Word Prediction Strategies in Program Editing Environments

Ian Sanders Chia-Ling Tsai
Department of Computer Science, University of the Witwatersrand, 1 Jan Smuts Avenue, Johannesburg

Abstract

The goal of this research is to demonstrate that word prediction in a constrained domain like a programming language can be done effectively and hence has the potential to facilitate typing for disabled people who are motor-impaired with regard to typing. This paper presents a set of word prediction strategies for a programming editor. In order to evaluate the effectiveness of these strategies, a system which simulates a Turbo Pascal editor was developed. The strategies are found to be efficient and improve the accuracy of word prediction to various degrees. Although the system is a simulator of a Turbo Pascal editor, the applicability of each of the strategies is not confined to the syntax of this language and, therefore, they can be implemented on any language-directed editors. The results could also have relevance for other text generation systems where the aim is to save the user effort.
Keywords: *Disabled users, word prediction, prediction strategies, predictive editors*
Computing Review Categories: *K4, J2*

1 Introduction

Computers are becoming almost indispensable in modern life and are used by people from all walks of life. A very common activity in computer usage is that of text generation — writing electronic mail, preparing reports and developing programs. In this area of text generation, the keyboard is widely accepted because of its low-cost and easy-to-use features. People with motor disabilities (for example, someone who has been affected by polio in their hands or arms and whose motor abilities have been impaired) often cannot type at a reasonable speed, and the communication process with computer systems becomes a painstaking process via a keyboard. In some instances a pointing device (mouse, etc.) and appropriate software are a good way to address the problem and, in addition, several systems (touch screens, voice recognition systems, etc.) and techniques have been developed, especially for disabled people, to improve the rate of text generation through different input devices. Unfortunately these input devices, e.g. eye typer [8], are often costly for target users.

Programming (a specific type of text generation) is an area where motor-impaired people can be gainfully employed if some way of addressing their difficulties with using a keyboard (or other text generation environment) can be found. The premise of this paper is that programming can be made more efficient for a motor-impaired person using a keyboard by predicting the next word to be typed by the programmer. The prediction strategies presented here could also be useful in a keyboard plus pointer type of system or indeed any system which is intended for text generation in a constrained domain.

Predictive editors (for example, R-K Button [3], a UNIX interface and Reactive Keyboard [4], a general purpose editor) have been implemented as tools that adopt prediction strategies to speed up typing by reducing the number of keystrokes required. Some of these have already found application as environments for physically disabled people. These tools have proved to be a hindrance to good or touch-type typists, but are useful writing tools for fair to poor typists, and are almost indispensable for disabled people [4].

This research is aimed at investigating approaches to make the typing of programs in some programming language a more simple task for motor-impaired users. The idea is to use information about the language being typed and the programmer's "style" to accurately predict the next word or pattern to be typed. This implies an editor which is a combination of a language-directed editor and a text-prediction system or editor. Language-directed editors [2, 5, 7, 9] generally concentrate on the efficient editing of syntactically and semantically correct programs without the intervention of the compiler (See also Welsh *et al* [10]). These editors concentrate mainly on the correct form of the program and do not place much emphasis on saving keystrokes by predicting the next word — variable name, variable type, etc. — that the programmer will type.

This paper discusses some strategies for the prediction of the next word to be typed by a programmer. The strategies were tested by developing a simulation of a predictive editor for the Turbo Pascal programming language. The prediction strategies (discussed in section 2 and section 3) are based on some of the features of the language and the programmer's use of variables, types, etc., and are used to guide the prediction. The simulation works as if the program was typed in from beginning to end in one session and records the number of key strokes which could be saved and the number of accurate predictions which would have been made by a system using the strategies. An editor which implemented these strategies would clearly not have this restriction but as the strategies are based on the content of the program there should still be keystroke savings. The experiment which was performed to test the strategies is described in section 4.

The results (see section 4) achieved show that making use of program structure and knowledge of what the programmer has done in the past can lead to accurate predictions and save on the number of keystrokes which must be typed. An editor using these ideas could be of immense benefit to a disabled programmer who has difficulty in typing — up to 40% of his/her typing could be saved by using these strategies. This figure could be potentially be increased by using programming templates but this is future research.

2 Prediction Strategies

Programs are highly structured text which must obey various rules to be syntactically and semantically correct. Language-directed editors make use of this structure to develop correct programs (after parsing) and this use of structure can be expanded upon by a text predictor which disallows predictions of words (or templates/patterns) which are syntactically impossible at the point of editing.

The variable names and types declared by programmers are often used repeatedly in programs and sometimes even across programs by the same programmer. A system which can accurately predict which of the names or types (or other parts of the program) the programmer wishes to use at any point in the program has the potential to save the programmer a considerable amount of typing. These ideas lead to the hypotheses discussed below which could lead to strategies for reducing typing.

1. The locality hypothesis states that at certain parts of a program some words are inhibited and these words should be excluded from the set of words from which the predictor is allowed to choose.
2. The recency hypothesis states that a word that the user has typed in the recent past is likely to be used again in the near future.
3. The repetitiveness hypothesis states that if a word has appeared frequently in the past, the word is likely to be used again in the future.
4. The optimality hypothesis states that a system should be able to apply locality, recency and repetitiveness together in order to achieve the best performance in terms of accuracy for text prediction.
5. The continuation hypothesis states that predictions done on very similar programs (programs with very similar variables names, type declarations, etc.) can be improved if information gained in one program is carried over to the other programs.

The locality hypothesis should prevent the predictor from making predictions which are syntactically impossible at the point of editing. For instance, the word `while` cannot be a candidate for the predicted word if the prediction takes place in the declaration part of the program. The ideas behind the recency and repetitiveness hypotheses originate from Lerner's research on the automated customization of structure editors [6]. The application of these two ideas to text or word prediction will hopefully increase the accuracy of prediction and thus ease the task of the programmer. The optimality hypothesis is based on the idea that if recency and repetitiveness are helpful then there should be some way of maximising the usefulness of these strategies by combining them. The continuation hypothesis again originates from Lerner's work on automated customization [6]. Very often a set of programs with similar variables are developed by a single programmer for a special purpose. The idea here is to treat a set of similar programs as a unit rather than as a number of discrete parts and thus achieve better predictions overall. Strategies based on these hypotheses should lead to good prediction and thus a saving of effort by the programmer.

3 Implementation of Prediction Strategies

In order to test the ideas discussed above, a system which simulates a predictive editor for a block-structured language, in this case Turbo Pascal, was developed. Although the system simulates a Turbo Pascal editor the ideas should be applicable to other languages.

Basically the simulator works by reading sample programs in as though they were typed in from first line to last line by a programmer. It is also possible to order the lines of code in the test data to simulate a programmer typing in "stubs" before going back and filling in the details of the various procedures and functions. Other modes of input can also be modeled.

As each word is encountered — the first letter of the word is encountered — the system predicts which word is to appear (in a full system, which word the programmer is about to type). If the prediction is correct then the word is accepted and input moves to the next word. If the prediction is not correct then further predictions are considered. This is done by considering the next letter in the current word. In a full editor using these strategies this would be when the programmer types the next letter in the word which he/she is entering. The reason that prediction is handled in this way is that if an incorrect prediction is offered then the programmer has to indicate that it is incorrect — this means another keypress or some other action. Using the next letter of the current word to indicate that the prediction was incorrect and at the same time to activate the next prediction is an efficient way of handling this problem. Clearly in a full editor the programmer would still have to accept the prediction offered, if it is correct, but this is more efficient than typing the complete word. This acceptance could be the typing of a valid program character which would be the separator (a space, comma or colon for example) between the word being predicted and the next word in the program.

This process of predicting the word to be typed, based on what has been seen so far, is continued until the complete word has been read by the simulator or until an accurate prediction based on the portion of the word read so far is made. This will then be repeated for all other words in the program.

The simulator works with a dictionary of words which can be predicted. The dictionary is initialised with the key words of the target language and updated as the simulator encounters new words in the program being processed (equivalent to the programmer typing new words in his/her program).

The simulator does not measure issues like the number of times a programmer makes mistakes and has to correct them or model how a programmer would move about in a full editor but it does give a measure of how well the predictions work and how many keystrokes could potentially be saved in such an editor.

The locality strategy was implemented by using a field in the word records to show the location restriction(s) of the words in the dictionary. If a word is syntactically impossible at the point of editing, the word will not be regarded as a valid match. A data type, for example, can never appear in the body of a Pascal program. Among those valid matches, if a word is appropriate to be the first word of a new line and the cursor is not on the first word, its confidence value will be lower than those valid words which have no location restrictions. A word can have more than one location restriction. The word `var`, for example, is restricted to being the first word of a line (no words can come before it in a valid Turbo Pascal statement) and it must appear in the declaration section of a program.

The recency strategy is applied by checking if a word in the dictionary has occurred recently — by recent is meant that it has been read from a line which appears in a range of lines read just before the current line from the test data. The line number of the last occurrence of the word is subtracted from the current line number. The smaller the result is, the more recently the word has occurred, and the higher the confidence value will be. If the last occurrence of a word is beyond a certain range the recency value is set to zero, and the word is not considered as being recent. This range is termed the scope of the recency.

To implement the repetitiveness strategy, a frequency counter is added to the record of the word. Every time a word is found in the test data file, the frequency counter of that word increases by one. The greater the frequency counter, the higher the confidence value.

To pursue the optimality of the accuracy of the predictions different combinations of recency and repetitiveness need to be tested. The system was designed to be able to deal with this.

To implement dictionary sharing (testing the continuation hypothesis), each word in the dictionary is assigned an attribute, e.g. variable, type, etc., and only those words with a variable attribute and their frequencies are exported to a file at the end of a run. If the system requires information from the previous simulation, the file, where those words and their frequencies are stored, is read into the dictionary.

The simulator predicts every single input word in four different ways, namely using a first-word strategy (that is effectively no prediction at all), using the locality strategy, using the locality and recency strategies, and using the locality and repetitiveness strategies. Using the first-word strategy, a prediction is chosen based on the first valid match (using the n-gram technique [4]) found in the dictionary. This strategy employs no heuristics for predictions. Using the locality strategy, the system precludes some predictions that are syntactically impossible — the first syntactically correct match is selected. Every valid match is assigned a confidence value which depends solely on the location constraints — 0 if the word is disallowed at this part of the program, 100 if it is only allowed in the current position (in the declarations or header section) and 50 in other cases. The higher the confidence value is, the more likely the word can be chosen as the predicted word. Thus predictions are based on the confidence values of valid matches. Using the recency strategy, the confidence value becomes a function of locality and recency. Based on the confidence value obtained from locality, the value increases accordingly if the match word appeared in the recent past — a word in the most recently read line will been given the maximum confidence value, a word which appears at the maximum distance (which is within the recency scope) from the current line is given the lowest confidence value. Words outside the scope of the recency strategy are given a 0 confidence value. Using the repetitiveness strategy, the same principle as recency is applied, and the confidence value becomes a function of locality and the frequency of the valid match.

The locality strategy precludes impossible predictions in an attempt to improve the accuracy, whereas recency and repetitiveness strategies attempt to improve the accuracy by selecting the valid match with the highest confidence value. The reason for applying recency and repetitiveness strategies separately is that their effects on the predictive editor were unknown initially, and it was thus initially difficult to determine the confidence value as a function involving recency and repetitiveness. After the evaluation of these strategies, further research was done to investigate the optimal function of confidence value in terms of locality, recency and repetitiveness.

4 The Experiment

Design

The test data in this experiment are Turbo Pascal programs written by various users. The only restriction being that the programs must compile and run in the Turbo Pascal environment. The data set consists of 89 programs which range in length from 14 lines of text to 3241 lines of text. Note that the number of text lines is not always exactly the same as valid lines of code — some programmers may put more than one line of code per text line — but the number of text lines does give some indication of the size of the program. In the test set there are eleven programs which have very similar variables and type names and were written by one programmer working on a research project. These programs were used to test the idea of dictionary sharing.

For each prediction method, the output from the simulator consists of four figures, namely the number of keys pressed, the number of keystrokes saved, the number of predictions made and the number of predictions accepted. The first two figures are for the computation of the percentage of keystrokes saved out of total keystrokes required — a good reflection of the effectiveness of the strategies. The last two figures are for computation of the percentage of predictions accepted out of total predictions made — the accuracy of the strategies. The total number of keystrokes required excludes white space and comments.

The whole experiment was broken down into three stages. The first stage is for the evaluation of the effectiveness of the four strategies, i.e. first-word, locality, recency and repetitiveness. In the second stage, recency and repetitiveness were combined in various manners to attempt to find the best way to integrate recency and repetitiveness together on one system in order to improve the result obtained in the first stage. In the last stage, the testing was concentrated on evaluation of the continuation hypothesis — dictionary sharing.

Results

Phase 1 — Testing the strategies

The locality hypothesis claims that the accuracy of the prediction should be increased if syntactically incorrect predictions can be inhibited. In this experiment the recency and repetitiveness strategies are applied after the locality strategy as this seems to be an appropriate way to apply the strategies. Various recency scopes were tested but only four are reported here. Recency(20) assigns recency values only to those words which occurred within the previous 20 lines, while recency(70) does the same thing to words within the previous 70 lines. Similarly for Recency(05) and Recency(10). The results of the first phase of testing are shown in Table 1. The probabilities given here are based on the zero hypothesis that the various strategies are no better than simply picking the first word which begins with the correct letter (the current letter being typed). The statistics are calculated using random sampling techniques as discussed by Box *et al* [1] — reference distribution based on random sampling, internal estimate of σ. The probabilities reported give the probability that the result returned by applying a strategy could be simply a random result returned by the zero hypothesis — a result within the range of results expected if no prediction strategies were applied. This means that a high probability value indicates that there is no significant difference between the strategy and the zero hypothesis. A low probability value indicates that the results are statistically significantly different from the zero hypothesis. This means that there is a statistically significant difference in the results. The smaller the probability the greater the confidence level of the difference.

(Note: In all of the tables the % Keys Saved and %Predictions accepted refer to the averages over the test. Also note that the variances have not been converted to percentages although the averages have.) This table shows that the locality hypothesis offers slight improvements (but with fairly low confidence). Recency(20), Recency(70) and Repetitiveness offer much better performance (improved key stroke savings and better prediction) and with much higher confidence. This makes sense as once a word has been encountered, the strategies should offer the word as a valid prediction and if it is correctly offered there should be keystroke savings.

Table 2 shows the results of comparing the only locality strategy to the locality strategy plus Recency(20), Recency(70) and Repetitiveness. From this table it is clear that there are statistically significant benefits to applying the recency or repetitiveness strategies after the locality strategy. Table 3 shows the results of comparing the recency strategy using only the 20 most recently typed lines compared to recency using 5, 10 and 70 lines and the repetitiveness strategy. The very small differences between mean values of recency(10), Recency(20) and Recency(70) could reveal an aspect of programming — in general, programmers tend to use a group of words/variables intensively over a very small part of a program. Therefore, considering words which are used within the larger scope of the previous 70 lines does not promote the accuracy of the prediction. In addition, the fact that the prediction accuracy for Recency(05) is significantly worse appears to indicates that one should not use too small a range for prediction. Due to the fact that there was no significant difference in the larger prediction scopes, Recency(20) was used throughout the rest of the experiment.

The performance of the recency strategy with respect to effectiveness and accuracy is slightly (but not significantly) better than that of the repetitiveness strategy in general. Again, this could be explained by the programming characteristic mentioned before; words are more localized than globalised.

Another issue which was tested was to determine how much difference the order of typing in the lines of the program would make to the prediction and key saving. This test was accomplished by reordering the lines of code in some of the test programs. An attempt was made to model a logical way of entering the code — for instance typing in the "stubs" for procedures and functions and then going back and filling in the details. Ten programs were changed in this way and processed using the simulator. In all cases there was no significant difference in either key savings or prediction accuracy. This appears to indicate that the prediction strategies are robust enough to be used in a real editor. Clearly this still needs to be tested.

A further issue which appeared to warrant investigation was whether the size of the input program affected the performance of the prediction strategies. To test this the programs in the test set were divided up into 4 groups — trivial (25 programs, 0 – 50 lines in length), small (25 programs, 51 – 150 lines in length), medium (25 programs, 151 – 450 lines in length) and large (14 programs, >450 lines in length). The results of this test are shown in Table 4. The data in this table show no clear trends and there are no really significant differences to what is measured over the entire test set (all 89 programs). It does, however, seem that larger programs can benefit more from key savings — this could possibly be explained by the fact that variables

Table 1. Results of comparing the four strategies

Strategy	% Keys Saved Variance	Probability	% Predictions Accepted Variance	Probability
None	32.9 0.061	—	43.1 0.071	—
Locality	34.3 0.064	0.060	46.5 0.083	0.018
Recency(20)	38.1 0.071	$\leq$ 0.001	56.1 0.076	$\leq$ 0.001
Recency(70)	38.3 0.072	$\leq$ 0.001	56.8 0.075	$\leq$ 0.001
Repetitiveness	37.8 0.071	$\leq$ 0.001	55.1 0.080	$\leq$ 0.001

Table 2. Results of comparing Locality to the other strategies

Strategy	% Keys Saved Variance	Probability	% Predictions Accepted Variance	Probability
Locality	34.3 0.064	—	46.5 0.083	—
Recency(20)	38.1 0.071	$\leq$ 0.001	56.1 0.076	$\leq$ 0.001
Recency(70)	38.3 0.072	$\leq$ 0.001	56.8 0.075	$\leq$ 0.001
Repetitiveness	37.8 0.071	$\leq$ 0.001	55.1 0.080	$\leq$ 0.001

Table 3. Results of comparing Recency, and Repetitiveness

Strategy	% Keys Saved Variance	Probability	% Predictions Accepted Variance	Probability
Recency(20)	38.1 0.071	—	56.1 0.076	—
Recency(05)	37.2 0.068	0.215	53.8 0.080	0.020
Recency(10)	37.7 0.070	0.380	55.2 0.077	0.215
Recency(70)	38.3 0.072	$\geq$ 0.400	56.8 0.075	0.275
Repetitiveness	37.8 0.071	0.380	55.1 0.080	0.185

Table 4. Results for programs of different lengths — using Recency(20)

Strategy	% Keys Saved Variance	Probability	% Predictions Accepted Variance	Probability
All programs	38.1 0.071	—	59.4 0.076	—
Trivial	34.8 0.070	0.185	55.3 0.076	0.310
Small	35.3 0.051	0.038	59.7 0.049	0.017
Medium	41.4 0.065	0.027	53.3 0.070	0.045
Large	42.5 0.066	0.017	56.5 0.102	0.045

are used more often, can thus be predicted later and once predicted then key savings can occur.

One further test was made in this section of the experiment. Nine of the programs from the test set were edited and the variable names in these programs were changed to be only two letters long and so that many of the variable names started with the same letter. Also many of the local variables were changed to be global variables. This simulates a "bad programming style". The null hypothesis in this case is that the strategies would make no difference. Table 5 presents the results of this test. These results seem to indicate that there is some benefit given to programmers using descriptive variables names and local variables.

Phase 2 — Optimality

The optimality hypothesis states that a system should be able to apply locality, recency and repetitiveness together to improve the results obtained from locality with recency or locality with repetitiveness. In this stage, three methods were applied to pursue the optimality of the outcome. The first method applied repetitiveness after recency (*rec+rep*) to reduce the conflicts between valid matches with maximum confidence value (obtained from locality and recency). Basically here the frequency count was used as a "tie-breaker" for words with the same prediction confidence value based on locality and recency. The second method is similar to rec+rep except that recency was applied as a tie-breaker after repetitiveness (*rep+rec*) if necessary. The third method (*both*) was to attempt to combine recency and frequency for the computation of the confidence value. This was done by simply adding the confidence values from the two strategies. This could result in a word which has occurred frequently but not recently being given a higher prediction confidence level that one which had appeared on the last line typed. Clearly there are cases when this would not be desirable. The results of this test are presented in Table 6. From this table, the data suggest that *both* is the best strategy, but none of the combined strategies appears to offer a statistically significant improvement over Recency(20). Since these predictions were done on programs whose structures are confined by the syntax of the language, the very subtle improvement *rec+rep* and *both* appears explainable. In a program there are normally a limited number of words on a line, and the chance of having two or more different words on the same line starting with the same character is slim. Therefore, in most of the predictions only one valid match has the maximum confidence value and, therefore, the recency strategy is often applied without the reinforcement of repetitiveness. Similarly, *rep+rec* normally applied repetitiveness alone. Occasionally, however, both strategies would have been applied. The data from Table 6 show no dramatic improvements and other methods should be sought to integrate these two strategies (recency and repetitiveness) to improve the results to a greater extent.

Phase 3 — Continuation

The continuation strategy is based on the premise that predictions done on similar programs, i.e. programs with very similar variables, sharing the same dictionary should have better results than on programs using their own dictionaries.

This strategy was tested by running the simulator on eleven programs developed by the same programmer for use on an image processing research project. Inspection of the programs shows that similar variable names do in fact occur in the programs (`Object`, `Image`, `size`, `percent`, `dim`, `value`, `found`, etc.). As the simulator executes it adds each word it encounters in processing a program to a common dictionary which is used for all later programs. The null hypothesis here is that there will be no difference in applying the prediction strategies without a common dictionary (standard processing) as compared with using a common dictionary. Table 7 presents the results of this experiment. There does seem to be some evidence of improved predictions using the shared dictionary but this is not highly statistically significant (possibly due to the small sample size). Inspection of the raw data shows that the use of the shared dictionary means that fewer predictions are made overall but that a greater percentage of the predictions made are accepted. This agrees with what would be expected. More often a word is matched on the first prediction as it is taken from the shared dictionary. Without the shared dictionary a word which had not appeared before in a program would have to "guessed at" a number of times before being added to the list of words found and made available for later predictions. Thus the shared dictionary means fewer predictions and a greater percentage of good predictions.

5 Future Work

This research concentrated on trying to save typing by using knowledge of the programmer's use of variables, types etc. and the target language. The simulator did not include generating correct parse trees reflecting the structure of the language — something which is done by language-directed editors. A full working editor should include text prediction for variables, types, etc. and parse tree generation. Such a system is still to be implemented but this research has proved its potential usefulness

Another area which should be investigated is that of programmer pattern prediction. For instance, being able to predict the pattern `var:=var op something` based on what the programmer has done before. This is a nontrivial problem which, if it can be solved, will offer enormous additional benefits to a disabled (or any) programmer. A related issue to consider would be to see whether applying semantic knowledge would improve accuracy. For example, if the types of the variables declared so far are known then it might be possible to restrict the predictions to those that would be type correct in the current context.

A different area which could be a useful avenue of re-

Table 5. Results of comparing "Bad style" using Recency(20)

Strategy	% Keys Saved Variance	Probability	% Predictions Accepted Variance	Probability
Good style	38.1 0.049	—	59.4 0.054	—
Bad style	32.2 0.058	0.015	54.4 0.074	0.065

Table 6. Results of comparing optimality strategies

Strategy	% Keys Saved Variance	Probability	% Predictions Accepted Variance	Probability
Recency(20)	38.1 0.071	—	56.1 0.076	—
rec+rep	38.4 0.074	0.380	56.9 0.074	0.240
rep+rec	38.0 0.072	$\geq$ 0.400	55.7 0.077	0.350
both	38.4 0.074	0.380	57.0 0.074	0.215

search would be to test the strategies presented in this paper in a different text generation environment — a windowing environment for example.

6 Conclusion

The results achieved here are very promising — the accuracy of the prediction of the next word to be typed can be dramatically improved if the information about when and how often a word is used is taken into account and combined with the syntax of the language. This research did not include the construct-based template-creation or tree-building methods of language-directed editors. Instead it concentrated on word prediction (variables, types, etc.) — something which the other systems do not tackle. Using the approaches suggested in the paper, prediction accuracies of between 43% and 59% and typing-savings of 32% to 40% can be expected. A system designed using these ideas could thus be of immense benefit to a motor-impaired programmer, allowing him/her to perform much more effectively and thus fulfill a more active role in society. The strategies discussed here do not have to be restricted to being implemented in a keyboard driven text based editor but could also be incorporated into a point-and-click environment. The important issue is that "accurate" prediction can save effort.

Acknowledgements: The authors would like to thank the referees and the editor of SACJ for their very helpful comments; Anjana Mistri for her work on the project and Scott Hazelhurst and Philip Machanick for proofreading the paper.

References

1. G E P Box, W G Hunter, and J S Hunter. *Statistics for Experimenters: An Introduction to Design, Data Analysis and Model Building*. Wiley-Interscience, 1978.
2. F J Budinsky, R C Holt, and S G Zaky. 'SRE— a syntax recognizing editor'. *Software—Practice and Experience*, **15**(5):489–497, (1985).
3. J Darragh. Adaptive predictive text generation and the reactive keyboard. Master's thesis, Computer Science Dept., University of Calgary, 1988.
4. J Darragh, I Witten, and M James. 'The reactive keyboard: A predictive typing aid'. *Computer*, **23**(11):41–48, (1990).
5. C N Fischer, G F Johnson, J Mauney, A Pal, and D L Stock. 'The Poe language-based editor project'. In *ACM SIGSOFT/SIGPLAN software engineering symposium on practical software development environment Proceedings*, pp. 21–30, (1984).
6. B S Lerner. 'Automated customization of structure editors'. *International Journal of Man-Machine Studies*, **37**:529–563, (1992).
7. R Mora and P Feiler. 'An incremental programming environment'. *IEEE Trans. Software Engineering*, **SE-7**:472–482, (1981).
8. J Preece. *Human-Computer Interaction*. Addison-Wesley, 1994.
9. T Teitelbaum and T Reps. 'The Cornell Program Synthesizer: A syntax-directed programming environment'. *Communications of the ACM*, **24**(9):563–573, (1981).
10. J Welsh, B Broom, and D Kiong. 'A design rationale for a language-based editor'. *Software—Practice and Experience*, **21**(9):923–948, (1991).

Table 7. Results of the continuation experiment

Strategy	% Predictions Accepted Variance	Probability
Standard processing	55.6 0.052	—
Shared Dictionary	58.5 0.057	0.115