Chapter 1

Introduction

Computer users are surrounded by text — personal documents, web pages, email messages, newsgroup postings, program source code, data files, configuration files, event logs, and more. Even with the rise of graphical user interfaces, multimedia personal computers, and broadband networks with streaming video and audio, most of the information consumed and produced by computer users still comes in textual form.

Text is full of *structure* designed to help readers understand and use the information it contains. Figures 1.1–1.4 illustrate some of the structure found in email messages, web pages, and source code. Some structure is explicitly labeled, like the headers of the email message in Figure 1.1; some is more implicit, like the phone number and signature in the body of the message. Some structure is formal and hierarchical, with every part appearing in a fixed, well-defined place, like the Java language syntax in Figure 1.4; some is more informal, like the book titles and authors in Figure 1.2, or the advertisement in Figure 1.3.

Often a document has several distinct layers of structure. For example, at the highest level, Figure 1.2 is a list of books, with attributes like title, year, author, genre, and format. At a lower level, the page is laid out as a table, with rows, columns, header cells, and data cells. At a still lower level, not visible in the figure, the page is a tree of HTML markup elements, and at the lowest level, a string of ASCII characters. All these layers of structure provide useful information for understanding or processing the document.

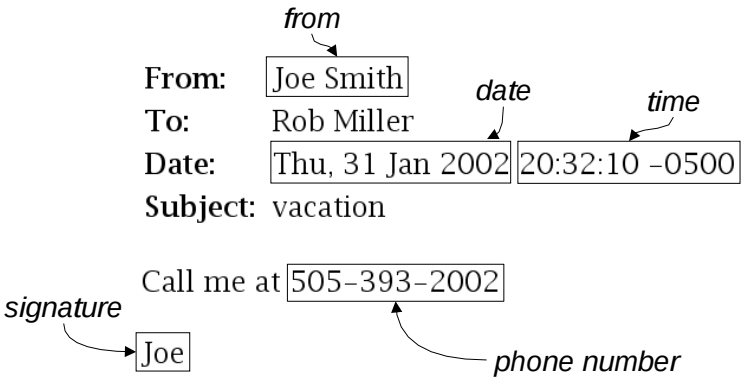

Figure 1.1: An email message with some of its structure labeled.

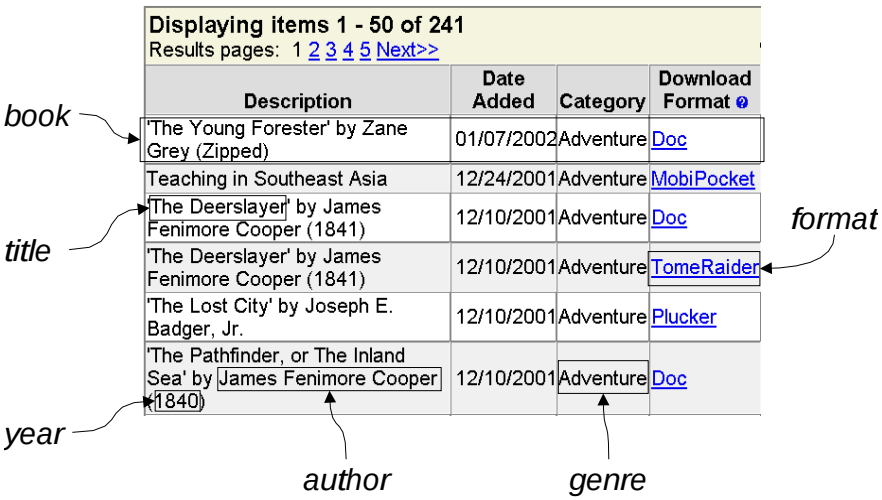

Figure 1.2: A web page that lists electronic books available for downloading.

Figure 1.3: A web page from Yahoo.

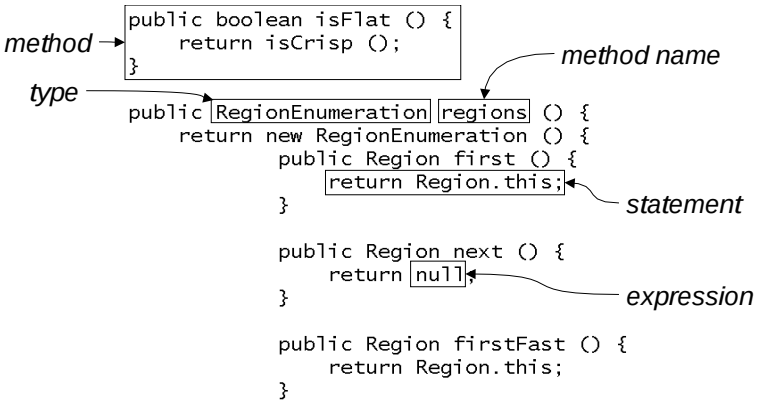

Figure 1.4: Part of a Java program.

Structure has value not only for reading and understanding text, but also for manipulating it — searching, editing, filtering, transforming, or rearranging its elements. An email user may want to find the message from Joe with his phone number in it. A book shopper may want to limit the books to a certain genre or format. A web surfer may want to strip advertisements from a web page. A programmer may want to find the methods that return `null` and edit them to throw an exception instead.

These manipulations require some way to describe the structure to be manipulated, such as books, advertisements, or Java methods.. Previous systems have generally chosen one of two well-known mechanisms for describing text structure: *grammars* or *regular expressions*. Grammars are well suited to describing formal, hierarchical structure imposed on an entire document, like programming language syntax or document markup. Grammars are used by syntax-directed program editors like Gandalf [HN86] and the Cornell Program Synthesizer [RT89], and by markup languages like SGML [Gol90] and XML [W3C00]. Regular expressions, on the other hand, are suitable for matching local or informal structure, like phone numbers or email addresses, but lack the expressive power to describe hierarchical structure. Regular expressions are used heavily by awk [AKW88] and Perl [WCS96].

One problem with previous approaches to structure description is that they provide little opportunity for *reuse*, especially between the two approaches. Perl programmers struggle to describe HTML elements with complex regular expressions, despite the fact that grammar-based HTML parsers already encapsulate that knowledge. A grammar-based XML tool cannot take apart an email address like `rcm@cs.cmu.edu` that appears in an XML document, because the parts of the address are not explicitly marked up. Yet a regular expression could handle this task easily.

This thesis describes a new approach to structure description that solves the problem of reuse: *lightweight structure*. Lightweight structure allows a system's basic structure concepts to be defined by a variety of mechanisms — grammars, regular expressions, or in fact any kind of pattern or parser. These structure concepts can then be composed and reused, regardless of the mechanism used to define them.

The lightweight structure approach has three parts:

1. a *model* of text structure as contiguous segments of text called *regions*. Each of the labeled rectangles in Figures 1.1–1.4 is a region.

2. an extensible *library* of structure abstractions. A structure abstraction is a named concept, like a word, a sentence, or a phone number. Every label in Figures 1.1–1.4 is a plausible structure abstraction. A structure abstraction can be defined by any kind of parser or pattern. The output of a structure abstraction is the set of all regions in the text that belong to the concept — e.g., the set of words, the set of sentences, or the set of Java expressions.

3. an *algebra* for combining sets of regions, based on relations like *before, after, in*, and *contains*. The algebra allows structure abstractions to be composed to create new abstractions.

The ability to compose and reuse structure abstractions is the most powerful aspect of lightweight structure. Suppose the library includes a hyperlink abstraction defined by an HTML parser, and an email address abstraction defined by a regular expression. Lightweight structure allows these abstractions to be composed to find hyperlinks that contain email addresses, without modifying

either of the constituent abstractions. The resulting composition, which might be called an email link, can be put back in the library as a new abstraction.

Lightweight structure does for text pattern matching what procedural abstraction does for programming. It provides a uniform interface, analogous to a procedure calling convention, that enables the creation of genuine abstractions with encapsulation and information hiding. Abstractions need not be shoehorned into one particular description language with limited expressive power. Instead, a structure abstraction can be expressed in any kind of pattern language, including context-free grammars, regular expressions, and region algebra expressions. Alternatively, an abstraction can be implemented by some process, such as a hand-coded scanner, a Turing machine, or even a human being doing manual selection with a mouse. Different ways of implementing a structure abstraction may have different properties in terms of performance, cost, or robustness, but functionally they are the same. To put it concretely, a user can use the hyperlink abstraction in a pattern without knowing how it is implemented, in the same way that a C programmer can call `strcpy` without knowing the details of its implementation. Lightweight structure enables the construction of a reusable text pattern library analogous to a function library.

Lightweight structure has the added advantage that the abstractions in the library are available at all times. Abstractions like phone numbers, URLs, and sentences can be used regardless of whether the user is looking at a web page, an email message, or a Java program.

1.1 Applications

Lightweight structure has a wide range of possible applications, a number of which are explored in this dissertation:

- **Pattern matching.** The region algebra serves as a basis for a new pattern language, called *text constraints* (TC), that permits the composition and reuse of lightweight structure abstractions. Because TC can use structure abstractions as primitives, it is higher-level than pattern languages that refer only to characters, such as regular expressions and grammars. As as result, TC patterns tend to be simpler, and easier to read and write. A user study showed that users can successfully read and write TC patterns.

- **Unix-style text processing.** A cornerstone of the Unix environment is its basket of generic tools, like `grep` and `sort`, that can be combined into pipelines and scripts to solve text-processing problems without writing a custom program. Unfortunately, these tools can be hard to apply to richly structured text like web pages, source code, and structured data files, because the tools generally assume that the input is an array of lines. Lightweight structure allows this limitation to be overcome, so that generic Unix-like tools can be used to filter and manipulate other kinds of structure, like the books in Figure 1.2 or the Java methods in Figure 1.4.

- **Web automation.** More and more information comes to users from the World Wide Web, and more and more work is done by interacting with services through the Web rather than running applications locally. Lightweight structure makes it easier for a user to script web interactions — clicking on hyperlinks, filling in and submitting forms, and extracting struc-

tured data from web pages — so that repetitive browsing activities can be automated, and web-based services and information sources can be incorporated into other programs.

- **Repetitive text editing.** Text editing is full of repetitive tasks. The region set model of lightweight structure provides a new way of performing these tasks: *multiple-selection editing*. Multiple-selection editing allows the user to edit multiple regions in a document at the same time, with the same commands as single-selection editing. Selections can be made by the mouse, by lightweight structure abstractions, or by writing a pattern. Multiple-selection editing is more interactive than other approaches to repetitive editing, like keyboard macros or find-and-replace, and lightweight structure makes it easier to describe the desired selections.

- **Inferring patterns from examples.** Many applications of text pattern matching can be improved by learning the patterns from examples, among them information extraction [KWD97, Fre98] and repetitive text editing by demonstration [WM93, Mau94, Fuj98, LWDW01]. Previous systems learned from fixed, low-level structure concepts, like words and numbers. Lightweight structure provides a library of high-level concepts, so inferences can directly refer to HTML or Java syntax without having to learn it first. This advantage is exploited by two techniques described in this thesis. The first is an algorithm that infers TC patterns from positive and negative examples. The inferred patterns can be used for any of the applications described previously, including Unix-style text processing, web automation, and adding more abstractions to the library. The second technique, called *simultaneous editing*, is expressly designed for repetitive text editing. Simultaneous editing is multiple-selection editing where the multiple selections are inferred from examples, using not only lightweight structure but also a special heuristic for repetitive editing to make very accurate inferences with very few examples. Simultaneous editing has been found to be fast and usable by novices, closely approaching the ideal of 1 example per inference.

- **Error detection.** *Outlier finding* is a new way to reduce errors by drawing the user's attention to inconsistent lightweight structure. Outlier finding can point out both possible false positives and possible false negatives in a pattern match or multiple selection. When outlier finding was integrated into simultaneous editing, it was found to reduce user errors.

1.2 LAPIS

These applications have been implemented in a system called LAPIS,[1] a web browser and text editor that supports lightweight structure. A screenshot of LAPIS is shown in Figure 1.5. The major new features of LAPIS are:

- **Multiple selections.** Multiple regions of text can be selected at the same time, using pattern matching, inference, or manual selection with the mouse. Multiple selections can be used to extract, filter, sort, replace, and edit text in a document.

[1] LAPIS stands for Lightweight Architecture for Processing Information Structure.

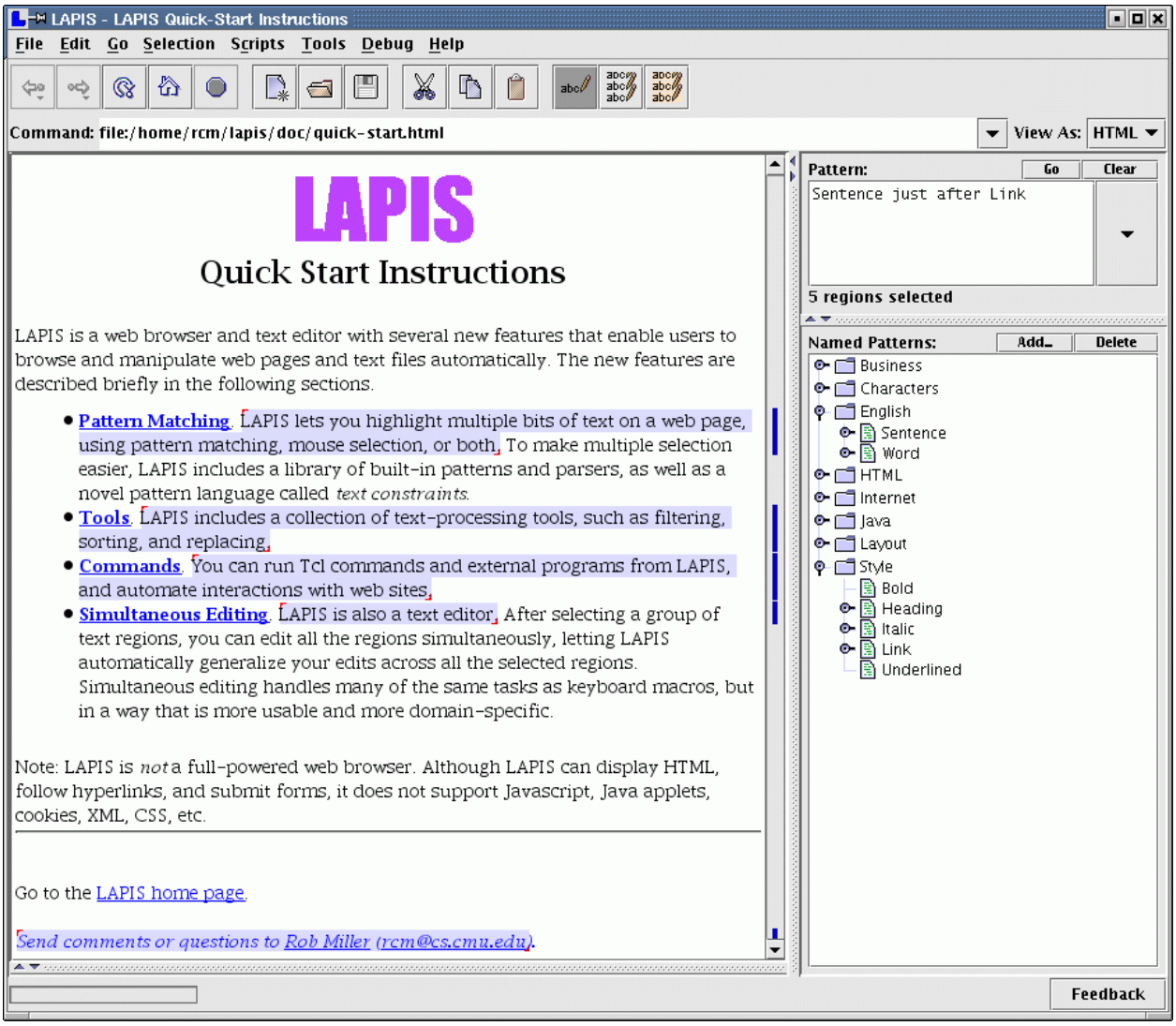

Figure 1.5: The LAPIS web browser/text editor, which demonstrates how lightweight structure can be used in a text-processing system.

- **Structure library.** LAPIS includes an extensible library of structure abstractions, shown on the right side of Figure 1.5 under the heading "Named Patterns." The default library includes HTML syntax defined by an HTML parser, Java syntax defined by a Java parser, and a variety of concepts like words, sentences, lines, phone numbers, and email addresses, defined by regular expressions and TC patterns. The library can be extended by plugging in new parsers, writing patterns, inferring patterns from examples, or making selections manually with the mouse.

- **TC pattern language.** The TC pattern language allows abstractions from the library to be composed and reused. Figure 1.5 shows an example of a TC pattern, `Sentence just after Link`. TC patterns are used to make multiple selections, add abstractions to the structure library, specify the arguments of script commands, and display feedback about inferences.

- **Scripting language.** LAPIS includes a scripting language, based on Tcl, that can be used to write text-processing scripts using the structure library and TC patterns.

- **Browser shell.** Script commands can also be invoked interactively, using a novel user interface called a *browser shell*. The browser shell integrates the command interpreter into a web browsing interface, using the LAPIS command bar to enter commands, the browser pane to display output, and the browsing history to manage the history of outputs. External command-line programs can also be invoked through this interface, allowing existing Unix tools to be interleaved with LAPIS script commands. The browser shell also offers a new way to construct Unix-style pipelines, described in more detail in Chapter 8.

- **Inference.** LAPIS can infer patterns from positive and negative examples given with the mouse. Inferences are displayed both as multiple selections and as TC patterns. Inferred patterns can be used for editing or invoking commands, or placed into the structure library as new abstractions.

- **Outlier finding.** When LAPIS is inferring patterns from examples, it uses outlier finding to highlight unusual matches to the inferred pattern, so that the user can check them for possible errors. The outlier finder can also be invoked directly by the user to find possible errors in *any* multiple selection, including selections made by library abstractions or patterns written by the user.

The target audience for LAPIS, and indeed for the entire lightweight structure approach, covers a wide spectrum of computer users. Programmers can plug in a parser for their favorite programming language or data format, and then use it with any of the features of LAPIS: pattern matching, multiple-selection editing, scripting, even inference and error detection. Power users who are not necessarily programmers can describe structure by writing TC patterns or inferring them from examples, and then manipulate the structure with Unix-style text-processing commands and multiple-selection editing. Even casual or novice users can benefit from LAPIS: the user studies described in this thesis show that users can understand and use simultaneous editing and outlier finding without learning anything about lightweight structure.

The current LAPIS implementation is targeted primarily at three domains: HTML web pages, Java source code, and plain text. The examples in this dissertation are drawn from these domains.

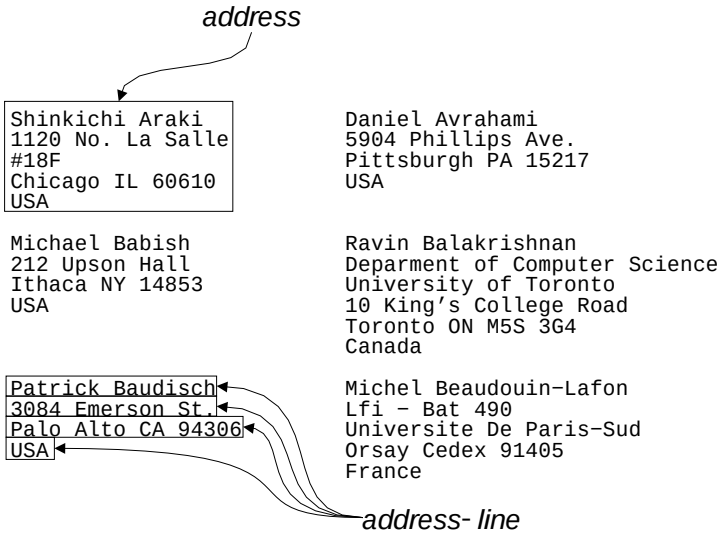

Figure 1.6: Not all useful abstractions can be represented by a contiguous region. In this plain text file laid out in two columns, each *address* actually consists of several noncontiguous regions. The closest we can get in the lightweight structure model is *address-line*.

Other text formats can also be processed, as long as their structure can be described by parsers, regular expressions, or TC patterns. LAPIS is not particularly useful for processing binary formats like Microsoft Word documents, but neither are Perl or awk. However, nothing precludes incorporating aspects of the lightweight structure approach, or particular techniques like simultaneous editing and outlier finding, into a monolithic word-processing application like Microsoft Word.

1.3 Limitations

The lightweight structure approach is designed for recognizing and exploiting structure, not creating or validating it. For example, a phone number abstraction can recognize phone numbers, but it cannot produce a template of a phone number for the user to fill in, or guarantee that all phone numbers in a document are formatted the same way. The grammar-based approach used by syntax-directed editors and SGML/XML is better suited to these tasks.

The model of structure as contiguous regions in a one-dimensional string can capture most kinds of text structure, but not all. In particular, some aspects of two-dimensional layout are impossible to represent with contiguous one-dimensional regions. Figure 1.6 shows a plain text file of postal addresses laid out as two columns. In this format, an address is not a single contiguous region in the file. Instead, each address is split across multiple lines, and the lines of one address are interleaved with the lines of another. There is no way to define a structure abstraction that returns each address as a single unit, corresponding to the *address* label shown in the figure. The closest we can get is the *address-line* abstraction shown in the figure, which returns the individual lines of the addresses but doesn't group them together. A similar problem is encountered with the *column* concept in HTML tables, since HTML specifies tables in row-major order. Although the model described in this thesis cannot represent these cases, it may be possible to extend the model; more

will be said about this possibility in the conclusion. Regular expressions and grammars cannot express the *address* abstraction either, for the same reasons.

Like regular expressions and context-free grammars, the region algebra has limits on the classes of languages that it can recognize. The recognition power of the region algebra is not fixed, however, but rather depends on the power of the abstractions being composed. For example, algebra expressions over regular abstractions can recognize only regular languages, but algebra expressions over context-free abstractions can recognize *more* than just context-free languages. These results are proved in Chapter 5.

1.4 Contributions

My thesis statement is:

> Lightweight structure enables efficient composition and reuse of structure abstractions defined by various kinds of patterns and parsers, bringing improvements to pattern matching, text processing, web automation, repetitive text editing, inference of patterns from examples, and error detection.

This dissertation makes contributions in a number of areas. Some contributions are theoretical:

- a model of text structure as region sets, which allow a variety of structure description mechanisms (regular expressions, grammars, or any kind of parser) to be encapsulated as simple abstractions;
- an algebra for region sets that enables structure abstractions to be composed and reused;
- data structures and algorithms for efficient representation of region sets and implementation of the region algebra;
- theoretical results about the classes of languages recognized by the region algebra;
- algorithms for inferring patterns from examples using lightweight structure;
- heuristics for repetitive editing that produce accurate inferences with few examples;
- algorithms for outlier finding that detect inconsistent pattern matches using lightweight structure.

Other contributions fall into the category of new languages and system designs:

- the TC pattern language, based on the region algebra, which allows users to write simple, readable text patterns using structure abstractions;
- a command language that extends Unix-style text processing to richly-structured text like web pages and source code;
- the browser shell, which integrates a command prompt into the web browsing interaction model.

Finally, several contributions are made to user interface design:

- various techniques for making multiple selections in text, including mouse selection, pattern matching, and combinations thereof;
- using multiple selections for interactive text editing;
- using mouse selection to give positive and negative examples of text patterns;
- using multiple selections and patterns to give feedback about inference;
- highlighting techniques for displaying region sets in a document;
- highlighting techniques that draw the user's attention to unusual selections or pattern matches, in order to reduce errors.

1.5 Thesis Overview

The next chapter, Chapter 2, surveys related work. Then the heart of the dissertation is divided into two parts. The first part describes lightweight structure itself:

- Chapter 3 defines the region set model and the region algebra, and shows how higher-level pattern-matching operators can be defined in terms of the core algebra.
- Chapter 4 shows how the region set model and algebra can be implemented efficiently.
- Chapter 5 proves some results about the expressive power of the region algebra, characterizing the classes of languages recognized by region algebra expressions that use various kinds of structure abstractions.

The second part discusses applications of lightweight structure:

- Chapter 6 describes text constraints (TC), the user-level pattern language based on the region algebra.
- Chapter 7 introduces the LAPIS web browser/text editor. A key feature of LAPIS is its support for multiple selections, represented by a region set. This chapter describes how multiple selections are made in LAPIS using the mouse, the pattern library, and TC patterns. A user study tested all three selection mechanisms, focusing in particular on the readability and writability of TC patterns.
- Chapter 8 describes the text-processing commands built into LAPIS. LAPIS includes commands that are similar to familiar Unix text-processing tools like `grep` and `sort`, but are more useful on richly-structured text like HTML and source code. This chapter also describes the browser shell, which integrates a command interpreter into the web browsing model, and shows how LAPIS can be used to automate interactions with web sites.

- Chapter 9 explains how LAPIS infers multiple selections from examples using two different techniques: selection guessing and simultaneous editing. Two user studies included in this chapter show that selection guessing and simultaneous editing help even on small repetitive tasks, and are more effective than another repetitive-text-editing system, DEED [Fuj98].

- Chapter 10 explains how outlier finding is used in LAPIS to highlight possible errors in a selection. A user study showed that outlier highlighting reduced the tendency of users to overlook inference errors.

Finally, Chapter 11 discusses some of the major design decisions in LAPIS, reviews the contributions of the dissertation, and outlines future directions for this work.