Chapter 9

Selection Inference

Chapter 7 described three ways to make selections in LAPIS — using the mouse, choosing a named pattern from the library pane, and writing a TC pattern. This chapter[1] describes a fourth way — inferring selections from examples.

Two techniques are described:

- *Selection guessing* is the most general technique. It takes positive and negative examples from the user and infers a TC pattern consistent with the examples, which is then used to make the selection. At any time, the user can invoke an editing operation or a menu command on the current selection, start a fresh selection somewhere else, or tell the system to stop making inferences and add or remove selections manually with the mouse.
- *Simultaneous editing* is a form of selection guessing specialized for a common case in repetitive text editing: applying a sequence of edits to a group of text regions. Simultaneous editing is a two-step process. The user first selects a group of *records*, such as lines or paragraphs or postal addresses. This record selection can be made like any other selection — using the mouse, writing a TC pattern, or using selection guessing. Once the desired records have been selected, the system enters a mode in which inference is constrained to produce exactly one selection in every record. The constraints of simultaneous editing permit fast inference with few examples, so few in fact that simultaneous editing approaches the ideal of one-example inference.

This chapter is divided into three parts. The first section describes the user interface techniques of selection guessing and simultaneous editing. The second section details the implementation of these techniques, showing in particular how region set data structures (Chapter 4) enable substantial preprocessing and fast search for hypotheses. The last section describes two user studies, one of simultaneous editing and the other of selection guessing.

9.1 User Interface

To use selection inference, the user must switch into a different *selection mode*, which affects how mouse selections are interpreted. LAPIS has three selection modes:

[1] Portions of this chapter are adapted from earlier papers [MM01a, MM02].

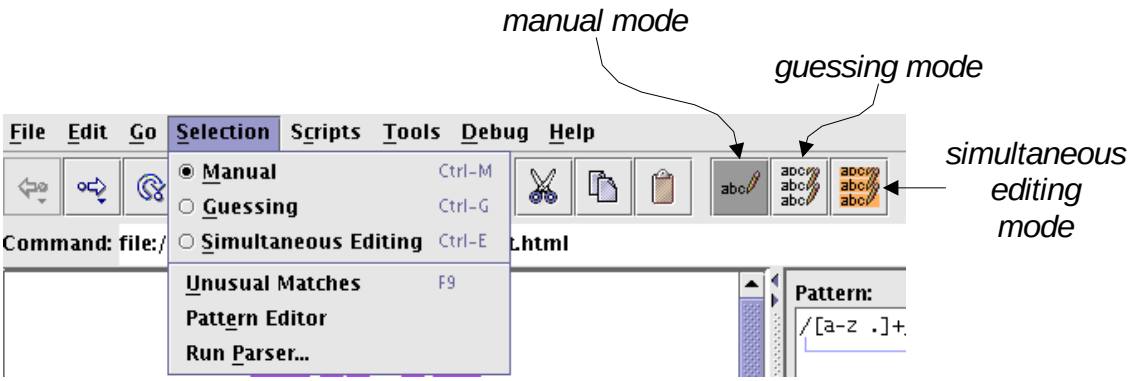


Figure 9.1: The selection mode can be changed with either the Selection menu or toolbar buttons.

- *Manual mode* is the default selection mode. In this mode, mouse selections add and remove regions from the current selection, and no inference is done. Manual selection mode was described in Section 7.4.1.
- *Guessing mode* is the simplest but least efficient inference mode. In this mode, the user's mouse selections are interpreted as positive and negative examples. Whenever the user provides a new example, the system infers a pattern and changes the selection to match its hypothesis.
- *Simultaneous editing mode* is a two-part mode. The first part behaves like guessing mode while the user selects the records. In the second part, the user's selections within the records are treated as positive examples for inference constrained to make exactly one selection per record.

The current selection mode can be changed either by the Selection menu or by a group of toolbar buttons. Both are shown in Figure 9.1.

The selection mode only affects how mouse selections are handled. All other features of LAPIS — editing commands, script commands, menu commands, pattern matching, etc. — can be invoked regardless of the current selection mode.

Introducing modes also introduces the danger of *mode errors*, i.e., performing an operation in the wrong mode. LAPIS tries to alleviate this problem somewhat by making the current selection mode prominent, displaying a dialog pane on the right side of the window (e.g., in Figure 9.2). In simultaneous editing mode, the record set is highlighted in yellow, as a further visual cue to the current mode. These techniques have not been sufficient to eliminate all mode errors, however. Users in the user study occasionally tried to give examples without switching into inference mode. In my own use of LAPIS, I occasionally forget to switch out of inference mode before trying to make a single selection. Selection feedback makes mode errors quickly apparent, however.

One solution to mode errors would be a *spring-loaded* mode, such as a modifier key. For example, holding down the Alt key while making a mouse selection might indicate that the selected region should be used as an example, triggering inference on that selection. Modifier keys offer less visible affordances than a toolbar mode, however, and manual mouse selection already uses several modifier keys (Control and Shift) that might easily be confused with the inference modifier key. Evaluating these tradeoffs on users is left for future work.

9.1.1 Selection Guessing Mode

When the user enters selection guessing mode, a dialog pane appears on the right side of the window (Figure 9.2(a)). The pane contains a help message and a small set of controls. Originally, this dialog was popped up as a modeless dialog box window which floated over the LAPIS window, but the floating window had two problems. First, users found that the window obscured their work, and often felt the need to move or resize it to see what was underneath. Second, some users assumed that the dialog box was modal, and hence had to be dismissed before they could interact with the main LAPIS window again. Both problems were solved by moving the dialog into the LAPIS window, at the cost of obscuring the library pane. The library pane is not usually needed during inference. For the occasional cases when it is, a future version of LAPIS may merely shrink it instead of hiding it completely, using a technique like Mozilla's Sidebar.

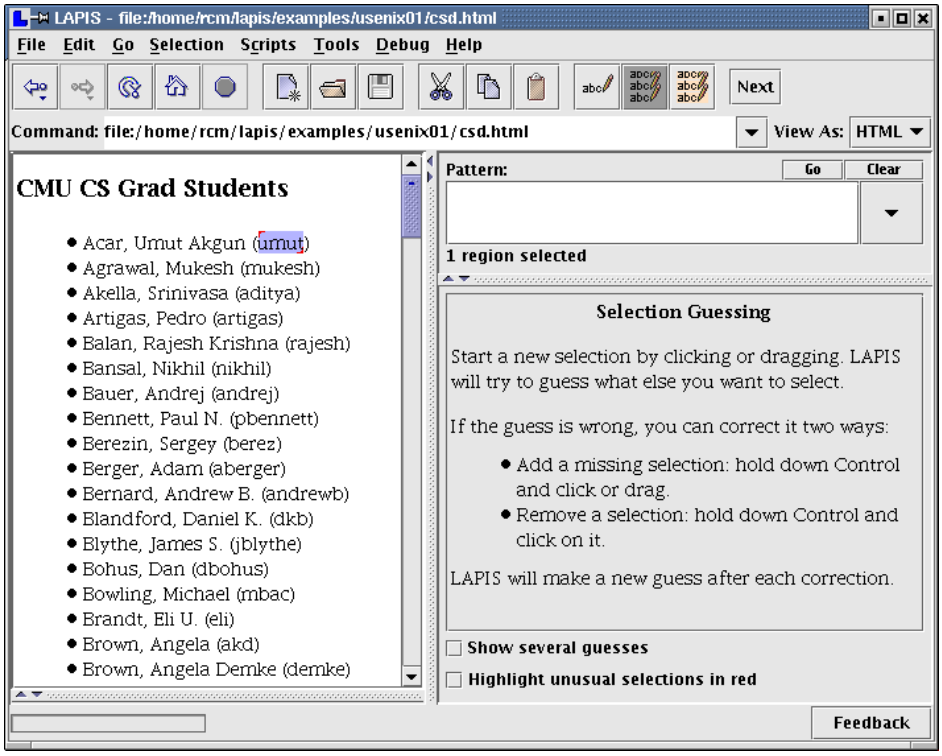
In selection guessing mode, when the user starts a new selection by clicking or dragging, the system uses the selected region as a positive example of the desired selection and infers a TC pattern that is consistent with it. The inferred pattern is displayed both as additional selections in the browser pane, and as a pattern in the pattern pane. Figure 9.2 illustrates this process.

If the inferred selection is wrong, the user can correct it in two ways. The first way is by giving additional examples. Additional positive examples are given by adding regions to the selection — holding down Control while clicking or dragging. Negative examples are given by removing regions from the selection — holding down Control and clicking on the selected region. After each example, the system updates its hypothesis to account for all the positive and negative examples given so far.

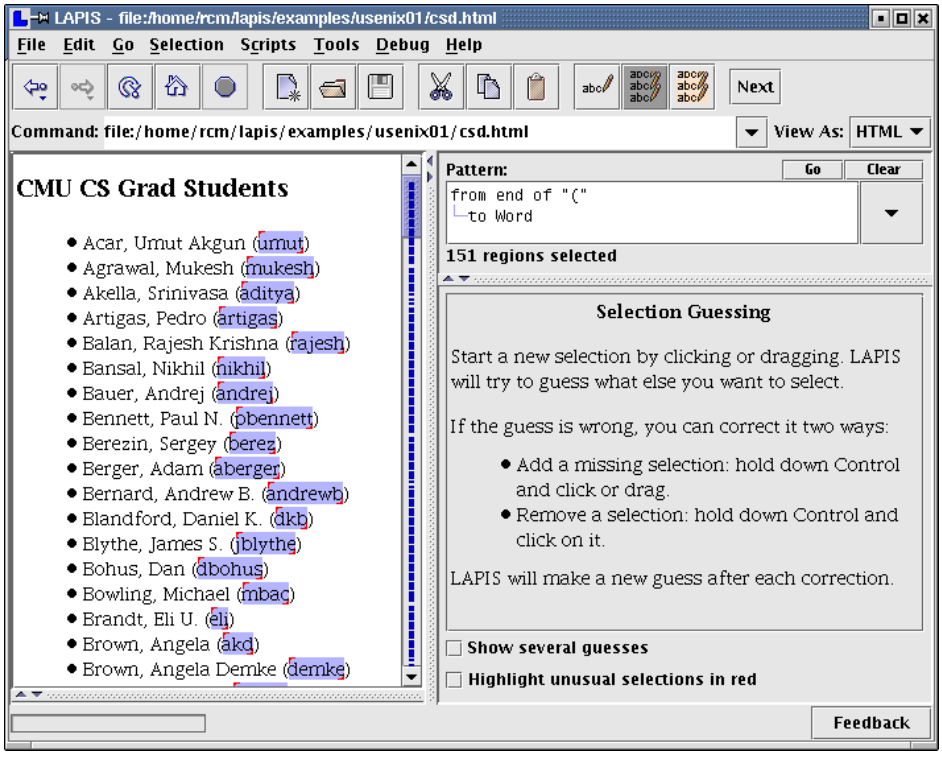
In the first design of selection guessing, positive and negative examples were highlighted in various colors, to help the user keep track of which examples had been given. Positive examples were colored dark blue, in order to stand out against the light blue inferred selections, and negative examples were pink. When example highlighting was tested in the user study, however, most users didn't understand what the different colors of blue and pink meant, so the feedback was largely useless. In retrospect, example highlighting seems largely unnecessary. Negative examples are irrelevant once they've been removed from the selection, and it seems unnecessary to recall precisely which selections were positive examples. In any case, users in the study generally gave only a few examples to correct a selection before giving up and trying to make the selection another way. Little can be gained from highlighting a few examples in a special way, aside from visual confusion. Positive and negative example highlighting was subsequently removed from LAPIS.

The second way to correct a selection is to choose an alternative hypothesis. When "Show several guesses" is checked in the dialog pane, the help message is replaced by a list of hypotheses that are consistent with the user's examples (Figure 9.3). Each hypothesis is described by a TC pattern, along with the number of regions it would select and a ranking score, which is described in more detail later in this chapter. By default, the system chooses the highest-ranked hypothesis as its guess. The user can click on any hypothesis in the list to switch to it and see its selection in the browser pane. The hypothesis list does not include *all* possible hypotheses consistent with the user's examples, however. Additional examples may be needed to constrain the hypothesis space sufficiently.

The hypothesis list also includes a "manual selection" choice, which inhibits inference and treats the user's last example as a manual correction on the previous selection. This feature was motivated by user study observations. In principle, if the desired selection lies in the space of



(a) user selects an example, "umut", with the mouse



(b) system infers a pattern and selects all matches to it

Figure 9.2: Making a selection by example in selection guessing mode.

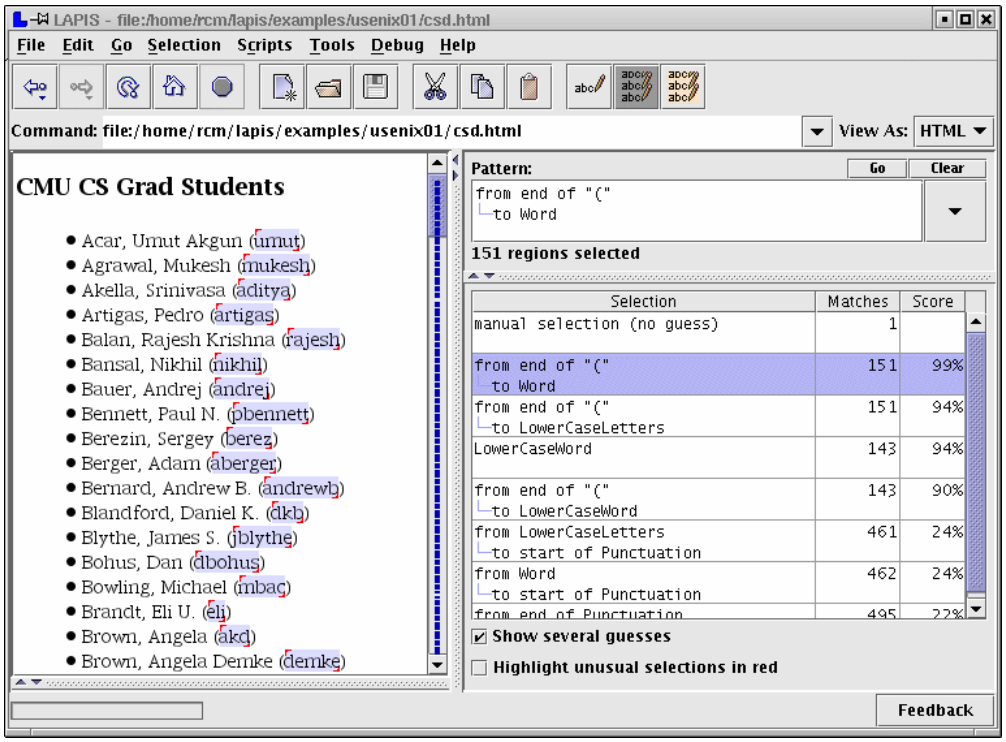


Figure 9.3: Selection guessing mode also offers a list of alternative hypotheses.

learnable hypotheses, then manual selection should be unnecessary, since the user's corrections would eventually converge on the right answer. In practice, however, users have no way to predict how many examples might be needed to learn the desired selection, or whether the system can learn it at all. As a result, as soon as an almost-correct hypothesis appeared, users expressed a desire to make the system stop guessing and let them fix the exceptions manually.

If the desired selection is outside the system's hypothesis space, inference will eventually fail to find a hypothesis consistent with the examples. When inference fails, the system stops updating its hypothesis and displays "unknown selection" as the pattern. The user can continue adding and removing regions from the selection manually. If inference failed because an incorrect example was given, the user can retract the example — e.g., if an incorrect positive example was given, the user gives it as a negative example — and the system will resume trying to make inferences.

Once the desired selection has been made, the user can edit or apply text-processing tools to it, as described in previous chapters. While the user is typing or deleting characters using an inferred selection, no inference is done. When the user starts a new selection with the mouse, the set of examples is cleared and the system generates a fresh hypothesis.

Inferred patterns can be edited by the user and run again. Once an inferred pattern has been edited, however, the system stops doing inference and throws away the user's examples, since the edited pattern may no longer lie in the hypothesis space.

The selection guessing dialog pane includes another checkbox, "Highlight unusual selections in red." When this box is checked, the outliers of the inferred selection are highlighted in red. Outlier highlighting is discussed in detail in Chapter 10.

9.1.2 Simultaneous Editing Mode

Many repetitive editing tasks have a common form: a group of things all need to be changed in the same way. Some examples from the PBD literature include:

- add "[author year]" to bibliographic citations [Mau94]
- reformat baseball scores [Nix85]
- change the styles of all section headings [Mye93]

Each of these tasks can be represented as an iteration over a list of text regions, called *records* for lack of a better name, where the body of the loop performs a fixed sequence of edits on each record. LAPIS addresses this class of tasks with simultaneous editing mode.

The first step in simultaneous editing is describing the record set, which is done by selecting all the records. When the user enters simultaneous editing mode, LAPIS displays a dialog pane (Figure 9.4) which prompts the user to select the records. The user may use the mouse to give positive and negative examples of records, using the same interaction techniques as selection guessing. The user may also enter a pattern to select the records. An experienced user can shortcut this dialog by selecting the records before entering simultaneous editing mode. If at least two nonzero-length regions are selected when the user clicks on the Simultaneous Editing menu command or toolbar button, then this selection is used as the record set.

Once the desired record set is selected, the user clicks on the Start Editing button in the dialog pane to enter the second step of simultaneous editing mode. The selected record set becomes highlighted in yellow, and the dialog pane changes to describe the new mode (Figure 9.5). Now, when the user makes a selection in one record, the system automatically infers exactly one selection in every other record. If the inference is incorrect on some record, the user can correct it by holding down the Control key and making the correct selection, after which the system generates a new hypothesis consistent with the new example. As in selection guessing, the user can edit with the multiple selection at any time. The user is also free to make selections outside the yellow records, but no inferences are made from those selections.

Figures 9.6–9.11 illustrate how simultaneous editing can be used to perform a common task in Java and C++ programming: replacing each public field of a class (member variable in C++ terminology) with a private field and a pair of accessor methods `getX` and `setX` that respectively get and set the value of the field.

In Figure 9.6, the user has selected "public" in one record, which causes the system to infer a selection of "public" in the other records as well. The pattern pane displays the TC pattern that matches this inference, `"public"`.

The user deletes this selection, then places the insertion point at the end of the records to start typing the `get` method: first pressing Enter to insert a new line, then indenting a few spaces, then typing "public" to start the method declaration. Following multiple-selection editing rules, the typed characters appear in every record (Figure 9.7). If typos are made, the user can back up and correct them, using all familiar editing operations, including Undo. No inference occurs while the user is typing.

Now the user is ready to enter the return type of the `get` method. The type is different for each variable, so it can't simply be entered at the keyboard. Instead, copy-and-paste must be used.

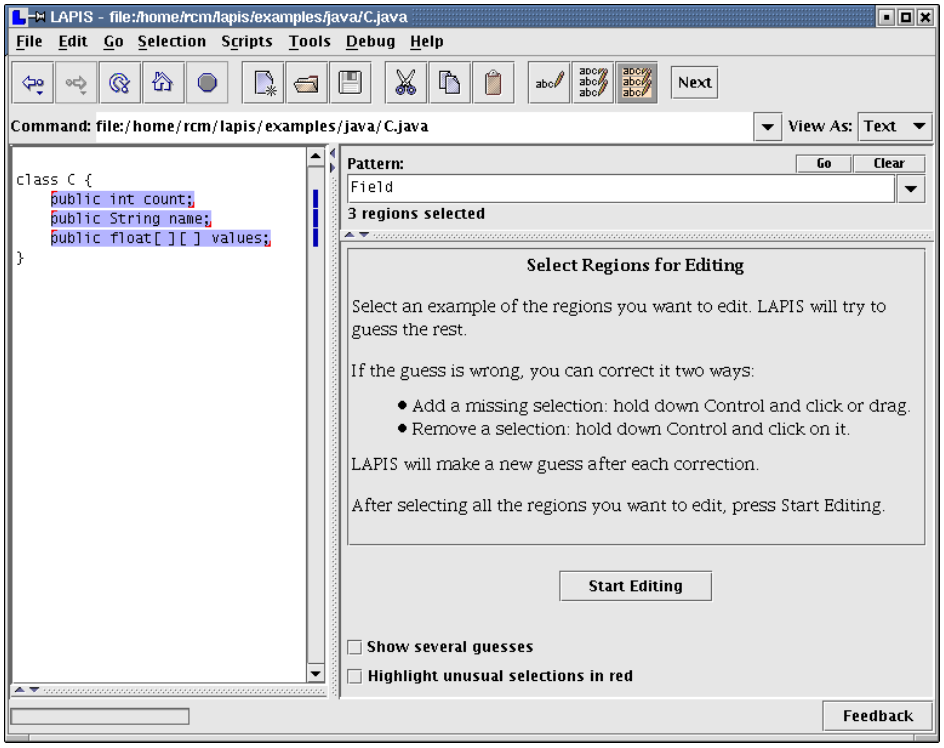


Figure 9.4: First step of simultaneous editing: selecting the records using unconstrained inference (selection guessing).

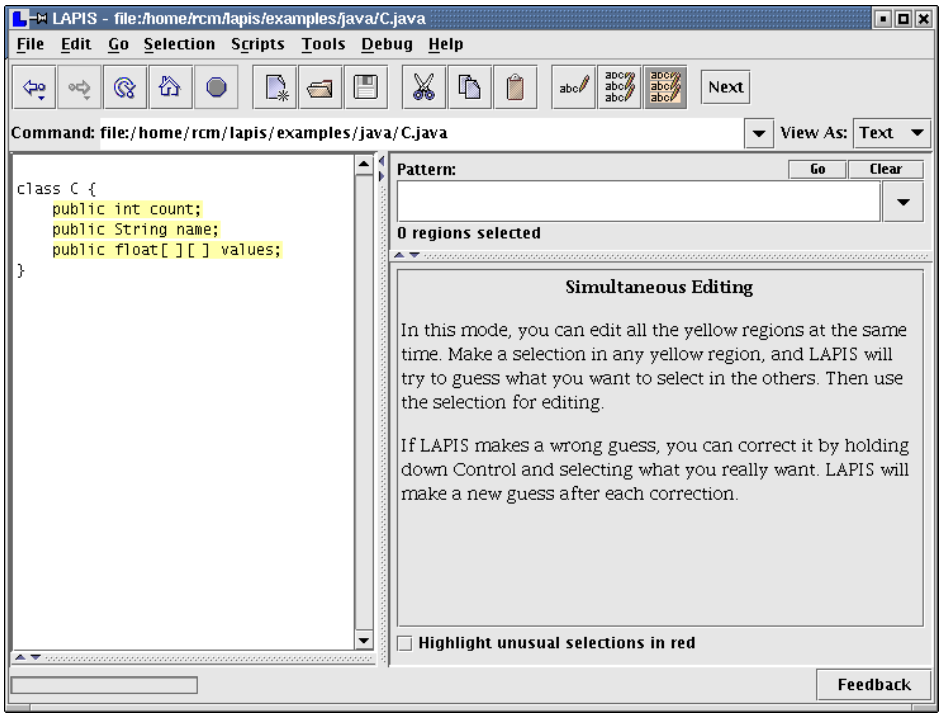


Figure 9.5: Second step of simultaneous editing: the records turn yellow, and inference is now constrained to make one selection in each record.

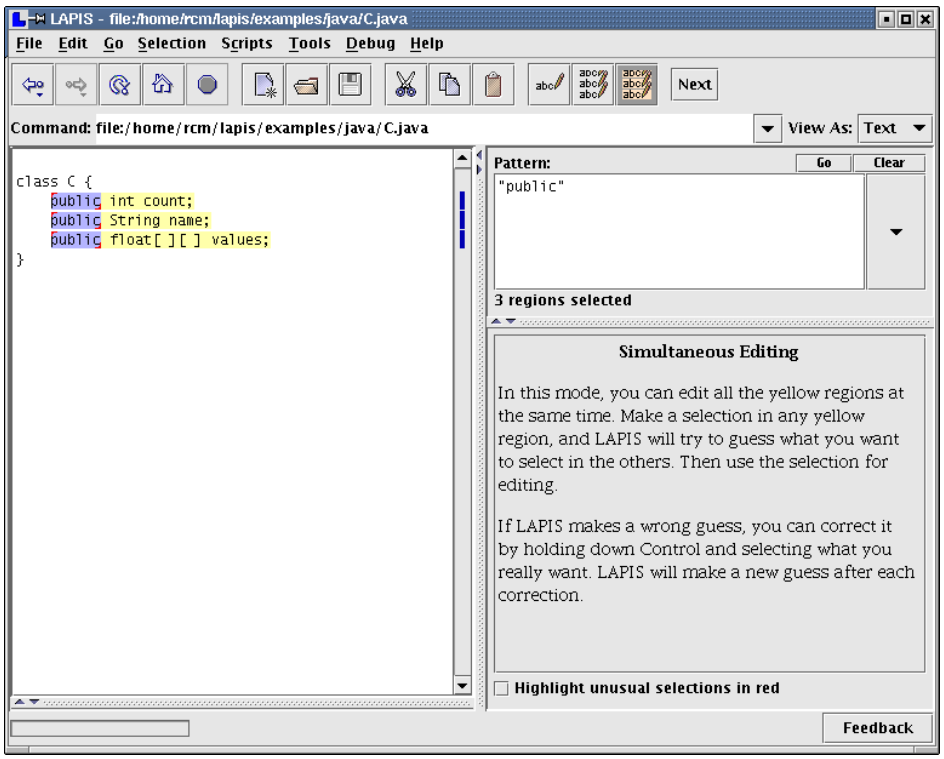


Figure 9.6: Selecting “public” in one record causes the system to infer equivalent selections in the other records.

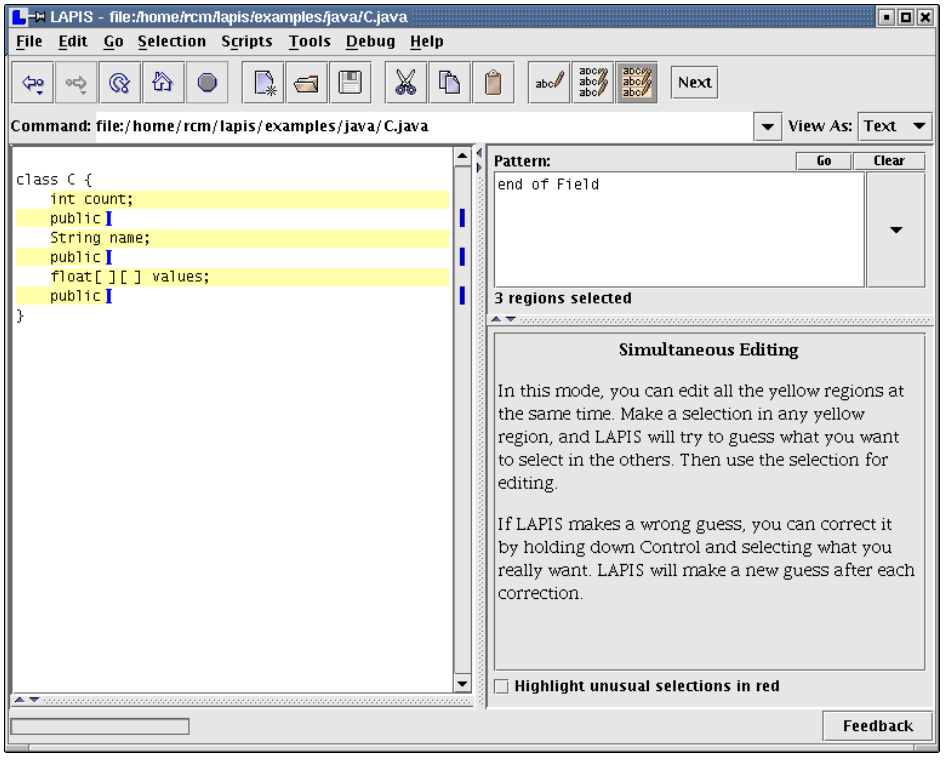


Figure 9.7: Typing and deleting characters affects all regions.

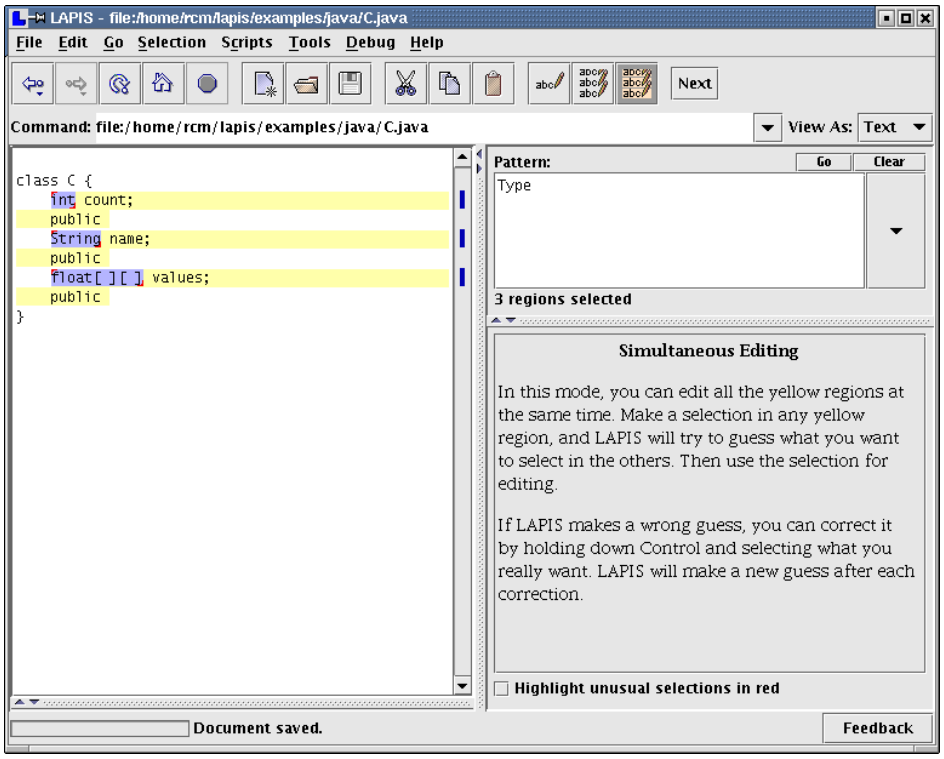


Figure 9.8: The user selects the types and copies them to the clipboard...

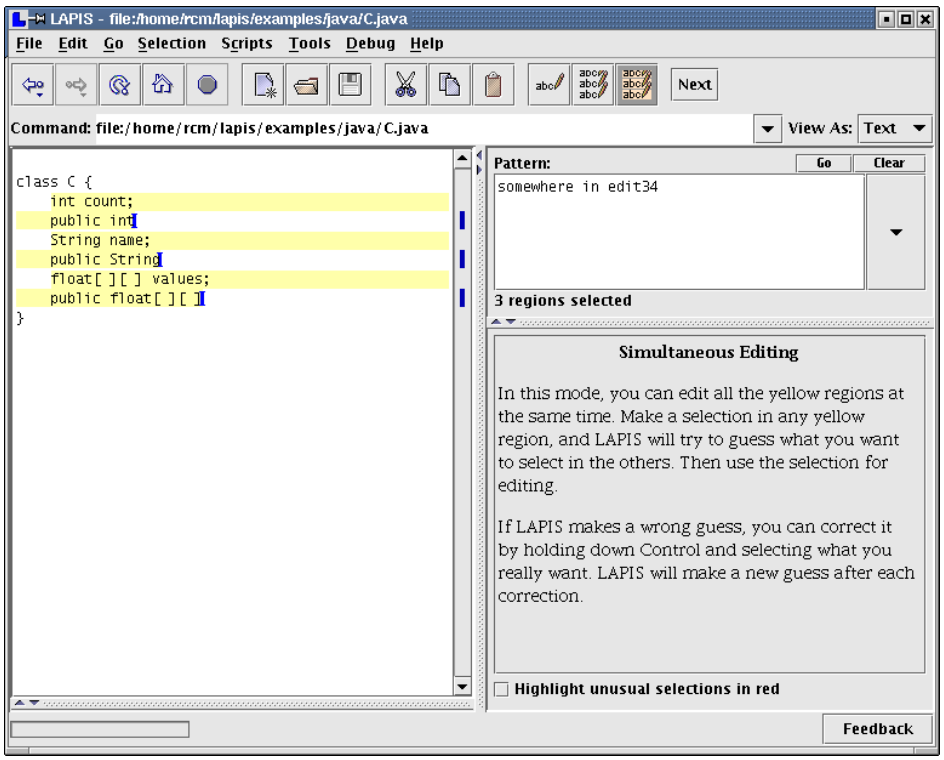


Figure 9.9: ... and then pastes the copied selections back.

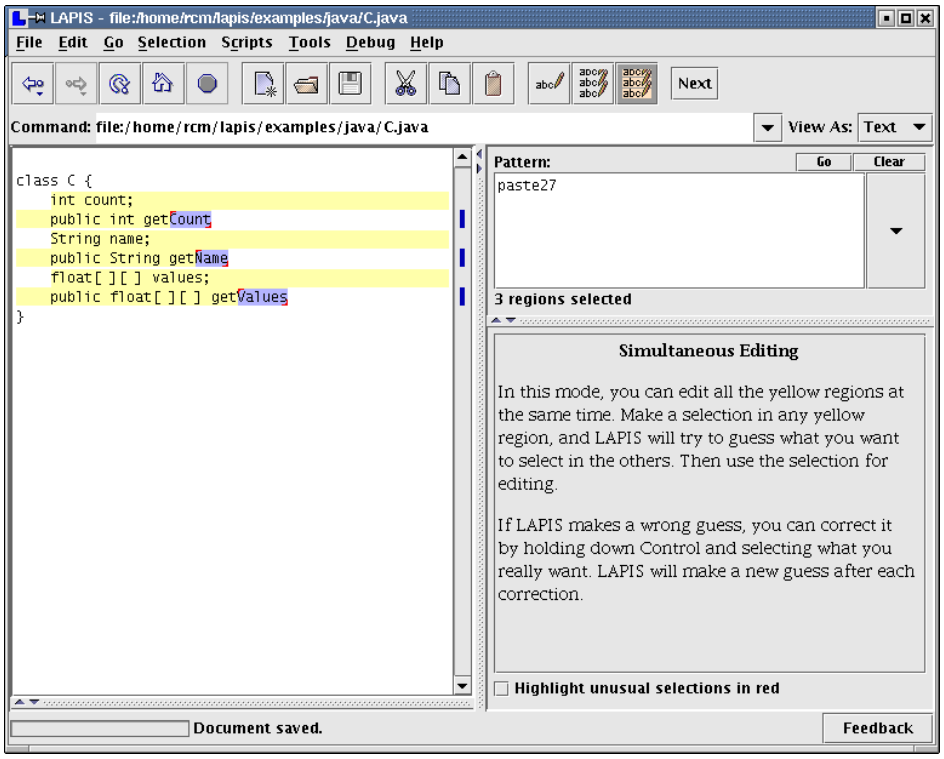


Figure 9.10: Changing the capitalization of the method name.

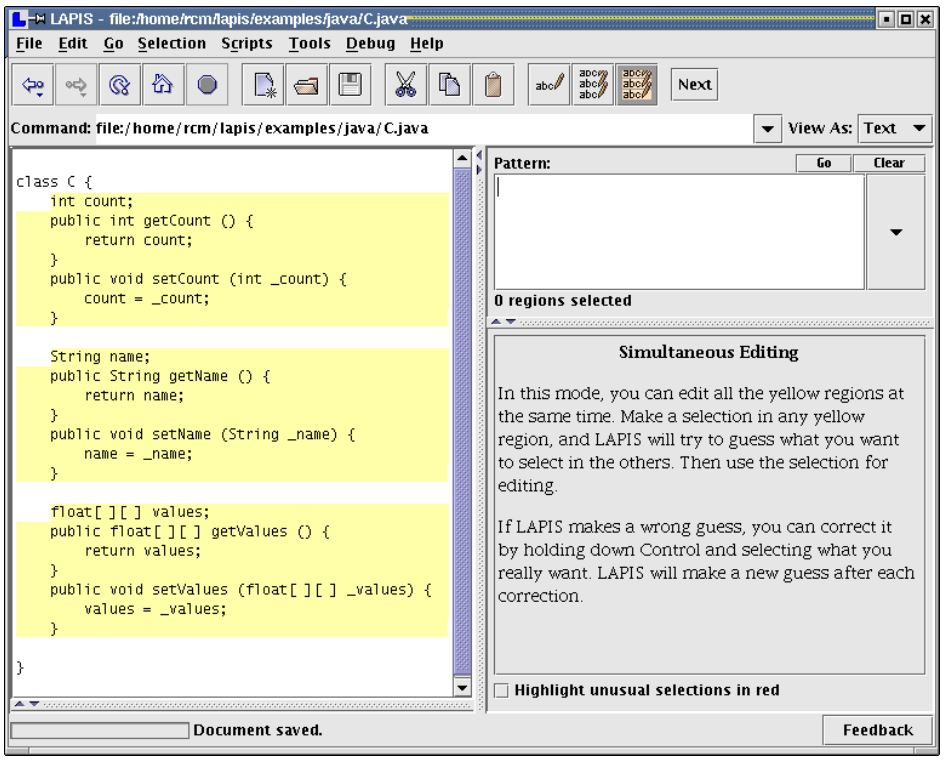


Figure 9.11: The final outcome of simultaneous editing.

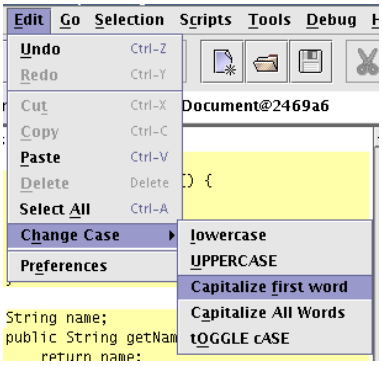


Figure 9.12: Commands on the Change Case menu change the capitalization of the selection.

The user first selects the type of one of the records, such as the “int” in “int x”. The system infers the pattern `Type` and selects the types in the other fields (Figure 9.8). This shows an important difference between simultaneous editing and selection guessing. In selection guessing, many other hypotheses would also be consistent with the user’s example: `"int"`, `2nd Word in Line`, `2nd from last Word in Line`, etc. In simultaneous editing, some of these hypotheses can be discarded immediately because they do not make exactly one selection in each record. For example, the literal string `"int"` does not appear in every record. Other generalizations are less preferable because they are more complicated than `Type`.

The user then copies the selection to the clipboard, places the insertion point back after “public”, and pastes the clipboard. Following multiple-selection editing rules, the system copies a list of strings to the clipboard, one for each record, and pastes the appropriate string back to each record (Figure 9.9). The one-selection-per-record bias helps here, too. Since the number of inferred selections always equals the number of records, there is no danger of trying to paste n copied selections back to m insertion points (which would cause LAPIS to pop up an error dialog, as explained in Section 7.5).

Note that the inference in Figure 9.9 is described as `somewhere in edit34`, which is not a valid TC pattern. This is another difference between selection guessing and simultaneous editing. In selection guessing, all inferences are TC patterns. In simultaneous editing, however, when the user makes a selection in new text — text that has been typed or pasted since entering simultaneous editing mode — LAPIS does not bother to search for a TC pattern describing the selection. Instead, it simply infers the selection that originally created the text, and displays an approximate description of the selection in the pattern pane. This technique has significant performance advantages — in particular, it eliminates the need to continually reparse the document as the user is editing, since new text need not be parsed. The tradeoff is that some inferences do not produce valid TC patterns, which is not ideal for self-disclosure of the TC pattern language. Fast response time seems to outweigh the value of self-disclosure, at least in this case.

Next, the user copies and pastes the name of the variable to create the method name (Figure 9.10). The lowercase variable name must be capitalized by applying a menu command that capitalizes the current selection (Figure 9.12). Any menu command that applies to a selection can be used in simultaneous editing mode.

The rest of the `get` and `set` methods are defined by more typing and copy-and-paste commands, until the final result is achieved (Figure 9.11). The user then switches back to manual selection mode, and the yellow record highlighting disappears.

Simultaneous editing is more constrained than selection guessing, because its hypotheses must have exactly one match in every record. But the one-selection-per-record bias delivers some powerful benefits. First, it dramatically reduces the hypothesis search space, so that fewer examples are needed to reach the desired selection. Second, the hypothesis search is faster. Since the record set is specified in advance, LAPIS can preprocess it to find common substrings and library patterns that occur at least once, significantly reducing the space of features that can be used in hypotheses. As a result, where selection guessing might take several seconds to deliver a hypothesis, simultaneous editing takes 0.4-0.8 sec, making it more suitable for interactive editing. Finally, the one-selection-per-record constraint makes editing semantically identical to single-selection editing on each record. In particular, a selection copied from one place can always be pasted somewhere else in the record, since the source and target are guaranteed to have the same number of selections.

The keyboard can also be used to make selections in simultaneous editing mode. When the user presses an arrow key, the system first clears most of the current selection, leaving only the first example selected. This selection is then moved by the arrow key in the same way that a conventional single-selection text editor would move it. The resulting selection is then treated as an example to infer a new selection.

The practical result of this behavior is that the user can move the selection with the keyboard while inference keeps the selections synchronized. For example, suppose the user places the cursor before “int” in Figure 9.5, causing the system to infer a selection `start of Type`. Pressing the right arrow key three times moves the initial example to the end of “int”, causing the system to infer the selection `end of Type`. A more naive approach to multi-cursor keyboard navigation would move all the cursors three characters to the right in lock step — leaving one in the middle of “String” and the other in the middle of “float”.

Most conventional keyboard navigation is supported, including Home, End, Page Up, Page Down, holding down Control to move by whole words, and holding down Shift to select a region. The keyboard cannot currently be used to give multiple examples; doing so would require decoupling the keyboard cursor from the selection, so that it can be moved around independently to add and remove selections.

9.2 Implementation

This section describes the algorithms used to infer selections from examples. Each mode, selection guessing and simultaneous editing, uses a different algorithm. Like most learning algorithms, both algorithms essentially search through a space of hypotheses for a hypothesis consistent with the examples, but each algorithm uses a different hypothesis space and a different method of ranking hypotheses, reflecting the different biases of the two modes.

In addition, the simultaneous editing algorithm does considerably more preprocessing than the selection guessing algorithm. Preprocessing allows much of the hypothesis space to be pruned away before the user even starts giving examples, which allows each selection to be inferred faster. The tradeoff is that the preprocessed information falls out of date as soon as the user edits the document, so some extra work has to be done to refresh it. Furthermore, because of preprocessing,

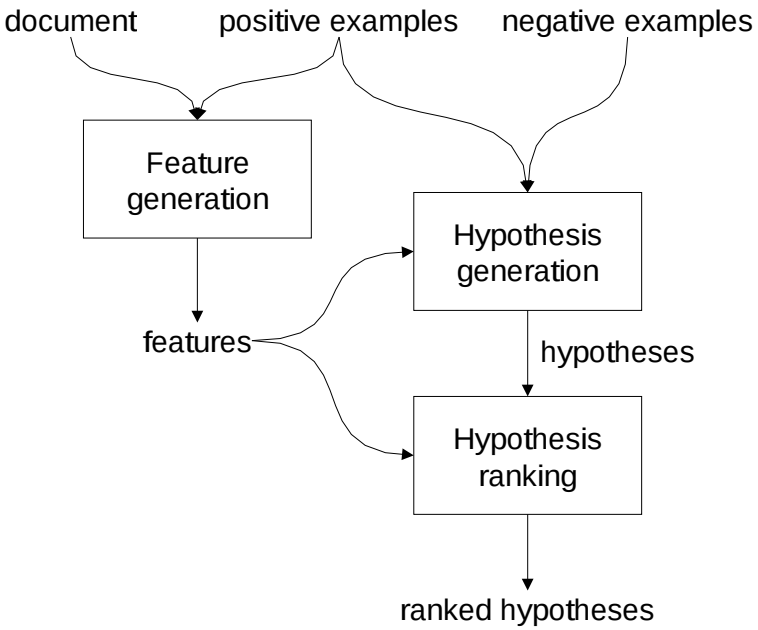


Figure 9.13: Block diagram of selection-guessing algorithm.

simultaneous editing does not always infer a TC pattern; sometimes its inference is simply a region set with no corresponding TC pattern. More will be said about this issue below.

9.2.1 Selection Guessing Algorithm

The inputs to the selection guessing algorithm are a document, a set of positive examples expressed as a region set, and a set of negative examples, also a region set. The output is a ranked list of hypotheses, each of which is a TC pattern that matches the positive examples and excludes the negative examples.

In this algorithm, a *feature* is a TC expression of the form *op F*, where *op* is one of the TC operators `equals`, `just before`, `just after`, `starting`, `ending`, `in`, or `contains`, and *F* is either a literal or a pattern identifier. A *hypothesis* is a Boolean conjunction of features.

The algorithm has three main parts, illustrated in Figure 9.13:

- *Feature generation* takes the document and the positive examples and produces a collection of features that match all the positive examples.
- *Hypothesis generation* takes the features and the positive examples and returns a list of hypotheses that match all the positive examples and exclude the negative examples.
- *Hypothesis ranking* takes the hypotheses and ranks them by a heuristic.

Each part is described below.

Feature Generation

Two kinds of features are generated: *library features* derived from the pattern library, and *literal features* discovered by examining the text of the positive examples.

Library features are generated by searching for every named pattern in the pattern library, to produce the set of regions matching each pattern. This is actually the most time-consuming part of the inference algorithm. Then, the seven relational operators `equals`, `just before`, `just after`, `starting`, `ending`, `in`, and `contains` are applied to each library region set, producing a region set representing all the regions that have the feature. Since these relations are represented by rectangle collections, generating these region sets is fast (Chapter 4). The region set for each feature is then intersected with the region set representing the positive examples. Features which don't match all the positive examples are discarded. Since the algorithm only learns monotone conjunctions of features, only features that match all the positive examples are useful for forming hypotheses.

Literal features are generated by combining the relational operators with literal strings. Instead of the generate-and-test approach used for the library features, however, the generation of literal features is driven by the text of the positive examples. Since useful features must match all the positive examples, the generation algorithm searches for *common substrings*, i.e., substrings that occur in or around all the positive examples. The generation technique for each relational operator is slightly different. In the description that follows, the ith positive example is denoted by the region $x_i[y_i]z_i$, where the whole document is the string $x_i y_i z_i$ and y_i is the part of the document selected by the region.

- `equals`: If all y_i are identical, i.e. $y = y_i$ for all i, then generate the feature `equals "y"`.
- `starting`: Find the common prefixes of all the y_i, i.e., all strings p such that p is a prefix of every y_i. If two prefixes p and pq are equivalent in the sense that every occurrence of p in the document is followed by q, then keep only the shorter prefix p. For example, "http" is equivalent to "http://" if every occurrence of "http" in the document is followed by "://". For all remaining prefixes p, generate the feature `starting "p"`.
- `ending`: Generate the common suffixes of the y_i, using an algorithm analogous to `starting`.
- `just before`: Generate the common prefixes of the z_i.
- `just after`: Generate the common suffixes of the x_i.
- `contains`: Find all common substrings of the y_i using a *suffix tree* [Gus97]. A suffix tree is a path-compressed trie into which all suffixes of a string have been inserted. In this case, the suffix tree represents the set of common substrings of all y_i examined so far. Figure 9.14 illustrates the algorithm. The algorithm starts by building a suffix tree from y_1. (y_1 may be any positive example, but it's useful to let y_1 be the shortest y_i to minimize the size of the initial suffix tree.) For each remaining y_i ($2 \leq i \leq n$), the suffixes of y_i are matched against the suffix tree one by one. Each tree node keeps a count of the number of times it was visited during the processing of y_i. This count represents the number of occurrences in y_i of the substring represented by the node. After processing y_i, all unvisited nodes are pruned from

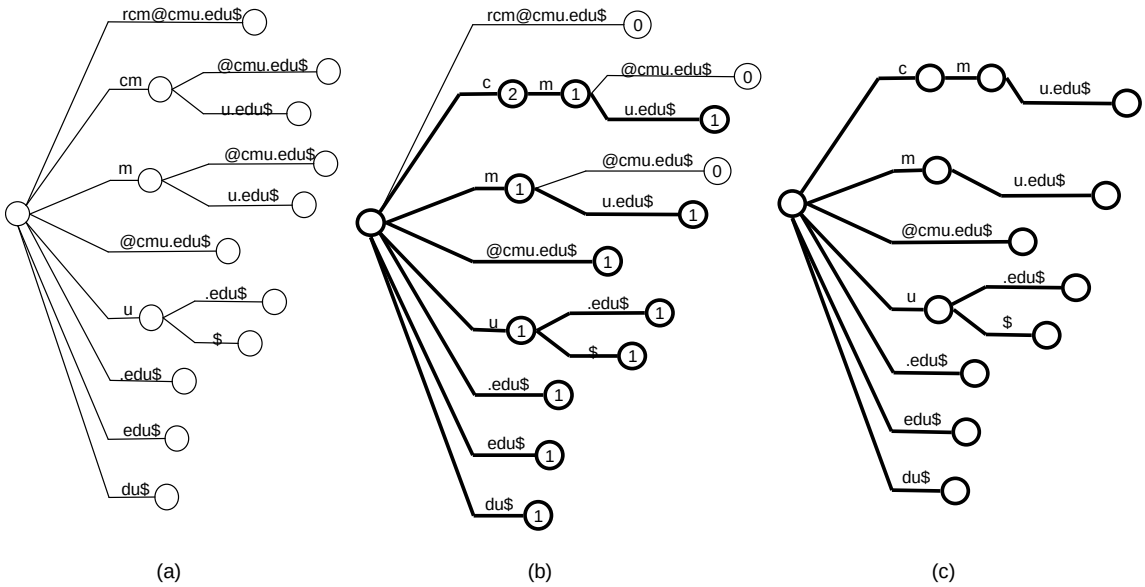


Figure 9.14: Finding common substrings using a suffix tree. (a) Suffix tree constructed from first example, `rcm@cmu.edu`; $ represents a unique end-of-string character. (b) Suffix tree after matching against second example, `ljc@cmu.edu`. Each node is labeled by its visit count. (c) Suffix tree after pruning nodes which are not found in `ljc@cmu.edu`. The remaining nodes represent the common substrings of the two examples.

the tree, since the corresponding substrings never occurred in y_i. After processing all y_i in this fashion, the only substrings left in the suffix tree must be common to all the y_i. Generate a feature `contains "s"` for each such substring s.

- `in`: No literal features of this form are generated, because `in "literal"` is usually redundant with an `equals` feature.

Hypothesis Generation

After generating features that match the positive examples, the algorithm forms conjunctions of features to produce hypotheses consistent with all the examples. Since a selection must have a clearly defined start point and end point, not all conjunctions of features are useful hypotheses, so the algorithm reduces the search space by forming *kernel hypotheses.* A kernel hypothesis takes one of the following forms:

- a single feature which fixes both the start and end, i.e. `equals` *F*. As a kernel hypothesis, this feature is represented simply by the pattern *F*.
- a conjunction of a start-point feature (`starting F` or `just after F`) with an end-point feature (`ending G` or `just before G`). As a kernel hypothesis, this conjunction is represented as a TC pattern of the form

```
from start/end of F
  to start/end of G
```

All possible kernel hypotheses are generated from the feature set. Kernel hypotheses that are inconsistent with the positive examples are then discarded.

If there are negative examples, then additional features are added to each kernel hypothesis to exclude them. Features are chosen greedily to exclude as many negative examples as possible. For example, after excluding negative examples, a kernel hypothesis `Link` might become the final hypothesis

```
Link contains "cmu.edu"
     just-before Linebreak
```

Kernel hypotheses that cannot be specialized to exclude all the negative examples are discarded.

This simple algorithm is capable of learning only monotone conjunctions. This is not as great a limitation as it might seem, because many of the concepts in the pattern library incorporate disjunction (e.g. `UppercaseLetters` vs. `Letters` vs. `Alphanumeric`). It is easy to imagine augmenting or replacing this simple learner with a disjunctive normal form learner, such as the one used by Cima [Mau94].

Hypothesis Ranking

After generating a set of hypotheses consistent with all the examples, there remains the problem of choosing the best hypothesis — in other words, defining the *preference bias* of the learner. Most learning algorithms use Occam's Razor, preferring the hypothesis with the smallest description, in this case, the fewest number of features in its conjunction. Since hypotheses can include library features, however, many hypotheses seem equally simple. Which of these hypotheses should be preferred: `Word`, `JavaIdentifier`, or `JavaExpression`? The algorithm supplements Occam's Razor with a heuristic I call *regularity*.

The regularity heuristic was originally designed for inferring record sets for simultaneous editing. It is based on the observation that simultaneous editing records often contain *regular features*, features that occur a fixed number of times in each record. For instance, most postal addresses contain exactly one postal code and exactly three lines. Most HTML links have exactly one start tag, one end tag, and one URL.

It is easy to find features that occur a regular number of times in all the positive examples. Not all of these features may be regular in the entire record set, however, so the algorithm finds a set of *likely regular features* by the following procedure. For each feature `contains` *`F`* that is regular in the positive examples (i.e., where F occurs exactly n_F times in each positive example), count the number of times D_F that F occurs in the entire document. Assuming for the moment that F is a regular feature that occurs *only* in records, then there must be D_F/n_F records in the entire document. Call D_F/n_F the *record count prediction* made by F.

Let M be the record count predicted by the *most* features, in other words, the mode of the record count predictions for all features F. If M is unique and integral, then the *likely regular features* are the features that predicted M. If M is not unique, or M is not an integer, then the algorithm abandons the regularity heuristic, and falls back to Occam's Razor, ranking hypotheses by the number of features.

The upshot of this procedure is that a feature is kept as a likely regular feature only if other features predict exactly the same number of records. Features which are nonregular, occurring fewer times or more times in some records, will usually predict a fractional number of records and be excluded from the set of likely regular features.

For example, suppose the user is trying to select the peoples' names and userids in the list below, and has given the first two items as examples (shown underlined):

> Acar, Umut (umut)
> Agrawal, Mukesh (mukesh)
> Balan, Rajesh Krishna (rajesh)
> Bauer, Andrej (andrej)

The two examples have several regular features in common, among them:

- `","` (comma), which occurs exactly once in each example;
- `CapitalizedWord`, which occurs twice;
- `Word`, which occurs 3 times;
- `Parentheses`, which occurs once.

Note that `CapitalizedWord` and `Word` are regular only in the two examples, not in the entire set of names.

Computing the record count prediction N_F/n_F for these features gives:

- comma: 4
- `CapitalizedWord`: 4.5
- `Word`: 4.33
- `Parentheses`: 4

The record count predicted by the most features is 4, so the likely regular features would be comma and `Parentheses`. This example is oversimplified, since the pattern library would find other features as well.

In addition to features of the form `contains` *F*, which look for regularity inside the examples, the algorithm also checks for regularity in features describing the context of the examples — before, after, and around.. The features `just before` *F* and `just after` *F* are considered regular if every positive example contains the same number of occurrences of *F* (just like `contains` *F*). The feature `in` *F* is considered regular if every positive example is in a different instance of *F*.

Likely regular features are used to test hypotheses by assigning a higher preference score to a hypothesis if it is in greater agreement with the likely regular features. A useful measure of

agreement is the *category utility*, which was also used in Cima [Mau94]. The category utility of a hypothesis H and a feature F is given by

$$\begin{aligned} U(H,F) &= P(H|F)P(F|H) \\ &= \frac{P(H \cap F)^2}{P(H)P(F)} \end{aligned}$$

where $P(F)$ is the probability of having feature F, $P(H)$ is the probability of satisfying the hypothesis H, and $P(H \cap F)$ is the probability of satisfying both the feature and the hypothesis. If a hypothesis and a feature are in perfect agreement, then the category utility is 1; if they are logical complements, then the category utility is 0. The probabilities can be estimated by drawing a sample of size N and counting the number of instances that satisfy H, F, or both:

$$\begin{aligned} U(H,F) &= \frac{(N_{HF}/N)^2}{(N_H/N)(N_F/N)} \\ &= \frac{N_{HF}^2}{N_H N_F} \end{aligned}$$

When category utility is used in selection guessing, these counts are defined as follows:

- N_H is the number of matches to the hypothesis H;
- $N_F = D_F/n_F$ is the number of records predicted by likely regular feature F;
- N_{HF} is the number of matches to hypothesis H that contain exactly n_F occurrences of likely regular feature F.

For each hypothesis, the category utility is averaged across all likely regular features to compute a score between 0 and 1, which is shown (as a percentage) in the Score column in Figure 9.2(b).

9.2.2 Simultaneous Editing Algorithm

We now turn to the algorithm used in simultaneous editing mode. The inputs to the algorithm are a document, a set of records, and a set of positive examples with at most one example per record. Negative examples are not used in this algorithm. The output is a region set that selects exactly one region in every record, plus a human-readable description of the region set, which is usually but not always a TC pattern.

The algorithm is split into three parts (Figure 9.15):

- *Feature generation* takes the set of records as input and generates a list of useful features as output. Unlike the selection guessing algorithm, which repeats feature generation every time an new example is given, feature generation takes place only once, when the user first enters simultaneous editing mode.
- *Hypothesis generation* takes the positive examples and the feature list and searches for a region set consistent with the examples. The search is repeated whenever the user gives a new positive example.

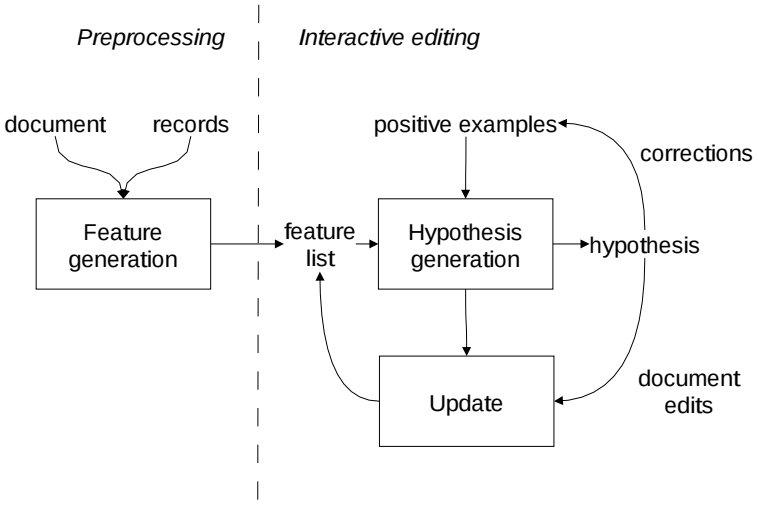


Figure 9.15: Block diagram of simultaneous editing algorithm.

- *Update* occurs when the user edits the records by inserting, deleting, or copying and pasting text. The update algorithm takes the user's edit action as input and modifies the feature list to update it.

Each of these parts is described below.

Feature Generation

Simultaneous editing uses a different set of features than selection guessing. In simultaneous editing, a feature is a region set that has at least one region in each record, and no regions outside any records. Initially, the algorithm generates two kinds of features: *library features* derived from the pattern library, and *literal features* discovered by examining the text of the records.

Library features are found by matching every named pattern in the pattern library, then discarding any patterns that do not have at least one match in every record. This is justified by two assumptions: first, that every hypothesis in simultaneous editing must have at least one match in every record; and second, that all hypotheses will be represented as conjunctions of features. Given these two assumptions, only features that match somewhere in every record will be useful for constructing hypotheses. The region set matching each named pattern is also intersected with `in` *`record`*, where *`record`* is the set of records, in order to eliminate matches that lie outside the record set.

By similar reasoning, the only useful literal features are substrings that occur at least once in every record. The common substrings of the records are found using a suffix tree, the same way that `contains` features are found for selection guessing.

After generating library features and literal features, the features are sorted in order of preference. This step essentially determines the preference bias of the learning algorithm. Placing the most-preferred features first makes the hypothesis search simpler. When the features are ordered, the search can just scan down the list of features and stop as soon as it finds the feature it needs to satisfy the examples, since this feature is guaranteed to be the most preferred consistent feature.

Features are classified into three groups for the purpose of preference ordering:

- *unique features*, which occur exactly once in each record;
- *regular features*, which occur exactly n times in each record, for some $n > 1$; and
- *varying features*, which occur a varying number of times in each record.

A feature's classification is not predetermined. Instead, it is found by counting occurrences in the records being edited. For example, in Figure 9.4, `Type` is unique because it occurs exactly once in every record. Regular features are commonly found as delimiters. For example, if the records are IP addresses like 127.0.0.1, then "." is a regular feature. Varying features are typically tokens like words and numbers, which are general enough to occur in every record but do not necessarily follow any regular pattern.

Unique features are preferred over the other two kinds. A unique feature has the simplest description: the feature name itself, like `Type`. By contrast, using a regular or varying feature in a hypothesis requires specifying the index of a particular occurrence, such as `5th Word`. Regular features are preferred over varying features, because regularity of occurrence is a strong indication that the feature is relevant to the internal structure of a record.

Within each kind of feature, library features are preferred over literal features. Among literal features, longer literals are preferred to shorter ones. Among library features, however, no ordering is currently defined. In the future, it would be useful to let the user specify preferences between patterns in the library, so that, for instance, Java features could be preferred over character-class features.

To summarize, feature generation orders the feature list in the following order, with most preferred features listed first:

1. unique library patterns
2. unique literals
3. regular library patterns
4. regular literals
5. varying library patterns
6. varying literals

Within each group of library patterns, the order is arbitrary. Within each group of literals, longer literals are preferred to shorter.

Hypothesis Generation

Hypothesis generation takes the user's positive examples and the feature list, and attempts to generate a region set consistent with the examples. Unlike the selection guessing algorithm, however, only one hypothesis is returned by this search. This decision was made so that the hypothesis search would be as fast as possible, allowing simultaneous editing to have the best possible response time.

The search process works as follows. Using the first positive example, called the *seed example*, the system scans through the feature list, testing the seed example for membership in each feature's region set. When a matching feature `F` is found, the system constructs one or more candidate descriptions representing the particular occurrence that matched, depending on the type of the feature:

- if `F` is a unique feature, then the candidate description is just `F`.
- if `F` is a regular feature, then the candidate description is either `i`th `F` or `j`th from last `F`, whichever index is smaller.
- if `F` is a varying feature, then two candidate descriptions are generated: `i`th `F` and `j`th from last `F`.

These candidate descriptions are tested against the other positive examples, if any. The first consistent description found is returned as the hypothesis.

The output of the search process depends on whether the user is placing an insertion point (e.g. by clicking the mouse) or selecting a region (e.g. by dragging). If all the positive examples are zero-length regions, then the system assumes that the user is placing an insertion point, and searches for a point description. Otherwise, the system searches for a region description.

To search for a point description, the system transforms the seed example, which is just a character offset b, into two region rectangles: $(b, b, b, +\infty)$, which represents all regions that start at b, and $(-\infty, b; b, b)$, which represents all regions that end at b. The search then tests these region rectangles for intersection with each feature in the feature list. Candidate descriptions generated by the search are transformed into point descriptions by prefixing `point just before` or `point just after`, depending on which region rectangle matched the feature, and then the descriptions are tested for consistency with the other positive examples.

To search for a region description, the system first searches for the seed example using the basic search process described above. If no matching feature is found — because the seed example does not correspond precisely to a feature on the feature list — then the system splits the seed example into its start point and end point, and recursively searches for point descriptions for each point. Candidate descriptions for the start point and end point are transformed into a description of the entire region by wrapping them with `from...to...`, and then tested for consistency with the other examples.

This search algorithm is capable of generalizing a selection only if it starts and ends on a feature boundary. For literal features, this is not constraining at all. Since a literal feature is a string that occurs in all records, every substring of a literal feature is *also* a literal feature. Thus every position in a literal feature lies on a feature boundary. To save space, only maximal literal features are stored in the feature list, and the search phase tests whether the seed example falls anywhere inside a maximal literal feature.

Update

In simultaneous editing, the user is not only making selections, but also editing the document. Editing has two effects on inference. First, every edit changes the start and end offsets of regions, so the region sets used to represent features become incorrect. Second, editing changes the document

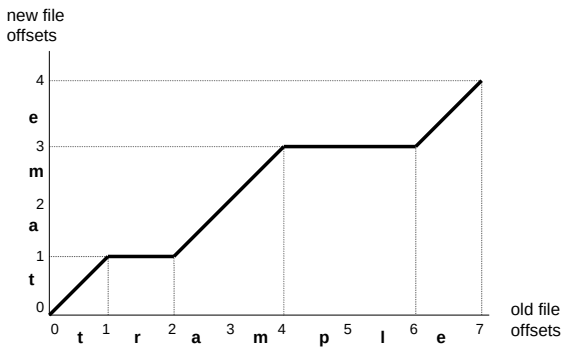


Figure 9.16: Coordinate map translating offsets between two versions of a document. The old version is the word `trample`. Two regions are deleted to get the new version, `tame`.

content, so the precomputed features may become incomplete or wrong. For example, after the user types some new words, the precomputed `Word` feature becomes incomplete, since it doesn't include the new words the user typed. The update algorithm addresses these two problems.

From the locations and length of text inserted or deleted, the updating algorithm computes a *coordinate map*, a relation that translates a character offset prior to the change into the equivalent character offset after the change. (Coordinate maps were described in Section 6.2.12.) The coordinate map can translate coordinates in either direction. For example, Figure 9.16 shows the coordinate map for a simple edit. Offset 3 in `trample` corresponds to offset 2 in `tame`, and vice versa. Offsets with more than one possible mapping in the other version, such as offset 1 in `tame`, are resolved arbitrarily; LAPIS picks the largest value.

Since the coordinate map for a group of insertions or deletions is always piecewise linear, it is represented as a sorted list of the endpoints of each line segment. If a single edit consists of m insertions or deletions (one for each record), then this representation takes $O(m)$ space. Evaluating the coordinate map function for a single offset takes $O(\log m)$ time using binary search.

A straightforward way to use the coordinate map is to scan down the feature list and update the start and end points of every feature to reflect the change. If the feature list is long, however, and includes some large feature sets like `Word` or `Whitespace`, the cost of updating every feature after every edit may be prohibitive. LAPIS takes the reverse strategy: instead of translating all features up to the present, the user's examples are translated back to the past. The system maintains a global coordinate map representing the translation between original document coordinates when the feature list was generated and the current document coordinates. When an edit occurs, the updating algorithm computes a coordinate map for the edit and composes it with this global coordinate map. When the user provides examples for a new selection, the examples are translated back to the original document coordinates using the inverse of the global coordinate map. The search algorithm generates a consistent description for the translated examples. The generated description is then translated forward to current document coordinates before being displayed as a selection.

An important design decision in a simultaneous editing system that uses domain knowledge, such as Java syntax, is whether the system should attempt to reparse the document while the user is editing it. On one hand, reparsing would allow the system to track all the user's changes and reflect

those changes in its descriptions. On the other hand, reparsing is expensive and may fail if the document is in an intermediate, syntactically incorrect state. LAPIS never reparses automatically in simultaneous editing mode. The user can explicitly request reparsing with a menu command, which effectively restarts simultaneous editing using the same set of records. Otherwise, the feature list remains frozen in the original version of the document. One consequence of this decision is that the human-readable descriptions returned by the inference algorithm may be misleading because they refer to an earlier version.

This design decision raises an important question. If the feature list is frozen, how can the user make selections in newly-inserted text, which didn't exist when the feature list was built? This problem is handled by the update algorithm. Every character typed in simultaneous editing mode adds a new literal feature to the feature list, since typed characters are guaranteed to be identical in all the records. Similarly, pasting text from the clipboard creates a special feature that translates coordinates back to the source of the paste and tries to find a description there. When a hypothesis uses one of these features created by editing, the feature is described as "somewhere in edit$N$", which can be seen in Figure 9.9.

A disadvantage of this scheme is that the housekeeping structures – the global coordinate map and the new features added for edits – grow steadily as the user edits. This growth can be slowed significantly by coalescing adjacent insertions and deletions, although LAPIS does not yet include this optimization. Another solution might be to reparse when the number of edits reaches some threshold, doing the reparsing in the background on a copy of the document in order to avoid interfering with the user's editing. In practice, however, space growth doesn't seem to be a serious problem. For most tasks, the user spends only a few minutes in a simultaneous editing session, not the hours that are typical of general text editing. After leaving simultaneous editing mode, the global coordinate map and the feature list can be discarded.

9.3 User Studies

Selection guessing and simultaneous editing were tested with a pair of user studies, described in this section.

The first study evaluated simultaneous editing, which was actually the first of the two selection inference modes to be designed and implemented. At the time the study was done, the user interface for simultaneous editing was somewhat different. The most significant difference affected the first step of simultaneous editing, in which the user selects the records. Because selection guessing had not been implemented yet, the records had to be selected a different way, by choosing a named pattern from the library pane. As a result, the dialog boxes were worded differently. Screenshots of the original dialog boxes are included in the description of the study below.

The second study evaluated selection guessing, using one of the tasks used in the simultaneous editing task, so that some comparisons can be made between the two techniques.

9.3.1 Simultaneous Editing Study

The first study was designed to evaluate the usability of simultaneous editing on small repetitive editing tasks. The study compared simultaneous editing with manual (single-cursor) editing along two quantitative dimensions: time to complete the task, and errors in the finished product. Manual

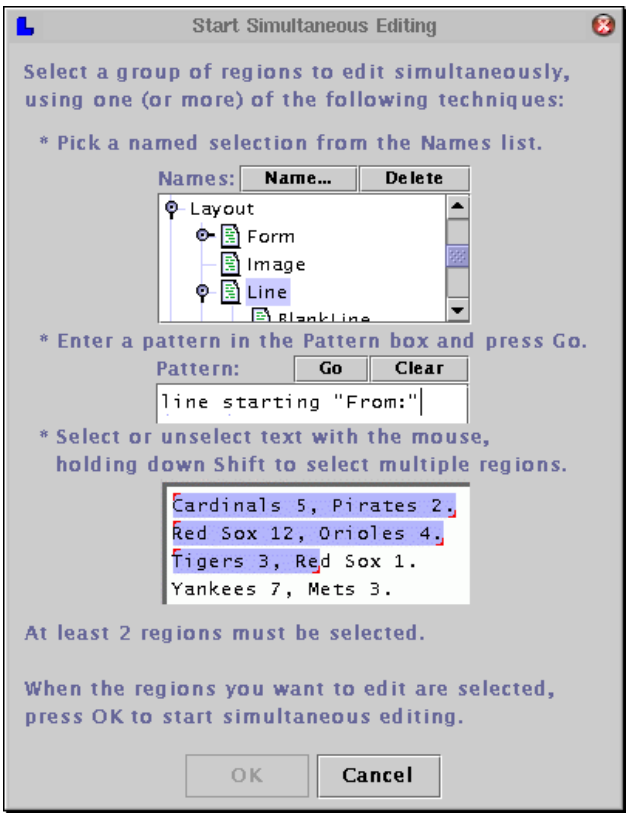


Figure 9.17: The Simultaneous Editing dialog box used in the user study.

editing was chosen as the basis for comparison because the users were already experts in manual editing, and manual editing would often be the natural alternative for these small tasks. Still, it is also desirable to compare simultaneous editing with other research approaches. Two of the tasks used in the study were borrowed from the user study of another programming-by-demonstration system, DEED [Fuj98], allowing the performance of simultaneous editing on those tasks to be compared with the performance of DEED.

In the version of LAPIS tested in this study, entering simultaneous editing mode popped up a dialog box window instead of a dialog pane. This dialog box (Figure 9.17) prompted the user to select the record set using one of the existing selection methods — mouse, pattern, or library. For all the tasks in the user study, the record set could be selected using a library pattern (either `Line` or `Paragraph`), so it was not necessary to teach users the TC pattern language.

Procedure

Eight users were found by soliciting campus job newsgroups (`cmu.misc.jobs` and `cmu.misc.market`). All were college undergraduates with substantial text-editing experience and varying levels of programming experience (5 described their programming experience as “little” or “none,” and 3 as “some” or “lots”). Users were paid $5 for participation.

Users first learned about simultaneous editing by reading a tutorial and trying the examples. The tutorial led the user through two example tasks, showing step by step how the tasks could be performed with simultaneous editing. The tutorial tasks are shown in Figures 9.18 and 9.19. The

1. Let a set of state variables R be null. Let a sequence of pieces of input s contain only the goal input. Let a piece of input i be the goal input.
2. Add state variables vj's in REF(i) to R, if not included already.
3. If R is null, end the phase.
4. Search the piece of input p that immediately precedes i in the source input.
5. If p is not found, i.e., the search reaches the beginning of the source input, add the pieces of input that initialize vj's in R to the head of s, and end the phase.
6. If C = DEF(p) * R is not null, add p to the head of s, remove vj's in C from R, let i be p, and go to 2.
7. Let i be p, and go to 4.

↓

(1) Let a set of state variables R be null. Let a sequence of pieces of input s contain only the goal input. Let a piece of input i be the goal input.
(2) Add state variables vj's in REF(i) to R, if not included already.
(3) If R is null, end the phase.
(4) Search the piece of input p that immediately precedes i in the source input.
(5) If p is not found, i.e., the search reaches the beginning of the source input, add the pieces of input that initialize vj's in R to the head of s, and end the phase.
(6) If C = DEF(p) * R is not null, add p to the head of s, remove vj's in C from R, let i be p, and go to 2.
(7) Let i be p, and go to 4.

Figure 9.18: Tutorial task 1: change the numbering of a list.

tutorial part lasted less than 10 minutes for all but one user, who spent 30 minutes exploring the system.

After the tutorial, each user performed three tasks with simultaneous editing. The three tasks are shown in their entirety in Figures 9.20–9.22. All tasks were obtained from other authors: tasks 1 and 2 from Fujishima [Fuj98] and task 3 from Nix [Nix85]. After performing a task with simultaneous editing, users repeated the same task in manual selection mode, but only on the first three records to avoid unnecessary tedium. Users were instructed to work carefully and accurately at their own pace. All users were satisfied that they had completed all tasks, although the finished product sometimes contained undetected errors, a problem discussed further below.

LAPIS was instrumented to capture all selections made by the user, all inferences made by the system, and all editing actions. An experimenter also observed all the sessions and made notes. No audio or video recordings were made.

Results

Aggregate times for each task are shown in Table 9.1. No performance differences were seen between programmers and nonprogrammers.

Following the analysis used by Fujishima [Fuj98], the leverage obtained with simultaneous editing can be estimated by dividing the time to edit all records with simultaneous editing by the time to edit just one record manually. This ratio, which I call *equivalent task size*, represents the number of records for which simultaneous editing time would be equal to manual editing time

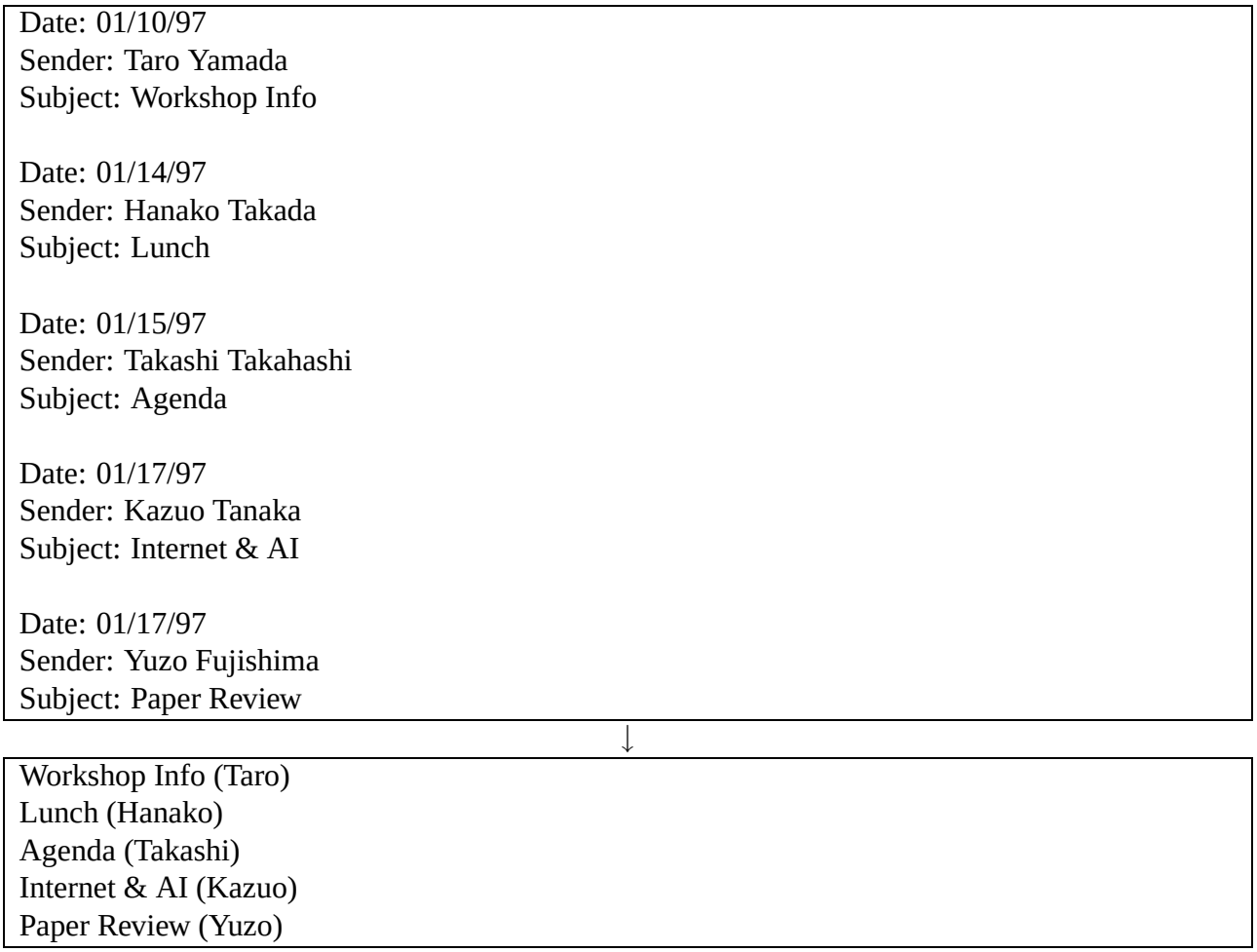


Figure 9.19: Tutorial task 2: reformat email message headers into a short list.

Task	Simultaneous editing	Manual editing
1	142.9 s [63–236 s]	21.6 s/rec [7.7–65 s/rec]
2	119.1 s [64–209 s]	32.3 s/rec [19–40 s/rec]
3	159.6 s [84–370 s]	41.3 s/rec [16–77 s/rec]

Table 9.1: Time taken by users to perform each task (mean [min–max]). *Simultaneous editing* is the time to do the entire task with simultaneous editing. *Manual editing* is the time to edit 3 records of the task by hand, divided by 3 to get a per-record estimate.

	Simultaneous editing		DEED
Task	novices	expert	novices
1	8.4 recs [2.1–12.2 recs]	4.5 recs	67 recs [6.5–220 recs]
2	3.6 recs [1.9–5.8 recs]	1.6 recs	28 recs [7–130 recs]
3	4.0 recs [1.9–6.2 recs]	2.4 recs	

Table 9.2: Equivalent task sizes for each task (mean [min–max]). *Novices* are users in the user study. *Expert* is my own performance, provided for comparison. *DEED* is another PBD system, tested on tasks 1 and 2 by its author [Fuj98].

1. Aha, D.W. and Kibler, D. Noise-tolerant instance-based learning algorithms. In Proceedings of the Eleventh International Joint Conference on Artificial Intelligence. Morgan Kaufmann, 1989, pp. 794-799.
2. Brajnik, G. and Tasso, C. A Shell for developing non-monotonic user modeling systems. Int. J. Human-Computer Studies 40 (1994), 31-62.
3. Cohen, P.R., Cheyer, A., Wang, M., and Baeg, S.C. An Open Agent Architecture. In Software Agents: Papers from the AAAI 1994 Spring Symposium, AAAI Press, 1994, pp. 1-8.
4. Corkill, D.D. Blackboard Systems. AI Expert 6, 9 (Sep. 1991), 40-47.
5. Cranefield, S. and Purvis, M. Agent-based Integration of General Purpose Tools. In Proceedings of the Workshop on Intelligent Information Agents. Fourth International Conference on Information and Knowledge Management, Baltimore, 1995.
6. Finin, T, Fritzson, R., McKay, D. and McEntire, R. KQML A Language and Protocol for Knowledge and Information Exchange. In Fuchi, K. and Yokoi, T., Eds. Knowledge Building and Knowledge Sharing. Ohmsha and IOS Press, 1994.
7. Hayes-Roth, B. Pfleger, K. Morignot, P. and Lalanda, P. Plans and Behavior in Intelligent Agents, Technical Report KSL-95-35, Stanford University, 1995.
8. Kosbie, D.S. and Myers, B.A. A System-Wide Macro Facility Based on Aggregate Events: A Proposal. In Cypher, A., Ed. Watch What I Do: Programming by Demonstration. The MIT Press, Cambridge, Mass., 1993.
9. Martin, D.L., Cheyer, A., and Lee, G-L. Development Tools for the Open Agent Architecture. In Proceedings of the First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology. The Practical Application Company Ltd., London, April 1996, pp. 387-404.

↓

[Aha 89] Aha, D.W. and Kibler, D. Noise-tolerant instance-based learning algorithms. In Proceedings of the Eleventh International Joint Conference on Artificial Intelligence. Morgan Kaufmann, 1989, pp. 794-799.
[Brajnik 94] Brajnik, G. and Tasso, C. A Shell for developing non-monotonic user modeling systems. Int. J. Human-Computer Studies 40 (1994), 31-62.
[Cohen 94] Cohen, P.R., Cheyer, A., Wang, M., and Baeg, S.C. An Open Agent Architecture. In Software Agents: Papers from the AAAI 1994 Spring Symposium, AAAI Press, 1994, pp. 1-8.
[Corkill 91] Corkill, D.D. Blackboard Systems. AI Expert 6, 9 (Sep. 1991), 40-47.
[Cranefield 95] Cranefield, S. and Purvis, M. Agent-based Integration of General Purpose Tools. In Proceedings of the Workshop on Intelligent Information Agents. Fourth International Conference on Information and Knowledge Management, Baltimore, 1995.
[Finin 94] Finin, T, Fritzson, R., McKay, D. and McEntire, R. KQML A Language and Protocol for Knowledge and Information Exchange. In Fuchi, K. and Yokoi, T., Eds. Knowledge Building and Knowledge Sharing. Ohmsha and IOS Press, 1994.
[Hayes 95] Hayes-Roth, B. Pfleger, K. Morignot, P. and Lalanda, P. Plans and Behavior in Intelligent Agents, Technical Report KSL-95-35, Stanford University, 1995.
[Kosbie 93] Kosbie, D.S. and Myers, B.A. A System-Wide Macro Facility Based on Aggregate Events: A Proposal. In Cypher, A., Ed. Watch What I Do: Programming by Demonstration. The MIT Press, Cambridge, Mass., 1993.
[Martin 96] Martin, D.L., Cheyer, A., and Lee, G-L. Development Tools for the Open Agent Architecture. In Proceedings of the First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology. The Practical Application Company Ltd., London, April 1996, pp. 387-404.

Figure 9.20: Task 1: put author name and year in front of each citation in a bibliography [Fuj98].

```
<DT><A HREF="mailto:cg@cs.umn.edu" NICKNAME="congra">Conceptual Graphs</A>
<DT><A HREF="mailto:DAI-List@ece.sc.edu" NICKNAME="dai">Distributed Artificial Intelligence</A>
<DT><A HREF="mailto:ii_chi@acm.org" NICKNAME="ii">Intelligent Interfaces</A>
<DT><A HREF="mailto:kif@cs.stanford.edu" NICKNAME="kif">KIF</A>
<DT><A HREF="mailto:kqml@cs.umbc.edu" NICKNAME="kqml">KQML</A>
<DT><A HREF="mailto:agents@cs.umbc.edu" NICKNAME="swagent">Software Agents</A>
<DT><A HREF="mailto:tscript@brownvm.brown.edu" NICKNAME="tscript">Telescript</A>
```

↓

```
;; Conceptual Graphs
congra: mailto:cg@cs.umn.edu
;; Distributed Artificial Intelligence
dai: mailto:DAI-List@ece.sc.edu
;; Intelligent Interfaces
ii: mailto:ii_chi@acm.org
;; KIF
kif: mailto:kif@cs.stanford.edu
;; KQML
kqml: mailto:kqml@cs.umbc.edu
;; Software Agents
swagent: mailto:agents@cs.umbc.edu
;; Telescript
tscript: mailto:tscript@brownvm.brown.edu
```

Figure 9.21: Task 2: reformat a list of mail aliases from HTML to plain text [Fuj98].

```
Cardinals 5, Pirates 2.
Red Sox 12, Orioles 4.
Tigers 3, Red Sox 1.
Yankees 7, Mets 3.
Dodgers 6, Tigers 4.
Brewers 9, Braves 3.
Phillies 2, Reds 1.
```

↓

```
GameScore[winner 'Cardinals'; loser 'Pirates'; scores[5, 2]].
GameScore[winner 'Red Sox'; loser 'Orioles'; scores[12, 4]].
GameScore[winner 'Tigers'; loser 'Red Sox'; scores[3, 1]].
GameScore[winner 'Yankees'; loser 'Mets'; scores[7, 3]].
GameScore[winner 'Dodgers'; loser 'Tigers'; scores[6, 4]].
GameScore[winner 'Brewers'; loser 'Braves'; scores[9, 3]].
GameScore[winner 'Phillies'; loser 'Reds'; scores[2, 1]].
```

Figure 9.22: Task 3: reformat a list of baseball scores into a tagged format [Nix85].

for a given user. Since manual editing time increases linearly with record number, and simultaneous editing time is roughly constant or only slowly increasing, simultaneous editing will be faster whenever the number of records is greater than the equivalent task size. Note that the average equivalent task size is not necessarily equal to the ratio of the average editing times, since $E[S/M] \neq E[S]/E[M]$.

As Table 9.2 shows, the average equivalent task sizes are small. In other words, the average novice user works faster with simultaneous editing if there are more than 8.4 records in the first task, more than 3.6 records in the second task, or more than 4 records in the third task.[2] Thus simultaneous editing is an improvement over manual editing even for very small repetitive editing tasks, and even for users with as little as 10 minutes of experience. Some users were so slow at manual editing that their equivalent task size is smaller than the expert's, so simultaneous editing benefits them even more.

Simultaneous editing also compares favorably to another PBD system, DEED [Fuj98]. When DEED was evaluated with novice users on tasks 1 and 2, the reported equivalent task sizes averaged 67 for task 1 and 28 for task 2, roughly an order of magnitude larger than simultaneous editing. The variability of equivalent task sizes across users was also considerably greater for DEED.

Another important aspect of system performance is inference accuracy. Each incorrect inference forces the user to make at least one additional action, such as selecting a counterexample or providing an additional positive or negative example. In the user study, users made a total of 188 selections that were used for editing. Of these, 158 selections (84%) were correct immediately, requiring no further examples. The remaining selections needed either 1 or 2 extra examples to generalize correctly. On average, only 0.26 additional examples were needed per selection.

Unfortunately, users tended to overlook slightly-incorrect inferences, particularly inferences that selected only half of the hyphenated author "Hayes-Roth" or the two-word baseball team "Red Sox". As a result, the overall error rate for simultaneous editing was slightly worse than for manual editing: 8 of the 24 simultaneous editing sessions ended with at least one uncorrected error, whereas 5 of 24 manual editing sessions ended with uncorrected errors. If the two most common selection errors had been noticed by users, the error rate for simultaneous editing would have dropped to only 2 of 24. These observations led to the idea of *outlier finding*, which is covered in Chapter 10.

After doing the tasks, users were asked to evaluate the system's ease-of-use, trustworthiness, and usefulness on a 5-point Likert scale. These questions were also borrowed from Fujishima [Fuj98]. The results, shown in Figure 9.23, are generally positive.

9.3.2 Selection Guessing Study

The second study was designed to evaluate the usability of selection guessing on repetitive editing tasks. In order to compare selection guessing with simultaneous editing, the overall design of the study was the same as the first study, except that users used selection guessing to perform tasks. Unfortunately, not all the tasks of the first study can be performed with selection guessing. Tasks 1 and 3 require counted patterns for some of their selections (e.g., `first CapitalizedWord`

[2] These estimates are actually conservative. Simultaneous editing always preceded manual editing for each task, so the measured time for simultaneous editing includes time spent thinking about and understanding the task. For the manual editing part, users had already learned the task, and were able to edit at full speed.

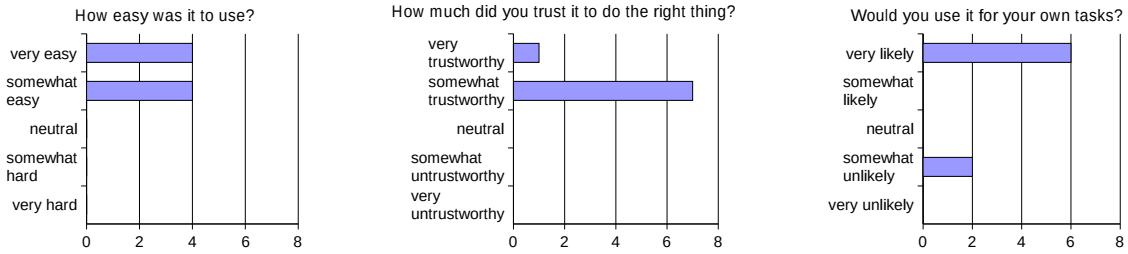


Figure 9.23: User responses to questions about simultaneous editing.

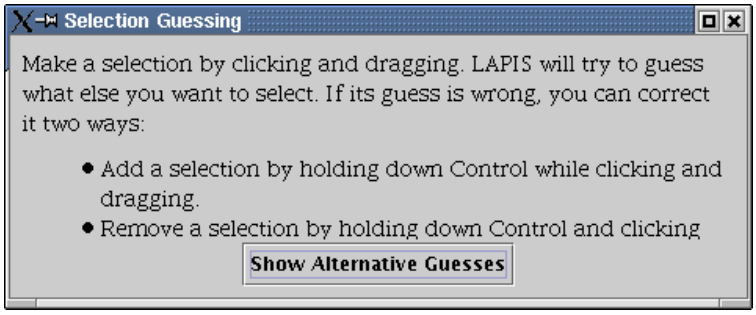


Figure 9.24: The Selection Guessing dialog box used in the user study.

`in Line`). These patterns can be inferred by the simultaneous editing algorithm, but not by the selection guessing algorithm. As a result, the selection guessing study used only task 2.

As was the case in the previous study, the user interface of selection guessing mode was slightly different when the study was performed. The selection guessing dialog was a popup window rather than a pane, and the controls were arranged somewhat differently, although all the same features were available. Figure 9.24 shows the dialog box used in the study.

Procedure

Five users were found by soliciting campus job newsgroups (`cmu.misc.jobs` and `cmu.misc.market`). All were college undergraduates with substantial text-editing experience and varying levels of programming experience (1 described their programming experience as "little," 3 as "some," and 1 as "lots"). Users were paid $5 for participation.

As in the previous study, users learned about selection guessing by reading a tutorial and trying the examples. The tutorial led the user through one example task (Figure 9.19). The tutorial part lasted less than 10 minutes for all users.

After the tutorial, each user performed one task in selection guessing mode (Task 2, Figure 9.21), and then repeated the first three records of the task in manual selection mode. Users were instructed to work carefully and accurately at their own pace.

Results

Four out of five users were able to complete the task entirely with selection guessing. The fifth user also completed the task, but only by exiting selection guessing mode at one point, doing a troublesome part with manual editing, and then resuming selection guessing to finish the task.

Task	Selection guessing	Manual editing
2	426.0 s [173–653 s]	43.0 s/rec [32–52 s/rec]

Table 9.3: Time taken by users to perform the task (mean [min–max]). *Selection guessing* is the time to do the entire task with selection guessing. *Manual editing* is the time to edit 3 records of the task by hand, divided by 3 to get a per-record estimate.

Task	Selection guessing		Simultaneous editing		DEED
	novices	expert	novices	expert	novices
2	9.3 recs [4.7–15.7 recs]	2.3 recs	3.6 recs [1.9–5.8 recs]	1.6 recs	28 recs [7–130 recs]

Table 9.4: Equivalent task sizes (mean [min–max]) for task 2, comparing selection guessing, simultaneous editing, and DEED. *Novices* are users in a user study. *Expert* is my own performance, provided for comparison.

Aggregate times for selection guessing and manual editing are shown in Table 9.3. The times for selection guessing include the detour into manual editing made by one user.

The same analysis as the first study is used to compute the equivalent task size for doing this task with selection guessing. The results are shown in Table 9.4. For comparison, the table also includes the equivalent task sizes for the same task in the simultaneous editing study and the DEED study. Selection guessing is not as fast as simultaneous editing on this task, but still outperforms DEED.

One reason that selection guessing was slower is less accurate inference. Of the 51 selections used for editing in the selection guessing study, only 34 (67%) were inferred correctly from one example. By comparison, in the simultaneous editing study, *all* selections on task 2 were inferred correctly from one example.

In selection guessing, an incorrect inference can be corrected in three ways: giving another positive example, giving a negative example, or selecting an alternative hypothesis. To judge the user cost of inference, then, we must measure the number of *actions* needed to create a selection, where an action is either giving an example or clicking on an alternative hypothesis. On average, 2.73 actions were needed to create each selection used for editing in selection guessing, compared to 1 action per selection in simultaneous editing.

After the study, users evaluated selection guessing's ease-of-use, trustworthiness, and usefulness on a 5-point Likert scale. The results, shown in Figure 9.25, are considerably more mixed than for simultaneous editing.

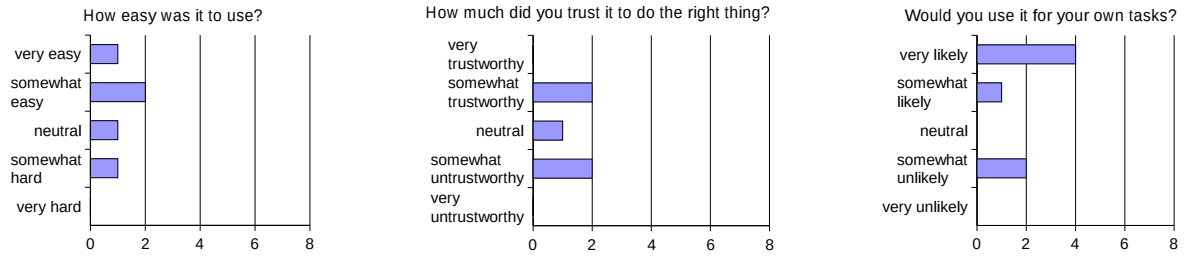


Figure 9.25: User responses to questions about selection guessing.