Chapter 8

Commands and Scripting

LAPIS includes a number of text-processing commands that are enabled by lightweight structure, primarily because the pattern library and TC pattern language make it possible to describe command arguments simply and concisely. This chapter[1] describes these commands and gives some examples of how they are used.

Many of the commands described in this chapter are inspired by the suite of text-processing tools in Unix [KP84]. Unix tools, such as `grep` and `sort`, are designed to be as generic and task-agnostic as possible, so that tasks can be done by combining a few generic tools rather than writing a program from scratch. Unfortunately, the generic nature of existing Unix tools is also a weakness, because generic tools can make only limited assumptions about the format of the text. Most Unix tools assume that the input is divided into records separated by newlines (or some other fixed delimiter character). But this assumption breaks down for richly structured text, such as source code, HTML, and XML. To overcome this difficulty, the LAPIS versions of these tools allow the structure of the input to be described by region sets — using library patterns, TC patterns, mouse selection, inference from examples (Chapter 9), or some combination.

One interface to these tools is graphical, using menus and dialog boxes to choose a command, configure its options, and invoke it. However, an important lesson from Unix is that generic tools provide the greatest leverage when they can be glued together into scripts and pipelines, creating new tools. Taking this lesson to heart, LAPIS also embeds a scripting language, Tcl [Ous94], and makes its text-processing tools available as Tcl commands as well. Tcl was chosen partly because of its syntactic simplicity, and partly because a good Java implementation was available [DeJ98].

Tcl is also well-suited to interactive command execution. Like the Unix shell, LAPIS has an interactive interpreter that allows script commands to be entered on the fly. Unlike the Unix shell, however, the LAPIS interpreter does not use a typescript shell, but a *browser shell*. The browser shell is a novel interaction model that integrates the interpreter into a web browsing interface, using the LAPIS command bar to enter script commands, the browser pane to display output, and the browsing history to manage the history of outputs. Among the benefits of a browser shell is an interesting new way to assemble pipelines of commands interactively, described in more detail later in this chapter.

This chapter also describes some commands for interacting with web sites, so that a user can write Tcl scripts that browse the Web automatically — clicking on hyperlinks, submitting forms,

[1] Portions of this chapter are adapted from an earlier paper [MM00].

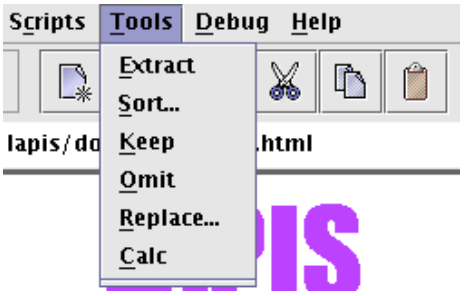


Figure 8.1: The Tools menu lists the text-processing tools built into LAPIS.

extracting data, and so forth. To create a browsing script quickly, the user can demonstrate it by recording a browsing sequence in LAPIS.

The chapter is sprinkled throughout with examples of scripts that use LAPIS commands to perform real tasks. An alphabetical reference to all LAPIS script commands can be found in Appendix C.

8.1 Text-Processing Commands

The Tools menu (Figure 8.1) presents a list of text-processing commands that can be applied to the current selection. Each menu command has an equivalent script command.

The menu commands are enabled only when at least one nonzero-length region has been selected in the browser pane. If the selection includes nested or overlapping regions, the selection is first flattened (Section 3.6.12), producing a flat region set, before being processed by the command.

All these commands produce a new document as output, rather than changing the current document. In that sense, they differ from commands in the Edit menu, like Cut, Copy, and Paste, which mutate the current document. For some tools, like Extract, generating a new document is appropriate; for others, like Sort, it may be better to change the current document, or at least give the user a choice. This design decision may be revisited in a future version of LAPIS.

After a command runs, its output document replaces the original input document in the browser pane. The original document is still available in the browsing history, so the user can use the Back button to undo the effects of the command.

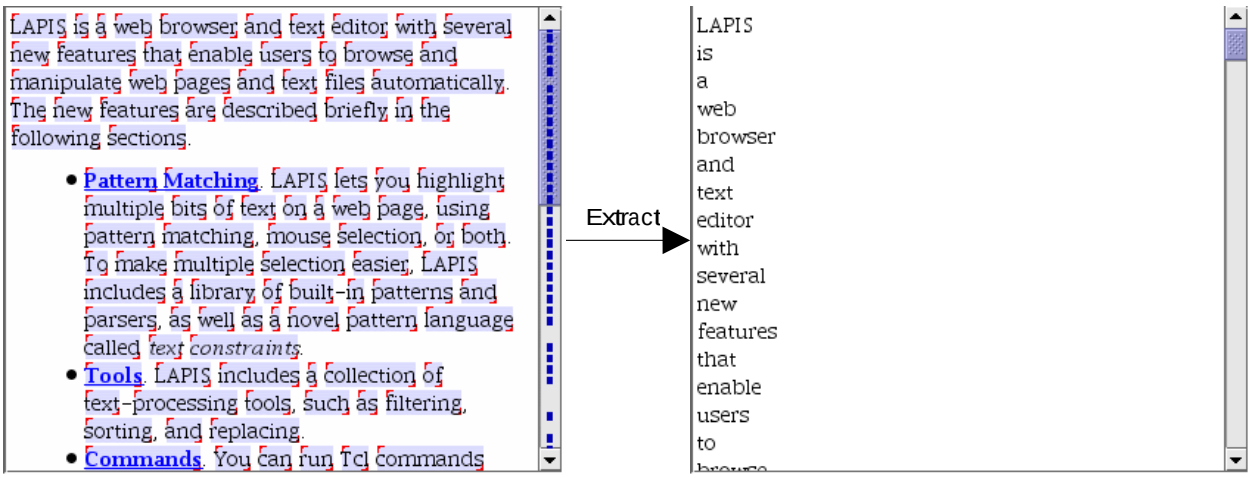
The next sections describe the current set of commands in LAPIS.

8.1.1 Extract

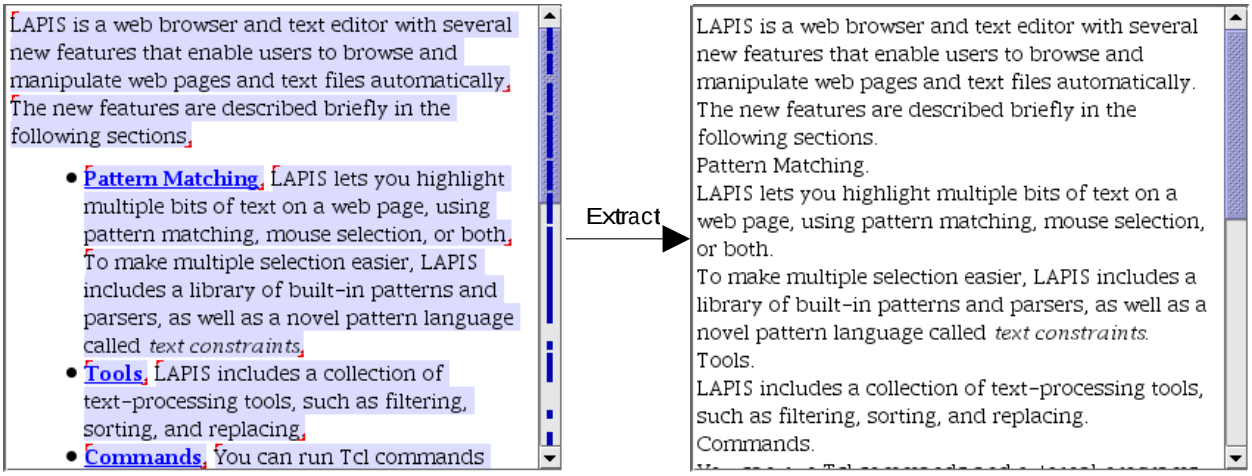
The Extract command extracts the regions in the current selection to produce a new document. The output consists of a list of items, one for each selected region. Figure 8.2 shows the results of applying Extract to various selections. The equivalent script command is

```
extract pattern
```

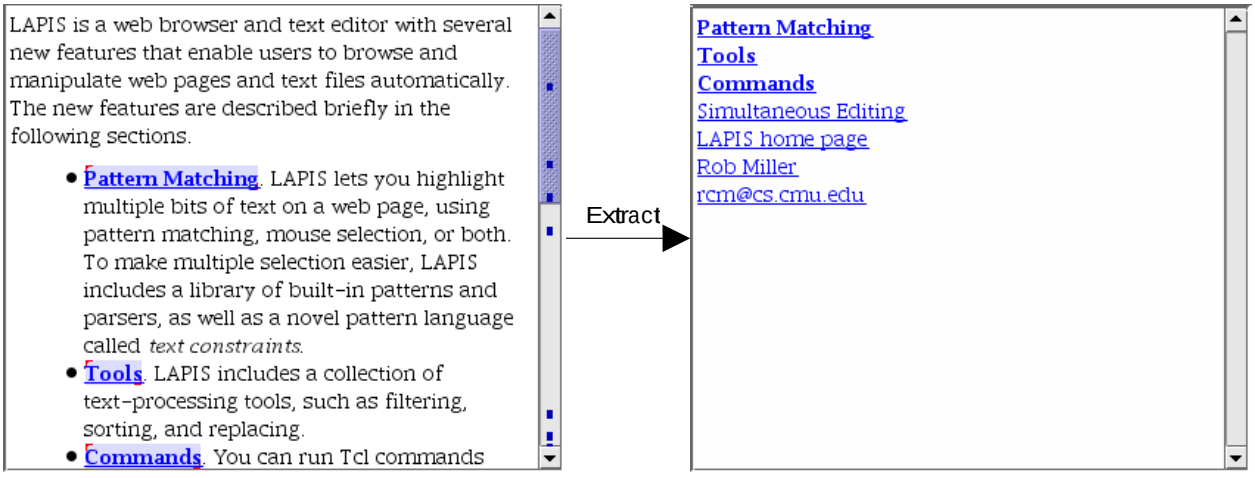
which extracts the region set matching a TC pattern. The named pattern `Selection` always returns the current selection in LAPIS, so a selection made with the mouse can be extracted by



(a) extracting words



(b) extracting sentences



(c) extracting links

Figure 8.2: Examples of the output produced by Extract for various selections.

```
extract Selection
```

By default, Extract formats its output as a list of lines. The exact formatting depends on whether the input document is HTML or plain text. For HTML, Extract inserts a line break tag, `<br>`, after every extracted region. For plain text, Extract inserts a newline character after every extracted region, unless the extracted region already ends with a newline. This default formatting is simple, compact and works well enough in most cases. When extracted regions span multiple lines, however, it may be difficult to tell where one item begins and another ends (see, for example, the extracted sentences in Figure 8.2(b)).

For more sophisticated formatting, the user can specify the text that should be printed before, after, or between the extracted items. These delimiters can be specified by giving options to the `extract` command:[2]

```
extract pattern
        [-startswith start]
        [-endswith end]
        [-separatedby sep]
        [-as type]
```

Here, *`start`* specifies a string that should be printed before each extracted region, *`end`* is a string to be printed after each region, and *`sep`* is a string to be printed between each pair of regions. For example, the command

```
extract Word -startswith ( -endswith ) -separatedby ,
```

would print all the words in the input surrounded by parentheses and separated by commas, e.g.:

```
(LAPIS),(is),(a),(web),(browser),...,(cs),(cmu),(edu)
```

A solution to the problem of distinguishing multi-line extracted regions in HTML is to separate them with horizontal rule tags, `<hr>`:

```
extract Sentence -separatedby <hr>
```

Presently, delimiters can be specified only using the script command, in order to keep the Extract menu command simple.

The `extract` script command has one other option. The `-as` *`type`* option chooses the type of the output document, where *`type`* can be one of three choices: `text`, `html`, or `tcl` (a Tcl list). By default, the output type is the same as the type of the input document, which is either text or HTML depending on how the document is being viewed in the browser pane. Thus an HTML page whose source is being viewed as plain text is considered to have type `text`.

The output type determines two things:

[2]The square brackets in this syntax specification should not be typed as part of the command; they merely denote optional arguments to the `extract` command.

- Default output formatting in the absence of `-startswith`, `-endswith`, or `-separatedby` options. As explained above, the default formatting of HTML output prints a `<br>` tag after every region. The default formatting of text output prints a newline after every region that doesn't already end with a newline. The default formatting of Tcl output prints spaces between the regions, which is the format for a Tcl list.

- Quoting and type conversion. If the output type differs from the input type, then the Extract command automatically does whatever quoting is necessary to translate from one type to the other. Converting text to HTML replaces `<` with `<`, `>` with `>`, and `&` with `&`. Converting from HTML to text deletes tags and expands entities like `<` into the corresponding characters. Converting either text or HTML to Tcl produces a Tcl list — i.e., any output region which is not a simple word is surrounded by curly braces, `{}`, and any embedded occurrences of curly braces are escaped with backslashes.

The Tcl output type is useful for extracting all pattern matches as a list for processing by other Tcl commands. For example,

```
extract Sentence -as tcl
```

produces a Tcl list of all the sentences in the document.

The Extract command is not the only way to extract the current selection in LAPIS. The selection can also be cut or copied to the clipboard and then pasted back, into either the same document or a new document. Section 7.5 described in detail how this works. Note that the Copy command works in both the HTML view and the plain text view, but Paste only works in the plain text view at present, since the HTML view is not editable. Copying from HTML to plain text does the same conversions as Extract, deleting tags and expanding entities. With copy and paste, the user can control the destination of the extracted text by making multiple selections for the paste target. Copy and paste also allows the user to change the delimiters between extracted regions interactively. Figure 8.3 shows a sequence of editing commands including copy and paste that achieve the same effect as the `extract Word` command described above.

Extract is analogous to the Unix utility `grep`. Whereas `grep` can only extract lines, Extract can pull out an arbitrary set of regions matching a pattern, such as `Sentence containing "LAPIS"` or `MethodCall starting "printf"`. The Unix command

```
grep regexp
```

is roughly equivalent to the LAPIS command

```
extract {Line containing /regexp/}
```

(The curly braces are required by Tcl to quote a multi-word pattern as a single argument.) One must say "roughly equivalent" because the regular expression may be interpreted differently by `grep` and LAPIS. Regular expression languages differ, not only between LAPIS and `grep`, but also between different versions of `grep`. Also, regular expressions in LAPIS are not constrained to match within a line, as they are in `grep`.

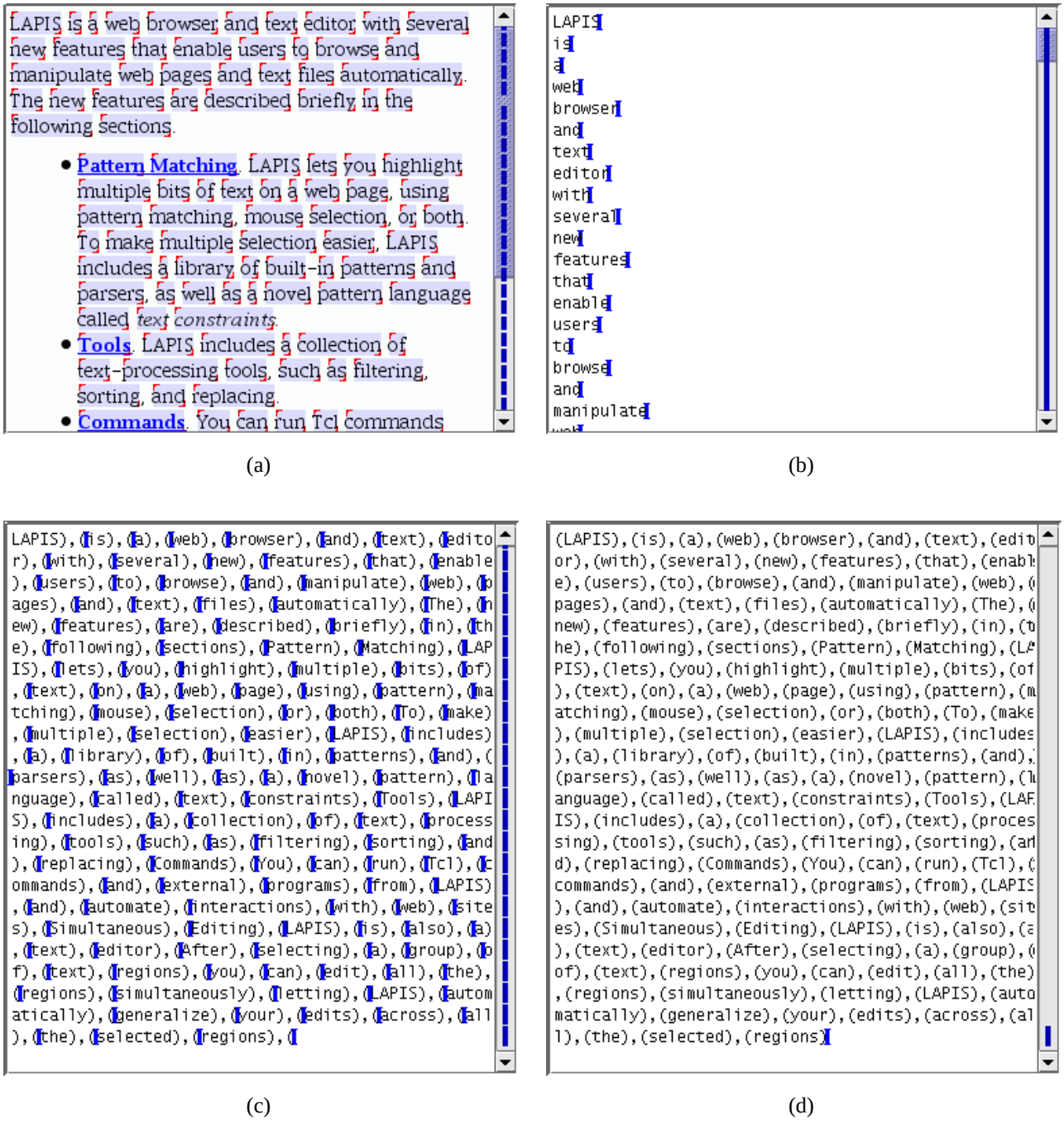


Figure 8.3: Using copy and paste to extract a selection. (a) The selection (`word`) is copied to the clipboard. (b) After making a new document with File/New, the clipboard is pasted, leaving an insertion point after each pasted region. (c) The user deletes the linebreaks that were inserted by default, and types `),(` instead. (d) The user fixes up the boundary cases, before the first and after the last extracted region.

Figure 8.4: The Sort dialog.

8.1.2 Sort

The Sort command sorts the regions in the current selection. The selection is sorted in place, not extracted, so all text outside the selection is left unchanged. The output is a new document in which the selected regions are sorted, but which is otherwise identical to the input document. Some examples of the Sort command are shown in Figure 8.5.

The Sort menu command pops up a dialog box (Figure 8.4), which offers two options. The *sort key* specifies the part of each region that is compared to determine how the regions are sorted. By default, the entire region is used as the sort key. If only part of the region should be used as the sort key, the user must give a TC pattern describing that part. The sort key for a region is then determined by concatenating all matches to the key pattern found in the region. If a region contains no matches to the sort key pattern, its sort key is the empty string.

The *sort order* specifies the collation order for sort keys. The following sort orders are available:

- **<Local Language> (a b c ...)** sorts the keys lexicographically in ascending order, case-insensitively, using the collation order for the current locale reported by the Java runtime. For users of the US locale, for example, this option reads English. For users of the Spanish locale, this option reads Spanish, and sorts accented characters in the conventional Spanish collation order.

- **Reverse <Local Language> (z y x ...)** sorts the keys lexicographically in descending order, case-insensitively.

- **Numeric (1 2 3 ...)** and **Reverse Numeric (9 8 7 ...)** sort the keys as if they were numbers. Regions which do not start with a recognizable number are treated as 0.

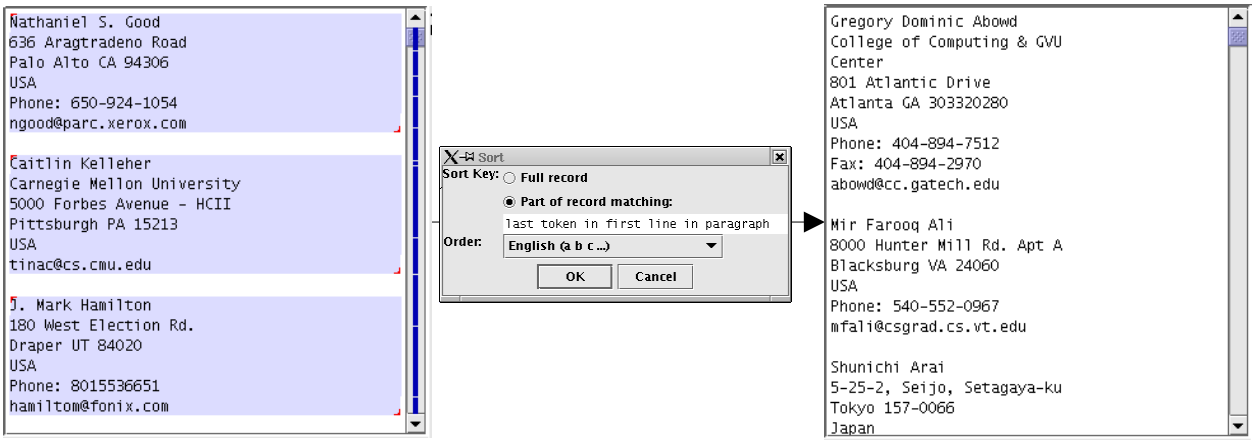
- **Unicode (A B C ... a b c ...)** and **Reverse Unicode (z y x ... Z Y X ...)** sort the keys lexicographically and case-sensitively by comparing Unicode values.

- **Random** applies a random permutation to the selected regions. The content of the sort keys is irrelevant to this sort order.

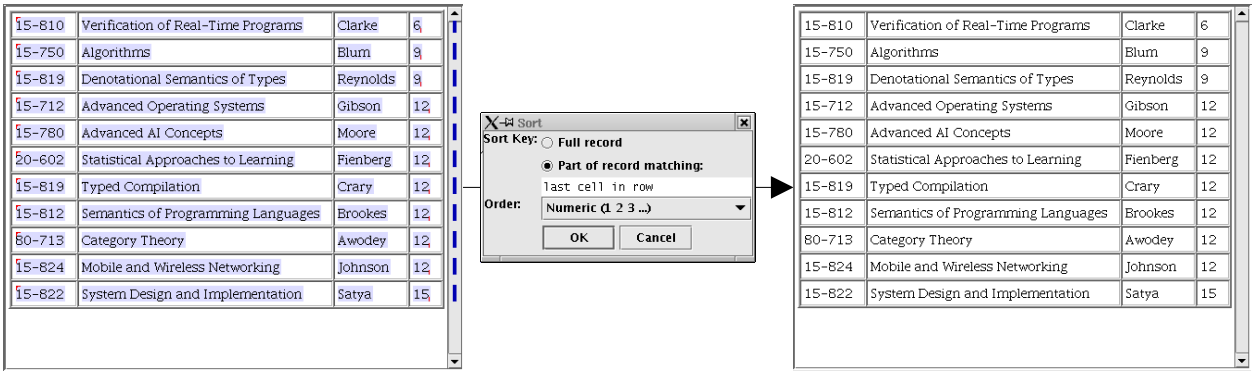
The default sort order is <Local Language>, e.g., English.

The sort orders in LAPIS were chosen to cover the common cases with a few simple choices. Not all possible orderings are available. For example, no case-sensitive local-language sort order

(a) sorting words



(b) sorting addresses by last name



(c) sorting courses by number of units

Figure 8.5: Examples of using Sort on various tasks.

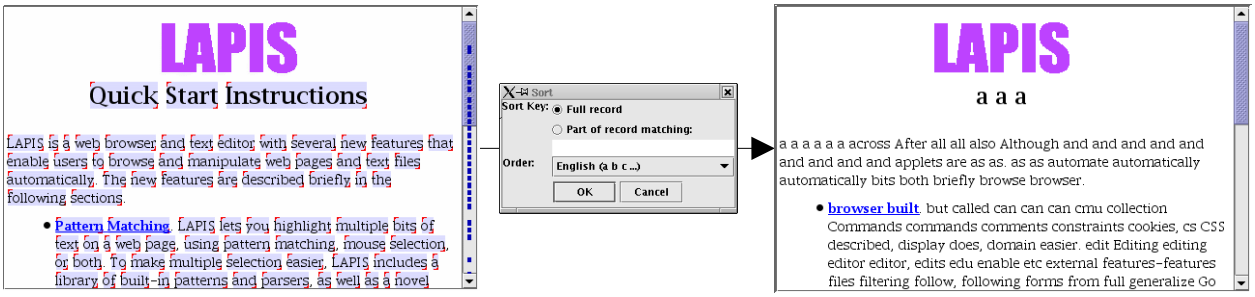


Figure 8.6: Sorting the words in a document produces gibberish. (Note that the big purple "LAPIS" was not sorted with the other words because it is an image, not text.)

is provided. If case-sensitivity is desired, the user must choose Unicode order. Other sort orders would also be useful, including dates, times, and sort orders that ignore whitespace and punctuation when comparing sort keys. The most general solution would let the user specify a custom sort order with a comparison function, like the standard C function `qsort` [KR88] or the Perl function `sort` [WCS96].

The script command for Sort has the following syntax:

```
sort pattern
     [-by keypattern]
     [-order [reverse] dictionary|numeric|unicode|random]
```

Multiple sort keys can be specified with multiple `-by` arguments. Sort keys are compared in the order that the `-by` arguments appear, so the first `-by` argument specifies the primary key. If two regions have the same primary key, the next sort key is compared, and so on. The default sort order for each key is `dictionary`, the local language. To change the sort order for a key, the `-by` option must be followed by a `-order` option. For example, the following command might sort a list of cars primarily by year (in reverse numeric order), then by make:

```
sort Car -by Year -order reverse numeric -by Make
```

The interactive Sort dialog may support multiple sort keys in a future version of LAPIS.

Sorting arbitrary regions in place is powerful, but also potentially dangerous. For example, it is easy to reduce a document to gibberish by sorting its words (Figure 8.6). More seriously, a user intending to sort a list of addresses by zip code might inadvertently select and sort the zip codes, instead of selecting the addresses and using the zip code as a sort key. Sorting just the zip codes in place would leave the addresses in the same order but reassign a different zip code to each address, resulting in corrupted data. This problem is not unique to LAPIS. In one case reported in the Boston Globe and `comp.risks`, a teacher sorted only one column in a spreadsheet of student grades, which scrambled the grades as a result [Lut00].

The Unix `sort` command is roughly equivalent to the LAPIS command `sort Line -order unicode`.

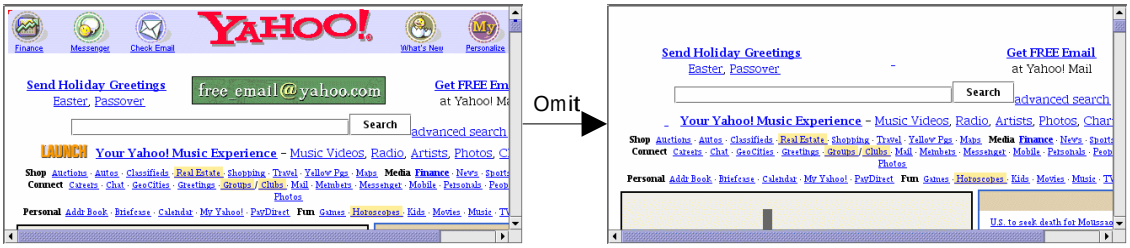


Figure 8.7: Images can be deleted from a web page by selecting them (with the `Image` pattern) and applying the Omit command.

8.1.3 Keep and Omit

The next two commands, Keep and Omit, are closely related, so this section describes both. The main use of these commands is *filtering* — removing undesired information from a document. Unlike the Extract command, the Keep and Omit commands filter information in place, preserving context outside the regions being filtered.

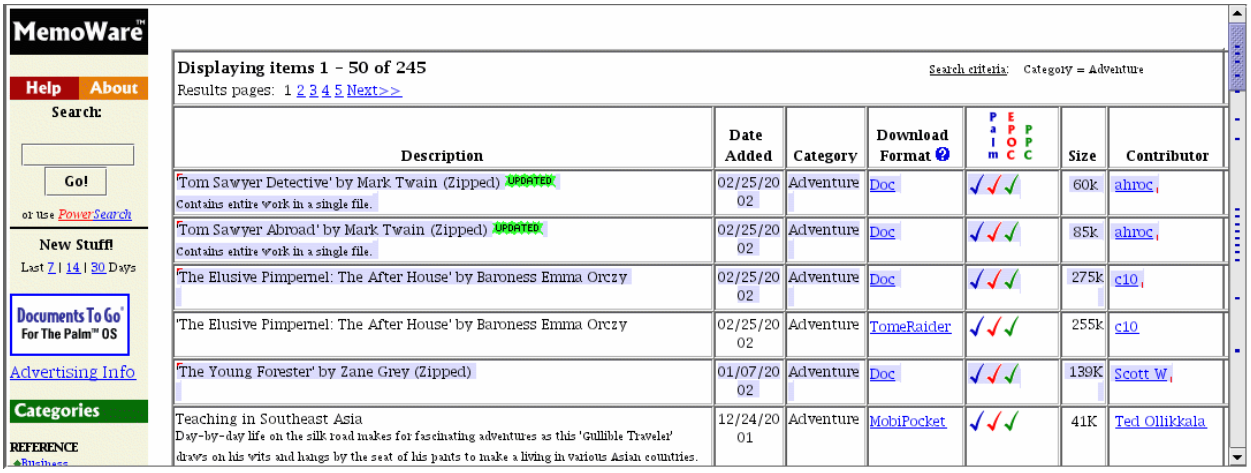
Omit is the easier of the two commands to understand. The Omit command produces a new document with the current selection deleted. For example, Figure 8.7 shows the result of omitting the images from a web page.

The Omit command requires the user to select the text that they *don't* want to see. Often, however, it makes more sense to select what you *do* want to see and filter everything else out. The Keep command is designed for this purpose. Keep does not simply delete all the text outside the selection, however.. Deleting all unselected text may delete important context, like table headings or data labels, which may be needed to understand the data that remains. Instead, Keep uses a simple inference technique to determine the set of *records* that the user is trying to filter. Then Keep deletes only the unselected records, preserving the selected ones and any context outside the records. Figure 8.8(b) shows an example of Keep applied to a web page.

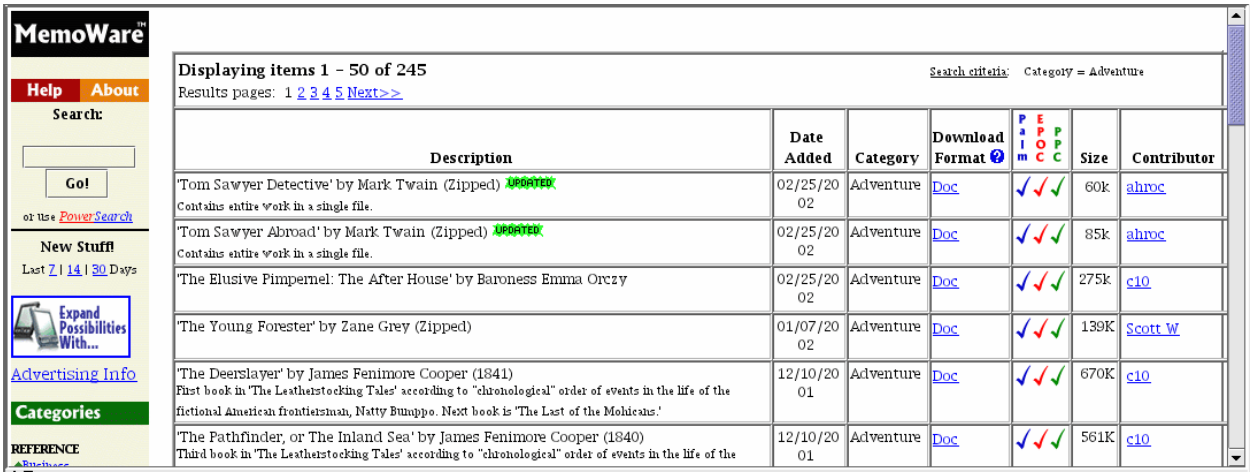
The record set used by Keep is inferred automatically by stripping constraints away from the TC pattern that created the current selection. (If the selection was created by the mouse, the appropriate record set cannot be inferred by this technique, and the Keep command is disabled.) As long as the pattern has the form *A op B* where *op* is either a relational operator, `and`, or `not`, the constraint *op B* is stripped away. Several examples of this reduction are shown below.

```
Word starting "s"          --> Word
     ending "e"

either Link or Image       --> either Link or Image
   just after Heading

from "<" to ">"            --> from "<" to ">"
   not containing "img"
```

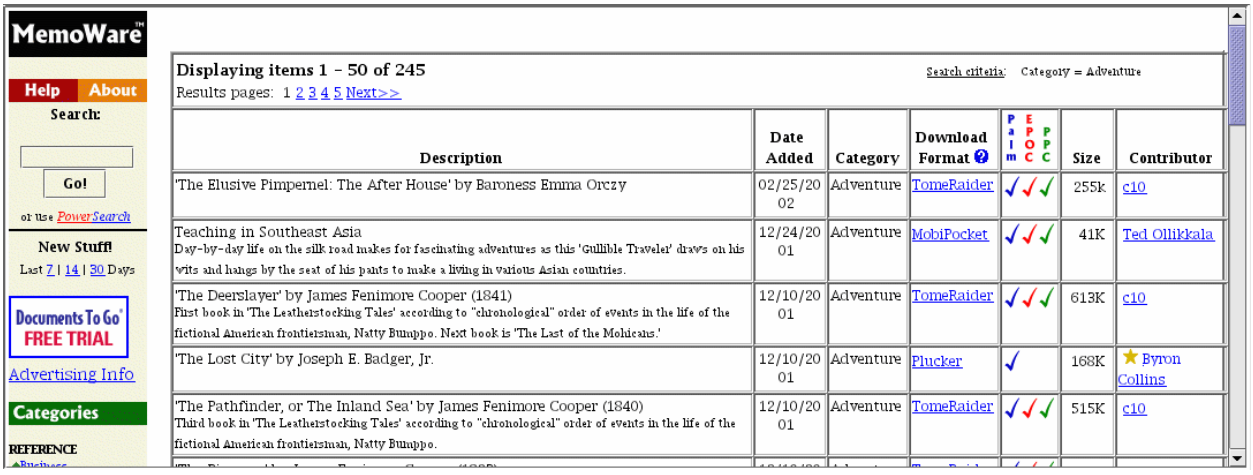
This inference technique is very simple, but it has the advantage of being more predictable than machine learning techniques like the selection inference algorithms described in Chapter 9. Often the record set is very simple, like `Line` or `Row` or `Method`. Sometimes, however, the record set



(a) Select `Book contains "Doc"`



(b) Invoke Keep command on (a)



(c) Invoke Omit command on (a)

Figure 8.8: Keep and Omit are complementary. Starting with the selection shown in (a), the Keep command keeps the selected books and deletes the rest (b), while the Omit command deletes the selected books and keeps the rest (c). `Book` is defined as `Row anywhere after "Description"`.

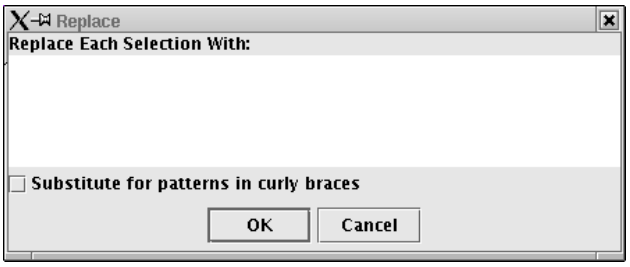


Figure 8.9: The Replace dialog.

needs to be described by a complex pattern. In this case, the user's selection pattern may include some constraints that define the record set and other constraints that select a particular subset of records. The Keep command cannot automatically discriminate among these constraints, so the best way to use Keep is to first define a record pattern and assign it a name, and then use that name in subsequent selection patterns.

When the selection pattern follows the form `A op B`, Keep and Omit are complementary operations. For example, applying Keep to the selection `Link containing EmailAddress` produces the same result as applying Omit to the selection `Link not containing EmailAddress`, and vice versa. Figure 8.8 contrasts the behavior of Keep and Omit on a selection.

The Keep and Omit menu commands have corresponding script commands:

```
keep pattern [-outof recordpattern]
omit pattern
```

By default, the `keep` command infers its record set using the same technique as the Keep menu command. The user can override the inference with the `-outof` option, which specifies the record set using another pattern. The `omit` command has no need for a record set.

No conventional Unix tools are precisely analogous to Keep and Omit. The closest equivalent might be `grep -v regexp`, which extracts lines that fail to match `regexp`. The LAPIS equivalent for this command would be, roughly,

```
omit {Line containing /regexp/}
```

or equivalently,

```
extract {Line not containing /regexp/}
```

8.1.4 Replace

The Replace command replaces every selected region with another string. Invoking Replace pops up a dialog box that prompts the user for replacement text (Figure 8.9). If the user types some text and presses OK, then the Replace command generates a new document in which every selection is replaced by the replacement text. The Replace dialog uses a multi-line text widget, so the replacement text may include embedded newlines.

The simplest kind of Replace substitutes the same string for every selection. If the user checks the "Substitute for patterns in curly braces" checkbox, however, then the replacement text is treated

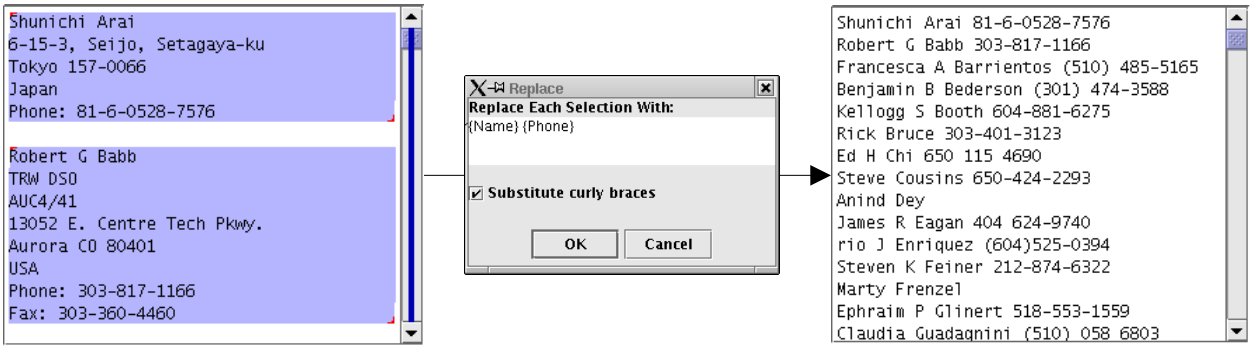


Figure 8.10: Replacing selections with a template. Every region in the selection on the left (made by the pattern `Paragraph`) is replaced by a template with two slots — one for the person's name (`Name`, defined as `first Line in Paragraph`) and the other for the person's phone number (`Phone`, defined as `from end of "Phone:" to start of Linebreak`). When a template pattern fails to match, e.g., when there isn't a phone number, that slot in the template is left blank.

as a *template*. The template may contain one or more slots for substitution, each denoted by a TC pattern surrounded by curly braces. Figure 8.10 shows an example of a template substitution. To perform the replacement, each selected region is searched for matches to the substitution patterns, and the text of the matches are filled into the template before replacing the selected region. If more than one region matches the substitution pattern, then the multiple matches are simply concatenated together. If no regions match the substitution pattern, then the slot is filled by the empty string. The substitution pattern `{all}` can be used to match the entire original region, so that the replacement template can specify text to insert before or after it. Literal curly braces in the replacement template must be escaped with backslashes, and backslashes must also be escaped.

The Replace dialog is clunky and largely unnecessary now that LAPIS supports multiple-selection editing (Section 7.5). Multiple-selection editing offers more interactive ways to do both kinds of replacement. To replace every selection with the same string, the user can simply type the string into the browser pane. By the rules of multiple-selection editing, this has the effect of deleting all the selections and replacing them with the typed string. To replace every selection with a template, the most interactive technique is simultaneous editing (Chapter 9), which effectively infers substitution patterns from the user's examples. Editing only works in the plain text view, however, so the Replace command is still the best option for replacements in the HTML view.

The Replace menu command has a corresponding script command:

```
replace pattern replacement
```

where *`pattern`* describes the regions to be replaced and *`replacement`* is the replacement template.

The closest Unix analog to Replace is the `sed` command, which allows the user to specify substitutions of the form

```
s/regexp/template/
```

Here, *template* may refer to parenthesized subexpressions of *regexp* by number, `\0` through `\9`, or to the entire string matched by *regexp* using the special character `&`. The Replace template provides similar functionality, except that the template specifies its own patterns for substitution, rather than referring back to subexpressions of the original pattern. Perl [WCS96] and awk [AKW88] provide a substitution command similar to `sed`'s.

8.1.5 Calc

The last command on the Tools menu is Calc. The Calc command attempts to interpret the selections as numbers and compute some statistics on them. By default, Calc produces an output document displaying all the statistics:

```
count = <number of numeric selections>
sum = <sum of selections>
average = <sum / count>
min = <minimum value of selections>
max = <maximum value>
stddev = <standard deviation of selections>
```

Nonnumeric selections are ignored, so the count only includes the selections that are recognizable as numbers. Statistics are computed using Java `doubles`, which are 64-bit IEEE floating point values.

The Calc command has a corresponding script command:

```
calc pattern
     [-count]
     [-sum]
     [-average|-mean|-avg]
     [-min]
     [-max]
     [-stddev]
```

in which *pattern* describes the regions that should be processed. If no options are specified, then `calc` returns all the statistics in the format shown above. If one or more options is specified, then `calc` only returns the requested statistics, separated by spaces. For example, the command

```
calc {Number just after "Price:"} -min -max -mean
```

might return

```
400 850 525.5
```

No conventional Unix command corresponds to Calc.

8.1.6 Count

There is no Count command on the Tools menu, because the number of matches to a pattern is displayed automatically. Whenever the user runs a pattern or changes the selection with the mouse, LAPIS displays the number of regions currently selected in the pattern pane (Figure 7.8).

The script command `count` does the same thing:

```
count pattern
```

In scripts, the `count` command is most useful for testing whether a pattern has any matches.

The closest Unix analog for this command is `grep -c regexp`, which counts lines that match a pattern. Another similar Unix utility is `wc`, which displays the number of words and lines in the document. The same effect can be achieved by `count Word` and `count Line`.

8.2 The Browser Shell

Most interactive interpreters use a *typescript* interface, in which command prompts are interleaved with command output in the same window. Lisp, Tcl, Python, and Unix shells like `sh` and `csh` all follow this interaction model.

In contrast, LAPIS integrates its interpreter directly into the browsing window, a new idea called a *browser shell*. A browser shell is different from a typescript shell in three ways:

- **Commands are typed into the Command bar.** The Command bar in LAPIS fills a dual role. Like the Location box in other web browsers, it displays the URL of the web page or file in the browser pane. However, the user can also type script commands into it.

- **The browser pane acts as standard input and output for commands.** When a command is executed, it takes its input from the current document in the browser pane, which may be a file, a web page, or the output of a previous command. After executing, the command's output is sent back to the browser pane as a new page, where the user can view it as either plain text or rendered HTML, scroll through it, edit it, or apply further commands.

- **Executed commands appear in the browsing history.** The Forward and Back browsing buttons navigate through command output pages as well as web pages. When the browsing history is viewed as a list, both URLs and executed commands appear in the list. Portions of the history list can be extracted and saved as a script for later reuse.

The remainder of this chapter discusses the details and implications of the LAPIS browser shell.

8.3 URLs as Commands

Either a URL or a script command may be typed into the Command bar. LAPIS takes this integration to its logical conclusion, and simply defines URLs as commands in its dialect of Tcl. All of the following are valid commands in LAPIS:

```
http://www.yahoo.com/
ftp://ftp.cs.cmu.edu/afs/cs/project/amulet
file:/home/rcm
```

The result of a URL command is the page or file loaded from that URL. File or FTP URLs that refer to directories return a page listing the files in the directory.[3] Regarding URLs as commands has two advantages. First, it simplifies the implementation of the Command bar, so that it can treat everything typed into it as a script command. Discrimination between URLs and other script commands happens at a lower level. Second, and more important, URLs can be used directly in scripts. For example, here is a simple script that extracts the Pittsburgh weather forecast from Yahoo:

```
http://weather.yahoo.com/forecast/USPA1290_f.html
extract {Table just after "5 day forecast"}
```

URL commands are implemented using Tcl's unknown-command feature. When Tcl encounters a command name that doesn't exist, it invokes `unknown` instead, passing the unknown command's name and arguments. In LAPIS, `unknown` checks whether the unknown command can be parsed as a URL. If so, the URL is fetched and its contents are returned as the result of the command.

LAPIS also treats filenames as commands. Thus these are all commands:

```
/etc/passwd
../../index.html
fruits.txt
```

Like a URL, a filename command loads the file into LAPIS. Because filename commands are also implemented with the Tcl unknown-command feature, files with the same name as a Tcl command cannot be loaded this way unless they are explicitly disambiguated using the `file:` prefix. For example, to load a file named `extract`, which is ambiguous with the LAPIS built-in `extract` command, the file command must be

```
file:extract
```

Filename commands may also be ambiguous with external programs (Section 8.6), in which case the `file:` prefix can again be used to disambiguate.

Relative filenames and URLs are resolved relative to the current directory. By default, the current directory is the directory of the last file loaded. The user can also change the current directory explicitly using the Tcl `cd` command. The current directory is used not only for resolving relative filenames, but also to invoke external programs, and as the starting directory for the File/Open and File/Save dialog boxes.

[3] The format of this page is not well-defined, unfortunately. LAPIS uses the Java library to load URLs, and the Java library has changed the format of the listing returned by directory URLs from one release of Java to the next.

8.4 Self-Disclosure in the Command Bar

The Command bar is also used for *self-disclosure* [DE95]. Self-disclosure is a technique for helping users learn more about a powerful user interface, particularly about features which are otherwise hidden, such as scripting language commands and syntax. Self-disclosure has been used in other systems to teach about keyboard shortcuts (Lyx [Lyx02]) and script commands (Emacs [Sta81], AutoCAD [Aut02], Chart N' Art [DE95]). In LAPIS, self-disclosure is used to expose the user to both the scripting language and the pattern language.

The Location box in other web browsers is already used for a kind of self-disclosure, because it continually tracks the URL of the current web page. When the user clicks on a hyperlink to browse to another page, the Location box is updated to show the new page's URL. In a sense, the Location box is disclosing a command — the URL — that can be used to achieve the same effect as the user's graphical interaction — a click on a hyperlink. Self-disclosure is probably not as valuable for URLs as it is for other kinds of commands, since there is no consistent "URL language" across web sites that would help a user generate a URL from scratch. Nevertheless, users do take advantage of the disclosed URL for saving bookmarks or sending web pages to other people via email or instant messaging.

In keeping with the unification of URLs and other script commands, LAPIS not only discloses URLs in its Command bar, but also script commands. When the user invokes a text-processing command from the Tools menu, a script command that would have had the same effect is displayed in the Command bar. For example, suppose the user selects all the words in the document using the pattern `Word`, and then invokes the Extract command to extract them. Then the command bar would display

```
extract Word
```

More examples of disclosed commands are shown in Table 8.11.

The arguments for a disclosed command are determined as follows. If the selection was created by running a pattern (like `Word`), then that pattern is used as the argument for the command. If any part of the selection was made by the mouse, then `Selection` is used as the argument for the command. If the user provided additional arguments to the menu command using a dialog box, then the corresponding arguments are included in the disclosed script command.

Self-disclosure is used in other parts of LAPIS as well. For example, clicking on a named pattern in the library pane shows in the pattern pane how the name would be referenced in a TC pattern (Section 7.4.3). Also, when LAPIS infers selections from examples, it discloses a TC pattern corresponding to the inferred selection (Chapter 9).

8.5 Command Input and Output

By default, commands take their input from the *current document*, which is the document displayed in the browser pane. After a command runs, its result becomes the new current document, and the original document is added to the browsing history. As a result, a sequence of commands implicitly forms a pipeline, with the output of one command becoming the input for the next. For example, this command sequence picks a random word starting with "a" from the list of words in `/usr/dict/words`:

Figure 8.2(a)	`extract Word`
Figure 8.2(b)	`extract Sentence`
Figure 8.2(c)	`extract Link`
Figure 8.5(a)	`sort Word`
Figure 8.5(b)	`sort Paragraph -by {last token in first line in paragraph}`
Figure 8.5(c)	`sort Row -by {last cell in row} -order numeric`
Figure 8.7	`omit Image`
Figure 8.8(top)	`keep {Book contains "Doc"}`
Figure 8.8(bottom)	`omit {Book contains "Doc"}`
Figure 8.10	`replace Paragraph {{Name} {Phone}}`

Figure 8.11: Commands disclosed for earlier examples in this chapter.

```
file:/usr/dict/words
extract {Word starting "a"}
sort Word -order random
extract {first Word}
```

The input for a command can be redirected from a different source by passing it as the last argument to the command. For example, this command extracts the words starting “a” from the given string:

```
extract {Word starting "a"} "apple banana pear apricot"
```

The result is

```
apple
apricot
```

since the `extract` command uses linebreaks as its default terminators. All the script commands described in Section 8.1 take an optional last argument specifying the input string to process.

The output of a command can be redirected using Tcl subexpression syntax `[`*`expr`*`]`. For example,

```
set dictionary [file:/usr/dict/words]
```

stores the document loaded from `/usr/dict/words` into the Tcl variable `dictionary`. A pipeline can also be expressed more verbosely by stacking up subexpressions, as in:

```
sort Word -order random \
     [extract {Word starting "a"} \
              [file:/usr/dict/words]]
```

(The backslashes in this command are Tcl syntax allowing a long command to be continued to the next line.)

When the LAPIS interpreter evaluates a nested context — a subexpression in square brackets — it automatically pushes the current document onto a stack, restoring it after evaluating the nested context. This behavior allows pipelines to be encapsulated in Tcl procedures and used in other pipelines. For example, the weather-fetching script shown earlier might be wrapped in a procedure called `get-weather`:

```
proc get-weather {} {
    http://weather.yahoo.com/forecast/USPA1290_f.html
    extract {Table just after "5 day forecast"}
}
```

and then used in a pipeline that inserts the weather forecast into the user's home page:

```
http://www.cs.cmu.edu/~rcm/personal-homepage.html
replace {"Weather Forecast"} [get-weather]
```

Note that the result of a Tcl procedure is the result of the last command in its body, so `get-weather` returns the weather forecast without needing an explicit `return` command. This example assumes that the user's home page contains the words "Weather Forecast" somewhere in it, as a placeholder for the weather forecast.

8.6 External Programs

LAPIS can also run an external command-line program from the Command bar. Like URL evaluation, this feature is implemented using Tcl's unknown-command facility. If a command name is not found as a built-in Tcl command or user-defined procedure, and cannot be parsed as a URL, then LAPIS searches for an external program by that name. For example, typing the command

```
ls
```

would display a listing of the files in the current directory (at least on systems where `ls` is available). If an external program has the same name as a Tcl command, then the Tcl command takes precedence. Thus, `sort` refers to the LAPIS sort command, not the system sort program. The user can force the external program to run instead by using the `exec:` prefix. Thus

```
exec:sort
```

runs the external `sort` program instead of the LAPIS `sort`. The `exec:` prefix is intended to be analogous to other URL protocol prefixes like `http:` and `file:`.

Like other commands, an external program automatically takes its input from the current document, and its output becomes the new current document in the browser pane. Thus, external programs can participate in pipelines. For example, if the user types (on BSD-style Unix)

```
ps -aux
```

then the browser displays a list of running processes. If the next command is

```
grep xclock
```

then the process listing is filtered to display only those lines containing `xclock`. (The same effect could be achieved by the LAPIS command `extract {Line containing "xclock"}`.)

To make this work with legacy programs like `ps` and `grep`, the external program is invoked in a subprocess with its input and output redirected. Standard input is redirected from the current page of the browser (passing the HTML source if the current page is a web page). Standard output is redirected to a new document displayed in the browser pane, which is displayed incrementally as the program writes its output. Standard error is redirected to a special subframe in the browser pane, in order to separate it from the standard output. Figure 8.12 shows the resulting output for two different invocations of `ls`. The standard error pane normally remains hidden until an external program prints to it.

Program output may be parsed and manipulated like any other page in LAPIS. For example, `ps -aux` displays information about running processes in a tabular form:

```
USER    PID %CPU %MEM SIZE RSS TTY...
bin     160  0.0  0.4  752 320  ? ...
daemon  194  0.0  0.6  784 404  ? ...
rcm     294  0.0  1.0 1196 660  ? ...
```

This output can be described by TC patterns entered interactively in the pattern pane.

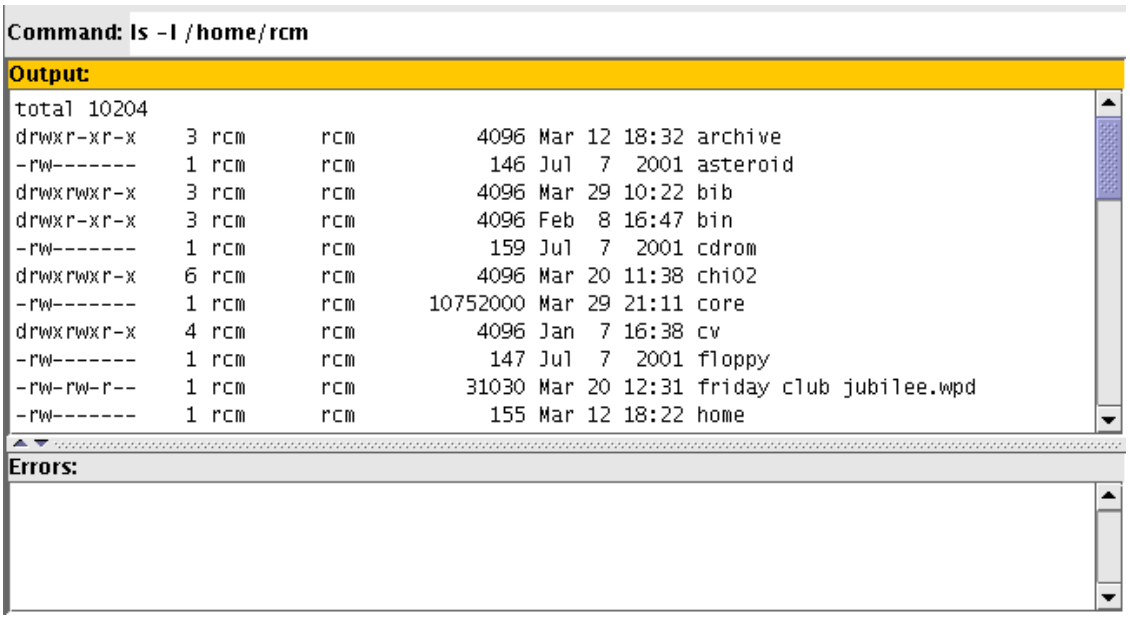
```
Process is Line not 1st Line
User is Word starting Process
PID is Number just after User
```

In a script, TC patterns can be run by the `parse` command, which is handy for defining a group of named patterns for use in subsequent commands. The `parse` command takes one argument, which is a TC expression or group of TC expressions:

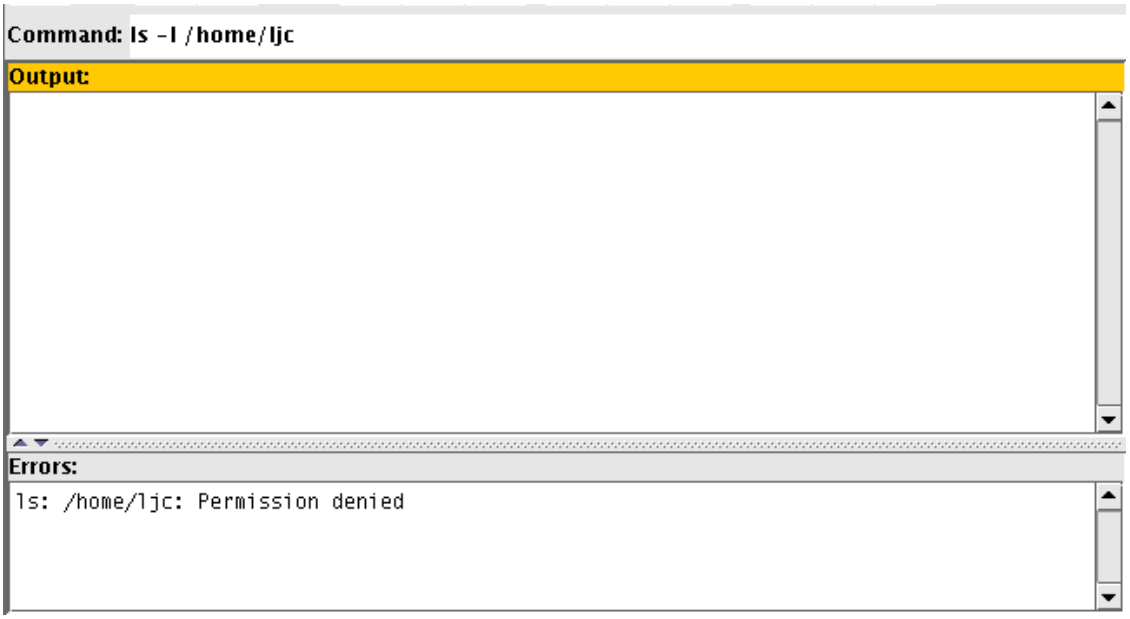
```
parse {
    Process is Line not 1st Line
    User is Word starting Process
    PID is Number just after User
}
```

These named patterns can then be used with LAPIS commands that search and manipulate `ps` output:

```
# sort processes by PID
sort Process -by PID -order numeric

# display only xterm processes
keep {Process contains "xterm"}

# kill all xterms
eval kill [extract {PID in Process contains "xterm"} -as tcl]
```



(a)



(b)

Figure 8.12: An external program's standard output and standard error streams are displayed in separate panes. (a) a successful command that prints only to standard output; (b) an unsuccessful command that prints an error message to standard error.

Note that `eval` is needed in the last example because `extract` may return a list of process IDs. Without `eval`, this list would be passed to `kill` as a single argument with embedded spaces; with `eval`, the list is exploded into separate arguments.

Normally, standard output becomes the current document for subsequent patterns and commands. The user can choose to process the standard error instead by clicking on the standard error pane to make it active. The same switch can be made in a script using the command

```
property stderr
```

which returns the standard error property of the current document. The `property` command accesses document metadata, explained in more detail below (Section 8.13).

8.7 The Browser Shell as a System Command Prompt

Since the LAPIS browser shell can invoke external programs, it is worth comparing it to the command prompt already available in operating systems like Unix, Windows, and Mac OS X. In a sense, the browser shell continues a trend which has seen the web browser become more and more tightly integrated into the desktop interface. Modern web browsers like Microsoft Internet Explorer and KDE Konqueror [KDE02] already include file management among the browser's responsibilities. Integrating the system command prompt is another step along the same path, a step which makes some sense because file management and command execution are often intertwined. Recognizing this fact, Konqueror also integrates a command prompt, which appears as a typescript shell in an optional pane at the bottom of the window (Figure 8.13).

The LAPIS browser shell interface behaves differently from a traditional typescript shell like Konqueror's, however. Whereas a typescript shell interleaves commands with program output in the same window, a browser shell separates the command prompt from program output. The browser shell also automatically redirects program input from the current browser page, and automatically sends program output to a new browser page.

One effect of these differences is on scrolling. In a typescript interface, long output may scroll out of the window. To view the start of the output, the user must either scroll back, or else rerun the command with output redirected to a command like `more` or `head`. The browser shell, by contrast, initially displays the *first* windowful of output, rather than the *last*, reducing the need for scrolling. When output is less than a windowful, a typescript shell can become cluttered by outputs of several commands, forcing the user to scan for the start of the latest output. The browser shell displays each program output on a new, blank page. In essence, the overall effect of the browser shell is like automatically redirecting the output of every command to `more`.

Unlike `more`, however, the browser shell's display is not ephemeral. The displayed output can be passed as input to another command, which allows pipelines to be assembled more fluidly than in the typescript interface. Developing a complicated Unix pipeline, such as `ps ax | grep xclock | cut -d ' ' -f 1`, is often an incremental process. In typescript interfaces, where input redirection must be specified explicitly, this process typically takes one of two forms:

- Repeated execution: run `A` and view the output; then run `A|B` and view the output; then (if `B` turned out wrong) run `A|B′` and view the output; etc. This strategy fails if any of the commands run slowly or have side-effects.

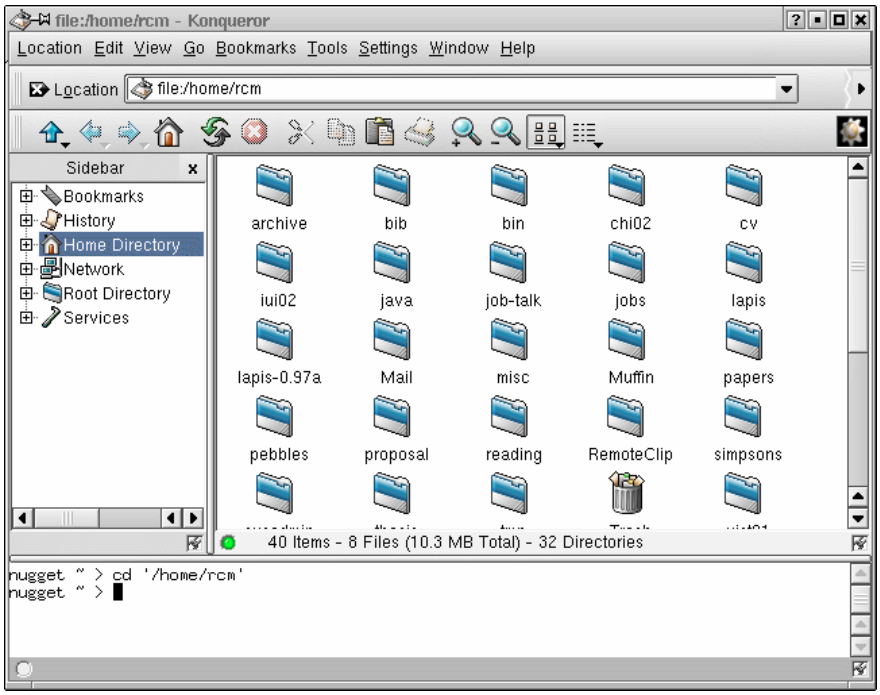


Figure 8.13: The Konqueror web browser/file manager [KDE02] can include a terminal pane (at the bottom of the window) for executing commands.

- Temporary files: run `A > t1` and examine `t1`; then run `B < t1 > t2` and examine `t2`; then (if `B` was wrong) run `B′ < t1 > t2`, etc.

The browser shell offers a third alternative: run `A` and view the output; then run `B` (which automatically receives its input from `A`) and view the output; then press Back (because `B` was wrong) and run `B′` instead. The browser shell displays each intermediate result of the pipeline while serving as automatic temporary storage.

Automatic input redirection makes constructing a pipeline very fluid, but it is inappropriate for programs that use standard input for interacting with the user, such as `passwd`. Such programs cannot be run directly in LAPIS. One solution is to pop up a terminal emulator in a different window, e.g.:

```
xterm -e passwd
```

In the future, it might be useful to embed a terminal emulator in LAPIS.

Another problem with the browser shell model is the linear nature of the browsing history. If the user runs `A`, backs up, and then runs `A′`, the output of `A` disappears from the browsing history. To solve this problem, LAPIS lets the user duplicate the browser window, including its history, so that one window preserves the original history while the other window is used to backtrack. (Netscape's New Window command used to work similarly before version 4.0.) A more complex solution might extend the linear browsing history to a branching tree, an idea which has been explored in at least one research web browser [AS95].

8.8 Page History vs. Command History

The browser shell actually has two distinct histories: a *page history* and a *command history*. Both histories are found in other web browsers, but only a command history is found in most typescript shells.

The *page history* consists of files, web pages, and command outputs. The current item in the page history is displayed in the browser pane. The user navigates through the page history using the Back and Forward buttons. The Go menu presents a listing of the page history, with each page identified by the URL or command that generated it. Clicking on a page in this list jumps directly to it. The page history in LAPIS corresponds to the browsing history in other web browsers.

The *command history* is a history of the commands that have been typed into the Command bar, made available as a drop-down menu under the Command bar. The user can pick an earlier command from the menu, edit it as necessary, and execute it again. The command history is analogous to the URL history in other web browsers, which is similarly rendered as drop-down menu under the Location box. Command history is also found in typescript shells, where it is usually accessed by pressing the up and down arrows on the keyboard.

As long as the user is simply typing commands, the page history and command history remain synchronized. Each time a command is entered, the command's output is added to the page history, and the command itself is added to the command history. For example, suppose the user has run two commands:

Command History	Page History
`http://www.cnn.com`	`http://www.cnn.com`
`extract Image`	`extract Image`

If the user clicks Back to undo the last command and run a different one, however, the two histories become unsynchronized. The new command is *added* to the command history, but its output *replaces* the output of the old `extract` in the page history:

Command History	Page History

`http://www.cnn.com`
`extract Image`
`extract {Image anywhere after "News"}`
	`http://www.cnn.com`
	`extract {Image anywhere after "News"}`

The page history only records the sequence of commands that produced the current page. Any commands that were "undone" by Back are not included in the page history.[4]The command history records all commands that have been executed, whether or not they were retracted by Back.

The two histories also differ in size. By default, LAPIS only keeps the last 20 pages in the page history, since pages may be large and the current version of LAPIS stores all pages in RAM. If LAPIS were a full-fledged web browser, however, the page history would simply be part of the browser cache, with some pages in RAM and some on disk, and old pages evicted as necessary. The command history can be much longer than the page history with little impact on memory use. LAPIS limits the command history to 100 commands to keep the drop-down menu at a manageable length.

[4]Saying that Back "undoes" commands is not strictly accurate, of course — LAPIS does not undo command side-effects like setting Tcl variables or changes the filesystem.

8.9 Generating Scripts from the Page History

The page history can be used to generate a script from a command session. The Convert History to Script command on the Scripts menu pops up a script editing window with the commands from the page history loaded into it. The example above would generate the script

```
http://www.cnn.com
extract {Image anywhere after "News"}
```

Self-disclosure helps here, because it labels all pages in the page history with a script command, even pages generated by menu commands. Thus the user can *demonstrate* a script using menu commands, although no inference is done to generalize the script automatically. The user can explicitly request inference by entering selection guessing mode; more about this can be found in Chapter 9.

The generated script usually requires some editing, because it includes the entire page history of the current LAPIS window. Old commands irrelevant to the desired script must be deleted manually. The Demonstrate menu command described below (Section 8.11), solves this problem by starting a script demonstration in a fresh window with an empty page history.

Currently, LAPIS has no convenient mechanism for naming the generated script or making it persistent. At the moment, the typical solution is to wrap the script in a Tcl procedure, e.g.:

```
proc cnn-pictures {} {
    http://www.cnn.com
    extract {Image anywhere after "News"}
}
```

If the script is only needed temporarily, then it is sufficient to use the Run command in the script editing window to define `cnn-pictures` in the Tcl interpreter for the duration of the session. To make the script persistent, it must be added to the user's Tcl startup script (whose location is defined in the Preferences dialog). Ideally, this step should be much easier. For example, scripts saved to a particular directory might automatically become available as script commands.

Scripts can be added to the LAPIS toolbar using the `toolbar` command. For example,

```
toolbar CNN cnn-pictures
```

adds a button labeled CNN to the toolbar which invokes the `cnn-pictures` procedure defined above.

8.10 Web Automation

Since LAPIS is a web browser, it includes commands for simple kinds of automated web browsing, so that a script can navigate through a dynamic web site or online transaction. Web browsing has two basic actions: clicking on hyperlinks and submitting forms. Automated web browsing requires equivalent script commands for each of these actions.

Clicking on a static link, whose target doesn't change over time, has the same result as typing the URL into the Command bar. Thus the script command for clicking on a static link is simply the link's URL, such as:

```
http://weather.yahoo.com/
```

For some links, however, the target URL varies depending on when the page is viewed. Variable links are often found in online newspapers, for example, where links to top stories change from day to day. The `click` command can be used to click on a variable link by describing its location in the web page with a pattern. For example, this script clicks on the top story in Salon:

```
http://www.salon.com/
click {Link after Image in 3rd Cell in Row}
```

The `click` command throws a Tcl exception if its pattern does not match exactly one hyperlink on the page.

For entering data into forms, LAPIS provides the `enter` command. The `enter` command takes two arguments: the first is a pattern describing the form field to modify, and the second is the value to enter in the field. For text fields, this value is a string which is entered in the field directly. For menus or lists, the value is selected in the list. For radio buttons or checkboxes, the value should be "on" or "off" (or yes/no, true/false, or 1/0).

Forms are submitted either by the `submit` command, which uses the default submit button on the form, or by a `click` command specifying which button to click. For example, here is a script that searches Google for the LAPIS home page and clicks the I'm Feeling Lucky button to jump right to it:

```
http://www.google.com/
enter Textbox {LAPIS Lightweight Structure}
click {Button contains view source "I'm Feeling Lucky"}
```

(Note that `view source` is needed here because of a LAPIS bug: the button's label, "I'm Feeling Lucky", is not searchable in the rendered HTML view.)

The examples presented so far have been web-site-specific, but some browsing tasks are sufficiently uniform across web sites to be handled by a generic script. For example, the following script can log into many web sites, assuming the user's login name and password have been stored in the Tcl variables `id` and `password`:

```
enter {
  Textbox just after Text containing either "login"
                                          or "email"
                                          or "id"
                                          or "user"
} $id
enter {
  Textbox just after Text containing "password"
} $password
submit
```

8.11 Web Automation by Demonstration

Combining web automation with script generation from the history makes it possible to create web browsing scripts by demonstration. An early version of LAPIS included this capability [MM00], but it has since ceased to work because of changes in other parts of LAPIS. This section describes the feature as it originally worked, and explains why it has become broken.

To create a browsing script quickly, the user can *demonstrate* it by recording a browsing sequence. The demonstration begins with an arbitrary example page, the *input page*, showing in the browser. Invoking the Demonstrate Script menu command pops up a new browser window, in which the browsing demonstration will take place. A new window is created so that the browsing sequence can refer to the input page for parameters. Like any LAPIS browser window, the Demonstrate window records a browsing history: URLs visited and commands typed. Unlike a normal browser window, however, the Demonstrate window's history also records user events in form controls. For example, if the user types into a form field, the history records an equivalent `enter` command.

To fill in a form with text from the input page, the user can make a selection in the input page, then copy and paste it to a form field in the Demonstrate window. If the copied text was selected by searching for a pattern, then this action records the command `enter` *`field-name pattern`* in the history (where *`field-name`* is the name given to the form field in the HTML). If the copied data was selected manually, then the command `enter` *`field-name`* `Selection` is recorded in the history, so that when the script is run at a later time, `Selection` will return the user's selection at that time. More complex dependencies can be expressed by typing a Tcl command instead of pointing-and-clicking. For example, if a radio button should be selected only if the input page has certain features, then the user might type the command `if {[count` *`pattern`*`]} {click` *`field-name`*`}`.

Using Back and Forward, the user can revise the demonstration as necessary until the desired results are achieved. When the user is satisfied with the demonstration, the Demonstrate window is closed and the history is generated as a script.

Other systems have provided web automation by demonstration, notably LiveAgent [Kru97], which records a sequence of Netscape browsing actions as a macro. LAPIS demonstrations have two advantages over LiveAgent. First, the recorded transcript is represented by the browsing history, which is visible, familiar, and easy to navigate. A crucial part of making this work is that LAPIS inserts commands as well as URLs in the browsing history. Second, an experienced user can generalize the demonstration on the fly by typing commands at crucial points instead of pointing-and-clicking. Since LAPIS embeds a full scripting language, the resulting scripts can be significantly more expressive than recorded macros, without taking much more time to develop.

The Demonstrate feature is currently broken because the *`field-name`* patterns used to identify form fields are outdated. Earlier versions of LAPIS followed a model in which *all* parsers were run on a page automatically, so that the HTML parser could create document-specific identifiers like *`field-name`* for each form field found in a page. This model doesn't scale well as the library of patterns and parsers grows. Instead, the current version of LAPIS only runs parsers on demand, as described in Section 6.2.4. This model is far more efficient, but it means that all the named patterns in the library must be document-independent — parsers cannot generate document-specific named patterns on the fly. In the new model, instead of simply using *`field-name`*, Demonstrate would have to describe each form field as follows:

```
Control
  contains name-attr
             contains AttributeValue
                       contains view source "field-name"
```

A better solution would be to add parameterized named patterns to TC, so that one can define a parameterized pattern `Control(`*`field-name`*`)` as the pattern above and refer to named form fields conveniently. Fixing Demonstrate, and adding parameterized named patterns, are both left for future work.

8.12 Other Tcl Commands

Built-in Tcl commands can also be entered in the Command bar. For example, the `expr` command performs arithmetic computation:

```
expr 5*5+2
```

This command returns 27 (as a string, the only data type in Tcl).

Unlike other LAPIS commands and external programs, built-in Tcl commands do not automatically use the current document for input and output. To give these commands access to the current document, LAPIS provides the `doc` command. Writing `doc` with no arguments simply returns the current document, so

```
string length [doc]
```

returns the length of the current document.

Writing `doc` *`string`* sets the current document to *`string`*. For example:

```
doc "This is the new document"
```

Both forms of `doc` can be used in a single Tcl command. For example, if the current document consists of a single number, the following command doubles it and sets the current document to the result:

```
doc [expr 2 * [doc]]
```

When a Tcl command is used in a pipeline, `doc` must be used to read and write the current document. For example, here is a pipeline that extracts the words from the current document, converts them all to lowercase using the built-in Tcl command `string tolower`, and then sorts them:

```
extract Word
doc [string tolower [doc]]
sort Word
```

At first, the `doc` command may seem like an unnecessary complication. Why not just modify the Tcl interpreter so that *all* Tcl commands set the current document — not only LAPIS extension commands like `extract` and `sort`, but also built-in commands like `string`? Then the pipeline above might be written instead as:

```
# this example won't work in LAPIS
extract Word
string tolower [doc]
sort Word
```

This approach does not entirely eliminate `doc`, because it is still needed to specify the current document as an argument to `string`. Worse, however, this approach turns out to be unworkable because *all* Tcl commands return a string, including control structures like `while`, `foreach`, and `proc`. These commands always return the empty string, but are otherwise indistinguishable from other Tcl commands.[5] If the result of every Tcl command became the new current document, then it would be impossible to use control structures in a pipeline without clearing the current document.

When entering commands interactively, however, preceding every native Tcl command with `doc` is tedious. LAPIS provides an automatic shortcut. When a Tcl command typed into the Command bar returns a nonempty result, regardless of whether it used `doc`, the result becomes the current document and is displayed in the browser pane. Thus, entering the command

```
expr 5*5+2
```

would display `27` in the browser pane. When an interactive command returns the empty string, however, LAPIS does not change the current document, but instead pops up a dialog box indicating that "The command returned no data." LAPIS treats empty results differently from nonempty results because many Tcl commands always return empty strings, and it seems unproductive to clear the browser pane when one of these commands is issued. This behavior also matches the behavior of many web browsers, which display a message like "The server returned no data" rather than an empty page. The user can override this behavior by explicitly passing the result of the Tcl command to `doc`, which always sets the current document even if the result is empty.

8.13 Document Metadata

A LAPIS document is essentially a string of content augmented with some metadata represented by name-value pairs. The current document's properties may be accessed by the `property` command. For example,

```
property url
```

returns the current document's `url` property. The command

```
property -set base http://www.cs.cmu.edu/
```

sets the current document's `base` property. The command

```
property -list
```

[5] In fact, this property is one of the appealing features of Tcl, since it allows users to define new control abstractions in the same way that they would define new procedural abstractions.

returns a Tcl list of all the property names defined on the current document.

LAPIS defines several standard properties:

- `content-type`: the document's MIME type. Typical values are `text/plain` for plain text and `text/html` for HTML.
- `command`: the command that created the document.
- `url`: the URL from which the document was loaded.
- `base`: the URL to which relative links refer, initialized from the HTML `<base>` tag.
- `process`: the `java.lang.Process` object that represents the process that generated the document.
- `stdout`: the standard-output document, if the document was created by a process.
- `stderr`: the standard-error document, if the document was created by a process.

Metadata properties are used in many ways in the LAPIS user interface. When a document is first displayed in the browser pane, its `content-type` property helps determine whether it should be shown as rendered HTML or plain text. When a relative URL appears in a hyperlink or inlined image (such as `<img src=/images/go.gif>`), the `base` and `url` properties are used to resolve it into an absolute URL. Documents are listed in the browsing history by their `command` property (or `url` property, if the `command` property is not found).

LAPIS commands automatically copy the `base` and `content-type` properties from the input document to the output document. The only exception to this rule is the `extract -as` *`type`* command, which sets the `content-type` of its output document to match *`type`*. If the input document lacks a `base` property, then the input document's `url` property is copied to the output document's `base` property. As a result of these rules, when a LAPIS command is applied to a web page, the output page will always be displayable as a web page with valid hyperlinks and inlined images, even if the URLs are relative.

When a LAPIS document is converted to a string, however, its metadata is lost. There are two ways this can happen: when the document is saved to a file, which only saves its content; and when the document is processed by other Tcl commands, which treat it as a string. The `relocate` command allows the `base` property to survive this conversion. Running

```
relocate
```

takes the current document's `base` property (or `url` property, if no `base` is found) and creates a new document with an explicit `<base>` tag inserted in the document content.

Another solution to this problem would have all LAPIS commands transform the content so that hyperlinks and inlined images are automatically preserved. This could be done either by adding the `<base>` tag automatically (essentially running `relocate` automatically after every LAPIS command), or by converting all relative URLs into absolute URLs. This solution would obviate the need for the `base` metadata property. This approach seems too draconian, however. It effectively hardcodes an absolute location for the web page, which is undesirable if the user is editing a web

page that might be visited through different URLs (e.g., the user might access it by a `file:` URL for editing, but web site visitors would use an `http:` URL). LAPIS adopts the philosophy that the content of documents should be under the user's control, and LAPIS commands should not change document content unless explicitly directed to do so.

The loss of the `content-type` property is not as serious, because it can usually be guessed. For example, when LAPIS loads a document from a filename ending `.htm` or `.html`, it infers that the document has content-type `text/html`. To guess the content type of a string obtained some other way, e.g., from the output of a Tcl command or an external program, LAPIS examines the beginning of the string for either `<html>` or `<!doctype html` (ignoring case and whitespace). If either of these strings is found, then LAPIS guesses content type `text/html`; otherwise, it assumes `text/plain`.

8.14 Command Line Invocation

LAPIS scripts can be invoked by external programs using the LAPIS command line interface. The design of this interface was inspired by the command line interfaces of awk and Perl.

The LAPIS command line interface is a batch interface, which does not display any GUI. It accepts input from command line arguments, files, and/or standard input, and sends output to standard output or files. To use the interface, the caller invokes LAPIS with three kinds of command-line arguments:

- the *invocation mode*, which can be any one of `-pipe`, `-inplace`, `-batch`, or `-argv`. If the mode is omitted, the default mode is `-pipe`. The modes are defined below.
- a script to run, which be specified directly on the command line (`-e` *`script`*) or loaded from a file (`-f` *`scriptfile`*).
- zero or more additional arguments, whose interpretation depends on the invocation mode. For most modes, these arguments are files or URLs; for the `-argv` mode, they are arbitrary string arguments that can be accessed by the LAPIS script.

The invocation modes are explained below.

Pipe Mode

Using one of the following command lines invokes LAPIS in pipe mode:

```
lapis -pipe -e script      file/URL file/URL ...
lapis -pipe -f scriptfile  file/URL file/URL ...
```

In response, the LAPIS commands in *`script`* or loaded from *`scriptfile`* are run on every file or URL listed on the command line, and the resulting documents are printed to standard output. To be precise, for every *`file/URL`*, LAPIS runs the following commands:

```
file/URL     # load the file or URL
script       # run the user's script
puts [doc]   # print result to standard output
```

For example, the following command prints all the comments in a set of Java source files:

```
lapis -pipe -e "extract Comment" *.java
```

Note that pipe mode is the default invocation mode, so `-pipe` can be omitted. This example prints the current price of Lucent stock:

```
lapis -e "extract {Number in Bold in Table starting 'Symbol'}" \
      "http://finance.yahoo.com/q?s=lu&d=v1"
```

If no *`file/URL`* arguments are specified, then pipe mode takes its input from standard input. This allows LAPIS to be used as a filter in a Unix pipeline:

```
ps -aux | lapis -e "sort Process -by CPUShare \
                                 -order reverse numeric"
```

Standard input may also be specified explicitly on the command line as `-` (dash).

Pipe mode is analogous to the default modes of awk and Perl, which apply the user's script to every file on the command line.

Inplace Mode

When LAPIS is invoked one of the following ways:

```
lapis -inplace -e script      file file ...
lapis -inplace -f scriptfile  file file ...
```

then LAPIS runs the script on every file listed on the command line, and saves the resulting document back to the same filename. The effect is the same as this sequence of commands:

```
file       # load the file
script     # run the user's script
save file  # save back to the same filename
```

The `save` command automatically creates a backup file named *`file`*`~`. At present, inplace mode only works on files, not `http:` or `ftp:` URLs.

Inplace mode is useful for performing a global find-and-replace across a group of files. For example, the following command would replace identifiers named `fetch` with `get` across a set of Java files:

```
lapis -inplace -e "replace {Identifier = 'fetch'} get" *.java
```

Perl also provides in-place processing using its `-i` option.

Batch Mode

Batch mode is similar to pipe mode in that it takes zero or more files or URLs:

```
lapis -batch -e script      file/URL file/URL ...
lapis -batch -f scriptfile  file/URL file/URL ...
```

Unlike pipe mode, however, batch mode is silent by default. It simply loads the file or URL and runs the user's script:

```
file/URL     # load the file or URL
script       # run the user's script
```

Batch mode assumes that the user's script generates its own output or causes some other side-effect. The command may use `puts` to print something to standard output, or `save` to save a document to disk, or may invoke some other command (or a web transaction) that has a side-effect.

Argument Mode

The last command line mode is argument mode, indicated by `-argv`:

```
lapis -argv -e script      arg arg ...
lapis -argv -f scriptfile  arg arg ...
```

Unlike the previous modes, argument mode does not interpret the additional arguments. Instead, the arguments are placed in the Tcl variable `argv`. The user's script is responsible for looking at the `argv` variable to obtain any arguments it needs, do its processing, and then produce its output.

Argument mode is used to invoke self-contained programs written in the LAPIS scripting language. For example, this program searches an online dictionary for the definition of every word passed as an argument:

```
  # look up dictionary definitions
  # for each word given in arguments
foreach word $argv {

    # get dictionary's search page
  http://work.ucsd.edu:5141/cgi-bin/http_webster

    # fill in and submit the search form
  enter Textbox $word
  submit

    # extract the word's definition
  extract {
    from Heading starting "From"
      to point just before Rule
  }
```

```
        # strip out HTML tags
    omit Tag

        # print definition to output
    puts [doc]
}
```

If this program were stored in a file named `lookup`, then it could be invoked by

```
lapis -argv -f lookup word word ...
```

For easier invocation, Unix allows scripts like `lookup` to begin with a line of the form `#!` that specifies the script interpreter to run. For LAPIS, this line looks like:

```
#!/home/rcm/lapis/bin/lapis-script -argv
```

where `/home/rcm` should be replaced by the location of the LAPIS installation. Using this trick, the `lookup` script can be run from the Unix command prompt, just like any other Unix script:

```
% lookup zygote
From WordNet (r) 1.6 (wn)
zygote
    n : the cell resulting from the union of an ovum and a
        spermatozoon (including the organism that develops
        from that cell) [syn: {fertilized cell}]
```