CRITAC— An experimental system for Japanese text proofreading

by Koichi Takeda
Emiko Suzuki
Tetsuro Nishino
Tetsunosuke Fujisaki

This paper describes an experimental expert system for proofreading Japanese text. The system is called CRITAC (CRITiquing using ACcumulated knowledge). It can detect typographical errors, Kana-to-Kanji conversion errors, and stylistic errors in Japanese text. We describe the basic concepts and features of CRITAC, including preprocessing of text, a high-level text model, Prolog-coded heuristic proofreading knowledge, and a user-friendly interface. Although CRITAC has been primarily designed for Japanese text, it appears that most of the concepts and the architecture of CRITAC can be applied to other languages as well.

1. Introduction

Text proofreading on a computer has been one of the most attractive facilities since word processing became an everyday use of computers. Although simple spelling checkers and writing tools [1] for English text are widely available, detection of grammatical and stylistic errors in text is still a research issue [2]. This is because such text proofreading involves a fair amount of natural-language processing. The task becomes even harder for Japanese text since, among other things, words are not separated by blanks, and segmentation of the text into words is not trivial. Therefore, human proofreading remains a time-consuming part of the text-preparation process even though editing, formatting, and distributing the text are fairly well automated.

CRITAC (CRITiquing using ACcumulated knowledge) [3, 4] is our solution to the above problem. It is an expert system for proofreading (and, we hope, critiquing) Japanese text. CRITAC is based on a conceptual representation of text (called *structured text*), a Prolog-coded heuristic knowledge base for proofreading, and text preprocessing to generate a physical representation of the structured text. Structured text allows us to express high-level proofreading knowledge without worrying about how text is physically stored. In particular, CRITAC exhibits an incremental growth of proofreading capability through the accumulation of heuristic knowledge. This advantage is further discussed in Section 2. The concepts of structured text and the proofreading knowledge base are explained by using English-like text in Section 3.

Implementation of CRITAC and its sample proofreading session are shown in Sections 4 and 5, respectively. (Since these issues are language-dependent, we provide some information about the Japanese writing system in the Appendix to help the reader understand the issues.) A summary and an outline of future work are given in Section 6.

2. Knowledge-based approach to text proofreading

There are many types of errors we can find in text. They might be classified into spelling, grammatical, and stylistic

TEXT T: "I have a pen. I found a notebook in the room."

GRAMMAR G: {S → NP VP PERIOD, S → NP VP PP PERIOD,
NP → DET NOUN, NP → PRONOUN,
VP → VERB, VP → VERB NP, PP → PREP NP,
DET → "the", DET → "a", NOUN → "pen",
NOUN → "notebook", NOUN → "room",
PRONOUN → "I", VERB → "have",
VERB → "found", PREP → "in", PERIOD → "."}

STRUCTURED TEXT

```
text(t1,{sec1}).                    word(w1,"I").
sect(sec1,{par1}).                  word(w2,"have").
para(par1,{s1,s2}).                 word(w3,"a").
sent(s1,{np1,vp1,w5}).              word(w4,"pen").
sent(s2,{np3,vp2,pp1,w13}).         word(w5,".").
phrase(np1,{w1}).                   word(w6,"I").
phrase(vp1,{w2,np2}).               word(w7,"found").
phrase(np2,{w3,w4}).                word(w8,"a").
phrase(np3,{w6}).                   word(w9,"notebook").
phrase(vp2,{w7,np4}).               word(w10,"in").
phrase(np4,{w8,w9}).                word(w11,"the").
phrase(pp1,{w10,np5}).              word(w12,"room").
phrase(np5,{w11,w12}).              word(w13,".").
```

Figure 1

Simple structured text.

errors, or syntactic and semantic errors, etc. It is important, however, to consider the range of complexity required for a computer to detect such errors (or problems) as the following:

- An abstract exceeds the preset limit of 300 words.
- A proper noun is found that does not begin with a capital letter.
- The incorrect expression *many money* occurs.
- The conclusion of the paper is too vague.

Some of these errors are detected easily, whereas others are so difficult to find that powerful syntactic and semantic analysis is required. Although text critiquing using a natural-language parser [5] and possibly its extension to semantic processing as in [6, 7] are promising methods to explore, we need a more integrated framework to cover each of the problems mentioned earlier.

Our requirement can be satisfied by a knowledge-based system approach [8] which has these properties:

- Fragments of proofreading knowledge can be described.
- Each piece of knowledge can be accumulated incrementally in a knowledge base.
- Proofreading knowledge can be modularized according to the types of errors to be detected by the knowledge.
- Procedural modules such as a syntactic parser and semantic analyzer can be plugged into the system to perform specific proofreading.

Thus, we can build up an expert system by collecting and formulating human proofreading knowledge. Our approach is implemented to satisfy the following requirements:

- *Representation of text* Text is a central object of the system. Proofreading knowledge is applied to the text, and a user should be able to modify the text to correct errors.
- *Representation of proofreading knowledge* Proofreading knowledge can be formed by a set of *rules*. Each rule describes a specific type of error declaratively even though it might actually invoke a procedure (e.g., a parser) to verify the rule.
- *User interface* In some cases, the system does not have the competence of a human expert. It may fail to correct/detect some errors, or may even make a mistake. Thus, the interface is crucial in helping a user confirm each system-detected error. The user interface should also be designed to support user-initiated proofreading, which can be intricate and *ad hoc*.

We discuss these components in the next section.

3. Structured text and proofreading knowledge base

It has often been posited that text has an inherent hierarchy. A simple version of the hierarchy is as follows:

```
text
  section
    paragraph
      sentence
        phrase
          word.
```

Recent text-processing systems can handle each element in the hierarchy to support advanced editing and formatting of text, which has motivated logical text modeling [9].

Proofreading also requires a logical text model, but to do a satisfactory job, grammatical knowledge and domain knowledge are also needed for handling textual content. Since domain knowledge is beyond the scope of this paper, we concentrate on how to incorporate logical structure and syntactic knowledge.

Grammatical knowledge in CRITAC comprises a *lexicon*, a set of *morphological rules*, and a *grammar*. The lexicon consists of a set of *words* associated with attributes such as

root form, *parts-of-speech*, and *pronunciation*. By morphological rules we mean a mapping between entries in the lexicon and specific forms of words occurring in text. A syntactic grammar is a mapping between sentences and syntactic structures.

• *Structured text*

Structured text is our conceptual view of text. It combines a logical structure of text and syntactic knowledge, as shown in **Figure 1**. Throughout the paper, we use Prolog [10] notation for structured text and proofreading knowledge. In Figure 1, we have structured text for text *T* represented by six types of *facts*—text(), sect(), para(), sent(), phrase(), and word(). The *facts* are defined in the section immediately following. It should be noted that variables begin with uppercase letters and constants either begin with lowercase letters or are quoted strings.

text(T,SectIds) Text *T* consists of a sequence of sections identified by *SectIds*. T is an identifier of text, and SectIds is a list of identifiers of sections.

sect(Sec,ParaIds) A section *Sec* consists of a sequence of paragraphs identified by *ParaIds*. Sec is an identifier of a section, and ParaIds is a list of identifiers of paragraphs.

para(Par,SentIds) A paragraph *Par* consists of a sequence of sentences identified by *SentIds*. Par is an identifier of a paragraph, and SentIds is a list of identifiers of sentences.

sent(S,Ids) A sentence *S* consists of a sequence of phrases and words identified by *Ids*. S is an identifier of a sentence, and Ids is a list of identifiers of phrases and words.

phrase(P,Ids) A phrase *P* consists of a sequence of phrases and words identified by *Ids*. P is an identifier of a phrase, and Ids is a list of identifiers of phrases and words.

word(W,String) W is an identifier of a word. String is a spelling of the word identified by W.

Although we represented "string" only at the word level, it is easy to produce the definition text(T,SectIds,String) instead of text(T,SectIds), thereby taking a different conceptual view. Other facts can also be modified similarly. One alternative way is to define an auxiliary set of facts *string(Id,String)*, which is *true* if *String* is a character string for an object *Id*. We can think of this fact as a function to compute a character string of a given object *Id*.

The hierarchy among these six types of objects is represented by logical links using identifiers (see **Figure 2**). An identifier is a *surrogate* in the sense of [11] of each object. Note that a syntax tree of a sentence defined by a grammar is embedded in logical links of sent() and phrase(). It is also important to note that we can associate a word in word() with as many attributes as we have in our lexicon.

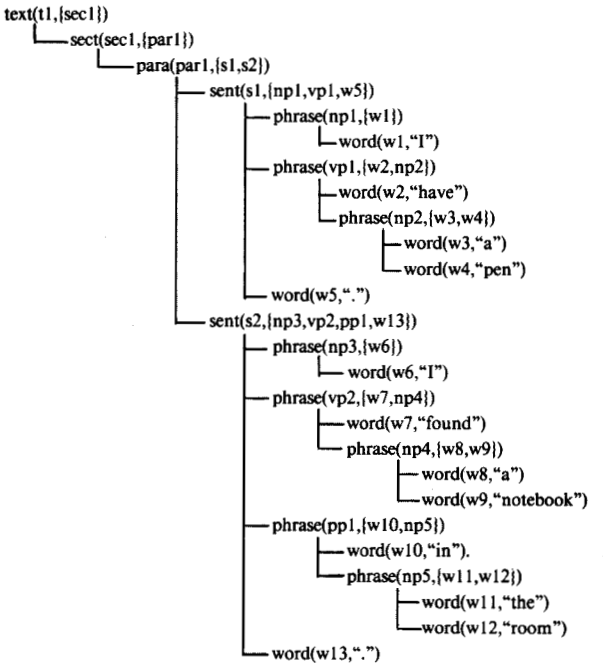


Figure 2

Logical links among objects in structured text.

For example, if we have *parts-of-speech* and *pronunciation* for each word in the lexicon, we will have

word(W,String,Parts-of-speech,Pronunciation)

instead of word(W,String). It follows that we can successfully hide procedural aspects of text processing from structured text. A lexical analyzer, a parser, and dictionary lookup are no longer necessary for structured text because all the information obtained from these procedures is already represented in the structured text. These procedures are either called prior to the generation of physical text representation or invoked dynamically to formulate the structured text. The former case is called *preprocessing* of text; the latter is known as *on-the-fly processing*.

It is apparent from our definition of structured text that the entire text can be managed by a relational database [12] system by associating each type of fact with one relation. This is a great advantage, because we can automatically enjoy such benefits of relational database systems as high-level query languages, view mechanisms, security protection, and concurrency control [13]. It is also possible to

A KWIC view is used to arrange a set of words consecutively.
A set of compound words containing common primitive
words can be arranged consecutively in a KWIC view.

(a)

A	KWIC	view is used to . . .
. . . consecutively in a	KWIC	view.
A	set	of compound words . . .
. . . to arrange a	set	of words consecutively.
. . . in a KWIC	view	.
A KWIC	view	is used to . . .
. . . a set of	words	consecutively.
. . . of compound	words	containing . . .

(b)

Figure 3

(a) Source view and (b) KWIC view.

implement structured text in an extended relational database model [14], semantic data models [15], or an object-oriented programming language [16].

A detailed examination of the types of word attributes that are required for a specific language (in this case, German) may be found in [17]. Attributes of higher types of objects, say the *heading* of a section, can be handled similarly. Choices of attributes and types of objects are language-dependent.

- *External views of text*

Structured text is a conceptual view of text primarily designed for programmers and knowledge engineers of the proofreading system. This view is not appropriate for end users because it is harder for them to browse and edit the structured text than a plain view of text, i.e., a sequence of characters. Therefore, *external views* are designed to provide users with an easy means of browsing and editing text as well as all the information available from structured text. Two kinds of external views are given here. One is called a *source view*, the other is called a *KWIC (Key Word In Context) view* (see **Figure 3**). Both of these views seem to be a plain representation of text, but we can support such operations as

- Movement/deletion of words, noun phrases, relative clauses, or sentences.
- Display of lexical information, synonyms, or antonyms of a word.
- Application of proofreading knowledge.

The KWIC view consists of n lines, where n is the number of *keywords* in the text. In the conventional sense of KWIC, keywords are usually content words (nouns, verbs, etc.) and each line is a sentence where each keyword appears. We extend the notion of a KWIC view such that

- *Keywords* can be any single type of object in the structured text which is specified by some predicate (e.g., proper nouns, noun phrases, or first sentences of paragraphs).
- The *context* of each keyword, which is a logical line of the KWIC view, can be any type of object in the structured text. The type of context must be either superordinate or equal to the type of keywords.
- The *ordering* of KWIC lines is a total ordering of keywords, such as an alphabetical ordering or an ordering by appearance in the text. A *predecessor* or *successor* of each keyword in the context might be used to define an ordering as well.

In accordance with the definitions above, we can get a KWIC view of all the proper nouns with noun phrases as context in an alphabetical ordering.

Updates through external views are immediately reflected by conceptual and physical representation of text. It is possible to switch multiple external views to update single physical texts. Thus, external views provide a very flexible user interface. Discussion of multiple views and text analysis can be found in [18].

- *Proofreading knowledge*

Proofreading knowledge can be described as a set of Prolog predicates of the form

error(Id,Type-of-Error) ← cond_1(Id) & . . . & cond_n(Id),

where *Id* is an identifier of an object in structured text, *Type-of-Error* is an error type to be detected, and cond_1(Id), . . ., cond_n(Id) are predicates which define a specific error.

For example, if we decide to detect text whose abstract contains more than 300 words,

```
error(Text,long-abstract) ← text(Text,SectIds) &
                            member(Sect,SectIds) &
                            isAbstract(Sect) &
                            numberOfWords(Sect,X) &
                            moreThan(X,300).
```

will be the proofreading rule. Here, *member(Sect,SectIds)* is true if *Sect* is in the list *SectIds*; *isAbstract(Sect)* is true if *Sect* is an identifier of an abstract; *numberOfWords(Sect,X)* is true if X is the number of words appearing in the section *Sect*; and *moreThan(X,300)* is true if X is more than 300. These predicates are further defined by other predicates unless they are either facts of structured text or *built-in predicates* [10].

Let us consider ambiguous sentences and invalid sentences. An ambiguous sentence *s1* might be represented by multiple facts

$sent(s1,Ids_1)$, $sent(s1,Ids_2)$, . . ., $sent(s1,Ids_m)$,

where Ids_1,. . .,Ids_m are distinct lists of identifiers. Thus, detection of ambiguous sentences is written as

```
error(Sent,ambiguous) ← sent(Sent,Ids1) &
                        sent(Sent,Ids2) &
                        notEqual(Ids1,Ids2).
```

Since a sentence tends to have multiple syntactic structures except in restricted domains, it might be better to modify this rule so that limited types of ambiguities (e.g., a noun phrase "A and B or C") can be detected. By invalid sentences we mean sentences which cannot be parsed by a given grammar. An invalid sentence has either an unknown word or a subsequence of words which the grammar cannot map to a tree. By adding auxiliary facts, say *unknown(Id)* and *invalidSequence(Sent,Ids)*, we can detect and locate the errors. Efforts to generate reasonable parses for "unorthodox" sentences [19] can be incorporated to detect stylistic problems.

We end this section with an interesting combination of KWIC views and proofreading rules. If we allow "pronunciation" ordering of keywords, we can arrange homonyms such as *hair* and *hare* adjacently in a KWIC view. For example, we can detect successive keywords *hair* and *hare* followed immediately, say, by *in the park*, where either keyword might be incorrect—a situation which could easily occur in text created with a voice-input method. We can also detect lack of conformity in word usage such as *style errors* and *stylistic errors* by an alphabetical ordering of keywords and their successors. Note that the detection of these possible errors can be performed in time proportional to the number of keywords once such a KWIC view is computed, and it is easy for a user to verify each error in the KWIC view because of the consecutive arrangement of keywords.

4. System architecture

CRITAC is our implementation of the concepts described earlier. The architecture of CRITAC, shown in **Figure 4**, consists of four major components: a text compiler, a text editor, an SQL/DS [20] dictionary server, and a proofreading knowledge base written in IBM Prolog [21].

- *Text compiler*

The *text compiler* is a preprocessor of text. It generates

- A physical representation of structured text. This is a collection of facts written in IBM Prolog. Since we use Prolog for physical representation, the conceptual and physical views of text are the same.
- A source view of text and proofreading (error) messages, which are cross-referenced. We call this *batch* proofreading; a user can get the same messages during *interactive* proofreading using the text editor. The same proofreading knowledge base is used in both cases. An example of proofreading messages is given in **Figure 5**.
- Optionally, a KWIC view of the text and some statistics of the sort described in [1]. Currently, only a few types of statistical information are available for Japanese text.

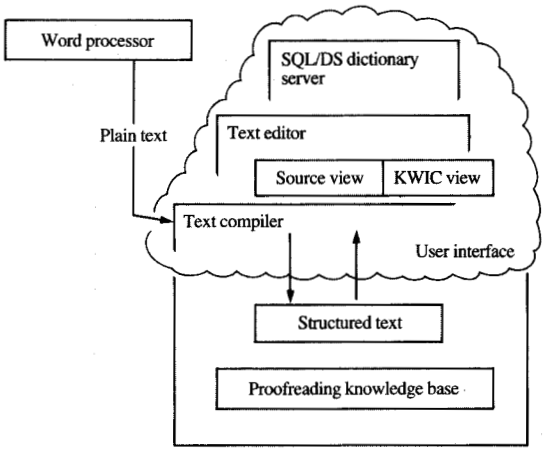


Figure 4

CRITAC architecture.

Table 1 Text compiler versus compilers of programming languages.

	Text compiler	*Compilers of programming languages*
Source	Plain text written in a natural language	A program written in a source programming language
Object	A Prolog program (a collection of facts)	A program written in a target programming language
Other output	External views and error messages	Source listing and error messages

As shown in **Table 1**, the text compiler is analogous to compilers of programming languages.

Figure 6 shows the types of objects that have been designed for Japanese structured text. As discussed in the Appendix, seg() is specific to Japanese. It corresponds to a segment (or, in Japanese, *bunsetsu*), which consists of one

Segment

＊ERROR 31 on line 23 position 6：『機械翻訳システム野ためには』
適当でない漢字が使われています。
誤変換ではありませんか？ Connectivity violation

＊ERROR 33 on line 40 position 35：『実用かと』
未変換の可能性はありませんか？
この「か」は漢字で書かれるはずではありませんか？
Unconverted Hiragana character

＊ERROR 17 on line 24 position 28：『よって』
1つの文の中に同じ言葉が繰返し使われています。
他の言い回しができませんか？ Repetitive use of the same expression

Figure 5

Sample proofreading message.

content word that may be followed by a sequence of function words. In Figure 6, head() corresponds roughly to word() in the previous section. Currently we allow only one text at a time in the system, and text consists simply of paragraphs, not sections. Moreover, a Japanese grammar has not yet been incorporated into the system. Thus, the present version of structured text is a combination of a simple hierarchy of objects and morphological knowledge. An example of structured text is shown in **Figure 7**.

The text processing necessary to generate the above structured text is illustrated in **Figure 8**. A sample Japanese sentence is given whose transliteration and translation appear, respectively, above and below the sentence. The steps involved in the process are as follows:

1. A segmentation algorithm is applied to the sentence. This algorithm contains about 100 heuristic rules, each of which specifies the cases where a segment boundary usually appears. The accuracy of this segmentation algorithm is about 97.5%.
2. Content words in the segments are recognized by looking them up in a primitive-word dictionary. If a content word is a compound word, it is further decomposed into primitive words. Since many compound words (in particular, Kanji compound words) have ambiguities of decomposition, we use a stochastic estimation algorithm [22] to find the most likely decomposition. This is a version of algorithms for Markov models found in [23, 24]. The accuracy of decomposition is about 96.5%.
3. Function words in each segment are identified. The *connectivity* of these function words (see the description in the Appendix) is verified. The connectivity is described by an automaton [25], and a valid sequence of function words corresponds to an acceptable transition in the automaton.
4. The internal structure of each Kanji compound word is analyzed. The internal structure is represented as a binary tree which is described by a probabilistic context-free grammar [26] of compound words. Again, a stochastic estimation algorithm is used to get the most likely parse

tree [27]. One example of a nontrivial parse tree is shown in **Figure 9**, where we have alternative parses such as

((very (large computing)) machinery) or
(((very large) computing) machinery)

in addition to the correct one, i.e.,

((very large)(computing machinery))

of Figure 9.

• *Text editor*

The *text editor* provides source and KWIC views, and facilitates the browsing and modification of text through these views. When a user asks the system to apply proofreading rules, diagnostic messages will appear on the screen, with possible errors underlined in the text. In addition, the types of errors and their explanations can be obtained by hitting a certain key on a keyboard with a cursor locating a particular underlined portion. A mechanism to reflect the updates in external views of structured text is not yet fully supported.

• *SQL/DS on-line dictionary server*

On-line access to system dictionaries or an encyclopedia [28] is a very useful facility in an advanced text-processing system. Such sources can be used to confirm system-detected errors or to help user-initiated proofreading. We have implemented a dictionary server using SQL/DS, a relational database system. It contains spelling, pronunciation, and parts of speech for about 30000 Kanji primitive words. Thus, retrieving attributes of a word, which we discussed in the previous section, is actually interpreted as a relational query to the dictionary server.

Since SQL/DS supports an excellent relational query language, most of the users' requests for lexical information can be expressed as simple relational queries. For example, retrieving *all the homonyms of a word* is mapped to the query

```
SELECT spelling
FROM word_table
WHERE word_table.pronunciation = X
```

SELECT, FROM, and WHERE are reserved words of SQL/DS. "SELECT spelling" asks SQL/DS to return values of *spelling*. "FROM word_table" means that the range of retrieval is a *word_table* relation. "WHERE word_table.pronunciation = X" specifies that we are only interested in those words whose pronunciation is X. When this query is issued, X is replaced with the pronunciation of a given word. Thus, the entire meaning of this query is 'Give me the spelling of all the words in word_table whose pronunciation is X.'

Searching for homonyms or words starting with the same characters appears to be common, and queries of these types are stored in the system beforehand. These queries are called *canned* queries. Canned queries are issued by either hitting an assigned key or giving the name of a query with its parameters. A user does not have to learn the query language to issue a canned query. Additional features of the SQL/DS dictionary server are *ad hoc* queries and the *view* mechanism. A user can access specific data by formulating a query. Once a query is expressed, it will be issued to the dictionary server just like a canned query. A view is a virtual relation whose contents are returned values of a query. For example, we can define a private "noun table" whose definition is the query

```
SELECT spelling, pronunciation
FROM word_table
WHERE word_table.part_of_speech = "noun"
```

The "noun table" has two attributes, spelling and pronunciation, and contains all the nouns from word_table.

• *Proofreading knowledge base*

Our proofreading knowledge base consists of about 40 rules for detecting about 20 types of errors. We describe a few of the rules which detect common errors.

seg(I,J,K,Seg) A character string *Seg* is the *K*th segment in the *J*th sentence of the *I*th paragraph. Here, a triple (I,J,K) is an identifier of Seg. I, J, and K denote the same indexes below.

head(I,J,K,Cont,Pro,Pos,Lab) *Cont* is a content word (possibly a compound word) of the segment identified by I, J, and K. *Pro* and *Pos* are the pronunciation and parts of speech of Cont, respectively. *Lab* is a list of labels to denote prefixes, suffixes, and primitive words appearing in Cont.

tail(I,J,K,Func,Pos) *Func* is a list of function words in the segment identified by I, J, and K. Pos is the part of speech of the last function word of Func.

punc(I,J,K,Punc) *Punc* is either a period or comma (if any) of the segment identified by I, J, and K.

sent(I,J,S) *S* is a sentence which is a sequence of segments seg(I,J,K,Seg) for all K.

para(I,P) *P* is a paragraph which is a sequence of sentences sent(I,J,S) for all J.

text(T) *T* is text which is a sequence of paragraphs para(I,P) for all I.

Figure 6

Types of objects in Japanese structured text.

```
seg(1,1,1,'ワードプロセッサの ').
head(1,1,1,'ワードプロセッサ '.nil,'わあどぷろせっさ '.nil,{13, 18, 19},
    'K'.nil).
tail(1,1,1,'の'.nil,{76}).
seg(1,1,2,'普及に').
head(1,1,2,'普及'.nil,'ふきゅう'.nil,{13, 19},'12'.nil).
tail(1,1,2,'に'.nil,{78}).
seg(1,1,3,'伴い').
head(1,1,3,'伴'.nil,'ともな'.nil,{10},'1'.nil).
tail(1,1,3,'い'.nil,{28}).
punc(1,1,3,'。').
seg(1,2,1,'日本語文書を').
head(1,2,1,'日本語'.'文書'.nil,'にほんご'.'ぶんしょ'.nil,{19},
    '12S'.'12'.nil).
tail(1,2,1,'を'.nil,{77}).
seg(1,2,2,'作成することが').
head(1,2,2,'作成'.nil,'さくせい'.nil,{13, 19},'12'.nil).
tail(1,2,2,'する'.'こと'.'が'.nil,{75}).
seg(1,2,3,'容易になってきた').
head(1,2,3,'容易'.nil,'よういい'.nil,{19},'12'.nil).
tail(1,2,3,'に'.'な'.'っ'.'て'.'き'.'た'.nil,{63}).
punc(1,2,3,'。').
seg(1,3,1,'しかし').
head(1,3,1,'しかし'.nil,'しかし'.nil, nil,'H'.nil).
```

Figure 7

Sample structured text.

Nihongo bunsho o mae shori suruto bunsetsu o eru.

1. 日本語 文書 を 前 処理すると 文節 を得る。
   When we preprocess Japanese text, we obtain segments.

2. 日本語文書を / 前処理すると / 文節を / 得る。

3. 日本・語 / 文書 — 前・処理 — を — する・と — 文節 — を — 得 — る。
   Japan language text — pre process — segment — obtain
   W S W — P W — W — W
   noun — noun/verb — noun — verb

4. (case:OBJ) (verb_conj. particle) (case:OBJ) (verb_conj)

5. 日本 語 文書 — C, C
   W S W

P: Prefix

W: Primitive word (2 characters)

S: Suffix

C: Compound word

Figure 8

Processing a Japanese sentence.

Typographical errors are among the most common. If a user has made a mistake in typing Hiragana characters, a connectivity violation of function words (i.e., an invalid sequence of function words) often results. This case is handled by the rule

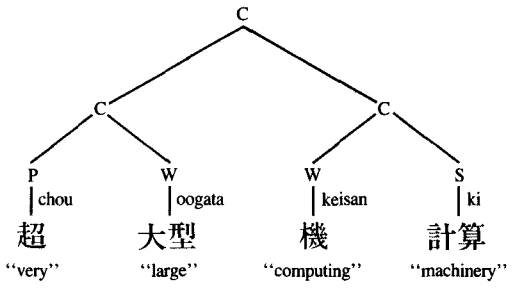
rule(I,J,K,invalidFunctionWord) ← tail(I,J,K,Func,nil),

because the part of speech of the last function word in such a sequence is undefined—that is, nil. Note that if this rule is *unified* with an instance of a tail( ), which means there is a tail( ) whose part of speech is nil, we can locate the error by using the value of I, J, and K. If a user has forgotten to convert a Hiragana string into the corresponding Kanji string, the Hiragana string is usually treated as an invalid sequence of function words also.

If a user has made a mistake in typing Katakana characters or in *converting* some Hiragana characters into an unknown combination of Kanji characters, an unknown content word results. This case is similarly captured by the rule

rule(I,J,K,invalidContentWord)
← head(I,J,K,Cont,Pro,nil,Lab),

because the part of speech of an unknown word is undefined.



P: Prefix
W: Primitive word (2 characters)
S: Suffix
C: Compound word

Figure 9

Parse tree of a Kanji compound word.

Some stylistic problems are expressed by our rules quite simply. We have rules to detect constructions such as

私の母の会社の位置は
My mother's company's location is...
(repetition of possessive noun phrase)

等式AあるいはBおよびCの適用により
Applying equations A or B and C,
(ambiguous modification)

or to detect incorrect ending of sentences.

Detection of a conversion error is harder because conversion gives a valid word but makes an incorrect choice of Kanji words. It is sometimes impossible to detect the error without understanding the context. For example, a user may convert a Hiragana string

kousei suru
こうせい する

into

kousei suru
構成 する
'to construct'

instead of

kousei suru
校正 する
'to proofread'

If a user makes such a mistake once out of several times (that is, there is one instance of 'to construct' and several instances of 'to proofread' in the text), we can detect the mistake by using a homonym detection rule:

```
rule(I1,J1,K1,I2,J2,K2,homonyms)
        ← head(I1,J1,K1,Cont1,Pro,Pos1,Lab1) &
          head(I2,J2,K2,Cont2,Pro,Pos2,Lab2) &
          notEqual(Cont1,Cont2).
```

Strictly speaking, we have to check whether Cont1 and Cont2 are primitive words. This kind of rule is efficiently verified by materializing a KWIC view, as discussed earlier; it is called a *KWIC rule* [3]. Other rules are called *source rules.*

5. Sample proofreading session

In this section, we give an example of interactive proofreading in CRITAC. A typical sequence is the following:

1. Invoke the text editor with the source view of the text. Let the system apply the proofreading rules. If errors are detected, the corresponding part in the source view is underlined (**Figure 10**). Proofreading rules are applied to a part of the structured text which is shown on the screen. The user can scroll the screen to see the rest of the text.
2. Display the explanation for each error (**Figure 11**). Explanations include rewriting suggestions for stylistic problems.
3. Locate an error in a KWIC view to see if the same errors have been made in the text (**Figure 12**). The default KWIC view comprises primitive content words as keywords, sentences as context, and pronunciation ordering of keywords. Pronunciation ordering in Japanese is basically a lexicographical ordering of Hiragana notations of keywords. Switching between source and KWIC views is one of the functions associated with function keys.
4. Display homonyms of a keyword by invoking a dictionary server (**Figure 13**) to suggest alternatives for potential homonym errors.

6. Summary and future work

We have introduced the basic concepts and architecture of CRITAC, a knowledge-based proofreading system. The advantages of conceptual and external text representations, a proofreading knowledge base, and a dedicated database system for text proofreading have been discussed.

Because our knowledge base does not yet contain a full complement of proofreading rules, evaluation of the system is still incomplete. Our preliminary results show that about 80% of 50 errors collected from first drafts of technical papers can be detected by our rules.

Future work includes

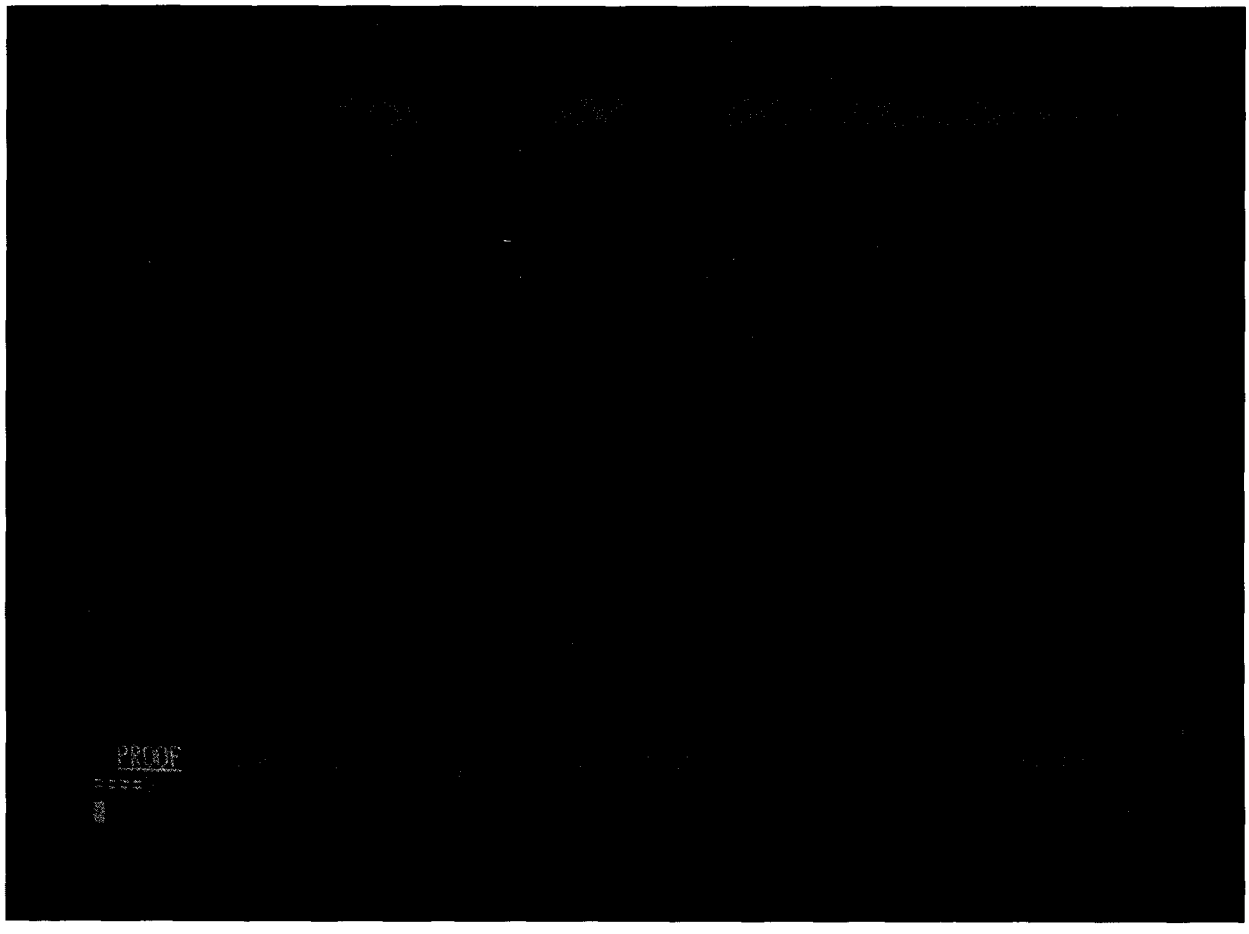


Figure 10

Source view with errors underlined. The numbers preceding the underlines are error-classification codes.

- Collection of text containing errors; classification of errors and writing proofreading rules for them.
- Design and implementation of a simple grammar and grammatical error checking based on case grammar [29].
- Use of the SQL/DS database system to manage the entire structured text.
- Combination of CRITAC with other text-processing application systems, such as a machine-translation system and a document-formatting system.

Appendix: Basics of the Japanese writing system

This appendix is intended for the reader who knows little about the Japanese language and writing system. The Japanese language has three separate sets of characters—*Hiragana*, *Katakana*, and *Kanji*. Hiragana and Katakana are phonetic character sets; they are collectively known as *Kana* characters, and are essentially isomorphic. Each of them has 83 characters (though three more characters are sometimes included in Katakana for special purposes) which are combined to form all the sounds in spoken Japanese; the spoken language has between 100 and 200 distinct syllables. The Kanji character set contains several tens of thousands of characters, but only a few thousand characters are in common use—that is, characters which are used in newspapers and other publications for the nonspecialist. Since many Kanji characters represent pronunciations of one or two syllables and there are fewer than 200 different syllables, homonyms are quite common in words consisting of Kanji characters.

Hiragana characters are used primarily as function words (case markers, aspect and modality markers, etc.) which follow content words. A few content words are also written in Hiragana characters. Katakana characters are used chiefly for words borrowed from foreign languages and for onomatopoetic words. Proper nouns and names in foreign languages are usually transliterated using Katakana characters. Kanji characters, which originated from Chinese

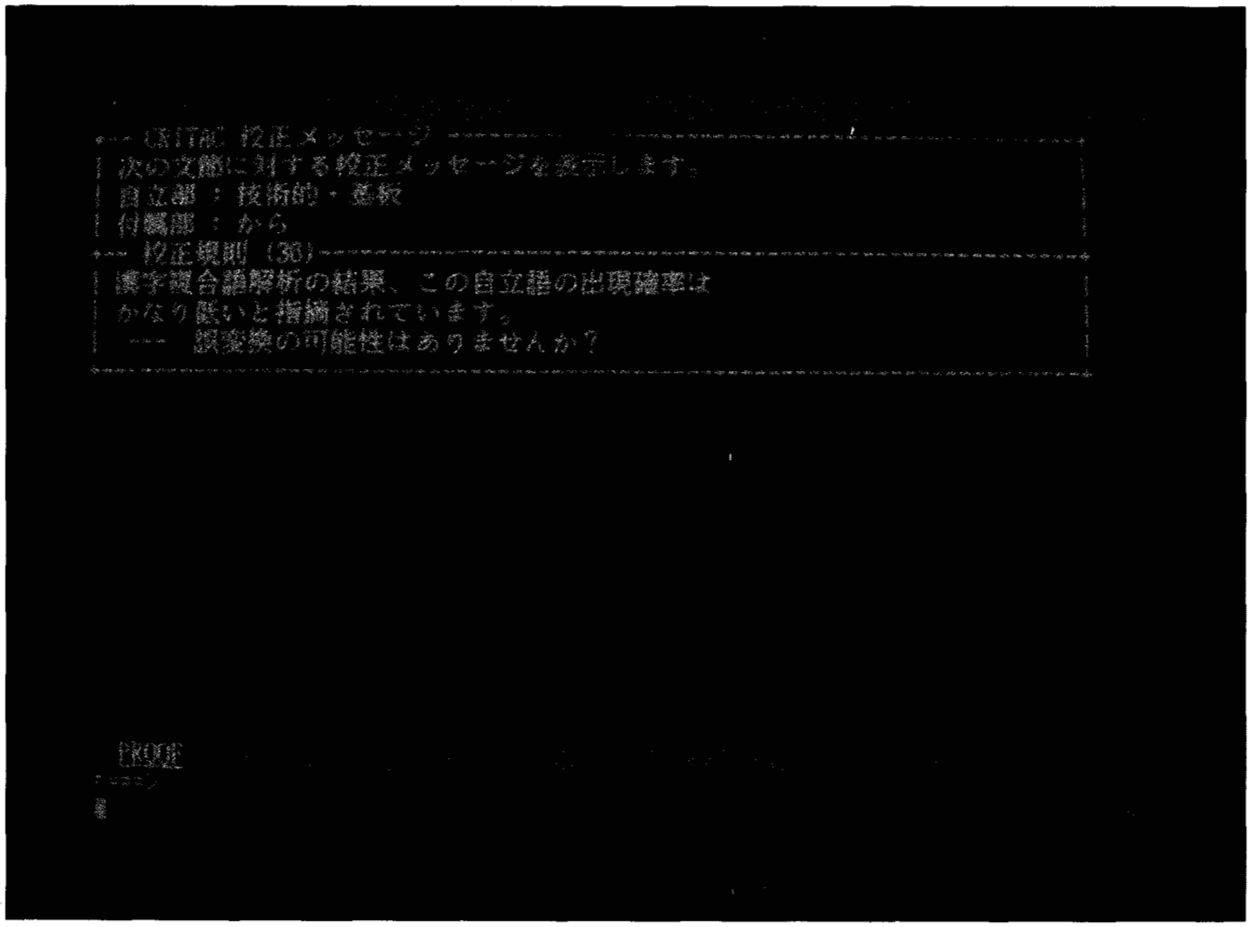


Figure 11

Source view of Figure 10 with explanation of one of the errors (numbered 36).

ideographs, are used for the stems of most content words (nouns, verbs, adjectives, etc.). In addition, Roman characters and Arabic numerals are frequently used to denote foreign words and numbers; thus it is possible for some readers to understand a mathematical paper written in Japanese without knowing the language.

One important property of Japanese text is that characters are written in a continuous stream, so the word boundaries (blanks) must be determined by the reader. The writing system uses commas and periods as in English text, but commas are inserted more arbitrarily. There is a tendency for readers to process Japanese sentences segment by segment rather than word by word. A *segment* (as described earlier) consists of a content word followed by zero or more function words. For example, the Japanese sentence shown in **Figure 14** will be understood by reading three segments, *watashi ha*, *toukyou ni*, and *yu ki ta i*.

The notion of *connectivity* is important in distinguishing meaningful segments. Connectivity can be described as a set of rules determining whether one word (either a content or a function word) can follow another. For example, the third segment in the above sentence is valid only if the three particles appear in this order. Other than connectivity, the Japanese language has very few rules of grammatical agreement, and the order of segments is relatively flexible. Any permutation of the above three segments can be understood, although the totally reversed sequence of segments sounds terrible. It follows that the language has only a few counterparts of the types of grammatical errors listed in [2].

One type of error specific to Japanese text is the Kana-to-Kanji *conversion* error. Japanese word processors in current use are equipped with a Roman alphabet keyboard, just like the ones used for, e.g., English. Each key of the keyboard has

Figure 12

KWIC view obtained by switching from the source view of Figure 10.

been labeled with both a letter of the Roman alphabet and a Hiragana (Katakana) character. With the combination of Roman alphabet keys and a Shift key, a user can type in any Hiragana character by pressing at most two keys at a time. There is a special key called the Kana key, which switches the entry mode between the Roman alphabet and Kana characters. A Kanji word is obtained by entering its sound in Hiragana, and then hitting another special key called the Conversion key. The word processor converts the Hiragana string into its corresponding Kanji word. If there is more than one Kanji word for a given Hiragana string, the word processor shows the most recently used Kanji word among those homonyms first; then, if necessary, the user finds the correct Kanji word by hitting the Conversion key to get the next possibility. Because of the large number of homonyms in Kanji words, the user is liable to make an incorrect choice of a Kanji word that has the correct pronunciation but the wrong meaning. This is called a Kana-to-Kanji conversion error.

Acknowledgments

We would like to thank Hiroshi Maruyama for his work on building an early prototype of the CRITAC knowledge base, Masayuki Morohashi and Shigeki Umeda for their comments and discussions on our approach, and two anonymous referees for their helpful comments and suggestions on this paper.

References

1. L. Cherry, "Writing Tools," *IEEE Trans. Commun.* **COM-30,** No. 1, 100–105 (January 1982).
2. G. E. Heidorn, K. Jensen, L. A. Miller, R. J. Byrd, and M. S. Chodorow, "The EPISTLE Text-Critiquing System," *IBM Syst. J.* **21,** No. 3, 305–326 (1982).

Figure 13

Homonyms returned from the dictionary server. Given a primitive word as a key, the dictionary server displays a list of its homonyms. The user can then select the correct primitive word.

3. K. Takeda, T. Fujisaki, and E. Suzuki, "A Japanese Text Proofreading System—CRITAC," *Proc. COLING '86 (Proceedings of the 11th International Conference on Computational Linguistics)*, Bonn, West Germany, August 25–29, 1986, pp. 412–417.
4. K. Takeda, E. Suzuki, and T. Fujisaki, "A User Interface of a Text Proofreading System," *Proceedings of the IEEE Computer Society Office Automation Symposium,* April 1987, pp. 15–24.
5. S. D. Richardson, "Enhanced Text Critiquing Using a Natural Language Parser," *Research Report RC-11332*, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, August 1985.
6. Jean Fargues, Marie-Claude Landau, Anne Dugourd, and Laurent Catach, "Conceptual Graphs for Semantics and Knowledge Processing," *IBM J. Res. Develop.* **30,** No. 1, 70–79 (January 1986).
7. John F. Sowa and Eileen C. Way, "Implementing a Semantic Interpreter Using Conceptual Graphs," *IBM J. Res. Develop.* **30,** No. 1, 57–69 (January 1986).
8. Adrian Walker, "Knowledge Systems: Principles and Practice," *IBM J. Res. Develop.* **30,** No. 1, 2–13 (January 1986).
9. A. J. H. M. Peels, N. J. M. Janssen, and W. Nawijn, "Document Architecture and Text Formatting," *ACM Trans. Office Info. Syst.* **3,** No. 4, 347–369 (October 1985).
10. A. Walker, M. McCord, J. F. Sowa, and W. G. Wilson, *Knowledge Systems and Prolog*, Addison-Wesley Publishing Co., Reading, MA, 1987.
11. E. F. Codd, "Extending the Database Relational Model to Capture More Meaning," *ACM Trans. Database Syst.* **4,** No. 4, 397–434 (December 1979).
12. E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Commun. ACM* **13,** No. 6, 377–387 (June 1970).
13. C. J. Date, *An Introduction to Database Systems*, Third Edition, Addison-Wesley Publishing Co., Reading, MA, 1981.
14. P. Dadam, K. Kuespert, F. Andersen, H. Blanken, R. Erbe, J. Guenauer, V. Lum, P. Pistor, and G. Walch, "A DBMS Prototype to Support Extended NF2 Relations: An Integrated View of Flat Tables and Hierarchies," *Proceedings of the ACM SIGMOD 1986 Conference on Management of Data*, May 1986, pp. 356–367.
15. R. Hull and R. King, "Semantic Database Modelling: Survey, Applications, and Research Issues," *Technical Report CRI 87-20*, Computer Research Institute, University of Southern California, Los Angeles, March 1987.
16. A. Goldberg and D. Robson, *Smalltalk-80—The Language and Its Implementation*, Addison-Wesley Publishing Co., Reading, MA, 1983.
17. B. Barnett, H. Lehmann, and M. Zoeppritz, "A Word Database

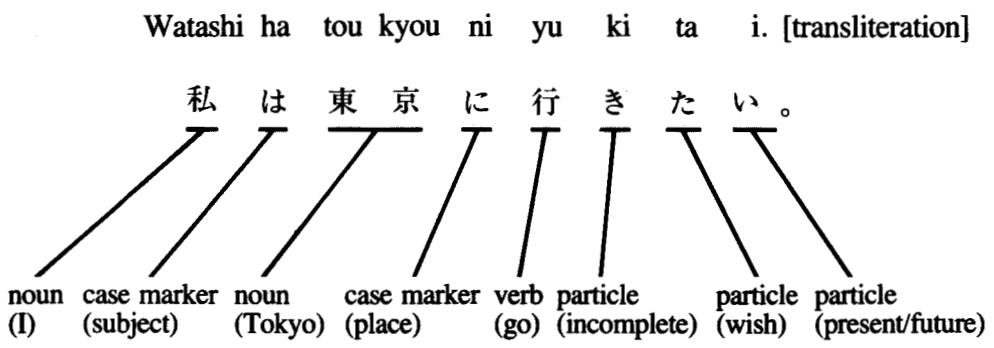


Figure 14

Sample Japanese sentence separated into segments (*bunsetsu*).

for Natural Language Processing," *Proc. COLING '86*, Bonn, West Germany, August 25–29, 1986, pp. 435–440.
18. L. D. Misek-Falkoff, "Data Base and Query Systems: New and Simple Ways to Gain Multiple Views of the Patterns in Text," *Research Report RC-8769*, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, March 1981.
19. K. Jensen, G. E. Heidorn, L. A. Miller, and Y. Ravin, "Parse Fitting and Prose Fixing: Getting a Hold on Ill-Formedness," *Amer. J. Computat. Linguist.* (now *Computat. Linguist.*) **9,** Nos. 3–4, 123–136 (July–December 1983).
20. *SQL/Data System Concepts and Facilities,* August 1983; Order No. GH24-5013, available through IBM branch offices.
21. *VM/Programming in Logic—Program Description/Operations Manual,* September 1985; Order No. SH20-6541, available through IBM branch offices.
22. T. Fujisaki, "Studies on Handling of Ambiguities in Natural Languages" (in Japanese), Ph.D. Thesis, Tokyo University, September 1985.
23. G. David Forney, Jr., "The Viterbi Algorithm," *Proc. IEEE* **61,** No. 3, 268–278 (March 1973).
24. L. R. Bahl, F. Jelinek, and R. L. Mercer, "A Maximum Likelihood Approach to Continuous Speech Recognition," *IEEE Trans. Pattern Recogn. & Machine Intell.* **PAMI-5,** No. 2, 179–190 (March 1983).
25. M. Okochi, "Japanese Morphological Rules for Kana-to-Kanji Conversion: Concepts" (in Japanese), *Technical Report N:G318-1560*, IBM Tokyo Scientific Center, Chiyoda-ku, Tokyo, December 1981.
26. T. Fujisaki, "A Stochastic Approach to Sentence Parsing," *Proc. COLING '84*, Stanford, CA, July 2–6, 1984, pp. 16–19.
27. T. Nishino and T. Fujisaki, "A Stochastic Parsing of Kanji Compound Words" (in Japanese), *Technical Report TR87-0026,* IBM Tokyo Research Laboratory, Chiyoda-ku, Tokyo, April 1987.
28. S. A. Weyer and A. H. Borning, "A Prototype Electronic Encyclopedia," *ACM Trans. Office Info. Syst.* **3,** No. 1, 63–88 (January 1985).
29. C. Fillmore, "The Case for Case," *Universals in Linguistic Theory*, E. Bach and R. T. Harms, Eds., Holt, Rinehart & Winston, New York, 1968, pp. 1–81.

Received March 19, 1987; accepted for publication September 7, 1987

Koichi Takeda *Center for Machine Translation, Carnegie Mellon University, Schenley Park, Pittsburgh, Pennsylvania 15213.*
Mr. Takeda received his B.E. and M.E. in information science from Kyoto University, Japan, in 1981 and 1983, respectively. He worked on database design and theory of nested relations while he was at Kyoto University. Since joining the Tokyo Research Laboratory, IBM Japan, in 1983, he has worked on stochastic Japanese language processing, a Japanese text-proofreading system, and machine translation. Mr. Takeda received an Honorable Mention Award for Outstanding Young Researchers from the Information Processing Society of Japan in 1985. He is currently a visiting researcher at the Center for Machine Translation, Carnegie Mellon University, working on a joint project on English-Japanese bidirectional machine translation. Mr. Takeda is a member of the Association for Computing Machinery, the Information Processing Society of Japan, the Institute of Electronics, Information and Communication Engineers, and the Japan Society for Software Science.

Emiko Suzuki *IBM Tokyo Research Laboratory, 5-19 Sanbancho, Chiyoda-ku, Tokyo 102, Japan.* Ms. Suzuki is a member of the Japanese Processing Group at the Tokyo Research Laboratory, IBM Japan. She received her B.S. and M.S. in information sciences from

the University of Tsukuba, Ibaraki, Japan, in 1981 and 1983, respectively. Ms. Suzuki has been working on Japanese language processing, a proofreading knowledge base, and a Japanese grammar for a machine-translation system since joining IBM in 1983. She is a member of the Information Processing Society of Japan and the Japan Society for Software Science.

Tetsuro Nishino *Tokyo Denki University, Department of Information Sciences, Hatoyama-machi, Hiki-gun, Saitama 350-03, Japan.* Mr. Nishino received his B.S. and M.S. degrees in mathematics from Waseda University, Tokyo, Japan, in 1982 and 1984, respectively. At the Tokyo Research Laboratory, IBM Japan, he worked in the areas of attribute grammar and stochastic natural-language processing from 1984 to 1986. He is currently a research associate in information sciences at Tokyo Denki University. Mr. Nishino's research interests include theory of automata, formal languages, and computational complexity. He is a member of the Japan Society for Software Science.

Tetsunosuke Fujisaki *IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598.* Dr. Fujisaki is currently manager of the Hand Input Recognition Group of the IBM Thomas J. Watson Research Center. He joined IBM in 1971 at the Tokyo Scientific Center, IBM Japan, and, until 1983, worked in areas of Japanese language processing, including database-query understanding and text processing. Dr. Fujisaki was a visiting lecturer at the Tokyo Institute of Technology during the period from 1979 to 1981. In 1982 and 1983, he spent eighteen months at the Thomas J. Watson Research Center continuing research on probabilistic language parsing. From 1983 until 1986, he managed an AI group at the IBM Tokyo Research Laboratory. Dr. Fujisaki obtained his B.S., M.S., and Ph.D. degrees in mathematical engineering from the University of Tokyo in 1969, 1971, and 1985, respectively. He is a member of the Association for Computational Linguistics, the Association for Computing Machinery, the Information Processing Society of Japan, and the Institute of Electrical and Electronics Engineers.