helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Przetwarzanie danych dla programistów



Autor: Greg Wilson
Tłumaczenie: Marek Pętlicki
ISBN: 83-246-0407-3
Tytuł oryginału: Data Crunching
Format: B5, stron: 264

Przetwarzanie danych to czynność, którą programiści na całym świecie wykonują niemal codziennie. Konwersja danych pomiędzy systemami informatycznymi, zmiana formatów plików tekstowych, przeszukiwanie logów serwerów WWW – to wszystko można nazwać przetwarzaniem danych. Znajomość technik, dzięki którym takie procesy odbywają się szybko i efektywnie, to jedna z najważniejszych umiejętności programisty. Trudnością może okazać się fakt, że nie istnieje brak jednego, uniwersalnego sposobu przetwarzania danych. Do każdego typu problemu należy podejść w sposób indywidualny, próbując rozłożyć go na sekwencję prostych przekształceń, łatwych do implementacji i testowania.

Czytając książkę „Przetwarzanie danych dla programistów", poznasz metody rozwiązywania problemów programistycznych związanych z konwersją danych różnego typu. Nauczysz się analizować istotę zagadnienia i dobierać najbardziej optymalny sposób realizacji zadania. Dowiesz się, jak w systemach Unix/Linux wykorzystać wyrażenia regularne i powłoki tekstowe systemów z rodziny Unix/Linux do przetwarzania danych tekstowych., Pprzeczytasz o użytecznych, lecz często niedocenianych cechach języków Java i Python oraz innych języków programowania. Przekonasz się, że mimo rozbieżnościróżnic pomiędzy różnymi typami danych istnieje kilka ogólnych wzorców, które powtarzają się w wielu zastosowaniach niezależnie od użytegozastosowanego języka programowania lub detali implementacyjnych.

- Przetwarzanie danych tekstowych za pomocą powłoki Uniksa
- Stosowanie wyrażeń regularnych
- Analiza dokumentów XML
- Pakowanie i rozpakowywanie danych binarnych
- Zapytania w relacyjnych bazach danych
- Testowanie mechanizmów konwersji danych

Opanuj jedną z podstawowych umiejętności profesjonalnego programisty

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

Spis treści

Rozdział 1. Wprowadzenie 5
1.1. Narysować molekułę 5
1.2. Czarna owca 7
1.3. Morał 8
1.4. Pytania o przetwarzanie danych 9
1.5. Plan książki 13

Rozdział 2. Tekst 17
2.1. Odwracanie kolejności wierszy w pliku 17
2.2. Przeformatowanie danych 20
2.3. Obsługa rekordów wielowierszowych 29
2.4. Testowanie kolizji 36
2.5. Włączanie plików zewnętrznych 42
2.6. Powłoka Uniksa 47
2.7. Bardzo duże zbiory danych 56
2.8. Podsumowanie 57

Rozdział 3. Wyrażenia regularne 59
3.1. Powłoka 61
3.2. Podstawy wzorców dopasowań 63
3.3. Wydobywanie dopasowanych wartości 72

3.4. Zastosowania praktyczne 83
3.5. Różne języki 95
3.6. Inne systemy 99
3.7. Podsumowanie 104

Rozdział 4. XML 105
4.1. Szybkie wprowadzenie 106
4.2. SAX 112
4.3. DOM 126
4.4. XPath 137
4.5. XSLT 143
4.6. Podsumowanie 153

Rozdział 5. Dane binarne 157
5.1. Liczby 158
5.2. Wejście i wyjście 165
5.3. Ciągi znaków 171
5.4. Podsumowanie 182

Rozdział 6. Relacyjne bazy danych 185
6.1. Proste zapytania 186
6.2. Zagnieżdżanie i negacja 197
6.3. Agregacje i perspektywy 203
6.4. Tworzenie, modyfikacja i usuwanie 207
6.5. Zastosowanie SQL-a w programach 216
6.6. Podsumowanie 220

Rozdział 7. Diabeł tkwi w szczegółach 223
7.1. Testy jednostkowe 223
7.2. Kodowanie i dekodowanie 235
7.3. Arytmetyka zmiennoprzecinkowa 239
7.4. Daty i czas 242
7.5. Podsumowanie 248

Dodatek A Bibliografia 249

Skorowidz 251

Rozdział 2.
Tekst

TEKST JEST JEDNYM Z NAJSTARSZYCH FORMATÓW danych i nadal należy do najbardziej popularnych. Jednym z powodów popularności takiego formatu jest fakt, że dane w nim zapisane można przetwarzać i przeglądać za pomocą dowolnego edytora tekstów. Głębszy powód tej popularności leży w tym, że pisanie programów do automatycznej manipulacji tekstem jest bardzo łatwe. W tym rozdziale przyjrzymy się programom tego typu, ich możliwościom i podstawowej strukturze. Zatrzymamy się też chwilę przy wierszu poleceń systemów Unix, środowisku zorientowanemu tekstowo, które nadal trzyma się bardzo dobrze, pomimo niemal czterdziestoletniej tradycji.

2.1. Odwracanie kolejności wierszy w pliku

Na początek weźmiemy pod uwagę zupełnie podstawowy problem przetwarzania danych tekstowych, jakim jest odwrócenie kolejności wierszy w pliku. Oto proste rozwiązanie tego problemu w Pythonie:

```
import sys
# Odczyt
input = open(sys.argv[1], "r")
lines = input.readlines()
```

```
input.close()
# Przetwarzanie
lines.reverse()
# Zapis
output = open(sys.argv[2], "w")
for line in lines:
    print >> output, line.strip()
output.close()
```

Kod jest trywialny, posiada jednak strukturę charakterystyczną dla większości procedur przetwarzających dane:

1. odczyt danych wejściowych,
2. przetwarzanie,
3. zapis wyniku.

W wielu przypadkach możliwe jest zapisywanie wyniku na bieżąco w ramach przetwarzania, lecz prawie zawsze najlepiej jest podzielić problem na powyższe etapy. Po pierwsze, dzięki temu kod staje się czytelniejszy i łatwiej go wykorzystać ponownie (problemy tego typu bywają bardzo powtarzalne). Po drugie, to podejście jest niezwykle uniwersalne i działa prawie zawsze[1], natomiast odmiana polegająca na połączeniu etapu drugiego z trzecim nie zawsze zadziała (przykładem jest właśnie odwracanie kolejności wierszy pliku).

Zwolennicy mniej dynamicznych języków programowania mogą preferować następującą wersję napisaną w Javie:

```
import java.util.*;
import java.io.*;
public class ReverseLines {
  public static void main(String[] args) {
    try {
       // Odczyt
      BufferedReader input = new BufferedReader(new
         FileReader(args[0]));
      ArrayList list = new ArrayList();
      String line;
      while ((line = input.readLine()) != null) {
        list.add(line);
      }
```

[1] Chyba że danych jest zbyt wiele, by mogły zmieścić się w pamięci. Do tego tematu wrócimy w dalszej części rozdziału.

```
            input.close();
             // Przetwarzanie
            Collections.reverse(list);
             // Zapis
            PrintWriter output =
              new PrintWriter(new BufferedWriter(new FileWriter(args[1])));
            for (Iterator i=list.iterator(); i.hasNext(); ) {
              output.println((String)i.next());
            }
            output.close();
          }
          catch (IOException e) {
          System.err.println(e);
        }
    }
```

Nietrudno zauważyć, że ten kod jest dłuższy od jego odpowiednika w Pythonie, lecz obydwa realizują dokładnie tę samą funkcję w niemalże identyczny sposób.

Oczywiście bezpośrednie przełożenie kodu z jednego języka na drugi nie zawsze daje najbardziej naturalne lub optymalne rozwiązanie. Prawdziwi programiści Javy z pewnością preferują umieszczenie odczytu, przetwarzania i zapisu danych w osobnych metodach klasy, zechcą też pominąć wywołanie metody `reverse()`, a raczej zechcą odczytywać elementy pojedynczo, jednak w odwróconej kolejności:

```
PrintWriter output =
    new PrintWriter(new BufferedWriter(new FileWriter(args[1])));
for (ListIterator i=list.listIterator(list.size()); i.hasPrevious(); )
{
    output.println((String)i .previous());
}
output.close();
```

A co z obsługą błędów? W jaki sposób ma zachować się program, jeśli plik wejściowy nie istnieje lub plik wyjściowy nie może być utworzony? Co zrobić, jeśli plik wynikowy już istnieje: czy należy go nadpisać, zadać pytanie użytkownikowi („Zamazać wynik całego roku pracy [t]ak, [n]ie?"), czy wypisać komunikat o błędzie i zatrzymać działanie? Co z obsługą wielu plików jednocześnie lub z łączeniem plików w jedną całość? Co z głodem na świecie i efektem cieplarnianym?

To wszystko są z pewnością bardzo ważne pytania, lecz dotyczą ergonomii aplikacji, nie przetwarzania danych. Jeśli jednak istnieje prawdopodobieństwo, że pisany program będzie wykorzystywany przez nieznane nam osoby

w czasie naszej nieobecności, warto poświęcić im przynajmniej kilka minut uwagi. A jeśli mamy za zadanie wykonanie jednorazowej operacji przeformatowania pliku w starym formacie w taki sposób, aby mógł być wczytany przez nowy program, nie warto zaprzątać sobie głowy takimi detalami.

2.2. Przeformatowanie danych

Wróćmy do problemu zmiany formatu zapisu definicji cząsteczek chemicznych na potrzeby wizualizacji 3D. Każda definicja cząsteczki jest zapisana w pliku PDB o następującej postaci:

```
COMPND      Ammonia
AUTHOR      DAVE WOODCOCK    97  10  31
ATOM      1   N           1           0.257   -0.363   0.000
ATOM      2   H           1           0.257    0.727   0.000
ATOM      3   H           1           0.771   -0.727   0.890
ATOM      4   H           1           0.771   -0.727  -0.890
TER       5               1
END
```

W pierwszym wierszu definicji cząsteczki zapisana jest nazwa potoczna związku. Drugi wiersz zawiera nazwisko autora pliku i datę jego utworzenia. Każdy wiersz rozpoczynający się od słowa kluczowego `ATOM` definiuje typ i położenie pojedynczego atomu. Nie wiadomo dokładnie, co oznaczają wiersze rozpoczynające się od słowa kluczowego `TER` ani do czego służy liczba `1` w czwartej kolumnie każdego wiersza `ATOM`, lecz w naszym przypadku nie ma to znaczenia.

W pierwszym odruchu chciałoby się zacząć kodowanie od razu, lecz doświadczenie[2] uczy nas, że warto chwilę zastanowić się i rozważyć pewne założenia dotyczące formatu wejściowego. Na przykład: czy format PDF może zawierać puste wiersze? Czy wiersze `COMPND` i `AUTHOR` występują zawsze i to w takiej samej kolejności?

Swoim początkującym studentom często powtarzam, że *godzina ciężkiej pracy pozwoli zaoszczędzić sześćdziesiąt sekund szukania za pomocą Google*. Wyszukiwanie hasła `PDB format` zwraca kilkadziesiąt adresów, w tym jeden

[2] Doświadczenie to nazwa, jaką nadajemy naszym błędom — Oscar Wilde.

do oficjalnej specyfikacji[3] napisanej pseudoprawniczym żargonem, który programiści chętnie stosują w przypadkach, gdy podejrzewają, że wśród odbiorców może znajdować się jakiś prawnik. Po chwili stwierdzamy, że pliki PDB mogą zawierać kilkadziesiąt różnych typów rekordów zorganizowanych w sekcje. Nam jednak potrzebne są tylko współrzędne atomów, zatem na razie możemy z powodzeniem zignorować wszelkie nierozpoznawalne przez nas dane. Zobaczymy, co z tego wyniknie.



Jaś pyta...

Czy tak wolno?

Ignorowanie nierozpoznanych danych kojarzy się dość niebezpiecznie. Co się stanie, jeśli atomy zostaną zdefiniowane w wierszach nierozpoczynających się słowem kluczowym `ATOM`? Co się stanie, jeśli ignorowane rekordy zmodyfikują znaczenie rekordów `ATOM`? Czy nie należy przestudiować specyfikacji z zakreślaczem w ręku, a program napisać po upewnieniu się co do słuszności decyzji?

W tym przypadku odpowiedź brzmi „nie". Czytanie specyfikacji formatu PDB zajęłoby więcej czasu, niż ręczne przekształcenie formatów. Oczywiście dogłębna znajomość specyfikacji *może* przyczynić się do uniknięcia błędów, lecz koszt takich błędów jest bliski zeru. Tego typu decyzja oczywiście byłaby nieodpowiedzialna, jeśli mielibyśmy do czynienia z formatem danych kartoteki pacjentów szpitala lub dokumentacji silników odrzutowych samolotów pasażerskich, lecz w tym przypadku jest to podejście jak najbardziej praktyczne. Zadanie realizujemy w jak najprostszy sposób i poprawiamy później, jeśli wystąpią błędy. W tym przypadku jednak kluczowe jest, aby dokładnie znać format wyjściowy, aby natychmiast zauważyć wszelkie błędy.

Przyjrzyjmy się zatem formatowi wyjściowemu:

```
-- atom azotu
1 17 0.5 0.257 -0.363 0.000
```

[3] *http://www.rcsb.org/pdb/docs/format/pdbguide2.2/guide2.2_frame.html*

```
-- atomy wodoru
1 6 0.2 0.257 0.727 0.000
1 6 0.2 0.771 -0.727 0.890
1 6 0.2 0.771 -0.727 -0.890
```

Wiersze rozpoczynające się od dwóch myślników to oczywiście komentarz, można zatem założyć, że możemy w ich charakterze wykorzystać dowolne informacje lub zupełnie je pominąć. Pierwsza kolumna o wartości `1` oznacza kulę, następnie występuje kod koloru, średnica atomu oraz współrzędne XYZ.

Na początek wiadomo, że do tłumaczenia symboli atomów (w przykładzie `N` i `H`) z pliku PDB na pierwsze trzy kolumny formatu VU3 potrzebna będzie tablica przeglądowa. Na razie jednak odłóżmy ten problem na bok i skupmy się na przekształceniu współrzędnych XYZ z pliku PDB. Ogólny schemat działania algorytmu będzie następujący:

```
dla każdego pliku PDB:
    odczyt atomów z pliku
    ustalenie nazwy pliku wynikowego
    zapis danych atomów i innych danych w pliku wynikowym
```

W Pythonie zakodujemy to następująco:

```
import sys
for inputName in sys.argv[1:]:
    atoms = readPdb(inputName)
    outputName = translateName(inputName)
    writeVu3(outputName, atoms)
```

Odczyt atomów z pliku PDB to dość proste zadanie: wystarczy wyłowić wiersze rozpoczynające się słowem kluczowym *ATOM* i podzielić je na indywidualne pola. Trzecie pole informuje o typie atomu; piąte, szóste i siódme zawiera współrzędne XYZ. Listy Pythona są indeksowane od zera (podobnie, jak tablice w C i Javie), więc musimy odczytać pola o indeksach 2 i 4 odpowiednio dla pól 3. i 5.

```
def readPdb(inputName):
    input = open(inputName, 'r')
    result = []
    for line in input:
        if line[:4] == 'ATOM':
            fields = line.split()
            atom = fields[2] + fields[4:7]
result.append(atom)
input.close()
return result
```

Wyrażenie `line.split()` dokonuje podziału wiersza na pola. Jako separator pól możemy podać dowolny ciąg znaków, na przykład w celu wykorzystania średnika jako separatora należy zastosować wyrażenie `line.split(';')`. Domyślnie jako separator stosowany jest dowolny ciąg białych znaków, co dokładnie odpowiada naszym potrzebom w tym przypadku.

Nie zadajemy sobie trudu sprawdzania poprawności składowej danych wejściowych. Jeśli rekord `ATOM` będzie zawierał mniejszą od oczekiwanej liczbę pól, wyrażenie `fields[2] + fields[4:7]` wywoła wyjątek przekroczenia zakresu. Nie jest to wielki problem, ponieważ w tej postaci kodu spowoduje to wypisanie komunikatu, stosu wywołań oraz zatrzymanie działania programu. W przypadku produktu przeznaczonego dla użytkowników końcowych takie zachowanie programu raczej nie byłoby wskazane, lecz w przypadku prostego narzędzia do przetwarzania danych jest jak najbardziej do przyjęcia.

Zanim przejdziemy dalej, zastanówmy się, co ta prosta funkcja ma rzeczywiście „do zrobienia". Jednym ze sposobów sprawdzenia tego jest uruchomienie kodu w debugerze i śledzenie zawartości listy `atoms` zwracanej przez funkcję `readPdb()`. Inny sposób polega na zmodyfikowaniu głównej pętli w taki sposób, aby na bieżąco wypisywała na ekranie listę atomów:

```
for inputName in sys.argv[1:]:
    atoms = readPdb(inputName)
    for a in atoms:
        print a
    outputName = translateName(inputName)
    writeVu3(outputName, atoms)
```

Uruchommy ten kod z plikiem *ammonia.pdb*:

```
Traceback (most recent call last):
  File "pdb2vu3.py", line 15, in ?
atoms = readPdb(inputName) File "pdb2vu3.py", line 7, in readPdb
  atom = fields[2] + fields[4:7]
TypeError: cannot concatenate 'str' and 'list' objects
```

Błąd: wyrażenie `fields[2]` wybiera z listy pojedynczy ciąg znaków, natomiast `fields[4:7]` wybiera podlistę, a Python nie wie, co w rzeczywistości chcemy zrobić, „dodając" do siebie ciąg znaków i listę. W każdym języku programowania występują tego typu „przypadki specjalne", gdy różne użyteczne koncepcje nie pasują do siebie wzajemnie. Jednym z powodów,

dla których języki Python, Ruby czy Java zdobywają stopniowo przewagę nad językami Perl czy C++ jest to, że w tych pierwszych, dzięki jasno zdefiniowanym regułom, ilość tego typu niejednoznaczności jest znacznie zmniejszona. Sytuację można naprawić, na przykład opakowując ciąg znaków w listę jednoelementową ([]):

```
def readPdb(inputName):
    input = open(inputName, 'r')
    result = []
    for line in input:
        if line[:4] == 'ATOM':
            fields = line.split()
            atom = [fields[2]] + fields[4:7]
            result.append(atom)
        input.close()
        return result
```

Tym razem wynik będzie następujący:

```
['N', '0.257', '-0.363', '0.000']
['H', '0.257', '0.727', '0.000']
['H', '0.771', '-0.727', '0.890']
['H', '0.771', '-0.727', '-0.890']
```

Ten przykład nie miał na celu pokazać skutku tego banalnego (aczkolwiek dość często popełnianego) błędu; chodziło mi o to, aby pokazać konieczność testowania napisanego kodu przed przejściem do kolejnego etapu prac. Podział pracy na etapy wczytywania danych, przetwarzania i zapisu w przetwarzaniu danych jest przydatny między innymi właśnie do tego typu przyrostowego testowania.

Następny etap polega na określeniu nazwy pliku wynikowego. W tym przypadku reguła jest prosta: należy zastąpić rozszerzenie *.pdb* rozszerzeniem *.vu3*:

```
def translateName(inputName):
    return inputName[ :-4] + '.vu3'
```

Już słyszę okrzyki purystów Pythona: właściwy sposób wydobycia rozszerzenia z nazwy pliku polega na zastosowaniu funkcji `os.path.splitext()`. Jednakże ten przykładowy kod piszę dla własnego użytku, jestem pewny, że użyję go tylko kilka razy w życiu i gdy odkryję, że wystąpił problem z danymi, mogę go usunąć i uruchomić skrypt jeszcze raz, a przede wszystkim nie chcę odstraszać mniej doświadczonych Czytelników, oszałamiając

ich nadmierną liczbą bibliotek, których nigdy wcześniej nie spotkali. Dlatego zdecydowałem, że w tym konkretnym przypadku zadanie nieco uproszczę.

Może się wydać, że nareszcie nadszedł czas, aby zapisać plik VU3. Jednakże jeszcze nie wiadomo, w jaki sposób przekształcić symbole atomowe formatu PDB na kod kształtu, koloru i średnicy, typowe dla formatu VU3. Najprostszy sposób polega po prostu na zakodowaniu w programie tablicy przeglądowej. Jeśli program napotka symbol atomu, którego nie znajdzie w tej tablicy, jego działanie zakończy się błędem, możemy jednak szybko uzupełnić tablicę o brakujące wpisy i ponownie uruchomić program. Taka decyzja jest do przyjęcia. Mamy więc:

```
Lookup = {
    'H' : (1, 6, 0.2),
    'N' : (1, 17, 0.5)
}
def writeVu3(outputName, atoms):
    output = open(outputName, 'w')
    for (symbol, X, Y, Z) in atoms:
        if symbol not in Lookup:
        print >> sys.stderr, 'Nieznany symbol atomu "%s"' % symbol
        sys.exit(1)
    shape, color, radius = Lookup[symbol]
    print >> output, shape, color, radius, X, Y, Z
output.close()
```

Tablice przeglądowe są prawie zawsze o wiele czytelniejsze od długich serii warunków `if…then…else`. Jednym z powodów, dla których zręczne języki, do jakich zalicza się Python, są tak użyteczne, jest fakt, że dużo informacji i logiki aplikacji można w nich zapisać w postaci struktury danych, w przeciwieństwie do bardziej sztywnych języków, do których zalicza się na przykład Java.

W tym momencie możemy przekształcić pierwszy plik z formatu PDB do formatu VU3. Czy uda się przekształcić kolejny? Przejdźmy nieco dalej w kolejce plików i spróbujmy definicji cząsteczki mentolu, która składa się z trzydziestu jeden atomów. Prawie natychmiast program wyrzuci następujący komunikat:

```
Nieznany symbol atomu "C"
```

Dobrze, właśnie tego się spodziewaliśmy. Po wprowadzeniu poprawek w kodzie i uzupełnieniu tablicy przeglądowej o definicje atomów węgla i tlenu (które również występują w mentolu) nasz program bez dalszych problemów wygeneruje odpowiedni plik VU3.

Najwyższy czas na ostateczny test. Po skopiowaniu wszystkich 112 plików PDB do jednego katalogu uruchamiamy następujące polecenie:

```
$ python pdb2vu3.py *.pdb
```

Potrzebujemy czterech podejść, aby uzupełnić wszystkie brakujące pierwiastki (siarka, chlor, żelazo i brom). Przy okazji okazuje się, że symbole atomów mogą być zapisywane wielkimi literami, np. *CL*, jak i za pomocą kombinacji wielkich i małych liter, np. *Cl*. Tego typu niespójności są na porządku dziennym w zadaniach przetwarzania danych, ponieważ dla człowieka nie stanowią problemu, są rozumiane intuicyjnie. Jednak program musi być odpowiednio zakodowany, aby był w stanie radzić sobie z nimi w sposób właściwy.

Mamy dwa wyjścia: albo dopisać do tablicy przeglądowej odpowiednie wpisy reprezentujące każdą z możliwych pisowni symbolu pierwiastka, albo znormalizować symbole przed odczytem z tablicy przeglądowej. Pierwszy sposób doprowadzi nas do tablicy o następującej postaci:

```
Lookup  = {
    'BR' (1, 2, 0.9),
    'Br' (1, 2, 0.9),
    'C' (1, 3, 0.5),
    'CL' (1, 8, 0.6),
    'Cl' (1, 8, 0.6),
    'FE' (1, 13, 1.1),
    'Fe' (1, 13, 1.1),
    'H' (1, 6, 0.2),
    'N' (1, 17, 0.5),
    'O' (1, 19, 0.5),
    'S' (1, 17, 0.7)
}
```

To jest dobra opcja w tej konkretnej sytuacji: liczba zduplikowanych wierszy jest niewielka, zadanie można sobie uprościć przez kopiowanie i wklejanie istniejących wierszy i modyfikację symbolu. A jeśli jakaś wersja pisowni zostanie pominięta, program się o to upomni.

DRY

Jedną z ogólnych zasad programowania określa się akronimem *DRY*: ang. *Don't Repeat Yourself* (nie powtarzaj się) [HT00]. Jeśli informacja zostanie powielona w dwóch miejscach lub ich większej liczbie, prędzej czy później zapomni się uzupełnić jedną z tych kopii, co doprowadzi do błędu trudnego do wykrycia, ponieważ programista będzie przekonany, że „przecież już to poprawił". Fakt, że programy do przetwarzania danych bardzo często z założenia służą rozwiązaniu jednorazowego problemu, nie powinien być wymówką dla niedbałego programowania. Doświadczenie uczy bowiem, że jednorazowy kod często przydaje się wielokrotnie, a złe nawyki nabyte przy okazji rozwiązań prowizorycznych pozostają na stałe i ujawniają się również przy tworzeniu bardziej odpowiedzialnego kodu.

Co się jednak stanie, gdy symbole pierwiastków będą w plikach źródłowych występowały we wszystkich możliwych kombinacjach wielkich i małych liter? Żelazo można zapisać jako `FE`, `Fe`, `fE` i `fe`. Wszystkie symbole jednoliterowe miałyby w pliku po dwa wpisy, natomiast symbole dwuliterowe aż po cztery.

Wydaje się dość mało prawdopodobne, że ktokolwiek celowo zapisał symbol żelaza jako `fE`. Jeśli jednak dane są wprowadzane ręcznie, jest to zupełnie możliwe w wyniku przypadkowego wciśnięcia klawisza *Caps Lock* lub innej omyłki[4].

Co się jednak stanie, gdy zdecydujemy się zmienić kolor lub średnicę kul reprezentujących atomy określonych pierwiastków? Jeśli w tabeli będziemy mieli cztery wpisy dotyczące żelaza, musimy pamiętać o tym, aby zmienić wszystkie cztery. Jeśli pominęlibyśmy jeden z nich, nie miałoby to większego znaczenia, lecz w przypadku tysięcy plików zawierających kilkadziesiąt różnych pierwiastków istnieje wysokie prawdopodobieństwo, że w końcowym rozrachunku mielibyśmy na różnych prezentacjach odmienne rozmiary i kolory kul reprezentujących atomy tego samego pierwiastka.

[4] I winę za takie sytuacje nie zawsze ponosi ludzkie zmęczenie. Użytkownicy „inteligentnych" edytorów tekstu niejednokrotnie padają ofiarą opcji *Poprawiaj dwa początkowe wersaliki* przekształcającej na przykład skrót `PL` na `Pl`.

To sprowadza nas do drugiej z dostępnych opcji: normalizacji symboli atomów. Jeśli zdecydujemy, że symbole muszą zawsze mieć poprawną postać (pierwsza litera wielka, druga mała) niezależnie od tego, w jaki sposób były zapisane w oryginalnym pliku, nasza tabela przeglądowa zawierać będzie po jednym wpisie dla każdego atomu. W takim przypadku warto jednak dopisać komentarz, że przetwarzane w programie nazwy atomów nie muszą odpowiadać temu, co jest odczytywane z plików. Taki komentarz to dodatkowe dziesięć sekund pracy, które mogą oszczędzić wielu minut poszukiwań przyczyny problemu w przyszłości, gdy przyjdzie nam poprawić spostrzeżony błąd w danych wyjściowych.

Pozostał jeszcze jeden wybór. Czy symbole pierwiastków powinny być normalizowane przy odczycie danych z pliku PDB, czy też lepiej wykonać dodatkowy przebieg po liście atomów wczytanych z pliku zamieniający ich symbole w tej liście? Aby każdy z etapów działania algorytmu był jak najbardziej czytelny i zwarty, lepiej jest zdecydować się na drugą opcję. W ten sposób główna pętla programu przyjmie następująca postać:

```
for inputName in sys.argv[1:]:
    atoms = readPdb(inputName)
    normalizeSymbols(atoms)
    outputName = translateName(inputName)
    writeVu3(outputName, atoms)
```

Nowa funkcja `normalizeSymbols()` będzie wyglądać tak:

```
def normalizeSymbols(atoms):
    for record in atoms:
        record[0] = record[0].capitalize()
```

Jak pamiętamy, każdy element listy `atoms` jest listą złożoną z podciągów wczytanej wcześniej definicji atomu w cząsteczce (zapisanej w pojedynczym wierszu pliku wejściowego). Jeśli wiersz miał poprawny format, nazwa atomu znalazła się w elemencie o indeksie zero. Metoda `capitalize()` obiektu tekstowego zwraca jego kopię z pierwszą literą zamienianą na wielką i pozostałymi zamienionymi na małe, na przykład ciąg `aBC` zostanie zamieniony na `Abc`.

Muszę coś wyznać. Gdy pisałem przedstawiony tutaj kod, nie przyszło mi nawet do głowy, aby duplikować wiersze w tabeli przeglądowej. Gdy się przez wiele lat spotyka podobne problemy, normalizacja danych staje się

nawykiem, podobnie jak logiczny podział operacji przetwarzania na osobne etapy. Nawyki tego typu zapewne nie w każdej sytuacji prowadzą do optymalnego kodu, lecz dzięki nim nie muszę każdego przypadku rozpatrywać w zupełnym oderwaniu od innych.

2.3. Obsługa rekordów wielowierszowych

Znamy już sposób przetwarzania plików po jednym wierszu, zdarza się jednak spotykać pliki, w których rekordy mogą zajmować większą liczbę wierszy. Aby nieco urealnić nasz kod, zajmijmy się przetwarzaniem plików *.ini*, typowych dla starszych wersji systemu Windows, w których zapisywało się ustawienia konfiguracyjne. Przekształćmy tego typu pliki *.ini* na pliki XML. Z tego typu zadaniem spotkałem się kilka lat temu, gdy firma, w której pracowałem, została zaangażowana do wsparcia w przejściu ze starego programu CAD do nowej wersji, z założeniem, że ustawienia użytkowników miały być przeniesione do tego nowego systemu.

Jak wiele nieustandaryzowanych formatów, pliki konfiguracyjne systemu Windows posiadają własną składnię, która z czasem stawała się coraz bardziej skomplikowana. Nie musimy obsługiwać wszystkich detali i dziwactw, skupimy się na następujących, podstawowych zagadnieniach:

- Plik *.ini* zawiera zero lub większą liczbę *sekcji*.
- Każda sekcja posiada *tytuł* i *treść*, treść może być pusta.
- Tytuł składa się z wiersza zawierającego tekst ujęty w nawiasy kwadratowe, np. `[laser6]` czy `[recently used]`. Żadna z sekcji nie może wystąpić w jednym pliku więcej, niż jeden raz.
- Treść zawiera zero lub większą liczbę właściwości. Każda właściwość składa się z klucza i wartości oddzielonych znakiem równości, np. `color=blue` czy `file3=C:\book\intro.pml`.
- Komentarze rozpoczynają się w dowolnym miejscu wiersza od znaku `#` i obowiązują do końca wiersza.

Typowy plik *.ini* ma następującą postać:

```
# Ustawienia instalacyjne
[Bootstrap]
Location=$SYSUSERCONFIG/sversion.ini
BaseInstallation=$ORIGIN/..
buildid=645m44(Build:8784)
InstallMode=STANDALONE&ALL_USERS
ProductPatch=                    # pusta
# Obsługa błędów
[ErrorReport]
ErrorReportPort=80
ErrorReportServer=services.caribou.org
```

Na razie nie będziemy zajmować się specjalnym znaczeniem wartości rozpoczynających się od znaku `$` i skupimy się na odczytywaniu pliku i przekształcaniu do postaci:

```
<configure>
<section title= "Bootstrap">
  <entry key="Location">$ SYSUSERCONFIG/sversion.ini</entry>
  <entry key="BaseInstallation">$ORIGIN/..</entry>
  <entry key="buildid">645m44(Build:8784)</entry>
  <entry key="InstallMode">STANDALONE&amp;ALL_USERS</entry>
  <entry key="ProductPatch"></entry>
</section>
<section title="ErrorReport">
  <entry key="ErrorReportPort">80</entry>
  <entry key="ErrorReportServer">services.caribou.org</entry>
</section>
</configure>
```

Odczyt pliku w formacie *.ini* jest trudniejszy od odczytu pliku PDB, ponieważ tam mieliśmy zagwarantowane, że każdy rekord mieścił się w jednym wierszu. W tym przypadku tak nie jest. Najprostsze podejście do tego problemu polega na odfiltrowaniu podczas odczytu wszystkich elementów pliku, które nie są poddawane przekształceniu, przekształceniu tego, co zostanie, i zapisaniu wyniku w pliku XML. Oczywiście należy również przekształcić znaki `&`, `<` i `>` odpowiednio w `&`, `<` i `>`, co jest wymagane przez format XML.

Zaczniemy od zwyczajowego szablonu program przetwarzającego:

```
for inputName in sys.argv[1:]:
    lines = readIni(inputName)
    settings = process(lines)
    outputName = translateName(inputName)
    writeXml(outputName, settings)
```

Funkcja `readIni()` nie polega na zwykłym wczytaniu wierszy tekstu z pliku *.ini*, do jej zadań należy też oczyszczenie danych z komentarzy i pustych wierszy. Najprostszy sposób realizacji tego zadania polega na odrzuceniu komentarza (o ile istnieje) i usunięciu białych znaków na początku i końcu pozostałego wiersza. Jeśli wynik takiej obróbki jest pustym ciągiem znaków, po prostu go odrzucamy.

W przeciwnym wypadku dopisujemy go na końcu listy wierszy, które będą zwrócone jako wynik tej funkcji:

```
def readIni(inputName):
    input = open(inputName, 'r')
    result = []
    for line in input:
        # Usunięcie części wiersza od znaku # do końca
        first = line.find('#')
        if first >= 0:
            line = line[:first]

        # Oczyszczenie tekstu z okalających białych znaków
        line = line.strip()

        # Jeśli nic nie zostało, pomijamy wiersz
        if not line:
            continue

        # W przeciwnym wypadku zapisujemy to, co zostało
        result.append(line)

        # Koniec pracy
    input.close()
    return result
```

To dość proste. Mamy jednak do czynienia z sytuacją potencjalnego błędu w przypadku, gdy dopuszczamy występowanie znaku `#` w wartościach atrybutów. Należy bowiem odpowiedzieć sobie na pytanie, czy w naszych plikach *.ini* dopuszczalne są wartości następującej postaci:

```
SongTitle="Love Potion #9" # ostatni odtwarzany utwór
```

Jeśli taki zapis jest poprawny, nasz algorytm zwróci błędną wartość, ponieważ odrzuca wszystkie znaki od znaku `#` do końca wiersza. Co gorsza, ten błąd wystąpi w sposób niezauważony — zamiast wywołać wyjątek, kod odrzuci w pełni poprawne dane.

Opisany problem można rozwiązać, wyszukując w wierszach znaki cudzysłowów i inne znaczniki specjalne. Zadanie znacznie uprościłyby wyrażenia regularne, które omówię w rozdziale 3., a na razie przejdę do następnego zadania naszego kodu: wygenerowania pliku XML zawierającego oczyszczone definicje z pliku *.ini*.

Krótkie przypomnienie: głównym blokiem w naszym pliku *XML* będzie `<configure>` i `</configure>`. Za każdym razem, gdy w pliku napotkamy sekcję `[nazwa]`, w pliku wynikowym musi znaleźć się zapis `<section title="nazwa">`. Oczywiście przed otwarciem nowej sekcji należy zamknąć poprzednią: `</section>`. Ponadto każdy atrybut postaci `nazwa=wartość` musi być odzwierciedlony w wyniku jako `<entry key="nazwa">wartość</entry>`.

Nie ma problemu, kod realizujący te funkcje jest następujący:

```
def process(lines):
    result = ['<configure>']
    for line in lines:
        # Początek nowej sekcji
        if line[0] == '[':
            # Zamknięcie poprzedniej sekcji
            result.append('</section>')
            # Początek nowej sekcji
            title = line[1:-1]
            result.append(' <section title="%s">' % title)
        # Wpis w bieżącej sekcji
        else:
            key, value = line.split('=', 1)
            value = escape(value)
            result.append(' <entry key="%s">%s</entry>' % (key, value))
    # Gotowe
result.append('</configure>')
return result
```

Wywołanie `line.split('=', 1)` powoduje rozłożenie zmiennej `line` na elementy z zastosowaniem znaku równości w charakterze separatora. Funkcja `escape()` zastępuje znaki specjalne dla formatu XML na odpowiadające im kody XML:

```
def escape(s):
    return s.replace('&', '&amp;').replace('<', '&lt;').replace('>',
       '&gt;').replace("'", '&apos;').replace('"', '&quot;')
```

Czy ten program jest już poprawny? Nie. Popełniłem dwa błędy i proponuję przyjrzeć się mu jeszcze raz. A jeszcze lepiej: proponuję wykonać go z następującym, testowym plikiem *.init*:

```
# Ustawienia instalacyjne
[Bootstrap]
ProductKey=Caribou CAD 1.1
Location=$SYSUSERCONFIG/sversion.ini
# Obsługa błędów
[ErrorReport]
ErrorReportPort=80
```

W wyniku funkcji `process()` powstanie następujący plik:

```
<configure>
</section>
<section title= "Bootstrap">
  <entry key="ProductKey">Caribou CAD 1.1</entry>
  <entry key="Location">$SYSUSERCONFIG/sversion.ini</entry>
</section>
<section title="ErrorReport">
  <entry key="ErrorReportPort">80</entry>
</configure>
```

W tym pliku znalazły się dwa błędy. Po pierwsze, znacznik zamykający *</section>* pojawił się przed znacznikiem otwierającym *<section title="Boot➥strap">*, ponieważ znacznik zamykający wypisujemy zawsze, nawet przed pierwszą sekcją w pliku. Po drugie, nie zamknęliśmy ostatniej sekcji w pliku, jak również nie zapisaliśmy znacznika kończącego konfigurację.

Pierwszy błąd naprawimy, wykorzystując znacznik boolowski zawierający informację o tym, czy przetwarzana sekcja jest pierwsza w pliku. Rozwiązanie drugiego błędu polega na tym, że zamykający znacznik *</section>* jest zapisywany zawsze przed zamykającym znacznikiem *</configure>*. Spowoduje to, że w przypadku próby przekształcenia pustego pliku *.ini* powstanie następujący plik wynikowy:

```
<configure>
</section>
</configure>
```

Zamykający znacznik *</section>* powinien być zapisywany wyłącznie w przypadku, gdy program przetworzył przynajmniej jedną sekcję. Do tego celu wykorzystamy kolejny znacznik boolowski. Inne rozwiązanie mogłoby polegać na zastąpieniu licznikiem wystąpień pierwszego z pomocniczych znaczników boolowskich. Jeśli licznik jest równy zeru i znajdujemy się w pętli, oznacza to, że nie należy wypisywać zamykających znaczników *</section>*. Jeśli licznik jest różny od zera i znajdujemy się poza pętlą, należy wypisać ten znacznik. Oto ostateczna postać naszej funkcji:

```
def process(lines):
    result = ['<configure>']
    count = 0
    for line in lines:
        # Początek nowej sekcji
        if line[0] == '[':
            # Zamknięcie poprzedniej sekcji
            if count > 0:
                result.append('</section>')
            # Początek nowej sekcji
            title = line[1:-1]
            result.append('<section title="%s">' % title)
            count += 1
          # Wpis w bieżącej sekcji
        else:
            key, value = line.split('=', 1)
            value = escape(value)
            result.append(' <entry key="%s">%s</entry>' % (key, value))
    # Gotowe
    if count > 0:
        result.append('</section>')
    result.append('</configure>')
    return result
```

Ostatnie brakujące dwie funkcje naszego programu są raczej oczywiste:

```
def translateName(inputName):
    return inputName[:-4] + '.xml'
def writeXml(outputName, settings):
    output = open(outputName, 'w')
    for line in settings:
        print >> output, line
    output.close()
```

Wystarczyło kilka prostych testów, aby przekonać się, że zaimplementowane funkcje realizują poprawnie swoje zadania, zatem uznałem program za ukończony.

Przykładowe błędy w funkcji `process()` są bardzo reprezentatywne dla tej klasy problemów, jako że błędy w przetwarzaniu danych pojawiają się najczęściej na początku i końcu procesu. Dlatego podczas sprawdzania poprawności kodu przetwarzającego dane, warto zastosować następujące dane testowe:

- puste dane wejściowe (o ile przetwarzany format na to zezwala);
- pojedynczy rekord;

Znaczniki ograniczające i automaty skończone

Ograniczanie bloków tekstu określonymi znacznikami początku i końca należy do bardzo powszechnych technik w informatyce. Obsługa tego typu formatów danych polega na zastosowaniu następującego algorytmu:

```
for (dla każdego rekordu wejściowego)
    if (rekord jest znacznikiem początku sekcji)
       if (nie pierwsza sekcja)
           zamknij poprzednią sekcję
       zacznij nową sekcję
    else
       utwórz rekord wyjściowy
if (utworzona jakakolwiek sekcja)
    zamknij sekcję główną
```

Jeśli przetwarzany tekst może być wczytany w całości do pamięci, można zastosować następującą uproszczoną postać:

```
for (dla każdej sekcji)
    rozpocznij nową sekcję
    for (dla każdego rekordu wejściowego w sekcji)
        utwórz rekord wyjściowy
    zakończ sekcję
```

Drugi z algorytmów upraszcza logikę przetwarzania, program napisany w ten sposób łatwiej jest zrozumieć i usuwać jego błędy, wymaga jednak zastosowania bardziej skomplikowanych struktur danych (mamy w tym przypadku do czynienia z listą list zamiast jednej, płaskiej listy). Zadania związane z przetwarzaniem danych z reguły wiążą się z koniecznością podejmowania kompromisów pomiędzy poziomem komplikacji danych a poziomem komplikacji kodu.

Dobrzy programiści wykorzystują w takich zastosowaniach technikę noszącą ogólną nazwę *automatów skończonych* (lub automatów o skończonej liczbie stanów). Automaty skończone wybiegają poza tematykę tej książki, zainteresowani znajdą bardzo czytelne wprowadzenie do tego zagadnienia w pozycji [HF04].

- dwa rekordy (to znaczy mamy tu do czynienia wyłącznie z pierwszym i ostatnim rekordem, nie ma rekordów „środkowych”);
- trzy rekordy lub ich większa liczba.

Jeśli zasady rządzące przetwarzanymi danymi są proste, dane testowe można przygotować, wykorzystując rzeczywiste dane. Jeśli przetwarzane dane są bardziej skomplikowane, najczęściej najprościej jest przygotować dane testowe samodzielnie.

2.4. Testowanie kolizji

Zadowolony z siebie przesłałem ten program do szefa. Kilka minut później otrzymałem odpowiedź: „Nie działa, zapraszam do siebie".

Zrozumienie przyczyny błędu zajęło poniżej minuty. Stary program CAD zezwalał na to, aby w plikach *.ini* te same klucze występowały wielokrotnie, na przykład:

```
[View]
WindowSize=1024,768
BackgroundColor=green7
WindowSize=1280,1024
```

Nowy program CAD zezwalał natomiast, aby każda wartość występowała w jednej sekcji tylko raz. W przypadku, gdy w pliku XML w jednej sekcji wystąpiło kilka wartości tego samego klucza, program nie uruchamiał się.

Szybkie śledztwo w kilku różnych plikach *.ini* potwierdziło, że wartości atrybutów pomiędzy sekcjami, a nawet w ramach sekcji, nie muszą być unikalne. Na przykład następująca zawartość pliku *.ini* jest zupełnie legalna:

```
[View]
WindowSize=1024,768
BackgroundColor=green7
WindowSize=1280,1024
[Print]
BackgroundColor=green5
```

To doskonałe pole do popisu dla słowników.

Słowniki

Jeśli przetwarzane dane muszą być w jakiś sposób unikalne, najczęściej okazuje się, że najprostszym rozwiązaniem jest wykorzystanie słownika. W zależności od języka programowania słowniki noszą nazwę odwzorowań

(ang. *map*, na przykład w Javie), haszy (ang. *hash*, np. Perl) albo *tablic asocjacyjnych* lub skojarzeniowych. Niezależnie od nazwy, słownik działa jak tabela dwukolumnowa, gdzie w lewej kolumnie wpisywane są klucze, a w prawej skojarzone z nimi wartości. Doskonałym przykładem słownika może być tabela cen, w której kodom produktów (kluczom) przyporządkowane są ich ceny (wartości):

```
ANW-400 179.95
ANW-407 179.95
ANW-460 209.95
```

Słowniki mają następujące, bardzo ważne cechy:

- Każdy klucz może wystąpić w słowniku tylko jeden raz. Pojedynczy słownik nie może zawierać na przykład trzech pozycji o kluczu ANW-400. Jeśli jedna pozycja w słowniku musi zawierać większą liczbę informacji, najczęściej w wartości umieszcza się listę (lub zbiór) wartości.
- Klucze nie posiadają ustalonej kolejności. Powyższa tabela przedstawia klucze w kolejności alfabetycznej, lecz fizyczna kolejność kluczy w słowniku może być odwrotna lub zupełnie dowolna. Każdy dobry podręcznik na temat struktur danych (jak [Sed97]) wyjaśni dokładnie przyczynę takiego stanu rzeczy. W tym miejscu wystarczy pamiętać, że tak właśnie jest, i unikać założenia o ustalonej kolejności kluczy.
- Wyszukiwanie w słowniku jest szybkie. Ta cecha jest największą zaletą słowników i głównym powodem ich stosowania w miejsce na przykład list par wartości. Zamiast sprawdzać wszystkie wartości klucza (co oznacza *N* porównań dla słownika o liczbie kluczy równej *N*), słownik przeszukuje klucze w czasie prawie niezależnym od rozmiaru[5]. W przypadku, gdy słownik zawiera kilkanaście elementów, różnica w stosunku do przeszukania zwykłej listy może być niezauważalna, ale może mieć zasadnicze znaczenie przy słownikach o rozmiarze milionów kluczy.

[5] Rzeczywisty czas wyszukiwania jest uzależniony od tego, jak bardzo klucze różnią się od siebie, co z kolei ma wpływ na potencjalne wystąpienie kolizji. Szczegóły można znaleźć w [Sed97].

Warto ponadto zauważyć, że klucze nie muszą być liczbami całkowitymi, jak indeksy list, mogą być na przykład ciągami znaków lub obiektami. W praktyce większość słowników wykorzystuje klucze w postaci ciągów znaków, lecz można stosować liczby całkowite, współrzędne XY, wskaźniki funkcji — wybór jest bardzo szeroki.

Załóżmy więc, że mamy listę nieunikalnych adresów e-mail i chcemy zliczyć, ile razy każdy z tych adresów powtarza się na liście. Lista adresów może pochodzić na przykład z listy dyskusyjnej, a taka funkcja pozwoli nam określić najbardziej aktywnych uczestników. Oto przykład tego typu listy:

```
see@spot.run.com
see@spot.run.com
jane@up-the-hill.org
see@spot.run.com
jane@up-the-hill.org
purple.dinosaur@bad.tv
jane@up-the-hill.org
```

A oto przykładowy program, który realizuje nasze zadanie:

```
Line  1   import sys
-
-   count = {}
-   for address in sys.stdin:
5       address = address.rstrip()
-       if address not in count:
-           count[address] = 1
-       else:
-           count[address] += 1
10
-   addresses = count.keys()
-   addresses.sort()
-   for address in addresses:
-       print address, count[address]
```

Przyjrzyjmy się temu programowi nieco bliżej. W wierszu 3. tworzony jest pusty słownik i przypisany zmiennej `count`. Pętla rozpoczynająca się w wierszu 4. odczytuje adresy e-mail po jednym, w wierszu 5. następuje oczyszczenie białych znaków (to znaczy sekwencji znaków końca wiersza odpowiednich dla różnych systemów operacyjnych, na przykład Unix i Windows). W wierszu 6. następuje sprawdzenie, czy dany adres był już odczytany. Jeśli nie, należy go dopisać do słownika z licznikiem równym 1. Jeśli pozycja występuje już w słowniku, należy zwiększyć o jeden wartość przypisanego jej licznika.

Druga część programu otrzymuje listę kluczy słownika zawierającą wszystkie odczytane adresy (wiersz 11.), sortuje je alfabetycznie (wiersz 12.), po czym wypisuje kolejno wraz z licznikiem. Program wywołuje się następująco:

```
python freq.py < email.txt
```

Przy powyższym wywołaniu program zwróci następujący wynik:

```
jane@up-the-hill.org 3
purple.dinosaur@bad.tv 1
see@spot.run.com 3
```

To samo zadanie możemy równie łatwo zrealizować w Javie (do zliczania adresów musimy wykorzystać obiekty klasy `Integer`, nie prymitywne wartości całkowite, ponieważ w Javie wartościami słowników mogą być wyłącznie obiekty):

```
import java.util.*;
import java.io.*;
class Freq {
    public static void main(String[] args) {
        BufferedReader input =
            new BufferedReader(new InputStreamReader(System.in));
        Map m = new HashMap() ;
        String line;
        try {
             while ((line = input.readLine()) != null) {
                 line = line.trim();
                 if (m.containsKey(line)) {
                     Integer tmp = (Integer)m.get(line);
                     m.put(line, new Integer(tmp.intValue() + 1));
                 } else {
                     m.put(line, new Integer(1));
                 }
             }
             input.close();
        }
        catch (IOException e) {
            System.err.println(e);
        }
        Set keySet = m.keySet();
        List keyList = new LinkedList(keySet);
        Collections.sort(keyList);
        for (Iterator i=keyList.iterator() ; i.hasNext(); )
        {
            String key = (String)i.next();
            Integer value = (Integer)m.get(key);
            System.out.println(key + " " + value);
        }
    }
}
```

Program w Javie jest dwa i pół raza większy od programu w Pythonie i w przypadku plików o niewielkich rozmiarach będzie też działał wolniej[6]. Przed uruchomieniem musimy go skompilować, lecz oprócz *tego* wszystkiego jest prawie tak samo prosty…

Powrót do plików konfiguracyjnych

Wróćmy zatem do naszych plików konfiguracyjnych XML i zadbajmy o to, aby wartość każdego atrybutu w sekcji była ustawiona tylko jeden raz. Funkcja wczytująca zawartość pliku *.ini* nie wymaga żadnych modyfikacji, podobnie funkcja zastępująca znaki specjalne `&`, `<` i `>` ich sekwencjami. Zmodyfikować musimy natomiast funkcję `process()`.

Na początku każdej sekcji utworzymy pusty słownik. Jako że atrybuty są same w sobie kluczami i wartościami, możemy wykorzystać to w naszym słowniku. Jeśli dowolny klucz wystąpi w sekcji więcej, niż jeden raz, kolejne wystąpienia przesłonią poprzednie, co spowoduje, że na końcu będzie widoczna tylko ostatnio ustawiona wartość:

```
def process(lines):
    result = []
    section = None
    content = {}
    for line in lines:
        # Początek nowej sekcji
        if line[0] == '[':
            # Zapisanie starych wartości (jeśli są)
            if section:
                entry = [section, content]
                result.append(entry)
            # Początek nowej sekcji
            section = line[1:-1]
            content = {}
```

[6] Java z reguły działa szybciej od Pythona, ponieważ w Javie typy zmiennych są kontrolowane na etapie kompilacji, natomiast w przypadku Pythona typ zmiennych musi być kontrolowany w trakcie wykonania. Maszyna wirtualna Javy ma jednak o wiele większe rozmiary od interpretera Pythona, podobnie jej biblioteki mają większe wymagania pamięciowe. To powoduje, że w przypadku przetwarzania plików o niewielkich rozmiarach czas niezbędny na załadowanie do pamięci całej maszyny wirtualnej sprawia, iż pomimo faktu, że Java jest szybsza w działaniu, program będzie wykonywał się dłużej.

```
            # Dodaj do bieżącego słownika.
            else:
                key, value = line.split("=", 1)
                content[key] = escape(value)
        # Zakończ
        if section:
            entry = [section, content]
            result.append(entry)
        return result
```

Ta funkcja ponownie musi wziąć pod uwagę przypadki specjalne na początku i końcu przetwarzania. Mimo zmian, jej działanie nadal jest bardzo intuicyjne. Zwracany wynik jest listą par. Pierwszym elementem każdej pary jest tytuł sekcji, drugim natomiast słownik zawierający ostateczną wartość każdego atrybutu w tej sekcji.

Zmieniliśmy format danych zwracanych z funkcji `process()`, musimy zatem zmienić też wykorzystującą je funkcję `writeXml()`:

```
def writeXml(outputName, settings):
    output = open(outputName, 'w')
    print >> output, '<configure>'
    for (section, content) in settings:
        print >> output, '<%s>' % section
        for key in content:
            value = content[key]
            print >> output, ' <%s>%s</%s>' % (key, value, key)
        print >> output, '</%s>' % section
    print >> output, '</configure>'
    output.close()
```

Na koniec należy przetestować działanie kodu. Przede wszystkim należy sprawdzić go w tych danych, których porażkę poniosła pierwsza wersja programu. Na potrzeby testów przygotowałem więc specjalnie spreparowany plik *.ini* zawierający szczególnie ważne cechy:

```
[Section1]
Key=a
Key=b
[Section2]
Key=c
[Section3]
Red=crimson
Green=lime
Red=vermilion
Green=chartreuse
[Section4]
```

Wynik działania programu jest następujący:

```
<configure>
<Section1>
  <Key>b</Key>
</Section1>
<Section2>
  <Key>c</Key>
</Section2>
<Section3>
  <Green>chartreuse</Green>
  <Red>vermilion</Red>
</Section3>
<Section4>
</Section4>
</configure>
```

Jeszcze chwila… Przyjrzyjmy się sekcji **Section3**. W pliku *.ini* klucz **Red** wystąpił przed kluczem **Green**. W pliku *.xml* pojawił się później. Czy to błąd?

Nie, to działanie zamierzone. Jak wspominałem wcześniej, klucze w słownikach nie mają uporządkowanej kolejności. Ta cecha słowników spowodowała, że **Green** pojawił się przed **Red**, nawet pomimo tego, że w danych wejściowych występował wcześniej.

Czy to jest problem? To zależy wyłącznie od tego, czy znaczenie ma kolejność kluczy w sekcji, co z kolei zależy od programu CAD wykorzystującego tę konfigurację. Jeśli nie ma to znaczenia dla programu, nie warto się tym przejmować. Jeśli jednak ma to znaczenie, można zmodyfikować kod, na przykład zachowując kolejność kluczy atrybutów w dodatkowych słownikach, dla każdej sekcji z osobna.

2.5. Włączanie plików zewnętrznych

Na szczęście w tym przypadku kolejność kluczy nie miała znaczenia. Pojawił się jednak inny problem. W trakcie konwersji plików *.ini* na format XML okazało się, że pliki *.ini* mogły odwoływać się do innych plików *.ini*, dzięki czemu część konfiguracji mogła być zdefiniowana w jednym, stałym miejscu. Na przykład jedna z grup inżynierów w firmie wykorzystywała plik *.ini* o następującej zawartości:

```
%general.ini%
[View]
WindowSize=1024,768
BackgroundColor=green7
WindowSize=1280,1024
```

Natomiast zawartość pliku *general.ini* była następująca:

```
[Drawing]
LineWidth=2
Corners=Rounded
[File]
DefaultName=$PROJECT.$VERSION
DefaultTitle=off
```

Niestety, nowy format XML nie uwzględniał tej możliwości: każdy plik musiał być samodzielną całością.

W pierwszej kolejności należy oszacować skalę problemu. W firmie było zatrudnionych około osiemdziesięciu inżynierów, z czego tylko piętnastu wykorzystywało konfiguracje włączające pliki zewnętrzne. Żaden z przypadków nie był wielokrotnie zagnieżdżony (to znaczy drugi plik nie zawierał włączenia kolejnego pliku). Pliki konfiguracyjne muszą być skonwertowane tylko raz, zdecydowałem więc, że po prostu ręcznie połączę te piętnaście plików *.ini* w całości i uruchomię procedurę przekształcającą.

To zadanie zajęło około dwudziestu minut, z których większość spędziłem na poszukiwaniu administratora sieci w celu udostępnienia mi tych plików do odczytu. Gdyby plików do przetwarzania było około setki lub gdyby okazało się, że wykorzystuję wielokrotne, rekurencyjne włączanie, z pewnością zdecydowałbym się zmodyfikować swój program *ini2xml.py*.

Jak zawsze na początek zdecydowałbym, w którym miejscu skryptu muszę wprowadzić poprawki. Jednym ze sposobów może być napisanie kolejnej funkcji filtrującej, zastępującej zawartością pliku wszystkie wystąpienia instrukcji włączającej plik zewnętrzny. Ta metoda zadziała jedynie w przypadku, gdy nie mamy do czynienia z rekurencyjnym włączaniem, jeśli jednak byłoby inaczej, należy zastosować odpowiednią pętlę, na przykład:

```
lines = readIni(filename)
while containsIncludes(lines):
    index = findFirstInclude(lines)
    expansion = readIni(index)
    lines = insert(lines, index, expansion)
```

To nie wygląda najgorzej, lecz wszyscy Czytelnicy, którzy mieli okazję przetwarzać pliki tekstowe włączające inne pliki, z pewnością uśmiechną się z powątpiewaniem. Należy bowiem pamiętać o możliwości wystąpienia błędnej sytuacji samowłączenia pliku, na przykład (plik nosi nazwę *selfinclusion.ini*):

```
[Foo]
a = b
%selfinclusion.ini%
```

Po pierwszym przebiegu pętli program przetwarzający będzie zawierał następujące dane:

```
[Foo]
a = b
[Foo]
a = b
%selfinclusion.ini%
```

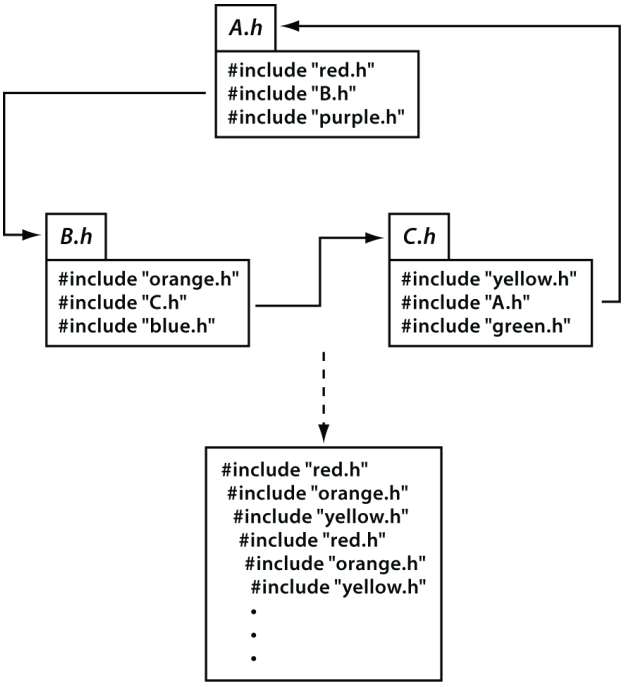
Po drugim przebiegu postać danych będzie następującą:

```
[Foo]
a = b
[Foo]
a = b
[Foo]
a = b
%selfinclusion.ini%
```

Przetwarzanie będzie odbywać się w nieskończoność, a raczej do momentu zapełnienia pamięci lub przerwania działania programu kombinacją *Ctrl*+C.

Wystąpienie takiego przypadku w rzeczywistości wydaje się dość mało prawdopodobne. W końcu, komu przyszłoby do głowy włączać plik w nim samym? Zupełnie prawdopodobny jest jednak tylko trochę bardziej skomplikowany przypadek, gdy plik A włącza plik B, który włącza C, który z kolei włącza na powrót plik A (rysunek 2.1). Taka sytuacja jest zupełnie powszechna w językach programowania C i C++, gdzie często zdarza się, że biblioteki standardowe są uzależnione wzajemnie od siebie.

Standardowe rozwiązanie tego problemu wykorzystuje stos przetwarzanych aktualnie plików i wówczas gdy zostanie napotkany przypadek włączenia jednego z przetwarzanych plików, procedura jest przerywana. W naszym przypadku możemy to zaimplementować za pomocą rekurencji w funkcji



Rysunek 2.1. Cykliczne włączanie plików nagłówkowych

`readIni()`. Wraz z nazwą włączanego pliku `redIni()` musi otrzymać listę (a dokładniej: stos) nazw aktualnie przetwarzanych plików. Gdy natrafi na instrukcję włączenia pliku, sprawdza, czy nazwa tego pliku nie znajduje się aktualnie na stosie. Jeśli nie, umieszcza tę nazwę na szczycie stosu, wywołuje się rekurencyjnie z tą nazwą pliku i zmodyfikowanym stosem. Wynik wywołania rekurencyjnego jest zapisywany na liście przetworzonych wierszy. Zmodyfikowany fragment skryptu znajduje się na poniższym listingu:

```
def error(msg):
    print >> sys.stderr, msg
    sys.exit(1)
def readIni(inputName, stack):
    input = open(inputName, 'r')
    result = []
    for line in input:
        # ten fragment pozostaje bez zmian
        line = line.strip()
        first = line.find('#')
        if first >= 0:
            line = line[:first]
```

```
        if (not line) or (line[0] == '#'):
            continue
        # zwykły wiersz
        if line[0] != '%':
            result.append(line)
        # włączenie
        else:
            filename = line[1:-1]
            if filename in stack:
                error("Rekurencyjne włączenie pliku %s: %s" %
                   (filename, repr(stack)))
            newStack = stack + [filename]
            inclusion = readIni(filename, newStack)
            result = result + inclusion
    return result
```

Wywołanie funkcji `readIni()` w głównej części programu zmieniamy na następujące:

```
lines = readIni(inputName, [inputName])
```

Czas na przetestowanie nowej wersji. Czy skrypt działa poprawnie w przypadku plików *.ini* nie włączających innych plików? Czy działa poprawnie w przypadku włączenia pliku w samym sobie? Praktycznie natychmiast zwraca komunikat o błędzie i kończy działanie[7]. Co się stanie w przypadku, „plik A włącza plik B, który włącza plik C, który włącza plik A"? Przygotowanie odpowiedniego zestawu testowego zajmuje dosłownie chwilę, warto więc sprawdzić taki przypadek. Wszystko w porządku, również w tym przypadku program zachowuje się zgodnie z założeniami.

Wnioski

To ćwiczenie z przetwarzania danych tekstowych pozwala zdobyć sporo uniwersalnych doświadczeń związanych z przetwarzaniem danych. Po pierwsze: nie można zbyt wcześnie uznać, że zadanie jest skończone. Pierwsze rozwiązanie problemu związanego z przetwarzaniem danych z reguły nie uwzględni kilku ważnych szczegółów, kluczowe jest zatem, aby kod pisać w sposób czytelny, nawet w przypadku gdy kod z założenia ma być jednorazowego użytku. Warto też przechować taki kod w systemie kontroli wersji, aby móc odszukać go za jakiś czas, gdy mimo wszystko okaże się potrzebny.

[7] W zależności od potrzeb, zamiast kończyć działanie z błędem, można po prostu pomijać plik włączany w sposób rekurencyjny.

Drugie spostrzeżenie dotyczy użyteczności programowalnych edytorów tekstu, jak Emacs czy Vim. W opisanej sytuacji edycji piętnastu plików (gdy włączałem w nich zawartość innych plików *.ini*) oczywiście nie wykonywałem tej czynności piętnaście razy. Zamiast ręcznej edycji zapisałem makro podczas edycji pierwszego pliku i wywołałem je czternaście razy (po czym zapisałem na wszelki wypadek). Gdybym miał do czynienia z przypadkiem rekurencyjnego włączania lub innych skomplikowanych struktur danych, zapewne nie poważyłbym się na ich obróbkę w edytorze tekstu[8], lecz zdumiewająco wiele przypadków z życia udaje się sprowadzić do tak prostych rozwiązań.

2.6. Powłoka Uniksa

Gdyby jakość produktu mierzyć czasem jego aktywnego użytkowania, powłokę Uniksa[9] należałoby uznać za najlepsze narzędzie do przetwarzania danych tekstowych w historii komputeryzacji. Powłoka Uniksa ma ponad trzydzieści pięć lat i jest nadal wykorzystywana przez wielu programistów jako ulubione narzędzie do przetwarzania informacji.

Jak stwierdzili ich twórcy [KP99], języki powłoki zawdzięczają swoje możliwości filozofii „dużej liczby niewielkich, specjalizowanych narzędzi". Zamiast bowiem oferować wielkie programy „do wszystkiego", powłoka Uniksa daje do użytku wiele małych programików, które realizują zaledwie pojedyncze funkcje, za to realizują je doskonale. Co więcej, powłoka ułatwia łączenie tych narzędzi na różne sposoby oraz dodawanie nowych.

Aby ta filozofia sprawdzała się w życiu, dobre narzędzie powłoki Uniksa powinno być napisane w zgodzie z następującymi zasadami:

---

[8] Programiści znający doskonale Emacs Lisp zapewne mogą być innego zdania.

[9] Jeśli chodzi o ścisłość, nie powinniśmy mówić „powłoka Uniksa", ponieważ nie istnieje jeden jedyny produkt noszący tę nazwę. Mamy bowiem do czynienia z dziesiątkami języków skryptowych, które mogą działać jako powłoki Uniksa, począwszy od klasycznego już */bin/sh*, po mój ulubiony *bash*. W tej książce zastosuję jak najbardziej przenośną składnię zgodną z większością języków powłoki.

- pobieranie danych ze standardowego wejścia i wypisywanie wyników na standardowym wyjściu, chyba że zostanie to określone inaczej za pomocą parametru wywołania;
- na wejściu oczekiwanie wierszy tekstu i zwracanie wyniku w takiej samej formie na wyjściu;
- jeśli przetwarzanie zakończy się powodzeniem, program powinien zwrócić powłoce kod wyjścia równy 0 (zero), każda inna wartość powinna sygnalizować sytuację błędu.

I to już wszystko, to są reguły, które muszą być przestrzegane przez program przeznaczony do pracy w powłoce. Dzięki nim bez problemu będzie mógł współpracować z dziesiątkami innych programów napisanych zgodnie z tą samą filozofią. Przestrzeganie tych reguł umożliwia bowiem *przekierowania* (ang. *redirection*) wyniku działania programu oraz łączenie kilku programów w *potoki* (ang. *pipe*). Przekierowanie służy przekazaniu danych z pliku na wejście programu, zamiast na przykład wpisywania ich z klawiatury. Druga funkcja przekierowania służy zapisywaniu danych z wyjścia programu. Pierwsza z tych funkcji realizuje się za pomocą operatora <, na przykład:

```
myprog < somefile.txt     # przekazanie zawartości pliku file.txt na
   wejście programu myprog
```

Drugą z funkcji przekierowania obsługuje się za pomocą operatora >, na przykład:

```
myprog > anotherfile.txt  # przekazanie danych z wyjścia programu
   myprog do pliku anotherfile.txt
```

Załóżmy na przykład, że chcemy zapisać w pliku wszystkie nazwy plików Javy zapisanych w katalogu bieżącym. Do sporządzania listingów zawartości katalogów służy program `ls`, lecz standardowo wyniki swojego działania wypisuje on na ekranie. Nie ma jednak problemu, ponieważ dzięki przekierowaniu wyjścia do pliku możemy zrobić coś takiego:

```
ls *.java > javafiles.txt
```

Potok to po prostu połączenie wyjścia jednego programu z wejściem innego. Załóżmy na przykład, że chcemy policzyć, ile w katalogu bieżącym zostało

zapisanych plików Pythona. Program `ls` sporządzi ich listę, natomiast `wc` zliczy znaki, słowa oraz wiersze w danych przekazanych mu na wejściu:

```
ls *.py | wc
```

Gdy wywołałem to polecenie w katalogu, w którym zapisałem przykłady z tego rozdziału, otrzymałem następujący wynik;

```
10    10    147
```

Dziesięć wierszy, dziesięć słów (w każdym wierszu znajduje się jedno słowo) oraz 147 znaków w całości. Aby zapisać ten wynik do pliku, wystarczy w jednym wywołaniu wykorzystać potoki i przekierowanie:

```
ls *.py | wc > numfiles.txt
```

Podobnych możliwości jest nieskończenie wiele. Ten mechanizm jest prosty, lecz elastyczny i tak użyteczny, jak „prawdziwy" język programowania.

W rzeczywistości języki powłoki są językami programowania, ponieważ obsługują zawansowane mechanizmy kontroli przebiegu, jak pętle, warunki, zmienne czy funkcje. Poniższy listing prezentuje prosty program, który można wykorzystać do odszukania katalogów domowych użytkowników, których identyfikatory systemowe (ID) są zapisane w pliku:

```
cat $1 | while read uid
do
    echo $uid
    grep $uid /etc/passwd | cut -d : -f 5
done
```

Załóżmy, że na wejściu podamy taki plik:

```
gvwilson:x:182:9:Greg Wilson:/h/6/gvwilson:/bin/bash
dave:x:180:7:Dave Thomas:/h/3/dave:/bin/bash
andy:x:181:7:Andy Hunt:/h/3/andy:/bin/tcsh
alant:x:196:9:Alan Turing:/h/6/alant:/bin/tcsh
```

Program należy uruchomić w następujący sposób:

```
./findhome.sh findhome-in.txt
```

W wyniku jego działania otrzymamy wynik:

```
gvwilson
/h/6/gvwilson
dave
/h/3/dave
andy
```

```
/h/3/andy
alant
/h/6/alant
```

Przeanalizujmy działanie tego skryptu:

- `$1` to argument wiersza poleceń o indeksie 1 (pod indeksem 0 znajduje się nazwa skryptu), zatem polecenie `cat $1` spowoduje wypisanie na standardowym wyjściu zawartość pliku, którego nazwa zostanie podana jako argument wywołania skryptu.
- Zamiast wysyłać zawartość pliku na standardowe wyjście, skrypt przekazuje wynik działania polecenia `cat` jako strumień wejściowy pętli `while`. Pętla ta przetwarza plik wiersz po wierszu (zmienna `uid`).
- Główna część pętli wypisuje na wyjściu identyfikatory użytkownika (UID). Przed nazwą zmiennej używany jest znak `$`. Wymusza on wydobycie wartości zmiennej w miejscu zastosowania jej nazwy.
- W następnym wierszu wykorzystane jest polecenie `grep`, za pomocą którego w pliku */etc/passwd* poszukiwany jest identyfikator użytkownika. Ten plik zawiera bazę kont użytkowników w większości systemów Unix. Wiersze wyszukane przez to polecenie są przekazywane do polecenia `cut`. Ten fragment algorytmu jest dość nieprecyzyjny: jeśli w pliku wejściowym wystąpi nazwa konta `jan`, zostaną uwzględnione konta `jan`, `janusz`, `janek` itp. oraz wszelkie inne pozycje, które zawierają ten ciąg znaków. W następnym rozdziale opiszę sposoby lepszego definiowania wzorców dopasowań dla polecenia `grep`.
- Polecenie `cut` dzieli wiersze wejściowe na pola, w charakterze separatora używając znaku dwukropka (parametr `-d`), po czym wybierane jest piąte pole utworzonego w ten sposób rekordu (parametr `-f`), w którym to polu znajduje się właśnie ścieżka katalogu domowego użytkownika.

Korzystanie z wiersza poleceń wymaga nieco czasu, aby się do niego przyzwyczaić , lecz po poznaniu mechaniki działania powłoki za pomocą kilku poleceń tekstowych można wykonać wiele zadań. Tabela 2.1 zawiera kilka poleceń szczególnie przydatnych przy przetwarzaniu danych.

Tabela 2.1. Przydatne polecenia

`cat`	połączenie plików i wypisanie ich na wyjściu
`cd`	zmiana katalogu roboczego
`chmod`	zmiana uprawnień do plików i katalogów
`cut`	wybór pól z rekordu tekstowego
`cp`	kopiowanie plików i katalogów
`date`	wypisanie bieżącej daty i czasu
`diff`	określenie różnic pomiędzy dwoma plikami
`du`	wypisanie zajętości dysku przez wskazane pliki i katalogi
`echo`	wypisanie argumentów
`env`	wypisanie zmiennych środowiska
`find`	wyszukiwanie plików i katalogów o zadanych właściwościach
`grep`	wypisanie wierszy zgodnych z wzorcem dopasowania
`head`	wypisanie pierwszych wierszy pliku
`lpr`	przesłanie pliku do drukarki
`ls`	wypisanie nazw plików i katalogów
`man`	dokumentacja poleceń
`mkdir`	tworzenie katalogów
`mv`	przesunięcie lub zmiana nazwy plików i katalogów
`od`	wypisanie zrzutu zawartości pliku w kilku formatach
`ps`	wypisanie statusu procesów systemowych
`pwd`	wypisanie ścieżki do bieżącego katalogu
`rm`	usuwanie plików
`rmdir`	usuwanie katalogów
`sort`	sortowanie wierszy
`tail`	wypisanie ostatnich wierszy pliku
`tar`	archiwizowanie plików
`uniq`	usuwanie zduplikowanych wierszy
`wc`	obliczenie liczby wierszy, słów i znaków pliku
`zip`	kompresowanie i dekompresowanie plików

Wiersz poleceń stanowi bardzo proste narzędzie do wykonywania prostych zadań i jest dość efektywny, jeśli weźmie się pod uwagę czas i nakład pracy; należy jednak pamiętać o jego ograniczeniach. Najważniejszym z nich (z punktu widzenia przetwarzania danych) jest brak obsługi danych strukturalnych[10]. W klasycznych systemach Unix wszelkie dane przetwarzane z poziomu powłoki stanowią listy ciągów znaków. Jeśli ktoś potrzebuje bardziej zaawansowanej struktury danych, na przykład drzewiastej, zapewne łatwiej będzie sięgnąć po Pythona, Ruby'ego czy podobny język programowania. Alternatywnym podejściem może być zainwestowanie w komercyjne narzędzia (na przykład TextPipe Pro firmy Crystal Software: *http://www.crystalsoftware.com.au*), które pozwalają wykorzystać wiele nietekstowych struktur danych również z poziomu języków powłoki.

Jak zbudować poprawnie działające narzędzie

Jak wspominałem wcześniej, poprawnie działające narzędzie wiersza poleceń powinno czytać dane ze standardowego wejścia, a wyniki swoich działań zapisywać na standardowym wyjściu, dzięki czemu programiści będą mieli możliwość łączenia tego narzędzia z innymi za pomocą potoków. Gdy mamy pewność, że narzędzie do przetwarzania danych nie będzie wykorzystane więcej, niż raz, nie warto upierać się przy zachowaniu tego wymogu. Jeśli jednak istnieje prawdopodobieństwo, że ten program będzie wykorzystywany wielokrotnie lub w przypadku gdy tworzony program ma być elementem większego systemu do przetwarzania danych, warto go stworzyć w zgodzie z pewną konwencją (która nie zawsze jest spełniona w pełni przez narzędzia uniksowe):

- jednoliterowe parametry wiersza poleceń powinny rozpoczynać się od jednego myślnika, na przykład `-p`;
- wieloliterowe parametry wiersza poleceń powinny rozpoczynać się od dwóch myślników, na przykład `--print`;

[10] Być może wolna od tego ograniczenia będzie powłoka nowej generacji firmy Microsoft, o nazwie kodowej Monad.

- jeśli nie zostanie podana nazwa pliku, program powinien czytać dane wejściowe ze standardowego wejścia i zapisywać wyniki swoich działań na standardowym wyjściu;
- jeśli zostanie podana jedna nazwa pliku, program powinien odczytać z niego swoje dane wejściowe, a wyniki działań zapisać na standardowym wyjściu;
- jeśli zostaną podane dwie nazwy plików, program powinien odczytać swoje dane wejściowe z pierwszego z nich, a wyniki działań zapisać w drugim;
- alternatywna zasada jest taka, że program odczytuje swoje dane wejściowe ze wszystkich plików określonych w wywołaniu po kolei, a wyniki działań zapisuje na standardowym wyjściu;
- komunikaty o błędach powinny być wypisywane na standardowym strumieniu błędów (co spowoduje, że zostaną wypisane na ekranie nawet w przypadku, gdy wynik będzie przekierowany do pliku);
- w przypadku poprawnego wykonania program powinien zwrócić status równy zeru, ponieważ powłoka interpretuje kod wyjścia różny od zera jako sygnalizację błędu wykonania.

Załóżmy na przykład, że chcemy pobierać próbki danych co *N*-ty wiersz. Jeśli użytkownik nie wskaże liczby *N* za pomocą parametru wywołania `–n`, zostanie przyjęta domyślna wartość `N=10`. Program będzie odczytywał dane z plików wskazanych przez użytkownika. Jeśli nie zostanie określona żadna nazwa pliku, dane będą czytane ze standardowego wejścia. Wynik działania będzie zapisywany na standardowym wyjściu, chyba że zostanie określona nazwa pliku wyjściowego za pomocą opcji `-o`.

W pierwszym kroku należy przetworzyć argumenty wiersza poleceń:

```
# ustawienie znaczników
input = []
output = None
sampling = None
# analiza parametrów wywołania
argdex = 1
while argdex < len(sys.argv):
    if sys.argv[argdex] in ["-n", "—number"]:
        if sampling is not None:
            fail("Błąd: częstotliwość ustawiona wielokrotnie")
```

```
            argdex, arg = getArg(argdex, sys.argv)
            try:
                sampling = int(arg)
            except ValueError:
                fail("Błąd: parametr –n/--number wymaga podania wartości
                   liczbowej (nie '%s')" % arg)
            if sampling <= 0:
                fail("Błąd: częstotliwość musi być liczbą dodatnią (nie
                   %d)" % sampling)
        elif sys.argv[argdex] in ["-o", "–output"]:
            if output is not None:
                fail("Błąd: za duża liczba plików wyjściowych")
            argdex, arg = getArg(argdex, sys.argv)
            try:
                output = open(arg, "w")
            except IOError:
                fail("Błąd: nie można otworzyć pliku '%s' w trybie do
                   zapisu" % arg)
        elif sys.argv[argdex][0] == "-":
            fail("Błąd: nierozpoznany parametr '%s'" % sys . argv[argdex])
        else:
            input.append(sys.argv[argdex])
            argdex += 1
        # sprawdzenie parametrów, ustawienie wartości domyślnych
    if input == []:
        input = ["-"]
    if output is None:
        output = sys.stdout
    if sampling is None:
        sampling = 10
```

Może się pojawić konieczność przetwarzania większej liczby plików wejściowych, zmienna `input` jest ustawiona początkowo na pustą listę. Jeśli po przetworzeniu parametrów ta zmienna jest nadal pustą listą, dane będą odczytane ze standardowego wejścia. Zmienne `output` i `sampling` są ustawione na `None`, więc istnieje możliwość zweryfikowania, czy ktoś nie usiłował ustawić ich wartości wielokrotnie (gdyby na samym początku ustawić ich wartości domyślne, trudno byłoby stwierdzić, czy parametr został ustawiony przez użytkownika). Program zawiera sporo kontroli błędów. Warto zadbać o tego typu szczegóły, jeśli program ma być używany wielokrotnie, warto ułatwić użytkownikom jego użytkowanie.

Funkcje przetwarzające parametry wywołania mają bardzo prostą konstrukcję:

```
# Wypisanie komunikatu o błędzie i zakończenie pracy z kodem błędu
def fail(msg):
    print >> sys.stderr, msg
    sys.exit(1)
```



Jaś pyta...

Czy zawsze muszę pisać tak wielką ilość kodu?

Nie. Większość języków programowania zawiera biblioteki i moduły przetwarzające listy parametrów wywołania, najczęściej biblioteki te noszą nazwę `getopt` lub podobną. W swoim kodzie zaprezentowałem samodzielny sposób przetwarzania listy parametrów wywołania wyłącznie dla celów demonstracji i na potrzeby programowania w języku, w którym nie mamy dostępnej biblioteki realizującej tę funkcję.

```
# Odczyt argumentu lub zakończenie pracy z kodem błędu
def getArg(argdex, args):
flag = args[argdex]
if argdex >= len(args)-1:
    fail("Należy podać wartość parametru %s " % flag)
return argdex+2, args[argdex+1]
```

Główny kod programu również jest bardzo prosty:

```
# Przetwarzanie jednego pliku lub strumienia
def process(filename, output, sampling):
    if filename == "-":
        stream = sys.stdin
    else:
        try:
           stream = open(filename, "r")
        except IOError:
            fail("Nie można otworzyć '%s' w trybie do odczytu" % filename)
    count = 0
    for line in stream:
        count += 1
        if count == sampling:
            output.write(line)
            count = 0
    stream.close()
# Przetwarzanie danych
for filename in input:
    process(filename, output, sampling)
output.close()
```

Warto zwrócić uwagę na funkcję `process()`, w której znajduje się sprawdzenie, czy „nazwą" pliku nie jest myślnik. W takim przypadku dane nie są odczytywane z pliku, tylko ze standardowego wejścia. Dzięki temu użytkownik

ma możliwość kontroli źródła danych dla programu, uwzględniające standardowe wejście.

Ten mechanizm można uprościć jeszcze bardziej, w zależności od używanego języka programowania. W Pythonie na przykład można wykorzystać moduł `fileinput`, który automatycznie przetwarza pliki podane w wierszu poleceń. Dzięki temu kod przetwarzający pliki można sprowadzić do następujących trzech wierszy:

```
import sys, fileinput
for line in fileinput.input():
    doSomething(line)
```

W ten sposób zostaną kolejno przetworzone wszystkie wiersze plików wejściowych, moduł wie nawet, że znak – oznacza „standardowe wejście".

W Perlu jest jeszcze prościej:

```
while (<>) {
    doSomething($_);
}
```

Symbol <> („diament") oznacza domyślny uchwyt pliku, natomiast symbol $_ oznacza „aktualnie przetwarzany fragment danych".

2.7. Bardzo duże zbiory danych

Własność Parkinsona [Par93] mówi, że każde zadanie prędzej lub później spuchnie tak, że zajmie cały dostępny czas i zasoby. Programiści mają bardzo często okazję stwierdzić, że dane zachowują się bardzo podobnie: nieważne, jak duże dyski mamy w swoich komputerach, przez 99% czasu swojego funkcjonowania są one zapełnione w 99%.

Dla przetwarzania danych oznacza to, że należy spodziewać się przypadków, w których załadowanie danych do pamięci spowoduje jej przepełnienie. Roczna zawartość logów popularnego serwera WWW może zająć od 10 do 15 gigabajtów miejsca na dysku. Jeśli komputer ma 1 lub 2 gigabajty pamięci operacyjnej (wraz z przestrzenią wymiany), w celu przetworzenia danych o takich rozmiarach należy podzielić je na mniejsze kawałki.

Jeśli mamy szczęście, dane możemy podzielić na części w ramach dwuetapowego procesu. W pierwszym dane są filtrowane w taki sposób, że to co pozostało mieści się już w pamięci, w drugim etapie wykonujemy nasze przetwarzanie z tymi odfiltrowanymi danymi. Załóżmy na przykład, że firma realizuje miliard transakcji rocznie i chcemy znaleźć milion największych z nich. Załóżmy, że w pamięci zmieści się najwyżej sto milionów rekordów. Problem możemy rozwiązać w następujący sposób:

```
for (każdy blok złożony ze 100 milionów rekordów)
    wczytaj do pamięci
    posortuj
    zapisz w pliku tymczasowym milion najwyższych wartości
wczytaj plik tymczasowy (o rozmiarze 10 milionów rekordów)
posortuj
wypisz milion najwyższych wartości
```

Co zrobić jednak, gdy nie możemy podzielić danych na mniejsze kawałki? Na przykład: co zrobić z multispektralnymi zdjęciami satelitarnymi albo obrazami z tomografu komputerowego i — nie ma wyjścia — potrzebne są od razu te 22 gigabajty danych? W tym przypadku jesteśmy skazani na *niestandardowe algorytmy*. Wiele z nich wywodzi się z czasów komputerów mainframe i pamięci taśmowych. Tego typu rozwiązania są z reguły o wiele bardziej skomplikowane od wersji „całe dane w pamięci". Jeśli ktoś znajdzie się w takiej sytuacji, oznacza to, że czas zwykłego przetwarzania danych skończył się, nadszedł czas na odejście od klawiatury i wykonanie solidnej pracy analitycznej i projektowej.

2.8. Podsumowanie

Ten rozdział zawierał wprowadzenie kilku podstawowych koncepcji i technik. Niektóre z nich są specyficzne dla przetwarzania strumieni tekstu, inne mają ogólne zastosowanie w technikach przetwarzania danych. Podsumujmy najważniejsze z nich:

- Należy rozpocząć od uogólnienia i dodawać do rozwiązania kolejne szczegóły dopiero wówczas, gdy napotkamy przypadki, których uogólnienie nie obsługuje. Należy unikać rozwiązań przekombinowanych, implementujących rozwiązania nieistniejących problemów. W strategii Extreme Programming tego typu rozwiązania

Struktury danych

Przetwarzanie danych opiera się w większości przypadków na listach i katalogach, lecz czasem rozwiązanie problemu wymaga zastosowania technik opisanych w bardziej zaawansowanych algorytmach informatycznych. Na przykład problem wyszukania miliona najwyższych wartości z listy miliarda można rozwiązać za pomocą kolejki priorytetowej, to znaczy listy, w której `q[k]` ma zawsze większą wartość od `q[2*k+1]` do `q[2*k+2]`. Za każdym razem, gdy do tej listy jest dodawany element, kolejka porządkuje się automatycznie, zachowując tę zasadę. Jeśli ograniczymy jej rozmiar do miliona rekordów, można przetworzyć miliard rekordów po kolei (pojedynczo), a w efekcie otrzymamy milion największych wartości.

Standardowa biblioteka Pythona zawiera obsługę kolejki priorytetowej, służy do tego moduł `heapq`. Podobne moduły są dostępne dla większości języków programowania. Jeśli Czytelnik wykorzystuje język, do którego nie ma odpowiedniej biblioteki, można samemu zaimplementować odpowiednią strukturę zgodnie z opisem w [Sed97] lub wielu innych podręcznikach omawiających algorytmy i struktury danych.

określa się akronimem *YAGNI*: ang. *You Ain't Gonna Need It*, czyli „nie będziesz tego potrzebował".

- Należy oddzielić etap odczytu od etapów przetwarzania i zwracania wyników. Dzięki temu każdy z etapów będzie łatwiej napisać, testować, użyć ponownie i rozwijać.
- Nie wolno się powtarzać.
- Nie należy szukać wymówki w fakcie, że rozwiązanie jest jednorazowe, można sobie więc pozwolić na nonszalanckie techniki programistyczne. Najczęściej okazuje się bowiem, że modularyzacja i testowanie kodu pozwalają szybciej uzyskać poprawne wyniki.
- Uczmy się swoich narzędzi. W szczególności warto poznać standardową bibliotekę swojego ulubionego języka programowania i możliwości standardowych narzędzi Uniksa. Warto też jak najlepiej opanować możliwości używanego edytora tekstu. Taka nauka znacznie procentuje w przyszłości.