Chapter 2

Related Work

This chapter surveys previous work in structured text processing. A common theme in much of this work is a choice between two approaches: the *syntactic* approach and the *lexical* approach.

In general terms, the syntactic approach uses a formal, hierarchical definition of text structure, invariably some form of grammar. Syntactic systems generally parse the text into a tree of elements, called a *syntax tree*, and then search, edit, or otherwise manipulate the tree.

The lexical approach, on the other hand, is less formal and rarely hierarchical. Lexical systems generally use regular expressions or a similar pattern language to describe structure. Instead of parsing text into a hierarchical tree, lexical systems treat the text as a sequence of flat segments, such as characters, tokens, or lines.

The syntactic approach is generally more expressive, since grammars can capture aspects of hierarchical text structure, particularly arbitrary nesting, that the lexical approach cannot. However, the lexical approach is generally better at handling structure that is only partially described. Other differences between the two approaches will be seen in the sections below.

2.1 Parser Generators

Parser generators are perhaps the earliest systems for describing and processing structured text. Lex [Les75] generates a lexical analyzer from a declarative specification consisting of a set of regular expressions describing tokens. Each token is associated with an action, which is a statement or block of statements written in C. A Lex specification is translated into a finite state machine that reads a stream of text, splits it into tokens, and invokes the corresponding action for each token. Lex is a canonical example of the lexical approach to text processing.

Lex is often used as a front-end for Yacc [Joh75]. A Yacc specification consists of a set of context-free grammar productions, each of which is associated with an action written in C. Yacc translates this specification into a parser that executes the associated action whenever it parses a production. In a Yacc specification designed for a compiler, the actions typically generate tree nodes and connect them together into a tree. Yacc is a good example of the syntactic approach.

Since Lex and Yacc are frequently used in concert — Lex for tokenization and Yacc for parsing — recent systems have recognized the value of combining both facilities into a single specification. In the parser generators JavaCC [Jav00] and ANTLR [Sch99], a single specification describes both tokens and nonterminals, the former by regular expressions and the latter by grammar productions.

JavaCC and ANTLR also support annotations that generate tree nodes automatically, so that there is no need to write action code for the common case of generating a syntax tree.

Parser generators like Lex and Yacc do not encourage reuse, because the structure descriptions — regular expressions and grammar productions — are mixed up with the actions that implement a particular task. A Yacc grammar designed for one task must be substantially modified to be applied to another task, even if the text structure is the same in both cases. As a result, parser generators are generally used for building compilers and tools that merit the substantial investment in developing or adapting the specifications. Parser generators are less often used for one-off tasks like searching source code, calculating code metrics, and global search-and-replace. In this sense, parser generators are a *heavyweight* approach to describing and processing text structure.

Lightweight structure offers a way to reuse the structure knowledge implicit in a parser generator specification. After a one-time investment of resources to retarget the specification so that it produces region sets, the generated parser can be installed in the structure library and used in a variety of tasks with no further development effort. Java syntax was added to LAPIS in precisely this way, using a Java parser generated by JavaCC (Section 6.2.4).

2.2 Markup Languages

Markup languages, such as SGML [Gol90] and its successor XML [W3C00], are a popular way to represent structured documents. SGML/XML use explicit delimiters, called *tags*, to represent structure in the document. A specification called a *document type definition* (DTD) defines the markup tags and specifies legal combinations of tags. A DTD is essentially a grammar over tags; the structure of text between tags is not described by the DTD. Given a document and a DTD that describes it, SGML/XML tools can parse the document, *validate* it (checking its syntax), and manipulate the document in a variety of ways.

SGML has been used mainly for publishing and documentation. Its most widespread application is the HTML format found on the World Wide Web. HTML documents are SGML documents that conform to the HTML DTD (although, strictly speaking, many HTML web pages do not actually conform). XML, which is a simpler subset of SGML, is seeing much wider application — not only for delivering hypertext content, but also for configuration files, application data files, remote procedure calls, and online data exchange.

Markup languages separate structure description, represented by the DTD, from structure manipulation, represented by the tools that operate on structured documents. Lightweight structure provides a similar separation. However, markup languages require explicit markup in the text to be processed. Text which is not marked up, like source code or plain text, cannot be processed.

One solution to this problem is to add explicit markup automatically. This approach is taken by JavaML [Bad99], which translates Java source code into XML. JavaML has been shown to be useful for tasks like calculating source code metrics, renaming variables and methods safely, and adding instrumentation code. However, JavaML requires its user to buy into a new representation for Java source code which is not shared by other Java tools. The user could convert back and forth between JavaML and conventional Java syntax, but the conversion process only preserves syntactic features, not lexical features like whitespace, indentation, and comment placement.

Even when explicit markup is available in a document, it may not completely describe the text structure. For example, suppose a DTD specifies that phone numbers are identified by a `<phonenumber>` tag, like this:

```
<phonenumber>+1 412-247-3940</phonenumber>
```

This level of markup allows entire phone numbers to be located and manipulated, but it does not represent the internal structure of a phone number, such as the country code or area code. Markup languages must make a tradeoff between ease of authoring and ease of processing — minimal markup makes authoring easier, but more thorough markup makes more structure available for processing.

Lightweight structure can use explicit markup to define structure abstractions, without sacrificing the ability to describe structure in other ways, such as regular expressions. LAPIS includes an HTML parser in its library.

2.3 Structured Text Editors

A number of tools have been designed to tackle the problem of editing structured text, particularly program source code. The evolution of structured text editors highlights some of the key differences between the syntactic and lexical approach.

Early structured editors, often called *syntax-directed editors*, followed the syntactic approach. Examples include Emily [Han71], the Cornell Program Synthesizer [TRH81], and Gandalf [HN86]. In a syntax-directed editor, text structure is specified by an annotated grammar that describes both the *parsing* process, which converts text into an abstract syntax tree, and the *unparsing* process, which renders the abstract syntax tree into formatted text for display on the screen. In a syntax-directed editor, the user can only select and edit complete syntactic units that correspond to subtrees, such as expressions, statements, or declarations. New tree nodes are created by commands that produce a template for the user to fill in. In order to make structural changes, e.g., changing a `while` loop into a `repeat-until` loop, the user must either apply an editing command specialized for the transformation, or else recreate the loop from scratch. The awkwardness of this style of editing is one of the criticisms that has been leveled at syntax-directed editors.

Other editors take a lexical approach to editing program source code. Rather than representing language syntax in detail, editors like Z [Woo81] and EMACS [Sta81] use lexical descriptions to provide some of the features of a syntax-directed editor while allowing the user to edit the program text in a freeform way. For example, when the user presses Enter to start a new line in a C program, Z automatically indents the next line depending on the last token of the previous line. For example, if the last token is `{`, then the next line is indented one level deeper than the previous line. Similarly, EMACS uses regular expressions to render program syntax in different fonts and colors, and offers commands that select and step through program syntax like statements and expressions. All these editing features are based only on local, lexical descriptions of structure, not on a global parse of the program, so these editors can be fooled into highlighting or selecting syntactically incorrect structure.

Defenses of each approach have been offered by Waters [Wat82] and Minor [Min92]. Arguing for plain text editors at a time when syntax-directed editors were in vogue, Waters addressed some commonly-voiced objections to plain text editing:

- *The text-oriented approach is obsolete.* Waters answers this objection by pointing out that plain text editing is familiar to users, and there is no point in throwing away an effective technique without good reason. His argument resounds even more strongly today, since an enormous base of computer users have become intimately familiar with plain text editing from email, instant messaging, and the World Wide Web. It seems preferable to use techniques that most users already know and understand.

- *Text-oriented editing is dangerous, because it does not guarantee syntactic consistency.* Waters points out that a plain text editor can use a parser to check syntax and highlight syntax errors in context, so syntax errors can at least be corrected. Unlike the syntax-directed approach, however, the text-oriented approach cannot prevent the errors in the first place.

- *Text-oriented editing is incompatible with syntax-directed editing.* In other words, implementing both approaches in the same editor is too hard. Waters agrees that hybrid editors are difficult but not impossible to build. In fact, several hybrid editors have been built since then, some of which are discussed below.

Arguing for the other side a decade later, after most programmers had chosen text-oriented editors over syntax-directed ones, Minor took up some common objections to syntax-directed editing:

- *Syntax-directed editing is awkward, particularly for expressions.* Minor responds to this point by arguing for a hybrid approach in which expressions are edited as plain text and then reparsed into a syntax tree. This hybrid approach is described in more detail below.

- *Enforced syntactic consistency limits the user's freedom to pass through inconsistent intermediate states.* Minor answers this objection by claiming that users do not care about this freedom in and of itself; what they care about is convenient editing. If syntax-directed operations can be made as convenient as plain text editing, then users will not care whether the editor permits syntactic inconsistency.

- *Expert users don't need syntactic guidance.* Syntax-directed editing is particularly helpful for novice programmers because it prevents syntax errors, by design. For experts, however, this assistance may be less valuable, and possibly inhibiting. Minor suggests the tradeoff here is similar to the tradeoff between menus and command languages in user interface design. Menus are easier for novices to learn, but commands allow experts to achieve higher performance. Well-designed systems should support both novices and experts, so Minor argues for a hybrid editor that provides syntax-directed editing for novices but less constrained editing for experts.

In fact, both Waters and Minor were arguing for the same thing: a hybrid approach that combines the best features of syntax orientation and text orientation. Several editors have taken this approach, either by adding freeform text editing to a primarily syntax-directed editor, or vice versa. The Synthesizer Generator [RT89], the successor of the Cornell Program Synthesizer, is a syntax-directed editor that permits limited freeform text editing. This editing is done by selecting a subtree of the syntax tree — such as an expression, a statement, or a function body — and editing its unparsed representation. When the user signals that editing is complete, usually by navigating to another subtree in the program, the edited representation is reparsed into tree nodes. Thus

the Synthesizer Generator's primary representation is a tree. In contrast, Pan [VGB92] is a text-oriented editor, with an incremental parser that creates a syntax tree on demand when the user invokes syntax-directed commands.

Lightweight structure offers another kind of hybrid approach. Like the text-oriented approach, the primary representation in lightweight structure is a string of characters, not a tree. Syntax-directed features are available on demand, implemented by parsers or patterns in the structure library. Unlike other hybrids, however, lightweight structure also combines the structure description methods of the two approaches. Structure can be described in syntactic fashion, as in syntax-directed editors, or in lexical fashion, as in text-oriented editors, or in some combination, according to the level of convenience, tolerance, or precision demanded by a particular task.

LAPIS is not a complete programming editor, like EMACS or Gandalf. LAPIS does not provide all the features programmers have come to expect from such editors, such as automatic indentation, syntax highlighting, and template generation. Although these features could be provided by lightweight structure, they lie outside the application areas targeted by this thesis, so they are left for future work.

2.4 Structure-Aware User Interfaces

A recent trend in structured text processing are systems that use recognized structure to trigger user interface actions. This category, which might be called *structure-aware user interfaces*, includes the Intel Selection Recognition Agent [PK97], Cyberdesk [DAW98], Apple Data Detectors [NMW98], LiveDoc [MB98a], and Microsoft Smart Tags [Mic02].

A structure-aware user interface has a collection of *structure detectors* (e.g., for email addresses or URLs), plus a collection of *actions* that can be performed on each kind of structure (e.g., sending an email message or browsing to a URL). When the user makes a text selection in an application, the system automatically runs structure detectors on the selection and builds a list of possible actions. The list of actions is then popped up as a menu.

The structure detectors in a structure-aware user interface can be implemented by arbitrary patterns and parsers, just like lightweight structure abstractions. Unlike lightweight structure, however, these structure detectors are not designed for reuse. For example, the structure detector for email addresses cannot be used in a search-and-replace pattern, or constrained so that it only matches email addresses from `cmu.edu`.

2.5 Pattern Matching

The oldest text pattern language is probably SNOBOL [GPP71], a string manipulation language with an embedded pattern language. Its successor was Icon [GG83], which integrated pattern matching more tightly with other language features by supporting goal-directed evaluation. Icon allows programmers to define their own pattern-matching operators, a degree of expressiveness seen in few other pattern-matching systems.

Today, most text-processing languages use regular expressions for text pattern matching, the most prominent examples being awk [AKW88] and Perl [WCS96]. Regular expressions are fast, easy to implement, and extremely compact, at the cost of being cryptic. In an effort to address

this problem, several researchers have designed visual languages for regular expressions [JB97, Bla01], finding that users can generate and comprehend the visual representations better than the standard textual representations. This research does not address more fundamental problems with regular expressions, however — namely, that regular expressions are too low-level, insufficiently expressive, and fail to encourage reuse.

The preceding pattern matching systems take a lexical approach, searching for patterns in a string of characters. *Structured text query languages* take a syntactic approach instead, searching for patterns in a syntax tree. A variety of query languages of this sort have been proposed, including p-strings [GT87], Maestro [Mac91], PAT expressions [ST92], tree inclusion [KM93], Proximal Nodes [NBY95], GC-lists [CCB95], and sgrep [JK96]. Baeza-Yates and Navarro give a good survey of structured text query languages [BYN96]. Some structured text query languages are designed expressly for searching SGML/XML markup, such as OmniMark [Omn99].

Markup systems have found other uses for pattern matching beyond searching. For example, DSSSL [Cov96], CSS [BLLJ98], and XSLT [Kay01] use patterns in stylesheets to describe how SGML, HTML, and XML documents (respectively) should be rendered on screen or on paper. Each rule in a stylesheet consists of a pattern and a set of rendering properties, such as color, font, or positioning information. Elements that match the pattern are assigned the corresponding properties. The pattern can constrain not only the element type, but also its attributes, its context (parent element), and its content (child elements and text). In XML, patterns are also used for hyperlinking. XLink [DMO01] uses the same pattern language as XSLT to specify hyperlink targets in XML documents.

Several pattern languages have been proposed for searching and transforming source code. Some take the syntactic approach, in which patterns are matched against a syntax tree produced from the source. Tawk [GAM96] uses the pattern-action model of awk, but patterns are syntactic tree patterns instead of regular expressions. ASTLOG [Cre97] is a Prolog-based pattern language in which pattern operators are Prolog predicates over tree nodes. Like Icon, ASTLOG allows users to define new pattern operators.

The lexical approach has also been used in source code processing, notably by the Lexical Source Model Extraction (LSME) system [MN96]. LSME is a pattern-action system similar to awk, in which patterns are regular expressions over program tokens, tested one line at a time, and actions consist of Icon code. Unlike awk, however, LSME supports hierarchical rules, in which child rules do not become active until the parent rule has matched. Hierarchical rules make it possible to generate a procedure call graph, for example, using a parent rule to match a procedure declaration and a child rule to find calls to other procedures within its body.

Finally, it is worth mentioning *aspect-oriented programming* [KLM+97], which uses pattern matching to insert code at various points in an object-oriented program. Unlike the previous systems discussed in this section, the patterns used in aspect-oriented programming can incorporate semantic information as well as syntactic. Aspect-oriented patterns can also refer to dynamic properties of a running program, such as the ancestors of the current method on the call stack. Other source code transformation systems — including LAPIS — are entirely static.

In comparison to all these pattern languages, the text constraints (TC) language described in this thesis is the first text pattern language that combines the syntactic and lexical approaches into a single unified framework. TC includes some operators that are syntactic in nature — particularly the relations *in* and *contains*, which appear in most structured text query languages, but not in regular expressions. It also includes operators that are lexical in nature — particularly concatena-

tion, which is a fundamental operation in regular expressions but is impossible to represent with a tree-based pattern that can only match one tree node. Furthermore, the region algebra underlying TC can express many of the operators found in these other pattern languages, although not all have been implemented in TC. More details about these operators can be found in Chapter 3.

However, the region algebra cannot express arbitrary repetition, like the Kleene star used in regular expressions. As a result, TC expressions that use only literal string matching are weaker than regular expressions (Chapter 5). In practice, TC expressions are used to combine and reuse structure abstractions defined by more powerful pattern languages, such as regular expressions and grammars, so this weakness is not a serious problem. Nevertheless, adding the Kleene star to the region algebra would be an interesting avenue of future work.

2.6 Web Automation

Web automation — automatically browsing the World Wide Web and extracting data – is one of the problems to which lightweight structure is applied by this thesis. Another approach to this problem is *macro recording*, typified by LiveAgent [Kru97]. LiveAgent automates a web-browsing task by recording a sequence of browsing actions in Netscape through a local HTTP proxy. Macro recording requires little learning on the part of the user, but recorded macros suffer from limited expressiveness, often lacking variables, conditionals, and iteration.

Another approach is *scripting*, writing a program in a scripting language such as Perl, Tcl, or Python. These scripting languages are fully expressive, Turing-complete programming languages, but programs written in these languages must be developed, tested, and invoked outside the web browser, making them difficult to incorporate into a web user's work flow. The overhead of switching to an external scripting language tends to discourage spur-of-the-moment automation, in which interactive operations might be mixed with automation in order to finish a task more quickly.

A particularly relevant scripting language is WebL [KM98], which provides high-level *service combinators* for invoking web services and a *markup algebra*, essentially a structured text query language, for extracting results from HTML. Like other scripting languages, WebL lacks tight integration with a web browser, forcing a user to study the HTML source of a web service to develop markup patterns and reverse-engineer form interfaces. In LAPIS, web automation can be performed while viewing rendered web pages in the browser, and simple tasks can be automated by demonstrating the steps on examples.

Other systems have tackled web automation by demonstration. Turquoise [MM97] and Internet Scrapbook [SK98] construct a *personalized newspaper*, a dynamic collage of pieces clipped from other web pages, by generalizing from a cut-and-paste demonstration. SPHINX [MB98b] creates a web crawler by demonstration, learning which URLs to follow from positive and negative examples.

2.7 Repetitive Text Editing

Users have a rich basket of tools for automating repetitive editing tasks. *Find-and-replace*, in which the user specifies a pattern to search for and replacement text to be substituted, is good enough for simple tasks. *Keyboard macros* are another technique, in which the user records a

sequence of keystrokes (or editing commands) and binds the sequence to a single command for easy reexecution. Most keyboard macro systems also support simple loops using tail recursion, where the last step in the macro reinvokes the macro. For more complicated tasks, however, users may resort to writing a script in a text-processing language like awk or Perl.

Sam [Pik87] and its successor Acme [Pik94] combine 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 like awk cannot. Unlike LAPIS, however, Sam and Acme do not provide mechanisms for naming, composing, and reusing the structure described by the regular expressions.

Another approach to the problem of repetitive text editing is *programming by example*, also called *programming by demonstration* (PBD). In PBD, the user demonstrates one or more examples of the transformation in a text editor, and the system generalizes this demonstration into a program that can be applied to the rest of the examples. PBD systems for text editing have included EBE [Nix85], Tourmaline [Mye93], TELS [WM93], Eager [Cyp93], Cima [Mau94], DEED [Fuj98], and SmartEDIT [LWDW01].

Unlike LAPIS, none of these systems used multiple selection for editing or feedback about inferences. Multiple selections completely reshape the dialog between a PBD system and its user. While a traditional PBD system reveals its predictions one example at a time, multiple selections allow the system to expose all its predictions simultaneously. The user can look and see that the system's inference is correct, at least for the current set of examples, which in many tasks is all that matters. Novel forms of intelligent assistance, such as outlier highlighting, can help the user find inference errors. The user can correct erroneous predictions in *any* order, not just the order chosen by the PBD system. Alternative hypotheses can be presented not only abstractly, as a data description or pattern, but also concretely, as a multiple selection. If the desired concept is unlearnable, the user may still be able to get close enough and fix the remaining mispredictions by hand, without stopping the demonstration.

The system that most closely resembles multiple-selection editing is Visual Awk [LH95]. Visual Awk allows a user to create awk-like file transformers interactively. Like awk, Visual Awk's basic structure consists of lines and words. When the user selects one or more words in a line, the system highlights the words at the same position in all other lines. For other kinds of selections, however, the user must select the appropriate tool: e.g., Cutter selects by character offset, and Matcher selects matches to a regular expression. In contrast, LAPIS is designed around a conventional text editor, operates on arbitrary records (not just lines), uses standard editing commands like copy and paste, and infers selections from examples.

2.8 Inferring Text Patterns From Examples

LAPIS can infer a text pattern from multiple examples given by the user (Chapter 9). Several other authors have tackled this problem as well. Kushmerick [KWD97] showed how to learn lexical patterns, which he called "wrappers", that extract data from structured text like web pages and seminar announcements. Freitag [Fre98] compared several approaches to learning wrappers, including a naive Bayesian classifier and a relational learner. Both Kushmerick and Freitag use statistical methods that need a large number of labeled examples to learn a concept, often 20 or

more. This approach is hard to apply in an interactive interface, where user become frustrated when forced to give more than a handful of examples.

Most PBD systems for text editing infer patterns from examples. The most advanced PBD inference algorithm is probably Cima [Mau94], which combines a powerful disjunctive normal form (DNF) learner with a dynamic bias that allows Cima to vary the space of features and hypotheses it considers in response to *hints* given by the user. Another PBD approach is found in the Grammex system [LNW98], which infers context-free grammars from a small number of examples chosen and annotated by the user.

Compared to these systems, LAPIS is the first that can learn patterns over arbitrary structure abstractions. Other approaches learn from a limited set of low-level features, such as literals, character classes (e.g., alphanumeric or digit), and capitalization. Patterns learned by LAPIS can use any abstraction in the structure library, including complex syntax like Java and HTML that would be infeasible to learn from examples.

2.9 Error Detection

LAPIS includes a novel feature called *outlier finding* that uses lightweight structure to detect unusual selections that might be erroneous (Chapter 10). The notion of an outlier comes from the field of statistics, which defines it as a data point which appears to be inconsistent with the rest of the data. Most work on outliers focuses on statistical tests that justify omitting outliers from experimental data [BL84]. A large number of statistical tests have been developed for various probability distributions. For outliers in text selections or pattern matches, however, the distribution is rarely simple and almost always unknown. The LAPIS outlier finder cannot make strong statistical claims like "this outlier is 95% likely to be an error," but on the other hand it can be applied more widely, with no assumptions about the distribution of the data.

Outlier finding has been applied to data mining by Knorr and Ng [KN98]. They propose a "distance-based" definition of an outlier, which is similar to the approach taken in this thesis. They define a $DB(p, D)$ outlier as a data object that lies at least a distance D (in feature space) from at least a fraction p of the rest of the data set. The choice of p and D is left to a human expert. The algorithm described in this thesis is simpler for inexpert users because it merely ranks outliers along a single dimension of "weirdness". Users don't need to understand the details of the outlier finder to use it, and appropriate weights and parameters are determined automatically by the algorithm.

The outlier finding algorithm draws on techniques better known in the machine learning community as *clustering* [And73]. In clustering, objects are classified into groups by a similarity measure computed from features of the object. Clustering is commonly used in information retrieval to find similar documents, representing each document by a vector of terms and computing similarity between term vectors by Euclidean distance or cosine measures. LAPIS is concerned with matching small parts of documents rather than whole documents, so term vectors are less suitable as a representation. Freitag confirmed this hypothesis in his study of inductive learning for information extraction [Fre98], which showed that a relational learner using features similar to the LAPIS outlier finder was much more effective at learning rules to extract fields from text than a term-vector-based learner.

One way to find borderline mismatches in text pattern matching is to allow errors in the pattern match. This is the approach taken by *agrep* [WM92], which allows a bounded number of errors

(insertions, deletions, or substitutions) when it matches a pattern. Agrep is particularly useful for searching documents which may contain spelling errors.

Spelling and grammar checking are well-known ways to find errors in text editing. Microsoft Word pioneered the idea of using these checkers in the background, highlighting possible errors with a jagged underline as the user types. Although spell-checking and outlier-finding both have the same goal – reducing errors – the approaches are drastically different. Spelling and grammar checkers compare the text with a known model, such as a dictionary or language grammar. Outlier finding has no model. Instead, it assumes that the text is mostly correct already, and searches for exceptions and irregularities. Whereas a conventional spell-checker would be flummoxed by text that diverges drastically from the model — such as Lewis Carroll's "The Jabberwocky" [Car72] — a spell checker based on outlier-finding might notice that one occurrence of "Jabberwock" has been mistyped because it is spelled differently from the rest. On the other hand, an outlier-finding spell checker would overlook systematic spelling errors. Morris and Cherry built an outlier-finding spell-checker [MC74] that computes trigram frequencies for a document and then sorts the document's words by their trigram probability, and found that it worked well on technical documents. The LAPIS outlier finder has not been applied to spell-checking, but doing so would be interesting future work.