*, base 8 or *octal*, and base 16 or *hexadecimal*. In binary, each column represents two times the value in the column to its right. Each column may contain either a 0 or a 1. Thus, binary 1010 represents 1 times 8, plus 0 times 4, plus 1 times 2, plus 0 times 1, or decimal 10. Octal and hexadecimal are discussed more in the section "Octal and Hexadecimal Numbers" in Chapter 5, *Expressions*.

Programs are written in programming languages. Hundreds, if not thousands, of programming languages exist. One of the most popular is the C programming language. The C language had a very strong influence on the design of the *awk* language.

There have been several versions of C. The first is often referred to as "K&R" C, after the initials of Brian Kernighan and Dennis Ritchie, the authors of the first book on C. (Dennis Ritchie created the language, and Brian Kernighan was one of the creators of *awk*.)

In the mid-1980s, an effort began to produce an international standard for C. This work culminated in 1989, with the production of the ANSI standard for C. This standard became an ISO standard in 1990. Where it makes sense, POSIX *awk* is compatible with 1990 ISO C.

In 1999, a revised ISO C standard was approved and released. Future versions of *gawk* will be as compatible as possible with this standard.

Floating-Point Number Caveats

As mentioned earlier, floating-point numbers represent what are called "real" numbers, i.e., those that have a fractional part. *awk* uses double-precision floating-point numbers to represent all numeric values. This section describes some of the issues involved in using floating-point numbers.

There is a very nice paper on floating-point arithmetic by David Goldberg, "What Every Computer Scientist Should Know About Floating-point Arithmetic," *ACM Computing Surveys* **23**, 1 (1991-03), 5-48.* This is worth reading if you are interested in the details, but it does require a background in computer science.

Internally, *awk* keeps both the numeric value (double-precision floating-point) and the string value for a variable. Separately, *awk* keeps track of what type the variable has (see the section "Variable Typing and Comparison Expressions" in Chapter 5), which plays a role in how variables are used in comparisons.

It is important to note that the string value for a number may not reflect the full value (all the digits) that the numeric value actually contains. The following program (*values.awk*) illustrates this:

```
{
    $1 = $2 + $3
    # see it for what it is
    printf("$1 = %.12g\n", $1)
    # use CONVFMT
    a = "<" $1 ">"
    print "a =", a
    # use OFMT
    print "$1 =", $1
}
```

This program shows the full value of the sum of `$2` and `$3` using `printf`, and then prints the string values obtained from both automatic conversion (via `CONVFMT`) and from printing (via `OFMT`).

---

* *http://www.validgh.com/goldberg/paper.ps.*

Here is what happens when the program is run:

```
$ echo 2 3.654321 1.2345678 | awk -f values.awk
$1 = 4.8888888
a = <4.88889>
$1 = 4.88889
```

This makes it clear that the full numeric value is different from what the default string representations show.

CONVFMT's default value is `"%.6g"`, which yields a value with at least six significant digits. For some applications, you might want to change it to specify more precision. On most modern machines, most of the time, 17 digits is enough to capture a floating-point number's value exactly.*

Unlike numbers in the abstract sense (such as what you studied in high school or college math), numbers stored in computers are limited in certain ways. They cannot represent an infinite number of digits, nor can they always represent things exactly. In particular, floating-point numbers cannot always represent values exactly. Here is an example:

```
$ awk '{ printf("%010d\n", $1 * 100) }'
515.79
0000051579
515.80
0000051579
515.81
0000051580
515.82
0000051582
Ctrl-d
```

This shows that some values can be represented exactly, whereas others are only approximated. This is not a "bug" in *awk*, but simply an artifact of how computers represent numbers.

Another peculiarity of floating-point numbers on modern systems is that they often have more than one representation for the number zero! In particular, it is possible to represent "minus zero" as well as regular, or "positive" zero.

This example shows that negative and positive zero are distinct values when stored internally, but that they are in fact equal to each other, as well as to "regular" zero:

* Pathological cases can require up to 752 digits (!), but we doubt that you need to worry about this.

```
$ gawk 'BEGIN { mz = -0 ; pz = 0
> printf "-0 = %g, +0 = %g, (-0 == +0) -> %d\n", mz, pz, mz == pz
> printf "mz == 0 -> %d, pz == 0 -> %d\n", mz == 0, pz == 0
> }'
-0 = -0, +0 = 0, (-0 == +0) -> 1
mz == 0 -> 1, pz == 0 -> 1
```

It helps to keep this in mind should you process numeric data that contains negative zero values; the fact that the zero is negative is noted and can affect comparisons.