Text Structure Recognition using a Region Algebra

Matthew Young-Lai

Department of Computer Science

University of Waterloo

Waterloo, Ontario, Canada, 2001

Technical Report Number: CS-2001-06

This report is a reprint of a thesis that embodies the results of research done in partial fulfillment of the requirements for the degree of Doctor of Philosophy of Mathematics in Computer Science at the University of Waterloo.

Abstract

We consider the problem of incrementally developing a parser for text structure. This means building the parser specification a piece at a time while simultaneously developing our understanding of the text.

We argue that existing solutions have usability and efficiency problems for this application and propose an alternative approach based on the type of region algebra that is often used as a query language for text databases. This is an appropriate interface for incremental development, but has no efficient batch parsing model such as those that exist for grammars. In this thesis, we propose an efficient batch parsing model and characterize the region algebras to which it applies.

Acknowledgements

I would like to thank my supervisor, Dr. Frank Tompa, and the members of my examining committee: Dr. Gordon V. Cormack, Dr. Kostos Kontogiannis, Dr. Jeffrey O. Shallit, and Dr. David T. Barnard of the University of Regina. Also, thanks to Dr. Howard Johnson for reading and commenting on an early draft of this thesis. Finally, I gratefully acknowledge financial assistance from the Natural Sciences and Engineering Research Council of Canada, the Institute for Computer Research, and the University of Waterloo.

Contents

Chapter 1

Introduction

Consider the problem of recognizing structure in text that contains markup. Without loss of generality, we take *text* to mean human readable characters (binary codes can easily be converted to such a representation). By *structure* we mean the breakup of text into elements. For example, elements may be paragraphs, sections, and chapters for text like this thesis, or head-words, etymologies, usages, and pronunciations for text from a dictionary. We use the term *markup* to mean any characters or codes interspersed with the text that tell us something about its structure while being separable from it at a conceptual level (Coombs, Renear, & DeRose, 1987). Figures 1.1, 1.2, and 1.3 are examples of marked up text.

By *recognition*, we mean the process of parsing a text into substrings that correspond to structure elements, and associating types with those elements. Often, the term *parsing* is used specifically to apply to recognition performed using a context-free grammar. We do not limit it in this way. From this perspective, our usage of the term can be considered an expanded definition.

```
+1000 00000000000 1 Malison
+PR (m+23 +11 list+21 n),
+PS sb.
+LA arch. +R and +I dial.
+VL Forms: 4 +B malisun(e, malysun, malesun, maliscun, malescun,
malicun, malicoun, +R 4+14 5 +B malyson(e,malisoun(e, +R 4+14 6
+B malysoun, +R 5+14 6 +B maleso(u)n(e, +R 6 +B malisone, +R 7
+B mallison, +R 4+14 +0 +B malison.
+ET +OB a. OF. +I maleison +R: +13 L. +I maledictio+1 n-em
+SC Malediction. +EB
```

Figure 1.1: Text fragment from the *Oxford English Dictionary (OED)* as keyed in from the printed version. Tags beginning with a "+" are codes in a markup system adapted for the *OED* computerization project.

Developing parsers for complex text is a difficult problem, especially if the text was created outside computer control and without the intent of supporting automated recognition. One example of the effort that can be needed to construct a grammar for complex text is found in the *Oxford English Dictionary* computerization project. Kazman (1986) describes the process of building a grammar for over 500 Mb of text like the excerpt shown in Figure 1.1. Overall, the process took about six months of work to bring to the state described in Kazman's thesis, and Oxford University Press continued to refine the result for some time thereafter.

While one source of difficulty in the project was understanding low-level details about the markup and structure in the text, the hardest part was constructing a grammar to describe those details. Grammars are an adequate specification mechanism for simple recognition tasks that do not involve much debugging. In this case, we can break the process into discrete steps:

```
.so /usr/share/lib/tmac/sml
.so /usr/share/lib/tmac/rsml
.SH NAME
.PP
\*Lls\*O \- Lists and generates statistics for files
.SH SYNOPSIS
.PP
.sS
\*Lls\*O
\*O[\*L-aAbcCdfFgilLmnopqrRstux1\*O]
\*O[\*Vfile\*O
\&...
|
\*Vdirectory\*O
\&...]
.sE
.PP
The \*Lls\*O command writes to standard output the contents of each
specified directory or the name of each specified
file, along with any other information you ask for with flags.
If you do not specify a file or a directory,
\*Lls\*O displays the contents of the current directory.
.SH FLAGS
.PP
.VL 4m
.LI "\*L-a\*O"
Lists all entries in the directory, including the entries that begin
with a \*L.\*O (dot). Entries that begin with a . are not displayed
unless you refer to them specifically, or you specify the \*L-a\*O
flag.
```

Figure 1.2: Part of a file for the troff typesetting system. Tags beginning with a period, and most non-alphabetic characters are markup with special meaning to troff.

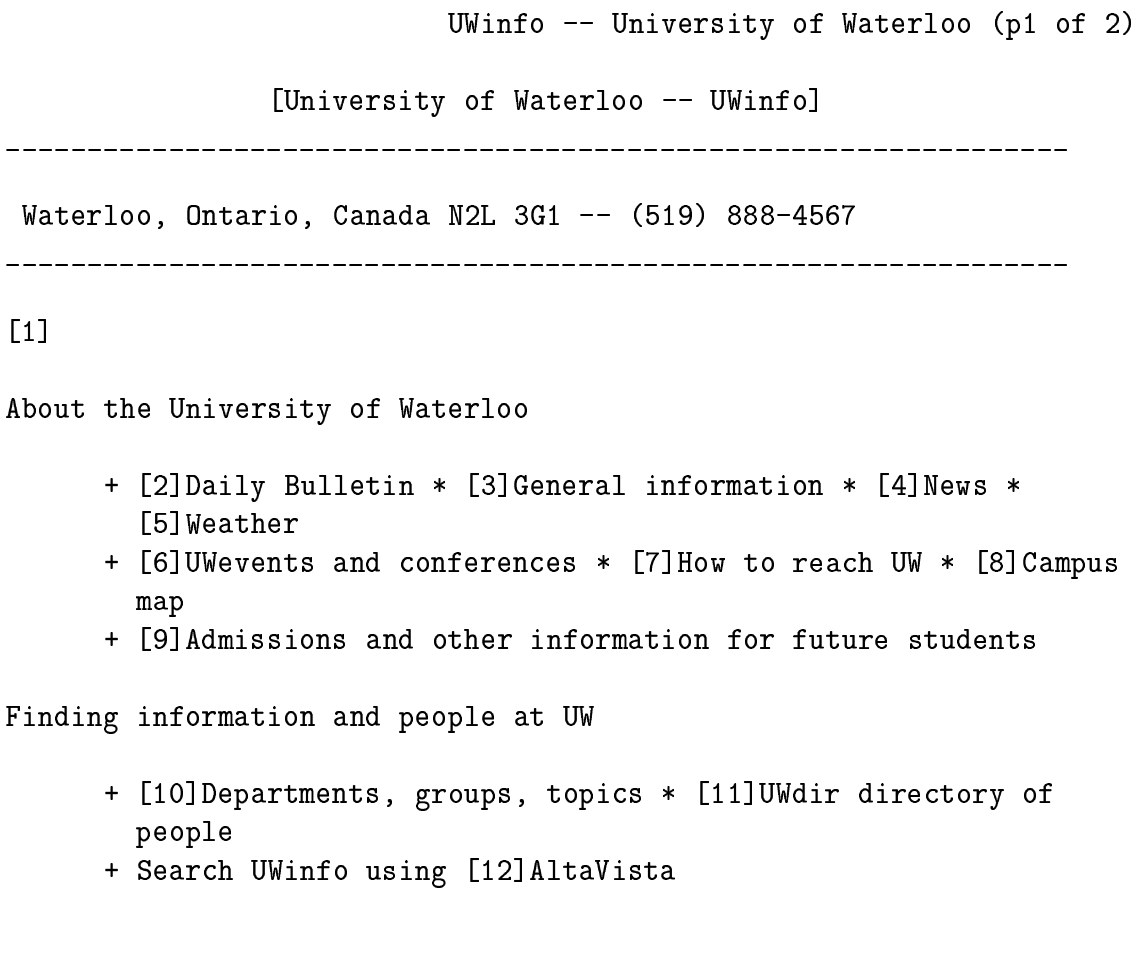


Figure 1.3: A web page rendered using lynx. Empty lines, indentation, whitespace, numbered links, horizontal lines, and punctuation are all examples of markup.

1. Understand the structure and markup in the data.

2. Express that understanding in a specification.

3. Use the specification to parse the data.

For complex data, however, it is not realistic to perform these steps separately. Instead, we need to gradually evolve our understanding of the structure and markup, build the specification a piece at a time, and debug errors in the specification. Thus, the process is an ongoing loop rather than a series of discrete steps: we form hypotheses about the data, write and debug specification fragments, see the results, and either update our hypotheses based on these results or continue forming new hypotheses. We term such a process *incremental* specification.

The motivation for this thesis is our premise that existing approaches are poorly suited to the task of incrementally developing large parser specifications. This is a view that we justify further in Chapter 3, after presenting background definitions and concepts in Chapter 2. We continue in Chapter 3 by describing our proposed approach based on a region algebra. A limitation of such an interface is that it is oriented towards interpreted, step-by-step evaluation. We therefore propose an efficient batch parsing model in Chapter 5, after giving examples of efficient models with similar characteristics in Chapter 4. We continue in Chapter 5 by describing how the requirements of the parsing model restrict the design of the algebraic specification language. Chapter 6 gives an example that illustrates the utility of the overall approach. Chapter 7 lists conclusions and possibilities for future work.

Chapter 2

Background

2.1 Conversion

Recognition is a sub-problem of conversion, which can be broken into the following parts:

1. **recognition** — identifying parts of the text that correspond to structural elements

2. **string transformation** — inserting, deleting, moving, or replacing substrings of the text

3. **structure transformation** — inserting, deleting, moving, or replacing structural elements

4. **schema generation** — constructing a grammar or other type of schema to describe correct usage of structural elements

Existing conversion approaches can be classified into the following categories:

1. tagging by hand or with SGML (or equivalent) editing software
2. custom programming with a general purpose programming language
3. manual specification using a pattern matching model
4. manual specification using a one-grammar model
5. manual specification using a two-grammar model
6. automatic learning and inference approaches

The first two approaches are unsuited to large, complex data, and we do not consider them further.

The manual specification approaches use a compiler-compiler paradigm: provide a tool that takes a specification and generates a compiler, which is then used to transform source code (unconverted text) into target code (converted text).

In the pattern matching specification model, the inputs are a text consisting of one long string, a set of patterns in some pattern language, and an action associated with each pattern. Examples of pattern matching systems are SNOBOL (Gimpel, 1973), **lex** (Lesk & Schmidt, 1984), **sed** (Dougherty, 1991), **awk** (Aho, Kernighan, & Weinberger, 1978), **perl**[1] (Wall, Schwartz, Christiansen, & Potter, 1996), DSSSL (ISO, 1996), XSLT (Clark, 1999), TranSid (Lindén, 1997), GOEDEL (Blake, Bray, & Tompa, 1992), and the TSIMMIS web interface (Hammer, Garcia-Molina, Cho,

[1]Despite being intended for more general purposes, the powerful string facilities of **perl** make it very well suited for conversion using the pattern matching model.

Aranha, & Crespo, 1997). In all these approaches, when a pattern matches a text segment, a corresponding action is performed. Characteristics such as precedence between patterns, whether the search is done left to right, lookahead, etc., are all defined by the pattern language. Also defined is the order in which matches for different patterns are generated (which affects the order in which actions are performed). Actions may be restricted, or they may be arbitrarily complex. Actions in **sed**, for example, are restricted to simple editing operations such as cut, copy, paste, delete, or replace; actions in **lex**, on the other hand, are specified in C and can be completely arbitrary.

In a one-grammar model, the inputs are a text, a grammar, and actions associated with specified parsing steps. Examples of one-grammar systems are yacc (Johnson, 1975), SGMLC (SGML Systems Engineering Ltd.,), Omnimark (Exoterica Corporation, 1993), DREAM (Göttke & Fankhauser, 1992), and INR (Johnson, 1989). Depending on the exact model, actions may be associated with productions (e.g., attribute assignments in attribute grammars), with terminals (e.g., local string substitution), or with the start and end points of substrings corresponding to non-terminals (e.g., inserting start and end tags). Actions are applied while the grammar is being used to parse the text, or afterwards when the complete parse tree is available.

In a two-grammar model, the inputs are a text, a source grammar, a target grammar, and rules for converting between the two. Examples of two-grammar systems are Chameleon (Mamrak, O'Connell, & Barnes, 1992), Alchemist (Lindén, 1997), and Grif (Quint & Vatton, 1986). The text is parsed according to the source

grammar. The rules specify how to rearrange subtrees in the resulting parse tree to give a tree conformant with the target grammar. Types of subtrees that can be specified depend on the specific formalism. For example, syntax directed translation schemas (SDTSs) only allow transformation of depth-one trees corresponding to productions (Lewis & Stearns, 1968), whereas text transformation (TT) grammars allow transformation of arbitrary subtrees (Keller, Perkins, Payton, & Mardinly, 1984).

Learning systems are based on the artificial intelligence sub-field of machine learning (Michalski, Carbonell, & Mitchell, 1983). Examples include Markitup! (Fankhauser & Xu, 1993), MINI-EDIT (Mo & Witten, 1992), U (Nix, 1989), STALKER (Muslea, Minton, & Knoblock, 1998), NoDose(Adelberg, 1998), WIEN (Kushmerick, Weld, & Doorenbos, 1997), Ariadne (Knoblock, Minton, Ambite, & Ashish, 1998), TexTamer (Reed-Lade, 1989), mod-ALERGIA (Young-Lai & Tompa, 2000), XTRACT (Garofalakis, Gionis, Rastogi, Seshadri, & Shim, 2000), and the work of Ashish and Knoblock (Ashish & Knoblock, 1997). They use the same models as specification approaches, but, rather than manually constructing a specification, the user demonstrates the desired results for a few examples and the system infers the underlying rules. This is then applied to the remaining data. Some learning approaches request clarifications and use them to evolve the specification whenever problems are encountered. This helps to overcome limitations of the learning model.

While learning approaches require less user effort than manual specification, they accomplish this by trading off flexibility (Crespo, Jannink, Neuhold, Rys, &

Studer, 2000). Generally speaking, learning approaches are only feasible if the training data, the user interaction, the learning method, and the target format are simple and uniform. We are interested in problems that do not have these characteristics, and therefore do not consider learning approaches further.

Overall, pattern matching and one-grammar approaches are most applicable to the recognition sub-problem. They are therefore the approaches we are interested in examining in this thesis. Note that both are also well suited to the text transformation sub-problem, while one-grammar and two-grammar systems are applicable to structure transformation.

2.2 Parser Efficiency

Recall that our expanded definition of a *parser* includes any entity that performs recognition. Informally, recognition takes a string, finds substrings of interest, and associates types with them. Formally, we define a parser to be a mapping from strings to sets of *regions*. A region is a *(type,left,right)* triple representing a substring labeled *type* that starts at position *left* in the string and continues to position *right*.

The size of a particular parsing problem is characterized by three values: n is the number of characters in the input string, m is the size of the output set of regions, and σ is the size of the parser specification. In these terms, the approximate size of the *OED* parsing problem, for example, is: $n \approx 10^8$, $m \approx 10^6$, and $\sigma \approx 10^3$.

Consider the following model of computation: we have a computer with a random access main memory and a secondary storage. The secondary storage is much

larger and much slower to access than main memory. Within this model, we can characterize the efficiency of an algorithm by: 1) the asymptotic upper bound on the time used to compute the mapping, 2) the asymptotic upper bound on the number of characters that are read and written from secondary storage (I/O), and 3) the asymptotic upper bound on the number of characters that are stored in main memory at any one time.

With current computing and secondary storage technology, an algorithm for parsing an input comparable in size to the *OED* should, at worst, have the following asymptotic bounds: $O(\sigma n + \sigma m)$ time, $O(n + m + \sigma)$ I/O, and $O(\sigma)$ memory. In practice, bounds any larger than this are too costly, and even large constant multipliers may not be acceptable. Note that the memory bound implies that the input must reside on secondary storage, and the output should end up there.

2.3 Regular Expressions and Finite Automata

Hopcroft and Ullman (1979) provide a standard account of regular expressions and languages. Formally, we define regular expressions over an alphabet Σ recursively as follows:

- If $a \in \Sigma$ then a is a regular expression representing the language $\{a\}$.
- If p and q are regular expressions for the languages $L(p)$ and $L(q)$, then their concatenation $p \circ q$ (abbreviated pq) is a regular expression for the language $L(p)L(q) = \{xy \mid x \in L(p) \wedge y \in L(q)\}$.
- If p and q are regular expressions for the languages $L(p)$ and $L(q)$, then $p \mid q$

is a regular expression for the language $L(p) \cup L(q)$.

- If p is a regular expression for the language L, then the *Kleene closure*, p^* is a regular expression for the language $\bigcup_{i=0}^{\infty} L^i$, where $L^0 = \{\epsilon\}$ and $L^i = LL^{i-1}$.
- If p is a regular expression for the language L, then the *positive closure*, p^+ is a regular expression for the language $\bigcup_{i=1}^{\infty} L^i$.

When writing a regular expression, brackets can be used to clarify precedence. Otherwise the order of precedence from highest to lowest is assumed to be $*, +, \circ, |$.

A non-deterministic finite automaton (NFA) is a tuple $M = (Q, \Sigma, \delta, q_0, F)$, where Q is a set of states, Σ is a finite input alphabet, δ is a transition function from $Q \times (\Sigma \cup \{\epsilon\})$ to the power set of Q, q_0 is the start state, and F is a set of final states. A directed graph is associated with an NFA as follows: the vertices of the graph correspond to states in Q. If there is a transition from state p to state q on input a, then there is an arc labeled a from the vertex corresponding to state p to the vertex for state q in the graph. For a string $x \in \Sigma^*$, we write $q \in \delta(r, x)$ or $((r, x) \rightarrow q) \in \delta$ if a sequence of transitions corresponding to the symbols of x leads from r to q (including any number of edges labeled with ϵ, the empty string). An NFA accepts a string x if $(\delta(q_0, x) \cap F) \neq \emptyset$.

Regular expressions and non-deterministic finite automata (NFAs) are equivalent mechanisms for defining regular languages. Regular expressions are a more convenient specification mechanism, but NFAs are the more convenient representation for manipulation and for finding matches. In practice, most regular expression matching strategies first convert a given regular expression to an equivalent NFA. This can be done with a simple construction (Hopcroft & Ullman, 1979; Aho,

Hopcroft, & Ullman, 1974). Given a regular expression r, the size of the NFA produced with this construction is $O(|r|)$.

2.4 Rational Relations and Functions

The formal definitions of a *monoid*, *morphism*, and *rational relation* are as follows:

Definition: A *monoid* $(M, \circ, s)$ is a non-empty set M with one binary operator $\circ$ and a constant element s such that $\forall a, b, c \in M$:

$$a \circ (b \circ c) = (a \circ b) \circ c$$

$$s \circ a = a = a \circ s$$

Definition: A *morphism* f between monoids $(M_1, \circ_1, s_1)$ and $(M_2, \circ_2, s_2)$ is a function defined on M_1 such that $\forall a, b \in M_1$:

$$f(s_1) = s_2$$

$$f(a \circ_1 b) = f(a) \circ_2 f(b)$$

Definition: A *rational relation* is defined as follows: given two alphabets Σ and Δ, a relation $R \subseteq \Sigma^* \times \Delta^*$ is *rational* if there is an alphabet Φ, a regular language $L \subseteq \Phi^*$, and two morphisms $\alpha : \Phi^* \to \Sigma^*$ and $\beta : \Phi^* \to \Delta^*$ such that $(x, y) \in R$ if and only if there is a $z \in L$ such that $x = \alpha(z)$ and $y = \beta(z)$.

A thorough discussion of rational relations can be found in (Eilenberg, 1974) or (Berstel, 1979). We can think of a rational relation in a static sense as a subset of the Cartesian product of two sets, or in a dynamic sense as a mapping from the first set to the set of subsets of the second.

A rational relation can be computed by a *finite transducer* that inputs a string from Σ^* and outputs a string from Δ^*. There are several equivalent forms of finite transducers (see (Johnson, 1983), for example). One is an NFA with the transition function redefined to be from $Q \times \Sigma$ to $Q \times \Delta^*$. This is the form of finite transducer we assume from now on. There are many ways to specify finite transducers (for example, INR (Johnson, 1983), or the finite-state calculus described by Karttunen (Karttunen, 1992, 1995, 1996)). We can test (x, y) for membership in a rational relation specified as a finite transducer in $O(|x| \times |y|)$ time and $O(\min(|x|, |y|)$ memory (Johnson (1983) gives an overview of the relevant algorithm).

Now consider rational relations where each x is associated with at most one y:

> **Definition:** A *rational function* f from Σ^* to Δ^* is a rational relation on $\Sigma^* \times \Delta^*$ that is a partial function. Thus for every $x \in \mathrm{dom}(f)$ there is at most one y such that $(x, y) \in f$.

A rational function can be computed more efficiently than a general rational relation as a consequence of the following theorem (Berstel, 1979):

Theorem 1 *Any rational function can be expressed as a length-preserving right-left sequential function composed with a left-right sequential function.*

A *sequential function* is a rational function that can be computed by a *sequential transducer*, which is a *deterministic* finite transducer with no distinguished set of

final states (any string corresponding to a path through a sequential transducer is accepted). A finite transducer is deterministic if $\delta(q, a)$ contains at most one state for any state q and character a.

A left-right sequential transducer reads its input from left to right and writes its output from left to right as well. A right-left sequential transducer reads and writes from right to left. Thus any rational function can be computed using a two-pass algorithm in $O(|x| + |y|)$ time, and $O(|x| + |y|)$ memory. Alternatively, the input and output can reside on secondary storage giving a two-pass algorithm with $O(|x| + |y|)$ time, $O(|x| + |y|)$ I/O, and $O(1)$ memory.

Another way of defining a rational function is to define an arbitrary finite transducer and use appropriate rules to choose a unique output during simulation (Johnson, 1987). This was the approach used for the *OED* computerization project (Kazman, 1986), for example.

2.5 Context-Free Grammars

A context-free grammar G is a 4-tuple (V, T, P, S), where V and T are disjoint, finite sets of variables and terminals. P is a finite set of productions, and S is a special variable called the start symbol. Each production is of the form $A \rightarrow \alpha$ where A is a variable and α is a string of symbols from $(V \cup T)^*$. A is called the *left-hand side* of the production and α is the *right-hand side*. See Hopcroft and Ullman (1979) for an overview of the properties of context-free grammars.

A string is in the language of a grammar if and only if a parse tree for that string can be built using the productions of the grammar. Such a parse tree is

of the following form: the root is labeled with S, internal nodes are labeled with variable names, and leaves are labeled with terminals. The labels on the terminals read from left to right must equal the string, and for any internal node labeled A with children $X_1, X_2, \ldots X_k$ there must be a production $A \rightarrow X_1 X_2 \ldots X_k$ in P. A parse tree has at most $2n$ nodes where n is the length of the string if there are no "useless" productions that simply rename non-terminals.

The process of finding a parse tree for a given grammar and string, if one exists, is what is normally defined as parsing (our definition includes this one). In general context-free parsing can be done in $O(n^3)$ time and $O(n^2)$ memory, where n is the size of the input string (Earley, 1970). Algorithms exist that achieve slightly better asymptotic time bounds but have too much overhead to be useful except for inputs much larger than are of practical interest (Valiant, 1975). Parsing algorithms also exist that use $O(n)$ time and memory provided the grammar belongs to some subclass that allows the parsing process to be conducted using constant lookahead (e.g., LL(k), LR(k), LALR(k)). See Aho and Ullman (1972) for an overview of parsing.

Lalonde (1977) describes regular right part grammars (RRPGs), a variant of context-free grammars where the right-hand sides of productions can be regular expressions of terminals and non-terminals rather than fixed strings. These describe the same set of languages as context-free grammars. They are easier to specify and understand, but have greater potential for ambiguity. This is because there are more ways to find regular expression matches in a string than there are ways to find constant strings. (We examine regular expression substring matching in detail

in Chapter 4.)

Chapter 3

Incrementally Specifying a Parser

3.1 Evaluation of Existing Models

Recall that pattern matching and one-grammar approaches are the models best suited to recognition. We now examine some of the weaknesses of these models for the task of incrementally specifying a parser. Recall that incremental development involves progressing our understanding of the data one piece at a time, and also writing and debugging the specification one piece at a time. We identify four areas where grammars and pattern matching are deficient for this style of use: interactive efficiency, structure model flexibility, scalability, and batch efficiency.

3.1.1 Interactive Efficiency

The computational work done after each incremental addition or change to a specification should be small enough that the process of incrementally specifying a parser can be done interactively. That is, modifications to the specification should be

separated by short delays. None of the tools or systems we are aware of (those surveyed in Section 2.1) have this characteristic. They all need to completely re-parse the data every time the user changes the specification. This makes it difficult to develop parsers for large data sets interactively.

3.1.2 Structure Model Flexibility

A structure model is a way of restricting how structure elements are used. For example, a grammar requires that all structure elements (non-terminal matches) fit into a nested hierarchy (a parse tree). Pattern matching approaches, however, have no inherent structure model.

We consider a hierarchy to be an overly restrictive structure model for incremental parser specification. One problem is that the most natural model for a given text may require overlapping sub-structures, which are not allowed in a hierarchy. Another is that incremental development may be easier if elements are allowed to occur in an unrestricted way until we determine how to fit them into a more restricted structure model.

3.1.3 Scalability

Large monolithic parser specifications share a problem with large monolithic programs: dependencies between components eventually become too complex to understand. Therefore, as a specification grows, modification eventually become impossible since small changes to one part can wreak havoc on the remainder. A survey by Clark (1991) examines several recognition tools and concludes that all of

them have this problem. As an analogy with software engineering, existing tools are like primitive programming languages with no support for object-oriented, or even structured, program organization.

Others have also pointed out this problem specifically for grammars, noting that their inherent "brittleness" increases the effort needed to construct and maintain them (Murphy & Notkin, 1996). A large part of the reason for this is the fact that dependencies between productions are effectively unrestricted.

A well-known way of decreasing the number of dependencies in a specification is to build it as separate modules and then combine them. One way to do this with a grammar, for example, is to build separate grammars with start symbols $S_1, S_2, \cdots$, then combine them by taking the union of their productions and adding the production $S \rightarrow S_1 \mid S_2 \mid \cdots$. Unfortunately, most commonly used context-free grammar subclasses are not closed under even this simple operation (van den Brand, Sellink, & Verhoef, 1998). More powerful forms of composition are therefore also impossible with grammars.

3.1.4 Batch Efficiency

Whether specified incrementally or not, we want a parser to be efficient in the sense explained in Section 2.2. Even if we specify incrementally and parse the data as part of the incremental process, we want to be able to apply the resulting parser to other data with the same format. This is necessary, for example, if new data is continually being generated.

There is no standard formal model for batch parsing with pattern matching.

Simple forms of pattern matching such as lexical analysis can be performed with finite automata or finite transducers. However, more powerful pattern languages that include structure relationships have not been studied from the perspective of batch parsing.

Efficient parsing with grammars, on the other hand, is a highly studied problem. Not all context-free grammars can be used to parse efficiently, but many context-free subclasses can with constant lookahead (e.g., LL(k), LR(k), LALR(k)). Constant lookahead tends to be appropriate for applications such as programming languages because it is closely related to readability, and because language designers can simply modify the languages to incorporate such constraints. However, arbitrary legacy data is not generally guaranteed to be parseable with constant lookahead.

A concern with grammar parsing is the amount of memory used to maintain a stack. In general, this is $O(n)$ for context-free grammars. One solution is to ignore this problem on the assumption that only pathological cases require large stacks when the grammars are specified with regular right part productions. Another solution is to bound the depth of the recursion as part of the formal model. For example, although the overall parsing model used by SGML is context-free, a constant (TAGLVL) is used to limit the maximum depth of nesting that is allowed in a document (ISO, 1986; Goldfarb, 1990). A more extreme solution is to disallow recursion altogether. This was the strategy used for recognition of the *OED* (Kazman, 1986). In that case, recursive structures were dealt with by assigning distinct types to every level of nested structure.

3.2 Proposed Model

Overall, both grammars and pattern matching approaches suffer from efficiency and scalability problems when used for incremental specification. In addition, pattern matching lacks a formal batch parsing model. Grammars have formal batch parsing models but use an overly restrictive structure model.

The approach that we propose to eliminate these problems is as follows: maintain a dynamic set of regions representing the result of the recognition process, and provide a set of functions for interactively updating this set. We refer to the set of regions as a *region inventory*, and the set of functions as a *region algebra*. Our intention is that a region algebra used for this purpose will have many similarities with the region algebras used as query languages for text databases or information retrieval (Salminen & Tompa, 1992; Burkowski, 1994; Kilpeläinen & Mannila, 1993; Clarke, Cormack, & Burkowski, 1995; Navarro, 1995; Dao, Sacks-Davis, & Thom, 1996).

The functions that comprise a region algebra all operate on subsets of the region inventory. That is, a function call takes a subset of regions from the inventory as arguments, and returns a set of regions as a result. This returned set then becomes part of the region inventory.

Functions can be composed by passing the result of one function call as an argument to another. An *algebra expression* is a composition of two or more function calls. Define the *expression graph* representation of an algebra expression as follows: each function call is represented by a node, and if one call A takes the result of another call B as an argument, then there is an edge from the node for A to the

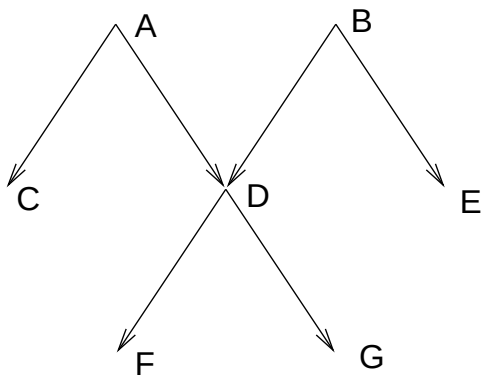


Figure 3.1: An expression graph that is a DAG.

node for B. The *height* of a node in an expression graph is the number of nodes on the longest path from that node to a leaf.

An expression graph must be acyclic. That is, a function call cannot use its own result as an argument, nor can its result be used by any function which contributed to its evaluation. An expression graph does not need to be a tree. It can be multi-rooted, thus representing a set of expressions rather than just one. The directed acyclic graph (DAG) in Figure 3.1, for example, represents two expressions with a common sub-expression rooted at D. A graph can also be disconnected, thus representing a set of expressions without common sub-expressions.

When using a region algebra as an interface, an expression graph plays the role of the parser specification (in the same way that a grammar plays the role of the parser specification when using grammars as an interface). The result of evaluating an expression graph, that is, of parsing with it, is as follows: 1) every node is marked as either producing an intermediate or a final result, and 2) the overall result is the region inventory that is equal to the union of all final results.

3.3 Evaluation of the Proposed Model

3.3.1 Interactive Efficiency

To ensure interactive efficiency, the functions in a region algebra should all be designed so that they can be evaluated efficiently. For example, we can guarantee $O(n)$ time and I/O by breaking the region inventory into subsets of at most n regions that serve as arguments to functions. An example of a useful subset model that allows many operations with $O(n)$ time, $O(n)$ I/O, and $O(1)$ memory is a flat, sorted region list such as those used in PAT (Salminen & Tompa, 1992).

Even with $O(n)$ time and I/O, there is always a size cutoff after which an input string is too large to use the algebra interactively. Where this cutoff is depends on the functions, the implementations, and the density of structure in the data. For data larger than the cutoff, our only choice is to build a parser for a subset of the data interactively, and then parse the remainder in batch mode. Note, however, that $O(n)$ time and I/O is an upper bound that may not be necessary for all functions. Many useful operations will typically work with sets of significantly fewer than n regions. Therefore, in practice, the average cost of an interactive function call may be much lower than n regions of I/O.

A more aggressive goal might be to provide sub-linear interactive efficiency. However, many useful functions require inputs and outputs that can be $O(n)$ in size. For this reason, we consider linear time and I/O to be the most appropriate upper bound to use when choosing operations.

3.3.2 Structure Model

A region inventory may have any structure model we choose. For example, we may require that all regions fit into a hierarchy, or we may allow regions to be completely unrestricted. As pointed out, a strict hierarchy has disadvantages for interactive use. Therefore, an unrestricted model is a better choice for this application[1].

With respect to the survey by Baeza-Yates and Navarro (1996), an unrestricted region inventory is classified as follows as a text structure model:

- It lies somewhere above a hierarchical model but below a full network model. A full network model would allow arbitrary, typed relationships between nodes.

- It uses an explicit non-hierarchical list of regions. Explicit means that the structure is separated from the text, as opposed to some models that use interspersed markup.

- It uses dynamic structure. This contrasts with models that require the structure to be static.

- It is strongly structure bound, which means that the structure is mostly separated from the text and the text is just used to restrict matches in the structure. The alternatives are strongly text-bound models which interleave the structure with the text and translate all queries into text operations, and models that are intermediate between strongly text bound and strongly

[1] Useful models may exist that restrict regions in some way, but not as severely as a hierarchy. However, we do not consider this further here.

structure bound. These intermediate systems allow both text substrings and structure subsets to be retrieved and manipulated as first class objects. We do not need this functionality since we are only performing recognition, not string transformation.

3.3.3 Scalability

With a region algebra, we can incrementally develop a parser specification *non-monotonically* (Aït-Mokhtar and Chanod (1997) introduce this term in the context of parsing). This means that, when we wish to modify a set of regions, we can change the set directly using function calls from the algebra. This contrasts with monotonic development where we have no choice but to go back and change the parts of the specification that initially generated the set. (This is the case with a grammar, for example.)

Non-monotonic development gives us the freedom to refine or revise earlier decisions without going back and changing earlier parts of the specification, or even having to understand them. This means that we also have the freedom to organize the specification with fewer dependencies — by building modular specifications, for example. This is a significant advantage from the point of view of scalability.

Another advantage of a region algebra is the freedom to include functions that combine or compose results in many different ways. For example, it is simple to provide algebra functions that takes sets of regions and perform standard set operations on them (e.g., intersection, union, and difference). In contrast, the closure properties of grammars mean that it is not generally possible to combine

them so as to perform set operations on their parse trees. This is another property of region algebras that gives us more freedom to organize a specification modularly, thus allowing fewer dependencies and more scalability.

3.3.4 Batch Efficiency

A disadvantage of a region algebra approach is that, in general, evaluating an expression graph may be much more expensive than parsing with an efficient grammar subclass. In a typical bottom-up evaluation, for example, the queries are evaluated one-by-one, starting at the leaf nodes and working upward. Heuristic techniques can be used to reduce the cost of evaluating an expression graph (Consens, 1998). However, Jaakkola and Kilpeläinen (1999), claim that, in general, the worst case evaluation cost is $O(n^2)$ time. This is excessive for use as a batch parser.

Others have described specific algebras for which linear time, constant memory, evaluation is possible for any expression graph (Clarke & Cormack, 2000; Ives, Levy, & Weld, 2000). In Chapter 5, we take this further, by developing a general characterization of classes of operations that can always be evaluated efficiently when included together in a region algebra.

Chapter 4

Parsing Models

In the previous chapter, we introduced the idea of using a region algebra as an interface for incrementally developing parsers. We now discuss batch parsing models, defining a *parsing model* to be a *set* of mappings from strings to sets of regions. For example, the set of all mappings that can be specified with grammars is a parsing model. The set of mappings that can be performed using the UNIX tool **lex** can also be considered a parsing model, as can the set of mappings that can be performed using the UNIX tool **grep**.

4.1 Regular Expression Parsers

We start by examining a simple example of a parsing model: given a regular expression and a string, return a list of regions all of which are matches for the regular expression. Note that regions in the list are not distinguished by type, i.e., this problem uses a simplified version of the structure model.

The longest matching results presented in this section are original. We include them here since they are a useful part of the overall parsing model that we describe in the next chapter.

4.1.1 Finding Substrings

It is well known how to simulate an NFA M on a given string x to determine whether x is in the language $L(M)$. See Aho et al. (1974), for example. Finding non-empty *substrings* of a string x that are in $L(M)$ is a more difficult problem. There are $\binom{n+1}{2}$ potential matches in a string of length n, corresponding to starting and ending a match before and after every character.

Practical algorithms restrict potential matches in some way to reduce this number when dealing with large strings. A widely used restriction is found in the POSIX standard (IEEE, 1992) which mandates left-most, longest, non-overlapping matching. This means that when there are two matches with one inside another, the inside one is discarded in favour of the outside; when there are two matches where one overlaps the other, the right one is discarded in favour of the left. Thus the result is a set of matches with no overlapping or nesting. Clarke and Cormack (1997) propose an alternative rule for information retrieval applications: find all shortest, possibly overlapping matches. For this rule, outside matches are discarded in favour of inside ones in cases of nesting, but overlapping matches are kept. **Perl** uses left-most matching and allows the user to specify whether each variable length operator (e.g., $*$ or $+$) should match the longest or shortest possible string (Wall et al., 1996).

All of the above rules guarantee that the number of matches in a result is at most equal to the number of characters in the string. This is ensured by the condition that nested matches are not allowed, a condition that is met by always keeping only a single element from a nested set of matches (for example, either the longest or the shortest). To resolve between *overlapping* non-nested matches, there are two strategies. One is to keep them all (the number of possible matches remains linear in the length of the string). The other is to keep just one, such as the leftmost. This is the most natural choice if the string is scanned left to right, but there is no inherent reason not to choose, say, the rightmost, or to do something more arbitrary like choosing non-deterministically which one to keep.

Finding all shortest, overlapping matches for a regular expression can be done in linear time and constant memory. Clarke and Cormack (1997) give an algorithm for this, extended from one given by Aho et al. (1974). However, finding longest matches is not always possible with a single pass and constant memory.

Consider searching for longest matches in a string by scanning from left to right. When a match completes, there is no way to know whether a longer match will complete later to supersede it. Thus, in the worst case, every match must be buffered indefinitely. For example, consider matching the regular expression $(ab) \mid (a\Sigma^*c)$ against a string of the form $(ab)^n c$ for some n. When scanning from left to right, every ab is a match which has to be stored until the final c is read, at which point they can all be discarded in favour of the single match equal to the entire string.

One solution is to bound the lookahead: if we have a match buffered, and no

longer match completes within a given number of characters, then the buffered match is output. Any matches in progress that may be longer if they complete are discarded at this time. This approach may be appropriate for a tokenizing application, for example, where we know in advance that tokens are never more than some maximum length. The trouble is that it does not, in general, always find the matches that are strictly longest. In the example above, it misses the match equal to the entire string if the string is longer than the chosen limit on how many characters to wait before outputting buffered matches. In sections 4.1.2 and 4.1.3, we describe methods that always correctly find longest matches for a regular expression using only bounded buffering.

4.1.2 Two-Pass Longest Matching

The first method we propose finds longest matches by abandoning the assumption that the search must be done in a single pass over the string. The algorithm uses two passes, the first of which outputs potential matches as they complete (this may require $O(n)$ secondary storage for a string of length n). It then makes a second pass over the potential matches, deleting those superseded by longer matches that completed later during the first pass. The second pass is done in the opposite direction of the first, i.e., if the first pass reads the string from left to right, then the second reads the potential matches from right to left.

The shortest matching algorithm given by Clarke and Cormack (1997) can be modified to perform the first pass of the two-pass algorithm as shown in Algorithm 1. States are designated by numbers in the range 1 to $|Q|$ with 1 representing

Algorithm 1: Perform the first pass of the longest matching algorithm.
Input: A string $x = a_1 a_2 \ldots a_n$, and an NFA $M = (Q, \Sigma, \delta, 1, F)$
Output: A list of regions, sorted by right end positions.

```
MATCH(x, M, S)
(1)    for j ← 1 to |Q|
(2)        P_j ← −1
(3)    for i ← 1 to n
(4)        if P_1 = −1 then P_1 ← i
(5)        for j ← 1 to |Q|
(6)            P'_j ← −1
(7)        for j ← 1 to |Q|
(8)            foreach q ← δ(P_j, a_i)
(9)                if P'_q = −1 OR P_j < P'_q then P'_q ← P_j
(10)       u ← −1
(11)       for j ← 1 to |Q|
(12)           if j ∈ F and P'_j ≠ −1
(13)               if u = −1 OR P'_j < u then u ← P'_j
(14)       if u ≠ −1 then OUTPUT((u, i))
(15)       temp ← P
(16)       P ← P'
(17)       P' ← temp
```

Algorithm 2: Perform the second pass of the longest matching algorithm.

Input: A list $L = \{l_1, l_2, \ldots, l_m\}$ of possibly nested regions sorted by right end positions. Each region l_i is a pair of natural numbers $(l_i.l, l_i.r)$ with $l_i.l \leq l_i.r$.
Output: The input list with any regions nested inside other regions deleted.

```
FILTER(L)
(1)    b ← l_m
(2)    for i ← m − 1 to 1
(3)        if l_i.l < b.l
(4)            OUTPUT(b)
(5)            b ← l_i
(6)    OUTPUT(b)
```

the start state q_0. The idea is to scan the text from left to right (lines 3–17) with a new execution of the NFA beginning at the start state for each character in the text (line 4). For a match in progress that starts at a previous character in the string and brings the NFA to state j, we record the start position in the array P at index j. If no match in progress ends in state j, then the entry P_j is equal to -1. The array P' is used for update purposes, and P and P' are swapped at the end of each pass (lines 15–17). If two intervals of text leave the NFA in the same state, we can immediately discard the shorter one since we are searching for longest matches (as opposed to discarding the longer one in shortest matching). This is performed in lines 9 and 13, and is also reflected in the condition at line 4 which does not begin a new match if there is already a match in progress currently in the start state. Line 10–14 find and outputs the longest match that ends in a final state at the current character, if one exists. Since matches are output immediately when they complete, the list of potential matches is sorted by right end.

Since the output of the first pass is a list of regions sorted by right ends, the second pass can iterate the list from right to left and be sure of always encountering a longer match before any shorter match that is nested inside of it. Thus we can perform the second pass using constant memory as shown in Algorithm 2. This keeps a single region, b, buffered at all times. Line 1 sets b to be equal to the last region in the list. Then the main loop (lines 2–5) visits each remaining region in turn from right to left. The value of b is always a region visited before l_i in the loop. Since the list is sorted by right ends and we visit the regions from right to left, the right end of b is therefore always greater than or equal to the right end of l_i. Therefore, if the condition of line 3 is false (i.e., the left end of l_i is greater than or equal to the left end of b), then we know that l_i is nested inside of b. In this case, the algorithm does nothing with l_i, therefore discarding it. If, on the other hand, the condition of line 3 is true, then l_i is either to the left of b or overlaps it on the left. In either case, b is output and replaced with l_i. After the loop completes, the algorithm outputs the final b.

The first pass uses $O(|P|)$ memory, where $|P|$ is the number of states in the NFA. This is linearly related to the size $|r|$ of the regular expression. Therefore, the pass uses $O(|r|)$ memory. The second pass uses $O(1)$ memory in the form of a single buffer. Therefore, the overall memory used by the two-pass algorithm is $O(|r|)$.

Another way of looking at the first pass algorithm is that it finds the longest match ending at every character in the string. A proof of this can be constructed based on two points: 1) the algorithm directly chooses the longest match when

several end at the same character, and 2) the way that it chooses between matches in progress that converge to the same state never results in a longer match being missed. To understand the second point, suppose we have two matches in progress with start characters a_i and a_j, where $i < j$. Consider what it means if the paths through the NFA corresponding to these two matches in progress converge to the same state s: any path from s to a final state ending at character a_k necessarily represents a match from a_i to a_k and another match from a_j to a_k. That is, there is no way for the shorter match from a_j to a_k to occur without the longer one from a_i to a_k also occurring. Therefore, discarding the match in progress with the a_j start point never results in a missed longer match.

The proof that the second pass correctly deletes all matches nested inside of longer matches is as follows: if, at any time, the l_i currently being visited is nested inside some other region in the input (possibly more than one), then l_i is necessarily nested inside the current b. We prove this by contradiction. Suppose it is possible for the current b to not contain l_i even though there is some region b' that does contain l_i. We know that $b.r$ must be greater than $l_i.r$, since b must be visited before l_i in the right-left iteration order. This implies that $b.l$ must be greater than $l_i.l$, otherwise b would contain l_i. This, in turn, implies that $b.l$ is greater than $b'.l$ since l_i is contained in b'. Given these restrictions, there are two cases of interest: 1) b is nested inside b', or 2) $b.r > b'.r$. In the first case, b' is visited before b, and b is discarded rather than replacing b'. In the second case, b is visited before b', but replaced by b' prior to reading l_i since $b'.l < b.l$. Both of these cases contradict the initial assumption that the current b is not b'. Therefore, if l_i is nested inside one

or more regions, then it is sure to be nested inside the current b, and we always know whether or not to discard an l_i by comparing it to the current b. Also, we know that we never discard an l_i that should not be since we always specifically have the b that contains it at the time when it is discarded.

Note that the second pass of the algorithm visits the regions in the list from right to left, this is also the order in which it outputs them. In some cases, it may be more appropriate for the final output to be sorted from left to right. To arrange this we can simply reverse the direction of both passes. This means that the first pass over the string is right to left, and the second pass over the potential matches is left to right. The necessary modifications to the two algorithms are straightforward.

4.1.3 One Pass Longest Matching

The second approach we propose for finding longest matches is to restrict the regular expressions that are allowed so that only a single pass is required. An example of a regular expression that requires only a single pass is $(0|1)^+$, which matches any non-empty string of 0's and 1's. We can find longest matches for this expression using constant memory: whenever a match completes, we need only check the next character to see if it is a 0 or a 1. If not, then we output the match, otherwise we discard it. In other words, this regular expression requires only a single character of lookahead to find longest matches.

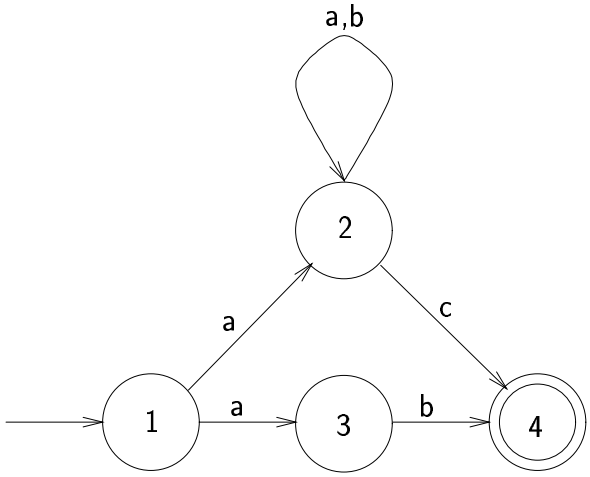


Figure 4.1: An NFA for which unbounded lookahead is required to find longest matches.

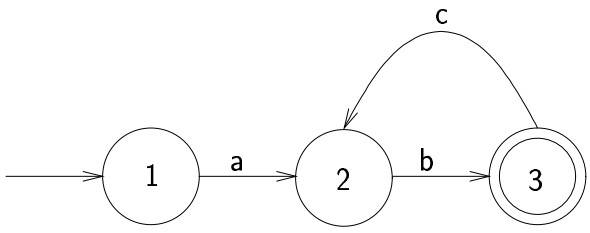


Figure 4.2: An NFA for which we can find longest matches using only two characters of lookahead.

Bounded Lookahead

We use this observation, together with the previously introduced idea that if we have a match buffered, and no longer match completes within a given number of characters, then we output the buffered match. However, rather than choosing a constant bound that may not accurately find longest matches for some regular expressions, we propose to choose the bound based on the regular expression, and further, to disallow regular expressions for which no constant bound is possible.

The amount of lookahead needed to find longest matches for a given regular expression can be determined by examining the corresponding NFA. Suppose that a match completes using Algorithm 1 and that S is the set of states that contain a match in progress. A match completes whenever the automaton enters a final state. Thus, the maximum lookahead required to decide whether to keep the completed match is the length of the longest possible string x such that $\exists s \in S$ for which $\delta(s, x)$ is a final state, and there is no prefix $\overline{x}$ of x such that $\delta(s, \overline{x})$ is a final state. For example, consider the NFA in Figure 4.1. Suppose we match against a string that starts with *ab*. When the first *ab* match is found, there is also a match in progress in state 2 that started on the same character. Because of the loop at state 2, a path starting at state 2 can be arbitrarily long without passing through a final state. Therefore, the lookahead required to decide whether to keep the *ab* match is unbounded. In contrast, consider Figure 4.2. When an *ab* match completes, the only match in progress that may eventually be longer is one that extends the original match by continuing from state 3. This can take at most two transitions before reaching a final state (i.e., returning to state 3). Therefore, the maximum

lookahead required to find longest matches is 2.

We now describe how to compute the maximum lookahead for any regular expression r. Let $M(r)$ be an NFA constructed from r, s be some state in $M(r)$, and prefix$(M(r), s)$ be the NFA formed by taking the NFA $M(r)$, changing the set of final states to $\{s\}$, and deleting all states and transitions from which it is not possible to reach s. For example, Figure 4.3 a) is prefix$(M(r), 2)$ where $M(r)$ is Figure 4.1.

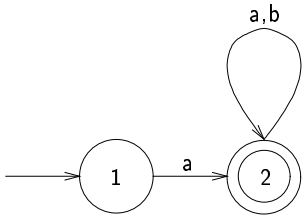
Let $M(r) = (Q, \Sigma, \delta, q_0, F)$ as before, and construct $M(\Sigma^* r) = (Q \cup \{q_0'\}, \Sigma, \delta \cup \{(q_0', \epsilon) \rightarrow q_0, (q_0', \Sigma) \rightarrow q_0'\}, q_0', F)$. For example, Figure 4.3 b) is $M(\Sigma^* r)$ if $M(r)$ is Figure 4.1. State 0 in the figure corresponds to q_0'.

We next construct the NFA for the intersection of $L(M(\Sigma^* r))$ and $L(\text{prefix}(M(r), s))$. A general construction for intersecting two NFAs M_1 and M_2 is as follows [1]: let M_1 be $(Q_1, \Sigma, \delta_1, q_1, F_1)$ and M_2 be $(Q_2, \Sigma, \delta_2, q_2, F_2)$. Then, define the NFA $M_{12} = (Q_1 \times Q_2, \Sigma, \delta, [q_1, q_2], F_1 \times F_2)$ with a transition function δ as follows: $\forall((p_1, a) \rightarrow p_2) \in \delta_1$, $\forall((p_3, a) \rightarrow p_4) \in \delta_2$ the transition $(([p_1, p_3], a) \rightarrow [p_2, p_4])$ is in δ. Also, $\delta([p_1, p_2], \epsilon) = (\delta_1(p_1, \epsilon) \times \{p_2\}) \cup (\{p_1\} \times \delta_2(p_2, \epsilon))$. For example, let Figure 4.3 b) be M_1, and Figure 4.3 a) be M_2. Then M_{12} is shown in Figure 4.3 c). Note that there should be a transitions from states $[1, 1]$ and $[1, 2]$ to a state $[2, 2]$. However, $[2, 2]$ is a dead-end from which it is impossible to reach a final state. It is therefore omitted.

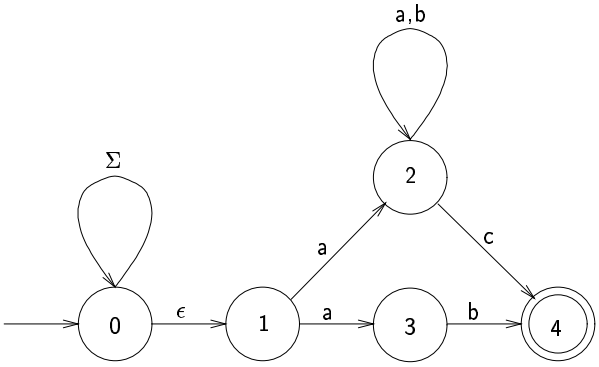
If the intersection of the languages $L(M(\Sigma^* r))$ and $L(\text{prefix}M(r), s)$ is non-empty for some state s in $M(r)$, then it is possible for the situation depicted in

[1] Hopcroft and Ullman (1979) give a construction for intersecting two DFAs. This is a straightforward extension.

a) prefix$(M(r), 2)$



b) $M(\Sigma^* r)$



c) $M(\Sigma^* r) \cap$ prefix$(M(r), 2)$

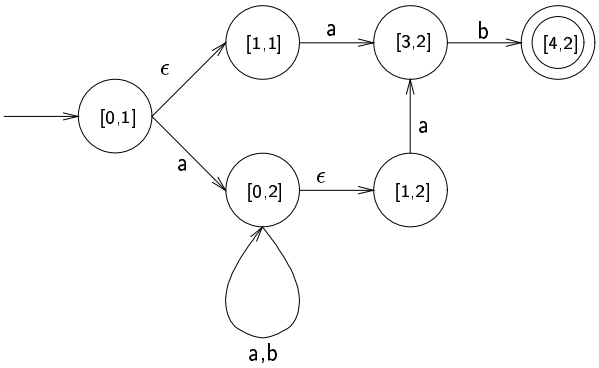


Figure 4.3: NFAs constructed from Figure 4.1.

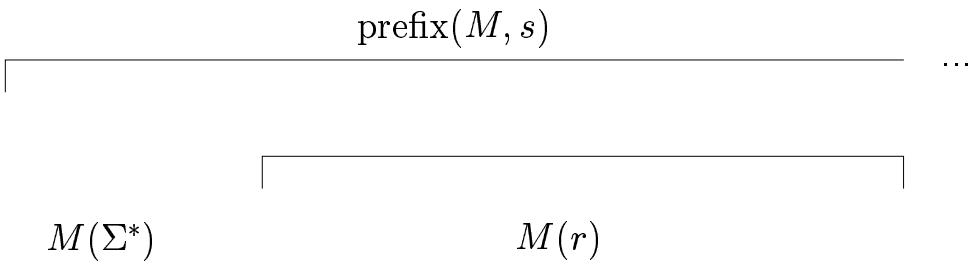


Figure 4.4: A match that completes inside a potentially longer match in progress.

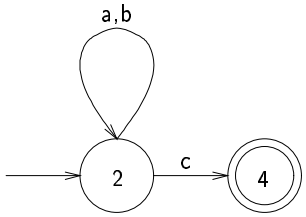


Figure 4.5: suffix$(M(r), 2)$.

Figure 4.4 to occur. This represents a match for the regular expression r at the end of a prefix match that finishes in state s. The lookahead required to decide whether to keep or discard the r match in this case depends on the possible paths from s to a final state.

Define suffix$(M(r), s)$ to be the NFA formed by taking $M(r)$, changing the start state to s, and deleting all states and transitions that cannot be reached starting from s. Figure 4.5 shows suffix$(M(r), 2)$ for our example. The longest path through this NFA from the start state to the final state that does not pass through a final state is the required lookahead (if s is itself a final state, then it can *start* in a final state). We denote the length of the longest such path by longest$(M(r))$ for an NFA

Algorithm 3: Find the maximum lookahead required to search for longest matches for a regular expression using a single pass.

Input: A regular expression r.

Output: The required lookahead.

MATCH(r)

(1) $lookahead \leftarrow 0$
(2) **foreach** state s in the state set of $M(r)$
(3) **if** $(L(M(\Sigma^* r)) \cap L(\text{prefix}(M(r), s))) \neq \phi$
(4) **if** longest(suffix$(M(r), s) > lookahead$
(5) $lookahead \leftarrow$ longest(suffix$(M(r), s)$
(6) **return** $lookahead$

$M(r)$. For example, longest(suffix$(M(r), 2)$) is ∞ because of the loop at state 2.

Algorithm 3 uses the ideas illustrated above to compute the maximum lookahead that may be needed when finding longest matches for a regular expression r. The efficiency is as follows: both $M(\Sigma^* r)$ and prefix$(M(r), s)$ are of size $O(|r|)$ since we can always construct an NFA for a regular expression proportional to the length of the regular expression (Hopcroft & Ullman, 1979). The construction of the NFA for $L(M(\Sigma^* r)) \cap L(\text{prefix}(M(r), s))$ uses the cross product of the two state sets as the state space, and therefore gives a result of size $O(|r|^2)$. If the intersection is non-empty, then suffix$(M(r), s)$ must be computed. This is of size $O(|r|)$, and finding the length of the longest path therefore takes $O(|r|)$ time. Since the algorithm performs one intersection for each state, the overall time complexity is $O(|r|^3)$. The memory used is the size of a single intersection, i.e., $O(|r|^2)$.

Bounded Buffering

Lookahead is the number of characters that must be read after a match completes before knowing whether it is superseded by a longer match. For the purpose of

Figure 4.6: A pattern of potential matches that requires buffering.

finding longest matches, requiring bounded lookahead is more restrictive than necessary. What we really need to bound is the number of buffered matches, i.e., the amount of required memory. This is bounded if the lookahead is bounded, but may also be bounded when the lookahead is not. Consider the expression $a\Sigma^*b$, for example. Matching against a string of the form aba^ib requires i characters of lookahead after the match ending at the second character to find the match ending at the last character. However, only one match is buffered while all these characters are visited.

For buffering to be required, it must be possible for one or more matches to complete inside a longer match as in Figure 4.6. In this case, all of the inside matches are buffered from the time they complete until the time that the outside match completes. Note, however, that matches that are more deeply nested may be discarded. For a regular expression r, an upper bound on the number of regions that may have to be buffered is the maximum k such that $L((\Sigma^*r)^k\Sigma^*) \cap L(r)$ is non-empty. Here, k is the number of matches that can occur directly inside a longer match as in Figure 4.6.

We now show how to calculate the maximum k. Let $M_1 = M(r)$, and $M_2 = M((\Sigma^*r)^+\Sigma^*)$ constructed as shown in Figure 4.7. Next, construct the NFA M_{12} for the intersection of M_1 and M_2 the same way as in the previous section. For

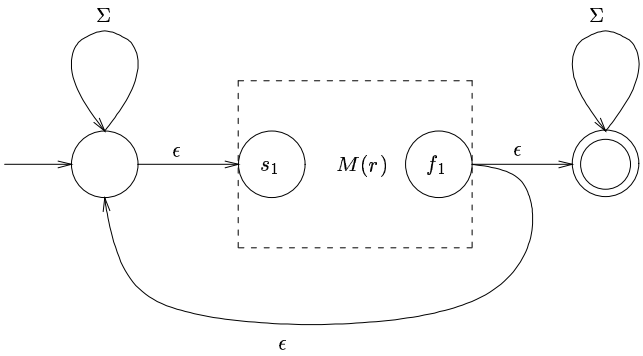
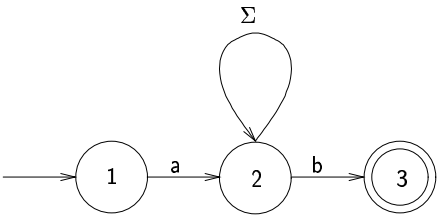


Figure 4.7: The NFA for $M((\Sigma^* r)^+ \Sigma^*)$. The ϵ transition into $M(r)$ goes to q_1. The ϵ transitions out of $M(r)$ come from any final state in F_1.

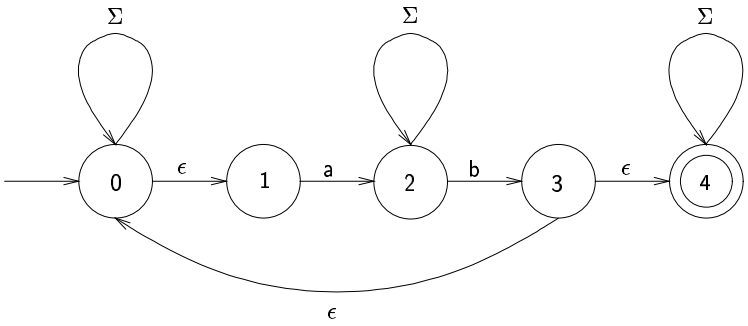
example, for $r = a\Sigma^* b$, M_1 is shown in Figure 4.8 a), M_2 is shown in Figure 4.8 b), and M_{12} is Figure 4.8 c). Note that there is a b transition from $[2, 2]$ to $[3, 2]$ that is not shown because it is not possible to reach a final state from $[3, 2]$.

Consider a path p through M_{12} that starts in a state $[x, q_1]$ for some $x \in Q_1$ (recall that q_1, the second element in the square brackets, is the start state of M_1, but also part of the state space of M_2). Suppose also, that the path ends in a state $[y, q_f]$ for some $y \in Q_1$ and $q_f \in F_1$ (again, q_f is a final state of M_1 but also part of the state space of M_2). An example is the path $[2, 1] \rightarrow [2, 2] \rightarrow [2, 3]$ in Figure 4.8 c). As q_1 and q_f are the second elements in the square brackets, they therefore represent states in M_2. Therefore, a path of this form corresponds to a path through the sub-automaton inside the rectangle in Figure 4.7. The path p therefore represents a match inside a potentially longer match, the basic condition for buffering. If p takes place inside a cycle of M_{12}, then any number of matches can occur inside a longer match. This is true, in Figure 4.8 c) since there is a

a) $M(r)$.



b) $M((\Sigma^* r)^+ \Sigma^*)$



c) $M(r) \cap M(\Sigma^* r)^+ \Sigma^*$

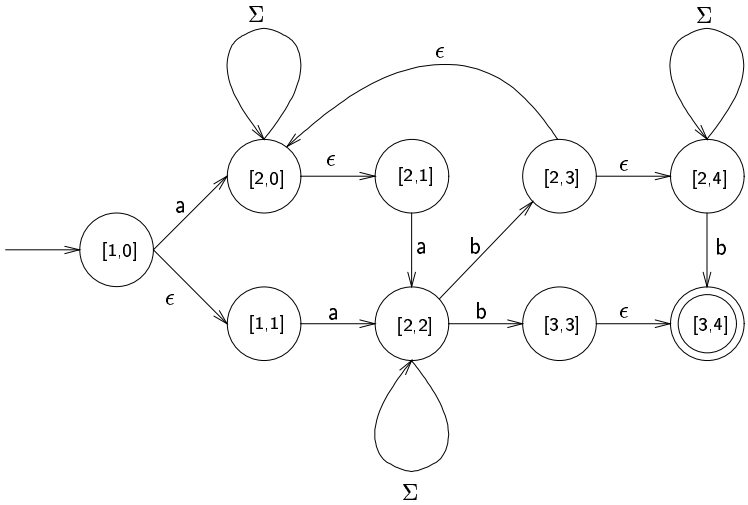


Figure 4.8: NFAs constructed with $r = a\Sigma^* b$.

Figure 4.9: A pattern of potential matches that requires only one buffer.

transition from $[2,3]$ to $[2,0]$ and then to $[2,1]$, the beginning of p. This means that the intersection $L((\Sigma^* r)^k \Sigma^*) \cap L(r)$ is non-empty for any k, i.e., k is ∞. If there is no such p inside a cycle, then the maximum k is the maximum number of matches on a cycle-free path through M_{12}.

The k found above is a weak upper bound on the required buffering since it neglects that some matches in progress are eliminated when they converge to the same state. Consider using the NFA in Figure 4.8 a) to match against a string of the form $a(ab)^i b$. A match begins at the first character, and at each subsequent a. However, all matches in progress converge to state 2, at which point the later starting match is discarded. Therefore, the potential matches are of the form shown in Figure 4.9 rather than Figure 4.6, and it is never necessary to buffer more than one match at a time.

To take discarded matches in progress into account, we need to consider states of the form $[x,x]$ in M_{12}. Since this represents convergence to a single state in $M(r)$, we are interested in the paths that do not contain a state of that form. If all paths containing a match also contain an $[x,x]$ state, as is the case in Figure 4.8

c), then the number of buffers required is 1. Otherwise, the required buffering is calculated from the paths that do not contain an $[x, x]$ state.

Overall then, to find the maximum buffering required to search for longest r matches, we first construct the NFA M_{12}. M_1 and M_2 are $O(|r|)$ in size, so M_{12} is $O(|r|^2)$. Finding the longest path through an NFA can be done in linear time. (The fact that we measure the length as the number of matches rather than the number of transitions does not change this, and neither does having to consider $[x, x]$ states.) Therefore, this procedure uses $O(|r|^2)$ time and $O(|r|^2)$ memory.

Longest Matching Subclasses

Consider an application that requires longest matching in a single pass with bounded memory. One way to provide this is to allow entry of any regular expression, but reject expressions that require unbounded buffering. An alternative approach is to only allow entry of regular expressions restricted in some well defined way so as to always require only bounded buffering. We give some examples of such restrictions.

Define the *finite closure* operator as follows: if p is a regular expression for the language L, then p^k is a regular expression for the language $\bigcup_{i=0}^{k} L^i$.

Lemma 1 *We can find longest matches using a single pass and bounded buffering for any regular expression where all closure operators are finite.*

> **Proof:** If there are only finite closure operators, then the language is finite. If x is the longest string, and y is the shortest string, then $|x|/|y|$ is a weak upper bound on the number of buffers required.

Lemma 2 *We can find longest matches using a single pass and bounded buffering for any regular expression where alternation is always between two languages where a string from one is never a substring of a string from the other.*

For example, the expression $(ab^*c)|(ad^*e)$ requires only bounded buffering because there are no strings in the language for (ab^*c) that are substrings of strings from the language for (ad^*e) and vice versa. On the other hand, in the expression $(a\Sigma^*c)|(ad^*e)$, strings from (ad^*e) can occur inside strings from $(a\Sigma^*c)$, and this expression requires unbounded buffering.

> **Proof:** For the situation in Figure 4.6 to occur, it is necessary for there to be alternation between languages where strings from one are substrings of the other. With no such alternation, this is not possible, and unbounded buffering is not required.

Lemma 3 *We can find longest matches using a single pass and bounded buffering for any regular expression where the only alternations are between strings with a single character*[2].

For example, the expression $(a|b|c)^*d$ only has alternations between single-character strings and therefore only requires bounded buffering.

> **Proof:** This is a direct consequence of Lemma 2.

The above three lemmas characterize restrictions on regular expressions that ensure the ability to find longest matches in a single pass with bounded buffering.

[2] *Equivalently, we can disallow the alternation operator, and add the operator* $[a_1a_2 \ldots a_n]$ *which defines the language* $\{a_1, a_2, \ldots, a_n\}$.

They are only examples, and many other such restrictions are possible. More research is needed to tell whether there is a simple way to characterize the class of *all* regular expressions that require only bounded buffering, or the class that requires only bounded lookahead.

4.2 Rational Function Parsers

4.2.1 Coding

We now discuss using rational functions as parsers. Rational functions map from strings to strings, so the first step is to define a string coding for the output region inventory. Let τ be a blank character and T be a set of types. Let Δ, the output alphabet, be the union of $\{\tau\}$ and the set $\bigcup_{a \in T} \{\acute{a}, \grave{a}\}$, where $\acute{a}$ is a start tag for type a, and $\grave{a}$ is an end tag. The idea is that the function should output τ for each character of the input string, and insert start and end tags at the appropriate locations to delimit regions. Given such an output, we can calculate the position of a tag in the input string by counting the number of preceding τ's. A string with properly paired start and end tags represents a set of regions. Therefore, the set of rational functions that always output strings of this form are parsers according to our expanded definition.

4.2.2 Efficiency

Any rational function can be computed using a two-pass algorithm that computes a length-preserving right-left sequential function composed with a left-right sequential

function (recall Theorem 1 from Section 2.4). The first pass reads the input string from left to right and generates an intermediate output of m characters — the size of the final output (since the second function can always be length-preserving). The second pass reads the intermediate output from right to left and outputs the final result. Therefore, the first pass reads n characters and writes m characters, and the second pass reads m characters and writes m characters. The total I/O is therefore $n+3m$ characters. Assuming an appropriate representation, a sequential transducer can process each character in constant time. Therefore the two-pass algorithm uses $O(n+m)$ time and I/O. The memory used to simulate a finite transducer is linear in the size of the transducer, i.e., $O(\sigma)$. Overall, these bounds are low enough to be acceptable for a batch parser, as discussed in Section 2.2.

4.2.3 Complications

A problem with using rational functions as a parsing model is that finite transducers are a very restricted model of computation. In particular, the need to code the output region inventory as a string leads to unnecessary complications just to handle things like tag pairing and the ambiguity inherent in regular expression matching.

Consider the form of an unambiguous finite transducer that always outputs properly paired tags. A start tag can only be output if the corresponding end tag is output eventually. The only way to guarantee this is to construct the finite transducer so that all paths from the state that outputs the start tag lead to a state that outputs a corresponding end tag. Therefore, if we wish to recognize a pattern for which we cannot know for certain after the first character that a match

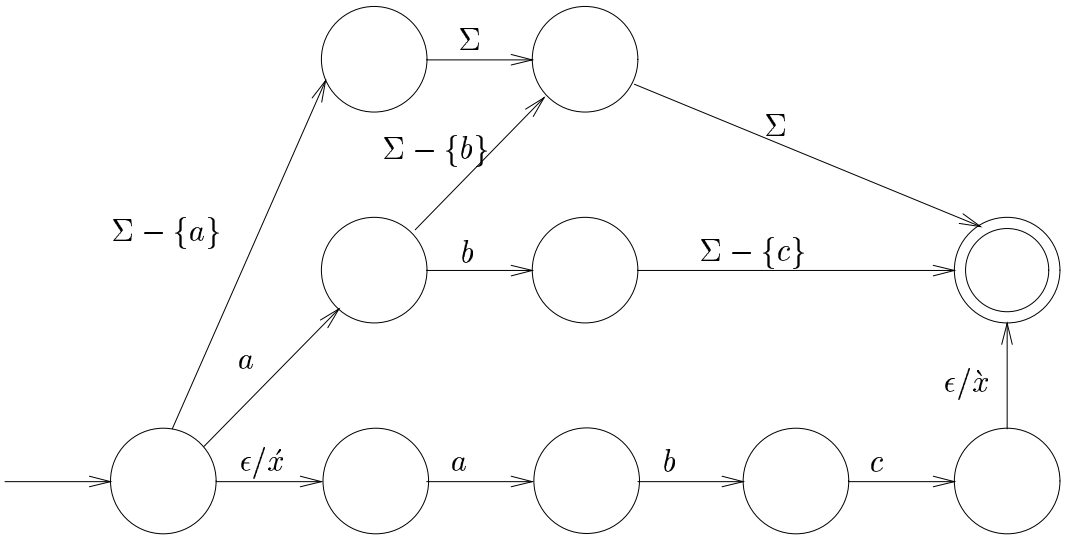


Figure 4.10: A finite transducer that recognizes any three-character string, and outputs surrounding start and end tags of type x only if it matches abc. The notation $\epsilon/\acute{x}$ at a transition means input an ϵ, and output an $\acute{x}$. All other transitions have only the input symbol marked, and the output symbol is implicitly τ.

will complete, then we need to include a non-deterministic choice between taking a path that always completes a match and one that does not. Only the first path should output the start tag. Consider the example finite transducer in Figure 4.10 that accepts any three character string and assigns it the type x if it is abc by outputting an $\acute{x}$ at the first character and an $\grave{x}$ at the last. The bottom path outputs start and end tags, and matches the string abc. The other paths output no tags, and match any string of three characters except abc. The non-deterministic choice between the bottom and middle paths is necessary since there is no way to tell without lookahead whether an initial a will eventually complete a match and therefore whether to output a start tag or not.

If we wish to use two passes to evaluate a rational function specified as a single transducer, then we need some way to decompose the single transducer into left and right sequential transducers automatically. For example, one way to decompose an unambiguous transducer that outputs properly paired start and end tags is to modify it in such a way that it outputs start tags on the left-right pass without worrying whether a match completes. The right-left pass can then delete unpaired start tags for matches that did not complete. The right sequential function to perform this deletion has a state space defined by a finite control that keeps track of a subset of T, the set of types. The transition function adds a type to the finite control on reading its end tag, and removes it on reading the next start tag. When a start tag is encountered and the type is not in the finite control, the tag is deleted from the output stream. All other inputs are simply echoed to the output.

We now give an example of a decomposition into two passes, this time for an

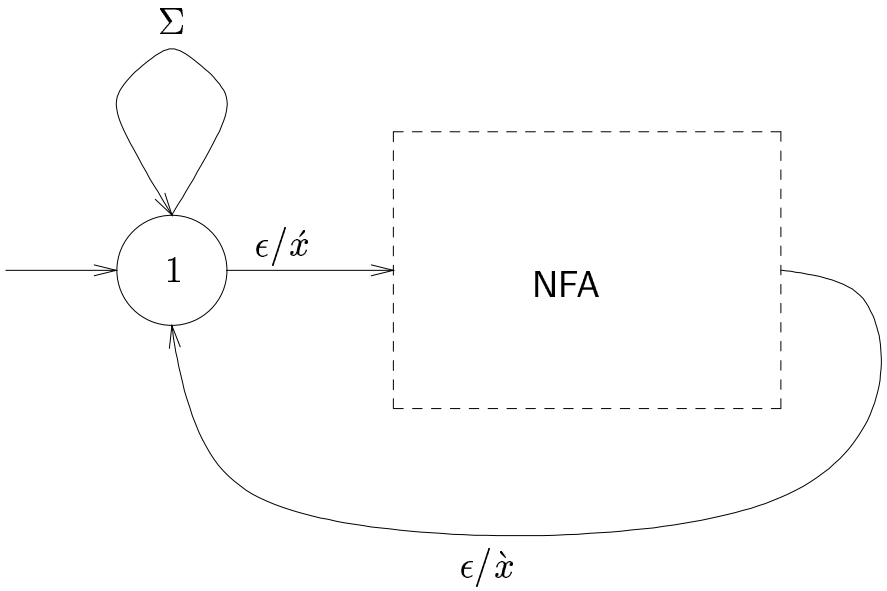


Figure 4.11: A finite transducer M that performs a regular expression matching operation. The dashed box contains the NFA for the regular expression. The transition into the NFA enters its start state. The transition out leaves from the final state (assume that we construct the NFA so that there is only one final state). The finite transducer outputs a τ for every non-ϵ input character.

ambiguous transducer that specifies a rational function only when taken together with a disambiguation rule. The transducer M is shown in Figure 4.11. The task that we wish to perform is to find a flat list of regular expression matches. Note that we give this example to demonstrate only that a finite state model leads to more complications than should really be necessary, even for this relatively simple operation. The construction below is not otherwise an integral part of the thesis.

There are two sources of ambiguity in M. The first is the Σ loop at the start state. This makes the position where a simulator should enter the NFA and output a start tag ambiguous. The second source of ambiguity arises from any nondeterminism within the NFA. This makes it ambiguous where a path can leave the NFA and output an end tag. As discussed in Section 4.1, this ambiguity is an inherent property of regular expression matching. Two possible disambiguation rules are to either keep only leftmost-shortest or leftmost-longest matches. Any such rule that results in a flat list of matches effectively chooses a single path through M.

The decomposition into two passes that we propose uses the same idea as for an unambiguous transducer: output when in doubt on the left-right pass and delete extra tags on the right-left pass. Now, however, we are in doubt about which path to take through M, not just whether a match will eventually complete. We proceed by considering how to deal with this.

Let the output of the first pass be the superimposed outputs of all possible paths through M, including dead end paths that never complete. By superimposed, we mean that all outputs go to a single string, but repeated characters at the same input location are deleted. Therefore, the τ character is output only once for each

input character, rather than being repeated once for each path. Tags output by different paths at the same input location appear in an arbitrary order.

The problem with superimposing the output in the above manner is that we can no longer be sure about how to pair start and end tags. For example, suppose we start a match at position 0, start another at position 1, abort the first match at position 2, and complete the second at position 3. In this case, all we see in the superimposed output is two start tags followed by an end tag. There is no indication that the end tag can only be paired with the second start tag and not the first. We need to include extra information in the output string to interpret it correctly. This can be done by redefining the output alphabet:

$$\Delta = \left(\bigcup_{i=1}^{\sigma} \{\acute{a}_i, \grave{a}_i\} \right) \cup \{\tau\}$$

where σ is the number of states in the NFA. Only start and end tags with the same subscript can be paired. When a path enters the NFA, it is assigned an unused subscript and a start tag with that subscript is output. The path then keeps that subscript until the match either aborts or completes. If it completes, then we output the end tag with the appropriate subscript. The number of subscripts in simultaneous use is bounded by deleting a path whenever two paths converge to the same state. Which path to delete depends on whether we are searching for shortest or longest matches. To make the decision, we need to keep a list that shows the order in which paths corresponding to each subscript began. Then we keep the earlier starting subscript for longest matching, and the later starting one for shortest.

We have now constructed a left sequential transducer that is completely determined from M. The first component of the state space for the sequential transducer is the reachable subset of the power set of Q, the state space of M. In addition, each state in Q is paired with a subscript when it is in the process of matching the NFA, and a list of matches in progress is stored to indicate the order in which the matches began. A state in the left-right transducer therefore consists of a subset of Q, a list of subscripts associated with some of the states in this subset, and a list of subscripts that indicates the order in which the matches in progress corresponding to those subscripts began.

We now define the right sequential transducer that is composed with the left sequential transducer to implement the rational function for M. On reading an end tag, the right sequential transducer stores the subscript in its finite control and deletes all other start and end tags until it finds the paired start tag with the proper subscript. At that point it removes the start tag from the finite control. Any start tag encountered when there is no end tag with the same subscript in the finite control is deleted. All other input is echoed from the input to the output.

There is a final detail when we are searching for shortest matches. Consider the case where we have an output $\acute{x}_1, \acute{x}_2, \grave{x}_2, \grave{x}_1$. Reading right to left on the second pass, we encounter subscript 1 first and have no way of knowing that there is shorter match and that $\grave{x}_1$ should be deleted. This is something that we do know on the left-right pass as soon as $\grave{x}_2$ is output. Therefore, whenever a shortest match completes in the left-right pass, we must delete any other matches in progress that may eventually output longer matches. This is a slight change to the construction

of the left sequential transducer.

The general strategy exemplified by the above construction can be applied to any finite transducer that defines a rational function directly, or does so together with extra disambiguation rules that also define a rational function. The first step is to superimpose the output of all paths through the finite transducer, adding any necessary information to distinguish between paths (e.g., the subscripts in the above example). The second step is to implement the disambiguation rules to delete all outputs except for those of the single correct path. Any lookahead required by the disambiguation rules or the original finite transducer is implemented in the right sequential function, any lookback is implemented as an additional part of the left sequential function (e.g., the rule to delete matches in progress for a shortest matching type whenever a match completes).

4.3 General, Multiple-Pass Parsing Models

We can view a two-pass algorithm for calculating a rational function parser as one example of a more general parsing model that uses multiple passes, possibly more than two. The longest matching algorithm for regular expressions is another example of such a parsing model that also uses two passes.

In the general parsing model, the first pass inputs the string and outputs a region inventory, and the second pass inputs the region inventory and outputs a modified region inventory. We can also generalize to more than two passes by having every subsequent pass input the region inventory output by the previous pass, and output a new region inventory.

For both example parsing models, the first pass is left-right and the second is right-left. Extending to more than two passes we could make the third left-right, the fourth right-left, and so on. That is, the direction alternates every pass. Alternatively, we could begin with a right-left pass and then alternate.

One advantage of a parsing model that uses multiple passes in alternating directions is that it allows computation of functions that require lookahead by using individual passes that do not require lookahead. This is true with rational functions, for example, since the sequential transducers used to compute left and right sequential functions are deterministic.

A general, multiple-pass parsing model need not be limited to a finite state computation model like a sequential transducer. If required, more powerful operations can be allowed, such as building a hash table of identifiers, or maintaining a queue or stack of symbols. The only restriction we assume on the computation is that it satisfy the efficiency bounds given in Section 2.2. Note that with multiple-pass models, reading and writing the intermediate outputs is part of the I/O cost.

Chapter 5

Algebra Design

This chapter shows how to design region algebras for which we can efficiently evaluate all possible expression graphs. Recall from Section 2.2 that efficiency in the context of batch parsing means at most $O(\sigma n + \sigma m)$ time, $O(n + m + \sigma)$ I/O, and $O(\sigma)$ memory. We propose an evaluation method that satisfies these bounds. We then characterize classes of functions that an algebra can include if all possible expression graphs are to be evaluated with this method.

5.1 Interactive Efficiency

Recall that we want interactive efficiency in addition to batch efficiency. Thus we restrict every function so that it accesses $O(n)$ regions. We do this by breaking the region inventory into subsets of size at most n that are used as arguments to functions. The subset model that we initially assume is a flat (non-overlapping, non-nesting), sorted region list similar to that used in PAT (Salminen & Tompa,

1992). All regions in a flat list have the same type, and can therefore be stored as left and right indexes only. A flat list contains at most n regions for a string of length n (this occurs only if there is one region for each character in the string). Zero length regions are not allowed.

One reason for choosing a flat list representation is that it allows simple bounds on memory usage for the evaluation method described later in this chapter. Another reason is that functions operating on flat lists are easy to define and understand. In particular, we can define many functions that correspond exactly to those in typical structured query languages.

A general disadvantage of a flat list representation is that it is more difficult to ensure, when the lists are dynamic, that all regions fit into an overall structure model such as a tree. This is not a limitation here since we are assuming that region inventories have no restrictions.

5.2 String-to-Region Functions

We start by considering functions that generate flat lists of regions from the input string. These are the only functions that do not take lists as arguments. Therefore, string-to-region function calls can never depend on each other in an expression graph. All the leaf nodes of an expression graph should be string-to-region functions if the graph is to specify a parser according to our extended definition.

5.2.1 Substring Matching

Matching functions find regions in the string that match a given pattern. The simplest form of pattern matching is finding substrings. For example, we can define the function MATCH(String s) to perform this operation. It finds substrings exactly matching s and returns them in a list. Note that the overall input string for the parsing process is an implied argument to MATCH(). To ensure a flat result, some matches must be deleted in cases where there is overlap. The natural way to do this is to select leftmost matches if the search is performed with a left-right pass, or rightmost matches if the search is performed with a right-left pass.

For interactive use, we can evaluate a function like MATCH() either by scanning the string, or by using an index. This could be an inverted list or a PAT array, for example, depending on what kinds of strings we wish to find. For the batch parser, there is no advantage to using an index since substring scans are independent and can be performed simultaneously during the scan that would be needed to build the index.

5.2.2 Regular Expression Matching

Now consider matching functions that use regular expressions as the pattern language. For example, we can define MATCH-SHORTEST(Regexp re) which finds shortest matches for the regular expression re, and MATCH-LONGEST(Regexp re) which finds longest matches. Recall our discussion of shortest and longest regular expression matching in Section 4.1.

Shortest matching can be useful for finding structures that have known end-

points, but content that is not well understood. For example, the expression $abc.^*def$ finds regions delimited by *closest* pairs of the substrings abc and def. This would be more complicated to do with longest matching since we would have to use a more specific description of the characters between the two endpoints to avoid absorbing abc or def as part of the content.

Longest matching is needed to perform the type of tokenizing done with programming languages. For example, the most natural way to find number tokens represented by strings of digits is to find longest matches for the regular expression Σ_d^+ where Σ_d represents the set of digits. Longest matching can also be useful for finding strings that represent large structures to be broken up later, rather than just short, bottom-level tokens. So, for example, we could search for phrases of words consisting only of alphanumeric characters and spaces with the regular expression $(\Sigma_w \mid \Sigma_s)^+$ where Σ_w represents the set of alphanumeric characters and Σ_s represents the set of whitespace characters.

We have the same choices for interactive and batch implementation as with substring matching. Interactively, we can use scanning as described in Section 4.1, or an index (see, for example, Baeza-Yates (1989), Baeza-Yates and Gonnet (1996), and Manber and Wu (1993).) In batch mode we should use scanning since there are no dependencies and all matching can be performed simultaneously.

5.3 Region-to-Region Functions

We now consider functions that take lists of regions as arguments and return lists of regions as results. These can be composed to give expression graphs that have

edges, unlike the case where only string-to-region functions are used.

Recall that the output region inventory for an expression graph is the merge of all the final output lists. Similarly the input region inventory is the merge of all the input lists. For an expression graph that represents a parser, all input lists are outputs of string-to-region calls. However, the following discussion begins by considering expression graphs made up exclusively of region-to-region calls. This means that the arguments to the leaf nodes in the expression graph are external inputs.

Imagine that the input and output region inventories are one merged, sorted region inventory. Regions can be sorted primarily by left end or by right end (left-sorted or right-sorted). We do not consider the secondary sort order for reasons explained later. Suppose there are M regions in total. Then consider the following loop that iterates the regions in the sort order:

for i **from** 1 **to** M
    process region i

Processing a region involves outputting it if it is an element of the output region inventory, or buffering it if it is an input region that will be needed later. A region must be output immediately if it is to be output at all. Buffered regions can only be used to make decisions about outputting other regions. Therefore, the sort order of the output is the same as for the input.

Suppose the loop buffers no more than $O(\sigma)$ regions at a time where σ is the

number of function calls in the expression. Then it uses $O(\sigma)$ memory. Assume that all input regions are generated while reading the string from secondary storage, and that all output regions are written to secondary storage. Then the I/O is $O(n+m)$. Finally, assume that M, the total number of regions in both the input and output region inventories, is $O(\sigma n + \sigma m)$, and that each iteration of the loop uses $O(1)$ time. Then the total time used by the loop is $O(\sigma n + \sigma m)$. Overall then, an evaluation model based on this loop satisfies our efficiency requirements for a batch parsing model.

Note that it is not necessary to store an input region inventory directly as a single list of typed regions to use such a loop. Rather, we can store separate flat lists and logically merge them while performing the loop. In the same way, it is possible to split the output into separate lists while executing the loop. One or both of these strategies might be appropriate depending on the form in which the input is available, and what form of output is needed. In particular, separate flat lists are the appropriate representation for using a region algebra interactively.

Depending on the input sort order of the loop, we refer to it as either a *left-right deterministic pass with left-sorted input*, or a *left-right deterministic pass with right-sorted input*. Here we are using the term *deterministic* as an analogy with deterministic finite transducers: in the same way that a deterministic finite transducer never needs to look ahead to a later character to choose which transition to take, a deterministic pass never needs to look ahead to a later region to decide whether to output the current region. Because of this restriction, such a pass cannot be used to evaluate arbitrary expressions. In the following sections, we explore

the properties of expressions that it can be used to evaluate.

5.3.1 Structure Selection Queries

We define a *structure selection query* $Q(S, D)$ to be a function with two list arguments S and D that selects a subset of the regions from S according to a dependency on the regions in D. Structure selection queries are expressed in the form $\{s \in S \mid C\}$, where C is a boolean expression that consists of one or more clauses joined by the logical operators $\wedge$ and $\vee$, using brackets to specify precedence. Each clause is of the form $(x \odot y)$ where $x \in \{s.l, s.r\}$, $\odot \in \{=, <, >, \neq, \leq, \geq\}$, and $y \in \{d.l, d.r\}$, or more generally y is an arithmetic expression involving $d.l$ or $d.r$. The overall boolean expression C is qualified by either $\exists d \in D$ or $\nexists d \in D$.

An example of a structure query is CONTAINED-IN(S,D) which returns $\{s \in S \mid \exists d \in D \quad (s.l \geq d.l) \wedge (s.r \leq d.r)\}$. This selects regions in S that are inside a region from D. Similarly, NOT-CONTAINING(S,D) returns $\{s \in S \mid \nexists d \in D \quad (s.l \leq d.l) \wedge (s.r \geq d.r)\}$, which selects regions in S that do not contain a region from D. An example of a function that uses a clause with an arithmetic expression involving $d.l$ is STARTS-SOON-AFTER(S,D), which returns $\{s \in S \mid \exists d \in D \quad (s.l > d.r) \wedge (s.l < d.r + 100)\}$. This selects s regions that have a left end less than 100 characters after the right end of a d.

Structure queries are well defined for arbitrary sets of regions, not just for flat lists. Therefore, another example of a query is the selection performed by the second pass of the longest matching algorithm. Let the query NOT-CONTAINED-IN() be $\{s \in S \mid \nexists d \in D \quad ((s.l > d.l) \wedge (s.r \leq d.r)) \vee ((s.l \geq d.l) \wedge (s.r < d.r))\}$.

This finds all d regions that are not contained inside of an s region, but does not treat a d region equal to an s region as being contained in it. Suppose we call this function with *both* arguments equal to the set of potential matches generated by the first pass of the longest matching algorithm. The effect is to select the longest matches, i.e., those that are not contained in any others.

From this point on, we use the convention that, unless otherwise specified, the arguments to a query are always named S and D. Furthermore, we assume that s is a region in S and d is a region in D. This allows us to specify query definitions unambiguously by writing just the constraint C. We also assume the presence of $\exists d$ if no existential qualifier is specified. For example, CONTAINED-IN() is written $(s.l \geq d.l) \wedge (s.r \leq d.r)$. When we talk about more than one query at a time, or specific calls to queries, we introduce a subscript, and refer to separate queries as Q_1, Q_2 etc. In this case, the arguments are $S_1, S_2, \ldots$, and $D_1, D_2, \ldots$, and regions from the arguments are $s_1, s_2, \ldots$ and $d_1, d_2, \ldots$.

We assume that any function included in a region algebra depends on all its arguments. For structure selection queries, this means that C must not be a tautology that always selects s, nor a contradiction that always fails to select it. We can efficiently recognize such a badly defined structure selection query:

Lemma 4 *Given a structure selection query defined by a boolean expression C with m clauses, we can determine whether C is a contradiction or tautology in $O(m^2)$ time.*

> **Proof:** All clauses refer to points on the same domain (character positions in the string), and compare either $s.l$ or $s.r$ to some expression

involving $d.l$ or $d.r$. To be a valid query, there must be at least one assignment of $s.l$ and $s.r$ making the expression true, and at least one making it false. If there are m different clauses, then there are at most $2m+1$ relevant assignment classes of $s.l$ or $s.r$ to test. These are the m points that $s.l$ and $s.r$ are compared to, the $m-1$ regions between them, the region from the start of the string to the first point, and the region from the last point to the end of the string. Thus there are exactly $\binom{2m+1}{2} + 2m + 1$ assignments of interest. If all of these are true or false then the expression is a tautology or contradiction.

The composition of a query Q_1 with another query Q_2 is found by passing the result of one call as an argument to the other. So, for example, the statement $Q_2(S_2, Q_1(S_1, D_1))$ means perform query Q_1 on S_1 and D_1, and then perform Q_2 on S_2 and the result.

Recall that the set returned by a query Q_2 is specified in the form $\{s_2 \in S_2 \mid C_2\}$ where C_2 is a boolean expression involving s_2 and d_2 with an existential qualifier on d_2. When we compose two functions Q_2 and Q_1 by passing the result of Q_1 as an argument to Q_2, the returned set is $\{s_2 \in S_2 \mid C_2 \wedge C_1\}$ with all instances of s_2 and S_2 replaced by s_1 and S_1 if the result of Q_1 is passed as the first argument to Q_2, or all instances of d_2 and D_2 replaced by s_1 and S_1 if it is passed as the second argument. For example, the set returned by the composition NOT-CONTAINING(S_2, CONTAINED-IN(S_1, D_1)) is

$$\{s_2 \in S_2 \mid (\nexists d_2 \in D_2 \quad (s_2.l \leq d_2.l) \wedge (s_2.r \geq d_2.r)) \quad \wedge$$

$$(\exists d_1 \in D_1 \quad (s_1.l \geq d_1.l) \wedge (s_1.r \leq d_1.r))\}$$

where D_2 is the result of the CONTAINED-IN() call and therefore a subset of S_1. Thus we replace instances of d_2 and D_2 by s_1 and S_1 to give

$$\{s_2 \in S_2 \mid (\nexists s_1 \in S_1 \quad (s_2.l \leq s_1.l) \wedge (s_2.r \geq s_1.r)) \quad \wedge$$

$$(\exists d_1 \in D_1 \quad (s_1.l \geq d_1.l) \wedge (s_1.r \leq d_1.r))\}$$

For a query Q_2 where the first argument is the result of a query Q_1 and the second is the result of a query Q_0, the set specification is of the form $\{s_2 \in S_2 \mid C_2 \wedge C_1 \wedge C_0\}$ with instances of s_2 and S_2 replaced by s_1 and S_1, and instances of d_2 and D_2 replaced by s_0 and S_0. Applying these rules recursively, we can write out the set specification for the result of any query in an expression.

We define the *dependency expression* for a set specification to be the boolean expression component with existential qualifiers removed. For a query Q, we denote this $E(Q)$. For example, E(CONTAINED-IN) is $(s.l \geq d.l) \wedge (s.r \leq d.r)$. We denote the dependency expression for a query Q in an expression graph G, $E_G(Q)$. For example, E_G(NOT-CONTAINING) for the graph defined by NOT-CONTAINING(S_2,CONTAINED-IN(S_1, D_1)) is

$$((s_2.l \leq s_1.l) \wedge (s_2.r \geq s_1.r)) \wedge ((s_1.l \geq d_1.l) \wedge (s_1.r \leq d_1.r))$$

A dependency expression can be seen as a general description of the possible positions of a d region that may affect whether a given s region satisfies a query.

For example, letting $\Rightarrow$ denote logical implication, we can see that $E(\text{CONTAINED-IN}) \Rightarrow (d.l \leq s.r)$ since $s.l \leq s.r$ for any region. In other words, any d match relevant to the decision whether to keep a given s match must start before the s ends.

Consider an expression constructed by composing only CONTAINED-IN() calls. We assert that this can always be evaluated using a left-right deterministic pass with left-sorted input. A CONTAINED-IN call never requires lookahead in this case since if a region s is contained inside another region d, then d is guaranteed to be read before s in left-sorted order. The amount of buffering required for each function call is a single region since, with flat lists, a d that contains a given s, if it exists, is guaranteed to the most recent d read. Therefore, exactly σ regions need be buffered for an expression consisting of σ CONTAINED-IN() calls.

Now consider the function CONTAINING(S,D) defined as $(s.l \leq d.l) \wedge (s.r \geq d.r)$. This selects all regions in S that contain a region from D. Any expression composed exclusively of CONTAINING() calls can be evaluated using a left-right deterministic pass with right-sorted input, as we will show below. However, arbitrary expressions formed by composing CONTAINING() and CONTAINED-IN() calls can not always be evaluated with a left-right deterministic pass.

Consider the regions shown in Figure 5.1. Suppose a must be contained in b, b must contain c, c must be contained in d and d must contain e. Suppose we iterate the list in left-sorted order: d, b, a, c, e. When we visit e we know it should be kept since it has no dependencies. However, we do not know whether to keep d without looking ahead to the point where we visit e. By extension, we do not know whether

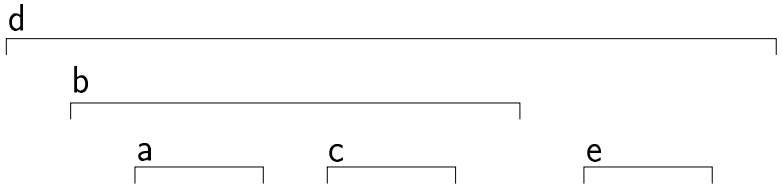


Figure 5.1: A list of regions related by both CONTAINED-IN() and CONTAINING().

to keep b, a, and c since they indirectly depend on e. Now consider a right-sorted order: a, c, b, e, d. In this case, we read e before d and therefore know to keep both of these regions. However, we read c before d and therefore do not know whether to keep it or a and b since they depend on c.

In general, the problem is that arbitrary compositions of CONTAINED-IN() and CONTAINING() can require lookahead regardless of the sort order of the pass. We need to characterize the conditions under which an expression does not require lookahead.

We start by considering individual queries that are not part of expressions. An individual query requires no lookahead if the decision to keep or discard a region s is always based on a region d that is read before s in the loop.

Lemma 5 *A single query Q requires no lookahead with left-sorted input iff $E(Q) \Rightarrow (d.l < s.l)$.*

> **Proof:** If $E(Q) \Rightarrow (d.l < s.l)$ then the decision whether to keep a given s always depends on a d with an earlier left end. If the input is left-sorted, then this d is read before s. Therefore, no lookahead is needed.

> If no lookahead is required, then the d needed to decide whether to keep a given s is read before that s. This means that $E(Q)$ must relate s to a d that is guaranteed to have an earlier left end. In other words, $E(Q) \Rightarrow (d.l < s.l)$.

For example, $E(Q)$ for the query $(s.l > d.l) \wedge (s.r < d.r)$ implies that $(d.l < s.l)$. However, $E(Q)$ for the query $(s.r = d.l)$ makes no such implication. Therefore, we can evaluate the first query using a left-right deterministic pass with left-sorted input, but not the second.

If regions have equal left ends in a left-sorted input, then either we must make no assumptions about their relative ordering, or we have to consider a secondary sort order to decide whether d comes before s in the pass. We avoid this issue by assuming that regions with equal left-ends are read and buffered before any processing, meaning we can treat them as if they are read simultaneously. When using this strategy, the lemma is:

Lemma 6 *A single query Q requires no lookahead with left-sorted input iff $E(Q) \Rightarrow (d.l \leq s.l)$.*

The corresponding lemma for right-sorted input is as follows. Again, we assume that equal regions according to the sort order are read and buffered before any processing. The proof is similar to that for the last lemma.

Lemma 7 *A single query Q requires no lookahead with right-sorted input iff $E(Q) \Rightarrow (d.r \leq s.r)$.*

We now consider multiple queries. We can evaluate all queries in an expression graph G using a left-right deterministic pass iff for every query Q in G, the

dependency expression $E_G(Q)$ does not imply lookahead. With left-sorted input, for example, $E_G(Q)$ must imply that $(x.l \leq s.l)$ for every region x referenced in $E_G(Q)$. In other words, all regions needed to decide whether a given s satisfies a query must be visited before s. Note that this assumes that every x list referenced in $E_G(Q)$ actually does affect the result of the query, which follows directly from our assumption that functions depend on all of their arguments.

We now show that $E_G(Q)$ implies no lookahead for every query in G exactly when $E(Q)$ implies no lookahead for every query in G:

Theorem 2 *We can evaluate an expression graph G made up of structure selection queries using a left-right, deterministic pass with left-sorted input iff for every query Q in G, $E(Q) \Rightarrow (d.l \leq s.l)$.*

> **Proof:** First we prove the necessity of the condition. If $\exists Q \in G$ such that $E(Q) \not\Rightarrow (d.l \leq s.l)$, then by Lemma 6, the query requires lookahead to calculate.
>
> Next we prove the sufficiency of the condition through induction on h, the height of a query in G. For the base case of $h = 1$ (any single query Q), the dependency expression $E(Q)$ implies $(d.l \leq s.l)$ by the requirement of the theorem and therefore requires no lookahead. The induction hypothesis is that any query of height at most $h - 1$ has a dependency expression $E_G(Q)$ that implies $(x.l \leq s.l)$ for every x in $E_G(Q)$.
>
> For the induction step, we consider a query Q_0 of height h. The induction hypothesis applies to all descendants of Q_0 since they have heights

less than or equal to $h-1$. Now, build the dependency expression $E_G(Q_0)$ in the usual way starting with $E(Q_0) \wedge E_G(Q_l) \wedge E_G(Q_r)$ where Q_l is the left child of Q_0, and Q_r is the right child of Q_0, and replacing instances of s_0 in $E(Q_0)$ by s_l, and instances of d_0 by s_r. Next we perform the induction step. We know that $E_G(Q_l) \Rightarrow (x.l \leq s_l.l)$ for all x in $E_G(Q_l)$ by the induction hypothesis. Also, we know that $E_G(Q_r) \Rightarrow (y.l \leq s_r.l)$ for all y in $E_G(Q_r)$ by the induction hypothesis. Finally, by the condition of the theorem, we know that $E(Q_0) \Rightarrow (d_0.l \leq s_0.l)$ which, after the substitution, is $E(Q_0) \Rightarrow (s_r.l \leq s_l.l)$. Thus $E_G(Q_0) \Rightarrow (x.l \leq s_l.l)$ for every x in $E_G(Q_0)$, from which it follows that the result of Q_0 can be calculated with no lookahead.

Theorem 3 *We can evaluate an expression graph G made up of structure selection queries using a left-right, deterministic pass with right-sorted input iff* $\forall Q \in G, E(Q) \Rightarrow (d.r \leq s.r)$.

Proof: The proof is of the same form as that of the previous theorem.

Consider an application of the theorems. The queries $(s.l \geq d.l) \wedge (s.r \leq d.r)$ and $(d.r = s.l)$ both imply $(d.l \leq s.l)$, which is the condition of Theorem 2. The theorem therefore states that, using one deterministic pass with left-sorted input and left-sorted output, we can evaluate *any* expression graph formed by composing calls to these two queries, regardless of how large.

We use the conditions of the theorems as the basis for a categorization of query functions. Henceforth, we refer to functions that satisfy $E(Q) \Rightarrow (d.l \leq s.l)$ as

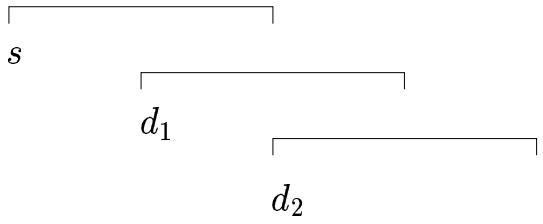


Figure 5.2: Three overlapped regions.

condition 1 queries for a left-right pass, and those that satisfy $E(Q) \Rightarrow (d.r \leq s.r)$ as *condition 2 queries for a left-right pass*. Note that a function can be in both categories. For example, the function $(d.r = s.l)$ is both condition 1 and condition 2.

Now consider a slight variation of the left-right deterministic pass. Suppose the input is left-sorted, but instead of outputting every region immediately in its left-sorted order, we buffer regions temporarily so as to give a right-sorted output. For an individual query Q to require no lookahead in this case, it is necessary that all regions on which s depends be read before s is *output*. That is, unlike the normal left-right deterministic pass, some of the regions s depends on may be visited after s is visited. However, in this case, they must be visited during the time that s is buffered. Since s is read in its left-sorted position, and buffered until its right-sorted position, evaluating a query Q in an expression graph G requires that $E(Q)$ implies $(d.l \leq s.r)$. We call a query that satisfies this implication a *condition 3 query for a left-right pass*. To evaluate a condition 3 query Q as part of an expression graph G, $E_G(Q)$ must imply $(x.l \leq s.r)$ for all x regions in $E_G(Q)$.

It is *not* generally true that we can evaluate all queries in an expression made

up of condition 3 queries using a deterministic pass with left-sorted input and right-sorted output. Consider the three regions in Figure 5.2. Suppose that s depends on d_1 and d_1 depends on d_2. It is true that $(d_1.l \leq s.r)$ and that $(d_2.l \leq d_1.r)$, but it is not true that $(d_2.l \leq s.r)$. Therefore, we cannot decide whether to keep s without using lookahead in a left-right pass with left-sorted input and right-sorted output. The problem is that the condition 3 implication is not transitive. Therefore, not all expressions composed of condition 3 queries can be evaluated using a deterministic pass.

We can now state the conditions under which we know for sure that *all* expressions constructible using queries from a given region algebra are evaluable with a left-right deterministic pass with properly sorted inputs and outputs:

Theorem 4 *For a set $\mathcal{Q}$ of query functions, we can evaluate every expression formed with queries from this set using a left-right, deterministic pass with properly sorted input and output iff all functions are condition 1 or all functions are condition 2.*

> **Proof:** Sufficiency of the condition is established by the preceding theorems. That is, we know that we can evaluate all expressions made up entirely of condition 1 or condition 2 functions using a left-right, deterministic pass.
>
> To show the necessity of the condition, we prove that, given any set of functions that are not all condition 1 and not all condition 2 it is always possible to construct an expression that requires lookahead to compute. This is true for a pass with left-sorted input and left-sorted

Figure 5.3: A situation that requires lookahead.

output by Theorem 2, and true for a pass with right-sorted input and right-sorted output by Theorem 3. It is not possible for a pass to use right-sorted input and left-sorted output since this may require that regions be output before they are read. The only remaining option is left-sorted input and right-sorted output. For all queries in a given expression graph G to require no lookahead with this option, $E_G(Q)$ must imply $(x.l \leq s.r)$ for every Q in G.

Suppose we have a query Q_1 such that $E(Q_1) \not\Rightarrow (d.l \leq s.l)$, and another query Q_2 such that $E(Q_2) \not\Rightarrow (d.r \leq s.r)$. That is, Q_1 is not condition 1 (although it may be condition 2), and Q_2 is not condition 2 (although it may be condition 1). Consider the composition $Q_2(S_2, (Q_1(S_1, D_1))$. For a region $s_1 \in S_1$, the decision whether s_1 satisfies Q_1 may require knowledge of a region $d_1 \in D_1$ with $d_1.l > s_1.l$ since $E(Q_1) \not\Rightarrow (d.l \leq s.l)$. Similarly, the decision whether a region $s_2 \in S_2$ satisfies Q_2 may require knowledge of a region $d_2 \in D_2$ with $d_2.r > s_2.r$ since $E(Q_2) \not\Rightarrow (d.r \leq s.r)$. Because D_2 is the result of Q_1, which is a subset of S_1, this means that determining whether s_2 satisfies Q_2 can depend on a region $s_1 \in S_1$ where $s_1.r > s_2.r$. Thus we can have the situation shown

in Figure 5.3. The decision whether s_2 satisfies Q_2 is based on an s_1 where $(s_1.r > s_2.r)$, and the decision whether s_1 satisfies Q_1 is based on a d_1 where $(d_1.l > s_1.l)$. Since these conditions are consistent with $(d_1.l > s_2.r)$, we cannot calculate Q_2 without lookahead.

Now consider a right-left deterministic pass which visits regions in descending left-sorted or right-sorted order:

for i **from** M **to** 1

process region i

All of the preceding results can be adapted to this case:

Corollary 1 *For a set $\mathcal{Q}$ of query functions, we can evaluate every expression formed with queries from this set using a* right-left*, deterministic pass with properly sorted input and output iff all functions $Q \in \mathcal{Q}$ satisfy $E(Q) \Rightarrow (d.r \geq s.r)$ or all functions satisfy $E(Q) \Rightarrow (d.l \geq s.l)$.*

Proof: The proof is symmetrically the same as Theorem 4, with all occurrences of left and right swapped.

We refer to functions satisfying $E(Q) \Rightarrow (d.r \geq s.r)$ as *condition 1 queries for a right-left pass*, and those satisfying $E(Q) \Rightarrow (d.l \geq s.l)$ as *condition 2 queries for a right-left pass.* The classification of a function is not necessarily the same for passes in both directions. For example, $(d.l < s.l) \wedge (d.r > s.l) \wedge (d.r < s.r)$ is both condition 1 and condition 2 for a left-right pass, but neither condition 1 nor condition 2 for a right-left pass.

In general, any structure selection query can be written as a set specification and then classified according to the implied order of s and d. For example, CONTAINED-IN() is condition 1 in either direction and therefore, according to the above results, we can evaluate any expression formed using CONTAINED-IN() queries with either a left-right deterministic pass or a right-left deterministic pass. Similarly, CONTAINING() is condition 2 in either direction and we can evaluate arbitrary expressions with a left-right or right-left deterministic pass. However, we cannot evaluate arbitrary expressions formed by composing CONTAINED-IN() and CONTAINING() with a deterministic pass since this mixes condition 1 and condition 2 queries as in the example of Figure 5.1.

Another example of a query is INTERSECT(S,D), which performs set intersection, returning all regions from S that are also in D. The set specification is $(s.l = d.l) \wedge (s.r = d.r)$, which is both condition 1 and condition 2 in either direction. Another set query is SUBTRACT(S,D) which returns all regions from S that are not in D. The set specification is $\nexists d(s.l = d.l) \wedge (s.r = d.r)$, which is also both condition 1 and condition 2 in either direction.

As a final example, consider AFTER(), which has the set specification $(s.l = d.r)$. This finds all regions in S that follow and abut a region in D. It is both condition 1 and condition 2 for a left-right pass, but neither for a right-left pass. The opposite function BEFORE() with the set specification $(s.r = d.l)$ is condition 1 and condition 2 for a right-left pass, but neither for a left-right pass.

Overall, if we have an algebra containing functions that are all of the same classification, then we can evaluate *any* expression formed with that algebra using

a deterministic pass. For example, CONTAINING(), INTERSECT(), SUBTRACT(), and AFTER() are all condition 1 in a left-right direction. Therefore, we can define an algebra consisting of just these functions, and evaluate any expression using a left-right deterministic pass with left-sorted input. Another example is the algebra consisting of CONTAINED-IN(), INTERSECT(), SUBTRACT(), and BEFORE() which are all condition 2 in a right-left direction. We can evaluate any expression in this algebra using a right-left deterministic pass with left-sorted input.

The memory used to evaluate an expression with a deterministic pass is equal to the sum of the memory used to evaluate each query in the expression. All the queries we have introduced as examples use constant memory, so the memory used to evaluate expressions made up of such queries is linear in the size of the expression.

For general region inventories, the assumption that equal regions according to the sort order are buffered before processing may use arbitrary additional memory. However, for a region inventory that is a merge of k flat lists, we have an upper bound of k regions that can share an endpoint. Since the number of lists in the input is linearly bounded by the number of queries in the expression, the overall memory usage therefore remains linear in the size of the expression.

5.3.2 Region Generation

Region generation functions take lists of regions and output new regions. This contrasts with structure query functions which simply return subsets of their arguments. One paradigm for region generation is to find pairs of regions satisfying a query, and calculate a new region from each pair. We define queries in the same

way as before, except that both s and d are returned to make a pair (s, d).

An example of a region generation function is MERGE-OVERLAPS(S,D). The query is $\{(s, d) \mid (s.l < d.l) \wedge (s.r > d.l) \wedge (s.r < d.r)\}$. This finds (s, d) pairs where d overlaps s on the right. For each pair, MERGE-OVERLAPS() generates a region running from the start of the s to the end of the d.

In general, the result of a query may be ambiguous in the sense that an s can pair with more than one d, or a d with more than one s. This requires a disambiguation rule. For example, the query $\{(s, d) \mid (s.l > d.r)\}$ can pair each s with any preceding d. One way to disambiguate this is to specify that it pairs with the closest d. This can be specified as follows: $\{(s, d) \mid (s.l > d.r) \wedge (\nexists s_1(s_1.l > d.l) \wedge (s_1.l < s.l))$.

The limitations on region generation functions are closely related to those on structure query functions. Consider using a function that queries for (s, d) pairs and generates g regions. We can express the relationship between g and s, and between g and d as *dependency queries*. For example, MERGE-OVERLAPS() has the dependency queries $\{g \mid (g.l = s.l) \wedge (g.r > s.r)\}$ and $\{g \mid (g.l < d.l) \wedge (g.r = d.r)\}$. These dependencies can be classified as condition 1 or condition 2 in the usual way. For example, both of the above are condition 2 for a left-right pass, meaning that generation of g regions is always based on regions that have already been read during a left-right deterministic pass with right-sorted input.

We refer to a region generation function as condition 1 or condition 2 if both of the dependency queries are condition 1 or condition 2. From the results of Section 5.3.1 it directly follows that all functions in an expression constructed from structure queries and region generation functions can be calculated with a deter-

ministic pass iff every function is condition 1, or every function is condition 2.

We can consider string-to-region functions to be region generation functions. Suppose that we view the string input to a string-to-region function as a list of regions, one for each character. Then MATCH(), for example, is a region generation function that finds a tuple of adjacent characters equal to a match, and generates a region running from the first character to the last character. Since the generated region contains all of the characters in the tuple, MATCH() is a condition 2 region generation function. This type of classification can be done for any function that generates regions based on characters in the string.

5.3.3 General Functions

We can now generalize beyond structure query and region generation functions. Suppose we have a function where the output of a region s always depends on some set D of other regions. Then we classify the function as follows:

- Condition 1 for a left-right pass if $\forall d \in D, (d.l \leq s.l)$.
- Condition 2 for a left-right pass if $\forall d \in D, (d.r \leq s.r)$.
- Condition 1 for a right-left pass if $\forall d \in D, (d.r \geq s.r)$.
- Condition 2 for a right-left pass if $\forall d \in D, (d.l \geq s.l)$.

An example of a useful operation that is not a region generation or a structure query is finding the set union of two flat lists. The set union of two flat lists is not, in general, a flat list itself. Therefore, any function we define has to return some subset of the entire set union. For example, we can include all regions from A in the

result, and all regions in B that do not nest or overlap a region in A. This subset of B can be found with a structure query: DISJOINT(B,A) $= \{b \in B \mid \nexists a \in A \quad (b.l < a.r) \wedge (b.r > a.l)$. This approach has the nice property that regions from A always have priority in case of conflicts. Unfortunately, the query is neither condition 1 nor condition 2 in either direction. A more symmetric alternative is to query both lists, always giving the first region encountered priority in case of conflict. In this case, the decision whether to output any a or b depends on whether there is any touching region of the other type with a smaller left end for a left-right pass, or a greater right end for a right-left pass. This is condition 1 but not condition 2.

There are also some useful variations of containment structure queries. For example, CONTAINING-N(S,D,n) finds s regions that contain at least n d regions. Also, NTH-CONTAINED-IN(S,D,n) finds the nth s contained in a d. These are not, strictly speaking, structure selection queries according to our previous definition since they take an additional argument. However, they are still clearly condition 2 and condition 1 functions, respectively.

We can also generalize beyond functions that operate on flat lists. Requiring that all lists be flat is a simple way to ensure interactive efficiency, but there are other models such as overlapped lists, or even full tree models, that can be made efficient with appropriate design. The ability to evaluate any expression using a single deterministic pass can still be characterized in terms of the dependencies of the component functions. For example, a structure query that is possible with trees but not with flat lists is DIRECTLY-CONTAINING(). This returns an s instance that contains a d instance only if there is no instance of another type that is contained

in the s and contains the d. This is only possible if regions other than those in S and D are available to the function, which is not the case with flat lists. However, the function is still condition 2, since any d that an s depends on has a less than or equal right end, and so does any other region that may contain the d and be contained in the s. Similarly, we can define a function DIRECTLY-CONTAINED-IN() that is condition 1 and works for trees but not for flat lists.

5.3.4 Constant Lookahead

Consider a variation of a deterministic left-right pass: a *constant lookahead* left-right pass has access to the next k regions in the input region inventory at the time when it make a decision whether to output the current region. If we compose two queries, each of which requires a pass with a constant lookahead of k regions, then the overall lookahead required may be as much as $2k$. In general, an expression graph with depth d where every function requires lookahead k will require lookahead dk to evaluate. Thus, we can evaluate an expression composed of constant lookahead functions using a constant lookahead pass, but the constant depends on the size of the expression.

In practice, there are not many useful functions that can be defined to require constant lookahead in the region inventory. Therefore, we do not consider constant lookahead passes further. Deterministic passes are more useful for our purposes. Also, constant lookahead passes are not strictly necessary since arbitrary lookahead can be achieved by using multiple passes in different directions as discussed in the next section.

5.4 Completely Composable Algebras

When using an algebra to construct a parser interactively, the simplest situation for a user is if every possible expression graph can be evaluated efficiently. In this case, the user does not have to worry about rules or limitations on how function calls can be composed, and we say that the algebra is *completely composable*.

Suppose that we want an algebra that is completely composable when using a single-pass evaluation method, either left-right or right-left. According to the previous results, we must include only condition 1 functions or only condition 2 functions in the algebra. Also, we must include at least one string-to-region function to occupy the leaves of the expression graphs. All the useful string-to-region functions of which we are aware are condition 2. This means that all functions in the algebra must also be condition 2. An algebra of condition 2 functions is therefore our only option if we want a single pass evaluation method.

A second option for a completely composable algebra is to evaluate string-to-region function calls with a first pass (either left-right or right-left) and all region-to-region calls with a second pass. In this case, the functions that we evaluate with the second pass can all be condition 1, or all condition 2. If we choose condition 1, then we can include a general longest matching function since the second pass of the general longest matching algorithm for regular expressions is a condition 1 structure query (NOT-CONTAINED-IN()). Note that if we do include general longest matching, then the second pass must be in the opposite direction to the first. Otherwise, this is not necessarily a requirement.

The output sort order of a pass must be the same as the input sort order of

the next pass in a multiple-pass evaluation model. Therefore, for example, if we output right-sorted regions (as all left-right string-to-region functions do) then the next pass must use right-sorted input. However, we can easily convert a region inventory from right-sorted to left-sorted input by buffering regions when they are read and processing them only when we reach their left-sorted positions. If a region inventory is made up of k flat lists, then this strategy buffers at most k regions at a time. Therefore, we have four options for the second pass: it can be left-right left-sorted, left-right right-sorted, right-left left-sorted, or right-left right-sorted. However, there is no benefit in using the same direction and sort order for both the first and second pass. If we do this, then the second pass cannot evaluate anything more than the first. Therefore, assuming the first pass uses string-to-region functions, there are a total of six useful two-pass combinations:

1. First pass left-right with right-sorted output; second right-left with right-sorted input.

2. First pass left-right with right-sorted output; second right-left with left-sorted input.

3. First pass left-right with right-sorted output; second left-right with left-sorted input.

4. First pass right-left with left-sorted output; second left-right with left-sorted input.

5. First pass right-left with left-sorted output; second left-right with right-sorted input.

6. First pass right-left with left-sorted output; second right-left with right-sorted input.

Remember that the sort order of a pass determines whether it can evaluate condition 1 or condition 2 functions. Also, recall that condition 1 and condition 2 mean different things depending on the direction of the pass. Therefore, every one of the above two-pass combinations corresponds to a different class of completely composable region algebras.

5.5 Stratified Algebras

The disadvantage of completely composable algebras is that the allowed functions are limited. For example, we cannot include both condition 1 and condition 2 structure queries. We now consider possible ways of loosening this requirement.

One possibility is to calculate dynamically how many passes are needed to find the result of any given function call in an expression, and disallow an expression if it requires too many. For example, an expression containing condition 3 function calls can be calculated in a single pass in some situations but requires more in others. Assuming it is possible to calculate the required number of passes efficiently, this approach could be used to decide dynamically how many passes are needed by a given call. The trouble is that not knowing whether functions can be composed in certain ways until after trying them is confusing to the user. Therefore, some way of pre-calculating this information and representing it is needed.

We propose an approach based on breaking the algebra into a small, ordered list of *strata*. A function call from one stratum can only be passed the result of a

function call from the same stratum or an earlier one. For example, consider a two strata model. Then every path through a legal expression graph that starts at an arbitrary node and continues to a leaf must pass through zero or more calls from the first stratum followed by zero or more calls from the second.

If we can evaluate any expression formed from the functions in a single stratum using one deterministic pass, then we can partition the expression graph by removing all edges that start at a function from one stratum and end at a function from another. Thereafter, we can evaluate each of these partitions with a single pass since each is formed using a completely composable algebra. The pass corresponding to the first stratum inputs the string, and all subsequent passes input the output of the previous pass. In other words, we evaluate all calls to functions from the first stratum in the first pass, all calls to functions in the second stratum in the second pass, and so on.

Having two strata allows us to mix condition 1 and condition 2 functions in a limited way. For example, condition 2 queries and region generation functions can be included in the first stratum along with functions like MATCH-SHORTEST() (which is a condition 2 region generator). Then, the second stratum can include condition 1 query functions. Generally, we can have more than two passes and a separate stratum for each one, alternating condition 1 and condition 2 functions for each successive class. We give an example of a useful two-strata algebra in the next chapter.

5.6 Restricted Matching

Some single function calls require more than one pass to compute. For example, MATCH-LONGEST() requires two passes to find general longest matches. One way to view such a function when part of a stratified algebra is as a composition of two functions in adjacent strata. For example, in a two-stratum algebra, we can view MATCH-LONGEST() as a composition of NOT-CONTAINED-IN() and a new function MATCH-POTENTIAL-LONGEST() which finds the output of the first pass of the longest matching algorithm. In a stratified algebra, for example, MATCH-POTENTIAL-LONGEST() could be part of the first stratum and NOT-CONTAINED-IN() could be part of the second stratum. We now consider a class of functions that generalize the two-pass longest matching algorithm by allowing arbitrary structure queries between matches for different expressions.

Consider searching for matches for a regular expression r and requiring these matches to satisfy a structure query. The simplest way to do this is to perform a regular expression search followed by a query on the result. We are interested in something slightly different, however: having the query affect the search itself. For example, suppose we define a function MATCH-SHORTEST-INSIDE(r, A) that finds shortest r matches that are inside a region in A. Calling MATCH-SHORTEST() followed by CONTAINED-IN() does not do the same thing since it may miss some matches that MATCH-SHORTEST() discards as overlaps. If MATCH-SHORTEST() keeps leftmost matches, for example, then it misses a match inside an a if it has another match overlapping it on the left that is not inside an a.

Now suppose that we wish to search for A matches using a regular expression

at the same time as we search for matches for r. This is the type of problem we examine in this section: simultaneously searching for multiple regular expressions that depend on each other according to queries. It is simple to search for multiple expressions simultaneously when they do not depend on each other. However, as we will see below, there are possibilities for efficiency problems with simultaneous restricted searches.

The general approach we propose is to find potential matches using a first pass, then filter them to satisfy any queries with a second pass. This gives us more freedom than performing independent matches with a first pass and querying with a second since a list of potential matches need not be flat. The only restriction on the list of potential matches is that its size be at most linear in the length of the string.

Consider MATCH-SHORTEST-INSIDE(). We can always guarantee a linear number of potential outputs with this function, regardless of the other matches that they depend on. In the worst case, we need only output *all* shortest matches, including any overlaps. This is linear even though it is not flat.

Now consider a function MATCH-LONGEST-INSIDE() that searches for *longest* matches inside other matches. Suppose we simultaneously search for unrestricted b matches using an NFA B, and longest a matches inside b matches using an NFA A. Using the normal matching algorithm, the number of b matches in progress, and therefore the memory used, is bounded by the number of states in B. However, in this case, we cannot bound the number of a matches in progress by the number of states in A. Figure 5.4 illustrates the situation where we have two b matches

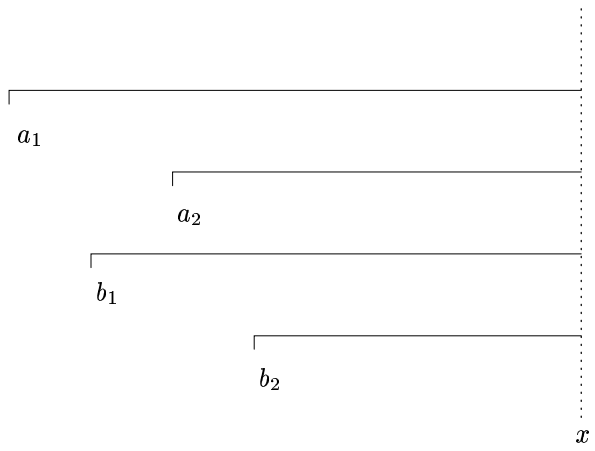


Figure 5.4: We want to discard one a, keeping the longest that is inside a b. It is impossible to know which one to discard, since we do not know if the longer b will complete.

in progress and two a matches in progress, each of which starts inside a different b match in progress. Suppose that the two a matches in progress converge to the same state in the A NFA at the point x. For unrestricted matching, we would be able to choose the longer one. However, in this case, we do not know whether one or both of b_1 and b_2 will complete. If we keep only a_1 and b_2 completes but b_1 does not, then we have missed the longest match inside of b_2. In general, the number of a matches in progress that we need to keep can be as high as $|A|$ times $|B|$. Now suppose we wish to simultaneously find the potential matches for a chain of k MATCH-LONGEST-INSIDE() calls with NFAs of size $s_1, s_2, \ldots, s_k$. Then uses $O(s_1 s_2 \cdots s_k)$ memory.

So why does shortest matching inside of other matches work, but not longest matching? There is an important relationship here between shortest matching

and the CONTAINED-IN() structure query. When searching for shortest matches inside other matches, the decision to always choose the shortest match when paths converge or matches complete at the same location is never wrong. This is easy to prove: if a region a is inside a region b, then any shorter region a_0 inside of a is also inside b. This property is not true for a longer match a_1 that contains a: a_1 is not necessarily inside b just because a is. So searching for shortest matches inside other matches is efficiently possible but searching for longest matches is not.

Queries also exist for which we can efficiently search for longest but not for shortest matches. Suppose we require a match to *contain* a match of another type (i.e., the CONTAINING() query). If a region a contains a region b, then any longer match a_1 that contains a also contains b. However, a shorter match a_0 inside a does not necessarily contain b. Therefore, the function MATCH-LONGEST-CONTAINING() can be efficient but MATCH-SHORTEST-CONTAINING() cannot.

To bound the memory used for any kind of matching, we need to be able to bound the number of matches in progress that must be stored at any one time. To bound the number of matches in progress, we must be able to choose which ones to keep and which to discard when we have too many. We call a way of making this choice a *linearization rule*.

Note that longest and shortest matching are more than just linearization rules. They also tell us how to choose between potential matches so that our final result is a non-nesting list. A linearization rule by itself only says how to choose between matches in progress. From now on, when we refer to shortest or longest matching as linearization rules, we mean only the linearization rule component.

When we enforce a bound on the number of matches in progress, we automatically bound the number of potential matches. That is, if we have at most c matches in progress at once, then no more than c complete at every input character. Thus the maximum number of potential matches is cn. With automaton matching, this immediately guarantees an output of at most n potential matches. Since any NFA can be written with a single final state, matches in progress must always converge to that single state before they complete. Therefore, at most one completed match is output for each input character.

In general, choosing permissible linearization rules for restricted matching needs to take into account both the matches in progress among which the choice is to be made, and information about other matches. As shown above, this is possible for some combinations of restrictions and linearization rules, but not for others. For example, we can perform shortest matching inside other matches, but not longest matching.

Let an *anchor* be a position in the string referred to by the right hand side of a clause in a query. For example, suppose we have two d regions $(1, 2)$ and $(5, 6)$, and a query $(s.l \leq d.l) \wedge (s.r \geq d.r + 5)$. Then the anchors for the first clause are the two $d.l$ points: 1 and 5. The anchors for the second clause are 7 and 11, the two points matching $d.r + 5$.

The general problem we are interested in is this: we wish to scan for matches that depend on anchors through a query, but when we apply a linearization rule to choose between matches in progress, we may only know potential locations of anchors, any of which can disappear. We say that a query and a linearization rule

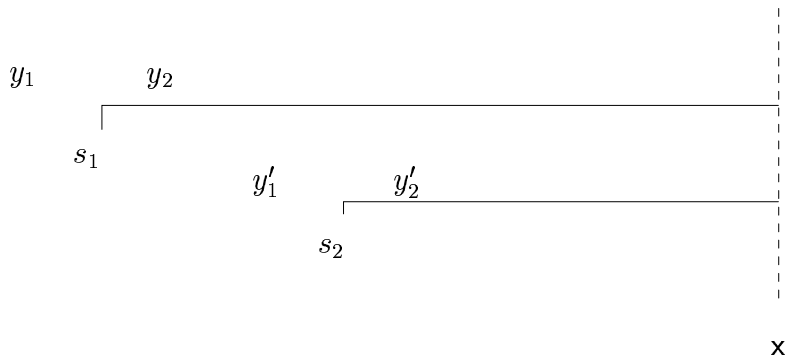


Figure 5.5: We want to choose between s_1 and s_2.

are *compatible* if the linearization rule never discards a match in progress that might turn out to be the correct choice if any or all of the potential anchors disappear.

We now present results regarding compatible queries and linearization rules:

Theorem 5 *Consider a search that depends on a set of anchors Y according to a query Q. If $\exists y_1, y_2 \in Y$ such that $E(Q) \Rightarrow (y_1 \leq s.l \leq y_2) \wedge E(Q) \not\Rightarrow (s.r \leq y_2)$, then* no *linearization rule is compatible with Q.*[1]

Proof: Suppose we need to choose between two matches in progress s_1 and s_2 as depicted in Figure 5.5. Let y_1, y_2 be two potential anchors such that $y_1 \leq s_1 \leq y_2$ is necessary for s_1 to satisfy the query. Similarly, s_2 satisfies the query only if $y'_1 \leq s_2 \leq y'_2$. Suppose s_1 and s_2 converge to the same state at position x in the string. (This can happen before s_1 and s_2 complete since $E(Q) \not\Rightarrow (s.r \leq y_2)$.) Given that any potential anchor may disappear, we have no way of choosing between s_1 and s_2

[1] *Note that $(s.l = y)$ is equivalent to $(y \leq s.l \leq y)$.*

that is guaranteed to be correct. Therefore, no linearization rule is compatible with Q.

Theorem 6 *Consider a search that depends on anchors in Y according to a query Q for which the condition of Theorem 5 does not apply (i.e., $\forall y_1, y_2 \in Y, E(Q) \not\Rightarrow (y_1 \leq s.l \leq y_2) \vee E(Q) \Rightarrow (s.r \leq y_2)$).*

1. *If $\exists y \in Y$ such that $E(Q) \Rightarrow (s.l \geq y)$ then shortest matching is the only linearization rule compatible with Q.*

2. *If $\exists y \in Y$ such that $E(Q) \Rightarrow (s.l \leq y) \wedge E(Q) \not\Rightarrow (s.r \leq y)$, then longest matching is the only linearization rule compatible with Q.*

Proof: For case 1, $s.l$ must be somewhere to the right of an anchor y. Suppose we have a set of potential y points, and matches in progress beginning at s_1 and s_2 where s_2 is to the right of s_1. Then no matter what subset of the y points disappears, if s_1 satisfies Q then this implies that s_2 does also. However s_2 satisfying Q does not imply that s_1 does. In other words, the only possible cases are for neither s_1 nor s_2 to satisfy Q, for both to satisfy Q, or for just s_2 to satisfy Q. Therefore, the only choice that is guaranteed not to discard a correct match in progress is s_2, which is the rightmost and therefore the shortest.

If both s_1 and s_2 are to the right of all potential y points, then it does not matter which we choose. However, shortest matching is the only linearization rule that consistently chooses the same way every time, regardless of y points.

Case 2 is similar to case 1. If $s.l$ can be anywhere to the left of a y point, then the only way to never discard a correct match in progress is to always choose the leftmost. Again, it does not matter which we choose if s_1 and s_2 are both to the left of all potential y points, but we require a consistent linearization rule.

Corollary 2 *If none of the conditions of Theorems 5 and 6 apply, then* any *linearization rule is compatible with* Q.

Proof: Neither of the previous theorems apply if $\forall y \in Y, (E(Q) \not\Rightarrow (s.l \leq y) \wedge E(Q) \not\Rightarrow (s.l \geq y)) \vee E(Q) \Rightarrow (s.r \leq y)$. In other words, either there is no restriction on $s.l$, or $s.l$ is never restricted except by anchors that occur after the match completes. In either case, it does not matter how we choose between matches in progress. Therefore, we can use any linearization rule.

Consider some examples of compatible queries and linearization rules. The query $(s.l \geq d.l) \wedge (s.r \leq d.r)$ is used by MATCH-SHORTEST-INSIDE(). According to the theorem, this is only compatible with shortest matching, which agrees with our previous conclusion. The query $(s.l \leq d.l) \wedge (s.r \geq d.r)$ is used by MATCH-LONGEST-CONTAINING() and is only compatible with longest matching according to the theorem, again in agreement with our previous conclusion. The query $(s.r = d.l)$ has no $a.l$ clause and is therefore compatible with any linearization rule. The query $(s.l = d.r)$ is not compatible with any linearization rule, and neither is $(s.l > d.l) \wedge (s.l < d.r)$. As an example of a distance query, $(s.l > (d.l + 10)) \wedge (s.r \leq d.r)$

requires that an s match start at least ten characters after the start of a d match; this is only compatible with shortest matching.

Corollary 3 *The rules for determining compatible linearization rules for a structural restriction are valid even if the potential anchors are completely unknown when we apply a linearization rule.*

The proofs of the theorems are equally valid if anchors can arbitrarily appear as well as disappear. Therefore, for compatible linearization rules and queries, we can find a linear sized list of potential s matches before we know anything about the points to which the restrictions refer. We simply find and output all potential matches as if performing unrestricted matching, and then filter them later when we know the anchors.

Corollary 4 *The theorems apply to right-left scanning as well if we replace every instance of $s.l$ by $s.r$, $s.r$ by $s.l$, $<$ by $>$, and $>$ by $<$.*

This is true by the symmetry of the proof and means that, just as queries may be of different types (i.e., condition 1 or condition 2) for left-right and right-left passes, they may also be compatible with different linearization rules. For example, the query $(s.l = d.l)$ is compatible with any linearization rule with left-right scanning but is not compatible with any with right-left scanning.

Overall, the results in this section characterize the conditions under which we can efficiently search for several regular expressions simultaneously when they depend on each other according to structure queries. Efficiency in this case means that there is at most one match in progress at a time for each state of each NFA,

and at most n potential matches are output for each regular expression. Recall that this is only an issue if we wish to have the queries affect the searches themselves. It is trivial to simultaneously search for several regular expressions simultaneously if the searches are performed independently.

Recall that we intend for a second pass to be used to choose between the potential matches output by the first pass. Compatibility between a linearization rule and a query does not mean that we can completely evaluate the query during the scan, just that we do not discard any regions in error. Evaluation of queries using a second pass is subject to the same rules detailed in previous sections. That is, they must all be condition 1 or all condition 2 if we wish to use a deterministic pass with properly sorted input and output.

5.7 Summary

In this chapter, we proposed a batch evaluation method for expression graphs that satisfies the efficiency bounds given in Section 2.2. The method uses a deterministic pass that iterates both input and output region inventories as one sorted list. This leads to a natural classification of functions according to how their dependencies relate to the sort order. We have examined several types of functions in detail, including structure selection queries, region generation functions, and restricted matching functions, and also considered more general functions.

The main result in this chapter describes the design of completely composable algebras, that is, algebras for which we can evaluate all possible expression graphs using the proposed method. We also examine the idea of stratified algebras that

mix functions of different classifications. The tradeoff that makes this possible is that these algebras incur restrictions on how expression graphs can be constructed if the number of passes required for evaluation is to be controlled.

Chapter 6

Example

6.1 An Algebra

We now describe a small algebra as an example. The functions are listed in Table 6.1 along with their classifications as either condition 1, condition 2, or both, for left-right and right-left pass directions. Descriptions of each function follow. There are two matching functions:

- MATCH-SHORTEST() is as defined previously. It keeps rightmost matches in the case of overlaps.
- MATCH-LONGEST() is as defined previously. It only accepts regular expressions that require at most one character of lookahead.

The following are query functions:

- UNION1(S,D) uses the query $\nexists d \quad (s.l < d.l) \wedge (s.r > d.l)$ to select s regions that do not have touching d region with a later left end. It uses the query

Name	left-right	right-left
MATCH-SHORTEST	2	2
MATCH-LONGEST	2	2
UNION1		2
CONTAINING	2	2
NOT-BEFORE		1,2
NOT-SAME-START	1	2
UNION2	1	
SUBTRACT	1,2	1,2
NOT-OVERLAP-AFTER	1,2	
CONTAINED-IN	1	1
NOT-CONTAINED-IN	1	1
AFTER	1,2	
FIRST-AFTER	1,2	
PAIR-STARTS		2
CUT		1,2
PAIR-REGIONS	2	2

Table 6.1: The functions of the example algebra, and their left-right and right-left classifications.

$\nexists s(d.l \leq s.l) \wedge (d.r > s.l)$ to select d regions that do not have a touching s region with an equal or later left end. The result of a UNION1() call is the set union of the results of the two queries, which is a flat list.

- CONTAINING() is as defined previously.

- NOT-BEFORE(S,D) finds s regions that do not occur immediately preceding a d region. The query is $\nexists d \quad (s.r = d.l)$.

- NOT-SAME-START(S,D) finds s regions that do not share a left end with a d region. The query is $\nexists d(s.l = d.l)$.

- UNION2(S,D) uses the query $\nexists d \quad (s.l > d.l) \wedge (s.l < d.r)$ to select s regions that do not have a touching d region with an earlier left end, and the query

$\nexists s \quad (d.l \geq s.l) \wedge (d.l < s.r)$ to select d regions that do not have a touching s region with an equal or earlier left end. The result of the UNION2() call is the set union of the results of the two queries, which is a flat list.

- SUBTRACT() is as defined previously.
- NOT-OVERLAP-AFTER(S,D) finds s regions that do not overlap after a d. The query is $\nexists d \quad (s.l > d.l) \wedge (s.l < d.r) \wedge (s.r > d.r)$.
- CONTAINED-IN() is as defined previously.
- NOT-CONTAINED-IN() is as defined previously.
- AFTER(S,D) finds s regions that immediately follow a d region. The query is $\exists d \quad (s.l = d.r)$.
- FIRST-AFTER(S,D) finds the first s following each d region. This uses the query $\exists d \quad (s.l > d.r) \wedge (\nexists s_1 \quad (s_1.l > d.r) \wedge (s_1.l < s.l))$.

The following are region generation functions:

- PAIR-STARTS(S) generates a region from the beginning of every s to the beginning of the next s. One additional region is also generated from the start of the last s to the end of the string.
- PAIR-STARTS(S,D) finds closest pairs (s, d) using the query $(\exists d \quad (s.r < d.l)) \wedge (\nexists s_1 \quad (s_1.l > s.r) \wedge (s_1.l < d.l))$. For each such pair, it outputs a g from $s.l$ to $d.l$.

- CUT(S,D) For every s, if there is a d overlapping it on the right, then this function outputs a region from $s.l$ to $d.l$. Otherwise, it outputs s unchanged.

- MERGE-ADJACENT(S) For every $s_1, s_2 \in S$ such that $s_1.r = s_2.l$, this generates the region $(s_1.l, s_2.r)$.

Note that there are no restricted matching functions in this example algebra.

Now consider the parsing model implied by this algebra. Since it contains both condition 1 and condition 2 functions, it must use more than one stratum. Recall that matching functions must always be the leaves of an expression graph. This implies that they must be part of the first stratum. Since matching functions are always condition 2, this means that the first stratum must consist entirely of left-right condition 2 functions or right-left condition 2 functions. Looking at the chart, we can see that some of the functions have been defined so that they are only condition 2 for a right-left pass. Therefore, we make all of the first stratum functions right-left condition 2 so that these functions can be included. The first stratum consists of MATCH-SHORTEST, MATCH-LONGEST, UNION1, CONTAINING, NOT-BEFORE, NOT-SAME-START, SUBTRACT, PAIR-STARTS, CUT, and MERGE-ADJACENT.

Examining the remaining functions, we see that they all have the left-right condition 1 classification in common. Therefore we group UNION2, NOT-OVERLAP-AFTER, CONTAINED-IN, NOT-CONTAINED-IN, AFTER, and FIRST-AFTER into a second stratum.

Overall, the given algebra is divided into two strata, and has a two-pass parsing model. The first pass is right-left and outputs a left-sorted region inventory. The second is left-right and inputs and outputs a left-sorted region inventory.

6.2 *OED* Bibliography

As an example of applying the algebra described in the previous section, we compare two ways of parsing the *Oxford English Dictionary* bibliography. An excerpt from this data is shown in Figure 6.1. The first parser is specified with an INR grammar (Johnson, 1983); the second uses an algebra specification.

6.2.1 The INR grammar

The INR grammar was developed by two different people. The first version was written to find the author, title, and date fields for use in an exploration of automatic citation resolution (Townsend, 1989). It was not important for this purpose that the parse be especially good, only that it find enough fields to give some data for experiments.

The second version of the grammar, which we use here, was an attempt to improve the first. It is still quite simplistic and has many problems. Of the 17,444 entries in the bibliography, it parses 15,410 (about 88%). Of those that it does not parse, 838 are not included in the language defined by the grammar, thus causing the parser to abort and recover starting at the next entry. Another 1196 are parsed by recognizing everything as junk rather than finding any internal structure. There are also many errors in the 15,410 entries that it does parse.

The complete grammar is below. Each production begins with a name, and is followed by an equals sign which separates the left-hand side from the right-hand side. Every production ends with a semi-colon. Right-hand sides are expressed in a notation similar to regular expressions. The plus (`+`), star (`*`), question mark (`?`),

```
<E><CR>+SC A. +CR +R 1593 See +I Passionate Morrice +R </CR>
<E>+SC A., +R A. +NR +I Reply to Dr. Sanderson +R 1650
<E>+SC A., +R D. +NR +I The art of converse +R 1683
<E><CR>+SC A., +R H. +CR 1613, 1633 +I See +SC Austin, +R Henry; +SC
Hawkins, +R Henry</CR>
<E>+SC A., +R W. +NR +I A speciall remedie against the furious force
of lawlesse loue +R 1579 (Roxb. Cl. 1844)
<E>+SC ‘Aarons, +R E. S.’ (Paul Ayres &amp;amp. Edward Ronns) +NR +I
Assignment treason +R 1956
<E>+SC Abbay, +R Richard +NR +I The Castle of Knaresborough +R 1887
<E>+SC Abbot, +R Charles +NR +I Jurisdiction and practice of the Court
of Great Sessions of Wales on the Chester Circuit +R 1795
<E>+SC Abbot, +R Abp. George +NR +I A briefe description of the whole
worlde +R (anon.) 1599 (1617, 1634) +BS An exposition upon the prophet
Jonah +R 1600 +BS A treatise of the perpetuall visibilitie and
succession of the true church +R (anon.) 1624
<E>+SC Abbot, +R George +NR +I The whole book of Job paraphrased +R 1640
<E>+SC Abbot, +R Robert +NR +I The old waye +R 1610
<E>+SC Abbott, +R Charles C. +NR +I Waste-land wanderings +R 1887
<E>+SC Abbott, +R David +NR +I Inorganic chemistry +R 1965
<E>+SC Abbott, +R Edwin A. +NR +I Francis Bacon: an account of his
life and works +R 1885
<E>+SC Abbott, +R Jacob +NR +I Wallace: a Franconia story +R 1853
<E>+SC Abbott, +R John Henry Macartney +NR +I Tommy Cornstalk +R 1902
<E>+SC Abbott, +R John S. C. +NR +I Life of Napoleon +R 1854 (1855)
<E>+SC Abbs, +R Akosua +NR +I Ashanti boy +R 1959
<E>+SC Abdy, +R Edward S. +NR +I The water cure +R 1842 (1843)
<E>+I Aberbrothoc. Liber S. Thome de Aberbrothoc. Registrorum Abbacie
de Aberbrothoc pars prior; pars altera +NR +R v.d. (Bannatyne
Cl. 1848, 1856)
<E>+SC Abercrombie, +R David +NR +I English phonetic texts +R 1964 +BS
Problems and principles: studies in the teaching of English as a
second language +R 1956
```

Figure 6.1: An excerpt from the *OED* bibliography data.

and alternation (`|`) symbols have their usual meanings. Curly brackets `{}` are used to enclose a set, the members of which are separated by commas (which, thus, also, means alternation in this context).

This grammar describes a *transduction* which outputs a modified form of the input text. In general, transduction differs from simple recognition, but in this case, the grammar only echoes the text unchanged except for the insertion of SGML-style tags around recognized elements. A single set of square brackets enclosing a string denotes writing that string to standard output. Thus, for example, `['<D>']` at the beginning of the `date` production inserts that tag at the beginning of any recognized `date`. The output `[RESTART]` is intercepted by the interpreter rather than being output. It denotes a position from which to start re-parsing when there is an error.

Double square brackets have a special meaning that is used to implement a font-checking mechanism. The outputs `[[wr]]` and `[[rr]]` are assertions, short for "write roman" and "read roman". Similarly, `[[wi]]`, `[[ri]]`, `[[ws]]`, `[[rs]]` are corresponding assertions for italics or small caps. The grammar makes a "write" assertion when it finds a code that signals a switch to a different font. For example, `wr` is used when it finds `+R` or `+DM`. It uses a read assertion when it needs to check that it is in a given font. For example, `rr` is used to check that it is in a roman font. The last two productions of the grammar define this behaviour. The `conform` production describes a language where any number of read assertions in a given font can follow a write assertion for that font. The last production then uses the composition (`@`) operator to require that all the assertions made in the other parts of the grammar conform to this language.

```
roman   =       {'+R ' [[wr]], [[rr]]};
ibreak  =       ' '? '+I ' [[wi]];
italic  =       {'+I ' [[wi]], [[ri]]};
sc      =       {'+SC ' [[ws]], [[rs]]};
bs      =       '+BS ' [[wi]];
nr      =       '+NR ';
cr      =       '+CR ';
m2      =       '+M2 ';
dm      =       '+DM ' [[wr]];

blank   =       ('+ES')? {' '} {' '}*;
nl      =       '\n';

see     =       {'S','s'} 'ee ';

digit   =       {'1','2','3','4','5','6','7','8','9','0'};
uc      =       {Q,W,E,R,T,Y,U,I,O,P,A,S,D,F,G,H,J,K,L,Z,X,C,V,B,N,M};
lc      =       {a,c,q,w,e,r,t,y,u,i,o,p,s,d,f,g,h,j,k,l,z,x,v,b,n,m};
spec    =       {'!','@','#','%','^','&','*','-','_','=','\'',
                '`','"','.',',',',','/','?',';',':','{','}','[',']'};

entref  =       '&' {uc, lc, digit, spec}* '.';
pretag  =       {'<i>','</i>','<su>','</su>','<ed>','</ed>','<R>','</R>'};
unit    =       {entref,pretag,nl};
letter  =       {uc,lc,unit};

nonwhite=       {digit,spec,letter,'(',')'};
anything=       {' ',nonwhite};

notword =       (anything* {nl,'<E>',' '} anything*)?;
word    =       notword:acomp;
anytext =       word (blank word)*;
letword =       {(word? letter word?)+, digit digit?};
letext  =       (letword blank)* letword;

number  =       digit+;

yearstr=        (('v.d'|{'1' digit, '7','8','9'}{digit, '.'}{digit, '.'})
                        {'-','.',',',',';',digit,'etc'}* )+;

year    =       (roman '?' ' '?)? (italic {'a','c'} blank)? roman yearstr;
```

```
date    =       ['<D>'] year (blank year)* ['</D>'];


bl      =       ' '+;
badword =       ({'S','s'} 'ee' ' also'? | yearstr );
badtext =       {(bl word?)* (word bl)*,
                (word bl)* badword (bl? word)*,
                {nonwhite,bl,nl}* '<E>' {nonwhite,bl}*  };
text    =       badtext:acomp;

notwork =       ( ({'S','s'} 'ee' ' also'?)?
                        (bl word?)* (word bl)* {'a','c'}?
                | {nonwhite,bl,nl}* '<E>' {nonwhite,bl}*  );

yeswork =       notwork:acomp;
worktext=       italic yeswork;

junk    =       roman (text | '(' yearstr ')'  );



soc     =       '<soc>' junk (blank date)? (blank c)? '</soc>';

author  =       ['<A>'] sc
                { letter anytext ({'.',',','} blank roman
                        ['<AA>'] anytext ['</AA>'])? ['</A>'],
                '`' anytext (',' blank roman ['<AA>'] anytext ['</AA>'])?
                        blank?  '\'' (blank anytext)? ['</A>']};


authors =       author
                 ({' &amp.',',',',blank roman ('or'|'and')} blank author)*;

work    =       ['<W>'] worktext ['</W>'];

datejunk=       date ({blank junk,')'} (blank anytext)?)?;

xref    =       {roman,italic} ({work,junk} blank)?
                see 'also '?
                (junk | {work, authors} {datejunk, blank junk});
```

```
qwork   =       (junk blank)? work
                        (blank junk)?
                        (blank datejunk)?;

dmqw    =       dm
                {['<W>'] junk ['</W>'],
                (junk blank)? ['<W>'] worktext ['</W>'] (blank junk)?}
                (blank {date,'('date')'}  (blank anytext)?  )?;

moreqw  =       (blank? [RESTART]
                                {bs
                                ( qwork (ibreak qwork)*
                                | xref
                                | date (blank junk)? ),
                                dmqw blank? }                   )*;

qworks  =       qwork (ibreak qwork)* moreqw;

aNR     =       authors blank? nr {qworks,xref};

emtwo   =       m2 {text blank?,(text blank)?(date blank)?authors blank?}
                        italic {qworks, qwork blank see 'also '? work};


wNR     =       work blank? nr (junk blank)?
                        datejunk
                        (italic {qwork,
                        see 'also '? roman? authors? roman? anytext?})?
                                moreqw?
                        (blank bs {junk,datejunk})?;


socReg  =       soc work blank
                        (junk blank)? datejunk
                        moreqw;

socSee  =       soc xref;
```

```
socNR   =       soc work blank nr date;

CR      =       '<CR>' {anything,'+','<' '/'? 'soc>'}* '</CR>';


entry   =       '<E>'{emtwo,aNR,wNR,CR,socNR,socReg,socSee} ['</E>'];


conform =       {(wr rr*), (wi ri*), (ws rs*)}* ;
biblio  =       [RESTART] ((entry nl)+ @ conform) ;
```

6.2.2 The Algebra Specification

Our goal with the algebra specification is to produce a result that is objectively comparable to that produced by the grammar. To this end, we developed the algebra specification using the grammar result as an oracle. For each element, we continued refining the set of results until the difference from the set produced by the grammar was small enough that we could examine it manually. In each case, we considered the specification for an element finished when the number of errors made by the algebra in this set was manually verified to be no more than the number of errors made by the grammar.

For interactive purposes, we require that the algebra have some basic functions for managing and examining lists and their identifiers. The following functions are a minimal interface for this purpose:

- SET(String i, List l) Set the identifier i to point to l.
- LIST(List l) Display the first few regions of l.

Note that we use some regular expression facilities beyond those defined in

Chapter 2 — the regular expression language is comparable to that provided by tools such as **grep** and **perl**. In particular, the notation `[a-z]` means the set of all ASCII characters between and including `a` and `z`, the notation `a?` means zero or one occurrences of `a`, the notation `.` means any character in Σ, the notation `\+` means a literal `+` symbol, the notation `\?` means a literal `?` symbol, and the notation `\s` means any whitespace character.

We start by recognizing entries. These are very simple, and the result is identical to that found by the grammar. Note that LIST() also indicates whether the set is an output from a first or second stratum function.

```
> set entry (pair-starts (match-shortest <E>))
> list entry

stratum 1

0,61    <E><CR>+SC A. +CR +R 1593 See +I Passionate Morric ...
61,116  <E>+SC A., +R A. +NR +I Reply to Dr. Sanderson +R  ...
116,168 <E>+SC A., +R D. +NR +I The art of converse +R 168 ...
168,260 <E><CR>+SC A., +R H. +CR 1613, 1633 +I See +SC Aus ...
...

17444 regions
```

Next we find regions delimited by SGML style `CR` tags:

```
> set cross-reference-entry (match-shortest <CR>.*</CR>)
> list cross-reference-entry

stratum 1

3,60         <CR>+SC A. +CR +R 1593 See +I Passionate Morrice + ...
171,259      <CR>+SC A., +R H. +CR 1613, 1633 +I See +SC Austin ...
11638,11705  <CR>+SC Addleshaw, +R W. P. +CR +I See +SC Hemingw ...
12979,13067  <CR>+I Adventures of Captain Robert Boyle +CR +R 1 ...
...
```

```
1196 regions
```

Next we find most of the markup in the text for later use:

```
> set tag (match-shortest (<E>)|(<CR>)|(</CR>))
> set code (union1 (match-longest \+[A-Z]*) tag)
> list code

stratum 1

0,3     <E>
3,7     <CR>
7,10    +SC
14,17   +CR
...

128599 regions
```

We continue by finding the last names of authors. The first command finds `+SC` codes and pairs each one with the nearest following code of any kind to give a region. The second line gets rid of all such regions that occur inside a cross-reference-entry (since the grammar does not recognize any substructures in such entries). Note that the `^` symbol refers to the result of the previous line. The resulting 13265 regions are identical to those found by the grammar, not counting the 838 entries for which the grammar aborts.

```
> pair-starts (match-shortest \+SC) code
> set lastname (not-contained-in ^ cross-reference-entry)
> list lastname

stratum 2

64,72   +SC A.,
119,127 +SC A.,
263,271 +SC A.,
```

```
374,387 +SC ‘Aarons,
...

13265 regions
```

The next step is to find first names. To do this, we pair the tag that switches to a roman font with the closest following region. Note that many of the resulting regions are dates or other text that does not represent a name. Notice also, that PAIR-STARTS() is a stratum 1 function, and that we used a stratum 2 function in the previous step. That is, we can mix calls to stratum 1 and 2 functions arbitrarily in the interactive process as long as results of stratum 2 functions are not used as inputs to stratum 1 functions.

```
> set roman (match-shortest \+R\s)
> set firstname0 (pair-starts roman code)
> list firstname0

stratum 1

18,30   +R 1593 See
72,78   +R A.
108,116 +R 1650
127,133 +R D.
...

44841 regions
```

The first line of the next part finds patterns that are used to separate multiple authors. Any regions in `firstname0` that contain the left end of one of these separators are then truncated using cut, and those that are reduced to just a roman code by this truncation are removed with SUBTRACT(). All resulting regions that immediately follow a last name are then selected and called firstname1.

```
> set author-sep (match-shortest (, \+SC)|(&amp. \+SC)|
  (and \+SC)|(or \+SC))
> subtract (cut firstname0 author-sep) roman
> set firstname1 (after ^ lastname)
> list firstname1

stratum 2

72,78   +R A.
127,133 +R D.
271,277 +R W.
387,429 +R E. S.' (Paul Ayres &amp. Edward Ronns)

13149 regions
```

The next step is to fix cases that occur inside of single quotes. This process is complicated slightly by the fact that a closing single quote is also used as an apostrophe, which can occur inside of some names. The first step finds close quotes followed by a space or + symbol (when used as an apostrophe, the following character is always a letter). Roman codes are paired with the results and called firstname2.

```
> set author-end (match-shortest '(\s|\+))
> set firstname2 (pair-starts roman author-end)
> list firstname2

stratum 1

387,395 +R E. S.
11049,11091     +R 1715 (1721) +BS Essay on ‘Paradise Lost
11974,11989     +R 1896 +BS Doc
17100,17107     +R Milo

855 regions
```

Next, we find complete regions surrounded by quotes. These generally correspond to a last name followed by a first name.

```
> set quoted (match-shortest '.*'(\s|\+))
> list quoted

stratum 1

378,397         'Aarons, +R E. S.'
11077,11093     'Paradise Lost'
17088,17109     'Ainsworth, +R Milo'
17881,17902     'Aird, +R Catherine'

526 regions
```

Finally, we select all firstname2 regions inside both a quoted region and a firstname1. We then combine the result with firstname1, using union2 so that firstname1 regions have lower precedence. Those that start at the same location as a first name inside a quoted region are therefore deleted. Of the resulting 13149 regions, there are 51 cases where the grammar and the algebra result disagree because the grammar makes an obvious error with first names that occur at the end of a line. There are also 63 first names found by the algebra specification that are skipped over as junk by the grammar. (This is not counting the entries for which the grammar aborts.) Overall, we consider these differences slight enough to say that the two results are essentially the same.

```
> contained-in (contained-in firstname2 quoted) firstname1
> set firstname (union2 ^ firstname1)
> list firstname

stratum 2

72,78   +R A.
127,133 +R D.
271,277 +R W.
387,395 +R E. S.
```

```
13149 regions
```

Next, we find titles of works. The first steps find regions running from the beginning of any italic region to the beginning of the next code, or date (`a` for ante, `c` for circa, `v.d.` for various dates), or cross reference (`See also`).

```
> set work-left (match-shortest (\+BS )|(\+I ))
> set work-right (match-shortest ( a \+)|( c \+)|( v\.d\. )|
  ( See (also )?))
> cut (pair-starts work-left code) work-right
> list work1

stratum 1

30,52   +I Passionate Morrice
82,108  +I Reply to Dr. Sanderson
137,160 +I The art of converse
281,346 +I A speciall remedie against the furious force of ...

29480 regions
```

This next part finds works beginning with `+DM` codes. These are different from italic works in that the end of the title is not usually separated from a date by a code. (This can be considered an inconsistency in the markup.) Therefore, we find the right end of these works by looking either for patterns that look like a year, or the end of a line.

```
> set dm-left (match-shortest (\+DM ))
> set dm-right (match-shortest ( \d\d(\d|\.))|(\(\d\d\d\d[,;])|(\n))
> pair-starts dm-left (union1 dm-left work-left)
> cut (cut ^ dm-right) work-right
> set work2 (con ^ dm-left)
> list work2

stratum 1
```

```
3121,3132       +DM (ed. 2)
5559,5577       +DM (complete ed.)
8717,8731       +DM (rev. ed.)
18883,18922     +DM (ed. 2, ed. by W. T. Aiton) 5 vols.

539 regions
```

The purpose of the next part is to simulate an inconsistency in how the grammar treats cases where the start of a date *is* signalled with a font change.

```
> set i (match-shortest \+I)
> not-same-start i (match-shortest \+I c)
> set work3 (not-before work2 ^)
> list (sub work2 work3)

stratum 1

170512,170536   +DM (another ed., with)
173349,173368   +DM Sunday ed., as
249305,249331   +DM another ed., entitled
378010,378021   +DM (with)

99 regions

> list work3

stratum 1

3121,3132       +DM (ed. 2)
5559,5577       +DM (complete ed.)
8717,8731       +DM (rev. ed.)
18883,18922     +DM (ed. 2, ed. by W. T. Aiton) 5 vols.

440 regions
```

The final section combines the italic and `+DM` cases and deletes those inside a cross-reference entry. Of the 29006 resulting works, there are six discrepencies between the grammar and the algebra result. Four of these are errors on the part of the

grammar, and two are cases that the grammar misses.

```
> union2 work1 work3
> not-contained-in ^ cross-reference-entry
> set work ^
> list work

stratum 2

82,108  +I Reply to Dr. Sanderson
137,160 +I The art of converse
281,346 +I A speciall remedie against the furious force of ...
433,455 +I Assignment treason

29006 regions
```

We next search for date elements. We start by finding patterns that look like a year, possibly preceded by an ante or circa:

```
> match-longest (( a \+R )|( c \+R ))?((1\d[\d\.][\d\.])|([789]\d\d))
  [\?\-.,;\d ]*

Warning: overlaps detected.  Keeping rightmost.

> set year ^
```

Note that the overlap occurs in cases such as the following when a year occurs as the last part of a title:

```
+I The autobiography and personal diary, from 1552 to 1602 a +R 1611
```

In this case, keeping the rightmost, `a +R 1611`, rather than `1602` is the correct behaviour. The next step is to find other regions that can be part of a date, and union them all into a single list. Next, adjacent parts are merged into a single date using MERGE-ADJACENT. Any resulting regions that cannot comprise a date

by themselves (a single italic code, or a single question mark) are then removed. Finally, all dates that are inside a work, or overlapped after one are deleted.

```
> set vd (match-shortest (v\.d\.? ?))
> set q (match-shortest ( \? ?))
> union1 (union1 (union1 year vd) i) q
> merge-adjacent ^
> sub (sub ^ i) q
> set date1 (not-overlap-after (not-contained-in ^ work) work)
> list date1

stratum 2

21,26   1593
111,115 1650
163,167 1683
193,206 1613, 1633 +I

36381 regions
```

The next part emulates an error in the grammar: it discards all dates except for the first one following each work.

```
> set date2 (first-after date1 work)
> set date3 (first-after date1 bs)
> union2 date2 date3
> set date (not-contained-in ^ cross-reference-entry)
> list date

stratum 2

111,115 1650
163,167 1683
349,354 1579
458,462 1956

28650 regions
```

Of the final 28650 results, 162 are correctly identified dates that are missed by the

grammar. There are also 4 cases that the grammar incorrectly marks as dates even though they are part of a title, 10 cases where the two results disagree because the grammar chooses a date other than the first one following a work, and 30 cases where the grammar finds only a substring of a full date found by the algebra specification.

Following is the complete algebra specification without commentary:

```
# find entries and codes
set entry (pair-starts (match-shortest <E>))
set cross-reference-entry (match-shortest <CR>.*</CR>)

set tag (match-shortest (<E>)|(<CR>)|(</CR>))
set code (union1 (match-longest \+[A-Z]*) tag)


# find last names
pair-starts (match-shortest \+SC) code
set lastname (not-contained-in ^ cross-reference-entry)


# find firstnames
set roman (match-shortest \+R\s)
set firstname0 (pair-starts roman code)

set author-sep (match-shortest (, \+SC)|(&amp. \+SC)|(and \+SC)|(or \+SC))
subtract (cut firstname0 author-sep) roman
set firstname1 (after ^ lastname)

set author-end (match-shortest \'(\s|\+))
set firstname2 (pair-starts roman author-end)

set quoted (match-shortest \‘.*\'(\s|\+))
contained-in (contained-in firstname2 quoted) firstname1
set firstname (union2 ^ firstname1)


# find works
```

```
set work-left (match-shortest (\+BS )|(\+I ))
set work-right (match-shortest ( a \+)|( c \+)|( v\.d\. )|( See (also )?))
cut (pair-starts work-left code) work-right

set dm-left (match-shortest (\+DM ))
set dm-right (match-shortest ( \d\d(\d|\.))|(\(\d\d\d\d[,;])|(\n))
pair-starts dm-left (union1 dm-left work-left)
cut (cut ^ dm-right) work-right
set work2 (con ^ dm-left)

set i (match-shortest \+I)
not-same-start i (match-shortest \+I c)
set work3 (not-before work2 ^)

union2 work1 work3
not-contained-in ^ cross-reference-entry
set work ^


# find dates
match-longest (( a \+R )|( c \+R ))?((1\d[\d\.][\d\.])|([789]\d\d))
[\?\-.,;\d ]*
set year ^

set vd (match-shortest (v\.d\.? ?))
set q (match-shortest ( \? ?))
union1 (union1 (union1 year vd) i) q
merge-adjacent ^
sub (sub ^ i) q
set date1 (not-overlap-after (not-contained-in ^ work) work)

set date2 (first-after date1 work)
set date3 (first-after date1 bs)
union2 date2 date3
set date (not-contained-in ^ cross-reference-entry)
```

6.2.3 Complexity

We now compare the understandability of the grammar and the algebra specification using some simple metrics. Not counting the `nl`, `digit`, `uc`, `lc`, and `spec` elements which are trivial, there are 53 productions in the grammar. This is about the same as the total number of function calls in the algebra specification which is 54 (omitting SET() and LIST() calls).

The two specifications differ more significantly in the number of dependencies. Let a *dependency graph* for a grammar be defined as follows: there is a node for every non-terminal, and an arc from one node to another if there is a production with the source as the left-hand side and the destination in the right-hand side. The dependency graph for the grammar has 53 nodes and 126 edges, as compared to 54 nodes and 68 edges for the algebra expression graph. The two graphs are shown in Figures 6.2 and 6.3[1]. From this, we conclude that the grammar is harder to understand, and also harder to extend, than the algebra specification.

The discrepancy is really worse than implied by the larger number of dependencies. This is because it is possible to extend an algebra specification without understanding it at all. A new element can always be created independently of existing elements, and optionally related to existing elements later. This is how the date element was developed, for example: we found dates independently of other elements, then deleted those that were inside a title. This type of modular development is not possible with the grammar: it is always necessary to determine where to fit in a new element, and to consider how this affects existing elements.

[1]We generated these graphs and performed edge-crossover minimization using the *daVinci* program (Fröhlich & Werner, 1994)

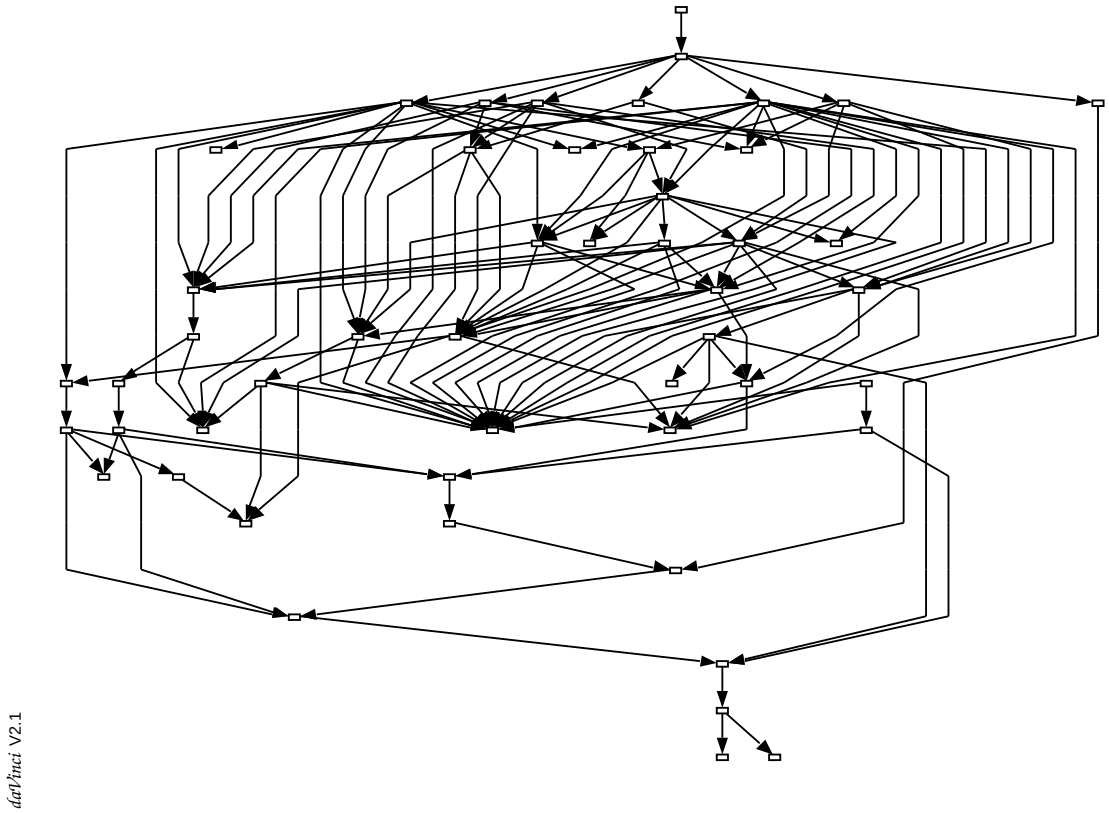


Figure 6.2: The dependency graph for the grammar

.

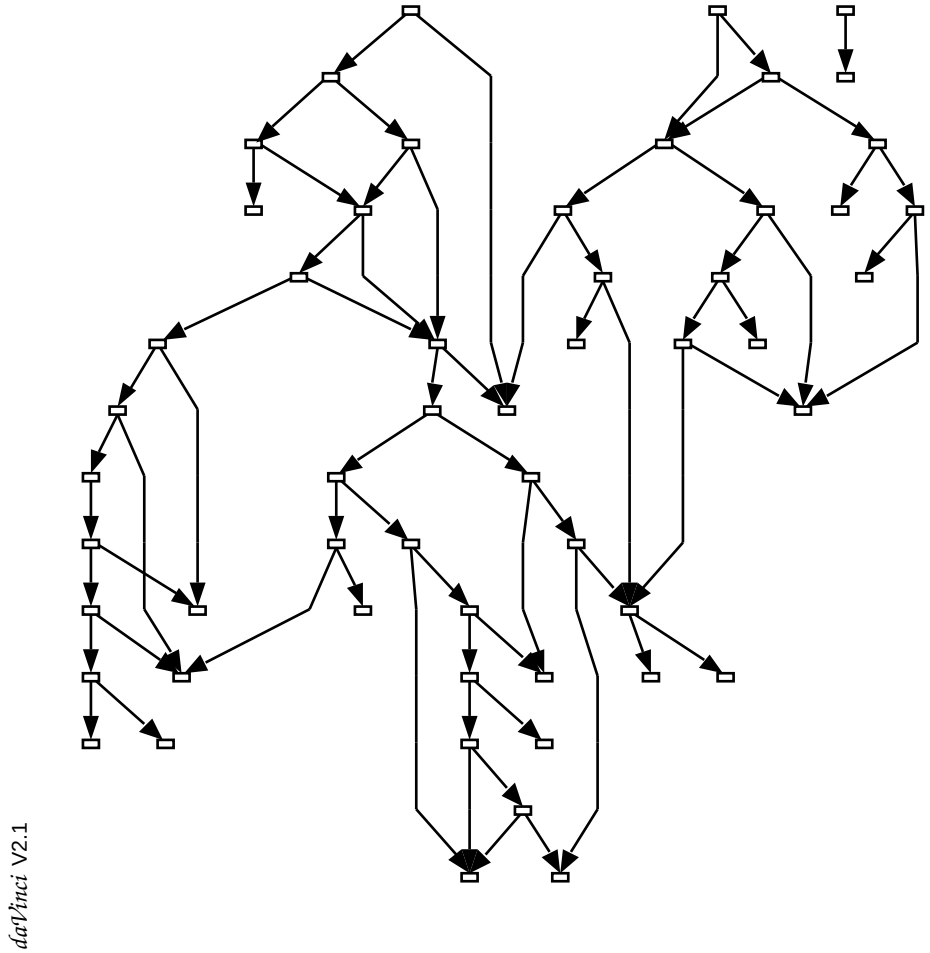


Figure 6.3: The expression graph for the algebra specification.

6.2.4 Batch Efficiency

When parsing with a grammar, we do not need to output the entire parse tree. Rather, we can specify a subset of the productions in which we are interested. For this example, these are the date, author, work, and entry productions. In the same way, for an algebra specification, we can specify a subset of the lists in which we are interested. This is the distinction we made earlier between lists that are final results, and those that are intermediate results.

Parsing with a grammar, it is only necessary to output final results. Intermediate results must be found but need never be output. For the example, the final results total 99,287 regions. Assuming that each region is represented with eight bytes (two integers), this gives a total output size of 794,296 bytes. The required input is the size of the bibliography which is 2,163,877 bytes. Therefore, the total I/O required to parse using the grammar is about 3 Mb.

Parsing an algebra specification requires that some of the intermediate results be output. Specifically, any lists that are outputs of class one functions and inputs to class two functions must be output during the first pass and read in during the second. In the example, there are 10 such lists containing a total of 152,828 regions. To write these once, and then read them again at 8 bytes per region uses a total of about 2.4 Mb of I/O. Together with reading the file, and writing the final results once, the total I/O cost of parsing the algebra specification is about 5.4 Mb.

Overall then, parsing with the algebra is about 80 percent more costly in terms of I/O than using the grammar — roughly what would be expected given that it uses two passes. What we gain from this extra cost is that both passes are

deterministic, in contrast with the grammar which has non-determinism that can require unbounded lookahead to resolve. This does not make any practical difference for this example where lookahead never extends past the beginning of the next bibliographic entry, but could be a problem for other data.

6.2.5 Interactive Efficiency

When developing a grammar, every change that we make requires a reparse of the data. Recall that, for the example, a single parse uses about 3 Mb of I/O. Thus, assuming that the number of output regions stays about the same for each step, the total I/O cost of an interactive grammar development process is the number of changes that we make times 3 Mb.

Recall that there are a total of 54 function calls in the algebra specification. 19 of these are matching calls that scan the data, and the remainder operate on lists of regions. Executed interactively one by one, the 19 matching calls therefore read the 2.2 Mb of the file 19 times for a total of 41 Mb read from secondary storage. (Thirteen of the matching calls are for constant strings that would be easy to search for using an index, but we do not take this into account for this discussion.) The remaining calls read or write a total of 3,350,114 regions from secondary storage. Assuming again 8 bytes (two integers) per region, this is 27 Mb. The overall total amount of I/O that is used to interactively execute the algebra specification one function call at a time is therefore about 68 Megabytes, or an average of 1.3 Mb per call.

Assuming that the above discussion is a reasonable estimation of the average

I/O required for an algebra call, we can conclude that more than twice as many interactive steps are possible with the algebra given the same amount of I/O as required for a grammar. In general, this means that interactive exploration and experimentation can be carried out at a more fine-grained level.

Overall, our conclusion is that incrementally developing a parser using the algebra is both significantly simpler and also more efficient, than using a more traditional grammar approach.

Chapter 7

Summary and Future Work

7.1 Summary

In Chapter 3, we identify four important characteristics that a system for text structure recognition should have if it is to used for incremental development of recognizer specifications. These are 1) interactive efficiency, 2) a flexible structure model, 3) scalability, and 4) batch efficiency. We argue that existing formalisms, and grammars in particular, all lack one or more of these characteristics, and propose an alternative approach based around a region algebra as an interface.

We argue in principle that an approach based on an algebra is better than existing approaches from the point of view of interactive efficiency, structure model flexibility, and scalability in Chapter 3. We also demonstrate anectodally in Chapter 6 that this is true specifically in comparison to a grammar-based approach. In addition, we have made the prototype tool used to construct the example in Chapter 6 available to the *Dictionary of Old English* project (Healey, Holland, McDougall,

& Mielke, 2000). They had previously evaluated and rejected the idea of using a grammar-based approach for recognition, and have found the prototype to be a much more appropriate tool (Healey & Mielke, 2000). Note also that other examples exist where grammar-based approaches were evaluated and rejected in favour of pattern matching approaches similar to region algebras (Murphy & Notkin, 1996; Miller & Myers, 1999; Aït-Mokhtar & Chanod, 1997; Grefenstette, 1996).

An important characteristic that a region algebra lacks for this application is efficient batch evaluation. Therefore, in Chapter 5, we propose a batch evaluation method for region algebra expression graphs. This views the set of inputs and outputs to an expression graph as a single, merged region inventory, and evaluates the graph using a single deterministic pass over this inventory. Unfortunately, it is easy, using common region algebra operations, to construct expressions that cannot be evaluated in this way.

Our main results characterize the conditions under which it is possible to evaluate an expression graph using a single deterministic pass (that is, one that does not use lookahead). A function that selects s regions based on d regions is classified as condition 1 for a left-right pass if its dependency expression $E(Q)$ implies $(d.l \leq s.l)$. It is condition 2 for a left-right pass if $E(Q) \Rightarrow (d.r \leq s.r)$, condition 1 for a right-left pass if $E(Q) \Rightarrow (d.r \geq s.r)$, or condition 2 for a right-left pass if $E(Q) \Rightarrow (d.l \geq s.l)$. For functions where s depends on a set D of other regions, the implication must hold for every d in D. Using this classification, we prove that, for a given algebra, we can evaluate all expression graphs that can be constructed with that algebra if and only if all functions are condition 1, or all are condition 2

for the same pass direction.

Based on this result, we can construct completely composable algebras for which we can evaluate all possible expression graphs in one pass. However, there are limitations on the functions that can be included in a completely composable algebra, which may be a problem in some cases. Therefore, we also consider one way of overcoming this limitation: divide an algebra into multiple strata, and use a multiple-pass evaluation method. We show that such an approach is feasible using a two-strata example in Chapter 6. Note, however, that we do not rule out the possibility that other useful ways of overcoming the limitation may be more appropriate.

We also give results having to do with regular expression matching. This is an important operation since we are interested in expression graphs that have string-to-region functions at the leaves and therefore recognize structure starting from nothing but a string. It is not possible to perform longest regular expression matching as part of a single deterministic pass. However, as proved in Section 4.1, we can perform longest matching using two deterministic passes, and we can perform longest matching using one deterministic pass for certain subclasses of regular expressions. An additional result in Section 5.6 characterizes the conditions under which we can perform regular expression matching that is restricted according to uncertain anchors in the string.

7.2 Future Work

7.2.1 Interface Concerns

The main consideration in this thesis is efficient batch evaluation when using a region algebra for specification. Thus, we did not design a comprehensive region algebra, nor examine interface issues in detail. Rather, we introduced as many functions as needed to demonstrate and prove specific principles, and to construct an example. More work is needed to look at user interface questions. For example, is it possible for an expert to design a comprehensive region algebra that is applicable to a wide range of recognition problems, or is designing a customized algebra a necessary part of the recognition process? Also, are stratified algebras appropriate from a user's perspective, or are the limitations on constructing expressions too distracting?

Another important user interface concern is whether there is a simple characterization of the type of processing that can be done using a deterministic pass. In principle, any processing that does not use lookahead in the region inventory is possible. However, this is restricted by the fact that we specify the processing by composing operations from an algebra. In practise, interface issues such as the desire for complete composability may limit the scope of the processing that can be specified. Further work is needed to determine whether it is possible to design an algebra of simple operations that is usable, small, orthogonal, and *complete* in the sense that it allows specification of any processing possible with the evaluation model.

A high level concern is that our approach is a tool for designing recognition specifications rather than a design methodology (in the same way that an object oriented programming language is a tool rather than a methodology). The process of designing large recognition specifications could still benefit greatly from the development of some kind of formal methodology similar to those used in software engineering.

As mentioned in Chapter 2, learning approaches to recognition have been explored by others, but suffer from the limitation of too many simplifying assumptions. A region algebra seems to be an interface with an appropriate level of power, but this level of power makes it too complicated to learn automatically. Perhaps learning approaches would be more usefully targeted at the task of helping the user understand what to specify, rather than trying to generate a specification themselves. In other words, they could be used as a tool for exploring the data.

7.2.2 Efficiency Concerns

Our main result shows the conditions under which all expressions constructible with an algebra are evaluable with a deterministic pass. Since these conditions restrict the functions that can be included in the algebra, we have also introduced the idea of stratified algebras, which allow a larger selection of functions but place restrictions on how expressions are constructed. It might be useful to explore other ways of arranging this tradeoff, that is, of allowing more functions, but adding restrictions on how they are composed.

We have only given detailed consideration to deterministic passes with properly

sorted inputs and outputs. However, it might be useful to take a closer look at other variants of deterministic passes. For example, suppose that the input and output region inventories are not strictly left-sorted or right-sorted, but rather, have all regions of a given type displaced by a constant number of positions from their left or right sorted positions; or, suppose that regions of each type are sorted within a region inventory but there is no restriction on the ordering of regions of different types. The goal of exploring such variants would be to find additional cases with rules for expression construction that could be as simply stated as those for deterministic, properly-sorted passes.

An efficiency issue with an unrestricted structure model is that there is no bound on the number of regions. A flat list representation gives a bound of ns regions where s is the number of lists. However, it might be useful to examine more restrictive models that allow a stronger bound without going as far as requiring a strict hierarchy.

Our results can be seen as a form of query optimization applicable to structured query languages. Traditional forms of query optimization involve performing transformations on the expression graph, or formulating a query plan for sequential evaluation of the queries so as to reduce the size of intermediate results (e.g., Consens (1998)). In contrast, our approach evaluates many queries in an expression simultaneously in each pass. Integrating this with traditional approaches could potentially result in a better overall query optimization method.

Bibliography

Adelberg, B. (1998). NoDoSE: A tool for semi-automatically extracting structured and semi-structured data from text documents. *SIGMOD Record*, *27*(2), 283–294.

Aho, A. V., Kernighan, B., & Weinberger, P. (1978). Awk – A pattern scanning and processing language. Tech. rep., Bell Telephone Laboratories.

Aho, A. V., & Ullman, J. D. (1972). *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, Inc., Englewood Cliffs, N.J.

Aho, A., Hopcroft, J., & Ullman, J. (1974). *The Design and Analysis of Computer Algorithms*. Addison-Wesly, Reading, Massachusetts.

Aït-Mokhtar, S., & Chanod, J.-P. (1997). Incremental finite state-parsing. In *Proceedings of ANLP'97*, pp. 72–79 Washington.

Ashish, N., & Knoblock, C. A. (1997). Wrapper generation for semi-structured internet sources. *SIGMOD Record*, *26*(4).

Baeza-Yates, R. (1989). *Efficient Text Searching*. Ph.D. thesis, Department of

Computer Science, University of Waterloo. Also available as technical report CS-89-17.

Baeza-Yates, R., & Gonnet, G. H. (1996). Fast searching for regular expressions or automaton searching on tries. *JACM*, *43*(6), 915–936.

Baeza-Yates, R., & Navarro, G. (1996). Integrating contents and structure in text retrieval. *SIGMOD Record*, *25*(1).

Berstel, J. (1979). *Transductions and Context-free languages.* Teubner, Stuttgart, Germany.

Blake, G. E., Bray, T., & Tompa, F. W. (1992). Shortening the *OED*: Experience with a grammar-defined database. *ACM Transactions on Information Systems*, *10*(3), 213–232.

Burkowski, F. J. (1994). An algebra for hierarchical organized text-dominated databases. *Information Processing and Management*, *28*, 313–324.

Clark, D. (1991). Finite state transduction tools. Tech. rep. OED-91-03, UW Centre for the New Oxford English Dictionary and Text Research, Waterloo, Ontario.

Clark, J. (1999). XSL transformations (XSLT) version 1.0. world wide web consortium reccomendation. `http://www.w3.org/TR/xslt`.

Clarke, C., & Cormack, G. (1997). On the use of regular expressions for searching text. *ACM Transactions on Programming Languages and Systems*, *19*, 413–426.

Clarke, C., & Cormack, G. (2000). Shortest-substring retrieval and ranking. *ACM Transactions on Information Systems*, *18*(1), 44–78.

Clarke, C., Cormack, G., & Burkowski, F. (1995). An algebra for structured text search and a framework for its implementation. *Computer Journal*, *38*(1), 43–56.

Consens, M. P. (1998). Algebras for querying text regions: Expressive power and optimization. *Journal of Computer and System Sciences*, *57*, 272–88.

Coombs, J. H., Renear, A. H., & DeRose, S. J. (1987). Markup systems and the future of scholarly text processing. *Communications of the ACM*, *30*(11), 933–947.

Crespo, A., Jannink, J., Neuhold, E., Rys, M., & Studer, R. (2000). A survey of semi-automatic extraction and transformation. Submitted to *Information Systems*.

Dao, T., Sacks-Davis, R., & Thom, J. (1996). Indexing structured text for queries on containment relationships. In *Proceedings of the 7th Australasia Database Conference*, pp. 82–91 Melbourne, Australia.

Dougherty, D. (1991). *SED & AWK*. O'Reilly & Associates, Inc., Newton, MA.

Earley, J. (1970). An efficient context-free parsing algorithm. *Communications of the ACM*, *13*(2), 94–102.

Eilenberg, S. (1974). *Automata, Languages, and Machines*, Vol. A. Academic Press, New York.

Exoterica Corporation (1993). OmniMark programmer's guide. (software manual).

Fankhauser, P., & Xu, Y. (1993). *MarkItUp!* an incremental approach to document structure recognition. *Electronic Publishing – Origin, Dissemination and Design*, *6*(4), 447–456.

Fröhlich, M., & Werner, M. (1994). The graph visualization system daVinci - a user interface for applications. Tech. rep. 5/94, Department of Computer Science; Universität Bremen. `ftp://ftp.uni-bremen.de/pub/graphics/daVinci/papers/techrep0594.ps.gz`.

Garofalakis, M., Gionis, A., Rastogi, R., Seshadri, S., & Shim, K. (2000). XTRACT: A system for extracting document type descriptors from XML documents. In *SIGMOD/PODS* Dallas, Texas.

Gimpel, J. (1973). A theory of discrete patterns and their implementation in SNOBOL4. *Communications of the ACM*, *16*(2), 91–100.

Goldfarb, C. F. (1990). *The SGML Handbook*. Clarendon Press, London.

Göttke, T., & Fankhauser, P. (1992). *DREAM 2.0 User Manual*. Arbeitspapiere der GMD, No. 660, Sankt Augustin.

Grefenstette, G. (1996). Light parsing as finite-state filtering. In Kornai, A. (Ed.), *Proceedings ECAI'96 workshop on Extended Finite State Models of Language* Budapest.

Hammer, J., Garcia-Molina, H., Cho, J., Aranha, R., & Crespo, A. (1997). Extracting semi-structured data from the web. In *Proceedings of Workshop on Management of Semi-structured Data*, pp. 18–25.

Healey, A., Holland, J., McDougall, I., & Mielke, P. (2000). *The Dictionary of Old English Corpus in Electronic Form, TEI-P3 conformant version, 2000 Release*. DOE Project, Toronto. `http://www.doe.utoronto.ca/corpus.html`.

Healey, A., & Mielke, P. (2000) Personal communication.

Hopcroft, J. E., & Ullman, J. D. (1979). *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, Massachusetts.

IEEE (1992). Standard for information technology — portable operating system interface (posix) — part 2 (shell and utilities) — section 2.8 (regular expression notation). Tech. rep. IEEE Std 1003.2, Institute of Electrical and Electronics Engineers.

ISO (1986). Information processing – text and office systems – standard generalized markup language (SGML). International Organization for Standardization ISO/IEC 8879.

ISO (1996). Document style semantics and specification language (DSSSL). International Organization for Standardization ISO/IEC 10179.

Ives, Z., Levy, A., & Weld, D. (2000). Efficient evaluation of regular path expressions on streaming xml data. Tech. rep., University of Washington.

Jaakkola, J., & Kilpeläinen, P. (1999). Nested text-region algebra. Tech. rep. C-1999-2, Department of Computer Science, University of Helsinki.

Johnson, H. J. (1983). *Formal Models for String Similarity.* Ph.D. thesis, University of Waterloo, Department of Computer Science.

Johnson, H. J. (1987). Single-valued finite transduction. In Ottmann, T. (Ed.), *Automata, Languages and Programming, 14th International Colloquium (ICALP87), Karlsruhe, Germany.*

Johnson, H. J. (1989). INR: A program for computing finite automata. Tech. rep., Department of Computer Science, University of Waterloo.

Johnson, S. C. (1975). Yacc: Yet another compiler-compiler. Tech. rep. CSTR32, Bell Laboratories, Murray Hill, New Jersey.

Karttunen, L. (1992). Constructing lexical transducers. In *Proceedings of the Fifteenth International Conference on Computational Linguistics*, pp. 405–411 Kyoto, Japan.

Karttunen, L. (1995). The replace operator. In *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics, ACL-95*, pp. 16–23 Boston, Massachusetts.

Karttunen, L. (1996). Directed replacement. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics, ACL-96* Santa Cruz, California.

Kazman, R. (1986). Structuring the text of the *Oxford English Dictionary* through finite state transduction. Master's thesis, Computer Science Department, University of Waterloo. Also available as technical report CS-86-20.

Keller, S., Perkins, J., Payton, T., & Mardinly, S. (1984). Tree transformation techniques and experiences. *SIGPLAN Notices (Proc. of the ACM SIGPLAN '84 Symposium on Compiler Construction)*, *19*(6), 190–201.

Kilpeläinen, P., & Mannila, H. (1993). Retrieval from hierarchical texts by partial patterns. In Korfhage, R., Rasmussen, E., & Willett, P. (Eds.), *SIGIR '93 – Proceedings of the 16th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 214–222 New York.

Knoblock, C., Minton, S., Ambite, J., & Ashish, N. (1998). Modeling web sources for information integration. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98), Madison, Wisconsin.*

Kushmerick, N., Weld, D., & Doorenbos, R. (1997). Wrapper induction for information extraction. In *Proceedings of 15th International Concerence on Artificial Intelligence*, pp. 729–735.

Lalonde, W. (1977). Regular right part grammars and their parsers. *Communications of the ACM*, *20*(10), 731–741.

Lesk, M. E., & Schmidt, E. (1984). Lex — a lexical analyser generator. In *Unix Programmer's Manual.*

Lewis, P. M., & Stearns, R. E. (1968). Syntax-directed transduction. *Journal of the ACM*, *15*(3), 465–488.

Lindén, G. (1997). *Structured Document Transformations*. Ph.D. thesis, University of Helsinki, Finland.

Mamrak, S. A., O'Connell, C. S., & Barnes, J. (1992). *Technical Documentation for the Integrated Chameleon Architecture*. `ftp://ftp.ifi.uio.no/pub/SGML/ICA/ICAdoc-1.2.tar.gz`.

Manber, U., & Wu, S. (1993). GLIMPSE: a tool to search through entire file systems.. Tech. rep. 93-34, Dept. Computer Science, University of Arizona.

Michalski, R. S., Carbonell, J. G., & Mitchell, T. M. (Eds.). (1983). *Machine Learning, an Artificial Intelligence approach*, Vol. 1. Morgan Kaufmann, San Mateo, California.

Miller, R. C., & Myers, B. C. (1999). Lightweight structured text processing. *Usenix Annual Technical Conference*, 131–144.

Mo, D. H., & Witten, I. H. (1992). Learning text editing tasks from examples: A procedural approach. *Behaviour & Information Technology*, *14*(1), 32–45.

Murphy, G. C., & Notkin, D. (1996). Lightweight lexical source model extraction. *ACM Transactions on Software Engineering and Methodology*, *5*(3), 262–292.

Muslea, I., Minton, S., & Knoblock, C. (1998). Wrapper induction for semistructured, web-based information sources. In *Proceedings of the Conference on Automatic Learning and Discovery (CONALD-98)*.

Navarro, G. (1995). A language for queries on structure and contents of textual databases. In *Proceedings of the 18th SIGIR*, pp. 93–101 Seattle, Washington.

Nix, R. (1989). *Editing by Example*. Ph.D. thesis, Computer Science Department, Yale University.

Quint, V., & Vatton, I. (1986). GRIF: An interactive system for structured document manipulation. In van Vliet, J. C. (Ed.), *EP86 — Proceedings of the International Conference on Text Processing and Document Manipulation*, pp. 200–213 Nottingham, UK. Cambridge University Press.

Reed-Lade, M. (1989). Grammar acquisition and parsing of semi-structured text. Master's thesis, Artificial Intelligence Laboratory, University of Texas at Austin.

Salminen, A., & Tompa, F. (1992). PAT expressions: an algebra for text search. *Acta Linguistica Hungarica*, *41* (1–4), 277–306.

SGML Systems Engineering Ltd. SGMLC overview. `http://www.dircon.co.uk/sgml/overview.htm`.

Townsend, G. (1989). Citation matching in the *OED*. Master's thesis, University of Waterloo, Department of Computer Science. Also available as technical report OED-89-06.

Valiant, I. (1975). General context-free recognition in less than cubic time. *Journal of Computer and Systems Sciences*, *10* (2), 308–315.

van den Brand, M., Sellink, A., & Verhoef, C. (1998). Current parsing techniques in software renovation considered harmful. In *Proceedings of the International Workshop on Program Comprehension*, pp. 108–117 Ischia, Italy.

Wall, L., Schwartz, R. L., Christiansen, T., & Potter, S. (1996). *Programming Perl* (2nd edition). O'Reilly & Associates.

Young-Lai, M., & Tompa, F. W. (2000). Stochastic grammatical inference of text database structure. *Machine Learning*, *40*(2), 111–137.