MACHINE TRANSLATION REVIEW

The Periodical
of the
Natural Language Translation Specialist Group
of the
British Computer Society
Issue No. 3
April 1996

The Machine Translation Review incorporates the Newsletter of the Natural Language Translation Specialist Group of the British Computer Society and appears twice yearly.

The Review welcomes contributions, articles, book reviews, advertisements, and all items of information relating to the processing and translation of natural language. Contributions and correspondence should be addressed to:

Derek Lewis,
The Editor,
Machine Translation Review,
Department of German,
Queen's Building,
University of Exeter,
Exeter,
EX4 4QH
UK

Tel.: +44 (0) 1392 264330
Fax: +44 (0) 1392 264377
E-mail: D.R.Lewis@exeter.ac.uk

ISSN 1358–8346

Contents

Editorial: the Aims of the NLTSG

As a reminder to members and readers, we reproduce here a summary of the aims of the NLTSG:

1. The Group should exist to further the end of achieving accurate, acceptable and elegant translations between natural languages by mechanical means;

2. The Group should act as a clearing house for information on the techniques of natural language translation;

3. The Group should act as a stimulus to research in university departments, in private industry, in schools and among individuals;

4. The Group should encourage and pursue those standards which it believes will hasten the achievements of its primary aim;

5. The Group should communicate with and encourage like groups in other countries;

6. The Group should hold meetings and issue newsletters and other publications as it thinks fit.

The Committee would welcome suggestions or proposals in relation to these aims. These could range from offers to give talks, to proposals for conferences, workshops, and publications. If you have any suggestions for topics or speakers that you would like to see incorporated in the programme of events for the year, please advise us.

David Wigg and Derek Lewis

April 1996

Group News and Information

Letter from the Chairman

On the first anniversary of the 'Review' we have cause for celebration, but also cause for some concern. The former being that we are still able to publish the review and for this we must thank our Editor, Derek Lewis, for his tireless efforts. The latter is for the fact that we are getting very little material for the Review volunteered by the membership. Please could members be more forthcoming with articles for the Review? And indeed we would also welcome ideas for talks and suggestions for enlivening the group and furthering our aims (as a reminder these are listed above in the editorial).

Since the last Review we have been able to recruit two Correspondent members on to the Committee: Gareth Evans and Dan Rootham. Gareth is a Principal Lecturer in Computing at Swansea Institute of Higher Education and is interested in supporting minority languages. Dan is a Director of Lexicon Software Ltd. and he has kindly agreed to represent suppliers of M(A)T products on the Committee. Gareth and Dan will be able to supply us with valuable information from their specialist knowledge. This information may be published in the Review from time to time, but more importantly it will be stored and accumulated on our web pages at the BCS for ready access at any time (for further information on the web pages and how to access them see the article by Roger Harris in this issue).

Finally, I must apologise for the continued delay in the publication of the Proceedings of the Conference on Machine Translation held at Cranfield University in 1994. Cranfield are still awaiting compatible documentation from some of the speakers.

All opinions expressed in this Review are those of the respective writers and are not necessarily shared by the BCS or the Group.

J.D.Wigg

The Committee

The telephone numbers and e-mail addresses of the Officers of the Group are as follows:

David Wigg (Chair)	Tel.: +44 (0) 1732 455446 (Home)
	Tel.: +44 (0) 171 815 7472 (Work)
	E-mail: wiggjd@vax.sbu.ac.uk
Monique L'Huillier (Secretary)	Tel.: +44 (0) 1276 20488 (H)
	Tel.: +44 (0) 1784 443243 (W)
	E-mail: m.lhuillier@vms.rhbnc.ac.uk
Ian Thomas (Treasurer)	Tel.: +44 (0) 181 464 3955 (H)
	Tel.: +44 (0) 171 382 6683 (W)
Derek Lewis (Editor)	Tel.: +44 (0) 1404 814186 (H)
	Tel.: +44 (0) 1392 264330 (W)
	Fax: +44 (0) 1392 264377
	E-mail: d.r.lewis@exeter.ac.uk
Tania Reynolds (Assistant Editor)	Tel.: +44 (0) 1444 416012 (H)
Catharine Scott (Assistant Editor)	Tel.: +44 (0) 181 889 5155 (H)
	Tel.: +44 (0) 171 607 2789 X 4008 (W)
	E-mail: c.scott@unl.ac.uk
Roger Harris (Rapporteur)	Tel.: +44 (0) 181 800 2903 (H)
	E-mail: rwsh@dircon.co.uk
Correspondent members:	
Gareth Evans (Minority Languages)	Tel.: +44 (0) 1792 481144
	E-mail: g.evans@sihe.ac.uk
Dan Rootham (Software Suppliers)	Tel.: +44 (0) 0181 299 0067
	Fax: +44 (0) 0181 299 4338
	E-mail: 70374.1122@compuserve.com

BCS Library

Books kindly donated by members are passed to the BCS library at the IEE, Savoy Place, London, WC2R 0BL, UK (tel: +44 (0) 171 240 1871; fax: +44 (0) 171 497 3557). Members of the BCS may borrow books from this library either in person or by post. All they have to provide is their membership number. The library is open Monday to Friday, 9.00 am to 5.00 pm.

Implementing an Efficient Compact Parser for a Machine Translation System

by

J. Gareth Evans

Faculty of Computing

Swansea Institute of Higher Education

As part of a larger machine translation project at the Swansea Institute of Higher Education, a simple compact parser has been implemented. It is being used to develop a system aimed at providing first draft translations into languages which have received little or no attention to date, such as English-Welsh and English-Romanian. The system runs under Aix but ultimately a PC base is envisaged.

In designing the parser, the following criteria had to be borne in mind:

1. The parser needed to be compact; it had to be incorporated within a larger system and downloaded to a PC at a later stage. This is especially important as use within small translation companies is envisaged.
2. The parser had to be fast to ensure efficient and practical operation.
3. The parser had to be independent of any specific language. All specific language requirements had to be stored in a separate file to allow flexibility in multiple language applications.
4. Access to code and ease of maintenance were necessary to allow modification of the code as required. This would allow future developments to be made, such as statistical reasoning in the case of sentences which can be parsed in more than one way.
5. Simplicity of code was desirable in order to reduce errors and facilitate maintenance.
6. The system had to be inexpensive in terms of terms of labour and purchase costs.

In view of points 4 and 6 above, it was decided to develop an in-house parser rather than utilise an existing package. In developing the system, C was chosen as the programming language. This was for a number of reasons:

1. Fast and compact code was required.
2. A C-code parser could be easily incorporated within a larger system.
3. Specific language features would be found in files and not embedded within the parser; consequently, the Prolog-like features of language definition were not required.
4. Recursive functions in C would allow recursive language definitions to be treated easily.
5. Fast recursive searches would be allowed.
6. The developer had programming expertise in C.

The syntactic analyser of which the parser forms a key component is illustrated schematically in Figure 1. An input sentence is split into various tokens (words) and the parts of speech of these words are noted. For the purposes of this paper, the following simplified coding is used.

1	article	5	conjunction
2	preposition	55	negative
3	adjective	6	noun
4	verb	7	pronoun
41	auxiliary verb	71	relative pronoun
42	participle	8	adverb
44	infinitive	9	full stop '.'
49	'to'	91	question mark '?'

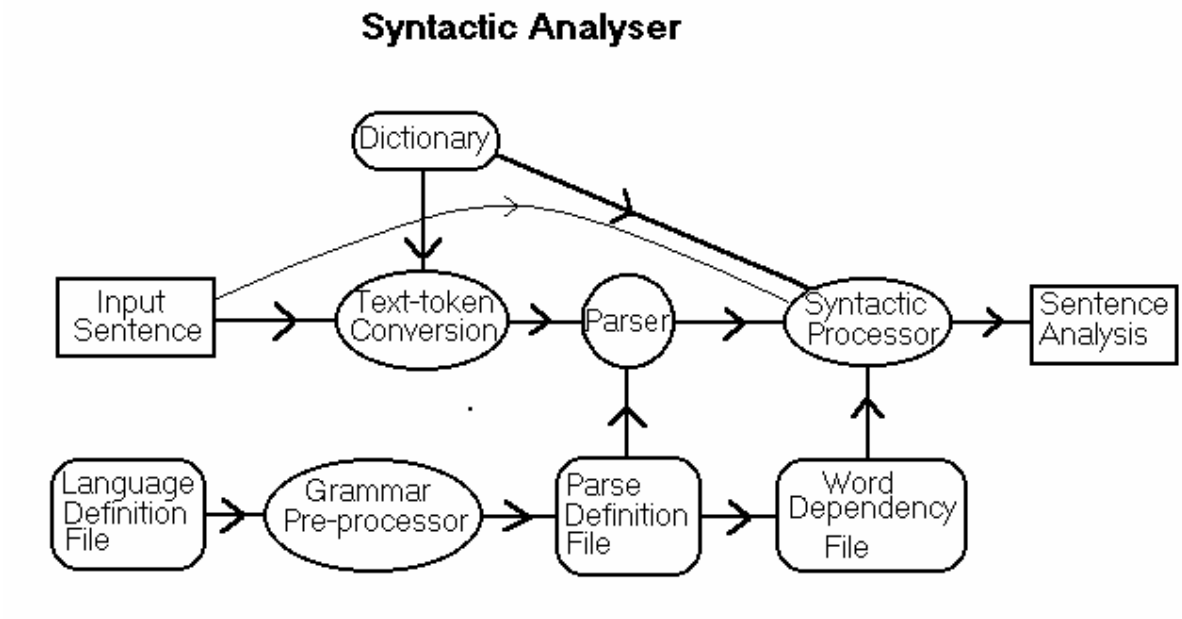

Figure 1: Schematic Diagram of Syntactic Analyser

In the case of words like 'light', where there are a number of different possible parts of speech, the system records all the possibilities. Thus, the sentence 'The man turns on the light' has three homonyms which can be of different language types, viz. *man* (verb or noun), *turns* (verb or noun), and *light* (verb, adjective or noun). As a result, the text-token converter will return 12 different sequences which are then analysed by the parser. These are as follows:

1	6	6	2	1	3
1	6	6	2	1	4
1	6	6	2	1	6
1	6	4	2	1	3
1	6	4	2	1	4
1	6	4	2	1	6
1	4	6	2	1	3
1	4	6	2	1	4
1	4	6	2	1	6
1	4	4	2	1	3
1	4	4	2	1	4
1	4	4	2	1	6

The parser will test each of these sequences against a set of definitions in the parse definition file. Only the sequence(s) which correspond to these definitions will be accepted; the others are rejected. Thus, in the example, the sequence 1 6 6 2 1 3 (article, noun, noun, preposition, article, adjective) is rejected as it does not correspond to a valid sentence.

The Language Definition File

The syntactical definition of the language to be parsed is given by a language definition file, whose syntax is loosely based on the Backus-Naur form. A simple example of such a file is provided below. The first line contains the number of subsequent definitions. Alternative definitions for each grammatical component are found as items on separate lines. Further, only two components are allowed on the right-hand side of each definition.

Sample Language Definition File

```
38
<sentence>::=<indicative><9>
<indicative>::=<indicative><link_indicative>
<indicative>::=<noun_phrase><verb_phrase>
<link_indicative>::=<5><indicative>
<verb_phrase>::=<verb_phrase><link_verb_phrase>
<link_verb_phrase>::=<5><verb_phrase>
<verb_phrase>::=<sverb_phrase><8>
<verb_phrase>::=<sverb_phrase>
<sverb_phrase>::=<verbal_form>
<sverb_phrase>::=<verbal_form><noun_phrase>
<sverb_phrase>::=<verbal_form><subordinate_clause>
<noun_phrase>::=<noun_phrase><link_noun_phrase>
<link_noun_phrase>::=<5><noun_phrase>
<noun_phrase>::=<noun_phrase><prep_phrase>
<noun_phrase>::=<prep_phrase>
<noun_phrase>::=<prep_phrase><noun_phrase>
<noun_phrase>::=<noun_phrase><subordinate_clause>
<prep_phrase>::=<2><noun_phrase>
<noun_phrase>::=<1><simple_noun_phrase>
<noun_phrase>::=<simple_noun_phrase>
<noun_phrase>::=<1><adverbial_phrase>
```

<noun_phrase>::=<7>
<adverbial_phrase>::=<8><3>
<simple_noun_phrase>::=<3><simple_noun_phrase>
<simple_noun_phrase>::=<6>
<verbal_form>::=<4>
<verbal_form>::=<41><participle_form>
<participle_form>::=<55><participle_form>
<participle_form>::=<42>
<verbal_form>::=<4><infinitive>
<infinitive>::=<49><44>
<subordinate_clause>::=<71><sclause1>
<subordinate_clause>::=<71><sclause2>
<subordinate_clause>::=<sclause2>
<sclause1>::=<verb_phrase>
<sclause2>::=<noun_phrase><restricted_vp>
<restricted_vp>::=<restricted_vp><prep_phrase>
<restricted_vp>::=<verbal_form>

The numbers enclosed in <> refer to specific parts of speech. The various grammatical forms are composed of two types: primitive forms such as <noun_phrase> corresponding to recognised grammatical structures; and additional forms such as <restricted_vp> which are of use in defining the primitive forms in the specified manner. The primitive form <sentence> forms the top-level definition for each set of tokens to be examined. The primitive forms are those which are likely to be conserved in any transformation to a target language.

Although considerably more complicated language definition files are being used with this parser, for the purposes of illustration the above language definition will suffice. Sentences such as the following conform to the above syntax:

I saw the pretty girl.
The thin man with the dog likes long walks in the rain.

The fact that only two components are allowed on the right hand side of the definitions is not a major restriction as multiple definitions with more than two right-hand components can easily be rewritten in the correct format. For example, the additional definitions

<sentence>::=<indicative><91>|<question1><91>|<question2><91>
<question1>::=<41><noun_phrase><42><noun_phrase>
<question2>::=<72><verb_phrase>

will allow valid sentences to be of the form

Is the big man hitting the small boy?
Who is selling newspapers at the market?

These may be easily transformed (manually or using a simple program) to the required language definition file format:

<sentence>::=<indicative><91>
<sentence>::=<question1>
<sentence>::=<question2>
<question1>::=<41><m_generated1>
<m_generated1>::=<noun_phrase><m_generated_2>
<m_generated_2>::=<42><noun_phrase>

<question2>::=<72><verb_phrase>
where <m_generated1> and <m_generated2> are further additional forms.

The Intermediate Definition File

A pre-processor converts the language definition file into an intermediate form which can be used directly by the parser. The converted file format is as follows:

```
 18                 Total no. of primitive and additional forms
100   1            No. of entries for <sentence>  =  1      (code = 100)
101   9                  Definition for 100
101   2            No. of entries for <indicative> = 2      (code = 101)
101 102                  Definition 1 for 101
106 103                  Definition 2 for 101
102   1            No. of entries for <link_indicative> = 1  (code = 102)
  5 101                  Definition of 102
103   3            No. of entries for  <verb_phrase> = 3   (code = 103)
103 104                  Definition 1 for 103
105   8                  Definition 2 for 103
105   0                  Definition 3 for 103
104   1            No. of entries for <link_verb_phrase>  = 1  (code = 104)
  5 103                  Definition for 104
105   3            No. of entries for  <sverb_phrase> = 3  (code = 105)
111   0                  Definition 1 for 105
111 106                  Definition 2  for 105
111 114                  Definition 3 for 105
106   9            No. of entries for <noun_phrase> = 9  (code = 106)
.....
.....
.....
```

The comments in italics are not part of the intermediate file and are simply included here for the sake of clarity. In the above example there are 18 different grammatical types with associated codes 100–117.

No.	Name	No. of Definition Lines
100	sentence	1
101	indicative	2
102	link_indicative	1
103	verb_phrase	3
104	link_verb_phrase	1
105	sverb_phrase	3
106	noun_phrase	9
107	link_noun_phrase	1
108	prep_phrase	1

No.	Name	No. of Definition Lines
109	adverbial_phrase	1
110	simple_noun_phrase	2
111	verbal_form	3
112	participle_form	2
113	infinitive	1
114	subordinate_clause	3
115	sclause1	1
116	sclause2	1
117	restricted_vp	2

Each entry consists of a header line containing the type number and the number of definition lines for that type. The definition lines contain two entries corresponding to definitions from the language definition file. These entries are either token types (1–99) or grammatical types (>=100).

The Parser

The parser reads data from the intermediate definition file. The following simple processing stores on-line the definitions found in the original language definition file within two tables or arrays:

Array ling_def1

Index	Ling Type	No of entries	Starting Point
0	100	1	0
1	101	2	1
2	102	1	3
3	103	3	4
4	104	1	7
5	105	3	8
6	106	9	11
.			
.			
.			
.			
17	117	2	36

Array ling_def2

Ling Type	Index	Definition Components	
100	0	101	9
101	1	101	102
	2	106	103
102	3	5	101
103	4	103	104
	5	105	8
	6	105	0
104	7	5	103
105	8	111	0
	9	111	106
	10	111	114
.....			

In the above example, array ling_def1 indicates that there are three definitions for verb_phrase (category 103) and these are found in array ling_def2 starting at index 4.

By storing the language definitions in this way, the parser itself is independent of the grammar of any particular language. All attempts at parsing are made against the definitions found in ling_def2. This means that the parser is not dependent in any way on the source language and is a reusable component for further development. Thus alternative improved

definitions for the current language may be easily incorporated into the overall system as well as definitions for other languages.

The parser consists essentially of a function *test_ling (ling_type, start_point, end_point)* which tests whether a sequence of numbers makes up the required grammatical type. For example, 1 3 6 should form a noun phrase (type no. 106). Thus, test_ling(106,0,2) should return a true (1) value for the token sequence 1 3 6 4 3 6 9.

The test for a language component is performed by examining the sequence under consideration against the set of definitions for that component. Since each language component is defined by only one or two components the options available for examination are strictly limited.

1. Primitive	-		e.g.	<4>
2. Compound	-			<prep_phrase>
3. Primitive	Primitive			<8><3>
4. Primitive	Compound			<5><verb_phrase>
5. Compound	Primitive			<indicative><9>
6. Compound	Compound			
<noun_phrase><prep_phrase>				

The sequence of numbers is tested recursively to see whether it conforms to the permissible definitions which are only of the above forms. The test for primitives is straightforward since each of the numbers in the sequence is a primitive. Compounds are tested recursively in a depth-first, modified, left-right (L-R) manner.

Example

Consider an input sentence is of the form 1 3 6 4 16 9. The call test_ling with parameters (100,0,6) will examine the sequence against the single definition 101 9 corresponding to the first entry in array ling_def2. Since the primitive 9 matches, the next stage is to test the sequence 1 3 6 4 1 6 as an indicative (language type 101). This is easily done by the recursive call test_ling with parameters (101,0,5).

In the case of possibility 6, i.e. compound-compound, a break must be introduced in the sequence at each possible point and a series of possibilities examined. For example, if the phrase ‘the little house on the corner’ is to be tested for a <noun_phrase> using definition <noun_phrase><prep_phrase>, then the following tests must be performed:

<noun_phrase>	**<prep_phrase>**
the	little house on the corner
the little	house on the corner
the little house	on the corner
the little house on	the corner
the little house on the	corner

It will be found that only the third combination will satisfy the tests.

The combinatorial explosion that results from examining fully every possibility in this way can result in unacceptably large search times for longer sentences. In order to overcome this problem a form of lazy evaluation is adopted whereby testing for a linguistic category is only

undertaken if there is a possibility of success at a higher level. Thus, in the above example, only the third combination passes the <noun_phrase> test. No attempt is made to test for <prep_phrase> except in this case, as the effort will prove futile: the overall definition cannot be satisfied if there is a failure on one count.

Further, in testing for compound-primitive sequences, the test for the correct primitive is performed prior to the test for the compound. These simple devices reduce parse times considerably.

Once the parse function has been able to parse a sentence successfully, then details of the analysis must be available for further analysis. Each time a call to the function test_ling succeeds, i.e. a linguistic category is successfully found, details are stored in a global array. These details include:

- The linguistic type found
- The start and end points of the number sequence
- The point at which a break between two categories was made.

If the attempt to find a successful parsing for an entire sentence succeeds, the stored information is used to generate a parse tree (stored appropriately in an array). The contents of the array are then scrutinised in a top-down manner starting with the information provided from the successful outcome of the top-level function call to determine whether the full sequence of numbers represents a sentence. Rows within the array that contribute to the successful parsing are marked and from these the parse tree can be constructed. The contents of these rows form the basic information passed on in order to further analyse the sentence. In some instances, more than one possible parsing of a sentence is possible. In this case, the various possible outcomes may be passed on to the next stage in the process.

The following provides a log of running a program which utilises the parser. It displays the contents of the array which contains details of the parse tree.

what is sentence length? *11*
type in sequence
1 3 6 4 1 3 6 2 1 6 9
valid sequence supplied
number of solutions is 1
number of branches is 1

Line Index	Language Component	Start Point	End Point	Break Point	Definition	
0	110	2	2	2	6	0
1	110	1	2	1	3	110
2	106	0	2	0	1	110
3	111	3	3	3	4	0
4	110	6	6	6	6	0
5	110	5	6	5	3	110
6	106	4	6	4	1	110
7	110	9	9	9	6	0
8	106	8	9	8	1	110
9	108	7	9	7	2	106

Line Index	Language Component	Start Point	End Point	Break Point	Definition	
10	106	4	9	6	106	108
11	105	3	9	3	111	106
12	103	3	9	0	105	0
13	101	0	9	2	106	103
14	100	0	10	9	101	9

The last line contains details of the sentence as a whole. Working backwards, the parse tree can be easily obtained.

sentence (0-10)
line 14

indicative (0-9) '.' (10)
line 13

noun phrase (0-2) verb phrase (3-9)
line 2 *line 12*

article (0) simple noun phrase (1-2) sverb_phrase (3-9)
line 1 *line 11*

adjective (1) simple noun phrase (2) verbal_form (3)
line 0 *line 3*
noun (2) verb (3) noun_phrase (4-9)

line 10

noun_phrase (4-6) prep_phrase (7-9)
line 6 *line 9*

article (4) simple_noun_phrase (5-6) preposition (7) noun_phrase (8-9)
line 5 *line 8*

adjective (5) simple_noun_phrase (6) article (8) simple_noun_phrase (9)
line 4 *line 7*
noun (6) noun (9)

This parse tree (and hence the array which generated it) contains all the information necessary for describing the syntactic structure of the sentence. This tree (or multiple possible trees) may be passed on to other components of the system for further analysis.

Conclusion

A simple parse program has been described which is suitable for inclusion into larger NLP and MT systems. The program has the advantage of being fast, compact, and source language

independent. It is suitable as a component for larger systems. A copy of the parser program, together with the language file pre-processor, may be obtained from the author.

Appendix: Program Listing

```
#include <stdio.h>
#include <time.h>

int ling_def1[120][2],ling_def2[200][2];
int s[20],slength,parsed[100][10],pindex,level,sol_num,branches;
int sub_count;

main()
{
int t,i,j,k,m,n,x,start,end,elapsed;
char c;
FILE *f1;

/* reads data from grammar definition file */

f1=fopen("gramdef1","r");
fscanf(f1,"%d",&n);
for(i=0;i<n;i++){
        fscanf(f1,"%d  %d",&m,&ling_def1[i][0]);
        if((m-100)!=i)printf("error in input\n");
        if(i==0)ling_def1[0][1]=0;
          else ling_def1[i][1]=ling_def1[i-1][1]+ling_def1[i-1][0];
        for(j=0;j<ling_def1[i][0];j++)
                fscanf(f1,"%d  %d",&ling_def2[j+ling_def1[i][1]][0],
                        &ling_def2[j+ling_def1[i][1]][1]);
           }

/* type in a sequence for parsing  */

printf("what is sentence length? ");
scanf("%d",&slength);
printf("type in sequence \n");
for(i=0;i<slength;i++) scanf("%d",&s[i]);
/* test for valid sequence */
pindex=0;
level=0;
start=time(0);
sub_count=0;
t=test_ling(100,0,slength-1);
 if(t==1)printf("valid sequence supplied\n");
   else printf("invalid sequence supplied\n");
```

```
if(t==1){for(i=0;i<pindex;i++){x=parsed[i][3];
                parsed[i][5]=ling_def2[x][0];
                parsed[i][6]=ling_def2[x][1];
                parsed[i][7]=0;
                parsed[i][9]=0;
                        }

if(t==1){ for(i=0;i<pindex;i++)
        for(j=0;j<pindex;j++)
        if((parsed[i][0]==parsed[j][0])&&(parsed[i][1]==parsed[j][1])
            &&(parsed[i][2]==parsed[j][2])&&(parsed[i][8]>1))
              parsed[j][9]=1;
      }

sol_num=1;
branches=1;
for(i=0;i<pindex;i++){sol_num=sol_num+parsed[i][8]-1;
      if(parsed[i][8]>1)branches++;
                }
scanf("%c",&c);
printf("\n\n");
printf("number of solutions is %d \n",sol_num);
printf("\n");
printf("number of branches is %d\n\n",branches);
printf("\n");
if(sol_num>2)printf("number of solutions too great\n");
else {
for(k=0;k<sol_num;k++){
for(i=0;i<pindex;i++)parsed[i][7]=0;
mark_line(pindex-1,pindex-1,k);
printf("\n\n");
for(i=0;i<pindex;i++){ if(parsed[i][7]==1)
      printf("%5d %5d %5d %5d %5d %5d \n",parsed[i][0],
 parsed[i][1],parsed[i][2],parsed[i][4],parsed[i][5],parsed[i][6]);
                        }
                  }
    }
}
end=time(0);
elapsed=end-start;
printf("\n time taken = %d\n",elapsed);
}

test_ling(ling_type,start_point,end_point)
int ling_type,start_point,end_point;
{
int x,y,z,t,u,search,br,success,brk,local_index,l_index1,finds;
```

```
sub_count++;
/* printf("%d\n",sub_count); */
level++;
t=0;
brk=0;
finds=0;
x=ling_def1[ling_type-100][1];
y=x-1+ling_def1[ling_type-100][0];
for(search=x;search<=y;search++){
local_index=pindex;
u=0;z=0;brk=start_point;
if(ling_def2[search][1]==0){if(end_point>=start_point){
                    /* search for p0 or c0  */
                   if(ling_def2[search][0]<100){
                      /* search for p0         */
        if((start_point==end_point)&&(s[start_point]==ling_def2[search][0]))
                                        {t=1;u=1;z=1;}
                                        }
                    else                  {
                      /*   search for c0    */
        if((test_ling(ling_def2[search][0], start_point,end_point))) {t=1;u=1;z=2;}
                                        }
                                        }
                  }
      else {if(end_point>=(start_point+1)){
          if ((ling_def2[search][0]<100)&&(ling_def2[search][1]<100)){
              /* search for pp  */
       if ((start_point+1==end_point)&&(s[start_point]==ling_def2[search][0])
                  &&(s[end_point]==ling_def2[search][1])) {t=1;u=1;z=3;}
                                              }
          if ((ling_def2[search][0]<100)&&(ling_def2[search][1]>=100)){
              /* search for pc  */
          if (ling_def2[search][0]==s[start_point])
            if  (test_ling(ling_def2[search][1],start_point+1,end_point)){t=1;u=1;z=4;}
                                              }
          if ((ling_def2[search][0]>=100)&&(ling_def2[search][1]<100)){
              /* search for cp  */
          if (ling_def2[search][1]==s[end_point])
            if  (test_ling(ling_def2[search][0],start_point,end_point-1))
                          {t=1;u=1;z=5;brk=end_point-1;}
                                              }
          if ((ling_def2[search][0]>=100)&&(ling_def2[search][1]>=100)){
              /* search for cc  */
               for(br=start_point;br<end_point;br++){l_index1=pindex;
           if((br-start_point)<(end_point-br)){
              if(test_ling(ling_def2[search][0],start_point,br)){
              if  (test_ling(ling_def2[search][1],br+1,end_point))
                            {t=1;u=1;z=6;brk=br;}
                 else pindex=l_index1;}
```

```
                else pindex=l_index1; }
                else  {
              if  (test_ling(ling_def2[search][1],br+1,end_point)){
              if(test_ling(ling_def2[search][0],start_point,br))
                                {t=1;u=1;z=6;brk=br;}
                 else pindex=l_index1;}
                else pindex=l_index1;
                     }
                 }
                                                    }
            }
          }
if(u==1){finds++;
       success=search;
       parsed[pindex][0]=ling_type;
       parsed[pindex][1]=start_point;
       parsed[pindex][2]=end_point;
       parsed[pindex][3]=success;
       parsed[pindex][4]=brk;
       pindex++;
      }
if(u==0)pindex=local_index;
}
if(t==1)parsed[pindex-1][8]=finds;
level--;
return t;
}

mark_line(range,line_no,exclusion)
int range,line_no,exclusion;
{ int i,mark1,mark2,count1,count2;
i=0;
count1=0;
count2=0;
parsed[line_no][7]=1;
if(parsed[line_no][5]>=100) count1++;
if(parsed[line_no][6]>=100) count2++;
mark1=0;
mark2=0;
if ((count1+count2)>0)
do{    if((parsed[i][9]==0)||(parsed[i][8]==(exclusion+1)))
         if(parsed[i][7]!=1)
                {
         if((parsed[line_no][5]>=100)&&(parsed[line_no][6]<100)){
              if(parsed[line_no][6]!=0){ /* tidy up cp  */
              if((parsed[i][0]==parsed[line_no][5])&&
                (parsed[i][1]==parsed[line_no][1])&&
                (parsed[i][2]==parsed[line_no][2]-1))
                        {mark_line(range,i,exclusion);mark1++;}}
```

```
                else{ /* tidy up c0  */
              if((parsed[i][0]==parsed[line_no][5])&&
                (parsed[i][1]==parsed[line_no][1])&&
                (parsed[i][2]==parsed[line_no][2]))
                        {mark_line(range,i,exclusion);mark1++;}
                    }
                            }
          if((parsed[line_no][5]<100)&&(parsed[line_no][6]>=100)){
              if(parsed[line_no][5]!=0){  /* tidy up  pc  */
              if((parsed[i][0]==parsed[line_no][6])&&
               (parsed[i][1]==(parsed[line_no][1]+1))&&
                (parsed[i][2]==parsed[line_no][2]))
                        {mark_line(range,i,exclusion);mark2++;}}
                            }
          if((parsed[line_no][5]>=100)&&(parsed[line_no][6]>=100)){
                           /* tidy up cc */
              if(mark1==0){
              if((parsed[i][0]==parsed[line_no][5])&&
                (parsed[i][1]==parsed[line_no][1])&&
                (parsed[i][2]==parsed[line_no][4]))
                        {mark_line(range,i,exclusion);mark1++;}}
              if(mark2==0){
              if((parsed[i][0]==parsed[line_no][6])&&
               (parsed[i][1]==(parsed[line_no][4]+1))&&
                (parsed[i][2]==parsed[line_no][2]))
                        {mark_line(range,i,exclusion);mark2++;}}
                            }
                }
i++;
} while ((i<=range)&&((mark1<count1)||(mark2<count2)));
}
```

Gareth Evans may be contacted at the Faculty of Computing, Swansea Institute of Higher Education, Mount Pleasant, Swansea SA1 6ED. E-mail: g.evans@sihe.ac.uk

USING ICON FOR TEXT PROCESSING

by

David Quinn

Centre for Applications of Advanced IT

BICC

Introduction

Icon is a general purpose programming language with high level semantics. It includes extensive features for processing strings and data structures, a novel expression evaluation mechanism, and automatic storage management. It is an imperative, procedural language.

Icon is especially suitable for tasks involving string processing. It was much influenced by SNOBOL, the original string-manipulation language, but the two languages are very different.

Icon is designed for ease of programming. It is particularly useful for text processing tasks, as a prototyping tool, and for experimental applications. Its high level nature also makes it suitable for large, complex applications.

The name *Icon* was chosen and established before the current usage of *icon* in graphical user interfaces. As quickly becomes clear, the two are not related. The name has no particular meaning, 'although the word "iconoclast" was immediately offered as describing the flavour of the new language' (Griswold and Griswold 1993: 54).

Icon is developed, implemented, distributed and supported by The Icon Project, which derives support from The University of Arizona and is not commercial. Much of the earlier funding was provided by the National Science Foundation (USA).

The major reference on Icon is the book (Griswold and Griswold 1990), which describes Version 8.0 of the language. This book is supplemented by the report (Griswold, Jeffery and Townsend 1995) on Version 9.1, the current version. Useful short overviews of Icon are Griswold 1995 and Griswold 1994, while Griswold, Townsend et al. 1996 — which focuses on frequently asked questions — is a good source of general information. The in-depth history of Icon is presented in Griswold and Griswold 1993. This paper draws on all these sources.

The two longer Icon code examples in this paper are adapted from programs written in the CISAU project. The project and these programs are presented in the last two sections of the paper.

Expression Evaluation

On evaluation, an Icon expression either *succeeds* or *fails*. If it fails, then it produces no value. For example, in the expression

```
word == "concrete"
```

the variable word and the string literal "concrete" are themselves expressions. If they both produce the value "concrete", then the whole expression succeeds, with value "concrete". If the two values are not equal then the expression as a whole fails, producing no value.

This notion of success or failure is much broader than that of a Boolean truth value, and permeates the whole of Icon. Even statements such as

```
if (word == "concrete") then
    write("Equal")
```

are Icon expressions.

If an expression has another expression nested within it, then the value of this inner expression serves as an argument to the outer expression. For example, the expression

```
write(find("the", line))
```

writes the first position at which the string "the" is found in the variable line.

If the inner expression fails, then it provides no value for this argument, and the whole expression fails. Thus, the above expression will fail if "the" does not occur in line, as it will have no argument to write.

On succeeding, an expression may produce a single result, or it may *generate* a sequence of results, one at a time. For example, the above expression find("the", line) is a generator, and produces all positions at which the string "the" is found. Generators return only one value at a time, and then *suspend* and may be *resumed*.

Icon provides *goal-directed evaluation*. In the case of failure during expression evaluation suspended generators are resumed to produce more values. In the following example, the first (smallest) value returned by the find expression may cause the if condition to fail. Each time this occurs the find expression is resumed, and its next value tried in the if condition. It is resumed until either the outer expression succeeds, or all values have been generated.

```
if (11 <= find("the", line) <= 20) then
    write("Found between positions 11 and 20")
```

Goal-directed evaluation provides *control backtracking*, where evaluation backtracks on failure to resume one or more generators, using depth first search. This is reminiscent of Prolog (Clocksin and Mellish, 1994).

Icon provides its own versions of conjunction and alternation, which use the respective symbols & and |. Alternation, for example, may be interpreted as disjunction (only one expression need succeed) or as generation of all alternatives.

Example: Expression Evaluation

```
every(write(find(("The" | "the"),
    "The child ate the apple. There was only one.")))
```

In the example, the first argument of find is the expression ("The" | "the"), an alternation which generates the two strings in turn. The first position at which "The" is found is written. The function every then repeatedly resumes the inner expression. The function write is not a generator, but the find expression generates all positions at which "The" occurs, which are

output in turn by write. The alternation is then resumed, generating "the". The positions at which "the" is found are then generated, and written. Thus, the expression writes the positions 1 ("The"), 26 ("The" in "There"), and then 15 ("the").

Strings

In Icon a string is a sequence of characters and is a simple (first class) data type. Natural language words, sentences, lines, or complete texts may be stored and processed as strings.

String *positions* occur between characters. As illustrated below for the string "horse", the positions may be numbered from the beginning or the end.

```
  h  o  r  s  e
1 2  3  4  5  6
5 -4 -3 -2 -1 0
```

For example, given a variable word with string value "horse", word[1:3] refers to the *substring* "ho", the characters between positions 1 and 3. A single character is referenced by the position to its left, and word[4] or word[-2] both refer to "s", the 4th or 2nd last character.

Thus, prefixes and suffixes may be handled with equal ease. Right-to-left languages may, in principle, be handled as easily as left-to-right ones. However, the character set supported by an Icon implementation is machine-dependent, and usually based on ASCII (or sometimes EBCDIC). Neither of these includes characters for more extensive alphabets than English, or for non-Latin alphabets. Nevertheless, the Unix implementation on my machine, for example, supports an extended ASCII character set which does include the extra characters in the main European Latin alphabets, though the sorting sequences are unreliable.

Strings are built up using the *concatenation* operator ||. For example,

```
"abc" || "def"
```

yields the string "abcdef".

String Analysis

Icon provides various string analysis functions. For example, the function find introduced earlier locates occurrences of one string within another. The expression

```
find("ing","string processing")
```

generates the positions (4 and 15) at which "ing" occurs in "string processing".

Where functions such as find locate a specific sequence of characters (a string) within another string, functions such as many and any work with *character sets*, written between single quotes. For example

```
many('aeiou',"ouija")
```

produces the position (4) in "ouija" after the longest initial sequence of one or more of any of the characters from the set (the lower case vowels). The function any is similar, and produces the position in the string after any single character from the set.

Icon provides a number of built-in character sets. Sets such as &ucase, &lcase and &letters, consisting respectively of upper case, lower case, and all (English) letters, are particularly useful for text applications.

String Scanning

A string may be made the *subject* of a string scanning expression. For example, in

"string processing" ? {tab(find("s") & move(1) & any('aeiou')}

the string on the left of the ? is the subject of the scanning expression on its right. This expression finds the first "s", tabs to its position, moves one further character (to the position after the "s"), and then looks for any lower case vowel. The scanning expression returns the value of the last expression in the conjunction, which in this case is the *position* after the first vowel immediately following an "s" (position 16, after the "i" in "processing").

String scanning allows the use of scanning expressions of arbitrary complexity, without repeating the subject as an argument of each function. It further allows for the *position* within the subject on which string analysis is taking place to be shifted during processing. In the example above, many operates on the subject beginning at the position after each "s" is found.

Matching functions change the position in the subject and return the substring between the old and new positions. For example, the function tab(i) changes the position of the subject to i, and returns that part of the subject from the current position up to position i.

Example: String Processing

Figure 1 below is an example adapted from one of the CISAU project programs, illustrating the application of string processing to *tokenisation*. Tokenisation is the segmentation of an input text into groups of characters, or tokens. For example, words, numbers and punctuation marks are convenient tokens.

This example is a complete Icon program, though with limited functionality. The symbol # is used for comments, which continue to the end of the line.

The procedure main() has a simple loop to read the input text one line at a time, and process all tokens in each line. The processing, done in procedure process(), is limited to writing the token. The token is recognised by procedure next_token(), which tries various token recognition procedures in turn.

```
procedure main()   #   Process all lexical, punctuation and unrecognised tokens in a text.
    while line := read() do   line ?              # Read the input one line at a time.
      while process(next_token())                 # Process all tokens in the current line.
    return
end

procedure process(token)                          # Simple processing procedure (writes the
token).
  write(token)
  return
end
```

```
procedure next_token()              # Returns next token.
  tab(many(' '))                        # Skip any leading space(s)
  return {
    lexical_token() |
    punctuation() |
    # Other types of token ... |
    unrecognised()                      # Token not recognised as any of the above.
    }
end

procedure lexical_token()           # Return next lexical token, or fail.

  return {
    tab(many(&letters)) ||                  # One or more letters
                                            # followed by
          ( ="-" || tab(many(&letters)) |   # A hyphen followed by letter(s); or
          "" ) ||                           # nothing;

                                            #  followed by

    ( =""" || tab(any('sS')) |                  #  's or 'S; or
      ="(" || tab(any('sS')) || =")" |          # (s) or (S); or
      ="(" || tab(any('eE')) || tab(any('sS')) || =")" |    # (es) or (ES) or (eS) or (Es); or
      "" )                                      # nothing.
   }

end

procedure punctuation()             # Return punctuation character or fail.
  return tab(any(',;:.?!()[]/-\'\'"'))  #  , ; : . ? ! ( ) [ ] / - ' or "
end

procedure unrecognised()            # Return next character.
  return move(1)
end
```

Figure 1: Tokenisation Program

The core of the example is the procedure lexical_token(), which recognises lexical tokens (i.e. word forms). The lexical token is built up using concatenation. It begins with one or more letters. This is followed, optionally, by a hyphen and one or more letters. The final part, also optional, is a single suffix for possession or bracketed pluralisation. These latter forms are neither single nor plural, but rather both at the same time: for example, *aggregate(s)*. The example only deals with the main noun forms.

The procedure punctuation() recognises any of a set of characters that are punctuation marks. For example, the characters **-** and **'**, if not already incorporated as a *hyphen* or *apostrophe* within a lexical token, will be recognised as the punctuation tokens *dash* or *single quote*.

If no other type is recognised, the procedure unrecognised() returns the next character as a token.

High Level Data Structures

Icon has lists, sets, tables and records.

Built-in functions allow Icon *lists* to be accessed as arrays (subscripting by position), stacks (access from one end only), queues (insertion at one end, and retrieval from the other), and dequeues (double-ended queues — access from both ends). Icon includes the basic access functions for using lists in the fashion of LISP (for example, Winston and Horn 1989), though it uses a different metaphor. In Icon, the list functions are analogous to those for strings. For example, list[1] returns the first element (without removing it — the LISP function car); list[2:0] returns the rest of the list (cdr); push() inserts a new element at the front of the list (cons); and ||| concatenates two lists (append).

An Icon *table* is a collection of keys and an associated value for each. The table is accessed through the key. Both the keys and values may be of any type. For example

```
frequency[word] +:= 1
```

adds one to the value of the key word in the frequency table. Further,

```
every word := key(frequency) do
    write(word, " (", frequency[word], ")")
```

generates each key of the frequency table in turn (the word), and for each writes the word followed by its frequency count in parentheses.

A *record* provides access by named fields. For example

```
record lexical_data(word, lemma, part_of_speech)
```

creates the record type lexical_data, with the three named fields.

```
item := lexical_data("canaries", "canary", "noun")
```

creates a variable with values for the corresponding fields. Access to the value of a field is obtained as in the example for lemma:

```
item.lemma
```

A *set* is an unordered collection of values, with associated functions. Character sets, such as 'aeiou', have already been introduced.

Example: High Level Data Structures

In the following example, adapted from one of the CISAU project programs, high level data structures are used for the accumulation of structured information. The information is about tokens (for example, *Apple*, *APPLE*, *apple, 48*) and the words to which the tokens correspond, by the mapping of all letters to lower case (for example, *apple* or *48*).

A *table* keyed by word is used, and each entry has as value a *record* containing the word frequency, number of tokens (number of distinct tokens grouped under this word), and a *list* containing, for each token mapped to this word, a further *record* containing the token itself, its frequency, and an embedded *list* of token occurrences (corpus index positions).

First, the component fields of the records word_data and token_data are set out in the following record declarations:

```
record word_data(frequency, no_of_tokens, token_data_list)
                                # Word data record definition

record token_data(token, frequency, occurrence_list)
                                # Token data record definition
```

Next, the table word_data_table is declared as a global variable (this is convenient, but not strictly necessary):

```
global word_data_table          # Global declaration of word
                                # data table
```

The table is then initialised:

```
word_data_table := table()# Initialise table
```

The example procedure tabulate() appears in figure 2, on the following page. The arguments to this procedure, with typical values alongside, are as follows: word — “apple”; token — “APPLE”; frequency — 3; and occurrence_list — [15 ,86 ,123].

The procedure works as follows. The word is looked up as a key in the table, and a pointer to its entry assigned to the variable index. If index has no value then the word does not yet have an entry in the table, and it is added along with its value. The value of each word entry in the table is as described in the second paragraph of this section. As this is the first entry, the number of tokens is set to 1.

Else, where index has a value, the existing entry for the word must be updated. The frequency of the word is incremented by the frequency of the token. The number of tokens is incremented by one. The token data (a record consisting of the token, its frequency, and a list of its occurrences) is added to the end of the token data list.

The following example illustrates the declaration and updating of structured information. Further Icon code would access the table and output the accumulated results.

```
procedure tabulate(word, token, frequency, occurrence_list)    #   Update table of words

    local index                             # Pointer to entry in word data table.

    index := word_data_table[word]          # Look up entry for word in the table.
    if /index then                          # No previous occurrence of this word
                                            #  in the table — add new table entry
        word_data_table[word] :=
            word_data(frequency, 1, [token_data(token, frequency, occurrence_list)])
    else {                                  # Update word data entry
        index.frequency +:= frequency       # Add token frequency to word frequency
        index.no_of_tokens +:= 1            # Add 1 to number of tokens
        put(index.token_data_list, token_data(token, frequency, occurrence_list))
                                            # Add token data to token data list
        }
    return
```

```
end
```

Figure 2: procedure tabulate()

Icon Implementations

The first implementation of Icon, for UNIX, was done at the University of Arizona. Other implementations have been performed by volunteers from around the world. The source code for Icon is written in C. All implementations are based on the same source code, and this contributes to Icon's portability.

The latest version of Icon (version 9) runs under UNIX, Macintosh/MPW, MS-DOS, VAX/VMS, and on the Acorn Archimedes. Current projects include implementations for Microsoft Windows, Windows NT, and a new Macintosh implementation. Earlier versions of Icon have been implemented on several other platforms. The notable changes in version 9 of Icon are in the area of support for graphics facilities.

The Icon interpreter is recommended for program development and also for most production situations. The Icon compiler requires a large amount of resources to run, but produces code that is around two or three times faster.

The Icon Program Library

There is a library of Icon procedures and programs. It contains hundreds of procedures, which supplement the built-in functions, as well as complete application programs. The library is useful both as a source of examples for helping to learn the language, and as a resource for Icon programming.

Obtaining Icon

Icon is available free by anonymous ftp and on the World Wide Web (WWW). It is also available on diskettes from the Icon Project, at a nominal charge, currently ranging up to 25 US dollars. The ftp address is

ftp.cs.arizona.edu

and the Icon directory is reached using the command

cd /icon

The WWW URL is

http://www.cs.arizona.edu/icon/www/

The answers to frequently asked questions (FAQs) about Icon (Griswold, Townsend et al., 1996), which include fuller details of how to obtain the package, are also available at the above WWW site.

The CISAU Project

Icon has been used for writing corpus analysis programs in the CISAU project (Construction Industry Specification Analysis and Understanding). This project is developing a system for automatically checking construction domain documents for errors. Information extraction enables cross-checking of project documents for incorrect or inconsistent values. The

documents use a specialised engineering sublanguage, with restricted vocabulary, many incomplete and elliptical constructions, and list and tabular structures. See Douglas, Hurst and Imlah (1995) and Quinn (1996) for further details of the CISAU project.

The CISAU Corpus Analysis Programs

The two longer examples presented in this paper were adapted from the CISAU project corpus analysis programs. An electronic corpus of construction documents was collected, mainly by using optical character recognition (OCR). The Icon corpus analysis programs have been used with this corpus for the development of the lexicon and grammars.

The following paragraphs briefly describe the various corpus analysis programs. Most of the programs described produce, in addition to the specific items indicated, at least the frequency counts and the corpus positions of these items, and usually other information as well. Also, many of the programs have an option to mark up their output in SGML (Standard Generalised Markup Language) (for example, van Herwijnen 1994).

The *tokeniser* segments the corpus text into tokens, such as lexical, punctuation or number. The string processing example is adapted from this program. The next program, coined the *wordifier*, converts all lexical tokens to lower case (normalisation), and does other processing based on case (for example, extraction of candidate proper names and abbreviations). The high level data structures example is adapted from this. The *lemmatiser* performs inflectional morphological analysis, for nouns only, producing the root form (lemma).

The *invert* and *collocation* programs produce n-grams (all sequences of n-adjacent lemmas; for example bigrams, where n equals 2). The *concordancer* produces key words in context (kwic), concordances of tokens, words, lemmas or n-grams. A further program produces concordances of arbitrary units output by the parser (the parser is not written in Icon).

The *statistics program* produces statistics for n-grams, to assist in identifying compound technical terms. Technical term identification is important for machine translation, as for example in the *termight* system (Dagan and Church 1994).

The *split* and *lexicon* programs create files of lexical entries in the format required by the parser.

Icon programs have been developed for at least two further uses within the project. One program filters parser output, removing ambiguous analyses for a specific set of grammar rules. Another is a preliminary program for recognition of tables and identification of the component cells, as part of work on interpreting tables within texts (Douglas, Hurst and Quinn 1995).

Conclusions

Icon, in addition to its other interesting features, is especially suitable for tasks involving string and text processing. It has been successfully used for this purpose in the CISAU project, where a set of corpus analysis programs have been developed.

Acknowledgements

I thank George Imlah, of BICC, for recognising the potential of Icon, and introducing it into the CISAU project. I acknowledge the contributions of Stephen McCarron, of BICC, and of George Imlah, to the implementation of some of the Icon corpus analysis programs, and the contributions of Shona Douglas and Masja Kempen, of the Human Communication Research Centre, University of Edinburgh, to the development of the ideas behind some of these programs. I further acknowledge the support for the CISAU project (IED4/1/5818) of the Department of Trade and Industry, the Engineering and Physical Sciences Research Council, and the BICC Group.

References

Clocksin, William and Chris Mellish (1994) *Programming in Prolog,* 4th edition, Springer-Verlag: Berlin, Germany

Dagan, I. and Church, K. (1994) '*Termight*: Identifying and Translating Technical Terminology' in *Proceedings of the Fourth Conference on Applied Natural Language Processing*, held in Stuttgart, Germany: 34–40

Douglas, S., Hurst, M. and Imlah, G. (1995) 'Construction Industry Specification Analysis and Understanding' in *Proceedings of Language Engineering* '*95*, held in Montpellier, France, June 1995: 291–300

Douglas, S., Hurst, M. and Quinn, D. (1995) 'Using Natural Language Processing for Identifying and Interpreting Tables in Plain Text' in *Proceedings of the Fourth Symposium on Document Analysis and Information Retrieval*, held in Las Vegas, Nevada, 1995

Griswold, R. (1994) 'The Icon Programming Language' in *BYTE*, May 1994: 193–200

Griswold, R. (1995) *An Overview of the Icon Programming Language,f Version 9*. The University of Arizona Icon Project Document IPD266

Griswold, R. and Griswold, M. (1993) 'History of the Icon Programming Language' in *ACM SIGPLAN Notices*, No. 3, Vol. 28: 53–68

Griswold, R. and Griswold, M. (1990) *The Icon Programming Language*, 2nd edition, Prentice-Hall: Englewood Cliffs, New Jersey

Griswold, R., Jeffery, C., and Townsend, G. (1995) *Version 9.1 of the Icon Programming Language*, The University of Arizona Icon Project Document IPD267

Griswold, R., Townsend, G., Hathaway, C., Jeffery, C. and Alexander, B. (1996) *Frequently Asked Questions about the Icon Programming Language*

Herwijnen, E. van (1994) *Practical SGML*, 2nd edition, Kluwer Academic Publishers: Boston, Massachusetts

Quinn, D. (1996) 'Content Analysis for Error Checking' in *Proceedings of the Language Engineering for Document Analysis and Recognition AISB Workshop*, held at the University of Sussex, England, April 1996

Winston, P. and Horn, B. (1989) *LISP*, 3rd edition, Addison-Wesley: Reading, Massachusetts

The NLTSG's Web-Site

by

Roger Harris

The British Computer Society maintains at its Swindon headquarters a computer which operates as an on-line information server.

The Natural Language Translation Specialist Group has, along with all the other BCS specialist groups, been allocated some space on the server. The space is accessed as a web-site and the full NLTSG site URL or address is:

http://www.bcs.org.uk/siggroup/sg37.htm.

If you have a computer, even a dusty 1980s veteran which may be connected to the Internet, then you can read and download information about the NLTSG and about all the other specialist groups, too.

The information is held in several independent files. The files are connected by HTML links and a large file may have HTML links to parts of itself, thus providing inter-file and intra-file connections.

HTML stands for Hyper-Text Markup Language. It is a system of embedded codes which form part of a file and which provide links to other files. Imagine the following sentence on your computer screen: 'The library contains dictionaries, English corpora, Dutch corpora and grammars.' The words in this sentence which are links to other files will appear in a contrasting colour, but here I shall show them in emboldened capital letters: 'The **LIBRARY** contains **DICTIONARIES**, **ENGLISH CORPORA**, **DUTCH CORPORA** and **GRAMMARS**.'

The HTML links do not appear on the screen but when you click on the word **'LIBRARY'** the server program executes the hidden HTML link which is attached to that word. The server program 'remembers' HTML links so that one may return to earlier screens and menus. HTML links may also be activated by the cursor keys and by the <enter> key. A mouse is not essential.

When you click on, say, the word **LIBRARY**, its hidden HTML link is executed. The HTML link is likely to be a filename and is an implicit instruction to fetch and do something with that file.

The file might be on the BCS computer or on another Internet-linked computer anywhere in the world. The file might contain text giving a description of the library, a detailed map of the floor plan of the library, or further links to library catalogues. If the library specialised in music, then a few bars of Mozart or Scott Joplin could be transmitted to your computer. Clicking on **GRAMMARS** would lead to files dealing with Dutch and English grammar.

The NLTSG web-site contains details of the group plus a quantity of linguistics information. The files are arranged in an hierarchical structure.

Some of the linguistics information which I have included was compiled by others. I have tried to include source and attribution details in such cases. Examples include the list of

newspaper corpora compiled by Isabel Trancoso of ELSNET's Resources Task Group and the list of 'Translators' Periodicals' compiled by Professor Daniel Gile of Université Lumière, Lyon, France.

The structure of the NLTSG's web-site is as follows:

http://www.bcs.org.uk/siggroup/sg37.htm
___NLTSG home page:
  ___The Committee
  ___Meetings
  ___*Machine Translation Review*
  \ ___No.1 – April 1995
  \ ___No.2 – October 1995
  \ ___No.3 – April 1996
  \
  ___Machine translation resources:
    ___A–Z of linguistic and MT items
    ___Books about MT
    ___E-mail linguistic lists
    ___MT resources on the Internet
    ___Newspaper corpora
    ___Suppliers of MT systems
    ___Translators
    ___Usenet newsgroups

When you log onto the BCS home page (http://www.bcs.org.uk/) you will be able to select the full list of specialist groups; from this list you can also select the Natural Language Translation group.

Alternatively, you can go straight to the NLTSG home page by logging onto http://www.bcs.org.uk/siggroup/sg37.htm.

The 'Committee' page contains the names of the various committee members together with their telephone numbers and e-mail addresses.

At present the meetings page contains only an invitation to the lecture on 21 March 1996 by Derek Lewis, but I plan to include abbreviated details of past meetings.

The *Machine Translation Review*, the NLTSG's twice-yearly publication, is not reproduced in full. Instead, the contents page of each issue is reproduced and, where appropriate, the entries are expanded with additional material. Inclusion of some details of the *NLTSG Newsletter*, which preceded *Machine Translation Review*, are being planned. An 'A-Z of linguistic and MT items' contains miscellaneous items for which there were no available categories.

'Books' contains details of books which were mentioned or reviewed in the *Machine Translation Review,* plus other books and texts in electronic format.

'E-mail linguistic lists' contains subscription details for a number of e-mail lists such as SALT, COLIBRI and LANTRA. Subscribing to such lists may be done easily and instructions are given. When your application to subscribe is accepted you will be sent an e-mail of details about how to unsubscribe. Keep this material for future reference.

'MT resources on the Internet' was incomplete at the time of writing, but it should contain much of the material published in *Machine Translation Review* in 1995, plus some additional items.

'Newspaper corpora' contains listings of on-line newspapers world-wide.

'Suppliers of MT systems' contains short descriptions of various MT systems in an address-book format.

The section for 'Translators' contains items for and about human translators. It includes details of the Aquarius Directory of Translators, the On-Line Career Centre, and a listing of periodicals aimed at translators. More items are planned.

'Usenet newsgroups' contains details of those newsgroups which should be of interest to translators and machine translation specialists. Details of how to obtain the relevant FAQ's (Frequently Asked Questions, and Answers) are included.

We are grateful to the BCS for making the web space available.

Roger Harris may be contacted at <rwsh@dircon.co.uk>

Book Reviews

Donald E. Walker, Antonio Zampolli and Nicoletta Calzolari (eds.) (1995) *Automating the Lexicon. Research and Practice in a Multilingual Environment*, Oxford University Press. Hardback. £45. xi + 413 pages. ISBN 0-19-823950-5.

Let me speak plain: I bear a grudge against this book. I pursued references to 'Walker et al., forthcoming' or 'Zampolli et al., in press' or 'Grosseto Proceedings' with various publishers and titles in 1989, in 1990, and in 1991; by 1992 I had given up. It gives me some pleasure, therefore, to be able to express my opinions on its publication in 1995.

The book is essentially the proceedings of a workshop which took place in 1986 at Marina di Grosseto. Some of the papers have been updated since then, but all except the introduction are largely as they were at the workshop. The introduction, which aims to bring the book up-to-date, appears to have been written in 1992. No explanation is offered for why the book took nine years to progress from a set of papers to publication. One is, however, offered for why the publication proceeded at all, so late in the day; and perhaps, whatever we may think of the absurdity of the delay, this decision does stand up to examination. The editors argue that the workshop 'marked a turning-point in the field and can therefore be considered as a point of departure for the major events taking place since then [...]. Although this book has "historical" relevance, it addresses many issues that are still being debated today and that guide research and development efforts' (p. 1).

In addition to the introduction, the book has eleven chapters, each addressing a distinct angle of lexical information and language processing. These are: Richard Hudson on linguistic foundations; Beth Levin on lexical semantics; Robert Ingria on parsing; Susanna Cumming on text generation; Roy Byrd on office systems; Jonathan Slocum and Martha Morgan on translation; Judy Kegl on educational uses; Michael Lesk on information retrieval; Bran Boguraev and Nicoletta Calzolari on what you can find in machine-readable dictionaries (hereafter MRDs); and finally a resource survey by Susan Armstrong-Warwick. The editors' introduction covers the goals of the original workshop, the recommendations which emerged from it, and a review of progress on these recommendations.

There are at least three ways to approach the review: (1) was what was being talked about in Grosseto novel and exciting then? (2) do the papers stand up, as statements of some stature and long-term validity about the field, so might they be used as introductory reading for newcomers to the field and as authoritative points of reference? and (3) have they played a major role in the history of computational lexicons since?

There is every reason to believe that many of the themes were novel and exciting then, but that alone is not a reason for reading the papers now, except possibly to the historian, for which see (3). In relation to (2), most of the papers discuss at length the shortcomings of the resources available as at 1986, so were not designed to be of interest to anyone either not using them or not thinking of using these or similar resources. Ten years on we have better resources and (often) different problems, so few of the papers are serious candidates for longevity. (3) also has its peculiarities: the Introduction is itself an historical overview

according to which the Grosseto workshop was an event of singular importance (thereby justifying the publication of the book). The remainder of the review addresses 'stature and longevity' where relevant, and 'historical importance' across the board.

Hudson starts by noting the possibility that linguists, like other citizens, have had their ideas about lexical information coloured by the dictionaries they have had around the home since childhood. He goes on to question some tenets of 'the mainstream' regarding boundaries between lexicon and grammar, between one lexical entry and another, and between linguistic and other kinds of knowledge. His call to arms has been answered by two, disjoint, areas of work. One is in 'cognitive linguistics', a largely non-formal approach to linguistics in which the cognitive structures underlying linguistic expression are seen as the chief object of study. Cognitive linguistics has made a significant impact on lexicographers, but has had little impact on those seeking to manipulate language by machine. The other is in the 'lexicalism' of normalisms such as DATR, HPSG and LTAG, where the boundary between grammar and lexicon does indeed disappear (and boundaries between lexical items related by, for example, derivational morphology is currently a hot topic). Hudson's own proposal, Word Grammar, now appears as something of a junior member of this club, as it has not been widely used in the NLP community.

Levin's chapter stands up well. It is one of a number of accounts of her work on the behaviour of English verbs: the arguments they take, the alternations they exhibit, and the implications for how verbs might be classified. Her 1993 book, which catalogues these and related facts for several thousand verbs, is the most impressive outcome of this enterprise and has been adopted as source material for a number of projects.

Ingria, Cumming and Armstrong-Warwick all include now outdated surveys in their chapters. All three note the wild differences between what constitutes a 'lexical entry' in different projects. Ingria's description of the difficulty of merging information from different dictionaries, even where they were designed to be mergeable, is highly salient to what is a live issue today. There are several projects listed in the Introduction which have, as their goal, a standard, theory-neutral format for a lexical entry. In 1996 there are still many problems. The authors also comment on the issues relating to multi-word items (including collocations, idioms, phrasal verbs) and to full-form versus base entries. Again, these are current topics.

Boguraev and Calzolari both discuss the sorts of information in a dictionary, and how the software for handling lexical information should be designed to optimise its availability. These chapters clearly state the problems and potential of developing lexical databases or knowledge bases since, and are also highly relevant to the 'standards' enterprise. Calzolari is rather more optimistic than Boguraev, who already sounds sceptical about the fitness of information in MRDs for computer use in 1986.

Kegl's chapter provides the best jokes of the book, where she takes the view of the bewildered anthropologist confronted with the practice — common, apparently, throughout US schools — of giving a child an unfamiliar word and a dictionary and telling them to make up a sentence with the word in it, so with 'chaste' as the target word, we get 'the amoeba is a chaste animal'.

The copyright issue is mentioned by several authors as a problem encountered in working with MRDs. I suspect they all hoped and expected that the issues would be resolved before ten years passed. In fact, unresolved questions of intellectual property and licence negotiations haunt the use of MRDs as much now as ever.

There are remarkably few mentions of sizes of lexicons. To a lexicographer, it is self-evident that a 1,000-headword dictionary does not bear direct comparison with a 50,000-headword one, but many of the discussions, while talking about various lexica, give no indication of size. Both Kegl and Boguraev cite a 1985 workshop in Manchester (transcripts published as Whitelock et al. 1987) in which it turned out that the average size of lexicons that a high-powered gathering of computational linguists were working with was (once one large lexicon was excluded from the sums) 'about 25'. This appalling truth made quite an impact. Judging by the frequency with which the fact has been cited in the literature, the Manchester workshop deserves a share of the historical role that the Introduction assigns to Grosetto.

October 1994 witnessed a workshop in a direct line of descent from the Grosetto one. Entitled 'The Future of the Dictionary', it was held near Grenoble under the auspices of Xerox and the EU project ACQUILEX.

The phrase 'direct line of descent' is appropriate because the EU (or EC as it then was) sponsored the Grosetto workshop, which proceeded to show how much more needed doing in relation to lexical resources; ACQUILEX was one major project which set out to do that work, with some of the key people at Grosetto being in ACQUILEX and at Grenoble. There had certainly been a huge quantity of research in the intervening years, and far more was known about MRDs and their potential for NLP. But there was also a sense that some seams of enquiry had been exhausted, as encapsulated in the title of a paper by Jean Veronis and Nancy Ide, 'Extracting knowledge bases from machine-readable dictionaries : Have we wasted our time?'

One development not discussed in the book, but which is, arguably, taking over from MRD research, is corpus work. Boguraev, whose book chapter explores the potential of MRDs for producing computational lexicons, was at Grenoble; he describes his work on deriving lexicons from corpora. Also, within lexicography, the developments envisaged in the book have been overshadowed by the advent of language corpora, which are transforming the modus operandum of large lexicographic projects.

Several of the book chapters compare what people want from dictionaries with what programs want, and envisage new developments in user-friendly computers for human use. (In one of the quainter moments of the book, Byrd mentions their use in electronic typewriters.) A talk by Sue Atkins at the Grenoble Workshop showed up the fact that, despite dictionary publishers now selling large numbers of dictionaries on CD-ROM, there are not yet any radically new dictionaries for human use.

Have we followed the route charted for us at Grosetto?

For most of the years between 1986 and 1995, the answer is essentially 'yes'. The workshop was undoubtedly instrumental in spreading the word about the potential of MRDs for NLP, and was particularly influential in relation to the EU. Without Grosetto, the major EU drive towards lexical resource development may well not have taken place, and collaborations between system builders, linguists and lexicographers may well have been fewer and later. But Grosetto did not play a major role in the development of 'lexicalism' in NLP and computational linguistics. Two 1987 publications which played a major role in that history — the special issue of *Computational Linguistics* on the lexicon, and Pollard and Sag's HPSG textbook — must already have been well into the planning stages when Grosetto took place. Another 1987 event — the publication of the COBUILD dictionary, with its ideology of corpus lexicography — looms far larger in the annals of lexicography.

Grosetto was very important for developments in Europe, particularly for the years circa 1987 to 1992 and particularly in relation to the exploration of MRDs. Boguraev's chapter considers a 'perhaps somewhat frivolous' distinction between problem-driven and interest-driven work. As often happens, much of the most engaging work has been interest-driven, with the researcher simply exploring what is possible, rather than pursuing a goal which will serve some other project's purposes. Now, we know roughly what is possible (but please note that I do here wish to note one important exception to the general trend: the boundaries continue to be pushed back at Microsoft; see, for example, Dolan 1994).

MRDs have an important and useful place, but they are far from a panacea for all our needs for lexical information. Getting information out of them is rarely straightforward; there are errors and inconsistencies, and a potential user of an MRD should give serious thought to the person-hours involved before assuming that MRD re-use is a sensible way ahead. Without Grosetto, we might not yet be in a position to state these conclusions.

And finally...

As a token, no doubt, of the grail-like nature of this book, a bibliographical feint is still in evidence. The dust cover announces the publisher as Clarendon. Once stripped of its cover, no evidence of Clarendon is to be found.

References

Dolan W. (1994) 'Word sense ambiguation: clustering related senses' in COLING 1994

Levin B. (1993) *English Verb Classes and Alternations*, University of Chicago Press

Pollard, C. and Sag, I. (1987) *An Information-Based Approach to Syntax and Semantics,* Volume 1: *Fundamentals*, Chicago University Press

Whitelock, P. et al. (eds) (1987) *Linguistic Theory and Computer Applications,* Academic Press: London

Adam Kilgarriff

P. Whitelock, M.M. Wood, H.L. Somers, R. Johnson and P. Bennett (eds.) (1987; 2nd printing 1990), *Linguistic Theory and Computer Applications*, London: Academic Press. Softback. £25. 330 pages. ISBN 0-12-747220-7.

All of the material in this book is over ten years old, so the question 'Is the book worth reading?' may arise in the mind of the industrious computational linguist. The 1985 workshop from which this book originated set out to deal with 'fundamental issues in the relation of linguistics to computation and their intersection.' (preface); and it deals with these well. The book divides into two sections: position papers (by Shieber, Gazdar, Marcus, Landsbergen and Kaplan) and discussion sessions (by Pulman, Ritchie, Johnson and Sparck Jones). For the student of computational linguistics, several sections of the book constitute worthy tutorials on particular sub-topics, delivered in clear conversational English, often by a leading researcher in that area. For example, Landsbergen's paper contains a very clear introduction to Montague grammar; Gazdar's presentation of default inheritance in GPSG is crisp; Marcus offers a wonderfully vivid description of description theory and deterministic parsing.

The back page touts a unique feature of the book: the presentations themselves and subsequent open-forum discussions were recorded; the entire book is the distilled transcript of the workshop (with the exception of a discussion session on 'Implementation', by Henry Thompson, which unfortunately seems to have fallen victim to a technical glitch). This spoken-word format should be aesthetically satisfying to those with performance-based research interests and stands as an evocative record of the workshop, but more importantly it gives the reader an opportunity to eavesdrop on the many points of clarification, challenges and thematic connections to wider issues which all serve to deepen our understanding of the papers.

The range of topics covered by the speakers is broad, ranging from the philosophical and theoretical-linguistic to implementation-specific issues. However, the loss of Thompson's presentation has presumably unbalanced the coverage which was achieved at the workshop. This may not be too serious a flaw, because the more theoretically oriented sections have stood the test of time better than particular implementations. We are reminded that the computational linguistic community has long been aware of the central problems: ambiguity (lexical and structural); the structure of the lexicon; tractability; and the relation of linguistic to extra-linguistic or real-world knowledge. For example, Sparck Jones devotes most of her presentation to analysing the language-world relationship, examining how well 'logical form' might hold as an interface between the two. Almost a decade later, in her Presidential Address to the Association for Computational Linguistics, we find her re-iterating the centrality of the language-world relation.

In that same Presidential Address, Sparck Jones offers a potted history of computational linguistics; this gives us enough of a perspective to situate the book. It belongs to a period in NLP to which Sparck Jones refers as the 'grammatico-logical' period; and indeed, much of the work involves discussions of the differences between certain grammar formalisms, the significance of these differences, the types of logical form (intensional logic, second order predicate calculus) and how they connect to syntax and semantics. Even though Sparck Jones' history of NLP dates it to the late 1950s, the discussions in the book indicate a relatively recent identification of computational linguistics as a distinct discipline. On many occasions, discussants contrast the differences between linguistics and computational linguistics. Schieber suggests that they have different (and perhaps partially incompatible)

goals: the former is aimed at universal language properties, described simply, generatively and declaratively.

It is interesting to examine what the presenters have to say concerning the expressed purpose of the workshop, namely examining linguistics and computation. Shieber ('Separating Linguistic Analyses from Linguistic Theories') is interested in the significance of the differences between certain grammar formalisms; he begins his analysis with an analogy from computer science. We can ask: how dependent are particular algorithms on particular programming languages? The simple answer is: not at all, if we accept that programming languages are Turing equivalent. The analogous linguistic question interests him: are particular linguistic analyses dependent on the formalism within which they are embedded? To help answer this question, Shieber tries making certain reductions between some formalisms. He concludes that the 'differences among the various formalisms are considerably less than is commonly thought' (p. 21). Gazdar examines the AI treatment of default inheritance and shows how it solves some problems.

Kaplan's paper deals squarely with issues on the boundaries of linguistics and computation: he enumerates three main kinds of mistake which people can make when thinking about computational linguistics or when building computational linguistic systems. The first is the procedural seduction; as computational linguists started using procedural languages over declarative ones, the temptation grew to include arbitrary structures (indexes, caches, registers, etc.) to which the researchers were not psycholinguistically committed. If it seemed like a seduction to Kaplan in the mid-80s, perhaps this phenomenon should now be openly embraced by language engineers: if a procedural hack works in building good-quality language engineering products, then this in itself is a justification. Next, he describes the substance seduction: just as the previous seduction warned about being undisciplined computationally, this one warns about being too promiscuous with the sorts of constraint which linguistics can impose. Kaplan's 'Linguistic Theory and Computer Applications' captures the best of computational linguistics in the mid-80s; much of the theoretical material remains relevant today, though the centre of gravity of the subject has shifted slightly towards language engineering and corpus-based research.

J.G. McMahon

Conferences and Workshops

The following is a list of recent or forthcoming conferences and workshops. Telephone numbers and e-mail addresses are given where known.

14–16 March 1996
Georgetown University Round Table on Languages and Linguistics 1996
Tel: +1 202/687 5726, fax: +1 202/687 0699, e-mail: gurt@guvax.georgetown.edu

2 April 1996
Language Engineering for Document Analysis and Recognition
University of Sussex, UK
E-mail: aisb@cogs.sussex.ac.uk,
http://www.cogs.susx.ac.uk/aisb/aisb96

11–12 April 1996
SALT workshop on Empirical and Theoretical Methods in Text and Speech Processing
University of Manchester, UK
E-mail: mary@cs.man.ac.uk

15–17 April 1996
SDAIR '96: Fifth Annual Symposium on Document Analysis and Information Retrieval
Alexis Park Resort, Las Vegas, Nevada, USA.

22–26 April 1996
PAP & PACT: PAP '96: The 4th International Conference on the Practical Application of PROLOG, London, UK.
Tel: +44 1253 358081, fax: +44 1253 353811, e-mail: info@pap.com
http://www.demon.co.uk/ar/PAP96/index.html

9–11 May 1996
International Translation Studies Conference
Dublin, Republic of Ireland.
Fax: +353 1 7045527

17–18 May 1996
Empirical Methods in Natural Language Processing
Pennsylvania State University, USA
Kathy Wohlschlaeger
Tel: +1 215 898 6564, e-mail: kathyw@eniac.seas.upenn.edu

4–6 June 1996
International Conference on Natural Language Processing and Industrial Applications
Moncton, New-Brunswick, Canada

4–7 June 1996
ICCC ’96: International Conference on Chinese Computing ’96
Institute of Systems Science, National University of Singapore, Singapore 0511

13–15 June 1996
INLG ’96: 8th International Workshop on Natural Language Generation
Herstmonceux, Sussex, UK
Tel: +44 1273 642900, e-mail: inlg96@itri.brighton.ac.uk

25–29 June 1996
ALLC-ACH ’96: Association for Literary and Linguistic Computing, Association for Computers and the Humanities
University of Bergen, Norway
Tel: +47 55 21 28 65, fax: +47 55 32 26 56, e-mail: espen.ore@hd.uib.no
http://www.hd.uib.no/allc-ach96.html

28 June 1996
SIGPARSE ’96: International Workshop on Punctuation in Computational Linguistics
Edinburgh,UK EH8 9LW
bernie@cogsci.ed.ac.uk

1–3 July 1996
DRH ’96: Digital Resources for the Humanities
Somerville College, Oxford, UK
Tel: +44 1865 288169, fax: +44 1865 288163.
http://info.ox.ac.uk/~drh96/

14–26 July 1996
CETH’s 5th Annual Summer Seminar on Methods and Tools for Electronic Texts in the Humanities, Center for Electronic Texts in the Humanities, 169 College Avenue, New Brunswick, New Jersey, USA 08903
Tel: +1 (908) 932 1384, fax: +1 (908) 932 1386, e-mail: pac@rci.rutgers.edu

17–18 July 1996
DAARC ’96: Discourse Anaphora and Anaphor Resolution Colloquium
University of Lancaster, UK
Tel: +46 8 162335, fax: +46 8 155389, e--mail: kicki.hellman@lingvistik.su.se

21–24 July 1996
2nd Conference on Information-Theoretic Approaches to Logic, Language and Computation
Regent’s College, London, UK.
e-mail: linguist@tamvm1.tamu.edu, penguists@linc.cis.upenn.edu lsm

1–3 August 1996
UAI ’96: 12th Annual Conference on Uncertainty in Artificial Intelligence
Reed College, Portland, Oregon, USA
Tel: +1 (206) 543 4784, fax: +1 (206) 543 2969, e-mail: hanks@cs.washington.edu
http://cuai-96.microsoft.com/

4 August 1996
WCLC-4: 4th Workshop on Very Large Corpora
University of Copenhagen, Copenhagen, Denmark
e-mail: WVLC-4@ling.umu.se, e-mail: dagan@bimacs.cs.biu.ac.il
http://www.ling.umu.se/SIGDAT/WVLC-4.html

5–9 August 1996
COLING '96: International Conference on Computational Linguistics
University of Copenhagen, Denmark.

9–12 August 1996
TALC '96: Teaching and Language Corpora
University of Lancaster, UK
e-mail: mcenery@computing.lancaster.ac.uk

11–22 August 1997
European Summer School in Logic, Language and Information
Aix-en-Provence, France
Tel: +255, fax: +42 2 2191 4 309, e-mail:esslli@ufal.mff.cuni.cz

12–13 August 1996
ECAI '96: Workshop on the Representations and Processes between Natural Language and Vision, Budapest, Hungary
http://zaphod.cs.uni-sb.de/~maass/ecai96-ws-v-nl.html

12–16 August 1996
Multilinguality in the Software Industry
Budapest, Hungary
Tel: +301 6510310 (ext. 520), fax: +301 6532175, e-mail: costass@iit.nrcps.ariadne-t.gr
http://www.iit.nrcps.ariadne-t.gr/~costass/mulsaic.html

12–23 August 1996
ESSLI: European Summer School in Logic, Language, and Information, Prague
Tel. +42 2 24510286, fax: +42 2 532742, e-mail: essli@ufal.mff.cuni.cz

12–16 August 1996
12th European Conference on Artificial Intelligence
Budapest, Hungary.
Tel: +49 681 3025267, fax: +49 681 3025341, e-mail: ecai-96-ws@dfki.uni-sb.de

13–18 August 1996
EURALEX '96: 7th Euralex International Congress
University of Gothenburg, Sweden
Tel: +46 317734544, fax: +46 31773 44 55, e-mail: gellerstam@svenska.gu.se

27 August 1996
PRCIAI ’96: 4th Pacific Rim International Conference on Artificial Intelligence: Future Issues for Multilingual Text Processing, Cairns, Australia
D. Estival, Department of Linguistics, Melbourne University, Victoria 3052, Australia
Tel: +61 3 9344 4227, fax:+61 3 9349 4326, e-mail:
Dominique.Estival@linguistics.unimelb.edu.au

16–18 September 1996
NeMLaP-2: International Conference on New Methods in Natural Language Processing
Bilkent UniversityAnkara, Turkey
http://www.cs.bilkent.edu.tr/~nemlap2/

25–27 September 1996
RECITAL ’96: Rencontre des Etudiants-Chercheurs en Informatique pour le Traitement Automatique de la Langue
Tel: +33 1 69858018, fax: +33 1 69858088, e-mail: ferrari@limsi.fr

8–10 January 1997
IWCS II: 2nd International Workshop on Computational Semantics
Tilburg, The Netherlands
Harry C. Bunt, Tilburg University, P.O. Box 90153, 5000 LE Tilburg, Netherlands
Tel: +31 13 466, fax: +31 13 466, e-mail Harry.Bunt@kub.nl
http://itkwww.kub.nl:2080/itk/Docs/