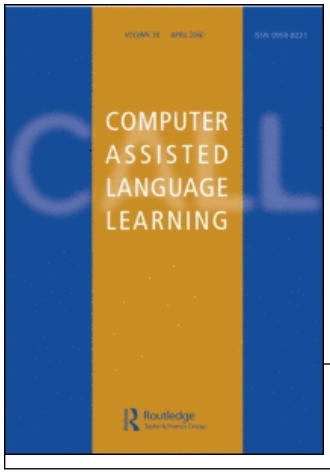

Computer Assisted Language Learning
Publication details, including instructions for authors and subscription information:
http://www.informaworld.com/smpp/title~content=t716100697

A RULE-BASED APPROACH TO COMPUTER-ASSISTED COPY-EDITING

Robert Dale [a]
[a] University of Edinburgh,

Online Publication Date: 01 January 1990

To cite this Article: Dale, Robert (1990) 'A RULE-BASED APPROACH TO COMPUTER-ASSISTED COPY-EDITING', Computer Assisted Language Learning, 2:1, 59 — 67

To link to this article: DOI: 10.1080/0958822900020105
URL: http://dx.doi.org/10.1080/0958822900020105

CALL, Volume 2, pp. 59-67.

A RULE-BASED APPROACH TO COMPUTER-ASSISTED COPY-EDITING

Robert Dale[1]
University of Edinburgh

Introduction

The copy-editing problem

Word processing is probably the single largest application of personal computing; and yet, although computers have made it easy to put words on paper, so far they have provided very little help in ensuring that the result is high-quality, error-free text. Spelling checkers are now used widely, but there are many kinds of textual errors that fall outside their scope. It is possible to buy software which claims to help the writer with matters of grammar and style, but these tools are very simple and of limited use.

The problem of textual errors is more severe in the publishing world, where the standards required are so much more exacting. In recognition of this, publishers typically employ a division of labour, with the copy originator being primarily responsible for the initial content of a text, and the copy-editor being responsible for its final form. The editor's job often involves extensive rewriting and high-level re-organization of a text, but the most time consuming tasks are copy-editing and proof-reading, where the major concern is with the checking of lower-level detail.

This paper describes a system whose purpose is to assist a human writer or editor in massaging a text to deal with these kinds of errors. The core idea behind the system is a simple but very powerful one: rules of grammar, style, punctuation and usage can be maintained as rules in a knowledge base, distinct from the mechanism that applies those rules against a text. This means that rule bases can be switched or modified at will, and the system's behaviour will change as a result. Rules are then applied against a text, yielding suggestions for improvements and corrections which the user can choose to accept or reject as desired.

Currently, the research described in this paper is concerned with assisting an editor in ensuring the correctness of low-level matters such as the use of punctuation, the format of numbers and numerical values, and the use of abbreviations. In many such areas, there are no right or wrong ways of doing things: what is important is *consistency*, and so different publishers may adopt different conventions, each such set of conventions being referred to collectively as a 'house style'. House style also

---

1 Much of the work described here was carried out while the author was employed by Syntek Ltd., Edinburgh.

encompasses typographical issues such as the formatting of quotations, titles and section headings, captions and legends, and so on: many of the problems in these categories are now being eased with the increased acceptance of standard markup languages such as SGML. For the copy-editor, however, tidying up the lower-level details remains a laborious task. It can also be quite a mind-boggling task: in book publishing in particular, an editor may be responsible for as many as twenty or so manuscripts in various stages of production, with different style sheets for each manuscript. A principal aim of the system described here is to overcome the difficulties inherent in managing tasks of this kind.

A knowledge-based solution

Many aspects of the copy-editor's task are still not well enough understood to be encoded in a program; other aspects, however, are relatively straightforward and mechanical. We take the view that it is useful to provide computer assistance with the better understood tasks, so that the copy-editor can expend more effort on those parts of the job that are more intellectually demanding.

The *Editor's Assistant* project started out with several design goals:

- the system should use a knowledge-based systems approach, with individual house style rules being maintained as rules in a knowledge base, thus permitting easy modification and customization;
- the system should be interactive in nature, so that the user has the option to accept or reject the system's recommendations: this was particularly important as, in the tradition of heuristic-based systems, the program's suggestions are likely to be wrong on some occasions;
- the mechanisms used should be sufficiently modular and general that, as more of the mechanics of the copy-editing task are understood, the capabilities of the system can be enhanced accordingly.

The current prototype has satisfied the first two of these goals, only further development will show whether we have been successful in the third.

An oft-cited problem in the construction of knowledge-based systems is what has become known as 'Feigenbaum's bottleneck': the problem of knowledge elicitation. A correspondingly off-cited solution to this problem is to choose (provided you have the choice, of course) a domain where the relevant expertise is available in written form. At first glance, the task of copy-editing seems to satisfy this requirement: editors typically work from written style sheets, and versions of these are often published. Two of the more well-known works of this sort are *Hart's Rules for Compositors and Readers* (Oxford University Press, 1983) and *The Chicago Manual of Style* (University of Chicago Press, 1982). Of course, these books were not written with our purpose in mind, with the result that formalizing the knowledge they contain is not trivial. This leads to the more long-term theoretical aim of the current

work: the construction a taxonomy of style issues and the development of a language for talking about texts. Some necessary first steps have been made here, but the implementation of a working prototype has remained the primary goal.

Related research

The possibility of using an 'expert systems' approach to address the copy-editing problem was first suggested by Lefrere, Waller & Whalley (1980), but does not appear to have been pursued any further by these researchers. One particular aspect of the copy-editing and proof-reading process, namely the checking of proper names, was addressed directly by Ishii (Ishii, 1985).

There have been other systems that address related concerns: the most well-known 'style checkers' are the UNIX Writer's Workbench (WWB) programs (MacDonald, 1983) and IBM's EPISTLE (Heidorn *et al.*, 1982) and CRITIQUE (Ravin, 1988). However, these systems are not concerned with the specific notion of house style used in publishing, and, more significantly, do not make use of a rule-based approach in the sense described here. In both cases, the user has very limited control over the kinds of things the system can deal with. In the *Editor's Assistant*, however, the aim is to construct a general language where the user can easily specify new rules or modify existing ones, in much the same way as this is done in current expert system shells. In terms of general architecture, this contrasts quite radically with the approach taken in the WWB system, for example. The WWB is a suite consisting of a considerable number of distinct, specialized programs: one of these detects occurrences of double words, another detects 'non-preferred phases'; and so on. In our system, however, we aim for a generic solution, with all the different tasks being described and dealt with by the same rule language.

The user's view of the system

The *Editor's Assistant* operates by interactively detecting and, where possible, offering corrections for those aspects of a text which do not conform to the rules of style embodied in the knowledge base in use. From the perspective of the user, the system operates in much the same way as the more sophisticated of currently available spelling correction programs. When running, the text file to be proofed is loaded into the editor and displayed on the screen. The program first performs some preprocessing of the text in order to build an internal representation that can be analysed by the rules. The rule application mechanism then applies the rules in the rule base against each sequence of words in the sentence. If a sequence of words causes a rule to trigger, that rule then takes control: the words in question are highlighted on the screen, and a menu of alternative actions appears. Using a mouse, the user then chooses the option to be executed and the screen display is updated accordingly. There are always at least four options on the menu:

- an option to replace the highlighted string of words by something else: sometimes there may be more than one replacement option;

- an option to ignore the rule on this occasion, so that it has no effect on the text currently highlighted;
- an option to disable the rule that has fired: this is appropriate when you realize that a particular rule is not applicable to the text with which you are currently dealing;
- an option to apply the rule automatically for the rest of the text, so that on subsequent occasions when the rule fires, substitution will happen without the user being asked for confirmation.

Using this kind of interface means that the user can perform a considerable amount of editing work with a minimum amount of effort.

System operation

The two essential ingredients in the system's behaviour are the structures used to represent style rules, and the preprocessing of the text which results in a data structure upon which the rules can operate.

Style rules

The rules used by the system are of various types. The simplest kinds of rules are those which match single words or sequences of words in the text, and specify one or more possible alternative words or phrases, much as is done in current 'style checkers'. However, the capabilites of the approach used here go far beyond rules of this sort.

- More complex rules match *patterns* found in the text, and specify replacements on the basis of these patterns. For example, one rule finds any sequence of two identical words (a common error introduced in text input at screen line breaks), and suggests replacing them by a single instance of the word; another rule identifies any date specified in the form **Day Month Year** and suggests the form **Month Day Year** as a replacement (a typical house style requirement). Similar pattern-based rules identify misspelling of *their* as in *their are* and misuse of *a* as in *a onion*.
- Yet more complex rules can perform arbitrary operations on the basis of the words that trigger them. For example, one rule detects any measurement specified in kilometres and offers a converted measurement in miles; another offers to insert the expansions of little-known abbreviations, provided, of course, the system knows what the expansion should be: if it does not, it can prompt the user to supply one.

Because the rule bases are separate from the mechanism that applies those rules, new rule bases can be constructed and used as required.

Each rule contains four items of information.

ApplicationMode: this can be one of three values: **QueryApply, AutoApply,** or **Ignore.** For a given rule, the action taken by the rule application mechanism depends on the value of this variable. **Ignore** means that the rule is not to be used; **AutoApply** means that the rule is to take effect automatically (i.e. without asking the user for confirmation); **QueryApply** means that the rule will only take effect if the user accepts the system's recommendation. Rules can be initialized to whatever state is appropriate, and modified during the use of the system as described above.

Trigger: this determines whether or not the rule applies in a given situation, by specifying, with reference to the contents of the tokens in the analysed text, a set of conditions that must be true for the rule to apply.

Replacements: the **Replacements** slot specifies a set of one or more replacements that may be made for the text string that was being considered when the rule triggered. These are in the form of expressions which, when evaluated, create new sequences of token for insertion into the text. If there is more than one such replacement specification in a given rule, one of these is marked as being the default replacement for use in situations where the rule in question operates in **AutoApply** mode.

CorrespondenceTable: this specifies, for each token in the replacement string, the token this corresponds to in the string that triggered the rule: this is used to control the inheritance of information, as described below.

In order to see how these rules are used, we must first describe the internal representation used for the text to be proofed.

Text Preprocessing

A text is represented as a hierarchical structure consisting of paragraphs, sentences and tokens. The current system does not incorporate a natural language parser, and so there is no structure intermediate between the sentence and the tokens that make up the sentence; however, it would be straightforward to incorporate more complex structures in this hierarchy if required.

Tokens can be of two types: **Word** tokens and **Punct** tokens. A **Word** token is used to represent, naturally, a word, along with any punctuation which is properly part of that word. A **Punct** token is used to represent any contiguous sequence of (non-word-internal) punctuation characters in the text. The ASCII character set does not divide straightforwardly into those characters which form **Words** and those which form **Puncts:** some characters can belong to either, depending on the context. There are a number of these ambiguous characters : the full stop or period is the most common, as it can appear as a sentence terminator (in which case it is part of

a **Punct**) or as punctuation within an abbreviation (in which case it is part of a **Word**).[1] So, in

the rhino(s) (eventually) ate the cake

the character string *rhino(s)* would be represented by a single **Word** token, as would the string *eventually*; the open parenthesis immediately before the first *e* in *eventually* is not part of a **Word** token, but is part of a **Punct** token, consisting of a space and an open parenthesis and falling between the two **Word** tokens. A number of heuristics are used in the text preprocessor to decide how to tokenize the text in such cases. This approach to tokenizing the text is unusual (most other systems would disregard the spaces between words as being separators), but it means that we can apply the analogue of spelling rules to punctuation tokens.

Each token is represented by an object that maintains information derived from the analysis of the word to which it corresponds. In the case of **Word** tokens, this structure contains the following information (analogous information is maintained in the case of **Punct** tokens):

Contents: the unanalysed token as it appears in the text

Type: a tag used to identify this token as one of a number of types determined on the basis of the characters it contains: in the context of style checking, it proves useful to have categories such as **LowerCaseWord, CapitalizedWord, AbbreviatedWord, CompoundWord, Number** and **CurrencyValue,** which can be used to determine quickly if a particular rule is applicable.

In addition, the **Contents** of the token are analysed into a **Root** and two sets of **Features.** The Root is similar to the notion of root used in morphological analysis: it represents a form of the token from which various details have been abstracted away. These details are registered by means of the **Features** slots: **SyntacticFeatures** is a specification of syntactic features (such as word class and number) and their values, and **TypographicFeatures** is a specification of typographic features (such as casing and font style) and their values. At any time, the **Root** of the token can be amalgamated with the information in the **SyntacticFeatures** and **TypographicFeatures** slots to reconstruct the **Contents.**

1 In certain circumstances, a full stop can perform both functions simultaneously, as in the word at the end of this sentence: B.B.C.

Rule application

Once analysed, a text can be viewed as a token string. Rules operate on substrings of this token string. By means of its **Trigger**, each rule matches a set of possible token substrings: effectively, a **Trigger** provides a specification for each token in the substrings it matches. Each token specification may be partial, in that it only specifies some aspects of the token in question: in such cases, it is possible that one rule will match a large number of possible token substrings.

The token specifications held in the **Replacements** slot may be of two types: each may be simply a pointer to a token in the source string, indicating that the original token is to be used, or it may be a specification for the construction of a new token. Again, these may be partial specifications. In order to produce a replacement for the token substring that caused the rule to trigger, a *partial* specification must be fully instantiated by inheriting data from the corresponding token in that substring, as defined by the rule's **CorrespondenceTable.**

This simple mechanism allows us to reduce vastly the number of rules that are required in the system. To take a simple example from a conventional style checking domain, suppose we require a rule that specifies that every occurence of *foo* is to be replaced by *bar*. If we also want to carry out this replacement at the beginning of sentences, we can write a rule whose **Trigger** requires that the **Root** should be *foo*, but says nothing about the **Case**; and the **Replacements** slot of this rule will similarly make no specification regarding **Case**. Using a somewhat simplified notation, this can be expressed as

Trigger: **FirstToken:Root = "foo"**
Replacement: **FirstToken:Root = "bar"**

The trigger will then match each of the following input tokens:

Contents: **"Foo"**
Root: **"foo"**
TypographicFeatures:Case: **capitalized**

Contents: **"foo"**
Root: **"foo"**
TypographicFeatures:Case: **lower**

In building the replacement token, the instantiation mechanism then adopts the case found in the triggering token string. It is not only the values of features that may be inherited by the new token substring. So, for example, suppose we want a rule that specifies that the first word following a colon should not be capitalized: the rule will look like the following.

Trigger:	**FirstToken:Contents**	**= ":"**
	SecondToken:Case	**= capitalized**
Replacement:	**FirstToken**	**= FirstToken**
	SecondToken:Case	**= lower**

When the second token in the replacement string is constructed, everything but the **Case** feature will be inherited from the corresponding token in the input string.

Each sentence in the text may go through a number of changes as rules are applied to it. Changes made to the text are never destructive: when new tokens are added to the text, a new *version* of the sentence in question is created. Each version consists of a list of pointers to the tokens that make up that version. This allows the effects of rules to be undone, although the facility for making use of this information is not currently implemented.

Current status and future work

The system described above was first constructed as an early prototype written in Interlisp-D and LOOPS, running on a Xerox 1186 AI workstation. The system has been subsequently ported to Common Lisp running on a PC-AT. On this hardware, the system performs almost acceptably fast, but has severe limitations on the size of file it can process due to the lack of a virtual memory facility under MS-DOS: the internal representation of a given text is many times larger than the text itself.

Two major enhancements are planned. First, from consultations we have carried out with working editors, it has become obvious that we need to allow the user to edit the text manually, simultaneously with the system's normal proofing operation; and second, we require the addition of some syntax checking capabilities, although it is not yet clear what form these will take. Given the unconstrained nature of the input, it is likely that we will use some combination of chart parsing (Thompson, 1983) and the 'fitted parsing' techniques developed for EPISTLE (Jensen & Heidorn, 1983). Apart from the need to deal with grammatical errors, there are also good implementational reasons for using a parser to enrich the structure of the text representation described above: doing this would allow us to apply rules on the basis of syntactic constituency, so that, for example, one class of rules will apply to noun phrases, and another to prepositional phrases.

Another, nearer-term, goal in the development of the system is the continued working out of a language for describing style rules. Currently this is fairly rudimentary: there are few signposts that indicate what the vocabulary of this language should be, and so it is only by continued effort in increasing the coverage of the system that we will reach a satisfactory answer.

The techniques described here are very simple, but have considerable scope and generality. Integration with even partial syntactic processing should lead to new capabilities for structure editors oriented towards natural language.

References

Carbonell, J.G. & Hayes, P.J. (1983) 'Recovery strategies for parsing extragrammatical language'. *Americal Journal of Computational Linguistics,* 9, 123-146.

Heidorn, G.E., Jensen, K., Miller, L.A., Byrd, R.J. & Chodorow, M.S. (1982) 'The Epistle text-critiquing system'. *IBM Systems Journal,* 21.

Ishii, S. (1985) 'Proofreading Japanese Word Usage and Proper Nouns by Computer'. *Technical Report No.125,* ICOT, Tokyo.

Jensen, K. & Heidorn, G.E. (1983) 'The Fitted Parse: 100% Parsing Capability in a Syntactic Grammar of English'. In *Proceedings of the Conference on Applied Natural Language Processing,* Santa Monica, California, February, 1983, 93-98.

Lefrere, P., Waller, R.H.W. & Whalley, P. (1980) 'Expert Systems' in Educational Technology? Chapter 12.5 in Winterburn, R. & Evans, L. (eds) *Aspects of Educational Technology, Volume XIV: Educational Technology to the Year 2000,* 338-343. London: Kogan Page.

MacDonald, N.H. (1983) 'The Unix Writer's Workbench software: rationale and design'. *Bell System Technical Journal,* 62.

Oxford University Press (1983) *Hart's Rules for Compositors and Readers,* 39th edn, Oxford: Oxford University Press.

Ravin, Y. (1988) 'Grammar Errors and Style Weakness in a Text-Critiquing System'. *IEEE Transactions on Professional Communication,* 31, 108-115.

Stutely, R. (1986) 'The Standard Generalized Markup Language'. In Earnshaw, R.A.(ed.) *BCS Conference on Workstations and Publication Systems,* London, October, 1986.

Thompson, H.S. (1983) 'MCHART - a flexible, modular chart parsing framework'. In *Proceedings of the 3rd Annual Meeting of the Americal Association for Artificial Intelligence,* Washington, DC, 1983.

Weischedel, R.M. & Sondheiner, N.K. (1983) 'Meta-rules as a Basis for Processing Illformed Input'. *Americal Journal of Computational Linguistics,* 9, 161-177.

The University of Chicago Press (1982) *The Chicago Manual of Style,* 13th edn. Chicago, Illinois: The University of Chicago Press.