Lightweight Structured Text Processing

Robert C. Miller and Brad A. Myers
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
{rcm,bam}@cs.cmu.edu
http://www.cs.cmu.edu/~rcm/lapis/

Abstract

Text is a popular storage and distribution format for information, partly due to generic text-processing tools like Unix *grep* and *sort*. Unfortunately, existing generic tools make assumptions about text format (e.g., each line is a record) that limit their applicability. Custom-built tools are one alternative, but they require substantial time investment and programming expertise. We describe a new approach, *lightweight structured text processing*, which overcomes these difficulties by enabling users to define text structure interactively and manipulate the structure with generic tools. Our prototype system, LAPIS, is a web browser that can highlight, filter, and sort text regions described by the user. LAPIS has several advantages over other systems: (1) the ability to define custom structure with a simple, intuitive pattern language; (2) interactive specification, showing pattern matches in context and letting users choose the most convenient combination of manual selection and pattern matching; and (3) external parsers for standard text formats. The pattern language in LAPIS, *text constraints*, describes text structure in high-level terms, with region relationships like *before*, *after*, *in*, and *contains*. We describe an implementation of text constraints using a novel, compact representation of region sets as collections of rectangles, or *region intervals*. We also illustrate some examples of applying LAPIS to web pages, text files, and source code.

1 Introduction

Structured text has always been a popular way to store, process, and distribute information. Traditional examples of structured text include source code, SGML or LaTeX documents, bibliographies, and email messages. With the advent of the World Wide Web, structured text (in the form of HTML) has become a dominant medium for online information.

The popularity of text is easy to explain. As an old, standard data format, ASCII text can be viewed and edited easily on any platform. Text can be cut and pasted into any application, printed by any printer, included in any email message, and indexed by any search engine. Unix in particular has a rich set of generic tools for operating on text files: *grep*, *sort*, *uniq*, *sed*, etc.

Unfortunately, the generic nature of existing text-processing tools is also a weakness, because generic tools can make only limited assumptions about the format of the text. Most Unix tools assume that a text file is divided into records separated by newlines (or some other delimiter character). But this assumption breaks down for most kinds of structured text, such as source code and HTML. Consider the following tasks, which are difficult for generic text-processing tools to handle:

1. Find functions that call exit() in a program.
2. Check spelling in program comments.
3. Extract a bibliography from a Web page.
4. Sort a file of postal addresses by ZIP code.

The traditional approach to these problems is to custom-build a tool for a particular text format. For example, tasks #1 and #2 might be solved by a development environment customized for the programming language. Tasks #3 and #4 are typically solved by hand-coded Perl or AWK scripts. The problem with this approach is that custom-built programs require substantial investment, are difficult to reuse for other tasks or text formats, and lie beyond the ability of casual users to create.

The deficiencies of the custom-built approach are best highlighted by *custom* text structure – structure which has not been blessed by standard grammars or widely-available parsers. Many users store small databases (such as address lists) as text files. Many programs generate reports and logs in text form. Nearly every web page uses some kind of custom structure represented in HTML; examples include lists of publications, search engine results, product catalogs, news briefs, weather reports, stock quotes, sports scores, etc. Given the proliferation of custom text formats, developing a tool for every combination of task and text format is inconceivable.

Our approach to generic tools for structured text is called *lightweight structured text processing.* Lightweight structured text processing enables users to define custom text structure interactively and incrementally, so that generic tools can operate on the text in structured fashion. We envision that a lightweight structured text processing system would have four components:

- a *structure description language* for describing text structure;
- an *interactive document viewer* for viewing documents, developing and testing structure descriptions, and invoking tools;
- *parsers* for standard structures, like HTML and programming language syntax;
- *tools* for manipulating text using structure descriptions: sorting, searching, extracting, reformatting, editing, computing statistics, graphing, etc.

Following this plan, we have built a prototype system called LAPIS (Lightweight Architecture for Processing Information Structure). LAPIS includes a new structure description language called *text constraints.* Text constraints describe a set of regions in a document in terms of relational operators (like *before, after, in,* and *contains*) and primitive regions generated by external parsers or literal matching. Text constraints can be used not only for queries (such as `Function contains "exit"`) but also for structure definition, as in the following example:

```
Sentence = ends with SentencePunct;
SentencePunct = ('.' | '?' | '!'),
                just before Whitespace,
                but not '.' at end of
                       Abbreviation;
Abbreviation = 'Mr.' | 'Mrs.' |
               'Ms.' | 'Dr.' | ...;
```

Text constraints differ in several ways from context-free grammars and regular expressions (the traditional techniques for structure description). Text constraints permit conjunctions of patterns (indicated by commas in the previous example) and references to context (such as "just before"). Text constraints can also refer to structure defined by external parsers – even *multiple* parsers simultaneously. For example, `Line at start of Function` refers to both `Line` (a name defined by a line-scanning parser) and `Function` (defined by a programming-language parser) to match the first line of every function. Finally, we believe that text constraints are more readable and comprehensible for users than grammars or regular expressions, because a structure description can be reduced to a list of simple, intuitive constraints which can be read and understood individually. In the LAPIS prototype, text constraints are implemented as an algebra operating on sets of regions, using efficient set representations to achieve reasonable performance.

LAPIS combines text constraints with a web browser that allows the user to develop text constraints interactively and apply them to web pages, source code, and text files. In the browser, the user can describe a set of regions either programmatically (using text constraints or an external parser), manually (by selection), or using any combination of the two. Combining manual selection and programmatic description can be quite powerful. Manual selection can be used to restrict attention to part of a document which can be selected more easily than it can be described, such as the content area of a web page (omitting navigation bars and advertisements). Manual selection can also fix up errors made by an almost-correct structure description, adding or removing regions from the set as necessary. Relying on manual intervention is not always appropriate, but sometimes it can help finish a task faster.

The LAPIS browser also includes a few commands that operate on sets of regions. *Find* simply highlights and navigates through a set of regions. *Filter* displays only the selected regions, eliminating other text from the display. *Sort* displays a set of regions

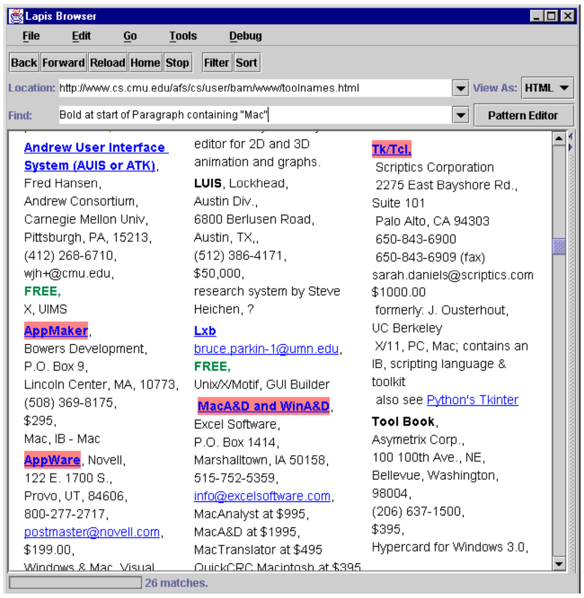

Figure 1: The LAPIS web browser, showing a web page that describes user interface toolkits. The user has entered the pattern `Bold at start of Paragraph containing "Mac"` to highlight the names of toolkits that support Macintosh development.

---

sorted by the value of a subfield. In LAPIS, these features are provided as interactive commands in the browser, but we also plan to implement batch-mode tools in the style of *grep* and *sort*, which would take as input a text file and its structure description.

The remainder of this paper is structured as follows: Section 2 describes the LAPIS browser and tools. Section 3 describes the text constraints language. Section 4 describes our current implementation of text constraints. Section 5 presents some applications of the system to web pages, text files, and source code. Section 6 covers related work, Section 7 describes future work, and Section 8 concludes.

2 LAPIS Web Browser

Our prototype lightweight structured text processing system is LAPIS, a web browser that has been extended with a pattern language (text constraints) and several generic text-processing tools. LAPIS is built on top of Sun's Java Foundation Classes. A screenshot of the browser is shown in Figure 1.

Like other web browsers, the LAPIS browser can retrieve any file that can be named by a URL and retrieved by HTTP, FTP, or from the local filesystem. The browser can display text files or HTML pages. HTML pages can be displayed either as text, which shows the source including tags, or as HTML, which renders the page according to the HTML formatting.

Several parsers are included in the browser, which run automatically when a page of a certain MIME type is loaded. A *parser* interprets a particular text format and labels its components in the document. The built-in parsers include:

- **HTML:** parses HTML pages, labeling HTML tags and elements while simultaneously building a parse tree for rendering the page;
- **Character:** parses plain text and HTML to find character classes like `Whitespace`, `Letters`, and `Digits`;
- **Java:** parses Java programs to find syntax constructs like `Class`, `Method`, `Statement`, and `Expression`;
- **USEnglish:** parses plain text and HTML to find regions like `Sentence`, `Line`, `Time`, `Date`, and `Currency`, according to conventions of American English.

Parsers can also be associated with URL patterns. For example, a parser that identifies components of an AltaVista search result page might be associated with URLs of the form `http://altavista.digital.com/*`.

New parsers can be defined in two ways: writing a Java class that implements our `Parser` interface, or by developing a system of text constraints. The HTML and Character parsers were written by hand in Java. The Java parser was automatically generated from an example grammar included with the JavaCC parser-generator [26], showing that LAPIS can take advantage of existing parsers without re-coding the grammar in text constraint expressions. USEnglish was developed interactively in LAPIS as a system of text constraints.

In the browser, the user can enter a text constraint expression and see the matching regions highlighted (see Figure 1). Highlighting is simple to implement and familiar to users, but unfortunately it merges adjacent and overlapping regions together, without distinguishing their endpoints. Future research should identify better ways to display overlapping region sets in context. To view highlighted regions, the user can either scroll the document or use the *Next Match* menu command to jump from one highlighted region to the next.

In addition to patterns, the user can also highlight regions by manual selection. In the prototype, a *selection* made with the mouse is distinct from the *highlighted region set* showing matches to a pattern. The selection is a single, contiguous region (colored blue), whereas the highlighted region set may be multiple, noncontiguous regions (colored red). The current selection in the document is always available as a one-element region set named `Selection`. By referring to `Selection` in a text constraint, for example, the user can limit the pattern's scope to a manually selected region of the document. The user can also construct a named region set by adding or removing regions. The *Label* menu command adds the current selection to the region set with the given name. A corresponding *Unlabel* command removes the selection from a given named region set by deleting regions that lie inside the selection and trimming the ends of regions that overlap the selection. By applying *Label* and *Unlabel* repeatedly to a sequence of selections, the user can build up a named region set by hand, or modify a named region set created by a parser or a pattern.

Several tools are provided for manipulating the highlighted regions. *Filter* eliminates all unhighlighted text from the display. By default, Filter inserts linebreaks between the highlighted regions to keep the display readable. Documents are filtered at the source text level – even HTML documents. The result is sometimes illegal HTML (with orphaned start tags or end tags), but the web browser can render it passably.

Like Filter, *Sort* filters the display down to highlighted regions, and also reorders the regions. Regions can be sorted alphabetically or numerically. By default, the sort key is the entire content of a region, but the user can provide an additional text constraint expression describing the sort field.

3 Text Constraints

Text constraints (TC) is a language for specifying text structure using relationships among regions (substrings of the text). TC describes a substring by specifying its start offset and end offset. Formally, a *region* is an interval $[b, e]$ of inter-character positions in a string, where $0 \leq b \leq e \leq n$ (n is the length of the string). A region $[b, e]$ identifies the substring that starts at the bth cursor position (just before the bth character of the string) and ends at the eth cursor position (just before the eth character, or at the end of the string if $e = n$). Thus the length of a region is $e - b$.

TC is essentially an algebra over sets of regions – operators take region sets as arguments and generate a region set as the result. TC permits an expression to match an arbitrary set of regions, unlike other structured text query languages that constrain region sets to certain types: nonoverlapping (regular expressions), nonnesting (GC-lists [5]), or hierarchical (Proximal Nodes [19]).

3.1 Primitives

TC has three primitive expressions: literals, regular expressions, and identifiers. A literal string enclosed in single or double quotes matches all occurrences of the string in the document. Thus `"Gettysburg"` finds all regions exactly matching the literal characters "Gettysburg". The literal matcher can generate overlapping regions, so matching `"aa"` against the string "aaaaa" would yield 4 regions.

A regular expression is indicated by `/regexp/`. Our regular expression matcher is based on the ORO-Matcher library for Java [20]. The library follows Perl 5 syntax and semantics [27], returning a set of nonoverlapping regions that are as long as possible.

An identifier is any whitespace-delimited token (except for words and punctuation reserved by TC operators). Identifiers refer to the named region sets generated by parsers. For example, after the HTML parser has run, `Tag` refers to the set of all HTML tags in the document. Only a single namespace is provided by the LAPIS prototype, so the names generated by different parsers must be chosen uniquely. A future version of LAPIS is expected to support multiple independent namespaces.

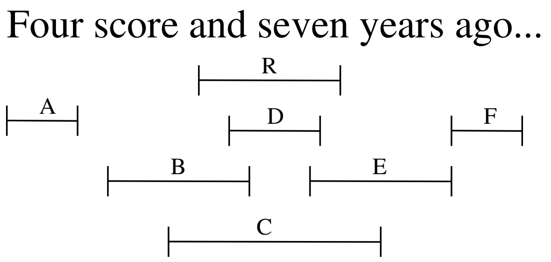

Figure 2: Fundamental region relations in an example string. Regions A through F are related to region R as follows: A *before* R; B *overlaps-start* R; C *contains* R; D *in* R; E *overlaps-end* R; and F *after* R.

3.2 Region Relations

TC operators are based on six fundamental binary relations among regions: *before*, *after*, *in*, *contains*, *overlaps-start*, and, *overlaps-end*. (Similar relations on time intervals were defined in [2].) The region relations are defined as follows:

$$\begin{array}{ll}
[b_1, e_1] \textit{ before } [b_2, e_2] & \Leftrightarrow e_1 \leq b_2 \\
[b_1, e_1] \textit{ after } [b_2, e_2] & \Leftrightarrow e_2 \leq b_1 \\
[b_1, e_1] \textit{ in } [b_2, e_2] & \Leftrightarrow b_2 \leq b_1 \wedge e_1 \leq e_2 \\
[b_1, e_1] \textit{ contains } [b_2, e_2] & \Leftrightarrow b_1 \leq b_2 \wedge e_2 \leq e_1 \\
[b_1, e_1] \textit{ overlaps-start } [b_2, e_2] & \Leftrightarrow b_1 \leq b_2 \wedge e_1 \leq e_2 \\
[b_1, e_1] \textit{ overlaps-end } [b_2, e_2] & \Leftrightarrow b_2 \leq b_1 \wedge e_2 \leq e_1
\end{array}$$

Note that *before* and *after* are inverses, as are *in* and *contains*, and *overlaps-start* and *overlaps-end*. The six region relations are illustrated in Figure 2.

The six region relations are complete in the sense that every ordered pair of regions is found in at least one of the relations. Some regions may be related in several ways, however. For example, in Figure 2, if A's end point were identical to R's start point, then we would have both A *before* R and A *overlaps-start* R. These relations are useful in pattern matching, so we define a set of derived relations in which regions have coincident endpoints:

$$\begin{array}{ll}
\textit{just-before} & = \textit{before} \cap \textit{overlaps-start} \\
\textit{just-after} & = \textit{after} \cap \textit{overlaps-end} \\
\textit{at-start-of} & = \textit{in} \cap \textit{overlaps-start} \\
\textit{at-end-of} & = \textit{in} \cap \textit{overlaps-end} \\
\textit{starts-with} & = \textit{contains} \cap \textit{overlaps-start} \\
\textit{ends-with} & = \textit{contains} \cap \textit{overlaps-end}
\end{array}$$

Figure 3 illustrates the derived relations.

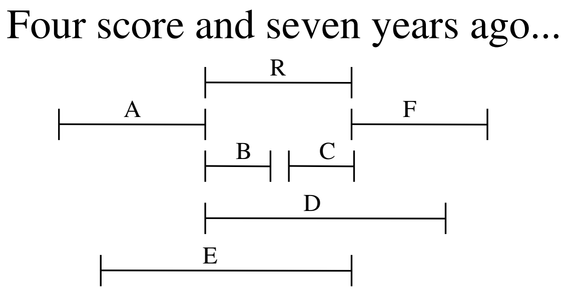

Figure 3: Region relations with coincident endpoints. Regions A through F are related to region R as follows: A *just-before* R; B *at-start-of* R; C *at-end-of* R; D *starts-with* R; E *ends-with* R; and F *just-after* R.

Another useful derived relation is `overlaps`:

$$\begin{array}{lcl}
\textit{overlaps} & = & \textit{in} \cup \textit{contains} \cup \\
 & & \textit{overlaps-start} \cup \textit{overlaps-end}
\end{array}$$

In Figure 2, the regions B, C, D, and E *overlap* R, but A and F do not. In Figure 3, all the regions *overlap* R.

3.3 Relational Operators

Each region relation corresponds to a relational operator in TC. Each relational operator takes two forms, one *unary* and the other *binary*. The unary form, ***op*** S, generates the set of regions that bear the relation *op* to some region matching S. For example, in an HTML document, the constraint expression `in Paragraph` returns all regions that are inside some paragraph element.

The binary form of a relational operator, R ***op*** S, generates all regions matching R that bear the relation *op* to some region matching S. For example, in HTML, `Paragraph contains "Lincoln"` returns all paragraph elements that contain the string "Lincoln."

For the sake of simplicity, all relational operators have equal precedence and right associativity, so that `X in Y in Z` is parsed as `X in (Y in Z)`.

3.4 Intersection, Union, and Difference

Constraints that must be simultaneously true of a region are expressed by separating the constraint expressions with commas. The region set matched by S_1 , S_2 , ... , S_n is the intersection of the region sets matched by each S_i. For example `just after "From:", just before "\n"` describes all regions that start immediately after a "From:" caption and end at a newline.

Alternative constraints are specified by separating the constraint expressions with "|". The region set matched by S_1 | S_2 | ... | S_n is the union of the region sets matched by each S_i.

Set difference is indicated by `but not`. The region set matched by S_1 `but not` S_2 is the set that matches S_1 after removing all regions that match S_2.

3.5 Delimiter Operators

When certain relational operators are intersected, the resulting region set can be larger than the user anticipates. For example, the expression `starts with` R`, ends with` S matches every possible pair of R and S, even if other R's and S's occur in between. For situations where only adjacent pairs are desired, any relational operator can be modified by the keyword `delimiter`. For example, `starts with delimiter` S matches regions that start with some region matching S and overlap no other region matching S.

3.6 Concatenation and Background

Concatenation of regions is indicated by `then`. The expression `"Gettysburg" then "Address"` matches regions that consist of "Gettysburg" followed by "Address", with nothing important in between. The meaning of *nothing important* depends on a parameter called the *background*. The background is a set of regions. Characters in the background regions are ignored when concatenating constraint expressions. For example, when the background is `Whitespace`, the expression `"Gettysburg" then "Address"` finds not only "GettysburgAddress", but also "Gettysburg Address", and even "Gettysburg Address" split across two lines. Relational operators that require adjacency also use the background, so the expression `"Gettysburg" just before "Address"` will successfully match the first word of "Gettysburg Address".

The LAPIS browser chooses a default background based on the current document view, following the guideline that any text not printed on the screen is part of the background. In the plain text view, the default background is `Whitespace`. In the HTML view, the default background is the union of `Whitespace` and `Tag`, since tags affect rendering but are not actually displayed.

The background can also be set explicitly using the `ignoring` directive. To change the background to R for the duration of a constraint expression *expr*, use the form *expr* `ignoring` R. For example, a query on source code might take the form *expr* `ignoring (Comment | Whitespace)`. The background can be removed by setting it to `nothing`, which generates the empty region set.

3.7 Definitions and Constraint Systems

A *constraint definition* assigns a name to the result of a constraint expression:

```
GettysburgAddress =
    starts with
        "Four score and seven years ago",
    ends with
        "shall not perish from the earth"
```

Region sets named by a constraint definition can be used in the same way as region sets named by a parser, as in the example `Sentence at start of GettysburgAddress`. A *constraint system* is a set of constraint definitions separated by semicolons.

3.8 Expressiveness

The theoretical power of TC — that is, the set of languages that can be matched by a TC expression — depends on the power of the matchers and parsers it uses. If its matchers and parsers generate only regular languages, then the TC expression is also regular, since regular languages are closed under the TC operators concatenation, intersection, and union [11]. Since context-free languages are not closed under intersection, however, a TC expression using context-free parsers may match a non-context-free language.

A TC constraint system that uses only literals (no regular expressions or external parsers) is less powerful than a regular expression, because TC lacks recursive constraints or repetition operators (such as the * operator). Future work discussed in Section 7 will address this issue.

4 Implementation

This section describes the implementation of text constraints used in LAPIS. Among the interesting features of the implementation is a novel region set representation, the *region interval*. Region intervals are particularly good at representing the result of a region relation operator. By a simple transformation, region intervals may be regarded as rectangles in two-dimensional space, allowing LAPIS to draw on earlier research in computational geometry to find a data structure suitable for storing and combining collections of region intervals.

4.1 Region Interval Representation

The key ingredient to an implementation of text constraints is choice of representation: how shall region sets be represented? One alternative is a bitvector, with one bit for each possible region in lexicographic order. With a bitvector representation, every region set requires $O(n^2)$ space, where n is the length of the document. Considering that the region sets generated by matchers and parsers typically have only $O(n)$ elements, the bitvector representation wastes space. Another alternative represents a region set as a list of explicit pairs $[b, e]$, which is more appropriate for sparse sets. Unfortunately the region sets generated by relational operators are *not* sparse. To choose a pathological example, `after` $[0, 0]$ matches every region in the document. In general, for any region relation `op` and region set S, the set matching `op` S may have $O(n^2)$ elements.

Other systems have dealt with this problem by restricting region sets to nested sets [19] or overlapped sets [5], sacrificing expressiveness for linear storage and processing. Instead of restricting region sets, we compress dense region sets with a representation called *region intervals*. A region interval is a quadruple $[b, c; d, e]$, representing the set of all regions $[x, y]$ such that $b \leq x \leq c$ and $d \leq y \leq e$. Essentially, a region interval is a set of regions whose starts and ends are given by intervals, rather than points. A region interval is depicted by extending the region notation for regions (|—|), replacing the vertical lines denoting the region's endpoints with boxes denoting intervals.

A few facts about region intervals follow immediately from the definition:

- The set of all regions in a string of length n can be represented by the region interval $[0, n; 0, n]$.
- The singleton region set $\{[b, e]\}$ is represented by the region interval $[b, b; e, e]$.
- A region interval represents the empty set if $b > c$ or $d > e$ or $b > e$.
- A region interval $[b_1, c_1; d_1, e_1]$ is a subset of another region interval $[b_2, c_2; d_2, e_2]$ if and only if $b_2 \leq b_1 \leq c_1 \leq c_2$ and $d_2 \leq d_1 \leq e_1 \leq e_2$.
- The intersection of two intervals $[b_1, c_1; d_1, e_1]$ and $[b_2, c_2; d_2, e_2]$ is

  $$[\max(b_1, b_2), \min(c_1, c_2);\ \max(d_1, d_2), \min(e_1, e_2)]$$

  which may of course be the empty set.

Region intervals are particularly useful for representing the result of applying a region relation operator. Given any region X and a region relation *op*, the set of regions which are related to X by *op* can be represented by exactly one region interval, as shown in Figure 4.

By extension, if a region relation operator is applied to a region set with m elements, then the result can be represented with m region intervals (possibly fewer, since some of the region intervals may be redundant).

This result extends to region intervals as well: applying a region relation operator to a region interval yields exactly one region interval. For example, the result of *before* $[b, c; d, e]$ is the set of all regions which lie before some region in $[b, c; d, e]$. Assuming the region interval is nonempty, every region ending at or before c qualifies, so the result of this operator can be described by the region interval $[0, c; 0, c]$.

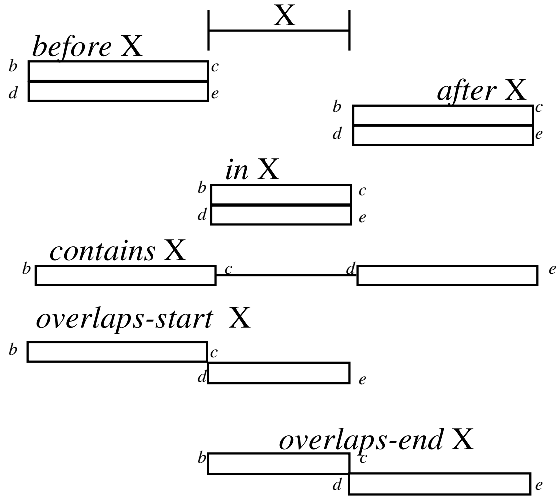

Figure 4: Region intervals corresponding to the relational operators.

We can represent an arbitrary region set by a union of region intervals, which is simply a collection of tuples. The size of this collection may still be $O(n^2)$ in the worst case (consider, for example, the set of all regions $[b, e]$ of even length, which must be represented by $O(n^2)$ singleton region intervals), but most collections are $O(n)$ in practice.

To summarize, the union-of-region-intervals representation enables a straightforward implementation of the text constraints language:

- **Primitives:** convert the set of regions generated by a literal, regular expression, or parser into a union of region intervals by replacing each region $[b, e]$ with the singleton region interval $[b, b; e, e]$.
- **Relational operators:** for each region interval $[b, c; d, e]$, compute a new region interval *op* $[b, c; d, e]$.
- **Union:** merge the two collections of region intervals, eliminating any region interval that is a subset of another.
- **Intersection:** intersect every possible pair of region intervals (one from each collection) and collect the results.

4.2 Region Space

It remains to choose a representation for the collection of region intervals that provides the operations we need (region relations, union, and intersection).

A 2D geometric interpretation of regions will prove helpful. Any region $[b, e]$ can be regarded as a point in the plane, where the x-coordinate indicates the start of the region and the y-coordinate indicates the end. We refer to this two-dimensional interpretation of regions as *region space* (see Figure 5). Strictly speaking, only points with integral coordinates correspond to regions, and even then only if they lie above the 45-degree line, where $b \leq e$.

Under this interpretation, a region interval $[b, c; d, e]$ corresponds to an axis-aligned rectangle in region space. Two region intervals intersect if and only if their region space rectangles intersect. A region interval is a subset of another if and only if its rectangle is completely enclosed in the other's rectangle.

A region set can be represented as a union of region intervals, which in turn can be represented as a union of axis-aligned rectangles in region space. We seek a data structure representing a union of rectangles with the following operations:

- **Create (P):** create a union of rectangles from a set of rectangles P.
- **Relation (op, R):** generate a new union of rectangles by applying a region relation operator *op* elementwise to R.
- **Union (R,S):** combine two unions of rectangles R and S.
- **Intersect (R,S):** intersect two unions of rectangles R and S.

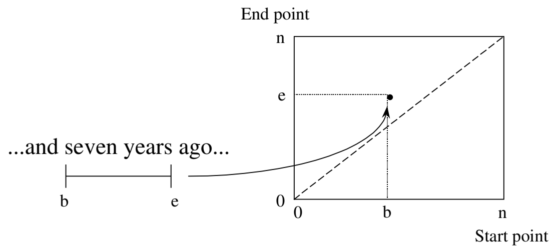

Figure 5: A region $[b, e]$ corresponds to a point in region space.

Ideally, these operations should take linear time and linear space. In other words, finding the intersection or union of a collection of M rectangles with a collection of N rectangles should take $O(N+M+F)$ time (where F is the number of rectangles in the result), and computing a region relation on a collection of N rectangles should take $O(N)$ time. The data structure itself should store N rectangles in $O(N)$ space.

Research in computational geometry and multidimensional databases has developed a variety of data structures and algorithms for storing and intersecting collections of rectangles, including plane-sweep algorithms, k-d trees, quadtrees of various kinds, R-trees , and R+-trees (see [24] for a survey).

LAPIS uses a variant of the R-tree [9]. The R-tree is a balanced tree derived from the B-tree, in which each internal node has between m and M children for some constants m and M. The tree is kept in balance by splitting overflowing nodes and merging underflowing nodes. Rectangles are associated with the leaf nodes, and each internal node stores the bounding box of all the rectangles in its subtree. The decomposition of space provided by an R-tree is adaptive (dependent on the rectangles stored) and overlapping (nodes in the tree may represent overlapping regions). To keep lookups fast, the R-tree insertion algorithm attempts to minimize the overlap and total area of nodes using various heuristics (for example, inserting a new rectangle in the subtree that would increase its overlap with its siblings by the least amount). One set of heuristics, called the R*-tree [4], has been empirically validated as reasonably efficient for random collections of rectangles. Initially we used the R*-tree heuristics in our prototype. The rectangle collections generated by text constraints are *not* particularly random, however; they tend to be distributed linearly along some dimension of region space, such as the 45-degree line, the x-axis, or the y-axis. We were able to improve overall performance by a factor of 5 by simply ordering the rectangles in lexicographic order, eliminating the expensive calculations that decide where to place a rectangle without sacrificing the tree's logarithmic decomposition of region space.

Two R-trees T_1 and T_2 can be intersected by traversing the trees in tandem, comparing the current T_1 node with the current T_2 node and expanding the nodes only if their bounding boxes overlap. Traversing the trees in tandem has the potential for pruning much of the search, since if two nodes high in each tree are found to be disjoint, the rectangles stored in their subtrees will never be compared. In practice, tandem tree intersection takes time $O(N+M+F)$. It will never do worse than $O(NM)$. Tandem tree traversal is effective for implementing set intersection, union, and difference.

4.3 Performance

The LAPIS prototype is written in Java 1.1. The core text constraints engine is implemented in about 3500 lines of code, not including matchers and parsers. The web browser consists of about 1000 lines of code on top of the JFC `JEditorPane` text component.

The text constraints engine can evaluate an operator at a typical rate of 20,000 regions per second, using Symantec JIT 3.0 on a 133 MHz Pentium. The actual evaluation time of a text constraint expression varies according to the complexity of the expression and the size of its intermediate results. The text constraint expressions used in the examples in Section 5 were all evaluated in less than 0.1 second, on text files and web pages ranging up to 80KB in size.

5 Applications

5.1 Web Pages

Many web pages display data in a custom format, using HTML markup to set off important parts of the text typographically or spatially. Figure 6 shows part of a page describing user interface toolkits [17]

The page describes over 100 toolkits with various properties: some are free, some are commercial; some run on Unix, others Microsoft Windows, others Macintosh, and others are cross-platform. To browse the page conveniently, we might want to restrict the display to show only toolkits matching certain requirements – for example, toolkits running under both Unix and Microsoft Windows, sorted by price.

Each toolkit on this page is contained in a single paragraph (<P> element in HTML). So we might start by describing the toolkit as the Paragraph element, which is identified by the built-in HTML parser:

```
Toolkit = Paragraph
```

AlphaWindow,
Cumulus Technology Corp.,
1007 Elwell Court,
Palo Alto, CA, 94303,
(415) 960-1200,
$750,
Unix, **Discontinued**,
Alpha-numeric terminal windows, Window System

Altia Design, Altia,
5030 Corporate Plaza Dr #300,
Colorado Springs, CO, 80919,
(800)653-9957 or (719)598-4299,
UNIX or Windows, IB

Amulet,
Brad Myers,
Human-Computer Interaction Institute,
Carnegie Mellon Univ,
Pittsburgh, PA, 15213,
(412) 268-5150,
amulet@cs.cmu.edu,
FREE,
X or MS Windows, portable toolkit, UIMS

Figure 6: Excerpt from a web page describing user interface toolkits.

Finding the prices is straightforward using `Number`, a region set identified by the built-in `USEnglish` parser:

```
Price = ("\$" then Number | "FREE")
        in Toolkit;
```

Finding toolkits that run under Macintosh is easy (`Toolkit contains "Mac"`), since the page refers consistently to Macintosh as "Mac". But Unix platforms are sometimes described as "X", "X Windows", or "Motif", and Microsoft Windows is also called "MS Windows" or just plain "Windows". We deal with these problems by defining a constraint for each kind of platform that specifies all these possibilities and further constrains the matched literal to be a full `Word` (not just part of a word):

```
Macintosh = Word, "Mac";
Unix = Word, ("Unix" | "X" | "Motif");
MSWindows = Word, ("PC" |
      "Windows" but not just after "X");
```

Using these definitions, we can readily filter the web page for toolkits matching a certain requirements (`Toolkit, contains Unix, contains MSWindows`) and sort them according to `Price`.

5.2 Plain Text

Plain text has less explicit structure than HTML, so text constraints for plain text typically refer to delimiters like punctuation marks and line breaks. Consider the following example of processing email messages. Several airlines distribute weekly email announcing low-price airfares. An excerpt from one message (from US Airways) is shown in Figure 7.

Describing the boundaries of the table itself is fairly straightforward given the delimiters (`BlankLine` is identified by the built-in `USEnglish` parser):

```
Table = starts with delimiter
          "Roundtrip Fares Departing From",
        ends with delimiter BlankLine;
```

The rows of the table can be found using `Line`, also identified by the built-in parser:

```
Flight = Line starts with "\$" in Table;
Fare = Number just after "\$" in Flight;
```

The origin and destination cities can be described in terms of their boundaries:

```
Origin = just after delimiter "From",
         just before delimiter "To",
         in Line at start of Table;
Destination = just after Price,
              in Flight;
```

```
Roundtrip Fares Departing From BOSTON, MA To
-----------------------------------------------
       $109          INDIANAPOLIS, IN
       $89           PITTSBURGH, PA

Roundtrip Fares Departing From PHILADELPHIA, PA To
-----------------------------------------------
       $79           BUFFALO, NY
       $89           CLEVELAND, OH
       $89           COLUMBUS, OH
       $89           DAYTON, OH
       $89           DETROIT, MI
       $79           PITTSBURGH, PA
       $79           RICHMOND/WMBG., VA
       $79           SYRACUSE, NY
```

Figure 7: Excerpt from an email message announcing cheap airfares.

```
/**
 * Convert a local filename to a URL.
 * @param file File to convert
 * @return URL corresponding to file
 */
public static URL FileToURL (File file)
        throws MalformedURLException {
    return new URL ("file:"
        + toURLDelimiters
             (file.getAbsolutePath ()));
}
```

Figure 8: A Java method with a documentation comment.

Using these definitions, we can readily filter the message for flights of interest, e.g. from Boston to Pittsburgh:

```
Flight,
contains Destination contains "PITTSBURGH",
in Table contains Origin contains "BOSTON";
```

The expression for the flight's origin is somewhat convoluted because flights (which are rows of the table) do not contain the origin as a field, but rather inherit it from the heading of the table. This example demonstrates, however, that useful structure can be described and queried with a small set of relational operators.

5.3 Source Code

Source code can be processed like plain text, but with a parser for the programming language, source code can be queried much more easily. LAPIS includes a Java parser, so the examples that follow are in Java.

Unlike other systems for querying and processing source code, TC operates on regions in the source text, not on an abstract syntax tree. At the text level, the user can achieve substantial mileage knowing only a few general types of regions identified by the parser, such as `Statement`, `Comment`, `Expression`, and `Method`, and using text constraints to specialize them. For example, our parser identifies `Comment` regions, but does not specially distinguish the "documentation comments" that can be automatically extracted by the `javadoc` utility. Figure 8 shows a Java method preceded by a documentation comment.

The user can find the documentation comments by constraining `Comment` with a text-level expression:

```
DocComment = Comment starts with "/**";
```

A similar technique can be used to distinguish public class methods from private methods:

```
PublicMethod = Method starts with "public";
```

In this case, however, the accuracy of the pattern depends on programmer convention, since attributes like `public` may appear in any order in a method declaration, not necessarily first. All of the following method declarations are equivalent in Java:

```
public static synchronized void f ()
static public synchronized void f ()
synchronized static public void f ()
```

If necessary, the user can deal with this problem by adjusting the pattern (e.g., `Method starts with Line contains "public"`) or relying on the Java parser to identify attribute regions (e.g., `Method contains Attribute contains "public"`). In practice, however, it is often more convenient to use typographic conventions, like `public` always appearing first, than to modify the parser for every contingency. Since text constraints can express such conventions, constraints might also be used to enforce them, if desired.

We can use `DocComment` and `PublicMethod` to find public methods that need documentation:

```
PublicMethod but not just after DocComment;
```

Text constraints are also useful for defining custom structure inside source code. Java documentation comments can include various kinds of fields, such as `@param` to describe method parameters, `@return` to describe the return value, and `@exception` to describe exceptional return conditions. These fields can be described by text constraint expressions:

```
DocField = starts with delimiter "@",
           in DocComment;
ParamDoc = DocField, starts with "@param";
ReturnDoc = DocField, starts with "@return";
ExceptionDoc = DocField, starts with
                          "@exception";
```

Using this structure, we can find methods whose documentation is incomplete in various ways. For example, this expression finds methods with parameters but no parameter documentation:

```
PublicMethod contains FormalParameter,
  just after (DocComment but not
              contains ParamDoc);
```

6 Related Work

Text processing is a rich and varied field. Languages like AWK [1] and Perl [27] are popular tools providing fast regular expression matching in an imperative programming language designed for text processing. These tools are not interactive, however, sacrificing the ability to view pattern matches in context (particularly important for web pages) and the ability to combine manual selection with programmatic selection. Visual Awk [15] made some strides toward interactive development of AWK programs which was inspirational for this work, but Visual AWK is still line-oriented, limited to regular expression patterns, and unable to use external parsers.

The concept of lightweight structured text processing described in this paper is independent of the language chosen for structure description. The text constraints language in LAPIS is novel and appealing for its simple and intuitive operators, its uniform treatment of parser-generated regions and constraint-generated regions, the concept of background regions, and its direct implementation, but another language may be used instead. A variety of languages have been proposed for querying structured text databases, such as Proximal Nodes [19], GC-lists [5], p-strings [8], tree inclusion [13], Maestro [16], and PAT expressions [23]. A survey of structured text query languages is found in [3]. Sgrep [12] is a variant of *grep* that uses a structured text query language instead of regular expressions, which helped inspire us to incorporate other Unix-style tools into a structured text processing system. Domain-specific query tools include ASTLOG [6], a query language specific to source code, and WebL [14], which combines an HTML query language with a programming language specialized for fetching and processing World Wide Web pages.

Structured text editors are a common form of structured text processing, but lacking the "lightweightness" that enables users to construct structure descriptions interactively. Examples of structured text editors include Gandalf [10], GRIF [22], and to some extent, EMACS [25]. These systems accept a structure description and provide tools for editing documents that follow the structure. The structure description is generally a variant of context-free grammar, although EMACS uses regular expressions to describe syntax coloring. EMACS is unusual in another sense, too: unlike structured text editors that enforce syntactic correctness at all times, EMACS uses the structure description to assist editing where possible, but does not prevent the user from entering free text. Our LAPIS system follows this philosophy, allowing the user to describe and access the document as free text, as structured text, or any combination of the two.

Sam [21] combines an interactive editor with a command language that manipulates regions matching regular expressions. Regular expressions can be pipelined to automatically process multiline structure in ways that line-oriented systems cannot. Unlike LAPIS, however, Sam does not provide mechanisms for naming, composing, and reusing the structure described by its regular expressions.

Also related are recent efforts to build structure-aware user interfaces, such as Cyberdesk [7] and Apple Data Detectors [18]. These systems associate actions with text structure, so that URLs might be associated with the "open in browser" action, and email addresses with "compose a message" or "look up phone number." When a URL or email address is selected by the user, its associated actions become available in the user interface. Action association is a useful tool that might be incorporated in LAPIS, but unlike LAPIS, these other systems use traditional structure description languages like context-free grammars and regular expressions.

7 Future Work

This work is part of the first author's PhD thesis research, and continues to evolve. This section describes some of the directions in which the work will be taken in the coming months.

LAPIS will be extended with new matchers, parsers, and tools. A more useful matcher for literals would optionally ignore alphabetic case, optionally match only full words, match spaces in the literal expression against any background character, and optionally do simple stemming. Parser support would be improved by allowing parsers to operate on limited parts of the document – for example, applying an HTML parser only to Java documentation comments, which may contain HTML tags. Useful new tools would include computing statistics on region sets (such as counts, sums, and averages) and reformatting text by template substitution.

Another fruitful area for research is integration of lightweight structured text processing into other applications, in particular an extensible text editor such as EMACS. Integration with a text editor poses at least two challenges: the interface problem of using named region sets fluidly in direct-manipulation text editing, and the implementation problem of updating region sets cheaply as the user edits.

The text constraint language has room for improvement. It should be possible to count (e.g. `2nd Line in Table`) and use numeric operators (e.g. `Toolkit contains Price < 100`). Constraint systems should support recursive or mutually recursive definitions. It would also be useful to precede a constraint expression by a *fuzzy qualifier*, such as `always`, `usually`, `rarely`, or `never`. A fuzzy qualifier describes how important it is for a matching region to satisfy the constraint. Finally, it will be important to determine the conditions under which our text contraints implementation (tandem tree intersection) runs in linear time.

8 Conclusions

This paper has described lightweight structured text processing, a technique for allowing users to define and manipulate text structure interactively. A prototype system, LAPIS, was described and evaluated on example applications, including web pages, source code, and plain text. LAPIS includes a structure description language called text constraints, which can express text structure in terms of relationships among regions.

The LAPIS prototype has several important advantages over other systems. First is the ability to handle *custom structure* with a simple language accessible to users. The second advantage is *interactive specification*, which allows users to see pattern matches in context and define text structure by the most convenient combination of manual selection and pattern matching. Finally, LAPIS supports *external parsers*, giving the user leverage over standard text formats, supporting existing parsers without recoding them in a new grammar language, and allowing the user to write patterns that refer to multiple parse trees at once.

Availability

The LAPIS prototype described in this paper, including Java source code, is available free from `http://www.cs.cmu.edu/~rcm/lapis/`.

Acknowledgements

For help with this paper, the authors would like to thank David Garlan, Laura Cassenti, and the anonymous referees.

This research was partially supported by a USENIX Student Research Grant, and partially by a National Defense Science and Engineering Graduate Fellowship. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied of the U.S. Government.

References

[1] Aho, A.V., Kernighan, B.W., and Weinberger, P.J. *The AWK Programming Language.* Addison-Wesley, 1988.

[2] Allen, J. "Time Intervals." *Communications of the ACM*, v26 n11, 1983, pp 822-843.

[3] Baeza-Yates, R. and Navarro, G. "Integrating contents and structure in text retrieval." *ACM SIGMOD Record*, v25 n1, March 1996, pp 67-79.

[4] Beckmann, N., Kriegel, H-P., Schneider, R., and Seeger, B. "The R*-tree: an efficient and robust access method for points and rectangles." *ACM SIGMOD Intl Conf on Managment of Data*, 1990, pp 322-331.

[5] Clarke, C.L.A., Cormack, G.V., Burkowski, F.J. "An algebra for structured text search and a framework for its implementation." *The Computer Journal*, v38 n1, 1995, pp 43-56.

[6] Crew, R. F. "ASTLOG: a language for examining abstract syntax trees." *Proceedings of the*

USENIX Conference on Domain-Specific Languages, October 1997, pp 229-242.

[7] Dey, A.K., Abowd, G.A., and Wood, A. "CyberDesk: a framework for providing self-integrating ubiquitous software services." *Proceedings of Intelligent User Interfaces '98*, January 1998.

[8] Gonnet, G. H. and Tompa, F. W. "Mind your grammar: a new approach to modelling text." *Proceedings 13th VLDB Conference*, 1987, pp 339-345.

[9] Guttman, A. "R-Tree: a dynamic index structure for spatial searching." *ACM SIGMOD Intl Conf on Managment of Data*, 1984, pp 47-57.

[10] Habermann, N. and Notkin, D. "Gandalf: Software development environments." *IEEE Transactions on Software Engineering.* v12 n12, December 1986, pp 1117-1127.

[11] Hopcroft, J.E. and Ullman, J.D. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[12] Jaakkola, J. and Kilpelainen, P. *Using sgrep for querying structured text files.* University of Helsinki, Department of Computer Science, Report C-1996-83, November 1996.

[13] Kilpelainen, P. and Mannila, H. "Retrieval from hierarchical texts by partial patterns." *Proceedings SIGIR '93*, pp 214-222, 1993.

[14] Kistler, T. and Marais, H. "WebL - a programming language for the Web." In *Computer Networks and ISDN Systems* (*Proceedings of the WWW7 Conference*), v30, April 1998, pp 259-270. Also appeared as DEC SRC Technical Note 1997-029.

[15] Landauer, J. and Hirakawa, M. "Visual AWK: a model for text processing by demonstration." *Proceedings 11th International IEEE Symposium on Visual Languages '95*, September 1995. `http://www.computer.org/conferen/vl95/talks/T32.html`

[16] MacLeod, I. "A query language for retrieving information from hierarchic text structures." *The Computer Journal,* v34 n3, 1991, pp 254-264.

[17] Myers, B.A. *User Interface Software Tools.* `http://www.cs.cmu.edu/~bam/toolnames.html`

[18] Nardi, B.A., Miller, J.R., and Wright, D.J. "Collaborative, programmable intelligent agents." *Communications of the ACM*, v41 n3, March 1998, pp 96-104.

[19] Navarro, G. and Baeza-Yates, R. "A language for queries on structure and contents of textual databases." *Proceedings SIGIR'95*, pp 93-101.

[20] Original Reusable Objects, Inc. *OROMatcher.* `http://www.oroinc.com/`

[21] Pike, R. "The Text Editor sam." *Software Practice & Experience*, v17 n11, Nov 1987, pp 813-845.

[22] Quint, V. and Vatton, I. "Grif: an interactive system for structured document manipulation." *Text Processing and Document Manipulation, Proceedings of the International Conference*, Cambridge University Press, 1986, pp 200-213.

[23] Salminen, A. and Tompa, F. W. *PAT expressions: an algebra for text search.* UW Centre for the New Oxford English Dictionary and Text Research Report OED-92-02, 1992.

[24] Samet, H. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA, 1990.

[25] Stallman, R.M. "EMACS - the extensible, customizable self-documenting display editor." *SIGPLAN Notices*, v16 n6, June 1981, pp 147-56.

[26] Sun Microsystems, Inc. *JavaCC.* `http://www.suntest.com/JavaCC/`

[27] Wall, L., Christiansen, T., and Schwartz, R.L. *Programming Perl*, 2nd ed. O'Reilly & Associates, 1996.