情報検索

実験概要

オブジェクト指向スクリプト言語 Ruby を用いて，簡単な検索システムを作成することにより，大量文書からの情報抽出および検索について理解する．

実験スケジュール

第 1 日目　情報検索の基礎，Ruby 入門
第 2 日目　N グラム，KWIC の作成
第 3 日目　索引語の自動抽出
第 4 日目　キーワード検索システムの作成
第 5 日目　応用課題

目 次

作成者: 小川 泰弘

最終更新日: 2004/9/13

1　目的

計算機を用いて大量の文章情報の中から必要な情報を抽出する作業を試みることにより，情報検索の基本的な手法と手順について理解する．

2　解説

現在，情報化，IT 化が盛んに謳われる中で，世間には多数の情報が溢れている．そうした中から求める情報を検索するのが情報検索 (information retrieval;IR) である．

この情報検索は言語処理と密接な関係がある．なぜなら，言語，すなわち「ことば」が情報を表現するのに最も広く使われているからである．事実，多くの情報が「ことば」によって表現されており，情報を探すということは，多くの場合，その情報について書かれたテキストを探すことに他ならない．

しかし,「ことば」と情報の関係には曖昧さがつきまとう．例えば，ある情報を表現する「ことば」は複数あるし，逆に「ことば」が文脈に依存して複数の意味をもつこともある．そうした場合，検索者の望む情報を得るにはどうしたら良いかが問題となる．

本実験では，人間の情報検索を支援するツールを作成し，その実現技術について主に自然言語処理の立場から考察を加える．

2.1　自然言語処理の基礎技術

2.1.1　N グラム

情報検索においては，検索対象となる大量の情報 (データ) の性質を捉えることが重要である．対象となる情報が「ことば」で表される以上，その「ことば」に対する処理が必須であるが，その中には自然言語に関する知識が必要なものと，単に文字列を処理するだけで行えるものがある．

例えばデータを検索する際に，バラバラのデータよりもソートしたデータの方が効率良く検索が行えるのは明らかであり，二分探索なども，データがソートされてこその探索手法である．また，さらにデータを効率良く検索するための手法として，ハッシュ法やトライ法などが知られている．

こうしたソートは単なる文字列処理であるが，言語の性質を捉える興味深い処理として N グラム (N-gram) がある．これは，隣接する N 個の文字 (もしくは単語) の共起関係の頻度をカウントしたものである．2 グラム (bigram) や 3 グラム (trigram) が良く利用される．また，N=1 の場合はユニグラム (unigram) と呼ばれるが，これは単なる文字 (単語) の出現頻度である．

文字 N グラムを計算するには，文字種すべてについて N 文字の組み合わせの表を用意し，出現頻度をカウント・アップするのが簡単である．しかし，N を大きくするとこの表は指数関数的に大きなものになる．また単語 N グラムの場合には，さらに表が大きくなるため，その計算には工夫が必要になる．

た相撲協会は２７日朝、理事会、	名古屋	場所番付編成会議を開き、正式決
古屋生まれ、名古屋育ちの生粋の	名古屋	人だ。
愛知万博構想は、	名古屋	人の執念を感じさせる。
	名古屋	人はケチくさいだの、しみったれ
私は	名古屋	生まれ、名古屋育ちの生粋の名古
	名古屋	製鉄所システム部では、昭和５６
オリンピック誘致の失敗で	名古屋	全体のイメージは、ガタッと低下
	名古屋	大が前年度より１８人、東北大が
高等科を卒業後、理化学研究所や	名古屋	大で宇宙線の研究に取り組んだ。
また	名古屋	大医学部グループの研究で、ＶＤ
富士通研究所は１８日、	名古屋	大学で開かれる応用物理学会学術
プログラムライブラリーとして,	名古屋	大学で開発されたＳＬＩＰがある
	名古屋	大学の経済学部が来年度の入試か
私は	名古屋	大学の飯田経夫さんがリーダーを
	名古屋	大学工学部で建築学を教える月尾
これは最近増加している	名古屋	地区のユーザーの声を反映させた
	名古屋	地区を中心に食肉業者らが、台湾
同特捜部では、	名古屋	地検から指紋鑑定の専門家を呼び
これまで東京、	名古屋	地裁でも判決が出ているが、いず
できる地位を求めて昨年１０月、	名古屋	地裁に提訴した 。
東海会社でも国労組合員１１人が	名古屋	地裁に同様の仮処分命令を申請し
単身赴任の	名古屋	地方気象台長時代、カラオケをお
っては、仙台を中心にする東北、	名古屋	中心の中部、大阪中心の関西の３

図 1: 「名古屋」をキーにした KWIC(EDR 日本語コーパスより)

2.1.2 KWIC

言語の使用状況を捉える別の手法として，KWIC(Key Word In Context) がある．KWIC は，テキスト中に出現するすべての単語 (文字列) を取り出してソートし，その単語が出現した前後の文字列を付加して出力するものである．KWIC の表は膨大なものとなるが，ある特定の単語を与えたときに，その部分だけを出力することも可能であり，例えば図 1 のようになる．

KWIC は，ある単語がどの用に利用されるかを調べるうえで重要な役割りを果たす．また，KWIC を索引に利用すると，単純な用語索引に比べて利便性が増す．例えば「並列オブジェクト指向言語」のようは複合語を「並列」だけでなく「オブジェクト」や「指向」，「言語」などからも引くことができ，さらに「オブジェクト」で検索すれば,「並列オブジェクト指向言語」の後に「オブジェクト指向データモデル」など関連する用語があることも分かる.『岩波情報科学辞典』では，このような KWIC による索引が採用されている．

また，検索エンジン Google の検索結果の出力も KWIC 形式になっている．

2.1.3 形態素解析

情報検索において，日本語と英語で異なる点として単語の扱いが挙げられる．

英語は分かち書きされることから，何をもって単語とするかの定義が明確であるが，日本語の場合，単語の定義が問題となる．例えば，“information retrieval” が 2 つの単語から構成されているのは明らかであるが「情報検索」が 1 単語なのか 2 単語なのかは人によって判断が分かれるであろう．

日本語における言語処理において単語を同定するのは，多くの場合，形態素解析の役割である．日本語における形態素解析は，分かち書きされていない入力文を単語に分割し，それぞれに品詞を付加するものである．英語においては，単語を分割する必要はないが，複数形や動詞の過去形などから原形を求める処理などが必要になる．

日本語情報検索においては，文書に含まれるキーワードが，形態素解析によって正しく分割されているかが問題となる.「辻元議員」を「辻」「元議員」と解析してしまったら,「辻元」で検索しても必要な情報が得られないかもしれない.

2.2 情報検索の基礎

情報検索は多くの場合，ある特定の文書群の中から求める文書を特定することになる．その検索手法は大きく分けて三つに分類される．一つは分類を用いた検索であり，その典型例は図書館の十進分類である．あらかじめ人手により文書を分類しておき，それに基づいて検索するのである．計算機による情報支援が普及する前には，主にこの方法が取られた.

二つ目は，キーワードによる検索である．あらかじめ文書にキーワードを付与し，さらに，それぞれのキーワードに対してどの文書に付加されているかの索引を作る．検索者は，この索引を利用して文書を検索する.

三つ目は，全文検索であり，文字通り，文書の全部を走査して検索者が与えたキーワードが含まれている箇所を見つけ出すものである.

キーワードによる検索および全文検索は計算機の利用により広く用いられるようになった．歴史的には，キーワード検索の技術がまず確立され，その後，多くの拡張がなされてきた．全文検索もその拡張の一つとして提案されたものである.

以下では，まずキーワード検索について紹介し，その次に全文検索の技術について説明する.

2.2.1 キーワード検索

キーワード検索の原理自体は簡単であり，与えられたキーワードがあらかじめ文書に付加されたキーワードと一致するかどうかをチェックし，一致したものを出力する，あるいは出力しないようにする．実際には，毎回全文書を走査するのは時間が掛かるため，転置インデックスと呼ばれる索引をあらかじめ作成しておく.

キーワード検索において問題となるのは，何をキーワードすなわち索引語 (index term) にするか，である．適切な索引語を選ばなければ，それは検索システムとしては役に立たない．人間が索引語を決定する場合もあるが，計算機を利用して自動的に索引語を抽出する方法が広く用いられている．その中でも，語の出現頻度を利用した索引語の自動抽出である tf.idf 法について紹介する.

tf.idf 法は，語頻度 (term frequency) と文書頻度 (document frequency) の二つの値を用いる．前者は，ある文書に何度も出現する語は，その文書の内容を象徴するキーワードになるということを反映する値であり，文書 d_i における語 t_j の出現した回数を表す.

$$tf_j^i = \text{文書 } d_i \text{における語 } t_j \text{の出現回数}$$

後者は，どの文書にも現れる語は，一般的すぎてキーワードにならないということを反

表 1: 文書と索引語

	索引語 1	索引語 2	索引語 3	索引語 4
文書 1	1	1	1	0
文書 2	0	1	1	1
文書 3	1	0	1	1
文書 4	0	0	1	1

表 2: 転置インデックス

	文書 1	文書 2	文書 3	文書 4
索引語 1	1	0	1	0
索引語 2	1	1	0	0
索引語 3	1	1	1	1
索引語 4	0	1	1	1

映する値で，ある語 t_j が文書群の中のいくつの文書に現れるかを表す．

$$df_j = \text{語 } t_j \text{を含む文書数}$$

tf.idf 法は，この二つを次にように組み合わせて索引語決定の際の指標とする．

$$w_j^i = tf_j^i \cdot \log \frac{N}{df_j}$$

ここで N は文書群の全文書数である．実際の運用では，ある特定の閾値を設け，これを超えたキーワードを索引語として使用する．なお，冠詞や前置詞などの機能語や，助詞や助動詞などの付属語はあらかじめ不要語 (stop word) として排除しておき，自立語あるいは名詞だけに限定して，これらの値を計算する手法もある．

索引語を決定した後は，それぞれの文書について索引語になるかどうかを判定して表を作成する．一番単純な判定法は，索引語が文書に出現するかどうかである．表 1 で “1” となっている部分は，文書の索引語として選ばれたことを，“0” は選ばれなかったことを示す．次に，この表 1 を行列と考え転置行列を作成することで索引語に対するインデックスとなる表 2 を作る．

実際の検索の際には，表 2 と入力キーワードの一致により，該当する文書を出力する．その際には，アンド検索やオア検索の他，入力キーワードに一致したものは出力しないノット検索などを組み合わせることもある．

2.2.2 全文検索

キーワード検索は索引語を使用して検索するため，あらかじめ適切な索引語が文書に付加されている必要がある．しかし，適切な索引語を付けることはそれほど簡単なことではなく，完璧な索引語付けは不可能である．そこで，索引語を付加することをやめ，文書全体を走査してキーワードが出現する位置をすべて求める手法が考えられる．

全文検索の実現は，文字列照合を行えばよく，例えば UNIX の grep コマンドだけでも実現可能である．また，出現位置の前後を含めて表示しようと思えば前述の KWIC を使用すればよい．しかし，大規模な文書群に対して，検索のたびに全テキストを走査するのは膨大な時間が掛かり現実的でない．そこで，テキストに出現するすべての単語に対してあらかじめ転置インデックスを作成する，すなわち，全出現単語を索引語とすることにより全文検索は実現される．

ただし，この方法では，単語でないものをキーワードをして検索することは不可能である．例えば，“inter-” で始まる単語をすべて検索したいとか，“-book” で終わる語を含む文をすべて抽出したい場合には利用できない．もちろん，grep を使用すれば可能であるが，前述の通り検索時間が問題となる．こうした問題に対する解決法としては，Suffix Array と呼ばれるデータ構造が知られている．これは，あらかじめ全ての文字の出現位置にインデックス・ポインタを割り当てソートし，それを二分探索することで膨大なテキスト文書に対する高速な検索を実現する方法である．

2.2.3 ベクトル空間法

これまで述べてキーワード検索法では，与えられたキーワードと一致したものしか検索できない．しかし，実際の検索では，ユーザの求める文書にキーワードが含まれておらず代わりに，キーワードの同義語もしくは，良く似た意味を持つ別の語が含まれている場合がある．例えば,「アメリカ」で検索したときに，該当文書に「米国」だけが含まれていた場合，その文書を探し出すことはできない．

そこで，文書と検索質問の両方を統一的な空間の中に配置し，その間の類似度を定義することによって似ている文書を探し出す方法であるベクトル空間法が提案されている．

ベクトル空間法では，検索に使用する索引語を T 個とし，それぞれの索引語 t_i に対して，ベクトル V_i を対応させる．これらのベクトルが線形独立である場合，T 次元のベクトル空間が定義されることになり，この空間におけるすべてのベクトルは，T 個のベクトルの線形結合で表される．このようなベクトル空間において，文書 d_r を以下のように表現する．

$$d_r = \sum_{i=1}^{T} a_i^r V_i$$

ここで，係数 a_i^r は，文書 d_r における索引語 t_i に対する重みであり，最も単純な場合，文書 d_r に索引語 t_i が出現すれば 1，そうでなければ 0 とする．そして，検索質問も同じように考え，ベクトル V_i を用いて表現する．

$$q = \sum_{i=1}^{T} q_i V_i$$

ここで，係数 q_i は，a_i^r と同様に，質問 q_i に索引語 t_i が出現すれば 1，そうでなければ 0 とする．

そして，二つのベクトル d_r と q の類似度を比較し，検索質問に類似しているものから順番に出力する．この類似度の判定には，多くの場合，ベクトルの内積が用いられる．

$$\mathrm{sim}(d_r, q) = d_r \cdot q = \sum_{i,j=1}^{T} a_i^r q_j V_i \cdot V_j$$

t 個の語ベクトル V が直行していれば,

$$\mathrm{sim}(d_r, q) = \sum_{i=1}^{T} a_i^r q_i$$

となる.

2.2.4 検索システムの評価

検索システムの評価には，通常，再現率 (recall) と精度 (precision)[1]の二つの値を用いる.

再現率はユーザの求める文書 (検索要求に該当する文書) がどの程度出力されたかを示す指標で，以下のように定義される.

$$\text{再現率 } R = \frac{\text{出力された該当文書の数}}{\text{全文書中の該当文書の数}}$$

一方，精度は出力された文書中にユーザが求める文書がどの程度含まれるかを示した指標で，以下のように定義される.

$$\text{精度 } P = \frac{\text{出力された該当文書の数}}{\text{出力された文書の数}}$$

理想的には，この両者を 1 に近付けることが望ましい．しかし，これらの二つの値にはトレードオフの関係があり，再現率を上げようとすると精度が下がり，精度を上げようとすると再現率が下がるということが起こる.

3 Ruby 入門

Ruby[2] はまつもとゆきひろ氏によって開発されたオブジェクト指向スクリプト言語である.「スクリプト (script)」とは台本，脚本といった意味で，ある一定の処理を行うために作られた小さなプログラムで，場合によってはその場限りで捨てられることもある．スクリプト言語とは，そのようなスクリプトの開発に向いた言語であり，シェル・スクリプトや JavaScript，Perl などもスクリプト言語である.

スクリプト言語は，豊富な組み込み機能を持つものが多く，やりたい事を短かいプログラムで実現可能である．また，作ったプログラムをすぐに実行できるようインタプリタ型の言語として実現されている.

Ruby はオブジェクト指向言語である．オブジェクト指向とは「もの」を中心とする考え方であり，これに対して，従来の「処理」を中心とするプログラミングを手続き指向と呼ぶ．Ruby はオブジェクト指向に基づいてプログラミングをするのに適しているだけでなく，オブジェクト指向を特別意識しなくてもプログラミングができる．よって，オブジェクト指向を学んでから Ruby を使用するのではなく，Ruby を使用する内にオブジェクト指向のメリットを体験し，その利点を学べばよい．その意味で，Ruby はオブジェクト指向の学習に適したプログラム言語といえる.

もっとも重要なことは,「Ruby はプログラミングを楽しくする言語である」ということである．従来の言語では，機械の都合に合わせての面倒な作業や同じことの繰り返しが必

[1]適合率とも呼ばれる.
[2]http://www.ruby-lang.org/ja/

要であり，プログラミングの創造性を妨げていたが，そうした部分は言語に任せて，プログラミングの本質的な部分に集中しようというのが Ruby の設計思想である．

3.1 Ruby の実行

Ruby のプログラムを実行するには，プログラムが書かれたファイル (例えば hoge.rb) を用意し，

```
% ruby hoge.rb
```

のように入力すれば良い．ここで，% はシェルのプロンプトである．また，UNIX であれば，まずファイルの先頭に

```
#! /opt/NSUG/bin/ruby
```

という行を追加する．ここで，/opt/NSUG/bin/ruby は ruby がインストールされている場所を示すので，環境によって適宜変更する．その後，

```
% chmod a+x hoge.rb
```

のようにして，ファイルを実行可能にすれば，hoge.rb を直接実行することが可能となる．

3.2 Ruby の特徴

この章では，Ruby の特徴をいくつかのサンプル・プログラムを示しながら説明する．Ruby には以下のような特徴がある．

1. オブジェクト指向言語である．
2. 変数に型がない．
3. 代入はコピーではない．ただし，普通は気にする必要はない．
4. メモリ管理が不要．
5. 文字列操作に適している．
6. 日本語対応

3.2.1 オブジェクト指向言語

Ruby はオブジェクト指向言語であり，Ruby の手続きはすべて，なんらかのオブジェクトのメソッド呼び出しである．Ruby では，あるオブジェクト obj のメソッドの呼び出しは以下のよう行われる．

```
obj.method()
obj.method(arg1)
obj.method(arg1, arg2)
```

なお，曖昧でなければメソッド呼び出しの括弧は省略可能である．また，メソッドが返す値もまたオブジェクトであるから，

```
obj.method1().method2().method3()
```

のように，続けてメソッド呼び出しを行うことが可能である．

しかし，Ruby ではオブジェクト指向を意識しなくてもプログラムが書ける．

まず，最初のプログラムとして以下を見てほしい．

```
print "Hello World\n"
```

上記は，""で囲まれた文字列を表示するプログラムであるが，このプログラムでは，メソッド呼び出しを行っているようには見えず，print はまるで関数のように見える．これは，print ではメソッドを実行する対象となるオブジェクト (レシーバと呼ばれる) が省略されているからである[3]．Ruby では，厳密な意味では関数は存在しないが，すべてのオブジェクトから呼び出すことが可能なメソッドが存在し，それは他の言語における関数と同じように使うことができる．

また，ユーザがメソッドを定義する際にも，クラスやモジュールの定義の外側で定義すれば，そのメソッドは関数のように扱える．さらに，クラスやメソッドについても，必要なものの多くは最初から用意されている．よって，Ruby では，オブジェクト指向を意識しなくても，プログラミングが可能である．

3.2.2 変数に型がない

Ruby の変数には 4 つの種類があり，これは名前で区別される．なお，Ruby では変数を使用する前にあらかじめ宣言をする必要はない．厳密には，ローカル変数の場合，最初の代入が変数の宣言になる．

- ローカル変数 (小文字，または _で始まる)
- グローバル変数 ($ で始まる)
- インスタンス変数 (@で始まる)
- クラス変数 (@@で始まる)
- 定数 (大文字で始まる)

Ruby では，データ (オブジェクト) に型はあるが，変数に型はない．よって，以下のような代入を続けて行なってもエラーは発生しない．

```
a = 10
a = "cat"
a = ["cat","dog"]
```

[3] こうした関数的メソッドでレシーバが省略されるのは，レシーバが何であっても処理の結果が変化しないからである．

Ruby の変数は，値が入る箱ではなく，オブジェクトに貼られる名札であると考えると分かりやすい．よって，a = "cat" とすると，まず "cat"という文字列オブジェクトが作成され，それを指し示すものとして，a と書かれた名札が用意される．この次に a = ["cat","dog"] とすると，["cat","dog"] という配列オブジェクトが作成され，a と書かれた名札は，こちらに貼り直される．変数はポインタのようなものであると考えると分かりやすいかもしれない．

3.2.3 代入とコピー

Ruby における代入とは，左辺の変数に右辺の計算結果のオブジェクトを指し示すように指示するものである．よって，代入によってオブジェクトのコピーは起こらない．

そうなると，b = a とした後に，a が指すオブジェクトの内容を変更すると， b の指す内容も変化してしまうことになり都合が悪い．そのため，a = b.clone のようにオブジェクトをコピーするメソッドも用意されている．しかし，実際にはそうした点を意識する必要はあまりない．

例えば，

```
a = 10
b = a
a = a + 10
print "a = ", a, " b = ", b
```

としたとき，出力は a = 20 b = 10 となり，a と b の値は等しくはならない．これは，a = a + 10 としたときに，a + 10 の計算結果である 20 というオブジェクトが新たに作成され，a はこの新しいオブジェクトを指すことになるからである．

このように，Ruby では多くのメソッドが必要に応じてコピーを自動的に作成するため，明示的なコピーをする必要はそれほどない．ただし，コピーではなく元の内容を変更するメソッドもあり，そうしたメソッドは**破壊的なメソッド**と呼ばれ，その多くは名前の末尾に!が付加されている．

例えば，tr は文字の置換を行うメソッドであり，s.tr([a-z],[A-Z])[4]とすると，s の中の小文字のアルファベットをすべて大文字にしたオブジェクトが得られるが，s.tr!([a-z],[A-Z]) とすると変数 s の内容自体が変更される．

また，Ruby では以下のような多重代入も可能である．

```
a,b = 0,1
a,b = 1, 2, 3       # a=1; b=2; 3 は捨てられる
a,b,c = 1, 2        # a=1; b=2; c=nil
a,b = ary           # 配列の最初の 2 要素を取り出す
a, = ary            # 代入先がひとつの多重代入．配列の先頭の要素を得る
```

[4]ちなみに Ruby では同じことをするメソッド upcase が用意されている．

3.2.4 メモリ管理が不要

Ruby ではオブジェクトの領域の割り当てやその管理はインタプリタが行うため，プログラマがメモリ管理について気にする必要がない．また，参照されなくなったオブジェクトは，ガーベージ・コレクションという機能で自動的に回収されるため，オブジェクトの解放についても心配する必要はない．

3.2.5 文字列操作に適している

Ruby では標準で様々なメソッドが用意されており，さらに正規表現やハッシュも扱えるため，ファイルや文字列を操作するプログラムが簡単に書ける．例えば，UNIX の grep は以下の 4 行で記述できる．

```
pat = Regexp.new(ARGV.shift)
while gets
  print $_ if $_ =~ pat
end
```

同じものを C 言語で記述するには，おそらく何十行以上のプログラムが必要になるだろう．

3.2.6 日本語対応

Ruby は日本生まれであることから，最初から日本語処理を意識して作成されている．Ruby で扱える文字コードは，EUC，SJIS，UTF-8，NONE(日本語を意識しない) の 4 種類である．いずれを用いるかの指定には複数の方法があるが，プログラムの先頭に

```
#! /opt/NSUG/bin/ruby -Ks
```

のように-K オプションで指定するのが簡単である．実際には最初の 1 文字だけを見て決定されるので，上記の場合は SJIS として処理される．日本語つまり多バイトの変数やメソッドも使うことが可能であるが，推奨されない．それよりも正規表現が日本語に対応している点が重要である．日本語の文字コードでは，場合によって，ある文字の 2 バイト目と次の文字の 1 バイト目が，まったく別の文字とマッチすることがある．Ruby の正規表現ルーチンは多バイト文字も 1 文字として認識するので，そうした問題は起きない．

一方で，正規表現以外の文字列は基本的には日本語に対応していない．日本語に対応した処理させるためには jcode ライブラリを requre する．これにより，String クラスのメソッドが置き換えられ，以下の例ではいわゆる半角英数を全角英数に置き換えることが可能となる．

```
require "jcode"
print "16th Lincoln".tr("0-9a-zA-Z","０-９ａ-ｚＡ-Ｚ"),"\n"
```

3.3 クラス

すでに何種類かのオブジェクトの例を挙げたが，こうしたオブジェクトの種類はクラスと呼ばれる．あらゆるオブジェクトはなんらかのクラス[5]に属している．また，あるオブジェクトがあるクラスに属していることを強調する意味でインスタンスと呼ぶこともある．

また，オブジェクト指向言語である Ruby では，クラス間に階層関係があり，下位のクラス (サブクラス) は上位のクラス (スーパークラス) の機能を継承している．また継承したメソッドを必要に応じて再定義する，いわゆるオーバライドも可能である．なお，Ruby では多重継承はできないが，この点については Mix-in という機能で補うことが可能である．クラス自体をユーザが定義することも可能であるが，その方法はここでは説明しない．実際，Ruby では自分でクラスを定義しなくとも，用意されている組み込みクラスを利用するだけでも，それなりのプログラムが書ける[6]．

ここでは，Ruby の組み込みクラスのうち，主なものについて簡単に説明する．なお，Ruby ではクラスの名前は大文字で始めることになっている．

3.3.1 Fixnum，Bignum，Float

`Fixnum` は，計算機のポインタのサイズに収まる長さの固定長整数のクラスであり，多くの場合，31 ビット幅である．それよりも大きな整数は `Bignum` というクラスに自動的に拡張される．`Bignum` はメモリの許す限り，いくらでも大きな整数を扱うことが可能である．なお，`Fixnum`，`Bignum` とも，`Integer` のサブクラスである．

`Float` は，C 言語の `double` に相当する浮動小数点数のクラスで，その精度は多くの場合せいぜい 15 桁である．また，`Integer` と `Float` は，ともに数値の抽象クラスである `Numeric` のサブクラスである．

3.3.2 String

`String` は，文字列を表すクラスである．Ruby では文字列のリテラル[7]は""もしくは’’で囲んで表現される．例えば，`"Hello World\n"`は文字列オブジェクトである．ここで，""と’’の違いであるが，""で囲んだ部分では，バックスラッシュ記法の解釈 (例えば`"\n"` は改行である) が行われるが，’’で囲まれた部分では，’\\’ と’\’’ 以外のバックスラッシュ記法の解釈は行われない．

Ruby では文字列クラスには，文字列の連結，置換，比較などの豊富なメソッドが用意されている．

3.3.3 Regexp

`Regexp` は，正規表現を表すクラスであり，そのリテラルは`//`で囲んだ形式で表現される．正規表現は，文字列のパターンの表現方法であり，その高い記述能力により，テキスト処理を簡単にすることが可能である．なお，正規表現にはさまざまな方言があるが，Ruby の正規表現は Perl のものとほぼ同じである．例えば，`/^[Rr]uby [0-9]+.?[0-9]*/` は，

[5]ちなみに，Ruby ではクラスも Class というクラスに属するオブジェクトである．
[6]分かりやすいかどうかは別にして．
[7]Ruby のプログラムの中に直接記述できる値のこと．

先頭が Ruby もしくは ruby でその後に数字が続くパターンを示す正規表現である．具体的な正規表現記号 (メタ文字) については各自で調べよ．

正規表現とのマッチは =~というメソッドで行う．最後にマッチした部分は，組み込み変数$&に格納される．また，正規表現は文字列の置換にも役立つ．

3.3.4 IO，File

Ruby では，基本的な入出力は関数的メソッド gets や print/printf で行うことが可能であるが，個別にファイルを開いて読み書きするための IO クラスが用意されている．

IO オブジェクトを作る一般的な方法は，関数 open を用いて

```
file = open("sample.txt","r")
file2 = open("output.txt","w")
```

のように記述することである．ファイルに対する入出力は，IO クラスのサブクラスである File クラスを使って行う．

UNIX のいわゆるフィルタ・プログラムは，ほとんどが引数として指定したファイル群から順に入力を受けとる．Ruby では，この引数として指定されたファイルを仮想的に連結したものを組み込み定数 ARGF(または組み込み変数$<) で参照可能である．また，関数的メソッド get は，ARGF に対するメソッド呼出しを省略したものである．

サンプル・プログラムとして，UNIX の cat と同等の機能を実現するプログラムを以下に挙げる．

```
while line = ARGF.gets  # while line = gets   としても良い
  print line
end
```

3.3.5 Array

配列とはオブジェクトを順番に並べてものであり，これもまた一つのオブジェクトである．配列を表すクラスが Array であり，そのリテラルは [] を使って，例えば以下のように表すことができる．

```
array = ["cat", "dog", "rat"]
```

配列の各要素は，C 言語と同じように参照することが可能であり，例えば先頭の要素を参照する場合は，array[0] とする．この例から分かるように，配列の添字は 0 から始まる．なお，これは array[0] という変数があるのではなく，Array クラスのオブジェクトに [] というメソッドを，0 という引数で呼び出しているのである．

Ruby では配列の要素は任意のオブジェクトを使用可能である．よって，例えば以下のような配列も可能である．

```
array = [12, "123",[["cat", "dog"], "rat"]]
```

この例では，第 1 要素が整数，第 2 要素が文字列，第 3 要素が配列となっている．ちな

みに，"dog"を参照するためには，array[2][0][1] とする．

配列には，先頭を取り出したり，要素のソートするメソッドや，配列をスタックとして扱うためのプッシュとポップなど，多彩なメソッドが用意されている．

3.3.6 Hash

Hash はハッシュ・オブジェクトを表すクラスであり，順序付けられていないオブジェクトから別のオブジェクトを参照することを可能とする．簡単に言えば，ハッシュは数字以外のもので参照にすることが可能な配列であり，連想配列 (associative array) と呼ばれることもある．ハッシュのリテラルは{}を使って以下のように表す．

```
hash = {"cat" => "猫", "dog" => "犬", "rat" => "鼠"}
```

この場合，"cat"がキー，"猫"が値と呼ばれ，ハッシュのキーから値へのアクセスは hash["cat"] のように [] というメソッドを用いる．ハッシュは文字列処理においては強力な機能であり，この例を見れば分かるように辞書も簡単に実現できる．

なお，この例ではキー，値とも文字列オブジェクトであったが，Ruby ではハッシュのキーおよび値に任意のオブジェクトを使用できる．なお，キーに対応する値がなかったときには，通常は nil が返る[8]．

3.4 モジュール

モジュールは，特定のメソッドや定数などの機能をまとめた単位で，クラスと良く似た働きをする．ただし，クラスと違い継承に係わることはできない．その代わり，インクルードという方法で他のクラスやモジュールにその機能を付加する．浮動小数点演算をサポートする Math モジュールなどがある．

3.5 Ruby の制御構造

3.5.1 文

Ruby の文は改行か; で区切られる．複数の文が並んでいるときには，その文を先頭から順に実行する．また，使われるかどうかは別にして，すべての文に値がある．#から行末まではコメントである．

3.5.2 if 文，unless 文

Ruby では，条件分岐には，if 文を使用する．if 文の形式は以下の通り．

[8] Ruby では nil と 0 は別物．

```
if a1 [then]
  b1
elsif a2
  b2
else
  b3
end
```

これは，もし，a1 が真ならば b1 を実行し，そうでないとき，もし a2 が真ならば b2 を実行し，a1 も a2 も真でないときに b3 を実行する．

[then] とあるのは，省略可能であることを示している．elsif 節は複数記述可能であり，また，elsif，else は省略可能である．なお，end は必ず必要である．

また，Ruby で「真である」とは「偽でない」ことを言う.「偽」とは，nil または false であることを言う．他の言語と違い，nil と false 以外はすべて「真」なので注意すること．例えば，0 も「真」である．

また，if の代わりに unless を使用すると条件の意味が逆になる．すなわち，条件が「偽」のときに対応する文が実行される．

なお，単一の式の後に if や unless を付けることができる．この場合，if の後にある式が先に評価され，その条件が成立しているときに前の式が評価される．例えば，

```
print "initialized.\n" if $DEBUG
```

とすれば，変数$DEBUG が真のときだけ，print 文が実行される．この形式を修飾子と呼び，条件の成立がそれほど重要でない場合に用いられる．

3.5.3 case 文

Ruby では，一つの値に対する条件分岐を行うためにの case 文も用意されている．case 文は when 節と組み合わせて使用する．その一例として，以下に，各月の日数を出力するプログラムを示す．

```
def days_of_month(m)
  case m
  when 2 [then]
    print "28 or 29\n"
  when 4,6,9,11 [then]
    print "30\n"
  else
    print "31\n"
  end
end
days_of_month(ARGV[0].to_i)
```

3.5.4 while 文，until 文

while 文はループを表現し，以下の場合，式 1 が真である限り，文を繰り返し実行する．

```
while 式 1 [do]
  文
end
```

なお，while の代わりに untile を使用すると，式 1 が偽である限り文を実行する．また，while 文，untile 文とも修飾子形式で利用可能である．

```
式 1 while 式 2
```

この場合も，まず式 2 が評価され，それが真である場合に式 1 が評価される．ただし，

```
begin
 文
end while 式 1
```

の場合には，まず文を 1 回実行してから式 1 を評価し，それが真である限り文の実行を繰り返す．これは C 言語の do … while(); 文に相当する．

3.5.5 for 文

while 文よりも高度なループには，for 文を使用する．なお，for 文の代わりに，後述するイテレータの形式で記述されることも多い．for 文の形式は以下のようになる．

```
for var in obj [do]
 文
end
```

ここで，obj は何らかの集合からなるオブジェクトであり，その各要素を変数 var に代入しながら文を繰り返す．例えば，配列の各要素を表示するプログラムは以下のようになる．

```
for i in [1,2,3]
  print i, "\n"
end
```

同じ働きをするプログラムを while 文を使用して書くと以下のようになる．

```
ary = [1,2,3]
i = 0
while i < ary.size
  print ary[i],"\n"
  i = i + 1
end
```

while 文を使用した場合と比較すると，for 文を使用した場合には，変数の初期化や次の要素の取り出し，終了条件の判定などが不要になることが分かる．このように，細かい指定を言語に任せ処理の本質に集中することを「ループの抽象化」と呼ぶ．実際には，オブジェクトからの各要素の取り出しは each というメソッドによって実現されているので，自分でオブジェクトを定義する場合にも，each メソッドを定義することで for 文での挙動を自由に定義できる．

3.6 イテレータ

イテレータ (iterator)[9]はメソッドの一種であり，もともとはループの抽象化のために使われた．Ruby ではメソッドを呼び出すときに引数とは別にブロックと呼ばれる「文のまとまり」を渡すことができる．イテレータとは，こうしたブロックを呼び出すメソッドのことである[10]．ブロックは do |var|… end，もしくは，{|var|…}のように変数と文から記述される．| |の中の変数は省略可能である．

ブロックを付けたメソッド呼び出しの具体的な例を挙げる．

```
[1,2,3].each do |i|
  print i, "\n"
end
```

この例から分かる通り，for 文は，メソッド each をブロック付きで呼び出しているのと同等の動作である．

他にも様々なイテレータが用意されている．また，ブロックは繰り返し以外でも，例えば sort メソッドのソート基準となる手続きを渡す場合などにも使われる．

```
ary = [[1,"orange"],[2,"kiwi"],[3,"apple"],[4,"lemon"]]
ary.sort{|a,b| b <=> a}        # 降順ソート
ary.sort{|a,b| a[1] <=> b[1]} # 第２項でソート
```

なお，ループを制御するメソッドとしては，each の他に回数を指定する times や，無限ループを起こす loop などがある．

3.7 break，next，redo

ループから脱出する命令として，break，next，redo がある．

break は，その時点でループ処理全体を終了し，ブロックを抜ける．

next は，C 言語の continue に相当する命令であり，現在実行中のブロックを最後までスキップする．例えば for ループの中で next を使用すると，その時点でブロックを終了し，次のオブジェクトを変数に代入してループを続ける．

redo は，そのブロックの実行を先頭からやり直す命令である．next と違い．for ループの例では，次のオブジェクトを変数に代入せず，今のオブジェクトのまま再実行する．なお，redo は使い方を間違えると無限ループを起こすので注意が必要である．

[9]イテレータとは「繰り返すもの」という意味である．

[10]ループになるとは限らないため，最近はイテレータという呼び方を避けるようになっている．

表 3: 組み込み変数/定数 (抜粋)

組み込み変数/定数	説明
$_	現在のスコープで最後に gets などで読み込んだ文字列
$~	現在のスコープでの最後のパターンマッチに関するデータ
$&	現在のスコープでの最後のパターンマッチでマッチした文字列
$n($n = 1, 2, \ldots$)	最後のパターンマッチで n 番目の括弧にマッチした値
$/	入力レコードの区切 (デフォルトは"\n")
$\	出力レコードの区切 (デフォルトは nil)
$<	コマンドライン引数あるいは標準入力の仮想ファイルハンドル
$stdin	標準入力
$stdout	標準出力
$stderr	標準エラー出力
TRUE	真
FALSE	偽
NIL	nil
ARGF	コマンドライン引数あるいは標準入力の仮想ファイルハンドル, $<と同じ
ARGV	ruby スクリプトに与えられたコマンドライン引数の配列

3.8 擬似変数

Ruby には擬似変数と呼ばれる特別な変数が存在する．これらの変数に代入を行うことはできない．self は現在の実行の実行主体を表す変数である．true，false，nil はローカル変数の外見をした定数．true は真を，false は偽を表し，nil は初期化されていないことを示す特別な値である．なお，条件判断の際には，false および nil が偽，それ以外はすべて真と判定される[11]ので，true は真の代表値にすぎない．なお，これらの擬似変数と同じ値をもつ定数 TRUE，FALSE，NIL も定義されている．

また Ruby では，真偽値を返すメソッドの名前は，empty? のように，末尾に?を付けるという慣習がある．

3.9 組み込み変数

Ruby では，最初から用意されている組み込み変数がある．その多くはグローバル変数であるが，$_，$~，$1 などはローカル変数である．また，$1 には代入ができないなど，通常の変数とは異なる働きをする物があるので注意すること．表 3 に主要な物を挙げる．

3.10 メソッドの定義

メソッドの定義は，def を用いる．クラスやモジュールの性質は，どのようなメソッドをもっているかによって決まる．また，クラスやモジュールの定義の外側でメソッドを定

[11]繰り返しになるが，0 も真である．

```
 1: freq = Hash.new(0)
 2: while gets
 3:   while sub!(/\w+/, '')
 4:     word = $&
 5:     freq[word.downcase] += 1
 6:   end
 7: end
 8:
 9: for word in freq.keys.sort{|x,y| freq[y] <=> freq[x]}
10:   printf("%d\t%s\n", freq[word], word)
11: end
```

図 2: サンプル・プログラム—英単語の数を数える

義すればそのメソッドは関数のように使える.

```
def f(x, y)
  x * x + y
end
```

メソッドの返り値は実行時に最後に評価した式の値である．また，return 文を用いることで，実行を中断して直接戻り値を返すことも可能である．なお，Ruby ではメソッドの定義も実行文であるから，def によりメソッドが定義される前に呼び出すとエラーが発生する.

3.11 Ruby による例題

例として，英単語の出現頻度を調べるプログラムを図 2 に示す.

1 行目ではハッシュ変数 freq を宣言している．その際に，デフォルト値を nil ではなく，0 にしている．2 行目はコマンドライン引数もしくは標準入力から 1 行ずつ読み込むループである．ファイルから入力が読めなくなったときにループを終了する．3 行目は while $_.sub!(/\w+/, '') と同義であり，正規表現を利用して，ファイルから読みこんだ 1 行から単語を切り出している．以後は 1 語ごとのイテレータであり，その行から単語を読めなくなったときにループを終了する．4 行目では，正規表現にマッチした部分が組み込み変数$& に入っているので，それを変数 word に代入している．なお，3 行目のメソッド sub!は破壊的なメソッドであり，実行のたびに$_ の中の単語が空列に置き換えられていく．5 行目は，word の内容を小文字にして，対応するハッシュの値を 1 増やしている．9 行目からは出力だが，その前にハッシュfreq のキーを配列に変更し，それを出現回数で降順ソートしている．その順番に従って，変数 word に値が代入されループを実行する．ループ中ではハッシュの内容を表示している.

なお，このプログラムとほぼ同じ物が /pub1/jikken/IR/ にあるので，それを利用しても良い．他にもサンプル・プログラムがあるので，適宜参照せよ.

4 実験

以下の課題に出てくるプログラムは，すべて Ruby を用いて作成せよ．なお，Ruby でプログラムが記述できない場合は，C 言語を用いても良い．

4.1 N グラムの作成

日本語および英語の N グラム統計を求めるプログラムを作成せよ．なお，英語の場合は単語 N グラムを，日本語の場合は文字 N グラムとせよ．N は引数として与え，(メモリが許す限り) 任意の数を扱えるようにせよ．また，英語の場合は，大文字と小文字の区別はしないものとする．

応用課題 1：語形変化への対応

英語 N グラムを作成する場合に，名詞の複数形や，動詞の三人称単数現在形や過去形などをまとめてカウントするようにプログラムを改造せよ．

4.2 KWIC の作成

与えられた入力語に対する KWIC を出力するプログラムを作成せよ．日本語と英語の両方に適用できるようにする．もしも，日本語と英語で対応を変える必要がある場合は，その点をレポートで明記すること．余裕があれば，キーワードの後に来る語でソートや，キーワードの前に来る語でソートする機能も付加せよ．

応用課題 2：GUI 化

Ruby には，Ruby/Tk などの GUI ライブラリが何種類か用意されており，それを利用することで GUI プログラミングが可能である．適当なライブラリを利用して，KWIC のプログラムを GUI 化せよ．

4.3 索引語の自動抽出

/pub1/jikken/IR/ の下のデータに対して，tf.idf 法を用いて索引語を決定せよ．なお，データの説明は `readme.txt` に記述されているので，それを参照せよ．

4.4 キーワード検索システムの作成

上記で決定した索引語を用いるキーワード検索型の検索システムを作成せよ．キーワードのアンド検索，オア検索，ノット検索の機能を備えたものとせよ．

また，いくつかのキーワードを自分で決め，その場合の再現率と精度を調べよ．その場合，再現率と精度の数値だけでなく，正解データと検索システムの出力結果も明示せよ．

応用課題 3：ベクトル空間法の導入

ベクトル空間法を用いて，キーワードと完全一致しない文書も検索できるようにせよ．

応用課題 4：検索の高速化

Suffix Array などを用いて，全文検索の高速化を実現せよ．なお，Suffix Array の部分は，自分で作成しても良いし，他のライブラリを利用しても良い．

5　レポートについて

レポートでは，各実験について

- 実験の方法
- プログラムの処理手順，プログラム作成上の工夫
- 実験で得られた結果
- 結果に対する考察

を記述せよ．

実験の方法については，情報検索について各自で調べて記述すること．作成したプログラムは，レポートに添付する必要はないが，説明に必要ならばリストを掲載しても構わない．ただし，いずれの場合も，レポートとは別に E-mail で *yasuhiro@is.nagoya-u.ac.jp* に提出すること．

また，本実験に関する感想・要望などを最後に付記せよ．

参考文献

[1] まつもとゆきひろ, 石塚圭樹: オブジェクト指向スクリプト言語 Ruby, アスキー出版局, 1999.

[2] 高橋征義，後藤裕蔵: たのしい Ruby — Ruby ではじめる気軽なプログラミング, ソフトバンク パブリッシング株式会社, 2002.

[3] 青木峰郎，後藤裕蔵，高橋征義: Ruby レシピブック 268 の技，ソフトバンク パブリッシング株式会社, 2004.

[4] オブジェクト指向スクリプト言語 Ruby リファレンスマニュアル, `http://www.ruby-lang.org/ja/man/`

[5] 長尾真，黒橋禎夫，佐藤理史， 池原悟，中野洋：言語情報処理, 岩波書店, 1998.