Chapter 11

Conclusion

This concluding chapter discusses the main design decisions in lightweight structure, summarizes the contributions of the dissertation, and suggests some directions for future work.

11.1 Discussion

Lightweight structure offers a new way to describe and manipulate structure in text. A number of design decisions are essential to the idea of lightweight structure and its implementation in LAPIS. This section reviews these decisions and justifies them, in light of the supporting evidence presented in previous chapters.

The strongest evidence of the value of lightweight structure lies in the range of applications to which it has been applied: pattern matching (Chapter 6), multiple-selection editing (Chapter 7), Unix-style text processing and web automation (Chapter 8), selection guessing and simultaneous editing (Chapter 9), and outlier finding (Chapter 10) are all applications that benefit from lightweight structure. The user studies and examples in these chapters demonstrate the usefulness of these applications.

The applications also prove that one of the main goals of the thesis has been met: namely, that lightweight structure enables reuse of structure abstractions. Users can compose abstractions to write TC patterns and define new abstractions. Unix tools can exploit abstractions to filter, sort, and transform richly structured text, not just plain text files. Web scripts can employ abstractions to click on hyperlinks, fill in form fields, and extract data form the Web automatically. Machine learning agents can use abstractions in features to infer patterns from examples and detect potential pattern-matching errors. That the same library of structure abstractions could be reused in all these applications testifies to the generality of the lightweight structure idea.

One potentially controversial aspect of lightweight structure is the decision to discard the conventional syntax tree representation of text structure, in favor of region sets. Although syntax trees make an appearance in LAPIS, since both the HTML parser and Java parser generate trees in the course of parsing, the trees are only used to generate region sets and are then discarded. Region sets are the primary representation of structure used in LAPIS.

The region set representation has a number of advantages. It is easy to represent multiple independent hierarchies in the same document, such as the physical hierarchy (lines and pages), the logical hierarchy (sentences and paragraphs), or the syntactic hierarchy in source code (expres-

sions and statements). Patterns and inferences can freely intermix references to abstractions from different hierarchies. No abstractions are preferred over any others by the region set representation, whereas a syntax tree representation tends to prefer patterns and operations on syntactic structure. The region set representation allows a lightweight structure system to support a range of approaches to text processing: lexical, syntactic, and hybrids.

One drawback of the region set representation, relative to a tree, is that some kinds of hierarchical queries are slower. In a tree, the children of a node can be found in $O(1)$ time simply by following pointers. With region sets representing the structure, however, finding the children of a region requires a query into a rectangle collection data structure, which typically takes $O(\log N)$ time. the applications to which lightweight structure is applied in this dissertation, however, finding the children of a *single* region is far less common than finding the children of a *set* of regions throughout a document, in which case the cost of traversing a rectangle collection can often be amortized across all the regions in the set using techniques like tandem traversal (Section 4.6.5) or plane-sweep intersection (Section 4.6.7).

When speed of access to one hierarchy is important, however, lightweight structure does not prohibit using a tree representation in concert with region sets. In fact, the rendered HTML view in LAPIS does precisely this. When a web page is displayed in LAPIS, it is represented both as a tree of HTML elements for efficient rendering, and as a string of content for matching and manipulation by region sets. Thus the region set representation can coexist with, and provide benefits to, a tree representation.

LAPIS also demonstrates that region-set structure can be added to an existing text editor or web browser without changing the system's internal data structures. LAPIS is designed around the `JEditorPane` component included in the Java library. Although the user interface of the component had to be overridden and extended substantially, mainly to permit rendering of multiple selections and editing with multiple cursors, the internal document representation used by `JEditorPane` was not modified at all. In fact, the components of LAPIS that implement lightweight structure — the region set data structures, algebra operators, and pattern language — are independent of the document representation. These components interact with the document through a generic interface, `Document`, which essentially presents a document as a string of content with a collection of metadata properties (Section 8.13). LAPIS actually includes several implementations of the `Document` interface. One is a wrapper for the internal document representation used by `JEditorPane`, which is used when LAPIS is running as a graphical user interface. Another implementation simply wraps around a `String`. This implementation is used when LAPIS is run from the command line without a graphical user interface (Section 8.14). Other editors or browsers could be supported by wrapping their internal representations with an implementation of `Document`. This level of independence is made possible by the fact that lightweight structure separates the representation of text from the structure that is used to manipulate it.

A key feature of the region algebra is the decision to support arbitrary region sets. Many systems support only flat structure, or only hierarchical structure, or only overlapping structure. The region algebra described in this dissertation can handle all of these kinds of structure, even combined into the same region set. The greatest benefit of this approach lies in its simplicity and uniformity. A user can write a pattern that refers to both `Line` and `Sentence` without paying undue attention to the fact that lines and sentences may overlap or nest in arbitrary ways. Although the cost of this uniformity is sometimes quadratic running time, the performance tests in Chapter 4 show that the common case is significantly better.

Supporting arbitrary region sets also opens the door to using the region set representation for *predicates* as well. This observation lies behind the decision to define unary operators in the region algebra. In other systems, a predicate like *contains PhoneNumber* can only be represented by an procedure, specifically one that tests whether a given region contains a phone number. In the region algebra, however, *contains PhoneNumber* can be represented explicitly by a region set — the set of all regions in the document that satisfy the predicate. A region set representing a predicate can be treated in the same way as a region set representing any other kind of structure. It can be searched, counted, combined with other region sets, and used for pattern matching, inference, or text manipulation. The explicit representation of predicates as region sets is essential to the interactive performance of selection guessing, simultaneous editing, and outlier finding, since predicate features can be precomputed, stored, and then used to form hypotheses or feature vectors.

Probably the most powerful advantage of lightweight structure — and the one most likely to lead to future innovations — is the way that all structure is treated as encapsulated abstractions with opaque implementations. From the user's point of view, `Sentence` is simply a set of sentence regions. The actual definition of `Sentence`, i.e., the parser that determines which regions are sentences and which aren't, is irrelevant. It might be a regular expression, a grammar, or a collection of TC patterns. It might also be a Bayesian classifier, a neural net, or some other learned classifier. It might be a collection of classifiers that vote, or an agent that accesses a dynamic structure library on the Web. It might even be a hybrid of machine intelligence and human intelligence, with a simple parser that correctly identifies 99% of the cases, leaving the remaining 1% (perhaps selected by outlier finding) to a human judge. The lightweight structure model opens up a wide spectrum of possibilities for structure description. Yet regardless of how structure is described, it can be used in all of the applications discussed in this dissertation: pattern matching, text processing, web automation, editing, inference, and outlier finding. More will be said at the end of this chapter about the larger possibilities of lightweight structure, for not only text but other kinds of media as well.

11.2 Open Questions

Like any thesis, lightweight structure raises new questions while answering others. This section gathers together some of the most salient and interesting questions about the specific ideas and techniques presented in this dissertation.

Region Set Model and Algebra (Chapter 3)

One limitation of the region set model is the difficulty of representing noncontiguous structure. Section 1.3 gave several examples of such structure, such as columns in plain text or HTML tables. One way to get around this limitation might be to use *sets* of region sets — i.e., representing each column as a region set, and the set of columns as a set of region sets. This extension should be taken with some care, however, to avoid combinatorial explosion; although the maximum size of a region set is only $O(n^2)$, a set of region sets may be $O(2^n)$.

Another kind of noncontiguous structure is linking. Hyperlinks, function calls, variable references, and bibliographic citations are all examples of linking structure. Links may represent relations not just within documents, which is the kind of structure that has occupied the inter-

est in this thesis, but *between* documents as well. Representing linking in the region set model would call for two changes. First, each region would have to be represented as a triple (document, start offset, end offset), and region set data structures would have to be extended to store regions from multiple documents in the same set. Second, the links themselves might be represented by document-generated relations mapping the region at one end of the link to the region at the other end, and vice versa. Rectangle collection data structures might be useful for storing these relations, once they have been parsed from a set of documents.

Representing two-dimensional relationships in page layout is another interesting issue. The region set model described in this thesis treats every document as a one-dimensional string, so the six fundamental relations *before*, *after*, *overlaps-start*, *overlaps-end*, *in*, and *contains* are sufficient basis for the region algebra. When a document is rendered on screen or on paper, however, it becomes two-dimensional, and its components may have two-dimensional relationships, such as *above*, *below*, *left*, and *right*. The region algebra could either be extended to capture these relationships, or they could be represented like linking structure, as document-specific relations generated by the renderer.

Region Algebra Implementation (Chapter 4)

LAPIS implements only two rectangle collection data structures, RB-trees and region arrays. It would be instructive to implement some of the other alternatives described in Chapter 4, such as quadtrees, 4D point collections, and plane-sweep intersection, and compare performance.

The measurements at the end of Chapter 4 suggest that TC pattern matching might outperform regular expression matching in some cases. A head-to-head comparison would require using the fastest regular expression package available, however, which would certainly be written in C or C++, so the region algebra implementation would also have to be rewritten in a higher-performance language to make the comparison fair.

Scaling up to large document collections is another important area of future work. Applying all the patterns and parsers in the LAPIS library, which is necessary for inference and outlier finding, takes about a minute per megabyte of text scanned (Table 4.6). Alternative data structures or high-performance implementation languages may reduce this time, but only by a constant factor, and the library is very likely to grow as the user adds more patterns and parsers. Long-term solutions may involve smarter decisions about which patterns or parsers to try (e.g., only run Java-related parsers and patterns on files that end in `.java`), or caching region sets on disk to avoid repeated parsing. Fortunately, the RB-tree used in LAPIS is derived from a disk-based data structure, so working with on-disk RB-trees would be easier.

Language Theory (Chapter 5)

Several questions remain to be answered about the recognition power of the region algebra, mostly involving the effect of the *forall* operator. It seems clear that $RSTA_{\emptyset}$ is more powerful than $RST_{\emptyset}$, but characterizing the class of languages recognized by $RSTA_{\emptyset}$ is still an open question, as are the languages recognized by $RSTA_{\mathcal{F}}$ and $RSTA_{\mathcal{CFL}}$. It would also be interesting to extend the region algebra with operators from other pattern languages — for example, the Kleene star from regular expressions, or recursive productions from context-free grammars — and study the effect on the class of languages that can be recognized.

TC Pattern Language (Chapter 6)

Probably the most valuable addition to the TC pattern language would be a facility for defining generic, parameterized structure abstractions. The HTML parser already uses this idea to a limited extent by embedding "parameters" in its abstraction names: a tag name in angle brackets (e.g., `<img>`) matches a start tag, and a tag name in square brackets (`[body]`) is an element. These abstractions are not truly parameterized, however, since the parser only recognizes a finite set of them, one for each tag defined in HTML 4.0. An XML parser using this kind of naming scheme would have to support arbitrary tag names, a good reason to introduce parameterization.

User-defined relational operators would be another way to introduce parameterization into TC. For example, program code might be searched for `Method` *`named`* `"copy"`, or an email archive for `Message` *`from`* `"John"`, where the operators *`named`* and *`from`* are user-defined relations. Forming database-like queries would be easier if the language supported numeric and date relations, such as `Price < 100`. The most generic way to support these kinds of constraints would be to embed script code in a pattern that tests each region in a region set to see if it satisfies the predicate.

Another useful way to compose parsers is to use the output of one parser to determine the input to another. For example, Java comments may include HTML tags, so it makes sense to apply the Java parser first to find the comments, then apply the HTML parser just to the comments. The rendered view in LAPIS is another example (Section 6.2.11), since it is generated by the HTML parser by stripping out tags and then used as input for other parsers and patterns. Specifying these kinds of relationships between parsers would require a parser-control language, which might use TC expressions to specify how data should flow between parsers.

LAPIS (Chapter 7)

The highlighting techniques used in LAPIS are not well-suited to displaying nested or overlapping region sets. Several proposals for improvement are shown in Figure 7.5.

Multiple-selection editing likewise only supports editing with flat region sets. In hierarchical structure, like source code or XML, it may sometimes be useful to apply editing commands to a nested region set. Extending the semantics of familiar editing commands to nested selections, and testing whether users understand them, would be interesting.

LAPIS can currently only edit plain text, not rendered HTML. Since the `JEditorPane` editor component on which LAPIS is based is capable of editing rendered text, extending LAPIS to do likewise would not be terribly difficult. More challenging, but more rewarding, would be integrating some or all of the features of LAPIS into an existing application like Emacs or Mozilla. The hardest part of the integration would probably be overriding the single-selection behavior that is deeply ingrained in most applications. If this obstacle can be overcome, however, the benefits would be significant, since both Emacs and Mozilla have large user bases who could provide more experience with the usability of these features in daily use.

Large documents with many selections pose a problem for multiple-selection editing. Making a million edits with every keystroke may slow the system down to a crawl, particularly if the text editor uses a *gap buffer* to store the text [Fin80]. Gap buffers are used by many editors, including Emacs and `JEditorPane`. With a gap buffer and a multiple selection that spans the entire file, typing a single character forces the editor to move nearly every byte in the buffer. One way to

address this problem is to delay edits to distant parts of the document until the user scrolls to them or saves the document. Another solution might be to have multiple gaps in the buffer, one for each selection.

Unix Tools and Browser Shell (Chapter 8)

The Unix tool set includes many more tools that might be usefully adapted to lightweight structure. Some likely candidates are:

- `diff`, for comparing two region sets in the same or different documents;
- `uniq`, for eliminating duplicates in a region set, where the notion of a duplicate may be generalized beyond simple character-by-character identity;
- `patch`, for merging differences between two region sets.

One drawback of the LAPIS tools, compared to their Unix counterparts, is that processed files are loaded entirely into memory. The LAPIS tools would scale better if they operated with a single pass across a file, using bounded working memory. Single-pass operation may not always be possible, since TC patterns can depend on nonlocal context which may require access to the entire file, but it would be desirable whenever possible.

Several features are needed to make the browser shell more useful and more efficient as a command shell, including:

- **Background processes.** Web browsers generally stop loading a page when a new URL is typed in the Location box. Similarly, LAPIS automatically stops the currently executing command when a new command is typed in the Command box. As a result, only one command can be running in each LAPIS browser window. An improvement would be support for background-process syntax. If a command ends with &, it could continue running in the background, saving its output in case the user ever backs up through the history.
- **Handling large outputs.** A command may generate too much output for the browser to display efficiently. The same problem often happens in typescript shells, usually forcing the user to abort the program and run it again redirected to a file. To handle this problem, the browser could automatically truncate the display if the output exceeds a certain user-configurable length. The remaining output would still be spooled to the browser cache, so that the entire output can viewed in full if desired, or passed as input to another program.
- **Streaming I/O.** A pipeline may process too much data for the browser's limited cache to store efficiently. Although the browser shell's automatic I/O redirection could still be used to assemble the pipeline (presumably on a subset of the data), the pipeline would run better on the real data if its constituent commands were invoked in parallel with minimal buffering of intermediate results. The browser shell might do this automatically when invoking a script, since the intermediate results of a script do not need to be shown unless it is being debugged.

- **Shell syntax.** Expert users would be more comfortable in the browser shell if it also supported conventional operators for pipelining and I/O redirection, such as |, <, >, and > >. The most direct way to accommodate expert users might be to embed an existing shell, such as `bash` or `tcsh`, as an alternative to Tcl.

Inference (Chapter 9)

The inference algorithms implemented in LAPIS only infer multiple selections in a single file. It would be straightforward to extend them to infer selections across multiple files, such as a collection of web pages or source code modules, but some additional issues would arise.

First, when inference is applied to multiple files, or large single files, it becomes harder for the user to check for incorrect inferences. Outlier highlighting helps, but it requires the user to browse through the documents to look at the outliers. One visualization that might avoid scrolling is a "bird's-eye view" showing the entire file, with greeked text, so that deviations or outliers in an otherwise regular selection can be noticed at a glance. Another useful visualization would be an abbreviated context view, showing only the lines containing selections.

Another question that must be answered for editing multiple files by inference is where the data should reside. For simultaneous editing at interactive speeds, all the documents to be edited must fit in main memory, with some additional overhead for parsing and storing feature lists. For large data sets, this may be impractical. However, it is easy to imagine interactively editing a small sample of the data to record a script which is applied in batch mode to the rest of the data. The batch mode could minimize its memory requirements by reading and processing one record at a time (or one translation unit at a time, if it depends on a Java or HTML parser). Scripts recorded from simultaneous editing would most likely be more reliable than keyboard macros recorded from single-cursor editing, since simultaneous editing finds general patterns representing each selection. The larger and more representative the sample used to demonstrate the script, the more correct the patterns would be.

The script could also be saved for later reuse, which raises another question. Simultaneous editing is designed for repetitive tasks where all the instances of the task are available at the same time. This class of tasks might be described as *repetition over space*. Another important class of tasks are those in which instances arise one by one over a period of time, which might be termed *repetition over time*. Email filtering is a good example. The inference techniques described in this thesis might be applied to this kind of task by first converting it to a repetition over space by collecting a set of instances over a period of time. Simultaneous editing could then be applied to those instances, recording a script that for future instances of the task. At some point in the future, the recorded script may encounter an instance that it cannot handle correctly, so the system needs some way to detect these cases, perhaps using outlier finding, and let the user fix the script with more simultaneous editing.

The inference techniques in simultaneous editing could be improved in several ways. The one-selection-per-record bias is a powerful heuristic, but it is not always sufficient. Allowing the user to override the bias, by making multiple selections in some records or removing all selections in other records, would allow the user to do nested iterations (e.g., over the varying numbers of parameters in each method declaration) or conditionals (e.g., editing public methods one way, private methods a different way) without leaving simultaneous editing mode. Currently, simultaneous editing can handle these nested iterations and conditionals only by choosing different record sets, which is not

particularly natural. Inference can also be improved by specifying different weights for different library abstractions. For example, when the user is editing Java code, features involving Java syntax might be preferred by the inference algorithm over lexical features like capitalization or whitespace.

Since the user studies described in Chapter 9 tested selection guessing and simultaneous editing in isolation, it is still an open question whether users can understand the difference between the two modes and determine when to use each one, or whether the two modes should be somehow combined into one.

Finally, it would be interesting to implement and evaluate simultaneous editing in other kinds of applications. For example, spreadsheets often have sequences of similar formulas which might be edited simultaneously. A column in a relational database or a set of related filenames (e.g., `*.cpp`) in a file manager might be edited in the same way.

Outlier Finding (Chapter 10)

One problem with outlier highlighting is that it requires the user to scan the document to look at the outliers, which does not scale well. In addition to highlighting outliers, it may also make sense for the system to save up outliers for later proofreading and allow editing commands to be undone on individual records.

Outlier finding suggests a new way to do global find-and-replace. Traditional find-and-replace commands offer two options: replacing matches one at a time with confirmation, or replacing all matches at once. Outlier finding offers a third way: replacing outlier matches one at a time with confirmation, but replacing typical matches all at once. Designing a user interface that seamlessly integrates outlier finding into the find-and-replace task would be an interesting future work.

The Unusual Matches dialog is a step in the direction of a general pattern debugger, which might be as useful for awk and Perl programmers as it is for LAPIS users. Possible improvements include better explanations, the ability to fix a pattern using features from the explanation, and the ability to indicate that a feature is irrelevant. The outlier finding algorithm could also be improved by supporting literal `contains` features and scalar and real-valued features.

Outlier finding can be applied to testing and debugging in more general programming domains as well. An outlier finder might highlight unusual variable values, unusual procedure calls, and unusual event sequences. New visualizations would be needed to highlight these kinds of objects. Other research issues include developing a set of features for judging whether values and events are unusual, and exploring the tradeoff between precision and recall (i.e., highlighting too little vs. highlighting too much).

More Applications

Lightweight structure has a number of text-processing applications beyond the ones discussed in this dissertation.

Lightweight structure would be helpful for defining structure detectors for Apple Data Detectors [NMW98] and Microsoft Smart Tags [Mic02]. In particular, the ability to describe patterns by giving examples would make it easier for non-programming users to define their own data detectors or smart tags.

Copy-and-paste is another feature that could be improved by lightweight structure. Applications like bibliography editors, calendars, and contact managers represent data objects in structured fashion, usually presenting a form interface to the user for editing. Yet new citations, appointments, and contact information often arrive as unstructured or semi-structured text in email messages or web pages. Lightweight structure would enable the construction of *clipboard transducers* that convert between plain text format and the structured formats expected by other applications. Ideally, a user would be able to copy a citation from a web page or a signature line from an email message, and paste directly into a bibliography manager or contact manager, with the transducer automatically taking care of parsing it into structure and filling in the appropriate fields in the other application's interface.

Another potential application for lightweight structure lies in adapting web pages to resource constraints, such as low bandwidth and small screens. The user might describe the format of a web site by writing patterns or inferring them from examples, and then specify how pages in that format should be filtered or transformed when the site is viewed on a device with limited resources, such as a personal digital assistant or a cellphone.

11.3 Summary of Contributions

This dissertation introduces the idea of lightweight structure as a way to describe structure in text. Lightweight structure consists of the region set model, an extensible library of structure abstractions, and an algebra for combining region sets.

Although the region algebra is simple, consisting of six fundamental relational operators and the set operators, Chapter 3 shows that many other pattern-matching operators can be derived from it. Furthermore, when regions are interpreted geometrically as points in region space, the fundamental region relations correspond neatly to rectangles, which motivates the efficient rectangle-collection implementation described in Chapter 4.

The languages recognized by region algebra expressions are defined formally in Chapter 5, and found to depend on the recognition power of the abstractions referenced by the expression. Region algebra expressions using only literal string matching (and omitting *forall*) recognize the same class of languages as generalized star-free regular expressions. Region algebra expressions over regular abstractions (i.e., regular expressions in addition to literal strings) recognize regular languages, but algebra expressions over context-free expressions recognize a strict superset of the context-free languages.

The reusability of lightweight structure has been demonstrated by employing it in a variety of applications. The most natural application, of course, is pattern matching. The TC pattern language described in Chapter 6 is based very closely on the region algebra, so that TC patterns can compose and reuse structure abstractions. TC is also novel for the way it uses indentation to structure infix expressions and avoids Boolean operators where possible. A user study showed that users can generate and comprehend TC patterns, and the other applications of lightweight structure rely heavily on TC patterns to give feedback to the user.

LAPIS is another application of lightweight structure, showing how lightweight structure can be incorporated into a web browser or text editor. In LAPIS, the selection is a region set. Selections can be made by the mouse, by selecting patterns from the library of structure abstractions (including HTML and Java syntax), by writing a TC pattern, or by some combination of these

techniques. Displaying a region set in a document requires some new highlighting techniques, which are discussed in Chapter 7. That chapter and the following one also show how a region set selection can be used for editing and text processing, generalizing some common Unix tools like `grep` and `sort` to the world of lightweight structure.

LAPIS also incorporates a scripting language, based on Tcl, designed for text processing with lightweight structure. Chapter 8 describes the language and showed how it can be used for web automation. LAPIS integrates the script interpreter with web browsing, text processing, and external command invocation to produce a novel user environment, the browser shell.

Lightweight structure has also been applied to machine learning. Chapter 9 shows two techniques for inferring region set selections from examples given by the user. Selection guessing, the more general technique, infers TC patterns using any of the structure abstractions in the library as features. Simultaneous editing, the second technique, adds a heuristic — restricting the inference to hypotheses that make exactly one selection in every record being edited — that reduces the hypothesis search space so much that most selections can be made with only one example. Both techniques use multiple forms of feedback to keep the user informed, displaying the inference both as a multiple selection and as a pattern. User studies showed that even users untrained in lightweight structure and TC pattern matching can use selection guessing and simultaneous editing for repetitive editing tasks.

Finally, this dissertation introduces the idea of outlier finding as a technique for reducing errors in pattern matching or repetitive text editing. Outlier finding is used in two places in LAPIS: to highlight unusual selections during simultaneous editing, and to generate the Unusual Matches dialog. Lightweight structure plays a role in outlier finding as well, since structure abstractions are used as features.

11.4 Looking Ahead

Lightweight structure raises the level of abstraction at which users interact with data. Instead of being limited to characters, words, or lines, like most text-processing systems, lightweight structure allows the user and the system to communicate at a higher level, about HTML elements, Java expressions, English sentences, or any other abstractions that might be installed in the library. As this dissertation has shown, raising the abstraction level brings improvements to pattern matching, Unix-style text processing, web automation, repetitive editing, inference of patterns from examples, and error detection.

There are two natural directions one could proceed — two dimensions along which a future research plan might be mapped out. One way leads deeper into the rich structure of text, or more accurately, to higher levels of abstraction. The other way leads to greater breadth, expanding the idea of lightweight structure beyond text into other domains.

First consider depth. This dissertation concerned itself chiefly with lexical and syntactic structure, but text is also rich with *semantic* structure. For example, a large body of research has been devoted to the problem of automatic text summarization, resulting in algorithms that can automatically reduce a long document to a short summary by locating and extracting pivotal sentences. If these algorithms were integrated into a LAPIS-like system and placed under user control, then a user could highlight individual threads of discussion in a document or collection of documents, focus on salient sentences, and quickly gain a sense of the meaning of the document, answer a

question, or generate an abstract. Such a tool would be a boon for digesting, analyzing, and report-writing, problems which are increasingly important but not as yet supported by automated tools.

Source code also abounds with semantic information, including types, inheritance relationships, data flow, and control flow. Modern optimizing compilers incorporate a variety of advanced static analyses, such as interprocedural flow analysis and pointer alias analysis, which are largely hidden from the programmer's eye. If these analyses were exposed in a LAPIS-like interface, so that the user could view, constrain, and correct them in critical areas of a program, then the combination of compiler and human might be able to understand the code much better, helping the compiler to make it faster and the human to maintain it.

Style, in both writing and programming, is another interesting kind of semantic structure. Tools that evaluate style and help writers and programmers improve their style would be a fertile field of future work. Primitive style advice can be found in grammar checkers, `lint`, and compiler warnings, but these systems are ad-hoc, limited, and inextensible. The lightweight structure approach suggests that style knowledge could be built up like lexical and syntactic knowledge, as a library of reusable concepts that can be shared, adapted to the needs of individuals or organizations, and evolved over time.

Looking beyond text, structure can be found and exploited in many other kinds of data, including sound, 2D graphics, 3D graphics, and video. Like text, structure in other domains may exist on several different levels, some of which are more accessible to automatic detection than others. For example, in images, low-level structure like edges, corners, and areas of color are easy to detect; high-level structure like automobiles and buildings are much harder to identify. Nevertheless, a uniform framework for describing and composing structure detectors may have as much value in other domains as it does in text.

Take video as an example. As video begins to rival the written word as a communications medium, even ordinary users will need to browse, search and edit digital video. Algorithms developed for computer vision, such as object tracking and face recognition, could help with many of these tasks, but only if the algorithms are integrated into a usable human interface in which the user can control, constrain, and correct their results. Many of the techniques in LAPIS, such as the usable pattern language, simultaneous editing, and outlier finding, might have useful analogs in video editing.

Unlike text, other kinds of media draw a sharp distinction between structured and unstructured data. For example, there are two kinds of 2D graphical editors: drawing editors, which represent an image as a structured collection of graphical objects like rectangles, circles, and lines; and paint programs, which represent an image as an unstructured array of pixels. Likewise, in digital audio, one can use a MIDI editor to edit structured music files, or a waveform editor to mix unstructured digital audio. Lightweight structure suggests a third alternative: editing unstructured data as if it were structured, which might be a powerful way to unify the structured and unstructured application models.

Text will continue to be important, and lightweight structure can help. The techniques developed in this dissertation form a solid foundation for future tools, helping users bring the power of abstraction and automation to bear on their text processing problems.