Visual C++によるXMLプログラミング

石田基広

XML Programming with Visual C++

Motohiro ISHIDA

Abstract

XML is a markup language for documents containing structured information. In terms of language processing, it means that one can input any linguistic information, for example, syntactic structures or semantic values, in any text in the form of xml tags. Such tagged texts are easily processed for any purpose.

This paper shows how to tag a text automatically in Microsoft's Visual C++ environment.

0 序

本稿の執筆目的は、XML/XSLによるアノテーション（annotation）プログラムを紹介することにある。今後、言語データのデジタル処理にあたっては、XML/XSLプログラミングによるアノテーションが必須の条件になるのは間違いない。[1]さらにXMLによってアノテートされた言語データは、自然言語処理（機械翻訳や情報抽出、知識処理など）のさまざまな局面において標準的な文書形式になる可能性が強い。[2]従って言語研究者は、自らの研究（応用）目的に応じて、自由にアノテーションを行う技法を身につけておく必要がある。今後、そのための汎用的なソフトが公開されていくであろうが、現状では、研究者自らがプログラミングするよりほかにない。

そこで本稿では、生言語データ（原テキスト）をトークンごとに分解し、それらのトークンの品詞を判定する単純なアルゴリズムとともに、解析結果の品詞情報をXML形式のタグとしてアノテーションする技法について、実際のプログラミング作業の中で紹介していく。作成に当たってはMason[3]によるJavaプログラミング技法を参考にしたが、開発環境をVisual C++に変更したため、相応の改変を試みている。またGUI化にはWindows APIを活用しているが、言語解析部分の実装に関しては、C++の標準STLに沿っている。[4]

その前に、言語データにXMLタグを付与することの意義について、以下で

---

1 XMLについては、そのプロセッサ部分にあたるXSLの勧告が、XSLTに遅れること、ようやく2002年10月に発表された。
http : //www.w 3.org/TR/xsl/
http : //www.w 3 c.org/Style/XSL/

2 Pazienza, Maria Teresa, 1997, Information Extraction: A Multidisciplinary Approach to an Emerging Information Technology, Springer Verlag.
Pazienza, Maria Teresa, 1999, Information Extraction: Towards Scalable, Adaptable Systems, Springer Verlag.

3 Mason, Oliver, 2000, Programming for Corpus Linguistics: How to do Text Analysis with Java, Edinburgh. 以下ではMasonと略記する。なおコーパスを利用した言語研究については以下を参照。
McEnery, Tony & Wilson, Andrew, 1996, Corpus Linguistics, Edinburgh. 同じくWilsonと略記。

4 C言語による文字列処理に関しては、以下を主に参照した。
林晴比古、1999、改定新C言語入門　応用編、ソフトバンク社。
柏原正三、2000、標準C++の基礎知識　実践編、アスキー。

もう一度考察する。

1 XMLとセマンティックWeb, GDA

情報抽出Information Extractionの実用化の提案のひとつとして、“セマンティックWeb (Semantic Web)”が注目を浴びている。ここでは、荻野 et al. に挙げられている例を援用して、ごく簡単に説明する。[5]

たとえば火曜日の朝になって歯が痛み出した徳島市民がInternetで「藤沢市内で火曜日に診察している歯医者」を検索する場合を考えてみる。この場合、「徳島」、「歯科」、「診察日」、「火曜」などのキーワードがすぐに思い浮かぶ。しかし仮に徳島市内で開業している歯科医X氏のホームページがあったとしても、診察日の指定は「月曜から金曜」あるいは「月～金」などと記述されているかもしれない。このとき、X氏の歯科医院はヒットしないことになってしまう。基本的な検索システムでは「火曜」という検索指定は、「火曜」という文字列がHTML文書に明記されていて、初めて検索対象となる。

しかしながら「火曜」という指定は、意味的には「月曜日から金曜日」に含まれるものであるから、これを処理する工夫が必要となる（大手の検索エンジンの多くは、独自にシーソラスを用意するなどして対応している)。

ひとつの方法としては、その意味的属性をメタデータとして付記すれば良い。たとえば「月曜から金曜」というのは、「平日」と同義であると定義し、メタデータとして「平日」を付記しておく。さらに「平日」は「月、火、水、木、金」からなるという情報が、何らかの方法でInternet上に一般的に与えられていれば、「火曜」を検索することにより、「月曜から金曜」を検索候補に含めることが可能になる。

この点でXMLは、自由にタグ階層を設計できる点で拡張に富んでいる。ここで必要となるのは、個人なり特定の組織なりが独自に設計したメタデータを、第三者が利用する方法を用意することである。これには二つの方法が考えられる。一つは、この世界にあるすべての事象について、それぞれの意味内容を明らかにし、同時にそれら相互の関係（類義、あるいは上位語、下位語などの関係など）を定義してしまうことである。しかし意味の関係性については、言語学や人工知能などの分野でも古くから研究が行われているが、それらの成

---

5 荻野達也 et alii, 2002, “セマンティックWebとは”, 情報処理7号, 情報処理学会。またhttp://www.net.intap.or.jp/INTAP/s-web/をも参照のこと。

果を総合して統一的な規格を作り上げることは不可能であろう。

そこで二つ目の方法としては、上位階層となるメタデータ階層を定義し、これのみを国際規格とするのが考えられる。この場合、個人や組織などの下位のレベルで独自に必要となるメタデータは、上位階層の定義を「継承」する方法で段階的に構築していく。継承関係がアノテーションによって明確化されていれば、「末端の」テキスト作者が作成した独自タグでも、その継承関係を上へとたどることにより、その意味を解釈する可能性が与えられることになる。[6] セマンティック Web は、このような仕組みを国際的に普及させようとするものである。

ところで言語情報にメタデータを与えることは、言語学の立場からすれば決して目新しい発想ではない。むしろ最近の形式的な言語理論の多くに共通する傾向である。たとえば主辞駆動句構造文法（HPSG）や、あるいはディスコース意味論（DRS）や動的意味論（Dynamic Semantics）では、メタデータの設計がその理論の中核となっている。[7] また緒方が指摘するように、XML の記述スタイルは、たとえばディスコース表示理論と非常に相性が良い。[8]

さらに電総研の橋田らは Global Document Annotation （GDA）を提唱している。[9] これは多くの言語に共通する統語的・意味的・語用論的属性をタグ形式でテキスト内部にアノテーションするためのプラットフォームである。この提案が標準化されれば、言語テキストの機械処理は一気に実用化が進むものと思われる。

XML とセマンティック Web、さらに GDA は、今後、実装を意図した言語理論の展開にとってもきわめて重要な意味をもつものと思われる。

---

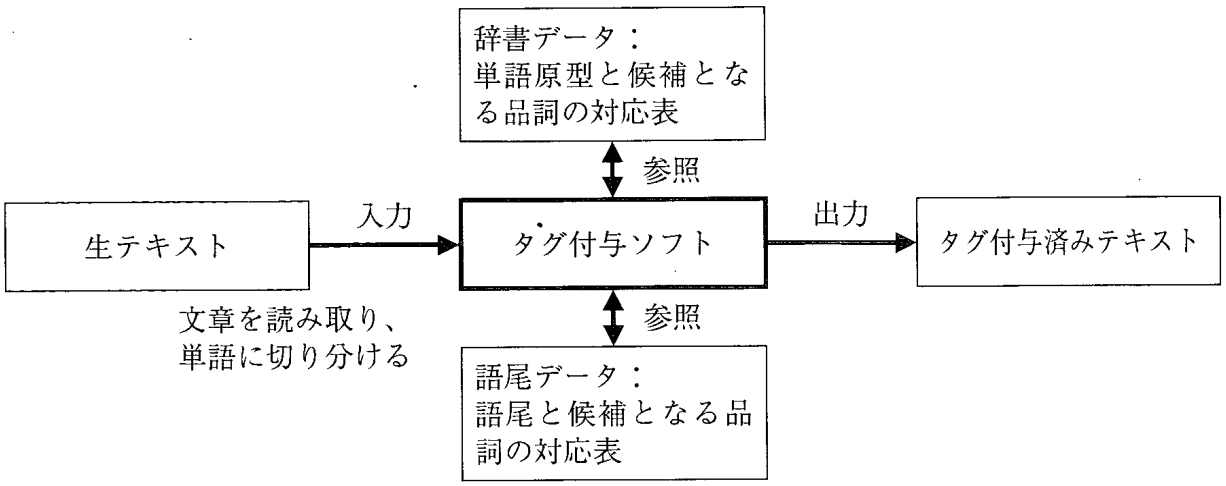
6 プログラミングで言えばオブジェクト指向(OOP)の設計。

7 たとえば、HPSG における「意味素性」や「統語素性」、あるいは DRS で重要な位置を占めるディスコース・リフェラントなど。
Sag, Ivan A., Wasow, Thomas, 1999, Syntactic Theory, CSLI.
Kamp, Hans, Reyle, Uwe, 1993, From Discourse to Logic, Kluwer.
Chierchia, Gennaro, 1995, Dynamics of Meaning, Chicago Univ. Press.

8 緒方典裕, 2002, *Dynamic Semantics meets XML and IE*, 言語処理学会第 8 回年次大会発表論文集, pp. 527－530.

9 http : //www.i-content.org/gda/

1　タグ付与プログラミング



以下、本稿で作成するプログラムのデータ処理の流れを図示する。

品詞を判定し、その情報をタグに付与するのがこのプログラミングの目的である。生テキストから単語を取り出し、その品詞を判定しては、その単語にタグをふるという方法で品詞情報を付記するわけである。

英語の場合、単語はスペースで区切られているから、単語の切り出しそのものは単純な作業である。切り出した各単語について、別に用意する辞書ファイルを参照し、品詞を判定する。辞書ファイルは単語の原型と品詞が対になったデータバンクである。

ただし生テキストに現れる単語、例えば動詞、名詞、形容詞は活用している可能性があるため、生テキストの語形のままでは辞書ファイルに見つからない場合もある。そのような場合、まず語形を原型に戻したうえで、もう一度辞書ファイルを参照する必要がある。ただし、本稿では単語に“品詞”情報だけを付与するプログラムを目的としている。またここで対象言語とする英語は、語末からも、ある程度その品詞が推測できる。そこで辞書ファイルとは別に、語尾（suffix）と、可能な品詞候補を対応付けたファイルを用意し、辞書ファイルに登録されていない単語については、語尾ファイルとの照合によって品詞を判定する方法を取るものとする。

次に品詞を判定する上で問題となるのは、例えばlikeのように複数の品詞属性を持つ語形は、それが利用される文脈によって品詞判定がなされなければならない。判定には、ある程度演繹的な方法も可能であるが、しかし精度はまだ不十分である。そこで多くの場合帰納的（統計的）手法が用いられている。すなわち分析対象の単語に複数の品詞候補がある場合、一般的な言語データを

収録したコーパスから得られるデータを別途参照して、確率的に一番確からしい品詞を取り出す方法である。これには、1）生テキストの文脈とは無関係に判定する方法、つまりその単語に可能な品詞分類のうち、一般的に英語においてもっともよく利用される品詞を優先する方法と、2）生テキストに現れる文脈を重視し、その単語の前後に現れている単語の品詞を参照し統計的に判定する方法との二つがある。

後者2）の方法は、ごく大雑把に言えば、例えば英語の定冠詞 the の直後に現れる単語の辞書データに、名詞と動詞の二つ品詞情報がある場合、名詞を優先する戦略である。ただし単語判定の確からしさを高めるため、今の例を挙げれば、直前の語だけではなく、直後の語の情報も参照する。例えば the apple is sweet という文中の apple の品詞を判定するに当たり、直前の the だけではなく、直後の is の品詞も参考にする。

ただし本書では品詞判定アルゴリズムに焦点があるではなく、その判定結果をタグに組み込む処理技法を紹介するのが目的である。そこで Mason にならい、品詞判定には Wilson の提示するスタティックな統計データ表をそのままプロミングに組み込む方法をとる。以下、辞書ファイルを読み込むためのクラス定義から解説していく。

2 CLexicon クラスの実装

単語の品詞を判定するために辞書ファイルを参照する。辞書ファイルの参照作業に特化したクラス CLexicon を作成する。ここで読み込み対象となる辞書ファイルは、例えば“draw　V　N”のように、単語、区切り文字(例えば半角スペースやコロン、ピリオドなど)、品詞略語（例えば V は動詞、N は名詞）で構成されるものとする。デフォルトの区切り文字は水平タブとするが、初期化の方法次第でユーザー定義の区切り文字も扱える。

```
CLexicon::CLexicon()
{

//  コンストラクタでデフォルトの区切りを水平タブ（アスキー16進法で
    0x09）とする。m_tab は CString クラスのオブジェクトである。
  m_tab = “¥x09” ;
  }
//  デストラクタは特に実装しない。
```

```
CLexicon : : ~CLexicon() {
}
```

次に辞書ファイルからの読み込みを行う。単語と、その品詞情報が対になるように記憶しておく。それにはハッシュ形式が便利であるが、Visual C++定義のCmapStringToStringクラスを利用する。辞書ファイル読み込みとマッピングを行うsetLexicon関数を以下のように定義する。

```
void CLexicon : : setLexicon (CString m_fname)
{

//  辞書ファイルから読み取る。
  CStdioFile fin (m_fname, CFile : : modeRead) ;
  CString line ;
  while (fin. ReadString (line) != NULL){

//  一行ずつ読み取り、区切りを境に前後の文字列に分ける。デフォルトではタブが区切り文字なので、その位置を取得する。
  int place = line. Find (m_tab) ;

//  タブの位置から右と左の文字列に分ける。右がキーとなる単語（key)、左がその単語の品詞情報（value）である。
  CString key, val ;
  key = line.Left (place) ;
  val = line. Mid (place + m_tab. GetLength ()) ;

//  key と value でマッピングする。
  if (key != "")

//  m_mapSt は CmapStringToString のオブジェクトである。
  m_mapStr.SetAt (key, val) ;
  }

//  次に定義するのは、辞書ファイルから読み込み、記憶されたデータを検索する関数である。CLexicon クラスでもっとも長く重要な関数となる。
```

```
CString CLexicon : : lookUp (CString word)
{

// 検索対象となる単語は変数 word として渡される。

// はじめに渡された文字列が空でないことを確かめる。
  if (word.GetLength()>0) {

// あとで使う文字列を空文字列で初期化しておく。
  CString retval = "" ;
  CString entry = "" ;

// 渡された単語の最初の一文字を取得する。
  unsigned char first = word. GetAt (0) ;

// C++標準ライブラリの isdigit 関数を利用して、数字かどうか判定する。
  if (isdigit (first)){

// isdigit 関数が使えない場合、以下の処理を行う（アスキー16進法で0x30
から0x39まで)。
  // if (first>=0x30&&first<=0x39)

// 数字なら略号 "M" (number) を返す。
  return ("M") ;
  }
  else if (isalpha(first)){

// アルファベットかどうか判定する。C++標準ライブラリを用いるが、利
用できない場合は、以下のコードを利用すればよい。0x41から0x5a まで
がアルファベット大文字、0x61から0x7a までが小文字になる。
  // else if ((first>=0x41&&first<=0x5a)||
  (first>=0x61) && (first<=0x7a))

// アルファベットであることが分かれば、この単語が辞書リストにあるか確
```

```
認する。
  if (!m_mapStr. IsEmpty ()){

//  この単語をキーとして対応する品詞の記載がマップ内にあるならば、フラグ check を 1 にする。
  check = m_mapStr.Lookup (word, entry) ;

//  見つからなければ小文字に変換して探してみる。これは語が文頭にある場合など、単語（の一部）が何らかの特殊な文脈で大文字化されている可能性を考慮するためである。
  if (!check){
  word. MakeLower () ;
  check = m_mapStr. Lookup (word, entry) ;
  if (!check){

//  なお見つからなければ、語尾の形から品詞を判定する方法を採用する。そのために当該単語を、別途用意する CSuffix クラスの match 関数に渡す。

//  CSuffix クラスの static メンバ関数に文字列を渡す。また this で、現在利用している辞書を読み込んだ CLexikon クラスのオブジェクト自身を渡している。
  retval = CSuffix : : match (word, this) ;

//  語尾からも判定がつかなかった場合は、ここまでの処理で少なくとも文字列ではあることは判明しているので、そのアスキー値を確認して、固有名詞である可能性を検討する。
  if (retval == "")
  if ((first>=0x41)&&(first<=0x5a))

  //  大文字で始まっているなら固有名詞 "N" である
  retval = "N" ;
  }

//  さもなければ、判定不可能として、デフォルトの値を与えてしまう。デフォ
```

```
ルトとして、活用語全ての可能性“N（名詞）V（動詞）J（形容詞）”を
戻り値となるCString型変数に代入する。
  else
  retval = “N V J” ;
  }

//  以下では、検索結果を戻り値用の変数に代入している。entryにはLookup
関数による検索結果が代入されている。
  } else
  retval = entry ;
  } else
  retval = entry ;
  } else
  retval = “Neither DIGIT nor Letter” ;
  }

//  wordに何らかの文字列が送られてきたのであれば、検索結果が代入され
たretvalを返す。
  return (retval) ;
  } else

//  以下はwordが空であった場合の処理である。
  return (“word has no length”) ;
  }
```

辞書ファイルに見つからなかった単語は、語尾から品詞を判定する。ここでは最短一致ではなく、最長一致法を採用している。つまり対象とする語の語頭から文字を一文字ずつ削っていき、Suffixファイルに登録された語尾と一致する形を探す。一致する語形が見つかれば、その品詞情報を戻り値にする。品詞候補が複数ある場合もあるが、この場合統計的に判定することになる点は、辞書ファイルに登録されている単語の場合と同様である。Suffixファイルは例えば次の書式になる。[10] Jは形容詞、Nは名詞、Vは動詞、Rは副詞を意味する。

---

10 Wilson, p. 159.

また理論的には可能な品詞候補であっても、実際にその品詞として利用される機会の極端に少ない候補には、略語の直後に@（これは÷10を意味する）、%（こちらは÷100を意味）の記号が振られている。

```
able   J
al   J N%
ance   N
ant   J N
ed   V J@
er   N V J
ers   N V
est   J
ful   J N%
ing   N V J@
ings   N
ity   N
ive   null
less   J
ly   R J@
ment   N
ness   N
th   N M
tion   N
n   J N
y   J N
```

例えば最後の例では、yで終わる単語は形容詞か名詞であることが示されてる。

```
void CLexicon : : setSuffixMap (CString s_filename)
{

// Suffix 情報ファイルから読み取る。
CStdioFile fin (s_filename, CFile : : modeRead) ;
CString line ;
while (fin. ReadString (line) != NULL){
```

```
//  一行ずつ読み取り、タブを境に前後の文字列に分ける。
    int place = line. Find (m_tab) ;
    CString key, val ;
    key = line. Left (place) ;
    val = line. Mid (place + m_tab.GetLength()) ;

//  読み取った語形を、key と value でマッピングする。
    if (key != "")
    m_mapSuff.SetAt (key, val) ;
    }
    }
```

3 CSuffix クラスの実装

このクラスの目的は、生テキストに現れる単語のうち、別に用意された辞書ファイルに登録された語形が見つかった場合、その語尾の形によって品詞を判定する手段を提供することにある。CSuffix の match 関数では、送られてきた単語を語頭から一文字削除しては、語尾対応表に一致する語尾が無いかを検索し、一致する語形があった場合、その語形と対になった品詞候補を取得する。

```
//  コンストラクタ、デコンストラクタともに特別な処理は行わない。
    CSuffix : : CSuffix() {}
    CSuffix : : ~CSuffix() {}
    CSuffix : : match (CString word, CLexicon *lex)
    {

//  保存用の文字列変数を用意する。
    CString retStr = "" ;
    CString entry = "" ;
    CString w_copy = word ;

//  引数を保存し、以下で頭から一文字ずつ削っては、語尾ファイルを検索する。
    BOOL find = false ;
```

```
    while (!find){
    if (lex->m_mapSuff. Lookup (w_copy, entry)){

//  もしもすぐに見つかれば、その品詞情報を返して、ループを終了する。ちなみにentryには品詞情報、例えば“J”（形容詞）などが代入される。
    retStr = entry ;

//  見つかった場合、フラグを立てる。
    find = true ;
    }
    else
    {

//  見つからなかった場合、最長一致法に基づき、最初の一文字から削っていき、語尾データに一致する語形が現れるまでループを繰り返す。ただし文字列が残っている場合に限り実行する。
    if(w_copy. GetLength ()>0)
    {
    if (w_copy. GetLength ()==1)
    {

//  見つからず、最後に残った一文字がsの場合は
    if (w_copy.GetAt (0)=='s')
    {

//  sのみを削除した語形で、辞書ファイルを検査する。
    retStr = lex->lookUp (word. Left (word. GetLength ()-1)) ;
    if (retStr == "")
    retStr = "Error" ;
    }
    find = true ;

//  ここでループを抜ける。
    }
```

```
        else
        {

//  頭から一文字ずつ削っていった検索対象語に、まだ文字が残っているかを
    調べる。
        w_copy = w_copy. Mid (1) ;
        }
        }
        else
        find = true ;
        }
        } ,
        return (retStr) ;
        }
```

4 CMatrix クラスの実装

このクラスの目的は、単語に複数の品詞候補があった場合、その前後に表れる単語の品詞情報から、当該単語の品詞を確率的に求めようとするものである。ここでは、McEnery&Wilson の確率表 (p. 162) をそのままハードコーディングしている。[11]

```
CMatrix : : CMatrix () {}
CMatrix : : ~CMatrix () {}

//  まず異なる（あるいは同一の）品詞同士が連続して使われる統計的確率を
    ハードコーティングしてしまう。マトリックス表を用いる。
    int CMatrix : : matrix [][18]={
      /*A*/{0,0,1,0,0,0,0,29,4,64,0,2,0,0,0,0,0,1},
      /*C*/{14,2,5,2,0,0,9,7,1,19,14,9,1,0,13,1,0,3},
      /*D*/{5,1,3,0,0,0,8,6,2,42,6,3,0,0,16,0,0,7},
      /*E*/{2,0,1,0,0,0,0,0,0,0,0,2,0,0,93,0,0,0},
```

11 Wilson, p. 162.

```
    /*F*/{0,2,0,0,49,0,2,0,1,7,0,1,0,0,6,0,0,30},
    /*G*/{1,1,1,0,1,0,0,7,1,84,0,1,0,0,0,0,0,2},
    /*I*/{46,0,10,0,0,0,0,0,6,4,22,4,1,0,0,4,0,0,1},
    /*J*/{0,5,0,0,0,0,0,5,3,0,72,0,1,4,0,0,0,0,9},
    /*M*/{0,2,1,0,1,0,4,3,2,57,1,2,0,0,2,0,0,26},
    /*N*/{1,6,1,0,0,1,27,0,1,8,1,3,1,0,17,0,0,31},
    /*P*/{1,2,1,0,0,0,4,1,0,1,1,7,2,0,71,0,0,9},
    /*R*/{9,4,3,0,0,0,14,11,1,3,6,11,1,0,21,0,0,16},
    /*T*/{0,0,0,0,0,0,0,0,0,0,0,0,0,99,0,0,0,0},
    /*U*/{1,1,1,0,0,0,1,0,0,1,1,2,0,3,0,0,0,89},
    /*V*/{17,2,3,0,0,0,11,5,1,4,5,16,5,0,18,3,0,10},
    /*X*/{7,1,2,0,0,0,5,4,0,1,6,8,4,0,56,0,0,6},
    /*Z*/{0,4,1,0,2,0,19,2,0,4,0,1,0,0,10,0,0,55},
    /*O*/{9,10,4,0,1,0,5,4,1,10,13,6,0,1,7,0,0,29}};
  CString CMatrix::labels = "ACDEFGIJMNPRTUVXZO";

//  これらの記号の意味ついて簡単に説明する。
```

'A' : Article	冠詞類	'C' : Conjunction	接続詞
'D' : Determiner	限定子	'E' : Existential	存在の there
'F' : Formulae	外来語や数式	'G' : Genitive	所有形
'I' : Preposition	前置詞	'J' : Adjective	形容詞
'M' : Number	数詞	'N' : Noun	名詞
'O' : Other	句読点	'P' : Pronoun	代名詞
'R' : Adverb	副詞	'T' : Infinitive maker	to(不定詞)
'U' : Interjection	感嘆詞	'V' : Verb	動詞
'X' : Negator	否定辞	'Z' : Letter (s)	アルファベットとしての文字(列)

このマトリックス表の利用方法は次の通り。

まず名詞であることが判明している語彙の後に名詞(N)か動詞(V)の二つの可能性のある語彙が来ているとする。さらにその不確定な語彙の後に、同じく名詞か動詞か不明な語彙が続いているとする。そのとき、それぞれの場合について表の数値を合計すると、次の合計が得られる。

N－N－V＝8＋17＝25

N－N－N＝ 8 ＋ 8 ＝16
N－V－V＝17＋18＝35
N－V－N＝17＋ 4 ＝21

このときもっとも数値の高い文字の並び、すなわち N－V－V という品詞の組み合わせが確率的にもっとも確からしいとして、品詞判定の根拠とするのである（この N－V－V の組み合わせの場合、名詞か動詞（助動詞）という二つの可能性があるから、35÷2で、17.5%の確率と計算される)。

```
int CMatrix : : lookup(char tag 1, char tag 2)
{

// Matrix 表から頻度値を基に可能性の最も高い品詞を決定する。
int index 1=labels.Find (tag 1) ;
int index 2=labels.Find (tag 2) ;
return (matrix[index 1][index 2]) ;
}
```

5 CTagger クラスの実装

このクラスの目的は、加工対象となる生テキストを読み込み、単語（トークン）に切り分けてから品詞判定し、品詞情報をその単語にタグ形式で付与することにある。またタグを付与した後、その結果をユーザーが指定するファイルに書き込む(GUI プログラムの場合、画面に表示も行う)。品詞判定の方法は、あらかじめ用意された辞書ファイルと照合を行う。複数の品詞候補がある場合は、別に用意された品詞確率表（マトリックス）から、共起可能性の数値が最も高い候補を採用する。

当該単語が辞書ファイルにない場合は、語尾から品詞を判定する。語尾との照合は最長一致法で行う。すなわち、検定対象となる語の最初から一文字ずつ削っていき、残された文字列が、別に用意された語尾対応表に記載された語形と一致したところで、対応表から候補となる品詞情報を取得する。複数の候補がある場合の処理は、マトリックス表から、もっとも確からしい候補を採用する。

```
CTagger : : CTagger()
{
```

```
// コンストラクタで、品詞判定結果を代入する変数を初期化しておく。ここ
   では仮に初期値にO(other　句読点)とプログラム作者名を代入しておく。
  prevTag = currTag = 'O' ;
  prevWord = "AllRight ISHIDA" ;
  }
  CTagger::~CTagger() {}
  void CTagger::makeLexicon (CString fname)
  {

// まずCLexiconクラスのオブジェクトを通して辞書ファイルをロードする。
  m_lex. setLexicon (fname) ;
  }
  void CTagger::readText (CString m_text)
  {

// 加工対象である生テキスト本体を読み込む。
  CStdioFile fin (m_text, CFile::modeRead) ;
  CString line ;

// トークンに分けるための変数の準備をする。生テキストの単語は、前後に
   スペースがあり、また文末などではピリオドやコンマが付加されているの
   で、それらは除去する。
  char seps[] = " .," ;
  char *token ;
  CString m_buff ;
  //char buff[1001] ;
  while (fin. ReadString (m_buff) != NULL){

// 一行ずつファイルの末尾まで読み取り、単語（トークン）に分ける。
  CString m_str ;
  CString m_string [50] ;
  int count=0 ;
  m_str = m_buff ;
```

```
// ここで文字列をトークンで区切る。strtok 関数は第一引数の文字列を、第
   二引数で指定した記号で切り分ける。ただしここで用いている m_str 変
   数は CString クラスのオブジェクトなので、キャストが必要となる。また
   strtok 関数の内部で、さらに strtok 関数を呼び出すことはできないので
   注意が必要である。
   token = strtok ((char *) (const char *)m_str, seps) ;
   while( token != NULL)
   {

// トークンがなくなるまで繰り返す。返された値は char 型なので、これを
   CString に変えてやる。
   m_string [count++].Format ("%s",token) ;

// 次のトークンを取得する処理を繰り返す。
   token = strtokw (NULL, seps) ;
   }
   for (int cnt=0 ; cnt <= count ; cnt++){

// ここでタグ付与を行う関数 Tag に単語（トークン）を引数にして送る。
   Tag (m_string[cnt]) ;
   }
   }
   }
   void CTagger : : Tag (CString m_word)
   {

// 引数として渡されたトークンを、別に読み込んだ辞書ファイルと照合し、
   品詞を判定して、その情報をタグ形式で書き込む。まず CLexikon クラス
   の lookup 関数に単語を送り、品詞情報を取得する。品詞候補が複数あれ
   ば、それぞれの候補語と前後の品詞との共起度を、確率表から調べ、もっ
   とも数値の高い品詞を採用する。品詞情報が確定した単語は、タグ付けを
   行う関数へ渡す。
   if (prevWord != "")
   printLine (prevWord, prevTag) ;
```

```
//  単語をひとつ進める。
    prevWord = currWord ;
    prevTag = currTag ;
    currWord = m_word ;
    currTag = 'O' ;
    double score=0.0 ;

//  現在の単語を辞書ファイルと照合し、品詞情報を取得する。
    CString possTags = m_lex.lookUpw (m_word) ;

//  仮に品詞情報が空のことがあれば、デフォルトの品詞情報で代用する。
    if (possTags == "")
    possTags = "N J V" ;
    char seps[] = "" ;
    char *token ;
    CString tempTag ;

//  品詞情報には複数の候補がある場合もあるので、統計的にもっとも確からしい候補を選ぶ処理を行う。その準備として、スペースで区切られた品詞情報の一つ一つを取得する。例えばＮＪＶであれば、Ｎ（名詞）、Ｊ（形容詞）、Ｖ（動詞）の三つの可能性があるわけなので、それぞれについて出現する確率を取得する。
    token = strtok ((char *)(const char *) possTags, seps) ;
    while(token != NULL)
    {

//  なお Wilson の共起度表では、頻度の少ない品詞には@や%の記号がつけられている。その場合は取得した数値をそれぞれ10ないし100で割る。
    tempTag. Format ("%s", token) ;

//  確率表と照合する。これを全部の品詞候補について繰り返す。
    double tmpScore =
    CMatrix : : lookup (prevTag, tempTag. GetAt (0)) ;
    if (tempTag. GetAt (tempTag.GetLength()－1) =='@')
```

```
        tmpScore/=10 ;
        if (tempTag. GetAt (tempTag.GetLength()－1) ==‘%’)
        tmpScore/=100 ;

//  もっとも数値の高い候補を求める。
        if (tmpScore > score){
        score = tmpScore ;
        currTag = tempTag.GetAt(0) ;
        }
        token = strtok (NULL, seps ) ;
        }
        }
        void CTagger : : makeSuffix (CString sufname)
        {

//  Lexicon クラスのオブジェクトを通して語尾対応表をロードする。
        m_lex. setSuffixMap (sufname) ;
        }
        CString CTagger : : map (char tag)
        {

//  語尾対応表の略語を、分かりやすい表記に変える
        switch(tag){
        case ‘A’ : return (“ART”) ;
        case ‘C’ : return (“CONJ”) ;
        case ‘D’ : return (“DET”) ;
        case ‘E’ : return (“EXTH”) ;
        case ‘F’ : return (“FORM”) ;
        case ‘G’ : return (“GENS”) ;
        case ‘I’ : return (“PREP”) ;
        case ‘J’ : return (“ADJ”) ;
        case ‘M’ : return (“NUM”) ;
        case ‘N’ : return (“NOUN”) ;
        case ‘O’ : return (“OTHER”) ;
```

```
case 'P' : return ("PRON") ;
case 'R' : return ("ADV") ;
case 'T' : return ("INFTO") ;
case 'U' : return ("INTERJ") ;
                case 'V' : return ("VERB") ;
                case 'X' : return ("NEG") ;
                case 'Z' : return ("LETTER") ;
default :
return ("UNDEF") ;
}
return ("Error") ;
}
void CTagger : : printLine (CString word, char tag)
{
```

品詞判定された語については、その情報をタグとして語に付記する。例えば linguistics という単語であれば

```
<w pos=NOUN> word="linguistics"</w>
```

とタグを付与する。pos は part of speech 品詞の略語、NOUN は名詞である。w は単語情報のタグの開始を意味し、/w はそれを閉じる。また GUI プログラムであれば、画面のテキスト欄にタグ付けされた結果を表示する。

```
CString line ;
CString trans = map (tag) ;
line. Format ("<w pos=%s> word=¥"%s¥"</w>",trans,word) ;
if (line. GetLength ()<55)
fout.WriteString (line) ;
}
void CTagger : : setOutputFile (CString outfile)
{

//  結果を書き込む出力先ファイルを準備する。
if (!fout. Open (outfile, CFile : : modeCreate | CFile : : modeWrite))
```

```
    {

//  万が一、出力ファイルを作成できなければ、ここを抜ける。
    exit (1) ;
    }
    }
    void CTagger : : closeProg()
    {

//  前後の共起関係を確率的に求めて品詞判定しているため、テキストの最後
    に現れる単語のタグ付けは終わってない。プログラム終了時に残っている
    単語とタグを書き込む。
    printLine (prevWord, prevTag) ;
    }
    void CTagger : : closeOutfile()
    {

//  ファイルを閉じる
    fout.Close() ;
    }
```

以上のコーディングを基に、ユーザーインターファイスを作成する。

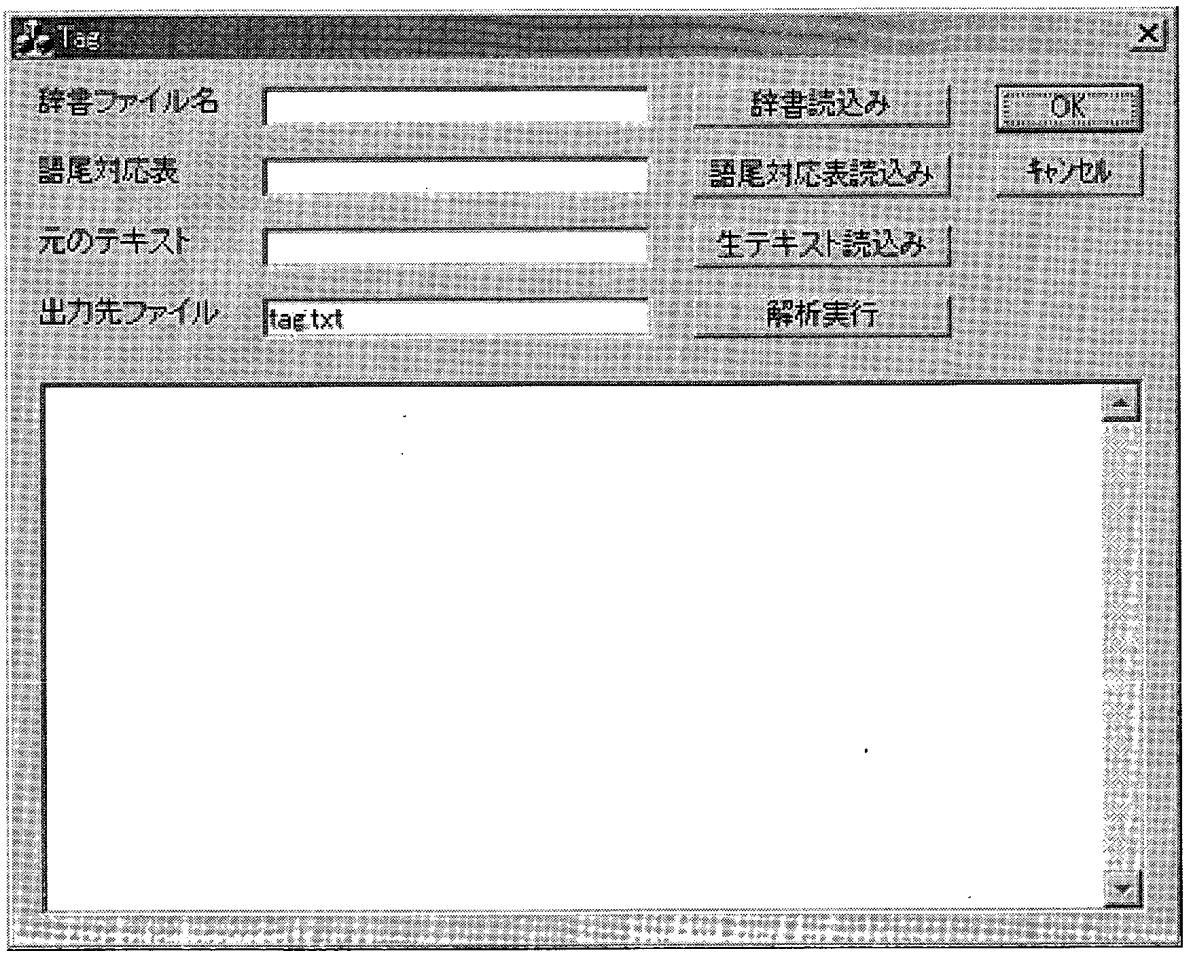
6 GUI インターフェイス

以上の章に取り上げたコーディングを基に、例えば以下のようなユーザーインターファイスを作成する。インターフェイスについては、デザインはもちろん、ユーザの誤操作を回避するための諸機能（あるいはヘルプ機能）などを備えるべきであるが、ここではごく単純なモデルを示すにとどめる。ユーザーは、四つの操作ボタンを利用して、必要となる四つのファイルを選択する。GUI化されたプログラム本体、また動作確認のためのサンプルファイルをホームページ上に公開しておく。[12]

---

12 http : //www.ias.tokushima-u.ac.jp/linguistik/program/index.html

ここで辞書ファイルとは、“ball　N”のように、単語原型と品詞（ここでは名詞N）を対にした辞書を意味する。辞書ファイルの選択は、まずボタンを押してファイル選択画面を出し、ファイルをダブルクリックしてロードする（サンプルとして、辞書ファイルにLEX. TXT、語尾対応表としてSUFF. TXT、生テキストとしてRAW. TXTがあるので実験してみて欲しい）。



Taggerプログラムの GUI例

語尾対応表は、例えば英語でilyで終わる単語は副詞と予測されるが、こうした語尾と品詞の組み合わせ可能性について記したファイルである。これもボタンを押し、用意したファイル名を選択してロードする（サンプルとしてSUFF. TXT）。

あとはタグを付与する対象となる生テキストファイル名（サンプルとしてRAW. TXT）と、タグを付与した結果を記録するファイル名を指定してやる。GUIの下半分を占めるテキスト欄には、処理結果（すなわちタグが付与されたテキスト）が表示される。