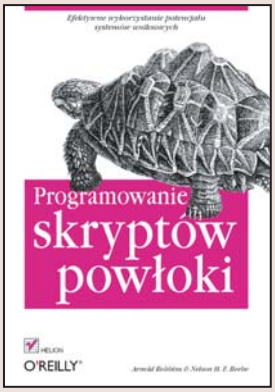
Programowanie skryptów powłoki



Autorzy: Arnold Robbins, Nelson H. F. Beebe
Tłumaczenie: Przemysław Szeremiota
ISBN: 83-246-0131-7
Tytuł oryginału: Classic Shell Scripting
Format: B5, stron: 560

Efektywne wykorzystanie potencjału systemów uniksowych

- Automatyzacja zadań
- Przeszukiwanie plików i katalogów
- Przenoszenie skryptów pomiędzy systemami

W dobie graficznych narzędzi programistycznych często pomijamy tradycyjne metody rozwiązywania przeróżnych zadań związanych z działaniem systemu operacyjnego. Skrypty powłoki, niegdyś podstawowe narzędzie administratorów i programistów systemów uniksowych, dziś są zdecydowanie mniej popularne. Skrypty powłoki są przydatne zarówno administratorom systemu, jak i szeregowym użytkownikom, ponieważ są jednym z najlepszych sposobów na zaprzęgnięcie do pracy setek narzędzi, w jakie wyposażony jest Unix. Z narzędzi tych w języku programowania powłoki łatwo stworzyć rozwiązanie niemal dowolnego zadania związanego z przetwarzaniem danych.

Książka „Programowanie skryptów powłoki" to kompendium wiedzy dotyczącej tej nieco już zapomnianej techniki. Przedstawia nie tylko język programowania powłoki,ale także narzędzia systemu Unix. Dostarcza informacji o tym, do jakich zadań się nadają, jak je wywoływać i jak łączyć je z innymi programami, konstruując z nich mechanizm przetwarzania danych. W książce opisano nie tylko sposoby pisania użytecznych skryptów powłoki, ale również metody dostosowywania powłoki do własnych potrzeb oraz przenoszenia skryptów pomiędzy różnymi wariantami Uniksa i różnymi implementacjami powłoki.

- Podstawowe elementy skryptów powłoki
- Wyszukiwanie i zastępowanie fragmentów tekstów
- Stosowanie wyrażeń regularnych
- Korzystanie z potoków
- Instrukcje warunkowe
- Definiowanie i stosowanie zmiennych
- Przetwarzanie plików
- Standardowe wejście i wyjście
- Korzystanie z możliwości awk
- Przenoszenie skryptów pomiędzy różnymi powłokami
- Bezpieczeństwo skryptów powłoki

Książka „Programowanie skryptów powłoki" zawiera wszystkie informacje niezbędne do mistrzowskiego opanowania narzędzi oferowanych przez systemy uniksowe.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

ROZDZIAŁ 9.

Nieuzbrojony a niebezpieczny — awk

Język programowania awk powstał jako narzędzie upraszczania wielu powszechnie wykonywanych zadań programistycznych związanych z przetwarzaniem tekstu. W niniejszym rozdziale omówimy tę jego część, którą wykorzystuje się najczęściej w skryptach powłoki, również tych prezentowanych w tej książce.

Pełnego omówienia języka programowania awk należy szukać w jednej z poświęconych mu w całości książek, wymienionych w bibliografii. Jeśli zaś w danym systemie zainstalowana jest wersja GNU awk (gawk), warto też zajrzeć do dołączonej doń dokumentacji, dostępnej za pośrednictwem systemu info[1].

Wszystkie systemy z rodziny Unix są wyposażone w przynajmniej jedną implementację awk. Po znaczącym rozszerzeniu języka, mającym miejsce w połowie lat osiemdziesiątych, doszło do pewnego rozłamu: niektórzy producenci systemów zachowali implementację pierwotną, instalując ją jako awk albo oawk, a nową wersję udostępnili pod nazwą nawk — w systemach AIX (IBM) i Solaris (Sun) ta praktyka została podtrzymana po dziś dzień. Jednak w większości innych wydań instalowane są jedynie nowsze wersje awk. W systemie Solaris wersja zgodna z POSIX instalowana jest jako /usr/xpg4/bin/awk. W niniejszej książce będziemy omawiać wersję rozszerzoną i będzie ona tu występować jako awk — w konkretnym systemie będzie to zaś albo nawk, albo gawk, albo np. mawk.

Musimy się tu jako autorzy przyznać do wielkiej zażyłości z awk. Całymi latami implementowaliśmy go, opiekowaliśmy się implementacjami, przenosiliśmy je na inne platformy, pisywaliśmy o nim i wreszcie wykorzystywaliśmy we własnych projektach. Większość programów języka awk to programy króciutkie, ale zdarzyło się nam popełnić i takie, które miały po parę tysięcy wierszy. Prostota i siła awk sprawiają, że jest on obowiązkowym elementem przybornika programisty uniksowego. Rzadko zdarzają się też takie zadania z dziedziny przetwarzania tekstów, w których potrzebna byłaby funkcja czy cecha nieobecna w języku (albo nie dałoby się jej prosto zaimplementować samodzielnie). Parokrotnie stanęliśmy w obliczu zadania przepisania programu w języku awk na jeden z konwencjonalnych języków kompilowanych, jak C czy C++ — programy te były zawsze znacznie dłuższe, znacznie trudniejsze do analizy i diagnostyki, tyle że działały nieco szybciej.

[1] Program info to czytnik dokumentacji GNU, stanowiący część pakietu *texinfo*, dostępnego pod adresem *ftp://ftp.gnu.org/gnu/texinfo/*. Do przeglądania tejże dokumentacji można też wykorzystać edytor tekstów emacs — wystarczy w sesji programu emacs nacisnąć klawisze *Ctrl+H* — *przyp. autora*.

W przeciwieństwie do większości innych języków skryptowych awk doczekał się wielu rozmaitych implementacji, co należy uznać za zaletę, bo programiści otrzymują zawsze taki sam, wspólny rdzeń języka i równocześnie dysponują swobodą wyboru takiej implementacji, która najlepiej odpowiada ich potrzebom. Do tego awk jest częścią standardu POSIX i doczekał się przeniesienia implementacji również na systemy operacyjne spoza rodziny Unix.

Jeśli w danym systemie zainstalowana jest wersja awk pochodząca sprzed ustalenia standardu, warto zaopatrzyć się w jedną z darmowych implementacji wymienionych w tabeli 9.1. Wszystkie one są przenośne i wyjątkowo łatwe w instalacji. Implementacja gawk służyła za poligon testowy dla szeregu ciekawych nowych funkcji wbudowanych i cech języka, w tym implementacji sieciowych operacji wejścia-wyjścia, jak również mechanizmów profilowania, umiędzynaradawiania i kontroli przenośności.

Tabela 9.1. Dostępne nieodpłatnie wersje awk

Program	Położenie
Wersja awk z laboratoriów Bella	*http://cm.bell-labs.com/who/bwk/awk.tar.gz*
gawk	*ftp://ftp.gnu.org/gnu/gawk/*
mawk	*ftp://ftp.whidbey.net/pub/brennan/mawk-1.3.3.tar.gz*
awka (translator awk – C)	*http://awka.sourceforge.net/*

9.1. Wywołanie awk

W wywołaniu interpretera awk można definiować zmienne, określać kod programu i wskazywać nazwy plików wejściowych:

```
awk [ -F sep ] [ -v zmienna=wartość ... ] 'program' [ -- ] \
    [ zmienna=wartość ... ] [ plik ... ]
awk [ -F sep ] [ -v zmienna=wartość ... ] -f plik-programu [ -- ] \
    [ zmienna=wartość ... ] [ plik ... ]
```

Krótkie programy są najczęściej przekazywane do interpretera z poziomu wiersza wywołania; dłuższe zapisuje się w pliku i wskazuje ów plik za opcją `-f`. Opcja ta może zostać powtórzona — w takim przypadku interpreter złoży program, konkatenując ze sobą zawartość poszczególnych plików. Można dzięki temu konstruować i wykorzystywać biblioteki kodu w języku awk. Do włączania bibliotek można też wykorzystać program igawk, wchodzący w skład dystrybucji gawk. Wszelkie opcje wywołania awk muszą znaleźć się przed plikami i parami *zmienna=wartość*.

Jeśli wywołanie nie określa plików wejściowych, interpreter będzie wczytywał dane ze standardowego wejścia.

Opcja w postaci `--` nie jest obowiązkowym elementem wywołania; jeśli występuje, oznacza koniec opcji wywołania interpretera awk. Wszelkie opcje umieszczone za tym symbolem stanowią opcje programu.

Opcja `-F` pozwala na zmianę domyślnego separatora pól. Przyjęło się, że jeśli już występuje, to jako pierwsza z opcji wywołania interpretera awk. Jej argumentem jest wartość *sep*, będąca wyrażeniem regularnym i umieszczona bezpośrednio za `-F`, albo występująca jako następny argument wywołania. Separator pól można też ustawić przypisaniem do wbudowanej zmiennej FS (zobacz tabelę 9.3, w której wymienione zostały zmienne skalarne awk):

```
awk -F '\t' '{ ... }' pliki FS="[\f\v]" pliki
```

W powyższym wywołaniu przy przetwarzaniu pierwszego zestawu *plików* będzie obowiązywał separator pól ustalony opcją `-F`. Separator ustawiony przypisaniem do zmiennej `FS` będzie zaś obowiązywał przy przetwarzaniu drugiego zestawu *plików*.

Przypisania do zmiennych podawane z opcją `-v` muszą poprzedzać wszelki kod programu przekazywany jawnie w wierszu wywołania; przypisania te są realizowane jeszcze przed uruchomieniem programu i przed przystąpieniem do przetwarzania plików wejściowych. Kiedy opcja `-v` występuje za kodem programu, jest interpretowana jako nazwa pliku (z oczywistych względów najprawdopodobniej nieistniejącego).

Przypisania określone w innych miejscach wiersza wywołania są realizowane w miarę przetwarzania argumentów; mogą być przetykane nazwami plików, jak tutaj:

```
awk '{...}' zm=1 *.tex zm=2 *.tex
```

W powyższym wywołaniu pliki **.tex* zostaną przetworzone dwukrotnie — raz z ustawieniem `zm` na jeden i drugi raz, po ustawieniu `zm` na dwa.

Wartości inicjalizujące zmienne ciągami znaków nie muszą być ujmowane w znaki cudzysłowu, chyba że jest to wymagane ze względu na mechanizmy powłoki, na przykład do zachowania znaków specjalnych czy odstępów.

Specjalna nazwa pliku wejściowego w postaci myślnika (-) reprezentuje standardowe wejście. Większość współczesnych implementacji `awk` rozpoznaje nazwę pliku specjalnego */dev/stdin*; reprezentuje ona standardowe wejście nawet w tych systemach, które nie rozpoznają tej nazwy jako nazwy istniejącego pliku. Podobnie jest z nazwami */dev/stderr* i */dev/stdout*, które w wywołaniu `awk` reprezentują standardowe wyjście i standardowe wyjście diagnostyczne.

9.2. Model programistyczny awk

Interpreter `awk` postrzega strumień danych wejściowych jako kolekcję *rekordów*, z których każdy da się podzielić na *pola*. Zwykle rekord pokrywa się z pojedynczym wierszem, a pole to jedno słowo takiego wiersza (ewentualnie dowolny jedno- lub wieloznakowy ciąg znaków nie będących znakami odstępu). Jednak sposób dzielenia rekordów i pól pozostaje całkowicie pod kontrolą programisty, a ich definicje mogą być zmieniane nawet w trakcie przetwarzania.

Program języka `awk` składa się z par wzorzec-akcja i może być ewentualnie uzupełniony definicjami funkcji implementujących szczegółowe operacje w ramach akcji. Za każdym razem, kiedy wzorzec uda się dopasować do wejścia, wykonywana jest skojarzona z wzorcem akcja. Przy tym każdy rekord wejściowy jest przetwarzany z uwzględnieniem wszystkich wzorców.

W parze wzorzec-akcja można pominąć zarówno część definiującą wzorzec dopasowania, jak i część definiującą akcję. W tym pierwszym przypadku akcja będzie stosowana do każdego rekordu wejściowego; w obliczu braku akcji dopasowanie wzorca do rekordu zostanie skwitowane wykonaniem akcji domyślnej, która polega na wypisaniu dopasowanego rekordu na standardowym wyjściu. Oto typowy układ programu `awk`:

```
wzorzec { akcja }       uruchomienie akcji po dopasowaniu wzorca
wzorzec                 wypisanie rekordu dopasowanego do wzorca
        { akcja }       uruchomienie akcji dla każdego rekordu
```

Wejście automatycznie przełącza się pomiędzy wskazanymi plikami wejściowymi. Otwieraniem plików wejściowych, odczytem danych i zamykaniem plików zajmuje się sam interpreter `awk`, więc programista może skupić się na właściwym problemie — przetwarzaniu rekordów. Tym zajmiemy się obszernie w podrozdziale 9.5.

Wzorce są najczęściej wyrażeniami liczbowymi albo ciągami, ale awk pozwala też na stosowanie dwóch wzorców specjalnych oznaczanych słowami kluczowymi BEGIN i END.

Akcja skojarzona z wzorcem BEGIN jest wykonywana jednokrotnie, tuż *przed* przystąpieniem do przetwarzania plików wejściowych i realizacją wszelkich zwykłych (bez opcji -v) przypisań zadanych w wywołaniu, ale już *po* zrealizowaniu przypisań przekazanych z opcją -v. Zwykle w bloku kodu tej akcji realizuje się specjalne procedury inicjalizacji wymagane przez właściwy program.

Akcja skojarzona z END również jest wykonywana tylko jednokrotnie, już *po* przetworzeniu kompletu danych wejściowych. Zwykle w jej obrębie realizuje się wypisywanie zestawień i podsumowań wyników przetwarzania, ewentualnie czynności porządkowe.

Wzorce BEGIN i END mogą występować w dowolnej kolejności i gdziekolwiek w programie. Utarło się jednak, aby wzorzec BEGIN stanowił pierwszy wzorzec programu, a wzorzec END kończył program.

Jeśli w programie występuje większa liczba wzorców BEGIN i END, są one przetwarzane w kolejności, w jakiej występują w programie. Dzięki temu można uzupełniać czynnościami wstępnymi i porządkowymi również kod biblioteczny, włączany do programu dodatkowymi opcjami -f.

9.3. Elementy programu

Jak większość skryptowych języków programowania, awk manipuluje przede wszystkim liczbami i ciągami znaków. W obrębie programu programista może przechowywać dane w zmiennych skalarnych i tablicowych, ma też do dyspozycji wyrażenia liczbowe i wyrażenia ciągów znaków oraz przypisania, instrukcje warunkowe, wywołania funkcji, instrukcje wejścia-wyjścia, instrukcje pętli i komentarze. Wiele cech instrukcji i wyrażeń awk upodabnia te wyrażenia i instrukcje do ich odpowiedników w języku C.

9.3.1. Komentarze i odstępy

Komentarze w awk oznacza się znakiem kratki (#). Komentarz rozciąga się od tego znaku do końca wiersza programu — tak, jak w skryptach powłoki. Wiersze puste są odpowiednikami pustych komentarzy.

Tam, gdzie składnia języka przewiduje stosowanie odstępów, można umieścić dowolną liczbę znaków odstępów. Można dzięki temu pustymi wierszami i wcięciami uwypuklać strukturę programu gwoli większej czytelności. Zwykle jednak nie można rozbijać pojedynczej instrukcji na wiele wierszy — chyba że znaki nowego wiersza będą w nich bezpośrednio poprzedzane znakami lewego ukośnika.

9.3.2. Ciągi i wyrażenia ciągów

Ciągi znaków, czyli stałe łańcuchowe, są w awk ograniczane znakami podwójnego cudzysłowu: "To jest ciag znakow". Ciągi znaków mogą zawierać dowolne 8-bitowe znaki, z *wyjątkiem* sterującego znaku pustego (znaku o wartości 0), który w języku C (a w nim pisany jest interpreter awk) służy za znak końca ciągu. W implementacji GNU gawk ograniczenie to zostało zniesione; dzięki temu gawk może bezpiecznie przetwarzać dowolne pliki binarne.

Ciąg w awk może zawierać zero albo więcej znaków; jego długość nie jest ograniczona niczym poza ilością dostępnej pamięci. Przypisanie wyrażenia ciągu znaków do zmiennej automatycznie tworzy ciąg znaków i przydziela mu pamięć. Pamięć zajmowana przez poprzednią wartość zmiennej jest zaś automatycznie zwalniana.

Do reprezentowania znaków niedrukowalnych służą sekwencje sterujące sygnalizowane znakiem lewego ukośnika, podobnie, jak to miało miejsce w argumentach polecenia echo (zobacz punkt 2.5.3). Ciąg `"A\tZ"` zawiera więc znak A, znak tabulacji i znak Z, a ciągi `"\001"` i `"\x01"` zawierają znak *Ctrl+A*.

W echo brak obsługi sekwencji sterujących z wartościami szesnastkowymi. W awk są one rozpoznawane, przynajmniej w implementacjach bazujących na wprowadzonym w 1989 roku standardzie ISO C. W przeciwieństwie do ósemkowych sekwencji sterujących składających się z najwyżej trzech znaków, sekwencje szesnastkowe obejmują wszystkie stanowiące zwarty podciąg cyfry szesnastkowe. Tak jest w gawk i nawk. Z reguły tej wyłamuje się mawk: tu sekwencja szesnastkowa jest ograniczona do najwyżej dwóch cyfr, co redukuje ciąg `"\x404142"` do postaci `"@4142"`. Gdyby nie ograniczenie długości sekwencji szesnastkowej do dwóch cyfr, ciąg ten zostałby zinterpretowany jako `"@AB"`. Implementacje awk zgodne ze standardem POSIX w ogóle nie obsługują szesnastkowych sekwencji sterujących.

Interpreter awk wspomaga operacje na ciągach szeregiem wygodnych i użytecznych funkcji wbudowanych; omawiamy je w podrozdziale 9.9. Na razie wspomnimy choćby o funkcji obliczającej długość ciągu: wywołanie `length(`*ciąg*`)` zwraca liczbę znaków w ciągu *ciąg*.

Ciągi można porównywać tradycyjnym zestawem operatorów relacji: == (równe), != (różne), < (mniejszy niż), <= (mniejszy lub równy), > (większy), >= (większy lub równy). Operatory relacji zwracają zero (wartość logiczna „prawda”), kiedy relacja jest spełniona dla zadanych operandów, albo 1 (logiczny „fałsz”), kiedy relacja pozostaje niespełniona. Kiedy porównuje się ciągi o różnej długości i jeden z tych ciągów stanowi podciąg drugiego, krótszy z nich obu jest uznawany za „mniejszy” niż dłuższy. Stąd `"A" < "AA"` daje spełnioną relację i wartość logiczną „prawda”.

W większości języków programowania obsługujących typy łańcuchowe stosuje się specjalny operator konkatenacji ciągów. W awk nie ma takiego specjalnego operatora — konkatenacji poddawane są automatycznie wszelkie sąsiadujące ze sobą ciągi znaków. Każde z poniższych przypisań ustawia więc zmienną s na ten sam czteroznakowy ciąg:

```
s = "ABCD"
s = "AB" "CD"
s = "A" "BC" "D"
s = "A" "B" "C" "D"
```

Automatyczna konkatenacja obejmuje nie tylko stałe (literały) łańcuchowe. Gdyby poprzednie przypisania uzupełnić poniższym:

```
t = s s s
```

zmienna t otrzymałaby wartość `"ABCDABCDABCD"`.

Konwersja liczby na ciąg odbywa się niejawnie, przez konkatenację literału liczbowego z ciągiem pustym: `n = 123` uzupełnione przypisaniem `s = "" n`, powoduje przypisanie do s ciągu `"123"`. Trzeba tu uważać na liczby, których nie da się dokładnie reprezentować — zajmiemy się tym w punkcie 9.9.8, przy okazji omawiania formatowanej konwersji liczby na ciąg.

Znaczna część potęgi awk wynika z implementowanej w tym języku obsługi wyrażeń regularnych. Ich wykorzystanie jest ułatwione przez dwa operatory: ~ (dopasowanie) i !~ (brak dopasowania). Wyrażenie "ABC" ~ "^[A-Z]+$" daje wartość „prawda", bo ciąg występujący w roli lewego operandu zawiera wyłącznie znaki wielkich liter, a wyrażenie regularne zadane prawym operandem dopasowuje właśnie ciągi wielkich liter ASCII (o dowolnej długości). W awk można korzystać z rozszerzonych wyrażeń regularnych ERE (*Extended Regular Expressions*), opisywanych w punkcie 3.2.3.

Wyrażenia regularne mogą być ograniczane albo parą znaków cudzysłowu, albo ukośników: "ABC" ~ /^[A-Z]+$/. Wybór jednej z tych konwencji jest kwestią gustu i wygody programisty, choć w sumie preferowana jest forma z ukośnikami, bo uwypukla ona fakt, że ujęty pomiędzy nimi ciąg jest wyrażeniem regularnym, a nie zwykłym literałem łańcuchowym. Są jednak przypadki, w których znak ukośnika ograniczający wyrażenie regularne może zostać pomylony z operatorem dzielenia — tam należałoby stosować znaki cudzysłowu.

Znak cudzysłowu w obrębie ciągu ograniczonego takimi znakami, jeśli ma zostać potraktowany dosłownie, powinien zostać poprzedzony znakiem ukośnika lewego ("...\"..."), to samo tyczy się znaków ukośnika w wyrażeniach ograniczanych parą znaków ukośnika (/...\/.../). Oczywiście również znak lewego ukośnika, kiedy ma być traktowany dosłownie, musi zostać poprzedzony takim znakiem, a w wyrażeniach ograniczanych znakami cudzysłowu nawet kilkoma: "\\\\Tex" i /\\Tex/ reprezentują to samo wyrażenie regularne dopasowujące ciąg \Tex.

9.3.3. Liczby i wyrażenia liczbowe

Wszelkie liczby w programie awk są reprezentowane jako wartości zmiennoprzecinkowe podwójnej precyzji, o których więcej można się dowiedzieć z sąsiedniej ramki. Programista awk nie musi być bynajmniej ekspertem w dziedzinie arytmetyki zmiennoprzecinkowej, ale ważne, aby zdawał sobie sprawę z charakterystycznych dla niej (i ogólnie dla arytmetyki realizowanej przez komputery) ograniczeń i aby nie oczekiwał od komputera nieosiągalnej dla niego dokładności, a przy okazji uniknął kilku pułapek.

Liczby zmiennoprzecinkowe mogą zawierać wykładnik za literą e (albo E) i część całkowitą, z ewentualnym znakiem. Na przykład wartość ułamka 1/32 można reprezentować wartościami zmiennoprzecinkowymi 0.03125, 3.125e-2, 3125e-5 albo 0.003125E1. Ponieważ wszelka arytmetyka w awk to arytmetyka zmiennoprzecinkowa, wyrażenie 1/32 można zapisać w ten sposób bez ryzyka, że przyjmie wartość zerową, jak to bywa w językach programowania bazujących na arytmetyce liczb całkowitych.

Nie istnieje funkcja jawnej konwersji ciągu na wartość liczbową, ale w awk nie jest to problemem: wystarczy do ciągu znaków reprezentującego liczbę dodać zero. Na przykład przypisanie s = "123", uzupełnione przypisaniem n = 0 + s, zaowocuje przypisaniem do zmiennej n wartości liczbowej 123.

Ciągi są konwertowane na liczby, o ile tylko zawartość ciągu, albo jego część, choćby przypomina liczbę: "+123ABC" da się konwertować na wartość 123, a ciągi "ABC", "ABC123" i "" mają wartość liczbową 0.

Więcej o arytmetyce zmiennoprzecinkowej

Współcześnie praktycznie wszystkie platformy sprzętowe są zgodne ze standardem binarnej arytmetyki zmiennoprzecinkowej według ustalonego w 1985 roku standardu IEEE 754 (*Standard for Binary Floating-Point Arithmetic*). Standard ten definiuje 32-bitowy format pojedynczej precyzji, 64-bitowy format podwójnej precyzji i opcjonalny format precyzji rozszerzonej, implementowany zwykle na 80 lub 128 bitach. Implementacje `awk` korzystają zazwyczaj z 64-bitowego formatu (odpowiadającego typowi `double` z języka C), choć ze względu na przenośność specyfikacja języka `awk` nie narzuca żadnych szczegółów w tym zakresie. Specyfikacja standardu POSIX mówi zaś jedynie, że implementacja arytmetyki powinna być zgodna ze standardem ISO C, który nie narzuca żadnej architektury zmiennoprzecinkowej.

Wartości formatu podwójnej precyzji wg IEEE 754 posiadają bit znaku, 11-bitowy wykładnik i 53-bitową mantysę, której najstarszy bit nie jest zachowywany. Pozwala to na reprezentowanie szesnastocyfrowych liczb dziesiętnych. Maksimum skończonej wielkości dającej się reprezentować w tym formacie przypada na 10^{+308}, a najmniejsza znormalizowana niezerowa wartość to 10^{-308}. Większość implementacji IEEE 754 obsługuje dodatkowo wartości nieznormalizowane, rozszerzające zakres reprezentacji do 10^{-324}, ale kosztem precyzji — ów *stopniowy niedomiar* do zera ma zresztą kilka własności, które choć pożądane w oprogramowaniu stricte obliczeniowym, w innych zastosowaniach są nieistotne.

Z racji jawnego reprezentowania znaku liczby osobnym bitem arytmetyka zmiennoprzecinkowa IEEE 754 rozróżnia dwie wartości zerowe: dodatnią i ujemną. W większości języków programowania jest to ignorowane; `awk` nie jest wyjątkiem: niektóre implementacje wypisują ujemne zero bez znaku minusa.

Arytmetyka IEEE 754 uwzględnia również dwie specjalne wartości reprezentujące nieskończoność i wartość nieliczbową (NaN, od ang. *not a number*). Obie mogą występować ze znakiem, choć znak wartości nieliczbowej jest ignorowany. Wartości te są wykorzystywane do wykonywania nieprzerwanych ciągów obliczeń na wysokowydajnych komputerach przy zachowaniu możliwości rejestrowania stanów wyjątkowych. Kiedy liczba jest zbyt duża, aby dało się ją skutecznie reprezentować, wynikiem jest nieskończoność, a dodatkowo procesor ustawia znacznik *przepełnienia*. W przypadku wartości niezdefiniowanych, jak nieskończoność-nieskończoność albo `0/0`, wynikiem jest wartość nieliczbowa.

Nieskończoność i wartość nieliczbowa podlegają propagacji w obliczeniach: nieskończoność + nieskończoność oraz nieskończoność * nieskończoność dają nieskończoność, a jakakolwiek operacja arytmetyczna angażująca wartość nieliczbową daje w wyniku wartość nieliczbową.

Porównanie dwóch nieskończoności o tym samym znaku daje równość. Porównanie dwóch wartości nieliczbowych daje różność; dla `x` będącego wartością nieliczbową spełniona jest więc relacja `(x != x)`.

Język `awk` powstał przed upowszechnieniem się standardu IEEE 754, przez co język nie obsługuje w pełni nieskończoności i wartości nieliczbowych. W szczególności w bieżących implementacjach `awk` próba dzielenia przez zero prowokuje wyjątek — mimo że wedle reguł arytmetyki IEEE 754 nie ma takiej konieczności.

Ograniczona precyzja wartości zmiennoprzecinkowych oznacza niemożność dokładnego reprezentowania niektórych liczb: liczy się przy tym kolejność wartościowania (arytmetyka zmiennoprzecinkowa nie podlega łączności), a obliczone wyniki są zwykle zaokrąglane do najbliższej wartości dającej się dokładnie reprezentować.

Ograniczony zakres reprezentacji wartości zmiennoprzecinkowych oznacza z kolei, że wartości bardzo wielkie i bardzo małe również nie dają się dokładnie reprezentować. We współczesnych systemach wartości te są konwertowane na (odpowiednio) nieskończoność i zero.

Choć obliczenia realizowane z poziomu programu `awk` wykonywane są wedle reguł arytmetyki zmiennoprzecinkowej, nie znaczy to, że nie można posługiwać się wartościami całkowitoliczbowymi — będą one reprezentowane dokładnie, o ile tylko będą utrzymywane w odpowiednim zakresie. Arytmetyka zmiennoprzecinkowa IEEE 754 przy 53-bitowej mantysie pozwala na reprezentowanie wartości całkowitych z zakresu od 0 do 2^{53}, czyli 9 007 199 254 740 992. Liczba ta jest w zastosowaniach związanych z przetwarzaniem tekstu aż nadto wystarczająca — ryzyko wyczerpania zakresu w wyniku np. zliczania jest bardzo nikłe.

Zbiór operatorów arytmetycznych języka `awk` odzwierciedla podobne zbiory znane z innych języków programowania. Komplet operatorów wymienia tabela 9.2.

Tabela 9.2. Operatory arytmetyczne języka awk (według priorytetu)

Operator	Działanie
`++ --`	Inkrementacja i dekrementacja (w wersji przed- albo przyrostkowej).
`^ **`	Potęgowanie (łączność prawostronna).
`! + -`	Negacja, jednoargumentowe operatory znaku.
`* / %`	Mnożenie, dzielenie, reszta z dzielenia.
`+ -`	Dodawanie, odejmowanie.
`< <= == != > >=`	Operatory relacji.
`&&`	Logiczny iloczyn (AND — ze skróconym wartościowaniem).
`||`	Logiczna suma (OR — również ze skróconym wartościowaniem).
`? :`	Operator wykonania warunkowego.
`= += -= *= /= %= ^= **=`	Operatory przypisania (prawostronnie łączne).

Jak w większości języków programowania, kolejność stosowania operatorów można modyfikować nawiasami. Mało kto rozeznaje się dokładnie we wzajemnym pierwszeństwie poszczególnych operatorów; dotyczy to zwłaszcza parających się programowaniem w różnych językach. Rada jest jedna: w razie wątpliwości, stosować nawiasy!

Operatory inkrementacji i dekrementacji działają identycznie, jak w powłoce (zostało to omówione w punkcie 6.1.3). Wyrażenia `++n` i `n++`, jeśli występują w odosobnieniu, są sobie co do ostatecznego efektu równoważne. Jednak z racji *efektu ubocznego* — bo obok zwrócenia wartości zmiennej rzeczone operatory modyfikują tę wartość — wielokrotne wystąpienia operatorów inkrementacji i dekrementacji w obrębie jednego wyrażenia mogą zaistnieć niejednoznaczności wynikające z kolejności wartościowania. Wynik wyrażenia `n++ ++n` jest więc zależny od implementacji. Mimo tego rodzaju niejednoznaczności operatory inkrementacji i dekrementacji są powszechnie wykorzystywane nie tylko w `awk`, ale we wszystkich obsługujących je językach programowania.

Operatory potęgowania podnoszą lewy operand do potęgi określonej prawym operandem. Stąd zarówno `n^3`, jak i `n**3` oznacza podniesienie wartości `n` do sześcianu. Oba operatory są sobie równoważne, choć mają różne korzenie — wywodzą się z różnych języków programowania. Programiści języka C powinni pamiętać, że operator `^`, mimo swojego podobieństwa do podobnie zapisywanego operatora z języka C, różni się od niego działaniem.

Potęgowanie i operacje przypisania to jedyne operatory awk cechujące się *łącznością prawostronną*. Łączność taka oznacza, że a^b^c^d to tyle, co a^(b^(c^d)), podczas gdy a/b/c/d oznacza tyle, co ((a/b)/c)/d. Reguły łączności są zbieżne z tymi stosowanymi w większości pozostałych języków programowania, stanowią też konwencję przyjętą w matematyce.

W pierwotnej specyfikacji awk wynik działania operatora reszty z dzielenia w przypadku, kiedy jeden z operandów był ujemny, był zależny od implementacji. POSIX wymaga, aby implementacja awk zachowywała się zgodnie ze standardem ISO C w zakresie ustalonym dla funkcji fmod(). Specyfikacja ta zakłada, że jeśli wartość x % y daje się w ogóle reprezentować, to posiada znak wartości x i wartość bezwzględną nie większą od y. Wszystkie testowane przez nas implementacje awk zachowywały się zgodnie z wymogami POSIX.

Tak, jak w powłoce, operatory logiczne && i || podlegają skróconemu wartościowaniu — wartość logiczna prawego operandu jest obliczana wyłącznie w przypadku, kiedy nie można ustalić wartości operacji na podstawie samego lewego operandu.

Operator z przedostatniego wiersza tabeli 9.2 to trójargumentowy operator warunkowy, który również stosuje regułę skróconego wartościowania. Otóż jeśli pierwszy z operandów ma wartość logiczną „prawda", to wynikiem operatora jest drugi operand; w innym przypadku operator zwraca wartość trzeciego operandu. Tak czy inaczej, z dwóch (drugiego i trzeciego) operandów wartościowany jest zawsze tylko jeden. Dzięki temu można w awk w zwarty sposób zapisać następujące przypisanie: a = (u > w) ? x^3 : y^7. W innych językach programowania wymagałoby to skonstruowania następującej instrukcji warunkowej:

```
if (u > w) then
    a = x^3
else
    a = y^7
endif
```

Ciekawe są operatory przypisania, a to z dwóch powodów. Po pierwsze, ich wersje złożone, jak /=, wykorzystują lewy operand w roli brakującego pierwszego operandu po lewej stronie przypisania; zapis n /= 3 to w istocie skrócone przypisanie n = n / 3. Po drugie, wartość zwracana przez operator przypisania może być wykorzystywana jako część wyrażenia; na przykład wyrażenie a = b = c = 123 powoduje najpierw przypisanie wartości 123 do zmiennej c, potem przypisanie wartości c (123) do zmiennej b i wreszcie przypisanie bieżącej wartości b (również 123) do a (wszystko dzięki prawostronnej łączności operatora przypisania). W efekcie wszystkie trzy zmienne (a, b i c) otrzymują wartość 123. Podobnie należy interpretować wyrażenie x = (y = 123) + (z = 321), ustawiające zmienne x, y i z na (odpowiednio) 444, 123 i 321.

Operatory ** i **= nie są ujęte specyfikacją POSIX i jako takie nie są rozpoznawane przez mawk. Dlatego też należałoby unikać ich stosowania, zastępując je operatorami ^ i ^=.

Nie wolno zapominać o zasadniczej różnicy pomiędzy operatorem przypisania (=) a podobnie wyglądającym (i często omyłkowo zapisywanym) operatorem równości (==). Ponieważ przypisania są jak najbardziej poprawnymi wyrażeniami, wyrażenie (r = s) ? t : u jest co prawda składniowo poprawne, ale zapewne zostało tak zapisane w wyniku omyłki. Realizuje ono bowiem przypisanie s do r, a jeśli wartość r będzie niezerowa, całość wyrażenia przyjmie wartość t; w innym przypadku całość przyjmie wartość u. Ostrzeżenie to dotyczy również języków C, C++ i Java, w których równie łatwo o zgubną w skutkach pomyłkę w zapisie operatorów = i ==.

Część całkowitą argumentu można wyłuskać z wartości liczbowej za pośrednictwem wbudowanej funkcji `int()`: wywołanie `int(-3.14159)` zwraca wartość `-3`.

Język `awk` udostępnia też programistom zestaw podstawowych funkcji matematycznych, znanych z kalkulatorów i innych języków programowania; mowa o `sqrt()`, `sin()`, `cos()`, `log()`, `exp()` i tym podobnych. Kompletny ich przegląd znajduje się w podrozdziale 9.10.

9.3.4. Zmienne skalarne

Zmienne skalarne to takie zmienne, które mogą przechowywać pojedynczą wartość. W języku `awk`, na wzór wielu innych języków skryptowych, zmiennych nie trzeba jawnie deklarować. Zmienne tworzone są automatycznie, w momencie kiedy po raz pierwszy pojawiają się w programie, zwykle w wyrażeniu przypisania wartości do zmiennej. Do zmiennej można przypisać wartość liczbową albo ciąg znaków. W miejscu użycia zmiennej kontekst określa, czy zmienna ma być interpretowana jako ciąg, czy liczba — interpreter automatycznie dokonuje stosownej konwersji.

Nowo tworzone zmienne programu języka `awk` są inicjalizowane ciągiem pustym, który liczbowo daje wartość zerową.

Nazwy zmiennych `awk` muszą rozpoczynać się od znaku litery ASCII albo znaku podkreślenia, na dalszych pozycjach mogą zaś zawierać również litery, znaki podkreślenia oraz cyfry. Słowem, nazwy zmienne muszą dać się dopasować do wyrażenia regularnego **`[A-Za-z_][A-Za-z0-9_]*`**. Język nie narzuca przy tym ograniczenia co do długości nazwy zmiennej.

W nazwach zmiennych w `awk` rozróżnia się wielkie i małe litery: `cos`, `Cos` i `COS` to trzy różne nazwy. Utarło się, że nazwy zmiennych lokalnych zawierają tylko małe litery, nazwy zmiennych globalnych rozpoczyna się wielką literą, a w nazwach zmiennych wbudowanych występują wyłącznie wielkie litery.

Zgodnie z powyższym, `awk` rezerwuje kilka zmiennych wbudowanych (ich nazwy zawierają rzecz jasna same wielkie litery). Najważniejsze z nich, wykorzystywane często nawet w prostych programach, zostały wymienione w tabeli 9.3.

Tabela 9.3. Najczęściej stosowane zmienne wbudowane awk

Zmienna	Znaczenie
`FILENAME`	Nazwa bieżącego pliku wejściowego.
`FNR`	Numer rekordu w bieżącym pliku wejściowym.
`FS`	Separator pól (wyrażenie regularne; domyślnie `" "`).
`NF`	Liczba pól w bieżącym rekordzie.
`NR`	Numer przetwarzanego rekordu.
`OFS`	Separator pól na wyjściu (domyślnie `" "`).
`ORS`	Separator rekordów na wyjściu (domyślnie `"\n"`).
`RS`	Separator rekordów na wejściu (w `gawk` i `nawk` jest określony wyrażeniem regularnym; domyślnie `"\n"`).

9.3.5. Zmienne tablicowe

Reguły nazywania zmiennych tablicowych w awk są identyczne, jak dla zmiennych skalarnych. Zmienna tablicowa tym się różni od skalarnej, że może przechowywać zero albo więcej elementów danych; odwołania do tych elementów konstruuje się, indeksując nazwę zmiennej tablicowej indeksem elementu.

Większość tradycyjnych języków programowania wymaga, aby tablice były indeksowane prostymi wyrażeniami dającymi wartości liczbowe-całkowite. W awk indeksy tablic, ujmowane w nawiasach prostokątnych za nazwą zmiennej tablicowej, mogą być dowolnymi wyrażeniami liczbowymi i wyrażeniami ciągów. Każdemu, dla kogo indeksowanie tablicy dowolnym typem wyrażenia jest nowością, będzie się to wydawać dziwaczne. Ale wystarczy fragment programu konstruującego katalog biurowy, aby uwidoczniła się cała wygoda tego rozwiązania:

```
telephone["janusz"] = "123-0123"
telephone["dorotka"] = "123-0146"
telephone["toto"] = "123-0459"
telephone["zbyszko"] = "123-0039"
```

Tablice dające możliwość indeksowania dowolnymi indeksami noszą miano *tablic asocjacyjnych*, bo dają możliwość kojarzenia wartości elementów z nazwami (tak, jak zwykli to czynić ludzie). Co ważne, implementacja tych tablic w awk gwarantuje wykonywanie operacji *wyszukiwania*, *wstawiania* i *usuwania* elementów tablicy w zasadniczo stałym czasie, niezależnie od liczby elementów przechowywanych w tablicy.

Tablice awk nie wymagają ani deklarowania, ani jawnego przydziału pamięci — pamięć tablicy jest alokowana dynamicznie, *w miarę* umieszczania w niej kolejnych elementów. Przy tym przydziały wykonywane są niezależnie dla poszczególnych elementów; dzięki temu można wykonać przypisanie `x[1] = 3.14159`, a zaraz potem `x[10000000] = "dziesiec milionow"` bez prowokowania niepotrzebnego przydziału elementów o indeksach od 2 do 9 999 999. Dalej, w większości języków programowania elementy tablicy muszą być tego samego typu; w awk mamy pod tym względem pełną swobodę.

Kiedy elementy tablicy przestaną być potrzebne, przydzieloną do nich pamięć można jawnie zwolnić. Służy do tego instrukcja `delete` *`tablica`*`[`*`indeks`*`]`. Nowsze implementacje awk udostępniają też instrukcję ogólniejszą, zwalniającą wszystkie elementy tablicy: `delete` *`tablica`*. Jest jeszcze inna metoda usuwania elementów tablicy — zostanie ona przedstawiona w punkcie 9.9.6.

Zmienna nie może być równocześnie skalarną i tablicową. Instrukcja `delete` usuwa elementy tablicy, ale nie zmienia *charakteru* zmiennej tablicowej — usunięcie wszystkich elementów tablicy nie zmieni jej w zmienną skalarną, przez co kod, np. taki:

```
x[1] = 123
delete x
x = 789
```

sprowokuje interpreter awk do zgłoszenia komunikatu o niemożności wykonania przypisania wartości do tablicy.

Niekiedy do jednoznacznego lokalizowania elementów tablicy trzeba zaprząc więcej niż jeden indeks. Na przykład adresata przesyłki pocztowej identyfikuje się na podstawie numeru domu, nazwy ulicy i kodu pocztowego. Dalej, skojarzenie pary wiersz-kolumna pozwala na zlokalizowanie pozycji elementu w dwuwymiarowej tabeli, jak na planszy do szachów. Z kolei w

bibliografiach poszczególne książki są identyfikowane nazwiskiem autora, tytułem, numerem wydania, nazwą wydawcy oraz rokiem wydania. Z kolei sklepikarz, jeśli ma przynieść na salę sprzedaży konkretną parę butów, musi znać producenta, nazwę modelu, kolor i rozmiar.

Tablice o mnogich indeksach można w awk symulować, stosując w roli indeksów ciągi zawierające wartości indeksów oddzielanych przecinkami. Ponieważ jednak przecinki mogą wystąpić w ciągach poszczególnych indeksów, awk zastępuje przecinki oddzielające indeksy niedrukowalnym ciągiem przechowywanym we wbudowanej zmiennej SUBSEP. Specyfikacja POSIX mówi, że wartość tej zmiennej jest zależna od implementacji; generalnie przyjęło się, że jest to wartość "\034" (znak sterujący separatora pól ASCII — FS), ale można ją dla własnych potrzeb dowolnie modyfikować. Kiedy interpreter napotyka zapis adresat["48", "Klonowa", "12-212"], konwertuje listę indeksów na ciąg "48" SUBSEP "Klonowa" SUBSEP "12-212" i dopiero tak skonstruowany ciąg wykorzystuje w roli indeksu tablicy. Interpreter można wyręczyć, samodzielnie konstruując ciąg indeksu; poniższe instrukcje dają identyczny efekt:

```
print adresat[ "48", "Klonowa", "12-212" ]
print adresat[ "48" SUBSEP "Klonowa" SUBSEP "12-212" ]
print adresat[ "48\034Klonowa", "12-212" ]
print adresat[ "48\034Klonowa\03412-212" ]
```

Trzeba jednak pamiętać, że ewentualna zmiana wartości zmiennej SUBSEP spowoduje unieważnienie indeksów do już zachowanych elementów tablicy. Dlatego zmienną SUBSEP, jeśli już jest to konieczne, należałoby zmieniać tylko raz w każdym programie, najlepiej w ramach sekcji BEGIN.

Właściwe stosowanie tablic asocjacyjnych ułatwia rozwiązywanie zaskakująco licznej grupy problemów przetwarzania danych. Dostępność tablic w prostym w sumie języku programowania, jakim jest awk, należy uznać za świetne udogodnienie.

9.3.6. Argumenty wywołania programu

Automatyzacja obsługi argumentów programu awk oznacza, że programiści korzystający z tego języka rzadko muszą szamotać się z obsługą argumentów wywołania. Odróżnia to awk od języków C, C++, Java czy nawet języka programowania powłoki, gdzie obsługa argumentów jest jawna.

Interpreter awk udostępnia argumenty wywołania za pośrednictwem wbudowanych zmiennych ARGC (licznik argumentów) i ARGV (wektor argumentów, czyli tablica wartości argumentów). Ich stosowanie najlepiej zilustrować prostym programem:

```
$ cat  showargs.awk
BEGIN {
    print "ARGC = ", ARGC
    for (k = 0; k < ARGC; k++)
        print "ARGV[" k "] = [" ARGV[k] "]"
}
```

Powyższy program sprawuje się następująco:

```
$ awk -v Jeden=1 -v Dwa=2 -f showargs.awk Trzy=3 plik1 Cztery=4 plik2 plik3
ARGC = 6
ARGV[0] = [awk]
ARGV[1] = [Trzy=3]
ARGV[2] = [plik1]
ARGV[3] = [Cztery=4]
ARGV[4] = [plik2]
ARGV[5] = [plik3]
```

Tak, jak w C i C++, argumenty wywołania są przechowywane w postaci elementów tablicy indeksowanych od 0 do ARGC - 1; element zerowy to nazwa programu interpretera awk. Tablica argumentów nie obejmuje jednak wartości przekazywanych do interpretera z opcjami -f i -v. Nie zawierałaby również (nieobecnego w powyższym wywołaniu) kodu programu awk:

```
$ awk 'BEGIN { for (k = 0; k < ARGC; k++)
>       print "ARGV[" k "] = [" ARGV[k] "]" }' a b c
ARGC = 6
ARGV[0] = [awk]
ARGV[1] = [a]
ARGV[2] = [b]
ARGV[3] = [c]
```

To, czy element zerowy będzie obejmował tylko nazwę pliku wykonywalnego interpretera awk, czy może również ścieżkę dostępu, zależne jest od implementacji:

```
$ /usr/local/bin/gawk 'BEGIN { print ARGV[0] }'
gawk

$ /usr/local/bin/mawk 'BEGIN { print ARGV[0] }'
mawk

$ /usr/local/bin/nawk 'BEGIN { print ARGV[0] }'
/usr/local/bin/nawk
```

Program awk może zmieniać wartości zmiennych ARGC i ARGV, choć doprawdy rzadko zachodzi rzeczywista potrzeba takich modyfikacji. Jeśli element tablicy ARGV zostanie ustawiony (w wywołaniu albo już w samym programie) na ciąg pusty albo usunięty, awk będzie go ignorował. Przy usuwaniu wpisów (końcowych) z tablicy ARGV należy pamiętać o odpowiednim dostosowaniu wartości ARGC.

Interpreter awk zaprzestaje prób interpretacji argumentów jako opcji wywołania, kiedy rozpozna w argumencie kod programu albo specjalną opcję --. Wszelkie argumenty występujące za tymi elementami wywołania, nawet te przypominające opcje, są zostawiane do obsługi programowi awk, który powinien je potem usunąć z ARGV albo zastąpić ciągami pustymi.

Wywołanie interpretera z programem awk wygodnie jest niekiedy ująć w skrypcie powłoki. Aby taki skrypt był bardziej cztelny, dłuższe programy należy uprzednio zapisać w zmiennej powłoki. Skrypt można też uogólnić tak, aby pozwalał na dynamiczny wybór implementacji awk na podstawie pewnej zmiennej środowiskowej; oczywiście należałoby wtedy przewidzieć implementację domyślną, np. nawk:

```
#! /bin/sh -
AWK=${AWK:-nawk}
AWKPROG='
    ...kod długiego programu...
'
$AWK "$AWKPROG" "$@"
```

Znaki pojedynczego cudzysłowu, otaczające kod programu, zabezpieczają go przed ingerencją ze strony powłoki (to jest ewentualnymi podstawieniami). Jeśli jednak sam program ma zawierać znaki pojedynczego cudzysłowu, taka ochrona nie wystarczy. Alternatywą wobec zapisywania kodu programu w zmiennej powłoki jest umieszczenie go w osobnym pliku w wydzielonym katalogu kodu współużytkowanego, wskazywanego względem katalogu, który zawiera skrypt wywołujący:

```
#! /bin/bash -
AWK=${AWK:-nawk}
$AWK -f `dirname $0`/../share/lib/myprog.awk -- "$@"
```

Polecenie dirname było już opisywane w podrozdziale 8.2. Powyższy kod zakłada, że jeśli skrypt przechowywany jest na przykład w katalogu */usr/local/bin*, wtedy plik kodu programu awk jest wyszukiwany w katalogu */usr/local/share/lib*. Zastosowanie polecenia dirname gwarantuje poprawne wyszukiwanie pliku kodu awk dopóty, dopóki zachowana zostanie względna ścieżka pomiędzy plikiem skryptu a plikiem kodu programu awk.

9.3.7. Zmienne środowiskowe

Z programu awk można odwoływać się do kompletu zmiennych środowiskowych powłoki odzwierciedlanych na czas wykonania programu w tablicy ENVIRON:

```
$ awk 'BEGIN { print ENVIRON["HOME"]; print ENVIRON["USER"] }'
/home/janusz
janusz
```

Tablica ENVIRON nie wyróżnia się niczym szczególnym: można do niej dodawać elementy, modyfikować je i usuwać. Jednakże POSIX wymaga, aby podprocesy dziedziczyły środowisko, a w żadnych testowanych przez nas implementacjach zmiany wartości elementów tablicy ENVIRON nie były propagowane ani do podprocesów, ani do funkcji wbudowanych. W szczególności oznacza to niemożność kontrolowania zachowań zależnych od schematu lokalizacji funkcji manipulujących ciągami znaków, jak tolower() — nie da się na czas jej wywołania zmienić bieżącego schematu przypisaniem do ENVIRON["LC_ALL"]. Tablicę ENVIRON należałoby więc traktować jako niemodyfikowalną perspektywę środowiska, jego lokalną kopię.

Wyborem schematu lokalizacji na potrzeby podprocesów można sterować za pośrednictwem stosownej zmiennej środowiskowej ustawianej w ciągu wywołania podprocesu. W ten sposób można z poziomu programu awk na przykład posortować wiersze pliku w oparciu o schemat lokalizacji dla języka hiszpańskiego:

```
system("env LC_ALL=es_ES sort plik_we > plik_wy")
```

Funkcja system() będzie opisywana w jednym z kolejnych podrozdziałów, w punkcie 9.7.8.

9.4. Rekordy i pola

W każdej iteracji niejawnej pętli przeglądającej pliki wejściowe, która jest podstawą modelu programistycznego awk, przetwarzany jest pojedynczy *rekord* będący zwykle pojedynczym wierszem tekstu. Rekordy dzieli się z kolei na *pola* będące podciągami wierszy.

9.4.1. Separatory rekordów

Rekordy to zazwyczaj pojedyncze wiersze tekstu rozdzielane znakami nowego wiersza; awk definiuje jednak rekordy ogólniej, na bazie specjalnego separatora rekordów określanego wartością zmiennej RS.

W tradycyjnej implementacji awk, tudzież wedle specyfikacji POSIX, zmienna RS musi być albo pojedynczym znakiem, na przykład znakiem nowego wiersza (to wartość domyślna RS), albo ciągiem pustym. W tym ostatnim przypadku stosowana jest specjalna interpretacja separatora rekordów: na rekordy składają się wtedy całe akapity tekstu, czyli grupy wierszy rozdzielane jednym bądź kilkoma wierszami pustymi; puste wiersze na początku i końcu pliku są wtedy ignorowane. Pola w tak grupowanych rekordach są oddzielane znakami nowego wiersza albo dowolnym innym separatorem definiowanym wartością zmiennej FS.

W gawk i mawk model ten doczekał się istotnego rozszerzenia: RS może określać wyrażenie regularne i może wtedy obejmować więcej niż jeden znak. Ustawienie RS = "+" ustala w roli separatora rekordów znak plusa, ale już RS = ":+" dopasowuje separator w postaci jednego bądź wielu sąsiadujących znaków dwukropka. Daje to możliwość zdecydowanie elastyczniejszego wyodrębniania rekordów — kilka przykładów zastosowań wyrażeń regularnych w roli separatorów pól znajdzie się w podrozdziale 9.6.

Jeśli separator pól jest wyrażeniem regularnym, to nie sposób wnioskować o tekście dopasowanym do wzorca separatora z samej zmiennej RS. W gawk przewidziano więc dodatkową zmienną wbudowaną RT, ustawianą po wczytaniu każdego rekordu na ciąg dopasowany do wzorca separatora (w mawk jej nie ma).

Przy braku implementacji separatorów rekordów w formie wyrażeń regularnych symulowanie takiej możliwości nie jest proste, zwłaszcza jeśli takie wyrażenia miałyby dopasowywać ciągi przekraczające granice wierszy — wszak większość narzędzi uniksowych operuje właśnie wierszami. Można pokusić się o zastosowanie polecenia tr do zamiany znaku nowego wiersza na inny, niewykorzystywany znak, scalając strumień danych wejściowych do postaci jednego gigantycznego wiersza. Wtedy mogą jednak ujawnić się rozmaite ograniczenia wynikające z niewystarczających rozmiarów buforów przydzielanych dla przetwarzanych wierszy w owych narzędziach. Na tym tle gawk, mawk i emacs wyróżniają się jako narzędzia nie narzucające wierszowej orientacji przetwarzanych danych.

9.4.2. Separatory pól

Pola w rekordzie są wyodrębniane przez dopasowanie bieżącego wyrażenia regularnego przypisanego do wbudowanej zmiennej FS, która pełni rolę separatora pól.

Domyślna wartość FS to pojedyncza spacja, ale nie jest ona interpretowana dosłownie: separator domyślny obejmuje dowolną (niezerową) liczbę znaków odstępów (spacji, tabulacji); wyodrębniane pola są „obierane" ze spacji poprzedzających i uzupełniających właściwą wartość pola. Stąd dla programu awk rekordy:

```
alfa beta gamma
    alfa      beta      gamma
```

są (przy założeniu domyślnego ustawienia FS) identyczne — oba składają się z trzech pól o wartościach "alfa", "beta" i "gamma". To szczególnie cenne, kiedy dane wejściowe są przygotowywane przez ludzi.

Jeśli zachodzi potrzeba, aby pola były separowane dokładnie jednym znakiem spacji, należy wykonać przypisanie FS = "[ ]". Przy tak określonym separatorze spacje poprzedzające i uzupełniające podciąg znaków drukowalnych wejdą do podciągu właściwej wartości pola. Można to sprawdzić na poniższych przykładach wykrywających w identycznym wierszu (rozpoczynającym się i kończącym parą spacji) wejściowym różne ilości pól:

```
$ echo '  raz dwa trzy  ' | awk -F' ' '{ print NF ":" $0 }'
3:  raz dwa trzy

$ echo '  raz dwa trzy  ' | awk -F'[ ]' '{ print NF ":" $0 }'
7:  raz dwa trzy
```

W drugim wywołaniu awk doliczył się siedmiu pól: "", "", "raz", "dwa", "trzy", "" i "".

Zmienna FS jest traktowana jak wyrażenie regularne tylko wtedy, kiedy zawiera więcej niż jeden znak. Ustawienie FS = "." oznacza więc wybranie do roli separatora pól znaku kropki; *nie jest to* w żadnym razie interpretowane jako wyrażenie regularne dopasowujące dowolny znak.

We współczesnych implementacjach awk dopuszcza się też puste wartości FS. Wtedy każdy znak stanowi osobne pole rekordu. Z kolei w starszych implementacjach przypisanie ciągu pustego do FS to rezygnacja z wyodrębniania pól — każdy rekord ma wtedy tylko jedno pole, rozciągające się na całość rekordu. POSIX mówi zaś jedynie, że zachowanie programu dla pustej wartości separatora pól jest nieokreślone.

9.4.3. Pola

Pola bieżącego rekordu są w programie awk dostępne za pośrednictwem specjalnych symboli: $1, $2, $3, ..., $NF. Indeksy symboli nie muszą być literałami (stałymi) — mogą być obliczane dynamicznie; w takim przypadku będą w razie potrzeby obcinane do wartości całkowitych. Dla k równego 3 to zarówno $k, jak i $(1+2), $(27/9), $3.14159 jak i $"3.14159" i wreszcie $3 będą odwołaniami do trzeciego pola bieżącego rekordu.

Symbol $0 odnosi się do całego bieżącego rekordu wejściowego w postaci odczytanej ze strumienia wejściowego, po obcięciu znaków (podciągów) separatora rekordów. Odwołania do symboli o numerach spoza zakresu od 0 do NF nie są odwołaniami błędnymi. Dają one w wyniku ciągi puste i nie tworzą nowych pól — chyba że wykonane zostanie przypisanie wartości do takiego symbolu. Efekt odwołania z nieliczbowym indeksem pola jest zależny od implementacji. Odwołania do pól o numerach ujemnych we wszystkich testowanych implementacjach prowokowały komunikaty o błędach krytycznych. POSIX zakłada w tej kwestii jedynie tyle, że odwołania do symboli z indeksami innymi niż dodatnie liczbowe indeksy pól dają efekt „nieokreślony".

Do pól, tak jak do zwykłych zmiennych, można przypisywać wartości. Na przykład przypisanie $1 = "alef" jest całkowicie dozwolone i poprawne, o ile pamięta się o jego efekcie ubocznym: jeśli potem nastąpi odwołanie do całego ciągu rekordu, zostanie on zmontowany z bieżących wartości pól, ale rolę separatora pól będzie przy montażu pełnić wartość wbudowanej zmiennej OFS (separator pól na wyjściu), która domyślnie zawiera pojedynczy znak spacji.

9.5. Wzorce i akcje

Sedno, właściwą treść programu w języku awk stanowią pary wzorców i akcji. To dzięki takiemu modelowi przetwarzania programy awk są tak zwarte i treściwe zarazem.

9.5.1. Wzorce

Wzorce są konstruowane z wyrażeń ciągów i (lub) wyrażeń liczbowych. Kiedy zastosowane dla bieżącego rekordu wejściowego dają wartość niezerową (logiczna „prawda"), interpreter podejmie wykonanie skojarzonej ze wzorcem akcji. Jeśli wzorzec składa się jedynie z wyrażenia regularnego, próba dopasowania obejmuje cały ciąg bieżącego rekordu — czyli tak, jakby zamiast */wyrażenie/* zapisać $0 ~ */wyrażenie/*. Oto kilka przykładów ilustrujących stosowanie wzorców:

```
NF == 0                                  wybiera rekordy puste
NF > 3                                   wybiera rekordy o co najmniej 4 polach
NR < 5                                   wybiera pierwsze cztery rekordy wejścia
(FNR == 3) && (FILENAME ~/[.][ch]$/)     wybiera trzeci rekord pliku źródłowego (nagłówkowego) języka C
$1 ~ /janusz/                            wybiera rekordy z ciągiem "janusz" w pierwszym polu
/[Xx][Mm][Ll]/                           wybiera rekordy zawierające ciąg XML (bez względu na wielkość liter)
$0 ~/[Xx][Mm][Ll]/                       jak wyżej
```

Elastyczność dopasowywania wzorców jest potęgowana możliwością stosowania *przedziałów rekordów*. Otóż dwa wyrażenia rozdzielane przecinkiem wybierają rekordy, począwszy od rekordu dopasowanego do lewego wyrażenia, po (włącznie) rekord dopasowany do prawego wyrażenia. Jeśli zdarzy się, że dany rekord pasuje do obu wyrażeń wyrażenia przedziałowego, wzorzec dopasuje tylko ten jeden rekord. Przedziały są więc konstruowane nieco inaczej niż w edytorze `sed`, gdzie rekordu kończącego przedział szukało się wśród rekordów znajdujących się *za* rekordem otwierającym. Oto kilka przykładowych wzorców z przedziałami:

```
(FNR == 3), (FNR == 10)    dopasowuje rekordy od 3. do 10. w każdym z kolejnych plików wejściowych
/<[Hh][Tt][Mm][Ll]>/, /<\/[Hh][Tt][Mm][Ll]>/    dopasowuje ciało dokumentu HTML
/[aeiouy][aeiouy]/, /[^aeiouy][^aeiouy]/        dopasowuje ciąg zaczynający się parą samogłosek
                                                a kończący parą spółgłosek
```

W obrębie akcji skojarzonej ze wzorcem `BEGIN` zmienne `FILENAME`, `FNR`, `NF` i `NR` są początkowo niezdefiniowane; odwołania do nich zwracają ciągi puste albo zera.

Jeśli program składa się jedynie z instrukcji zgrupowanych w obrębie akcji wzorca `BEGIN`, przetwarzanie kończy się po wykonaniu ostatniej instrukcji z akcji wzorca `BEGIN` — interpreter nie podejmuje przetwarzania plików wejściowych.

Na wejściu do pierwszej akcji wzorca `END` zmienna `FILENAME` zawiera nazwę ostatnio przetworzonego pliku, a zmienne `FNR`, `NF` i `NR` zachowują wartości obowiązujące dla ostatniego rekordu wejściowego. Odwołanie do `$0` w akcji skojarzonej z `END` jest odwołaniem niepewnym — w `gawk` i `mawk` (ale już nie w `nawk`) symbol ten zachowuje ciąg ostatniego rekordu. A POSIX milczy w tej kwestii.

9.5.2. Akcje

Znamy już większość elementów języka `awk` związanych z konstruowaniem wzorców, które wybierają rekordy do przetworzenia. Właściwe przetwarzanie rekordu podejmuje się w obrębie akcji kojarzonej opcjonalnie z takim wzorcem.

Język `awk` za pośrednictwem szeregu typów instrukcji i struktur pozwala na konstruowanie niemal dowolnych programów. Jednak omówienie większości instrukcji zostanie odłożone do podrozdziału 9.7. Na razie — poza instrukcjami przypisania — będziemy rozważać jedynie instrukcję `print`.

W najprostszej postaci instrukcja `print` oznacza żądanie wypisania ciągu bieżącego rekordu wejściowego (`$0`) na standardowym wyjściu i uzupełnienia tego ciągu znakiem separatora rekordów wyjściowych, określanego zmienną `ORS`, która domyślnie zawiera pojedynczy znak nowego wiersza. Skoro tak, to wszystkie poniższe programy okazują się równoważne co do efektu wyjściowego:

```
1                  wzorzec ma wartość "prawda", akcja domyślna to print
NR > 0 { print }   "wypisuj, jeśli na wejściu są rekordy"
1      { print }   wzorzec ma wartość "prawda", akcja to print z wartością domyślną
       { print }   wybór każdego rekordu (brak wzorca), akcja to print z wartością domyślną
       { print $0 } wybór każdego rekordu (brak wzorca), akcja to print z wartością jawną
```

Działanie wszystkich tych jednowierszowych programów języka awk sprowadza się do przepisania rekordów z wejścia na wyjście.

W ogólniejszym przypadku instrukcji print może towarzyszyć lista zera bądź większej liczby wyrażeń oddzielanych przecinkami. Każde z tych wyrażeń podlega wartościowaniu, ewentualnej konwersji na ciąg znaków, a następnie jest wypisywane na standardowym wyjściu. Kolejne elementy listy są na wyjściu oddzielane wartością separatora pól wyjściowych — OFS. Ostatni element jest uzupełniony ciągiem separatora rekordów wyjściowych — OSR.

Lista argumentów instrukcji print (a także instrukcji printf i sprintf, zobacz punkt 9.9.8) może być ujęta w nawiasy. Zastosowanie nawiasów eliminuje ewentualne niejednoznaczności przy przetwarzaniu listy argumentów, jeśli ta zawiera np. operatory relacji — symbole reprezentujące te operatory, czyli < i >, są bowiem wykorzystywane również w roli operatorów przekierowania wejścia-wyjścia (patrz punkty 9.7.6 i 9.7.7).

Pora na kilka przykładów kompletnych programów języka awk. Każdy z nich wypisuje na wyjściu wartości pierwszych trzech pól; brak określenia wzorca wyboru rekordu powoduje przetworzenie wszystkich rekordów wejściowych. Średniki oddzielają kolejne instrukcje programu języka awk. Efekt działania programów jest różnicowany zmianami separatora pól wyjściowych:

```
$ echo 'raz dwa trzy cztery' | awk '{ print $1, $2, $3 }'
raz dwa trzy

$ echo 'raz dwa trzy cztery' | awk '{ OFS = "..."; print $1, $2, $3 }'
raz...dwa...trzy

$ echo 'raz dwa trzy cztery' | awk '{ OFS = "\n"; print $1, $2, $3 }'
raz
dwa
trzy
```

Zmiana separatora pól wyjściowych, jeśli po niej nie nastąpi przypisanie wartości do któregoś z pól, *nie modyfikuje* ciągu $0:

```
$ echo 'raz dwa trzy cztery' | awk '{ OFS = "\n"; print $0 }'
raz dwa trzy cztery
```

Gdybyśmy jednak zmienili separator pól wyjściowych, a potem przypisali nową wartość do jednego (dowolnego) z pól, to nawet gdyby przypisanie faktycznie nie zmieniło wartości pola, interpreter dokonałby ponownego montażu $0 z uwzględnieniem nowej wartości separatora pól:

```
$ echo 'raz dwa trzy cztery' | awk '{ OFS = "\n"; $1 = $1; print $0 }'
raz
dwa
trzy
cztery
```

9.6. „Jednowierszowce" w awk

Znamy już awk na tyle, aby skutecznie konstruować krótkie, jednowierszowe programy. Mało który język pozwala na tak wiele przy tak krótkim kodzie. Przyjrzymy się więc kilku jednowierszowcom, choć niektóre z nich ze względu na ograniczenia składu zostaną rozbite

na kilka wierszy. W niektórych z tych przykładów postawione zadanie będzie rozwiązywane na kilka sposobów, z użyciem awk i alternatywnie, za pomocą innych standardowych narzędzi uniksowych.

- Na początek prosta implementacja (w awk) popularnego uniksowego narzędzia zliczającego słowa — wc:

  ```
  awk '{ C += length($0) + 1; W += NF } END { print NR, W, C }'
  ```

  Zauważ, że grupy wzorzec-akcja nie muszą być oddzielane znakami nowego wiersza, choć zwykle stosuje się je dla zwiększenia czytelności kodu. Skoro awk nie wymaga deklarowania i wstępnej inicjalizacji zmiennych, blok `BEGIN { C = W = 0 }`, choć nie zaszkodziłby, jest zbędny. Licznik znaków wejścia, przechowywany w zmiennej C, jest przy każdym rekordzie zwiększany o rozmiar tegoż rekordu i dodatkowo jeden znak, który reprezentuje wycięty z ciągu rekordu znak nowego wiersza. Licznik słów (W) zlicza liczbę pól w kolejnych wierszach. Licznik wierszy jest niepotrzebny, bo jego rolę z powodzeniem pełni wbudowana zmienna licznika rekordów, NR. Jednowierszowe zestawienie, podobne do tego generowanego przez wc, wypisuje akcja skojarzona z wzorcem END.

- Jeśli program jest pusty, awk kończy działanie bez wczytywania wejścia, może więc rywalizować z systemową czeluścią */dev/null*:

  ```
  $ time cat *.xml > /dev/null
  0.035u 0.121s 0:00.21 71.4%     0+0k 0+0io 99pf+0w
  $ time awk '' *.xml
  0.136u 0.051s 0:00.21 85.7%     0+0k 0+0io 140pf+0w
  ```

  Poza pewnymi problemami związanymi z obsługą znaków pustych awk może z powodzeniem emulować polecenie cat, jak tu, gdzie oba polecenia dają identyczny efekt:

  ```
  cat *.xml
  awk 1 *.xml
  ```

- W awk można łatwo uzupełnić kolumnę wartości liczbowych kolumną ich logarytmów:

  ```
  awk '{ print $1, log($1) }' plik(i)
  ```

- Problemem nie jest też wypisanie losowo dobranej próbki obejmującej 5% wierszy plików tekstowych (przy użyciu funkcji generatora liczb pseudolosowych — patrz podrozdział 9.10 — dającej równomierny rozkład losowanych wartości pomiędzy 0 a 1):

  ```
  awk 'rand() < 0.05' plik(i)
  ```

- Łatwo wypisać sumę wartości *n*-tej kolumny w tabeli z kolumnami oddzielanymi znakami odstępów:

  ```
  awk -v COLUMN=n '{ sum += $COLUMN } END { print sum }' plik(i)
  ```

- Po drobnej modyfikacji otrzymujemy program liczący średnią wartość *n*-tej kolumny:

  ```
  awk -v COLUMN=n '{ sum += $COLUMN } END { print sum / NR }' plik(i)
  ```

- Wypisanie łącznej kwoty wydatków zapisywanych w plikach, które zawierają rekordy składające się z opisu pozycji wydatku i kwoty wydatku w ostatnim polu, sprowadza się do odpowiedniego wykorzystania wbudowanej zmiennej NF:

  ```
  awk '{ sum += $NF;  print $0, sum }' plik(i)
  ```

- A tak można wyszukiwać ciągi w plikach tekstowych:

  ```
  egrep 'wzorzec|wzorzec' plik(i)
  awk '/wzorzec|wzorzec/' plik(i)
  awk '/wzorzec|wzorzec/ { print FILENAME ":" FNR ":" $0 }' plik(i)
  ```

- Jeśli wyszukiwanie trzeba ograniczyć do wierszy z zakresu 100 – 150, można skonstruować potok obejmujący dwa narzędzia (niestety, kosztem utraty informacji o położeniu znalezionego ciągu):

```
sed -n -e 100,150p -s plik(i) | egrep 'wzorzec'
```

  Ze względu na wywołanie z opcją `-s` potrzebna jest tu implementacja `sed` z projektu GNU; opcja ta zeruje licznik wierszy dla każdego nowego pliku wejściowego. Alternatywnie można do tego samego zadania zaangażować sprytny wzorzec `awk`:

```
awk '(100 <= FNR) && (FNR <= 150) && /wzorzec/' \
        { print FILENAME ":" FNR ":" $0 }' plik(i)
```

- Do zamiany miejscami drugiej i trzeciej kolumny czterokolumnowej tabeli (przy założeniu, że kolumny są separowane znakiem tabulacji), mogą posłużyć:

```
awk -F'\t' -v OFS='\t' '{ print $1, $2, $3, $4 }' stare > nowe
awk 'BEGIN { IFS='\t'; OFS='\t'} { print $1, $2, $3, $4 }' stare > nowe
awk -F'\t' { print $1 "\t" $2 "\t" $3 "\t" $4 }' stare > nowe
```

- Zamiana znaku separującego kolumny (tu tabulatora reprezentowanego znakiem) na znak & może odbyć się dwojako:

```
sed -e 's/ /\&/g' plik(i)
awk 'BEGIN { FS = "\t"; OFS = "&" } { $1 = $1; print }' plik(i)
```

- A oba poniższe potoki eliminują wiersze-duplikaty ze strumienia posortowanych wierszy:

```
sort plik(i) | uniq
sort plik(i) | awk ' Last != $0 { print } { Last = $0 }'
```

- Kończące wiersze pary znaków powrotu karetki i nowego wiersza można łatwo zamienić na znaki nowego wiersza:

```
sed -e 's/\r$//' plik(i)
sed -e 's/\^M$//' plik(i)
mawk 'BEGIN { RS = "\r\n" } { print }' plik(i)
```

  Pierwszy z powyższych przykładów wymaga jednej z współczesnych implementacji polecenia `sed`, rozpoznającej sekwencje sterujące. W drugim przykładzie symbol `^M` reprezentuje kombinację klawiszy *Ctrl+M*, która wprowadza znak powrotu karetki. W trzecim przykładzie należy zastosować `gawk` albo `mawk`, bo `nawk` i implementacje `awk` zgodne ze standardem POSIX nie obsługują wieloznakowych separatorów rekordów.

- Poniższe polecenia realizują konwersję pojedynczego odstępu międzywierszowego na podwójny odstęp międzywierszowy:

```
sed -e 's/$/\n/' plik(i)
awk 'BEGIN { ORS = "\n\n" } { print }' plik(i)
awk 'BEGIN { ORS = "\n\n" } 1' plik(i)
awk '{ print $0 "\n" }' plik(i)
awk 'BEGIN { print; print "" } ' plik(i)
```

  Jak poprzednio, potrzebne są tu w miarę współczesne implementacje polecenia `sed`. Zauważ, jak prosta zmiana separatora rekordów wyjściowych w pierwszym przykładzie angażującym `awk` rozwiązuje postawiony problem: cała reszta programu sprowadza się do wypisania wszystkich rekordów plików wejściowych. Pozostałe rozwiązania wykorzystujące `awk` angażują już przetwarzanie poszczególnych rekordów, przez co są zazwyczaj wolniejsze niż pierwsze.

- Równie nieskomplikowana jest konwersja odwrotna:

```
gawk 'BEGIN { RS = "\n *\n" } {print}' plik(i)
```

- Do wyszukania w kodach programów języka Fortran 77 wierszy o długości przekraczającej 72 znaki[2] można zaprząc następujące wywołania:

```
egrep -n '^.{73,}' *.f
awk 'length($0) > 72 { print FILENAME ":" FNR ":" $0 }' *.f
```

  Tu znów potrzebna jest implementacja `egrep` zgodna z POSIX, obsługująca rozszerzone wyrażenia regularne i zdolna do dopasowania 73 i więcej powtórzeń danego znaku.

- Do wyłuskania z tekstu prawidłowo zapisanego numeru ISBN (*International Standard Book Number*) trzeba zastosować przydługie, ale w zasadzie nieskomplikowane wyrażenie regularne ustawiające separator pól na wszystkie znaki, które nie mogą wchodzić w skład ISBN:

```
gawk 'BEGIN { RS = "[^-0-9Xx]" } \
 /[0-9][-0-9][-0-9][-0-9][-0-9][-0-9][-0-9][-0-9][-0-9][-0-9][-0-9]-[0-9Xx]/' \
 plik(i)
```

  W implementacjach `awk` zgodnych z POSIX takie przydługie wyrażenie regularne można skrócić do postaci `/[0-9][-0-9]{10}-[-0-9Xx]/`. Testy wykazały, że jedynymi implementacjami obsługującymi rozszerzenie w postaci możliwości powielania dopasowania są `gawk` (z opcją `--posix`), `awk` z systemów OSF/1, `awk` z systemów HP-UX, `awk` z IBM AIX i `/usr/xpg4/bin/awk` z Sun Solaris.

- Zadanie wycięcia znaczników z dokumentów HTML i potraktowania znaczników jako separatorów rekordów można zrealizować tak:

```
mawk 'BEGIN { ORS = " "; RS = "<[^<>]*>" } { print }' *.html
```

  Przypisanie spacji do separatora rekordów wyjściowych (`ORS`) wymusza konwersję znaczników HTML na spacje, przy zachowaniu podziału wierszowego z wejścia.

- A tak można wyłuskać i wypisać (po jednym w wierszu) wszystkie nagłówki z dokumentów XML, na przykład takich zawierających wstępny tekst niniejszego rozdziału. Program zadziała poprawnie, nawet jeśli tytuły będą rozwleczone po kilku wierszach; obsługuje też nieczęsty, ale dozwolony przypadek, kiedy to pomiędzy nazwą znacznika a nawiasem ostrym zamykającym element znajdzie się spacja:

```
$ mawk -v ORS=' ' -v RS='[\n]' '/<title *>/, /<\/title *>/' *.xml |
>        sed -e 's@</title *> *@&\n@g'
...
<title>Nieuzbrojony a niebezpieczny — awk</title>
<title>Dostępne nieodpłatnie wersje awk</title>
<title>Wywołanie awk</title>
...
```

  Program `awk` generuje na wyjściu pojedynczy wiersz, więc do podziału na wiersze posłużył program `sed`. Można by go wyeliminować, ale to wymagałoby zastosowania instrukcji `awk`, których omówienie odłożyliśmy do następnego podrozdziału.

[2] Niegdyś, w epoce kart perforowanych, ograniczenie rozmiaru wiersza do 72 znaków nie było problemem, ale po upowszechnieniu się terminali i ekranowych edytorów plików tekstowych przekraczanie limitu długości wiersza stało się częstą przyczyną dziwacznych błędów, które wynikają z ignorowania przez kompilator dalszego ciągu zbyt długiego wiersza kodu — *przyp. autora.*

9.7. Instrukcje awk

Języki programowania, aby były użyteczne, powinny dawać możliwość sekwencyjnego, warunkowego i iteracyjnego wykonania bloków kodu. Język awk nie odstaje tu od konkurencji, zapożyczając większość instrukcji i struktur z języka C — choć nie brak mu też konstrukcji charakterystycznych tylko dla niego.

9.7.1. Sekwencje instrukcji

Sekwencje kodu mają w awk postać listy instrukcji zapisywanych w osobnych wierszach albo rozdzielanych znakami średnika. Poniższe trzy wiersze:

```
n = 123
s = "ABC"
t = s n
```

można więc zapisać również tak:

```
n = 123; s = "ABC"; t = s n
```

Znak średnika przydaje się do izolowania poszczególnych instrukcji zwłaszcza w jednowierszowcach. W programach awk zapisywanych w osobnych plikach zwykło się jednak umieszczać instrukcje w osobnych wierszach, z rzadka stosując średniki.

Wszędzie tam, gdzie składnia języka przewiduje obecność pojedynczej instrukcji może wystąpić blok instrukcji albo *instrukcja złożona* (ang. *compound statement*) składająca się z ciągu instrukcji ujętych w nawiasy klamrowe. Akcje kojarzone z wzorcami są więc po prostu instrukcjami złożonymi.

9.7.2. Instrukcje warunkowe

Wykonanie warunkowe realizuje się w awk przy pomocy instrukcji `if` w jednej z dwóch wersji:

```
if (wyrażenie)
    instrukcja1

if (wyrażenie)
    instrukcja1
else
    instrukcja2
```

Jeśli *wyrażenie* stanowiące warunek wykonania przyjmie wartość logiczną „prawda”, podjęte zostanie wykonanie instrukcji *instrukcja1*. W innym przypadku następuje wykonanie opcjonalnej instrukcji *instrukcja2*. Obie te instrukcje mogą być rzecz jasna instrukcjami złożonymi. Ponadto każda z instrukcji może być sama w sobie instrukcją warunkową. Daje to możliwość konstruowania wielokierunkowych rozgałęzień wykonania:

```
if (wyrażenie1)
    instrukcja1
else if (wyrażenie2)
    instrukcja2
else if (wyrażenie3)
    instrukcja3
...
else if (wyrażeniek)
    instrukcjak
else
    instrukcjak+1
```

Opcjonalna, ostatnia klauzula `else`, zawsze kojarzona jest z najbliższym poprzednim `if` tego samego poziomu.

W wielokierunkowych rozgałęzieniach instrukcji `if` wyrażenia warunkowe są testowane kolejno aż do pierwszego dającego wartość „prawda" i tym samym wybierającego instrukcję do wykonania. Po jej wykonaniu sterowanie przechodzi do pierwszej instrukcji za kompletną instrukcją `if`; ewentualne pozostałe wyrażenia warunkowe tej instrukcji nie są już wartościowane. Jeśli żadne z wyrażeń nie będzie miało wartości „prawda", do wykonania jest wybierana instrukcja z ostatniej klauzuli `else` (jeśli takowa istnieje).

9.7.3. Iteracje

W `awk` programista może realizować iteracje (pętle) na cztery sposoby:

- w pętlach `while`, z testem warunku kontynuacji na początku pętli;

```
while (wyrażenie)
    instrukcja
```

- w pętlach `do ... while`, z testem warunku kontynuacji na końcu pętli;

```
do
    instrukcja
while (wyrażenie)
```

- w pętlach `for` o określonej liczbie przebiegów;

```
for (wyr1; wyr2; wyr3)
    instrukcja
```

- w pętlach `for` przeglądających elementy tablicy asocjacyjnej.

```
for (klucz in tablica)
    instrukcja
```

W znakomitej większości zadań wymagających powtarzalnego wykonania kodu wystarczająca jest pętla `while`. Pętla `do ... while` jest znacznie mniej popularna — pojawia się na przykład w problemach optymalizacyjnych sprowadzających się do *wykonania obliczenia, oszacowania odchylenia i powtórzenia obliczenia, jeśli odchylenie jest zbyt duże*. Obie pętle są wykonywane dopóty, dopóki wyrażenie warunkowe ma wartość niezerową (wartość logiczną „prawda"). Jeśli pierwotnie wartością wyrażenia warunkowego będzie zero, ciało pętli `while` nie zostanie wykonane ani razu. Pętla `do ... while` dla analogicznego przypadku zostanie wykonana jednokrotnie (bo warunek zostanie sprawdzony dopiero po pierwszej iteracji).

Pierwsza z postaci pętli `for` określona jest trzema wyrażeniami oddzielonymi średnikami; każde z tych wyrażeń (a także wszystkie trzy) może być puste. Pierwsze wyrażenie jest wartościowane przed rozpoczęciem pętli. Drugie jest wartościowane na początku każdej iteracji. Warunkiem wykonania instrukcji ciała pętli jest niezerowa („prawda") wartość tego wyrażenia. Wreszcie trzecie wyrażenie jest wartościowane po wykonaniu przebiegu pętli. Tradycyjną pętlę od 1 do *n* zapisuje się tak:

```
for (k = 1; k <= n; k++)
    instrukcja
```

Ale indeks pętli niekoniecznie musi się zwiększać po każdym przebiegu. Równie dobrze można pętlą `for` odliczać wstecz, jak tu:

```
for (k = n; k >= 1; k--)
    instrukcja
```

Z racji nieodłącznej niedokładności obliczeń zmiennoprzecinkowych należy unikać stosowania w pętli `for` wyrażeń dających w wyniku wartości niecałkowite. Na przykład pętla:

```
$ awk 'BEGIN { for (x = 0; x <= 1; x += 0.05) print x }'
...
0.85
0.9
0.95
```

wcale nie wyświetli w ostatnim przebiegu wartości 1, bo wielokrotne dodawanie nieprecyzyjnie reprezentowanej wartości 0,05 daje ostatecznie wartość `x` nieznacznie większą od 1 (a warunek wykonania pętli dopuszcza najwyżej równość `x` z 1).

Programiści języka C powinni zapamiętać, że w `awk` nie ma operatora przecinka, więc w wyrażeniach pętli `for` nie wolno stosować list wyrażeń.

Druga postać pętli `for` służy do iteracyjnego przeglądania elementów tablic asocjacyjnych o nieznanej z góry liczbie elementów albo takich, których sekwencji indeksowania nie da się wyrazić szeregiem całkowitych wartości liczbowych. Elementy tablicy do przetwarzania w kolejnych iteracjach są wybierane w bliżej nieokreślonej kolejności, więc poniższy kod:

```
for (name in telephone)
    print name "\t" telephone[name]
```

raczej nie spowoduje wypisania elementów tablicy `telephone` w jakiejkolwiek oczekiwanej kolejności. Problem ten można rozwiązać metodami prezentowanymi w punkcie 9.7.7. Tablice „wielowymiarowe", indeksowane wieloma indeksami, można uprzednio przetworzyć funkcją `split()`, opisywaną w punkcie 9.9.6.

Do przerywania pętli służy (tak, jak w języku programowania powłoki) instrukcja `break`:

```
for (name in telephone)
    if (telephone[name] == "123-0123")
        break

print "Numer telefonu" name "to 123-0123"
```

W `awk` brakuje jednak znanej z powłoki możliwości jednorazowego przerywania dowolnej liczby zagnieżdżonych pętli instrukcją `break` *n*.

Instrukcja `continue` pozwala wymusić (również podobnie, jak w powłoce) przeskok na koniec ciała pętli i przejście do następnej iteracji. Język `awk` nie obsługuje przy tym wersji wielopoziomowej tego polecenia, znanej z powłoki `continue` *n*. Ilustrację działania instrukcji `continue` w `awk` stanowi listing 9.1; ów implementuje siłową metodę (przez sprawdzanie kolejnych podzielników) sprawdzania, czy dana liczba jest liczbą pierwszą (dla przypomnienia, liczba pierwsza to każda całkowita liczba pierwsza, która dzieli się bez reszty tylko przez 1 i przez samą siebie). Pogram wypisuje też wytypowany dla liczby podział na czynniki pierwsze.

Listing 9.1. Czynniki pierwsze liczb całkowitych

```
# Wyznacza czynniki pierwsze liczb całkowitych podawanych na wejście
# (po jednej w wierszu)
#
# Stosowanie:
#    awk -f factorize.awk
{
    n = int($1)
    m = n = (n >= 2) ? n : 2
```

```
    factors = ""
    for (k = 2; (m > 1) && (k^2 <= n); )
    {
        if (int(m % k) != 0)
        {
            k++
            continue
        }
        m /= k
        factors = (factors == "") ? ("" k) : (factors " * " k)
    }
    if ((1 < m) && (m < n))
        factors = factors " * " m
    print n, (factors == "") ? " to liczba pierwsza" : ("= " factors)
}
```

Zauważmy, że po zwiększeniu licznika pętli `k` następuje przejście do następnej iteracji (instrukcja `continue`), ale całość odbywa się tylko wtedy, kiedy `k` *nie jest* całkowitym podzielnikiem `m`, stąd puste wyrażenie w miejsce trzeciego wyrażenia pętli `for`.

Gdyby uruchomić program dla próbki danych testowych, otrzymalibyśmy następujące wyniki:

```
$ awk -f factorize.awk test.dat
2147483540 = 2 * 2 * 5 * 107374177
2147483541 = 3 * 7 * 102261121
2147483542 = 2 * 3137 * 342283
2147483543 to liczba pierwsza
2147483544 = 2 * 2 * 2 * 3 * 79 * 1132639
2147483545 = 5 * 429496709
2147483546 = 2 * 13 * 8969 * 9209
2147483547 = 3 * 3 * 11 * 21691753
2147483548 = 2 * 2 * 7 * 76695841
2147483549 to liczba pierwsza
2147483550 = 2 * 3 * 5 * 5 * 19 * 23 * 181 * 181
```

9.7.4. Sprawdzanie przynależności do tablicy

Test przynależności *`klucza`* do *`tablicy`* — w postaci *`klucz`* `in` *`tablica`* — to wyrażenie przyjmujące wartość 1 („prawda”), jeśli *`klucz`* jest indeksem istniejącego elementu *`tablicy`*. Wynik testu można odwrócić operatorem negacji: `!(klucz in tablica)` zwraca 1 („prawda”), kiedy *`klucz`* nie jest poprawnym indeksem *`tablicy`*. Ujęcie negowanego testu w nawiasy jest konieczne.

W przypadku tablic o wielokrotnych indeksach można w miejsce klucza zastosować ujętą w nawiasy listę indeksów: `(`*`i`*`, `*`j`*`, ..., `*`n`*`) in `*`tablica`*.

Test przynależności w żadnym razie nie prowokuje wstawienia elementu do tablicy; to ważne, bo każde zwykłe odwołanie do nieistniejącego elementu tablicy tworzy taki element. Stąd, zamiast:

```
if (telephone["dorotka"] != "")
    print "Mamy telefon Dorotki w katalogu"
```

należałoby raczej pisać:

```
if ("dorotka" in telephone)
    print "Mamy telefon Dorotki w katalogu"
```

Pierwsza z tych wersji spowoduje wszak wstawienie do tablicy elementu z pustym numerem telefonu.

Trzeba też rozróżniać wyszukiwanie *indeksu* od wyszukiwania konkretnej *wartości* elementu. Otóż test przynależności indeksu elementu jest realizowany w stałym, niezależnym od ilości elementów tablicy, czasie. Tymczasem wyszukiwanie wartości odbywa się w czasie proporcjonalnym do liczby elementów, co ilustruje przykład demonstrujący stosowanie instrukcji `break` w pętli `for`. Jeśli w programie przewiduje się częste wyszukiwanie kluczy i wartości, warto rozważyć utworzenie drugiej tablicy — odwracającej skojarzenie indeks-wartość:

```
for (name in telephone)
    name_by_telephone[telephone[name]] = name
```

Do tak skonstruowanej tablicy można odwoływać się wyrażeniem `name_by_telephone--["123-0123"]`, co da w wyniku (zwracanym w czasie liniowym, niezależnym od liczby elementów tablicy) ciąg `"janusz"`. Utworzenie zwierciadlanej tablicy asocjacyjnej jest możliwe pod warunkiem, że wartości pierwotnej tablicy będą w jej obrębie niepowtarzalne. Jeśli z danego numeru telefonu będzie korzystać więcej niż jedna osoba, w tablicy `name_by_telephone` wyląduje jedynie element ostatniej z tych osób. Problem ten można wyeliminować, rozbudowując nieco kod tworzący tabelę zwierciadlaną:

```
for (name in telephone)
{
    if (telephone[name] in name_by_telephone)
        name_by_telephone[telephone[name]] = \
            name_by_telephone[telephone[name]] "\t" name
    else
        name_by_telephone[telephone[name]] = name
}
```

Teraz tablica `name_by_telephone` będzie w elementach numerów telefonów przypisanych do wielu osób zawierać listę tych osób (oddzielaną znakami tabulacji).

9.7.5. Inne instrukcje kontroli przepływu sterowania

Omówiliśmy już instrukcje `break` i `continue` służące do selektywnego zaburzania przepływu sterowania w pętlach. Niekiedy zachodzi też potrzeba ingerencji w przepływ sterowania w zakresie dopasowywania rekordów wejściowych do par wzorzec-akcja. Tu można wyróżnić trzy przypadki:

Zaprzestanie dopasowywania wzorca dla bieżącego rekordu
: Służy do tego instrukcja `next`. W niektórych implementacjach nie można jej stosować we własnych funkcjach (opisywanych w podrozdziale 9.8).

Zaprzestanie dopasowywania wzorca dla reszty rekordów bieżącego pliku
: Takie żądanie wyraża instrukcja `nextfile` implementowana w `gawk` i ostatnich wydaniach `nawk`. Wymusza ona natychmiastowe zamknięcie bieżącego pliku wejściowego i wznowienie dopasowywania wzorców do rekordów następnego pliku wymienionego w wywołaniu programu.

  Instrukcję `nextfile` można w starszych implementacjach łatwo symulować kosztem pewnego zmniejszenia wydajności programu. Otóż instrukcję `nextfile` można z powodzeniem zastąpić parą instrukcji `SKIPFILE = FILENAME; next`, pod warunkiem uzupełnienia programu poniższymi parami wzorzec-akcja (należy je umieścić na początku programu):

```
FNR == 1                { SKIPFILE = "" }
FILENAME == SKIPFILE    { next }
```

Pierwsza z powyższych par ustawia zmienną SKIPFILE na ciąg pusty. Odbywa się to przed rozpoczęciem przetwarzania każdego pliku (FNR == 1), dzięki czemu program będzie działał poprawnie również wtedy, kiedy w wywołaniu pojawią się kolejno identyczne nazwy pliku (czyli program zostanie wywołany dwa razy dla tego samego pliku). Co prawda interpreter kontynuuje wczytywanie rekordów bieżącego pliku, ale każdy z nich jest ignorowany (instrukcja next). Po osiągnięciu końca pliku, przy otwieraniu następnego, drugi wzorzec nie daje się już dopasować, więc akcja next jest ignorowana.

Zaprzestanie wykonywania programu i zwrócenie kodu powrotnego do powłoki
Realizowane jest poleceniem exit *n* (gdzie *n* jest wartością kodu powrotnego).

9.7.6. Kontrola wejścia

Przezroczystość, automatyzm obsługi plików wejściowych wymienianych w wywołaniu programu awk pozwala większości programów na uwolnienie się od kłopotów związanych z samodzielnym otwieraniem i przetwarzaniem zawartości plików. Nie znaczy to, że nie można regulować procesu wczytywania danych z wejścia; otóż można i służy do tego instrukcja getline. Okazuje się przydatna na przykład w programach kontrolujących poprawność pisowni, które z reguły przed rozpoczęciem właściwego przetwarzania zbioru wejściowego muszą wczytać do programu stosowny słownik (słowniki).

Instrukcja getline zwraca wartość i dzięki temu może być wykorzystywana jak wywołanie funkcji, choć nim nie jest — jest instrukcją i to instrukcją o dość dziwacznej składni. Wartością zwracaną jest +1, kiedy uda się wczytać dane z wejścia, 0, kiedy odczyt utknie na końcu pliku, bądź –1, w przypadku błędu odczytu. Instrukcja getline może być wykorzystywana na kilka sposobów, podsumowanych w tabeli 9.4.

Tabela 9.4. Odmiany instrukcji getline

Składnia	Działanie
`getline`	Wczytuje następny rekord z bieżącego pliku wejściowego do symbolu $0, aktualizując przy okazji wartości zmiennych NF, NR i FNR.
`getline` *zmienna*	Wczytuje następny rekord z bieżącego pliku wejściowego do zmiennej *zmienna*, aktualizując przy okazji wartości zmiennych NF, NR i FNR.
`getline <` *plik*	Wczytuje następny rekord z pliku *plik* do symbolu $0, aktualizując przy okazji wartość zmiennej NF.
`getline` *zmienna* `<` *plik*	Wczytuje następny rekord z pliku *plik* do zmiennej *zmienna*.
polecenie`|getline`	Wczytuje do symbolu $0 następny rekord wypisywany na wyjście polecenia zewnętrznego *polecenie*, aktualizuje wartość zmiennej NF.
polecenie`|getline` *zmienna*	Wczytuje do zmiennej *zmienna* następny rekord wypisywany na wyjście polecenia zewnętrznego *polecenie*.

Sprawdźmy, jak wykorzystuje się poszczególne warianty getline. Na początek spróbujmy zadać użytkownikowi pytanie i wczytać wpisaną na wejście programu odpowiedź:

```
print "Ile wynosi pierwiastek kwadratowy z 625?"
getline answer
print "Odpowiedz (", answer, ") ", (answer == 25) ? "poprawna." : "niepoprawna."
```

Aby mieć pewność, że odpowiedź została wprowadzona z terminala, z którego wywołano program, a nie po prostu przekazana na standardowe wejście programu, można w instrukcji `getline` wskazać jako źródło plik terminala:

```
getline answer < "/dev/tty"
```

Spróbujmy teraz wczytać listę słów z pliku słownika:

```
nwords = 1
while ((getline words[nwords] < "/usr/dict/words") > 0)
    nwords++
```

Programista języka `awk` może z poziomu programu konstruować potoki. Potok taki jest definiowany ciągiem znaków i może zawierać wywołania dowolnych poleceń powłoki. Korzysta się z niego za pośrednictwem instrukcji `getline` w sposób następujący:

```
"date" | getline now
close("date")
print "Aktualny czas to ", now
```

W większości implementacji liczba równocześnie otwieranych plików jest ograniczana, więc po wyczerpaniu potoku najlepiej zamknąć plik potoku wywołaniem funkcji `close()`. W starszych implementacjach `close` było instrukcją (nie funkcją); nie istnieje przez to niezawodny i przenośny sposób wywołania `close` za pomocą składni wywołania funkcji i otrzymania wiarygodnego kodu powrotnego.

Potok może skutecznie zasilać pętlę:

```
command = "head -n 15 /etc/hosts"
while ((command | getline s) > 0)
    print s
close(command)
```

Zastosowana tu zmienna przechowująca zawartość potoku eliminuje konieczność wielokrotnego powtarzania potencjalnie złożonego ciągu polecenia i gwarantuje jednorodność potoku. Mianowicie w ciągach wywołań poleceń zewnętrznych ważny jest każdy znak, więc nawet nieznaczna z pozoru modyfikacja ciągu w postaci dodania gdzieś jednej spacji oznaczałaby odwołanie do zupełnie innego polecenia.

9.7.7. Przekierowywanie wyjścia

Normalnie instrukcje `print` i `printf` (zobacz punkt 9.9.8) wypisują dane na standardowe wyjście programu. Wyjście to można jednak na przykład skierować do pliku:

```
print "Ahoj, przygodo!" > plik
printf("%d do dziesiatej potegi to %d\n", 2, 2^10) > "/dev/tty"
```

Jeśli wyjście ma być dołączane do istniejącego już pliku (a w przypadku jego braku tworzyło taki plik) należy skorzystać z operatora >>:

```
print "Ahoj, przygodo!" >> plik
```

Operator przekierowania wyjścia można zastosować z wskazaniem tego samego pliku docelowego w dowolnej liczbie instrukcji wypisujących dane na wyjście. Po zakończeniu wypisywania należy pamiętać o zamknięciu pliku docelowego wywołaniem `close(plik)`.

Nie należy mieszać operatorów > i >>, jeśli miałyby odwoływać się do tego samego pliku, a między ich wywołaniami zabrakłoby wywołania `close()`. W `awk` operatory te wskazują tryb otwarcia pliku. Raz otwarty plik pozostaje otwarty aż do momentu jawnego zamknięcia

albo zakończenia programu. Przekierowanie działa więc inaczej niż w powłoce, gdzie każdy operator przekierowania oznaczał zarówno otwarcie, jak i zamknięcie pliku.

Program `awk` może też stanowić źródło danych dla potoku:

```
for (name in telephone)
    print name "\t" telephone[name] | "sort"
close("sort")
```

Tu również wypada pamiętać o jak najszybszym zamknięciu potoku (po zakończeniu wypisywania danych). Ma to istotne znaczenie zwłaszcza wtedy, kiedy wypisane wyjście ma być w ramach tego samego programu wczytane na wejście. Na przykład kiedy wyjście zostanie skierowane do pliku tymczasowego, a program po skompletowaniu pliku wczyta jego zawartość:

```
tmpfile = "/tmp/telephone.tmp"
command = "sort > " tmpfile
for (name in telephone)
    print name "\t" telephone[name] | command
close(command)
while ((getline < tmpfile) > 0)
    print
close(tmpfile)
```

Możliwość korzystania z potoków otwiera programiście `awk` możliwość korzystania z całego dostępnego w Uniksie zestawu narzędzi, eliminując w znacznym stopniu potrzebę korzystania z obszernych bibliotek zewnętrznych charakterystycznych dla innych języków programowania, przy zachowaniu zwartości programów i samego języka `awk`. Na przykład język `awk` nie udostępnia wbudowanej funkcji sortowania ciągów, bo byłoby to niepotrzebnym powielaniem implementacji dostępnych zewnętrznie narzędzi, a konkretnie polecenia powłoki `sort`, opisywanego w podrozdziale 4.1.

Nowsze implementacje `awk` (niekoniecznie te bazujące na specyfikacji POSIX) udostępniają funkcję `fflush(`*`plik`*`)`, służącą do opróżniania (ang. *flush*) bufora strumienia wyjściowego skojarzonego z *`plikiem`*. Funkcja zwraca wartość 0 w przypadku powodzenia lub –1 w razie porażki. Efekt wywołania `fflush()` (bez argumentu) czy `fflush("")` (argument w postaci ciągu pustego) jest zależny od implementacji — takich konstrukcji należy unikać, zwłaszcza w programach aspirujących do miana przenośnych.

9.7.8. Uruchamianie programów zewnętrznych

Wiadomo już, że za pośrednictwem instrukcji `getline` i operatorów przekierowania wyjścia można komunikować się z poziomu programu `awk` z programami zewnętrznymi. Wywołanie programu zewnętrznego można też zrealizować funkcją `system(`*`polecenie`*`)`. Wartością zwracaną przez tę funkcję jest kod powrotny *`polecenia`*. Wywołanie funkcji `system()` prowokuje poróżnienie buforowanych danych wyjściowych i uruchomienie egzemplarza procesu powłoki `/bin/sh` wywołanej z zadanym *`poleceniem`*. Standardowe wyjście diagnostyczne i standardowe wyjście tej powłoki są kierowane do tego samego ujścia, do którego wypisuje swoje dane program `awk` — chyba że wywołanie polecenia zawierać będzie operatory przekierowania.

Znajomość funkcji `system()` pozwala na skrócenie kodu programu sortującego biurową książkę telefoniczną — zamiast potoku `awk` można w nim wykorzystać wywołanie `system()` i plik tymczasowy:

```
tmpfile = "/tmp/telephone.tmp"
for (name in telephone)
    print name "\t" telephone[name] > tmpfile
close(tmpfile)
system("sort < " tmpfile)
```

Plik tymczasowy musi zostać zamknięty jeszcze przed zainicjowaniem wywołania `system()`, inaczej może dojść do utraty części buforowanych danych, które nie zostały zapisane w pliku.

Dla poleceń uruchomionych funkcją `system()` nie trzeba wywoływać funkcji `close()`, bo `close()` operuje jedynie na plikach i potokach otwieranych operatorami przekierowania oraz instrukcjami `getline`, `print` i `printf`.

Funkcja `system()` może stanowić wygodny środek usuwania plików tymczasowych tworzonych przez skrypt:

```
system("rm -f " tmpfile)
```

Ciąg polecenia przekazywany do powłoki uruchamianej wywołaniem funkcji `system()` może składać się z wielu wierszy:

```
system("cat << EOFILE"\nraz\ndwa\ntrzy\nEOFILE")
```

Powyższe polecenie sprowokuje na wyjściu wypis utworzony przez skopiowanie na wyjście programu `awk` wyjścia polecenia `cat` korzystającego z wejścia wsobnego:

```
raz
dwa
trzy
```

Każde wywołanie funkcji `system()` inicjuje osobny proces powłoki; nie istnieje przez to prosty sposób przekazywania danych pomiędzy poleceniami uruchomionymi osobnymi wywołaniami `system()`. Najprostszym sposobem jest chyba zastosowanie plików pośrednich (tymczasowych). Ale i temu można zaradzić — wystarczy za pomocą potoku wyjściowego do powłoki uruchomić w niej wiele poleceń:

```
shell = "/usr/local/bin/ksh"
print "export INPUTFILE=/var/tmp/plik.we" | shell
print "export OUTPUTFILE=/var/tmp/plik.wy" | shell
print "env | grep PUTFILE" | shell
close(shell)
```

Takie rozwiązanie ma tę dodatkową zaletę, że pozwala na wybranie powłoki do uruchomienia; wadą jest zaś brak przenośnego sposobu pobrania kodu powrotnego.

9.8. Funkcje definiowane przez użytkownika

Omówione dotychczas instrukcje `awk` wystarczyłyby do napisania niemal każdego programu przetwarzającego dane. Ale ponieważ programiści, jako ludzie, a więc istoty ułomne, mają problemy z postrzeganiem i właściwą analizą zbyt rozległych bloków kodu, najlepiej kod ten podzielić na bloki, z których każdy wykonywałby jakąś dobrze wyodrębnione zadanie. W większości języków programowania możliwość taka jest realizowana za pośrednictwem rozmaicie nazywanych funkcji, procedur, podprogramów, metod, pakietów i tym podobnych. Dla uproszczenia `awk` zna tylko funkcje. Funkcje języka `awk` mogą (jak w C) opcjonalnie zwracać skalarne wartości. Interpretacja wartości zwracanej jest całkowicie dowolna i wnioskuje się o niej z dokumentacji funkcji albo (w przypadku naprawdę krótkich funkcji) samego kodu ciała funkcji.

Funkcje można definiować w dowolnym miejscu programu na jego głównym poziomie, to znaczy przed, pomiędzy albo za parami wzorzec-akcja. W programach jednoplikowych definicje funkcji umieszcza się zwykle za blokiem kodu definiującym pary wzorzec-akcja. Przyjęło się też, że definicje funkcji są porządkowane alfabetycznie (według nazw funkcji). Sam język awk nie narzuca żadnych tego rodzaju konwencji — to tylko i aż niepisana umowa pomiędzy programistami.

Definicja funkcji prezentuje się następująco:

```
function nazwa(arg1, arg2, ..., argn)
{
    instrukcja(instrukcje)
}
```

Nazwane argumenty funkcji występują w obrębie jej ciała w roli zmiennych lokalnych względem funkcji, przesłaniając istniejące ewentualnie zmienne globalne o takich samych nazwach. Użycie funkcji w programie polega na jej wywołaniu w jednej z dwóch możliwych form, które zaprezentowano poniżej:

```
nazwa(wyr1, wyr2, ..., wyrn)            wywołanie bez zachowania wartości zwracanej

wynik = nazwa(wyr1, wyr2, ..., wyrn)    wywołanie z zachowaniem wartości zwracanej w zmiennej
```

Wyrażenia umieszczone w nawiasie za nazwą wywoływanej funkcji inicjalizują argumenty funkcji. Nawias zawierający listę argumentów powinien znajdować się bezpośrednio za nazwą funkcji, bez dodatkowych znaków odstępów.

Zmiany wprowadzone w toku wykonania ciała funkcji do wartości jej argumentów skalarnych nie są widoczne na zewnątrz funkcji. Innymi słowy, argumenty skalarne są przekazywane do funkcji przez wartość, tablice są natomiast przekazywane przez referencję (podobny model przekazywania argumentów obowiązuje w języku C).

Do zakończenia wykonania ciała funkcji i zwrócenia sterowania do wywołującego służy instrukcja `return wyrażenie`; wartość *wyrażenia* staje się wartością zwracaną funkcji. W razie jego braku wartość zwracana zależy od implementacji. We wszystkich testowanych przez nas systemach funkcje pozbawione jawnej instrukcji zwracającej wartość zwracały albo zero, albo ciąg pusty. Standard POSIX nie reguluje przypadku wywołania instrukcji `return` bez wyrażenia określającego wartość zwracaną.

Wszystkie zmienne wykorzystywane w obrębie ciała funkcji, a które nie występują na liście argumentów wywołania funkcji, są zmiennymi *globalnymi*. Język awk dopuszcza wywołania funkcji z mniejszą niż deklarowana liczbą argumentów. Dodatkowe, niezainicjalizowane argumenty mogą być wykorzystane jako zmienne *lokalne* funkcji. Zmienne takie przyjęło się wymieniać w liście argumentów funkcji, oddzielając je od właściwych argumentów paroma dodatkowymi znakami odstępu, jak na listingu 9.2. Jak wszystkie inne zmienne języka awk, dodatkowe argumenty są pierwotnie (w momencie wywołania funkcji) inicjalizowane ciągami pustymi.

Listing 9.2. Funkcja przeszukująca tablicę

```
function find_key(array, value,            key)
{
    # Szuka value w tablicy array[], zwraca klucz key taki, aby
    # array[key] == value; przy niepowodzeniu wyszukiwania zwraca ""
```

```
    for (key in array)
        if (array[key] == value)
            return key
    return ""
}
```

Pominięcie zmiennych lokalnych w liście argumentów w definicji funkcji jest przyczyną trudnych do rozpoznania błędów polegających na niezamierzonym zamazywaniu z wnętrza funkcji wartości zmiennych globalnych. W `gawk` dostępna jest opcja `--dumpvariables`, pomocna w wyśledzeniu tego rodzaju przypadków.

Funkcje programu `awk` mogą oczywiście wywoływać same siebie — powstaje wtedy *rekurencja*. Oczywiście w takim przypadku programista powinien zadbać o jakiś mechanizm przerwania rekurencji — zwykle rekurencja jest pomyślana tak, aby w następnych zagnieżdżonych wywołaniach zbiór danych do przetworzenia był coraz mniejszy, aż w pewnym momencie jest redukowany całkowicie i rekurencja się kończy. Rekurencyjne wywołanie funkcji prezentowane jest na listingu 9.3, na przykładzie klasycznego już problemu obliczeniowego polegającego na wyszukiwaniu największego wspólnego mianownika dwóch liczb całkowitych. Rozwiązanie zostało zaimplementowane według algorytmu przypisywanego Euklidesowi (około 300 roku p.n.e.), choć prawdopodobnie był już znany przynajmniej dwieście lat wcześniej.

Listing 9.3. Euklidesowy algorytm szukania największego wspólnego dzielnika

```
function gcd(x, y,         r)
{
    # Zwraca największy wspólny dzielnik dwóch liczb całkowitych x i y

    x = int(x)
    y = int(y)
    # print x, y
    r = x % y
    return (r == 0) ? y : gcd(y, r)
}
```

Gdyby kod z listingu 9.3 uzupełnić o akcję:

```
{ g = gcd($1, $2); print "gcd(" $1 ", " $2 ") =", g }
```

a następnie usunąć znacznik komentarza z wiersza kodu funkcji `gcd` zawierającego instrukcję `print` i uruchomić całość z pliku, można by obserwować zagłębianie się rekurencji:

```
$ echo 25770 30972 | awk -f gcd.awk
25770 30972
30972 25770
25770 5202
5202 4962
4962 240
240 162
162 78
78 6
gcd(25770, 30972) = 6
```

Algorytm Euklidesa obejmuje stosunkowo małą liczbę kroków rekurencji, więc nie wprowadza dużego ryzyka przepełnienia *stosu wywołań*, czyli utrzymywanego przez interpreter `awk` rejestru zagnieżdżonych wywołań funkcji. Jednak nie zawsze rekurencja jest tak korzystna i bezproblemowa. Istnieje na przykład dość złośliwa funkcja odkryta w 1926 roku przez niemieckiego matematyka Wilhelma Ackermanna[3], która cechuje się niezwykle szybkim, bo większym od

[3] Tło historyczne odkrycia i właściwości samej funkcji opisywane są między innymi na stronie *http://mathworld.wolfram.com/AckermannFunction.html — przyp. autora.*

wykładniczego, wzrostem wartości i głębokości rekurencji. W języku awk można ją wyrazić kodem z listingu 9.4.

Listing 9.4. Funkcja Ackermanna — ekstremalnie szybki wzrost liczby kroków rekurencyjnych

```
function ack(a, b)
{
    N++                    # licznik głębokości rekurencji
    if (a == 0)
        return (b + 1)
    else if (b == 0)
        return (ack(a - 1, 1))
    else
        return (ack(a - 1, ack(a, b - 1)))
}
```

Gdyby uzupełnić powyższy kod akcją:

```
{ N = 0; print "ack(" $1 ", " $2 ") = ", ack($1, $2), "[" N "krokow rekurencji]" }
```

I uruchomić całość z pliku z kilkoma zestawami testowych argumentów funkcji, otrzymalibyśmy:

```
$ echo 2 2 | awk -f ackermann.awk
ack(2, 2) =  7 [27 krokow rekurencji]

$ echo 3 3 | awk -f ackermann.awk
ack(3, 3) =  61 [2432 krokow rekurencji]

$ echo 3 4 | awk -f ackermann.awk
ack(3, 4) =  125 [10307 krokow rekurencji]

$ echo 3 8 | awk -f ackermann.awk
ack(3, 8) =  2045 [2785999 krokow rekurencji]
```

Wartości `ack(4, 4)` nie da się już obliczyć z powodu przepełnienia stosu wywołań.

9.9. Funkcje operujące na ciągach

W punkcie 9.3.2 zasygnalizowaliśmy tematykę funkcji operujących na ciągach wzmianką o funkcji `length(`*`ciąg`*`)`, która zwraca rozmiar (liczbę znaków) ciągu *ciąg*. Wśród innych popularnych funkcji manipulujących ciągami są funkcje konkatenacji, formatowania, konwersji wielkości znaków, dopasowywania wzorców, wyszukiwania podciągów, podziału ciągów, wstawiania i zastępowania podciągów oraz funkcje wyłuskiwania podciągów.

9.9.1. Wyłuskiwanie podciągów

Składnia wywołania funkcji wyodrębniania podciągów to `substr(`*`ciąg`*`, `*`start`*`, `*`długość`*`)`. Funkcja zwraca kopię podciągu ciągu *ciąg* przekazanego w wywołaniu, o zadanej *długości* i rozpoczynającym się od *start* w pierwotnym ciągu. Znaki ciągu są liczone od 1: `substr("abc--de", 2, 3)` zwraca ciąg `"bcd"`. Argument długości podciągu może zostać pominięty — w takim układzie dla trzeciego argumentu wywołania zostanie przyjęta wartość domyślna, obliczana jako `length(`*`ciag`*`) - `*`start`*` + 1`. Słowem, przy braku ograniczenia funkcja skopiuje maksymalnie długi podciąg.

Przekazanie do funkcji `substr()` argumentów wskazujących poza ciąg źródłowy nie jest traktowane jako błąd, ale wyniki takiego wywołania są zależne od implementacji. Na przykład w `nawk` i `gawk` wywołanie `substr("ABC", -3, 2)` zwraca ciąg `"AB"`, podczas gdy w `mawk` zwracany jest wtedy ciąg pusty (`""`). Dla wywołania `substr("ABC", 4, 2)` wszystkie te im-

plementacje zwracają ciąg pusty. Z kolei wywołanie `substr("ABC", 1, 0)` interpreter `gawk` uruchomiony z opcją `--lint` kwituje komunikatem o wykroczeniu wartości argumentów poza dopuszczalny zakres.

9.9.2. Konwersja wielkości liter

W niektórych alfabetach litery występują w dwóch wariantach — rozróżnia się litery wielkie i małe. Z kolei przy wyszukiwaniu i dopasowywaniu wzorców niekiedy zachodzi potrzeba ignorowania różnic pomiędzy znakami, jeśli sprowadzają się wyłącznie do wielkości litery. Można ewentualnie przed podjęciem takiej operacji zrównać wielkość liter w ciągu; w `awk` można do tego wykorzystać funkcje `tolower()` i `toupper()`. Wywołanie `tolower(`*`ciąg`*`)` zwraca kopię ciągu *ciąg* różniącą się od oryginału tym, że wszystkie wielkie litery zostały w nim zastąpione swoimi odpowiednikami ze zbioru liter małych; `toupper(`*`ciąg`*`)` działa dokładnie odwrotnie — w zwróconej kopii ciągu *ciąg* wszystkie małe litery są zastępowane wielkimi. Wartością `tolower("aBcDeF123")` jest więc ciąg `"abcdef123"`, a `toupper("aBcDeF123")` zwraca ciąg `"ABCDEF123"`. Funkcje te świetnie sprawdzają się przy konwersji wielkości liter kodowanych w ASCII, ale nie radzą sobie z konwersjami np. znaków diakrytycznych i akcentowanych różnych alfabetów narodowych. Nie poradzą więc sobie choćby z konwersją niemieckiej litery ß, która powinna w zbiorze liter wielkich zostać zapisana parą `SS`.

9.9.3. Wyszukiwanie w ciągu

Funkcja `index(`*`ciąg`*`, `*`szukany`*`)` przeszukuje ciąg *ciąg* w poszukiwaniu występującego w nim podciągu *szukany*. Wartością zwracaną funkcji jest pozycja pierwszego znaku znalezionego podciągu, ewentualnie wartość 0, jeśli nie udało się go znaleźć. Na przykład wywołanie `index("abcdef" "de")` zwraca wartość 4.

Zgodnie z treścią punktu 9.9.2 wyszukiwanie pozycji podciągu można uniezależnić od wielkości liter w obu podciągach, stosując wywołanie postaci `index(tolower(`*`ciag`*`), tolower(`*`szukany`*`))`. Jeśli w danym programie ignorowanie wielkości liter jest wymagane szerzej, `gawk` pozwala na ustawienie specjalnej zmiennej wbudowanej `IGNORECASE`. Przypisanie do niej wartości niezerowej wymusza ignorowanie wielkości liter we wszystkich wywołaniach funkcji porównujących, przeszukujących i dopasowujących ciągi.

Funkcja `index()` zadowala się zwróceniem pierwszego wystąpienia szukanego podciągu. A co zrobić, jeśli interesuje nas inne, na przykład ostatnie wystąpienie? Z braku realizującej takie zadanie funkcji wbudowanej możemy posłużyć się własną, bardzo prostą funkcją, prezentowaną na listingu 9.5.

Listing 9.5. Funkcja przeszukująca ciąg wstecz

```
function rindex(string, find,          k, ns, nf)
{
    # zwraca indeks ostatniego wystąpienia podciągu find w ciągu string,
    # albo 0, jeśli nie uda się go znaleźć

    ns = length(string)
    nf = length(find)
    for (k = ns + 1 - nf; k >= 1; k--)
        if (substr(string, k, nf) == find)
        return k
    return 0
}
```

Pętla rozpoczyna się od wartości k zrównującej końce ciągu przeszukiwanego (string) i szukanego (find). Teraz następuje wyizolowanie z ciągu przeszukiwanego podciągu o długości równej długości ciągu szukanego, począwszy od pozycji k. Jeśli wyizolowany podciąg jest identyczny z szukanym, to k jest zwracane jako pozycja szukanego ostatniego wystąpienia find w string. Jeśli równość nie zachodzi, k jest zmniejszane i następuje wyizolowanie i porównanie następnego podciągu string z find. Pętla jest kontynuowana do momentu, w którym k zmniejszy się do zera. Kiedy to nastąpi, wiadomo już, że w ciągu przeszukiwanym w ogóle nie występuje ciąg szukany i należy zwrócić zero.

9.9.4. Dopasowywanie ciągów

Funkcja match(*ciąg*, *wyrażenie*) dopasowuje przekazany ciąg do zadanego drugim argumentem wywołania wyrażenia regularnego, zwracając indeks początku dopasowania, albo 0, kiedy ciągu nie udało się dopasować do wyrażenia. Wywołanie funkcji match() daje więcej informacji niż wyrażenie (*ciąg ~ wyrażenie*), które daje albo 1 (przy udanym dopasowaniu) albo 0 (brak dopasowania). Dodatkowo wywołanie match() wiąże się zwykle z dodatkowym efektem ubocznym: otóż funkcja ustawia zmienne globalne RSTART i RLENGTH; RSTART to indeks początku podciągu dopasowania, a RLENGTH to rozmiar dopasowanego podciągu. Można dzięki nim zaraz po wykryciu dopasowania wyizolować je z ciągu wywołaniem substr(*ciąg*, RSTART, RLENGTH).

9.9.5. Zastępowanie podciągów

Język awk udostępnia parę funkcji wbudowanych służących do zastępowania podciągów w ciągach. Mowa o sub(*wyrażenie*, *wstawiany*, *przeszukiwany*) i gsub(*wyrażenie*, *wstawiany*, *przeszukiwany*). Pierwsza z nich — sub() — próbuje dopasować w ciągu przeszukiwanym wyrażenie regularne i jeśli to się uda, zastępuje pierwsze od lewej i maksymalnie długie dopasowanie ciągiem wstawianym. Druga funkcja działa bardzo podobnie, tyle że zastępuje ciągiem wstawianym wszystkie dopasowania w ciągu przeszukiwanym (przedrostek g oznacza tu podstawienie globalne). Obie funkcje zwracają liczbę wykonanych podstawień. A w przypadku braku trzeciego argumentu wywołania obie przyjmują, że jest to ciąg bieżącego rekordu, czyli $0. Funkcje te są o tyle niezwykłe, że modyfikują przekazane do nich argumenty skalarne; takich funkcji nie da się napisać w języku awk. Co do zastosowania, to np. w aplikacji przygotowującej faktury można wywołaniem gsub(/[^-0-9.,]/, "*", kwota) w ciągu kwota zastąpić wszystkie znaki, które nie powinny występować w zapisie wartości pieniężnej, znakami gwiazdek.

W wywołaniu sub(*wyrażenie*, *wstawiany*, *przeszukiwany*) czy gsub(*wyrażenie*, *wstawiany*, *przeszukiwany*) każde wystąpienie znaku & w ciągu wstawianym jest zastępowane ciągiem dopasowanym do wyrażenia regularnego. Jeśli w ciągu wstawianym ma pojawić się literalnie interpretowany znak &, należy go zapisać jako \&. Trzeba też pamiętać o konieczności dublowania znaków lewego ukośnika w ciągach ujmowanych w znaki podwójnego cudzysłowu. Na przykład wywołanie gsub(/[aeiouyAEIOUY]/, "&&") zdubluje wszystkie samogłoski znajdujące się w bieżącym rekordzie wejściowym $0; tymczasem wywołanie gsub(/[aeiouyAEIOUY]/, "\\&\\&") spowoduje zastąpienie każdej z samogłosek ciągu $0 parą znaków &.

W gawk do dyspozycji jest jeszcze jedna odmiana funkcji podstawiającej — gensub(). Jej działanie opisuje szczegółowo dokumentacja systemowa man pod hasłem *gawk(1)*.

W zadaniu redukcji danych podstawienie sprawdza się często lepiej niż kombinowane operacje indeksowania i wyłuskiwania podciągów. Weźmy choćby problem wyodrębnienia wartości przypisania z wiersza pliku kodu:

```
composer =         "P. D. Q. Bach"
```

Dzięki funkcjom podstawień problem można rozwiązać następująco:

```
value = $0
sub(/^ *[a-z]*+ *= *"/, "", value)
sub(/" *$/, "", value)
```

Alternatywą byłaby poniższa konstrukcja:

```
start = index($0, "\"") + 1
end = start - 1 + index(substr($0, start), "\"")
value = substr($0, start, end - start)
```

Drugie rozwiązanie wymaga starannego zliczania znaków, nie pozwala na dopasowanie wzorca danych i wymaga dwukrotnego wyodrębniania podciągów.

9.9.6. Podział ciągu

Niezwykle wygodny a automatyczny podział rekordu wejściowego $0 na pola $1, $2, . . ., $NF można zrealizować również jawnym wywołaniem funkcji: split(*ciąg*, *tablica*, *wyrażenie*). Funkcja ta dzieli ciąg *ciąg* na podciągi umieszczane w kolejnych elementach tablicy *tablica*, przy czym rolę separatora sterującego podziałem pełnią ciągi dopasowywane do wyrażenia regularnego przekazywanego trzecim argumentem wywołania. W przypadku braku owego argumentu rolę tę przejmuje bieżąca wartość wbudowanej zmiennej separatora pól FS. Wartością zwracaną przez funkcję jest liczba elementów dołączonych do tablicy *tablica*. Zastosowanie funkcji split() ilustruje listing 9.6.

Listing 9.6. Program testujący funkcję split()

```
{
    print "\nSeparator pol = FS = \"" FS "\""
    n = split($0, parts)
    for (k = 1; k <= n; k++)
        print "parts[" k "] = \"" parts[k] "\""

    print "\nSeparator pol = \"[ ]\""
    n = split($0, parts, "[ ]")
    for (k = 1; k <= n; k++)
        print "parts[" k "] = \"" parts[k] "\""

    print "\nSeparator pol =", ":"
    n = split($0, parts, ":")
    for (k = 1; k <= n; k++)
        print "parts[" k "] = \"" parts[k] "\""

    print ""
}
```

Po zapisaniu programu z listingu 9.6 w pliku i wywołaniu go w trybie interaktywnym (wejście czytane z wejścia standardowego) można dokładnie przetestować działanie funkcji split():

```
$ awk -f split.awk
  Harold  i Maude

Separator pol = FS = " "
parts[1] = "Harold"
parts[2] = "i"
parts[3] = "Maude"

Separator pol = "[ ]"
parts[1] = ""
parts[2] = ""
parts[3] = "Harold"
parts[4] = ""
parts[5] = "i"
parts[6] = "Maude"

Separator pol = :
parts[1] = "  Harold  i Maude"

root:x:0:1:Wszechmocny Super Administrator:/root:/sbin/sh

Separator pol = FS = " "
parts[1] = "root:x:0:1:Wszechmocny"
parts[2] = "Super"
parts[3] = "Administrator:/root:/sbin/sh"

Separator pol = "[ ]"
parts[1] = "root:x:0:1:Wszechmocny"
parts[2] = "Super"
parts[3] = "Administrator:/root:/sbin/sh"

Separator pol = :
parts[1] = "root"
parts[2] = "x"
parts[3] = "0"
parts[4] = "1"
parts[5] = "Wszechmocny Super Administrator"
parts[6] = "/root"
parts[7] = "/bin/sh"
```

Szczególnie warta odnotowania jest różnica pomiędzy interpretacją separatora pól w jego domyślnej postaci (`" "`), który wymusza ignorowanie odstępów poprzedzających i uzupełniających właściwą zawartość pola — słowem, traktowanie ciągów sąsiadujących odstępów jako pojedynczego znaku odstępu. Dla porównania, stosowanie separatora o wartości `"[ ]"` oznacza rozpoznawanie pola nawet pomiędzy dwoma znakami spacji (albo pomiędzy początkiem wiersza a pierwszą spacją). W zastosowaniach polegających na przetwarzaniu tekstu pożądane jest zazwyczaj domyślne ustawienie separatora pól.

Przykład pierwszej próby zastosowania separatora w postaci znaku dwukropka ilustruje przypadek, kiedy to funkcja `split()` dodaje do tablicy tylko jeden element obejmujący cały ciąg źródłowy. Odbywa się to w przypadku braku znaku separatora w przekazanym ciągu; druga próba zastosowania separatora w postaci znaku dwukropka ilustruje przydatność funkcji `split()` do wyodrębnienia pól z rekordów pliku kont — */etc/passwd*.

Ostatnie implementacje `awk` udostępniają też uogólnioną wersję `split` postaci `split(ciąg, znaki, "")`, dzielącą ciąg na jednoznakowe podciągi umieszczane w kolejnych elementach tablicy: `znaki[1]`, `znaki[2]`, …, `znaki[strlen(ciąg)]`. W starszych implementacjach identyczny podział trzeba realizować poniższym, mniej efektywnym kodem:

```
n = length(ciag)
for (k = 1; k <= n; k++)
    znaki[k] = substr(ciag, k, 1)
```

Wywołanie `split("", tablica)` usuwa wszystkie elementy *tablicy*. To znacznie szybsza metoda usuwania elementów niż pętla:

```
for (klucz in tablica)
    delete tablica[klucz]
```

konieczna, kiedy dana implementacja `awk` nie obsługuje instrukcji `delete tablica`.

Funkcja `split()` jest też powszechnie stosowana w przeglądaniu tablic o wielu indeksach, jak poniżej:

```
for (trojka in adresat)
    split(trojka, elementy, SUBSEP)
    nr_domu = elementy[1]
    ulica = elementy[2]
    kod = elementy[3]
    ...
}
```

9.9.7. Rekonstrukcja ciągu

W `awk` nie istnieje wbudowana wersja funkcji o działaniu odwrotnym do `split()`, za to łatwo taką funkcję napisać samodzielnie — kod takiej funkcji prezentowany jest na listingu 9.7. Funkcja `join()` upewnia się, że przekazany w wywołaniu zakres indeksów pokrywa się z zakresem indeksów elementów tablicy. Inaczej wywołanie funkcji z zerowym rozmiarem tablicy mogłoby doprowadzić do utworzenia elementu `array[1]` i tym samym zmodyfikowania tablicy przekazanej przez wywołującego. Wstawiany do konstruowanego ciągu separator pól to zwykły ciąg znaków (nie wyrażenie regularne), co oznacza, że w ogólnym przypadku `join()` nie poradzi sobie z wierną rekonstrukcją ciągu podzielonego w `split()` na bazie wyrażenia regularnego.

Listing 9.7. Rekonstrukcja ciągu na podstawie tablicy podciągów

```
function join(a, n, fs,    k, s)
{
    # Scala elementy a[1]...a[n] do postaci ciągu;
    # w ciągu wynikowym podciągi są rozdzielane ciągiem separatora fs
    if (n >= 1)
    {
        s = a[1]
        for (k = 2; k <= n; k++)
            s = s fs a[k]
    }
    return (s)
}
```

9.9.8. Formatowanie ciągów

Ostatnie z prezentowanych tu funkcji operujących na ciągach będą funkcjami formatowania ciągów tekstowych i ciągów reprezentujących wartości liczbowe. Chodzi o funkcje `sprintf()` i `printf()`. Funkcja `sprintf(format, wyr1, wyr2, ...)` przekazuje sformatowany ciąg do wywołującego za pośrednictwem wartości zwracanej. Funkcja `printf()` działa bardzo podobnie, z tym że sformatowany ciąg wypisuje na standardowym wyjściu (albo do pliku, jeśli

zostanie poddana przekierowaniu). Współczesne języki programowania zastępują stopniowo charakterystyczne dla obu tych funkcji ciągi sterujące formatowaniem potencjalnie efektywniejszymi funkcjami formatującymi, ale odbywa się to kosztem zwartości kodu. A w typowych zastosowaniach związanych z przetwarzaniem tekstu funkcje `printf()` i `sprintf()` są zwykle aż nadto wystarczające.

Ciągi formatujące, sterujące działaniem funkcji `printf()` i `sprintf()`,bardzo przypominają pełniące analogiczną rolę ciągi formatujące polecenia powłoki `printf`, omawianego w podrozdziale 7.4. Specyfikatory formatu rozpoznawane w ciągach formatujących funkcji `awk` zostały wymienione w tabeli 9.5. Można je uzupełniać modyfikatorami szerokości pola, modyfikatorami precyzji oraz znacznikami omówionymi szczegółowo w rozdziale 7.

Tabela 9.5. Specyfikatory formatu funkcji printf i sprintf

Specyfikator	Interpretacja
`%c`	Pojedynczy znak ASCII. Specyfikator ten wymusza wypisanie pierwszego znaku odpowiadającego mu argumentu, który jest ciągiem znaków, albo znaku reprezentowanego wartością odpowiadającego mu argumentu liczbowego (po wykonaniu na nim operacji modulo 256).
`%d, %i`	Liczba całkowita (dziesiętna).
`%e`	Wartość zmiennoprzecinkowa (w formacie `[-].precyzjae[+-]dd`).
`%f`	Wartość zmiennoprzecinkowa (w formacie `[-]ddd.precyzja`).
`%g`	Wartość zmiennoprzecinkowa w formacie `%e` albo `%f` (wybierany jest format dający krótszą reprezentację znakową) pozbawiona zer uzupełniających właściwą wartość.
`%o`	Wartość liczbowa ósemkowa bez znaku.
`%s`	Ciąg znaków.
`%u`	Wartość liczbowa bez znaku. W języku `awk` wartości liczbowe są wartościami zmiennoprzecinkowymi: niewielkie ujemne wartości całkowite są więc (przez mylącą wartość bitu znaku) wypisywane jako wielkie liczby dodatnie.
`%x`	Liczba całkowita szesnastkowa bez znaku (wartości od 10 do 15 są reprezentowane literami od `a` do `f`).
`%X`	Liczba całkowita szesnastkowa bez znaku (wartości od 10 do 15 są reprezentowane literami od `A` do `F`).
`%%`	Znak `%`.

Specyfikatory: `%i`, `%u` i `%X` nie weszły do specyfikacji języka po 1987 roku, ale we współczesnych implementacjach wciąż są obsługiwane. Mimo podobieństwa specyfikatorów formatu z `awk` i języka programowania w `awk` interpretacja specyfikatora `%c` jest inna niż w powłoce. Różnica dotyczy obsługi odpowiadających specyfikatorowi argumentów liczbowych. Różnice występują też w zakresie specyfikatora `%u` i argumentów ujemnych — tu podłożem różnicy jest odmienna wewnętrzna reprezentacja liczb w `awk` i powłoce.

Większość specyfikatorów formatu nie wymaga dodatkowego komentarza. Trzeba jednak zwrócić uwagę na trudność zagadnienia *dokładnej* konwersji binarnych wartości zmiennoprzecinkowych na reprezentujące je ciągi znaków i trudności konwersji odwrotnej; rozwiązanie problemu znaleziono dopiero w 1990 roku, a wymaga ono wysokiej precyzji na etapach pośrednich konwersji. Implementacje języka `awk` w obliczu konieczności wykonania konwersji wymaganych przez `sprintf()` do bibliotek języka C, a choć jakość owych bibliotek wciąż się podnosi, do dziś można znaleźć platformy, na których brak wystarczającej dokładności w konwersjach wartości zmiennoprzecinkowych. Sytuację pogarszają różnice w sprzętowych implementacjach operacji zmiennoprzecinkowych i kolejności wykonywania instrukcji, przez co niemal każdy język programowania boryka się z drobnymi różnicami w obsłudze wartości zmiennoprzecinkowych na różnych platformach.

Jeśli w instrukcji `print` pojawi się wartość zmiennoprzecinkowa, `awk` sformatuje ją zgodnie z wartością wbudowanej zmiennej `OFMT`, która domyślnie zawiera ciąg `"%.6g"`. Wartość owej zmiennej można modyfikować wedle potrzeb.

Podobnie realizowana jest konwersja wartości zmiennoprzecinkowej na ciąg znaków przy konkatenacji ciągu z wartością liczbową. Wtedy formatowaniem steruje kolejna zmienna wbudowana — `CONVFMT`[4]. Również jej domyślną wartością jest ciąg `"%.6g"`.

Program testowy z listingu 9.8, uruchomiony w jednej z ostatnich implementacji `nawk` w systemie Sun Solaris SPARC, generuje następujące wyniki:

```
$ nawk -f ofmt.awk
[ 1] OFMT = "%.6g"       123.457
[ 2] OFMT = "%d"         123
[ 3] OFMT = "%e"         1.234568e+02
[ 4] OFMT = "%f"         123.456789
[ 5] OFMT = "%g"         123.457
[ 6] OFMT = "%25.16e"       1.2345678901234568e+02
[ 7] OFMT = "%25.16f"         123.4567890123456806
[ 8] OFMT = "%25.16g"            123.4567890123457
[ 9] OFMT = "%25d"                             123
[10] OFMT = "%.25d"      0000000000000000000000123
[11] OFMT = "%25d"                      2147483647
[12] OFMT = "%25d"                      2147483647   oczekiwane 2147483648
[13] OFMT = "%25d"                      2147483647   oczekiwane 9007199254740991
[14] OFMT = "%25.0f"              9007199254740991
```

Listing 9.8. Testowanie efektów różnych ustawień zmiennej OFMT

```
BEGIN {
    test( 1, OFMT,         123.4567890123456789)
    test( 2, "%d",         123.4567890123456789)
    test( 3, "%e",         123.4567890123456789)
    test( 4, "%f",         123.4567890123456789)
    test( 5, "%g",         123.4567890123456789)
    test( 6, "%25.16e",    123.4567890123456789)
    test( 7, "%25.16f",    123.4567890123456789)
    test( 8, "%25.16g",    123.4567890123456789)
    test( 9, "%25d",       123.4567890123456789)
    test(10, "%.25d",      123.4567890123456789)
    test(11, "%25d",       2^31 - 1)
    test(12, "%25d",       2^31)
    test(13, "%25d",       2^52 + (2^52 - 1))
    test(14, "%25.0f",       2^52 + (2^52 - 1))
}

function test(n, fmt, value,      save_fmt)
{
    save_fmt = OFMT
    OFMT = fmt
    printf("[%2d] OFMT = \"%s\"\t", n, OFMT)
    print value
    OFMT = save_fmt
}
```

[4] Pierwotnie zmienna `OFMT` sterowała zarówno konwersją na wyjściu (w instrukcji `print`), jak i konwersją wymuszoną konkatenacją. Rozróżnienie pomiędzy tymi operacjami wprowadził standard POSIX. W większości implementacji dostępne są obie zmienne, ale na przykład w `/usr/bin/nawk` z systemu Solaris firmy Sun brakuje zmiennej `CONVFTM` — *przyp. autora.*

Jak widać, mimo 53-bitowej precyzji wartości zmiennoprzecinkowych nawk na tej platformie przy konwersji do wartości całkowitych (%d) stosuje ograniczenie do 32 bitów. Ta sama implementacja nawk uruchamiana na nieco innych platformach da odrobinę inne wyniki. Kod źródłowy programu testowego prezentuje listing 9.8.

Testy ujawniają, że wyjście generowane przez program jest dość wrażliwe na różnice w implementacjach awk, a nawet potrafi się zmieniać pomiędzy różnymi kolejnymi wersjami tej samej implementacji. Na przykład gawk daje takie wyniki:

```
$ gawk -f ofmt.awk
...
[11] OFMT = "%25d"        2147483647                    oczekiwane wyrównanie do prawej
...
[13] OFMT = "%25d"                          9.0072e+15  oczekiwane 9007199254740991
```

Nieformalna definicja języka awk z roku 1987 określa domyślną wartość zmiennej OFMT, ale nie wspomina nic o efekcie jej modyfikacji. Być może w obliczu wykrytych różnic w implementacji w standardzie POSIX stwierdza się, że jeśli zmienna OFMT nie zawiera specyfikatora formatu zmiennoprzecinkowego, to jej wpływ na formatowanie wyjścia jest nieokreślony i zależny od implementacji. Skoro tak, zachowanie gawk w tym zakresie można uznać za dopuszczalne.

W mawk mamy z kolei:

```
$ mawk -f ofmt.awk
...
[ 2] OFMT = "%d"          1079958844                    oczekiwane 123
...
[ 9] OFMT = "%25d"                          1079958844  oczekiwane 123
[10] OFMT = "%.25d"       0000000000000001079958844     oczekiwane 00...00123
[11] OFMT = "%25d"        2147483647                    oczekiwane wyrównanie do prawej
[12] OFMT = "%25d"                          1105199104  oczekiwane 2147483648
[13] OFMT = "%25d"                          1128267775  oczekiwane 9007199254740991
...
```

Zdaje się, że w zakresie obsługi wyprowadzania wielkich wartości liczbowych w miejsce specyfikatora %d (tudzież, jak dowiodły osobne testy, %i) wciąż nie doszło do konsensusu. Na szczęście wszystkie różnice można zatrzeć, stosując format wyjściowy %.0f.

9.10. Funkcje matematyczne

W języku awk przewidziane zostały elementarne funkcje matematyczne, wymienione w tabeli 9.6. Większość tych funkcji działa identycznie, jak ich odpowiedniki w innych językach programowania. Co do dokładności zwracanych wartości, to jest ona w dużej mierze zależna od jakości bibliotek matematycznych, do których odwołuje się dana implementacja awk.

Obszarem największych różnic implementacyjnych w zakresie funkcji numerycznych (matematycznych) są obie funkcje odwołujące się do generatora liczb pseudolosowych: rand() i srand(). Niektóre z nich są implementowane odwołaniami do funkcji systemowych, a implementacje generatorów pseudolosowych i ich dokładność różnią się pomiędzy platformami. Większość algorytmów generowania takich liczb polega na konstruowaniu szeregu liczb ze skończonego zbioru wartości bez ich powtarzania aż do momentu, w którym zbiór zostanie wyczerpany i sekwencja zacznie się na nowo. Długość sekwencji to inaczej interwał generatora pseudolosowego. Dokumentacja milczy zwykle o tym, czy do zakresu wartości zwracanych przez rand() wchodzą wartości skrajne zbioru — 0,0 i 1,0.

Tabela 9.6. Podstawowe funkcje matematyczne awk

Funkcja	Działanie
`atan2(y, x)`	Zwraca arcus tangens z *y*/*x* (w radianach) jako wartość z przedziału od $-\pi$ do $+\pi$.
`cos(x)`	Zwraca cosinus z *x* (w radianach), jako wartość z przedziału od –1 do +1.
`exp(x)`	Zwraca potęgę naturalną o wykładniku *x* (e^x).
`int(x)`	Zwraca całkowitą część argumentu *x* (z obcięciem części ułamkowej).
`log(x)`	Zwraca logarytm naturalny z *x*.
`rand()`	Zwraca liczbę pseudolosową *r* taką, że 0 <= *r* <= 1. Rozkład wartości *r* jest równomierny w całym przedziale.
`sin(x)`	Zwraca sinus z *X* (w radianach), jako wartość z przedziału od –1 do +1.
`sqrt(x)`	Zwraca pierwiastek kwadratowy z *x*.
`srand(x)`	Ustawia zarodek generatora liczb pseudolosowych na *x* i zwraca poprzednią wartość zarodka. Jeśli w wywołaniu zabraknie argumentu, w roli zarodka zostanie wykorzystany bieżący czas systemowy wyrażony w sekundach liczonych od dnia rozpoczynającego „erę Uniksa" (ang. *epoch*). Jeśli w programie nie zostanie wywołana funkcja `srand()`, `awk` każdorazowo przyjmuje domyślną wartość zarodka pseudolosowego.

Niejednoznaczność co do zawierania w generowanym szeregu skrajnych wartości zakresu znacznie utrudnia programowanie. Załóżmy, że zamierzamy wygenerować sekwencję pseudolosowych wartości całkowitych z zakresu od 0 do 100. Pójście po linii najmniejszego oporu i skorzystanie z wywołania `int(rand()*100)` może nigdy nie zwrócić wartości 100, jeśli `rand()` nie będzie uwzględniał w algorytmie losowania skrajnych wartości zakresu. A jeśli nawet będzie uwzględniał, to wartość 100 wystąpi zapewne znacznie rzadziej niż pozostałe wartości zbioru, bo wartość skrajna będzie generowana tylko raz w interwale generatora, kiedy to generator zwróci dokładną wartość 1,0. Zmiana mnożnika z 100 na 101 również nie pomoże, bo w niektórych systemach losowany szereg zasili wartość 101.

Pewnym rozwiązaniem problemu generowania losowych sekwencji liczb całkowitych jest funkcja `irand()`, prezentowana na listingu 9.9. Funkcja ta wymusza całkowite granice zbioru i potem, jeśli zadany zakres okaże się pusty albo niepoprawny, zwraca jedną z granic. W innych przypadkach funkcja losuje wartość całkowitą, która może być o jeden większa od rozmiaru zakresu, dodaje ją do granicy dolnej (`low`) i następnie *ponawia próbę*, jeśli wynik będzie wartością spoza zakresu. W takim algorytmie jest bez znaczenia, czy `rand()` kiedykolwiek zwróci 1,0, a wartości zwracane z `irand()` zachowają — charakterystyczny dla funkcji `rand()` — równomierny rozkład w całym zakresie wartości.

Listing 9.9. Generowanie pseudolosowych liczb całkowitych z ograniczonego zakresu

```
function irand(low, high,     n)
{
    # Zwraca pseudolosową liczbę całkowitą n taką, że low <= n <= high

    # Wymuszenie całkowitych wartości granic zakresu
    low = int(low)
    high = int(high)

    # Kontrola kolejności argumentów
    if (low >= high)
        return (low)

    # Wyszukanie wartości z zadanego zakresu
    do
```

```
        n = low + int(rand() * (high + 1 - low))
    while ((n < low) || (high < n))

    return (n)
}
```

Pod nieobecność wywołania `srand(x)` implementacje `gawk` i `nawk` stosują zawsze taki sam zarodek, dzięki czemu w kolejnych uruchomieniach programów korzystających z generatora liczb pseudolosowych można uzyskać powtarzalność rezultatów. Inicjowanie generatora liczb pseudolosowych bieżącym czasem celem zróżnicowania wartości wykorzystywanych w kolejnych uruchomieniach programu jest zasadne, pod warunkiem uwzględnienia precyzji zegara systemowego. Niestety, choć szybkość komputerów wciąż gwałtownie rośnie, większość odczytów bieżącego czasu systemowego wciąż ogranicza dokładność do pojedynczych sekund. Jest więc całkiem prawdopodobne, że kilka kolejnych wsadowo uruchamianych programów albo kilka przebiegów pętli symulacyjnej w obrębie jednego programu — pomimo wywołania `srand()` — otrzyma identyczne sekwencje pseudolosowe. Rozwiązaniem jest unikanie wywoływania funkcji `srand()` więcej niż raz w każdym uruchomieniu programu albo wymuszenie przynajmniej jednosekundowego odstępu pomiędzy kolejnymi wywołaniami:

```
$ for k in 1 2 3 4 5
> do
>     awk 'BEGIN {
>                     srand()
>                     for (k = 1; k <= 5; k++)
>                         printf(".5f ", rand())
>                     print ""
>                 }'
>     sleep 1
> done
0.29994 0.00751 0.57271 0.26084 0.76031
0.81381 0.52809 0.57656 0.12040 0.60115
0.32768 0.04868 0.58040 0.98001 0.44200
0.84155 0.56929 0.58422 0.83956 0.28288
0.35539 0.08985 0.58806 0.69915 0.12372
```

Przy braku polecenia `sleep 1` mogłoby dojść do wypisania pięciu identycznych sekwencji.

9.11. Podsumowanie

Zaprezentowane instrukcje i funkcje wbudowane `awk` okazują się wystarczające do implementacji rozwiązań zaskakująco szerokiego zbioru problemów związanych z przetwarzaniem tekstu. Po zrozumieniu sposobu konstruowania wiersza wywołania interpretera `awk` i przyzwyczajeniu się do automatycznej obsługi plików wejściowych programista może skupić się na określaniu akcji przetwarzających wybrane grupy rekordów. Tego rodzaju minimalistyczny, *ukierunkowany na dane* (ang. *data-driven*) model programistyczny okazuje się niezwykle efektywny. Dla porównania, w większości tradycyjnych języków programowania znaczna część kodu obsługuje przeglądanie listy plików wejściowych, otwieranie i zamykanie plików, wczytywanie danych z plików, zamykanie plików i tak dalej. Właściwe zadania programu, czyli rozpoznawanie, dopasowywanie i wreszcie przetwarzanie rekordów, schodzą jakby na dalszy plan.

Kto przekonał się, jak prosto i wygodnie przetwarza się rekordy i pola w języku `awk`, zmienia zasadniczo postrzeganie problemów przetwarzania danych. Nowe podejście umożliwia podział większych problemów na mniejsze zadania. Na przykład do przetwarzania złożonych

plików binarnych, takich jak pliki baz danych, pliki fontów, pliki graficzne, pliki arkuszy kalkulacyjnych i edytorów tekstów, warto rozejrzeć się za narzędziami konwertującymi owe formaty binarne na łatwe do przetworzenia, odpowiednio oznaczone formaty tekstowe. Tak przygotowane dane można następnie wygodnie i efektywnie przetwarzać prostymi filtrami pisanymi w języku awk i innych językach skryptowych.