Division of text into sentences

Stanisław Galus

2 August 2005

1 Introduction

One often needs to divide text written in Polish into sentences. Let us make it our aim to write a simple algorithm performing this task. The simplicity of the algorithm shall consist in using only information contained in the dividing text and not using external lexical data.

The major part of definitions characterizes a sentence as "a group of words related on the ground of grammatical dependence, containing predicate, most often as a personal form of a verb".[1] With a view to simplicity it will be better to be based on the orthographical criterion [10, p. 39], which allows to isolate sentences on the base of capital and small letters and punctuation marks. For example, Grochowski [4, p. 218] defines a sentence as "a segment of text, from both sides delimited by a graphical sign of a dot, including a personal form of a verb". Since the algorithm will not be recognizing verbs, segments isolated by it will only potentially be sentences. The decision about recognizing a segment to be a sentence will have to be made outside the algorithm, depending on whether or not the segment contains a desired form of a verb.

The rules of Polish orthography concerning the discrimination of sentences are given in the dictionary [8]. A sentence may be terminated with a full stop, a question mark, an exclamation mark or dots (§§85, 86). If a sentence is terminated with a question mark, exclamation mark or dots, the border between it and the next sentence is signaled by a capital letter beginning that next sentence (§85). The other functions of the full stop are ending abbreviations, initials and ordinal numbers written with digits (§87). A sentence may be ended with substantially any sequence of question marks, exclamation marks and dots (§§94.5, 94.6, 95.7). Spelling rules concerning the meeting point of the other punctuation marks with the full stop, question mark, exclamation mark and dots (§§97.7, 98.D) are often used inconsistently and we will pay no attention to them.

In recent language engineering literature, a brief description of an algorithm dividing text into sentences can be found in the work of Przepiórkowski [9, p. 13].

[1]See [13, vol. III, p. 926]. For similar definitions, also [6, p. 1775], as well as [7, p. 240], [5, p. 13], [10, p. 43], [12, p. 301].

2 End of sentence

We assume that text is composed of words, punctuation marks, hyphens, numbers written with digits and graphical signs such as =, +, %, §. We denote:

U	–	the set of words beginning with capital letters,
L	–	the set of words beginning with small letters,
W	=	$U \cup L$ – the set of words,
P	=	{., ;, ,, :, –, . . . , ?, !, (,), [,], /, {, }, <, >, „, ”, ≪, ≫, ‘, ’, "} – the set of punctuation marks,
-	–	the hyphen,
N	–	the set of numbers expressed with digits,
G	–	the set of graphical signs,
Z	=	{., ;, ,, :, . . . , ?, !,),], }, >, ”, ≫, ’, -},
A	=	{–, (, [, /, {, <, „, ≪, ‘, "} $\cup G$.

The set Z consists of symbols which – as it seems – may not appear at the beginning of a sentence. The set A consists of all characters which do not belong to any of the sets W, N and Z. A and Z are disjoint and $A \cup Z = P \cup \{\text{-}\} \cup G$. The universe of lexical symbols is $W \cup P \cup \{\text{-}\} \cup N \cup G = W \cup N \cup A \cup Z$.

A text may be defined as a sequence of potential sentences and runs of rubbish with no adjoining runs of rubbish. Denoting the set of potential sentences and the set of runs of rubbish by S and R, repectively, we may define the set of texts by[2]

$$R?(S^+R)^*S^*$$

The set of potential sentences may be defined as

$$(A^* \cup A^+Z(A \cup Z)^*)(U \cup N)(W \cup N \cup A \cup Z)^*\{., ?, !, \ldots\}$$

The problem of detecting the end of sentence is equivalent to finding a proper number of full stops, question marks, exclamation marks and dots between the opening capitalized word or number and the end of sentence.

A proposed algorithm for detecting the end of sentence can be informally formulated as follows.

1) Read the text from the current lexeme until a lexical symbol $y \in \{., ?, !, \ldots\}$ is detected.

2) If y is the last lexical symbol in the text, it closes a potential sentence.

3) If y is immediately followed by a lexeme from Z, y does not close a sentence. Continue from the lexeme immediately following y.

4) If neither a word nor a number occurs between y and the end of text, y closes a potential sentence (on the strength of step 3), y is not immediately followed by a lexeme from Z). All lexemes following y will form a run of rubbish.

[2]For regular expressions, [2, ch. 3].

5) Let us denote by z the first word or number following y. At this point we assert $y(A^* \cup A^+Z(A \cup Z)^*)z$, $z \in U \cup L \cup N$. If z is a word beginning with a small letter, y does not close a sentence. Continue from the lexeme immediately following z.

6) We assert $y(A^* \cup A^+Z(A \cup Z)^*)z$, $z \in U \cup N$. If y is not a full stop, y closes a potential sentence.

7) Let x be a lexeme preceding y. We assert $xy(A^* \cup A^+Z(A \cup Z)^*)z$, $y \in \{.\}$, $z \in U \cup N$. If x is neither a word nor a number, y closes a potential sentence. This mainly concerns full stops immediately following the closing brackets and the closing question marks.

8) We assert $xy(A^* \cup A^+Z(A \cup Z)^*)z$, $x \in U \cup L \cup N$, $y \in \{.\}$, $z \in U \cup N$. The decision should be made as follows.

a) If x is a number expressed with digits and z is an capitalized word, the full stop closes a potential sentence. We prefer correct splitting sentences "Urodził się w roku 1974. Było to w lecie" than incorrect splitting the sentence "Było to tuż po 6. Konferencji".

b) If both x and z are numbers expressed with digits, the full stop does not close a sentence. The motivation behind this is the correct treatment of dates expressed with digits and dots.

c) If a known abbreviation is a suffix of x followed by a full stop which should not close a sentence, the full stop does not close a sentence.

d) If x is an initial and z is a capitalized word, the full stop does not close a sentence as we intend not to split the sentence "A. Kowalski urodził się w lipcu".

e) If x is an initial and z is a number expressed with digits, the full stop closes a potential sentence, as we would like to split the sentences "W lipcu urodził się K. 12 sierpnia miał już miesiąc".

f) In the remaining cases, the full stop closes a potential sentence.

The list of abbreviations mentioned in step 8c must contain abbreviations which end with a full stop, can hardly close a sentence and are not unabbreviated words. To prepare the list we used the lists from [11] and [6]. The following abbreviations have been removed because they often close sentences:

A. M. D. G., D. O. M., I. H. S., I. N. C., I. N. D., I. N. R. I., N. N., N. T., P. S., R. I. P., R. S. V. P., S. P. Q. R., S. T. T. L., S. V. P., V. S. O. P., b. a., b. d., b. m., b. m. r., b. m. r. w., b. m. w., b. opr., b. r., b. r. w., b. w., bdb., c. b. d. o., c. d., c. d. n., cd., cdn., cel., d. c. n., d. n., db., dcn., dok. nast., dost., dst., etc., itd., itp., j. n., jw., mrn., nag., ndp., ndst., op. cit., p. niż., p. wyż., r. s. v. p., s. v. p., v. v.,

or they are unabbreviated words:

asyst., aut., bar., bat., cal., cyt., dok., dom., druk., form., fort., gal., gen., im., kadm., kand., kap., kart., kat., kier., kol., kop., krypt., kur., kwart., lab., lek., mar., marsz., miejsc., min., nap., os., par., pil., por., prac., proc., prom., pseud., red., rep., rys., sap., słuch., tłum., ucz., ul., ust., wyż., zgrom.

3 Implementation

An implementation in the C programming language consists of a header file `dts.h` and a source file `dts.c`. The header file exports two data types and one function:

⟨dts.h⟩≡

```
#ifndef DTS_H
#define DTS_H

struct symbol                    /* lexical symbol */
{
  int type;                      /* type as in getsym.h */
  char val[32];                  /* literal value */
};

struct parse                     /* parsing context */
{
  FILE *inputf;                  /* input file */
  int line;                      /* current line */
  int status;                    /* internal status for getsym */
  struct symbol que[256];        /* input queue */
  int first, last;               /* first and last elements in que */
  int type;                      /* 0 = EOF, 1 = sentence, 2 = rubbish */
  struct symbol run[1024];       /* sentence or rubbish */
  int n;                         /* length of run */
};

int dts(struct parse *p);

#endif /* DTS_H */
```

The `struct symbol` holds a lexical symbol. The field `type` holds type of the symbol as defined in the header file `getsym.h` and the field `val` holds literal value of the symbol. The `struct parse` is used to pass parameters to the function `dts`. The field `inputf` is a pointer to an open input file, `line` is a number of the current line of that file and `status` is a variable that holds the internal status of lexical analyser. The table `que` is an input queue and the variables `first` and `last` are indexes of the first and the last elements of the queue. The table `run` is used to return the next run of lexical symbols from `inputf` from the function `dts`. After a call to this function, the field `type` contains 0 if the end of file has been encountered, 1 if `run` contains a potential sentence and 2 if it contains a run of rubbish. In the last two cases, the field `n` returns length of the run. One call to the function `dts` gets one run of lexical symbols. The value returned by the function is the same as that in the field `type` of its argument. If the returned value is 0, the status of `inputf` should be examined. A filter calling the function `dts` to extract all potential sentences from standard input could be like this:

⟨*exemplary program*⟩≡
```
int main(void)
{
  struct parse p;
  int i;

  ⟨initialize p⟩
  while (dts(&p) != 0)
    if (p.type == 1)
    {
      for (i = 0; i < p.n; i++)
        printf("%s\n", p.run[i].val);
      printf("\n");
    }
  assert(feof(p.inputf) && !ferror(p.inputf));
  return 0;
}
```

⟨*initialize* p⟩≡
```
p.line = p.status = 0;
p.first = p.last = -1;
p.inputf = stdin;
```

The sizes of arrays `que` and `run` have been determined as twice as much as the minimum powers of two greater than sizes enough to process some novels from [1]. The longest run accepted as a sentence, having 421 symbols, comes from „Kariera Nikodema Dyzmy".

The program file consists of the following sections:

⟨*dts.c*⟩≡
```
  #include <assert.h>
  #include <stdio.h>
  #include <string.h>
  ⟨includes⟩
  ⟨macros⟩
  ⟨data⟩
  ⟨static functions⟩
  ⟨functions⟩

  #ifdef TEST_DTS

  ⟨exemplary program⟩

  #endif /* TEST_DTS */
```
There are four non-system include files:

⟨*includes*⟩≡
```
  #include "l2.h"
  #include "getsym.h"
  #include "dts.h"
```

The files `l2.h` and `getsym.h` declare external `ctype`-like functions for the ISO 8859-2 character encoding as well as the lexical analyser and constants for lexical symbols. These files as well as the program files `l2.c` and `getsym.c` are not explained here.

The very well known macro `NELEMS` gives the number of elements in an array:

⟨*macros*⟩≡
```
  #define NELEMS(a) (sizeof(a) / sizeof(a[0]))
```

The function `isabb` determines whether the word before the full stop is a suffix of an abbreviation. If it is, true is returned, otherwise false. The function is used in step 8c of the algorithm.

⟨*static functions*⟩≡
```
  static int isabb(struct parse *p)
  {
    char s[MAXABBLEN + 1];
    int i = p->n, n1 = i - 1, h;
    int avail = MAXABBLEN, request;
    size_t ls;

    ⟨prepare the string for searching⟩
    ⟨search in the table of abbreviations⟩
  }
```

The constant

⟨*data*⟩≡
```
  #define MAXABBLEN 11            /* the maximum length in abb */
```

is the maximum length of an abbreviation kept in the table `abb`. Since the table is defined as

```
static const struct {const char *str; size_t len;} abb[],
```

`MAXABBLEN` must not be less than maximum of `abb[i].len`. Because the run of lexical symbols being checked is at first converted to the variable `s`, it is no use to declare it with a length greater than `MAXABBLEN`. In the first step of the conversion the right-hand side part of the run of lexical symbols is determined which will fit into `s`. One space is added between two consecutive words and two consecutive numbers. In the second step, the string representations of the symbols are copied to `s`.

⟨*prepare the string for searching*⟩≡

```
while (i- > 0)
{
  request = strlen(p->run[i].val);
  if ((p->run[i].type == WORD || p->run[i].type == NUMBER)
      && i < n1 && p->run[i].type == p->run[i + 1].type)
    request++;
  if (request > avail)
  {
    i++;
    break;
  }
  avail -= request;
}
*s = '\0';
while (i < p->n)
{
  strcat(s, p->run[i].val);
  if ((p->run[i].type == WORD || p->run[i].type == NUMBER)
      && i < n1 && p->run[i].type == p->run[i + 1].type)
    strcat(s, " ");
  i++;
}
```

When `s` is formed, its length is determined and the abbreviations not longer than `s` are examined. If a suffix in `s` exists which matches an abbreviation, the run of symbols ends with it.

⟨*search in the table of abbreviations*⟩≡

```
ls = strlen(s);
for (i = 0; abb[i].str != NULL; i++)
  if (ls >= abb[i].len)
  {
    h = ls - abb[i].len;
    if (strcmp(s + h, abb[i].str) == 0)
      if (h == 0 || !l2_isalpha(*(s + h - 1)))
        return 1;
  }
return 0;
```

The function `getlex` is used to get the next lexical symbol from the file `p->inputf` and put it in `p->que`. After an assertion is made about free place in the queue,[3] the new index for the last element in the queue is determined and a call to lexical analyser is made.

⟨*static functions*⟩+≡
```
static void getlex(struct parse *p)
{
  struct symbol *s;

  assert(!((p->first == 0 && p->last == NELEMS(p->que) - 1) ||
           p->first == p->last + 1));
  if (p->last == NELEMS(p->que) - 1 || p->last == -1)
  {
    p->last = 0;
    if (p->first == -1)
      p->first = 0;
  }
  else
    p->last++;
  s = &p->que[p->last];
  s->type = getsym(p->inputf, s->val, NELEMS(s->val), &p->line,
                   &p->status);
}
```

The function `move` moves the first element from the queue to the last position in the run. First, it is asserted that the queue is not empty and that the run has a free place. Then the lexical symbol is moved and indexes are updated.

⟨*static functions*⟩+≡
```
static void move(struct parse *p)
{
  struct symbol *s = &p->que[p->first], *t = &p->run[p->n];

  assert(p->first != -1);
  assert(p->n < NELEMS(p->run));
  t->type = s->type;
  strcpy(t->val, s->val);
  p->n++;
  if (p->first == p->last)
    p->first = p->last = -1;
  else if (p->first == NELEMS(p->que) - 1)
    p->first = 0;
  else
    p->first++;
}
```

[3]The implementation of the queue in `struct parse` follows [3, p. 113, fig. 6.10].

The function `isZ` returns non-zero result if and only if the lexical symbol `t` belongs to the set Z defined at the beginning of section 2.

⟨*static functions*⟩+≡
```
static int isZ(int t)
{
  switch (t)
  {
    case FULL_STOP: case SEMICOLON: case COMMA: case COLON: case DOTS:
    case QUESTION_MARK: case EXCLAMATION_MARK: case RIGHT_ROUND_BRACKET:
    case RIGHT_SQUARE_BRACKET: case RIGHT_CURLY_BRACKET:
    case RIGHT_ANGLE_BRACKET: case RIGHT_QUOTATION_MARK:
    case RIGHT_ANGLE_QUOTATION_MARK:
    case RIGHT_DEFINITION_QUOTATION_MARK: case HYPHEN:
      return 1;
  }
  return 0;
}
```

The function `get_next_run` does the whole work.

⟨*static functions*⟩+≡
```
static void get_next_run(struct parse *p)
{
  int i, x, y, z;

  p->type = 0;
  ⟨check for the end of file⟩
  p->n = 0;
  p->type = 2;
  ⟨check for a run of rubbish⟩
  p->type = 1;
  ⟨check for a candidate for potential sentence⟩
}
```

⟨*check for the end of file*⟩≡
```
if (p->first == -1)
  getlex(p);
if (p->que[p->first].type == 0)
  return;
```

If the first lexical symbol belongs to Z, read all symbols until a symbol not belonging to Z or the end of file:

⟨*check for a run of rubbish*⟩≡
```
if (isZ(p->que[p->first].type))
  for (;;)
  {
    move(p);
    if (p->first == -1)
      getlex(p);
    if (!isZ(p->que[p->first].type))
      return;
  }
```

The most difficult part of the function exactly follows the algorithm:

⟨check for a candidate for potential sentence⟩≡

```
for (;;)
{
  ⟨collect lexical symbols until y is a ., ?, !, ...⟩
  if (p->que[p->first].type == 0)
    return;                     /* step 2 */
  if (isZ(p->que[p->first].type))
    continue;                   /* step 3 */
  ⟨read text until a word or a number⟩
  if (l2_islower(p->que[i].val[0]))
  {
    while (p->first != i)
      move(p);
    continue;                   /* step 5 */
  }
  if (y != FULL_STOP)
    return;                     /* step 6 */
  assert(p->n > 1);
  x = p->run[p->n - 2].type;
  if (x != WORD && x != NUMBER)
    return;                     /* step 7 */
  if (x == NUMBER)
  {
    if (z == WORD)
      return;                   /* step 8a */
    if (z == NUMBER)
      continue;                 /* step 8b */
  }
  if (isabb(p))
    continue;                   /* step 8c */
  if (⟨ends with an initial⟩)
  {
    if (z == WORD)
      continue;                 /* step 8d */
    if (z == NUMBER)
      return;                   /* step 8e */
  }
  return;                       /* step 8f */
}
```

When collecting lexical symbols in `p->run`, if the end of file is reached before a punctuation mark from $\{., ?, !, \ldots\}$ is detected, the type of the run is reset to 2 and the function returns:

⟨*collect lexical symbols until y is a ., ?, !, . . .*⟩≡
```
  move(p);                    /* step 1 */
  if (p->first == -1)
    getlex(p);
  y = p->run[p->n - 1].type;
  if (y != FULL_STOP && y != QUESTION_MARK && y != EXCLAMATION_MARK &&
      y != DOTS)
  {
    if (p->que[p->first].type == 0)
    {
      p->type = 2;
      return;
    }
    else
      continue;
  }
```
When searching for a next word or a number, `p->queue` is searched first. If nothing is found, subsequent symbols are read from the input file:

⟨*read text until a word or a number*⟩≡
```
  if ((i = p->first) != -1)
    for (;;)
    {
      z = p->que[i].type;
      if (z == WORD || z == NUMBER)
        break;
      else
      {
        if (i == p->last)
        {
          i = -1;
          break;
        }
        if (i == NELEMS(p->que) - 1)
          i = 0;
        else
          i++;
      }
    }
  if (i == -1)
    for (;;)
    {
      getlex(p);
      i = p->last;
      z = p->que[i].type;
      if (z == 0)
        return;               /* step 4 */
      if (z == WORD || z == NUMBER)
        break;
    }
```

The run `p->run` ends with an initial if it ends with a capital word whose length is 1:

⟨*ends with an initial*⟩≡
```
strlen(p->run[p->n - 2].val) == 1 &&
l2_isupper(p->run[p->n - 2].val[0])
```

The exported function `dts` calls `get_next_run` and if a candidate for potential sentence is returned, it is tested to contain a capitalized word or a number expressed with digits. If the test fails, the type of run is changed to rubbish.

⟨*functions*⟩≡
```
int dts(struct parse *p)
{
  get_next_run(p);
  if (p->type == 1)
  {
    int i;
    for (i = 0; i < p->n; i++)
      if ((p->run[i].type == WORD && l2_isupper(p->run[i].val[0])) ||
          p->run[i].type == NUMBER)
        break;
    if (i >= p->n)
      p->type = 2;
  }
  return p->type;
}
```

And here is the table of abbreviations `abb` constructed in the way described in the latter part of section 2. Characters given by octal codes correspond to the ISO 8859-2 character set.

⟨*data*⟩+≡

```
static const struct {const char *str; size_t len;} abb[] =
{
  {"A.D.", 4}, {"A.M.", 4}, {"Adm.", 4}, {"Adr.tel.", 8}, {"Al.", 3},
  {"Am.Pd.", 6}, {"Am.Pn.", 6}, {"Amer.P\263d.", 9},
  {"Amer.P\263n.", 9}, {"Archip.", 7}, {"Az.", 3}, {"Bl.", 3},
  {"Bryg.", 5}, {"C.O.", 4}, {"Dw.", 3}, {"Dyr.", 4}, {"Dz.U.", 5},
  {"Fot.", 4}, {"Fr.", 3}, {"G.", 2}, {"I.I.", 4}, {"ILek.", 5},
  {"ILot.", 5}, {"IPGum.", 6}, {"Inst.", 5}, {"J.", 2}, {"J.M.", 4},
  {"J.O.", 4}, {"J.P.", 4}, {"J.PP.", 5}, {"Jez.", 4}, {"Kan.", 4},
  {"Kier.", 5}, {"M.", 2}, {"Min.", 4}, {"N.", 2}, {"NB.", 3},
  {"Niz.", 4}, {"Np.", 3}, {"Nr rej.", 7}, {"O.", 2}, {"Ob.", 3},
  {"Oc.", 3}, {"Os.", 3}, {"O\266r.", 4}, {"P.", 2}, {"P.M.", 4},
  {"P.T.", 4}, {"PP.", 3}, {"PTAr.", 5}, {"Pd.", 3}, {"Pl.", 3},
  {"Prez.", 5}, {"P\263w.", 4}, {"P\363\263.", 4}, {"Rp.", 3},
  {"Rz.P.", 5}, {"Rzpl.", 5}, {"S.A.", 4}, {"S.T.", 4}, {"St.", 3},
  {"St.Zjedn.", 9}, {"Stow.", 5}, {"Szt.G\263.", 7}, {"T.T.", 4},
  {"Tow.", 4}, {"W.", 2}, {"W.Bryt.", 7}, {"Woj.", 4}, {"Wy\277.", 4},
  {"Zarz.G\263.", 8}, {"Zat.", 4}, {"Zw.", 3}, {"Zw.Radz.", 8},
  {"a.C.", 4}, {"a.C.n.", 6}, {"a.Chr.", 6}, {"a.Chr.n.", 8},
  {"a.a.C.", 6}, {"a.c.", 4}, {"a.h.l.", 6}, {"a.m.", 4}, {"a.p.C.", 6},
  {"ab init.", 8}, {"abp.", 4}, {"abs.", 4}, {"ad lib.", 7},
  {"ad loc.", 7}, {"adapt.", 6}, {"adj.", 4}, {"adm.", 4},
  {"adr.telegr.", 11}, {"adw.", 4}, {"agd.", 4}, {"al.", 3}, {"am.", 3},
  {"amb.", 4}, {"amer.", 5}, {"ang.", 4}, {"arch.", 5}, {"archip.", 7},
  {"arcybp.", 7}, {"arcyks.", 7}, {"art.", 4}, {"art.mal.", 8},
  {"art.rze\274.", 9}, {"aud.", 4}, {"b.", 2}, {"b.z.", 4},
  {"ba\263k.", 5}, {"ba\263t.", 5}, {"biul.", 5}, {"bl.", 3},
  {"bm.", 3}, {"bosm.", 5}, {"bp.", 3}, {"br.", 3}, {"bryg.", 5},
  {"bryt.", 5}, {"b\263.", 3}, {"c.", 2}, {"c.at.", 5}, {"c.cz.", 5},
  {"c.o.", 4}, {"c.par.", 6}, {"c.w\263.", 5}, {"cf.", 3}, {"chor.", 5},
  {"ci\352\277.", 5}, {"cz.", 3}, {"czyt.", 5}, {"cz\263.", 4},
  {"del.", 4}, {"dew.", 4}, {"dn.", 3}, {"doc.", 4}, {"dod.", 4},
  {"dol.", 4}, {"dop\263.", 5}, {"dos\263.", 5}, {"dot.", 4},
  {"dr h.c.", 7}, {"dr hab.", 7}, {"dr med.", 7}, {"dr.", 3},
  {"ds.", 3}, {"dwumies.", 8}, {"dwustr.", 7}, {"dwuszp.", 7},
  {"dwutyg.", 7}, {"dypl.", 5}, {"dyr.", 4}, {"dz.", 3}, {"dziek.", 6},
  {"dzien.", 6}, {"d\263.", 3}, {"d\263.geogr.", 9}, {"e.c.", 4},
  {"egz.", 4}, {"ekg.", 4}, {"ekw.", 4}, {"em.", 3}, {"err.", 4},
  {"ew.", 3}, {"ewent.", 6}, {"f.d\263.", 5}, {"f.kr.", 5},
  {"f.szt.", 6}, {"f.\266r.", 5}, {"fel.", 4}, {"fl.", 3}, {"flor.", 5},
  {"fot.", 4}, {"fotom.", 6}, {"fotorep.", 8}, {"fr.", 3},
  {"fragm.", 6}, {"g.", 2}, {"gat.", 4}, {"gen.bryg.", 9},
  {"gen.dyw.", 8}, {"godz.", 5}, {"gosp.", 5}, {"grub.", 5},
  {"g\263.", 3}, {"h.a.", 4}, {"h.c.", 4}, {"hon.", 4}, {"hr.", 3},
  {"i.e.", 4}, {"i.m.", 4}, {"ib.", 3}, {"ibid.", 5}, {"id.", 3},
  {"ilustr.", 7}, {"im.w\263.", 6}, {"in.", 3}, {"in\277.", 4},
  {"in\277.agr.", 8}, {"in\277.arch.", 9}, {"in\277.chem.", 9},
  {"in\277.elektr.", 11}, {"in\277.g\363r.", 8}, {"in\277.hut.", 8},
  {"in\277.in\277.", 8}, {"in\277.le\266n.", 9}, {"in\277.mech.", 9},
  {"j.", 2}, {"j.em.", 5}, {"j.es.", 5}, {"j.m.", 4}, {"jedn.", 5},
{"jez.", 4}, {"jun.", 4}, {"j\352z.", 4}, {"j\352z.oryg.", 9},
{"k.", 2}, {"k.c.", 4}, {"k.k.", 4}, {"k.o.", 4}, {"k.p.a.", 6},
{"k.p.c.", 6}, {"k.p.k.", 6}, {"k.r.", 4}, {"k.r.i o.", 8},
{"k.r.o.", 6}, {"k.tyt.", 6}, {"k.z.", 4}, {"kad.", 4}, {"kal.", 4},
{"kan.", 4}, {"kanc.", 5}, {"kand.n.", 7}, {"kard.", 5},
{"kartogr.", 8}, {"kc.", 3}, {"kk.", 3}, {"kl.", 3}, {"kol.red.", 8},
{"kom.", 4}, {"kom.red.", 8}, {"koment.", 7}, {"koresp.", 7},
{"ko\266c.", 5}, {"kpc.", 4}, {"kpr.", 4}, {"kpt.", 4}, {"kr.", 3},
{"ks.", 3}, {"ksi\352g.", 6}, {"kw.", 3}, {"kz.", 3}, {"l.", 2},
{"l.a.", 4}, {"l.at.", 5}, {"l.c.", 4}, {"l.dz.", 5}, {"l.l.", 4},
{"l.ub.", 5}, {"litogr.", 7}, {"lotn.", 5}, {"ludn.", 5},
{"m kw.", 5}, {"m.", 2}, {"m.at.", 5}, {"m.b.", 4}, {"m.cz.", 5},
{"m.in.", 5}, {"m.st.", 5}, {"m.woj.", 6}, {"maks.", 5}, {"mal.", 4},
{"margr.", 6}, {"med.", 4}, {"mgr.", 4}, {"mies.", 5}, {"mieszk.", 7},
{"mjr.", 4}, {"m\263.", 3}, {"n.", 2}, {"n.e.", 4}, {"n.p.m.", 6},
{"nak\263.", 5}, {"nb.", 3}, {"niz.", 4}, {"np.", 3}, {"nt.", 3},
{"ob.", 3}, {"obj.", 4}, {"obr.", 4}, {"obwol.", 6}, {"odb.", 4},
{"odc.", 4}, {"oddz.", 5}, {"ok.", 3}, {"okr.", 4}, {"op.", 3},
{"opr.", 4}, {"ork.", 4}, {"oryg.", 5}, {"osk.", 4}, {"o\266r.", 4},
{"p.", 2}, {"p.Ch.n.", 7}, {"p.Chr.n.", 8}, {"p.cz.", 5},
{"p.k.s.", 6}, {"p.m.", 4}, {"p.n.", 4}, {"p.n.e.", 6}, {"p.o.", 4},
{"p.p.m.", 6}, {"pchor.", 6}, {"pd.", 3}, {"pd.-wsch.", 9},
{"pd.-zach.", 9}, {"pks.", 4}, {"pkt.", 4}, {"pl.", 3}, {"plut.", 5},
{"pn.", 3}, {"pn.-wsch.", 9}, {"pn.-zach.", 9}, {"pok.", 4},
{"polit.", 6}, {"pos.", 4}, {"pow.", 4}, {"poz.", 4}, {"po\263.", 4},
{"po\263ud.", 6}, {"pp.", 3}, {"ppanc.", 6}, {"ppor.", 5},
{"pp\263k.", 5}, {"pr.", 3}, {"prez.", 5}, {"prob.", 5}, {"prof.", 5},
{"proj.", 5}, {"prosp.", 6}, {"przedst.", 8}, {"przet\263.", 7},
{"przew.", 6}, {"prze\263.", 6}, {"przyg.", 6}, {"przyw.", 6},
{"ps.", 3}, {"pt.", 3}, {"p\263.", 3}, {"p\263d.", 4}, {"p\263k.", 4},
{"p\263n.", 4}, {"p\263p\263t.", 6}, {"p\263t.", 4}, {"p\263w.", 4},
{"p\363\263sk.", 6}, {"p\363\263w.", 5}, {"q.v.", 4}, {"r.", 2},
{"r.szk.", 6}, {"r.ub.", 5}, {"radz.", 5}, {"rb.", 3}, {"rec.", 4},
{"red.nacz.", 9}, {"red.nauk.", 9}, {"ref.", 4}, {"re\277.", 4},
{"ros.", 4}, {"rozdz.", 6}, {"rtg.", 4}, {"rtm.", 4}, {"rubr.", 5},
{"ryc.", 4}, {"rz.", 3}, {"s.", 2}, {"s.c.", 4}, {"s.o.s.", 6},
{"san.-epid.", 10}, {"sekr.", 5}, {"sier\277.", 6}, {"sk.", 3},
{"sp.z o.o.", 9}, {"sp.z o.p.", 9}, {"spo\263.", 5}, {"ss.", 3},
{"st.", 3}, {"st.mar.", 7}, {"st.ogn.", 7}, {"st.sier\277.", 9},
{"st.strz.", 8}, {"st.szer.", 8}, {"stow.", 5}, {"str.", 4},
{"strz.", 5}, {"sygn.", 5}, {"szer.", 5}, {"szer.geogr.", 11},
{"sze\266c.", 6}, {"szt.", 4}, {"t.", 2}, {"t.m.", 4}, {"tab.", 4},
{"tabl.", 5}, {"techn.", 6}, {"tel.", 4}, {"telegr.", 7},
{"temp.", 5}, {"tj.", 3}, {"tow.", 4}, {"tw.", 3}, {"tw.szt.", 7},
{"tys.", 4}, {"tyt.", 4}, {"tzn.", 4}, {"tzw.", 4}, {"ub.", 3},
{"ub.m.", 5}, {"ub.r.", 5}, {"uniw.", 5}, {"ur.", 3}, {"usg.", 4},
{"v.", 2}, {"w st.sp.", 8}, {"w.", 2}, {"w.c.", 4}, {"wach.", 5},
{"wewn.", 5}, {"wf.", 3}, {"wicemin.", 8}, {"woj.", 4}, {"wsch.", 5},
{"ww.", 3}, {"wym.", 4}, {"wys.", 4}, {"wz.", 3}, {"z o.o.", 6},
{"z.", 2}, {"zach.", 5}, {"zam.", 4}, {"zat.", 4}, {"za\263.", 4},
{"zewn.", 5}, {"zm.", 3}, {"zob.", 4}, {"zw.", 3}, {"zw.zaw.", 7},
{"zw\263.", 4}, {"z\263 dew.", 7}, {"\246w.", 3}, {"\263ac.", 4},
{"\266.", 2}, {"\266p.", 3}, {"\266r.", 3}, {"\266rodk.", 6},
{"\266w.", 3}, {NULL, 0}
```

```
};
```

References

[1] Marek Adamiec. Biblioteka literatury polskiej w Internecie. Retrieved in December 2003 from http://monika.univ.gda.pl/~literat/.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Kompilatory. Reguły, metody i narzędzia.* Wydawnictwa Naukowo-Techniczne, Warszawa, 2002. (Compilers. Principles, Techniques, and Tools).

[3] Adam Drozdek and Donald L. Simon. *Struktury danych w języku C.* Wydawnictwa Naukowo-Techniczne, Warszawa, 1996. (Data Structures in C).

[4] Maciej Grochowski. Składnia wyrażeń polipredykatywnych. (Zarys problematyki). In Zuzanna Topolińska, editor, *Składnia*, pages 213–300. Państwowe Wydawnictwo Naukowe, Warszawa, 1984.

[5] Renata Grzegorczykowa. *Wykłady z polskiej składni.* Wydawnictwo Naukowe PWN, Warszawa, 1999.

[6] Andrzej Markowski, editor. *Nowy słownik poprawnej polszczyzny PWN.* Wydawnictwo Naukowe PWN, Warszawa, 2000.

[7] Alicja Nagórko. *Zarys gramatyki polskiej (ze słowotwórstwem).* Wydawnictwo Naukowe PWN, Warszawa, 2000.

[8] Edward Polański, editor. *Nowy słownik ortograficzny PWN.* Wydawnictwo Naukowe PWN, Warszawa, 1996.

[9] Adam Przepiórkowski. The IPI PAN Corpus. Preliminary version. Retrieved in April 2005 from http://dach.ipipan.waw.pl/~adamp/Papers/-2004-corpus/book_en.pdf.

[10] Zygmunt Saloni and Marek Świdziński. *Składnia współczesnego j ezyka polskiego.* Wydawnictwo Naukowe PWN, Warszawa, 1998.

[11] Jan Stanisławski. *Wielki słownik polsko-angielski*, volume II. Wydawnictwo „Wiedza Powszechna” – Philip Wilson, Warszawa, 1993. (The Great Polish-English Dictionary).

[12] Stanisław Szober. *Gramatyka języka polskiego.* Państwowe Wydawnictwo Naukowe, Warszawa, 1957.

[13] Mieczysław Szymczak, editor. *Słownik języka polskiego PWN.* Wydawnictwo Naukowe PWN, Warszawa, 1999.