Chapter 3

Region Algebra

The fundamental unit of lightweight structure is a contiguous segment of text, called a *region*. Other names for the same concept are runs, substrings, and pieces [KM98]. Many features of text-processing systems operate on regions:

- **Styles.** In a word processor, styles like font, color, boldface, italics, underlining, or strikeout are associated with regions.
- **Selection.** When a user selects text with a mouse, the selection is a region. The text-insertion point may also be represented as a *zero-length* region, i.e., a region whose start point and end point are identical.
- **Hyperlinking.** In a web browser, hyperlinks are represented as clickable text regions. The target of a hyperlink is also a region, either in the same web page or a different page.
- **Pattern matching.** The result of searching for a pattern in a text editor or browser is a region that matches the pattern.
- **Logical document structure.** Chapters, sections, paragraphs, sentences, and words can all be represented by regions.
- **Physical document structure.** Pages, frames, columns, and lines can be represented by regions.
- **Language syntax.** Expressions and statements in source code and phrases in natural language can be represented by regions.

The lightweight structure model is mainly concerned with *sets* of regions, because a structure abstraction returns a set of regions. Figure 3.1 shows some examples of region sets.

The goal of this chapter is to develop an algebra for combining sets of regions. The algebra will enable a number of implementation alternatives and optimizations (Chapter 4), and will serve as the basis of a user-level pattern language (Chapter 6). The region algebra plays the same role in lightweight structure that relational algebra does in relational databases: relational algebra is a formal model of relational data that acts as a target for implementations (database engines) and the basis of a user-level language (SQL).

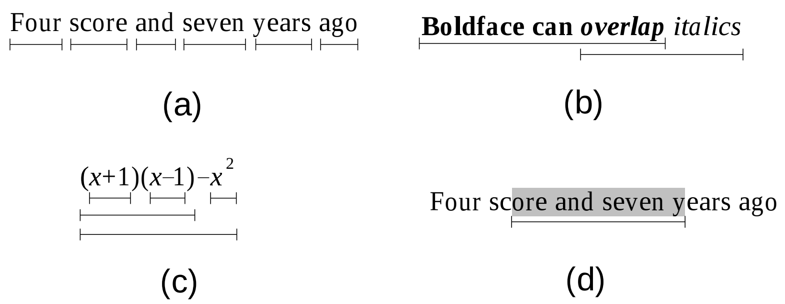

Four score and seven years ago
Boldface can overlap italics
(a)
(b)
(x+1)(x–1)–x2
Four score and seven years ago
(c)
(d)

Figure 3.1: Example region sets. (a) words; (b) styles; (c) expressions; (d) a selection made by a user.

The applications listed above suggest some properties that need to be captured by the region algebra:

- **Regions may be nested in other regions**. In documents, words are completely contained in a sentence, sentences in a paragraph, and paragraphs in a section. In programs, expressions are nested in other expressions, as in Figure 3.1(c). Natural language also obeys a nested structure: noun phrases are nested in verb phrases, and verb phrases in clauses or sentences.

- **Regions may overlap other regions.** Logical structure can overlap physical structure in complex ways. For example, sentences and lines often overlap without nesting. Styles may also overlap, as in Figure 3.1(b).

- **Regions may be zero-length.** In a text editor, it is often convenient to treat every user selection as a region, including the text insertion caret that appears *between* two characters. Some structuring attributes may also be represented as zero-length regions, such as hyperlink targets, page breaks, and hyphenation hints.

Since text structure is so varied — nested, overlapping, zero-length — the region algebra should be capable of representing *arbitrary* sets of regions. Unfortunately, the size of an arbitrary region set may be quadratic in the length of the string. For a simple example, consider the set of all regions in a string of length n. There are n regions starting at the beginning of the string, $n-1$ regions starting after the first character, etc. Thus the total number of regions is $\sum_{i=1}^{n} i = n(n+1)/2$, which is $O(n^2)$. Quadratic region sets cause problems for efficient implementation. Previous systems have dealt with this problem by restricting region sets to certain types that are guaranteed to be linear, sacrificing expressiveness for linear processing.

This chapter[1] takes a more general approach, describing an algebra that can combine arbitrary sets of regions. The general algebra has two main advantages. First, it is very simple, consisting only of a handful of relational operators, the standard set operators, and an iteration operator. The simplicity will make it easier to prove its theoretical power in Chapter 5. Previous systems needed far more operators. Second, supporting arbitrary region sets means that the algebra can combine structure found by arbitrary parsers. Users of the algebra need not be concerned with whether two region sets satisfy a particular restriction before combining them. This property helps human users

[1]Portions of this chapter and the next are adapted from an earlier paper [MM99].

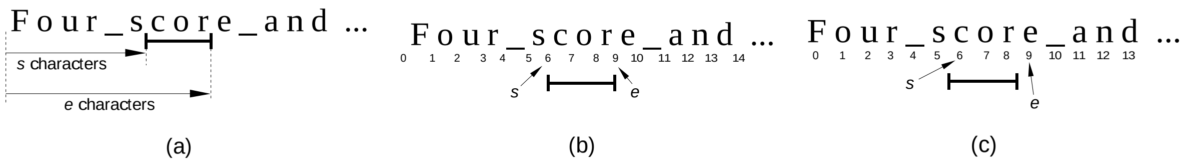


Figure 3.2: Equivalent definitions of a region $[s, e]$: (a) s and e are offsets from the start of the string; (b) s and e denote positions between characters, and $[s, e]$ is a closed interval; (c) s and e denote characters, and $[s, e]$ is a half-open interval.

by making it easier to write patterns (Chapter 6). It also helps machine learning agents by making it easier to define features and hypotheses (Chapters 9 and 10).

The next chapter will consider the question of implementation, showing that the general algebra can be implemented efficiently.

3.1 Regions

Given a string of length n, a *region* in the string is denoted by a pair $[s, e]$, where $0 \leq s \leq e \leq n$. The start offset s is the number of characters preceding the region's start point, and the end offset *e* is the number of characters preceding the region's end point (Figure 3.2(a)). The length of the region is $e - s$. If $s = e$, then the region has zero length, corresponding to a position between characters instead of a span of characters.

One can interpret a region $[s, e]$ in other ways that are equivalent to the definition above. If the positions between each character in the string are numbered 0, 1, 2, etc., then $[s, e]$ corresponds to a closed interval from position s to position e (Figure 3.2(b)). If the characters of the string are numbered instead, then $[s, e]$ corresponds to the half-open interval from character s to character e (Figure 3.2(c)). It should be clear that these definitions are all equivalent.

Another common way to describe a one-dimensional interval uses its start offset and *length,* instead of its end offset. Still another method, often seen in computational geometry, describes an interval by its *centroid* (the average of its start and end offsets) and length [HN83]. I prefer the definition given above, because it is easier to define relations like *before* and *after* if regions are described by end points rather than lengths.

3.2 Region Relations

The algebra is based on three fundamental binary relations among regions:

- $[s, e]$ *before* $[s', e']$ if and only if $e \leq s'$ and $s < s'$
- $[s, e]$ *overlaps-start* $[s', e']$ if and only if $s \leq s'$ and $e \leq e'$
- $[s, e]$ *contains* $[s', e']$ if and only if $s \leq s'$ and $e' \leq e$

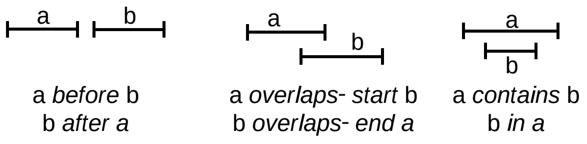


Figure 3.3: Fundamental region relations.

The *overlaps-start* and *contains* relations are reflexive, but *before* is not.[2] The *before* and *contains* relations are transitive, but *overlaps-start* is not. None of the relations are symmetric, so let us name their inverses as well:

- $[s, e]$ *after* $[s', e']$ if and only if $[s', e']$ *before* $[s, e]$
- $[s, e]$ *in* $[s', e']$ if and only if $[s', e']$ *contains* $[s, e]$
- $[s, e]$ *overlaps-end* $[s', e']$ if and only if $[s', e']$ *overlaps-start* $[s, e]$

All six relations are illustrated in Figure 3.3. Taken together, the six relations are complete in the following sense:

Claim 1. *For any two regions* $[s, e]$ *and* $[s', e']$*, there exists at least one of the six region relations,* op*, such that* $[s, e]$ *op* $[s', e']$.

The truth of this claim will be more obvious with the help of a two-dimensional, geometric interpretation of regions, described in the next section.

These relations between regions in a string are similar to the relations between time intervals defined by Allen [All83], which is natural because both domains involve intervals in one dimension. Allen's temporal relations use strict comparisons, however, so his *before* is defined as $[s, e]$ *before* $[s', e']$ if and only if $e < s'$. As a result, Allen must also include relations for equality and adjacency (e.g. $[s, e]$ *meets* $[s', e']$ if and only if $e = s'$), for a total of thirteen relations. In contrast, my formulation allows adjacency and equality to be expressed as Boolean combinations of the six fundamental relations, as will be seen below in Section 3.6.1. The choice between the two formulations is largely a matter of taste, however, since all of Allen's relations can be derived from my region relations and vice versa.

3.3 Region Space

Region space is a two-dimensional plane in which the x-coordinate is the start of a region and the y-coordinate is the end (Figure 3.4). A region $[s, e]$ corresponds to the point (s, e) in region space. Strictly speaking, region space does not occupy the entire real plane. Only points with integral coordinates correspond to regions, and only if they lie in the closed triangular area above the $45°$ line, where $0 \leq s \leq e \leq n$.

When the string is short, it is often convenient to write the characters of the string along the x and y axis, as is done in Figure 3.4. Since regions begin and end between characters, the characters

[2]In fact, *before* is defined so that no region can be *before* itself. This is the reason for the second inequality in the definition, $s < s'$, which prevents a zero-length region $[x, x]$ from being *before* itself.

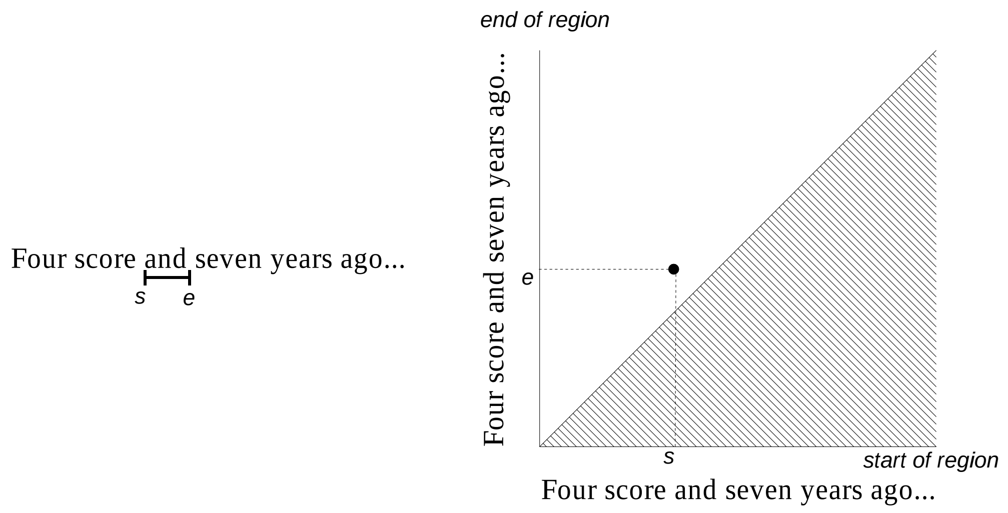

end of region
Four score and seven years ago...
Four score and seven years ago...
s
e
e
s
start of region
Four score and seven years ago...

Figure 3.4: Region space. A region $[s, e]$ corresponds to the point (s, e) in region space. Only integer-valued points on or above the $45°$ line are valid regions.

written along the axis actually label the intervals between region endpoints, rather than the region endpoints themselves.

The upper-left corner of region space corresponds to the region $[0, n]$, which spans the entire string. Points along the $45°$ diagonal represent zero-length regions in the string. The most interesting ones are the lower-left corner $[0, 0]$, which corresponds to the start of the string, and the upper right corner $[n, n]$, which is the end. The length of a region, $e - s$, is given by its y-distance above the $45°$ diagonal. Thus zero-length regions must lie on the diagonal itself, whereas the longest possible region, the entire string itself, must be the upper-left corner.

The region relations correspond to geometric relationships in region space. Suppose we hold some region $b = [b_s, b_e]$ fixed, and consider the set of all regions $a = [a_s, a_e]$ such that a *before* b. From the definition of *before*, it must be true that $a_e \leq b_s$. The set of regions a satisfying this constraint corresponds to the closed triangular area in region space shown in Figure 3.5(a). Parts (b) and (c) of the same figure show the corresponding closed rectangular areas for a *overlaps-start* b and a *contains* b, respectively.

Extending this analysis to include the three inverse relations *after*, *overlaps-end*, and *in* produces the region space “map” shown in Figure 3.6. The areas in this map are all closed. Each area includes its boundary, and adjacent areas intersect on their common boundaries.

The map completely covers region space, the triangular area above the diagonal line. Thus, for any region a, a must lie in some relation to b (or more than one, if a lies on a boundary between map areas). A similar map can be drawn for any choice of b. Points lying on the boundary of region space have degenerate maps, however (Figure 3.7). For example, Figure 3.7(a) shows the map for a zero-length region b lying on the diagonal. The areas for a *overlaps-start* b and *a overlaps-end b* have shrunk to lines, and the area for *a in b* has shrunk to the point b itself. The remaining parts of Figure 3.7 show the maps for other degenerate points in region space. In every case, the degenerate map still completely covers region space. Thus, a geometric argument suffices to establish Claim 1. For any pair of regions a, b, it must be the case that a lies in some area op in b’s region space map because the map covers region space, so *a op b*.

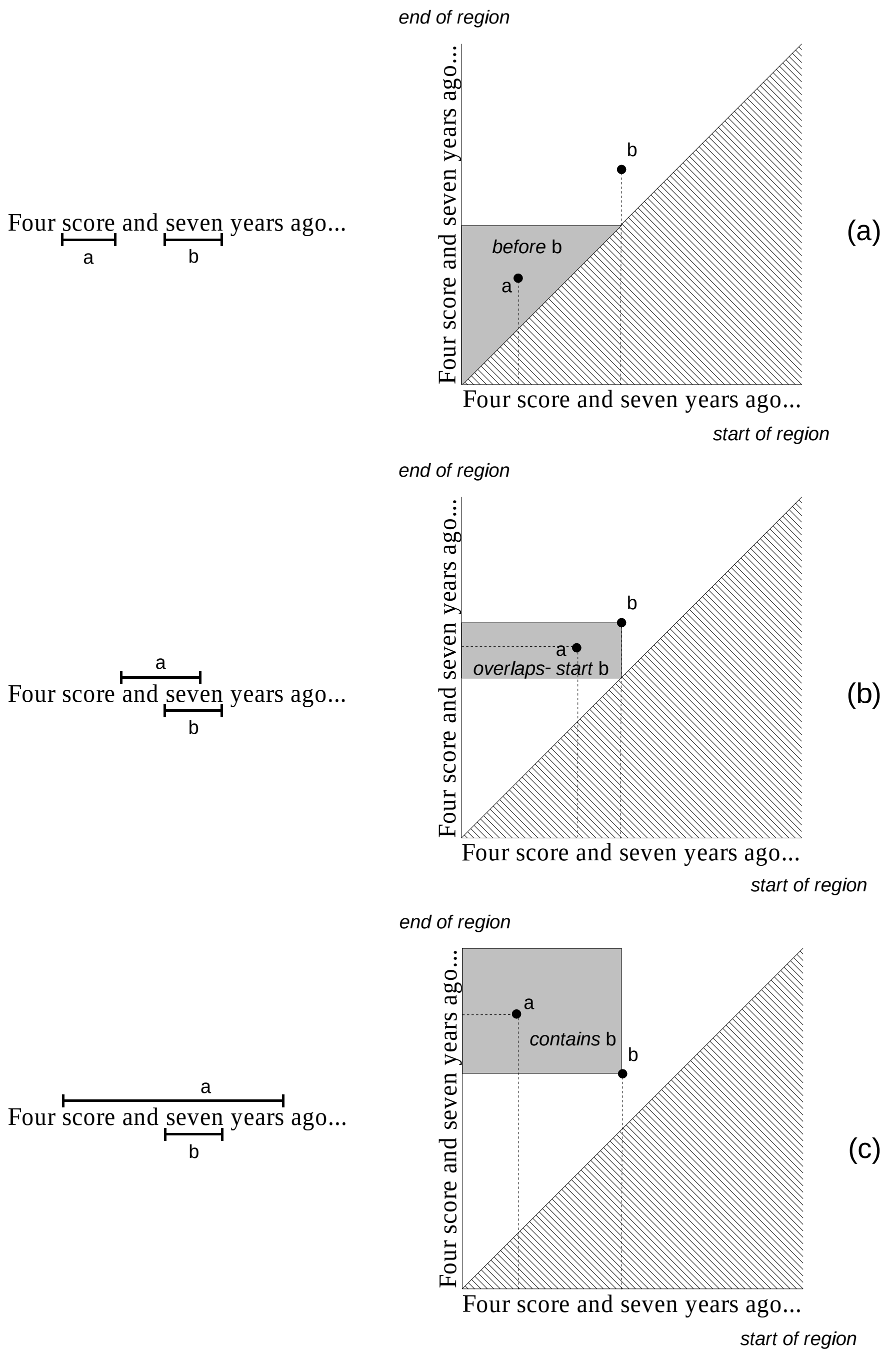


Figure 3.5: Areas of region space representing region relations relative to a fixed region b. The shaded rectangles represent all regions a satisfying a *before* b, a *overlaps-start* b, and a *contains* b, respectively.

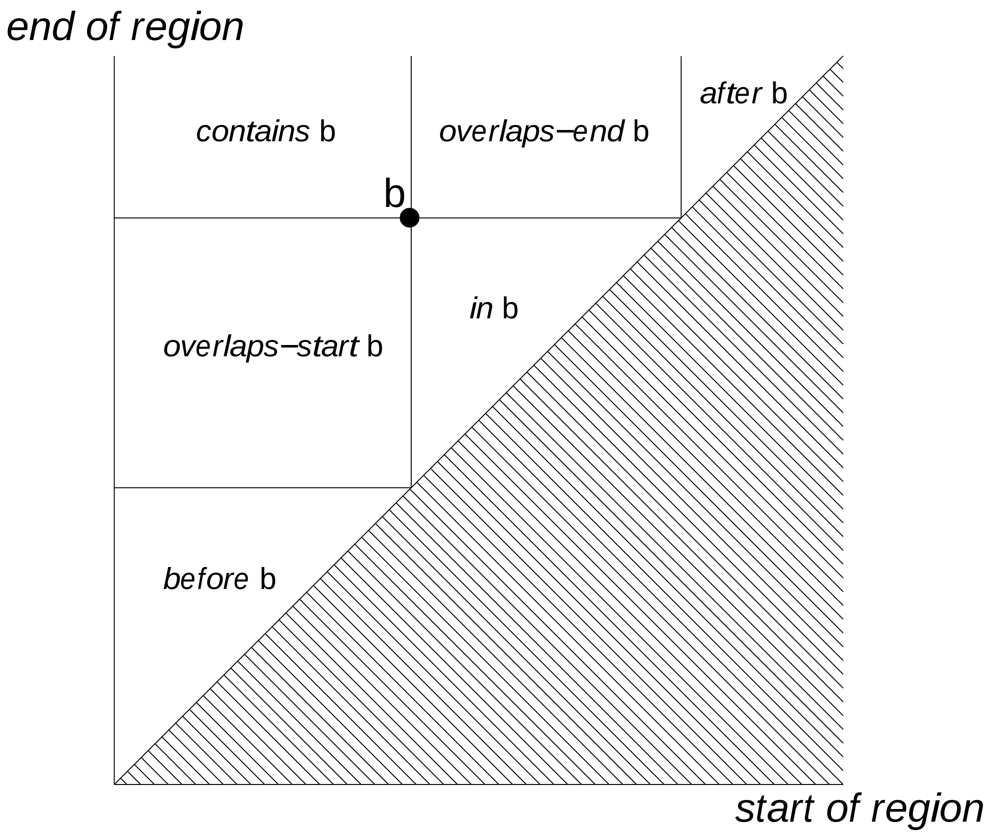


Figure 3.6: Region space map. “*op* b” labels the set of regions a such that a *op* b.

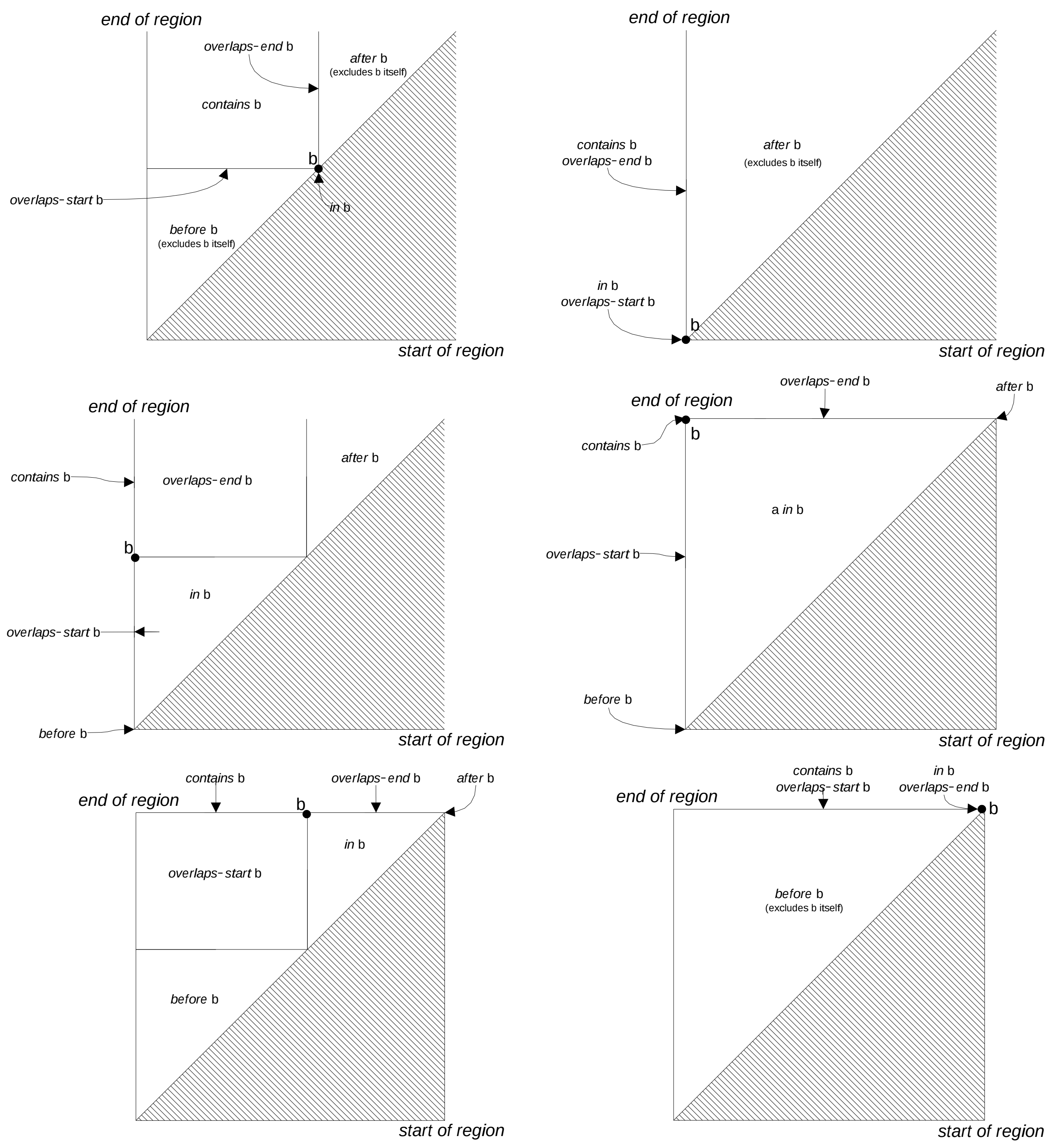


Figure 3.7: Degenerate region space maps, for regions b that lie on boundaries of region space.

Region space will be used repeatedly to illustrate the discussion of region algebra in this chapter and the next. Although it may take some effort to shift one's perspective from intervals along a one-dimensional string to points and areas in a two-dimensional plane, the rewards are worth it. A pictorial representation that shows every region as a clearly defined point makes it much easier to find boundary cases and ensure that a definition or algorithm handles them properly, as we did in Figure 3.7. Region space will also motivate the implementation of region algebra presented in the next chapter.

Transforming intervals in one dimension into points in two dimensions is not a new idea; the technique is well known in computational geometry [PS85] and spatial data structures [Sam90]. Rit [Rit86] presented a two-dimensional map of Allen's time interval relations that was very similar to the region space map shown in Figure 3.6, and Kulpa [Kul97] explored the maps that result when other interval representations are used, such as (centroid, length). However, this work is apparently the first that applies the transformation to text processing.

3.4 Region Set Types

The region relations can be used to define some important classes of region sets. If A denotes a set of regions, then:

- A is **flat** if and only if for every $a, b \in A$, at least one of the following holds: a *before* b; a *after* b; or $a = b$.
- A is **nested** if and only if for every $a, b \in A$, at least one of the following holds: a *before* b; a *after* b; a *in* b; a *contains* b; or $a = b$.
- A is **overlapping** if and only if for every $a, b \in A$, at least one of the following holds: a *before* b; a *after* b; a *overlaps-start* b; a *overlaps-end* b; or $a = b$.

From the definitions, it should be clear that A is flat if and only if A is both nested and overlapping. It also follows directly that if A is flat, nested, or overlapping, then all subsets of A are flat, nested, or overlapping, respectively.

An example of each kind of region set can be found in Figure 3.1. The words in Figure 3.1(a) are a flat set, the styles in Figure 3.1(b) are an overlapping set, and the expressions in Figure 3.1(c) are a nested set. Flat sets and nested sets are generally more common in text structure than overlapping sets.

Flat, nested, and overlapping region sets are particularly important because they are always linear in the length of the string, as the following theorems show.

Theorem 1. *If A is an overlapping region set in a string of length n, then $|A| \leq 2n + 1$.*

Proof. Consider A as a set of points in region space. Sweep a -45° line ($s + e = k$ for all integers k) across region space, from the lower left corner to the upper right corner. See Figure 3.8. When the sweep line intersects some point $a \in A$, then one part of the sweep line lies in *contains* a and the other part in *in* a, but the region space areas for *before* a, *after* a, *overlaps-start* a, and *overlaps-end* a touch the line only at a, so the sweep line can intersect no other points in A. Therefore every sweep position of the line intersects at most one region in A. Since there are $2n + 1$ sweep positions (k can range from 0 to $2n$), there are at most $2n + 1$ regions in A.

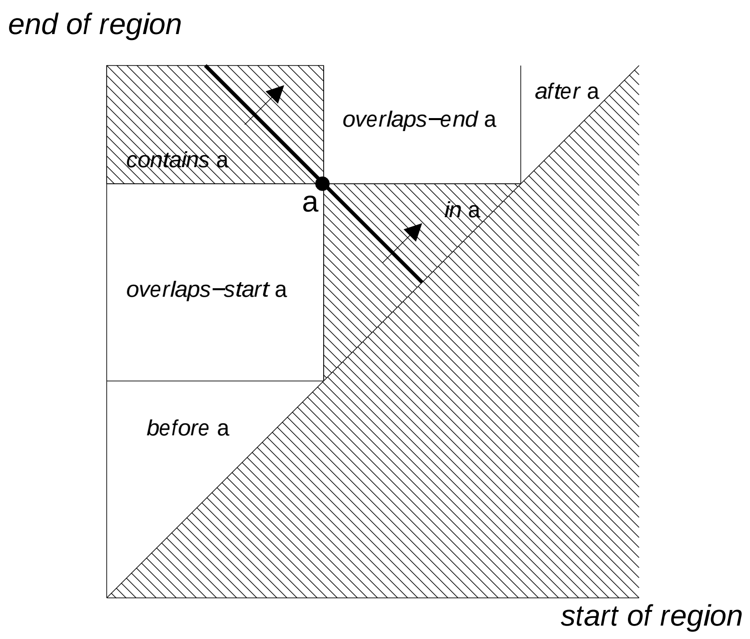


Figure 3.8: When a $-45°$ line is swept across an overlapping region set, it can intersect at most one region at every sweep position, because the rest of the line lies in the off-limits *in* and *contains* areas.

$\square$

Corollary 1. *If A is a flat set, then $|A| \leq 2n + 1$.*

This bound is tight, because the set of all length-0 regions $[k, k]$ and length-1 regions $[k, k + 1]$ is a flat set of size $2n + 1$.

Theorem 2. *If A is a nested region set in a string of length $n > 0$, then $|A| \leq 3n$.*

Proof. The following argument shows, by induction on n, that if A is a nested set with no zero-length regions, then $|A| \leq 2n - 1$. Since there are at most $n + 1$ zero-length regions in a string of length n, it will follow that an arbitrary nested region set can have at most $3n$ elements.

When $n = 1$, the only possible nonzero-length region is $[0, 1]$, so $|A| \leq 1$ as desired. Now suppose that all nested sets on strings of length $n' < n$ have at most $2n' - 1$ nonzero-length regions, and consider a nested set A with no zero-length regions on a string of length $n > 1$. There are two cases:

1. $[0, n] \notin A$. The following argument shows that there exists some k, $0 < k < n$, that partitions A such that every region in A is either *before* or *after* $[k, k]$. If A is empty, then this is trivially true. Otherwise, let r be the longest region in A (choosing arbitrarily if there's a tie). Some endpoint of r must lie strictly between 0 and n (otherwise $r = [0, n]$, which was previously excluded). Let k be this endpoint, and assume without loss of generality that k is the start of r. Because A is a nested set, for any region $a \in A$, at least one of the following holds:

   (a) a *before* r, which implies that a *before* $[k, k]$;

(b) a ***after*** r, which implies that a ***after*** $[k, k]$;

(c) a ***in*** r, which implies that a ***after*** $[k, k]$;

(d) a ***contains*** r, which implies that $a = r$ since there is no region in A longer than r;

(e) $a = r$, which implies that a ***after*** $[k, k]$.

Thus $[k, k]$ partitions A into $B = \{a \in A | a$ ***before*** $[k, k]\}$ and $C = \{a \in A | a$ ***after*** $[k, k]\}$. B is a nested set with no zero-length regions on a string of length $k < n$, and C is the same kind of set on a string of length $n-k < n$. Therefore $|A| = |B|+|C| \leq (2k-1)+(2(n-k)-1) = 2n-2$.

2. $[0, n] \in A$. Then $A - \{[0, n]\}$ satisfies Case 1, so $|A| \leq 2n - 1$.

□

The $3n$ bound for nested sets is also tight. Given a string of length $n = 2^k$, form a complete binary tree whose leaves are the n characters in the string. The $2n - 1$ nodes in the tree correspond to $2n - 1$ nested regions. Adding $n + 1$ zero-length regions gives a nested set with $3n$ elements.

Theorems 1 and 2 imply that any flat, nested, or overlapping region set on a string of length n has $O(n)$ regions. Other text-processing systems take advantage of this fact by restricting all operations to flat, nested, or overlapping region sets. For example, most regular expression libraries return only a flat set of matches — finding the leftmost, longest match to the regular expression, and resuming the search for more matches *after* each match. Therefore, even though the regular expression a^* matches every region in the string $aaaa$, most regular expression libraries return only the single region $aaaa$. Similarly, Proximal Nodes [NBY95] handles only nested sets, and GC-lists [CCB95] handles only overlapping sets.

Unlike these other pattern languages, the region algebra presented in the next section makes no special distinction among flat, nested, and overlapping sets. To achieve an efficient implementation of the region algebra, however, it will be important to optimize for these kinds of sets, because they are very common in practice.

3.5 Region Algebra

This section defines the region algebra. The algebra consists of a set of operators that take zero or more region sets as arguments and produce a region set as a result.

3.5.1 Set Operators

The algebra uses the familiar operators for intersection, union, and set difference:

$$\begin{aligned} A \cap B &\equiv \{r | r \in A \wedge r \in B\} \\ A \cup B &\equiv \{r | r \in A \vee r \in B\} \\ A - B &\equiv \{r | r \in A \wedge r \notin B\} \end{aligned}$$

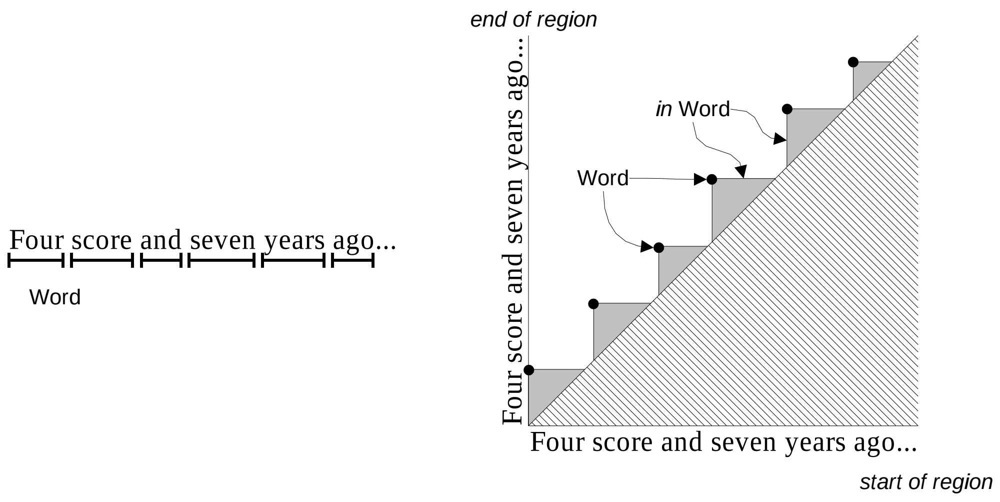

Four score and seven years ago...
Word
end of region
Four score and seven years ago...
in Word
Word
Four score and seven years ago...
start of region

Figure 3.9: The result of *in Word* depicted in region space. The isolated points make up the *Word* region set. The shaded areas are *in Word*.

In addition, the algebra defines operators for the set of all possible regions in the string and the empty set.

$$\begin{aligned} \Omega &\equiv \{[s,e] | 0 \leq s \leq e \leq n\} \\ \emptyset &\equiv \{\} \end{aligned}$$

Technically $\emptyset$ is redundant, since it could be expressed by $\Omega - \Omega$.

3.5.2 Relational Operators

The fundamental operators of the region algebra are unary operators that extend each of the fundamental region relations over a set:

$$\begin{aligned} \textit{before}\, B &\equiv \{a \in \Omega | \exists b \in B . a\, \textit{before}\, b\} \\ \textit{after}\, B &\equiv \{a \in \Omega | \exists b \in B . a\, \textit{after}\, b\} \\ \textit{overlaps-start}\, B &\equiv \{a \in \Omega | \exists b \in B . a\, \textit{overlaps-start}\, b\} \\ \textit{overlaps-end}\, B &\equiv \{a \in \Omega | \exists b \in B . a\, \textit{overlaps-end}\, b\} \\ \textit{contains}\, B &\equiv \{a \in \Omega | \exists b \in B . a\, \textit{contains}\, b\} \\ \textit{in}\, B &\equiv \{a \in \Omega | \exists b \in B . a\, \textit{in}\, b\} \end{aligned}$$

For example, the result of applying the operator *in* to a region set B is the set of all regions that lie *in* at least one region in B.

Region space offers a handy representation for showing the results of region algebra operators. In region space, a region set corresponds to a set of points in the plane. Figure 3.9 shows the result of *in Word*, where *Word* is the set of all word regions in the string. Notice that *in Word* is just the union, across all regions w $\in$*Word*, of the region space map areas corresponding to *in* $\{w\}$. Specifically, *in Word* is the union of the triangles below and to the right of each point in *Word*.

In other systems [CC97, CCB95, GT87, NBY95], operators like *in* are defined as binary operators:

$$A\,\textit{in}\,B \quad \equiv \quad \{a \in A \mid \exists b \in B . a\,\textit{in}\,b\}$$

In the region algebra, relational operators are defined as unary operators instead. Since the region algebra includes operators for set intersection and for the set of all regions Ω, the binary and unary definitions are equivalent:

$$\begin{aligned} A\,\textit{in}\,B &= A \cap (\,\textit{in}\,B) \\ \textit{in}\,B &= \Omega\,\textit{in}\,B \end{aligned}$$

In this dissertation, $A\,\textit{in}\,B$ will frequently appear as syntactic shorthand for $A \cap (\,\textit{in}\,B)$.

Unary operators have some advantages. First, they offer a compact description of a *predicate*, such as $(\,\textit{before}\,A \cup \,\textit{after}\,B) \cap \,\textit{in}\,C$, without mentioning the region set to which the predicate applies. Predicates will be very useful for representing features of regions for machine learning. Second, unary operators can eliminate redundancy in some expressions. Where $(A\,\textit{before}\,B) \cup (A\,\textit{after}\,C)$ must mention A twice, the unary equivalent $A \cap (\,\textit{before}\,B \cup \,\textit{after}\,C)$ mentions A only once. When A is a complicated expression, the notational savings can be significant. Third, unary operators will simplify the implementation description presented in Chapter 4.

The set difference operator $A - B$ also has a unary equivalent $\neg B \equiv \{a \in \Omega \mid a \notin B\}$. Hereafter, the unary version $\neg B$ will occasionally be used as shorthand for $\Omega - B$, but the advantages of unary complement over binary difference do not seem as substantial.

3.5.3 Iteration Operator

The last operator in the algebra iterates through a region set and applies an expression to each region in the set. The iteration operator is written $\textit{forall}\,(a : A) \,.\, f(a)$, where A is a region set, a is a variable of type region set that takes on the singleton value $\{r\}$ for every region r in A, and $f(a)$ is an expression in the region algebra with a as a free variable. Formally, *forall* is defined as:

$$\textit{forall}\,(a : A) \,.\, f(a) \equiv \bigcup_{r \in A} f(\{r\})$$

Many uses of the iteration operator will be seen in the examples in the next section.

3.6 Derived Operators

To illustrate how the region algebra can be used, we now define some higher-level pattern matching operators in terms of the basic algebra. The user-level pattern language described in Chapter 6 includes many of these operators as primitives.

3.6.1 Adjacency

The intersections of the fundamental region relations correspond to boundary cases in which region endpoints touch. For example, the relation *just-before* is the intersection of the relations *before* and *overlaps-start*. Some care must be taken when applying these intersections to region sets, however. We can't simply define *just-before* as

$$\textit{just-before}\ A \equiv\ \textit{before}\ A \cap\ \textit{overlaps-start}\ A \qquad \text{(incorrect)}$$

If A contains more than one region, this definition would allow regions that are *before* one region in A and *overlaps-start* a different region in A. For example, suppose the set A consists of the two underlined regions below:

$$\texttt{Super}\underline{\texttt{cali}}\texttt{fragi}\underline{\texttt{list}}\texttt{ic}$$

Using the definition above, *just-before* A would correctly include the region `Super` because it lies *before* and *overlaps-start* the underlined region `cali`. Likewise, *just-before* A would correctly include `Supercalifragi`, which is *before* and *overlaps-start* the underlined region `list`. Yet the definition above would also match `Supercal`, because it is *before* `list` and *overlaps-start* `cali`. `Supercal` is clearly not adjacent to either of the underlined regions. In order to represent adjacency properly, we need to constrain the definition so that *before* and *overlaps-start* are only intersected when they refer to the same region in A.

The correct definition uses *forall* to iterate through the regions in A, applying the intersection to one region at a time:

$$\textit{just-before}\ A \equiv\ \textit{forall}\ (a : A)\ .\ \textit{before}\ a \cap\ \textit{overlaps-start}\ a$$

The other adjacency operators are defined similarly:

$$\begin{aligned}
\textit{just-after}\ A &\equiv \textit{forall}\ (a : A)\ .\ \textit{after}\ a \cap\ \textit{overlaps-end}\ a \\
\textit{starts-contains}\ A &\equiv \textit{forall}\ (a : A)\ .\ \textit{contains}\ a \cap\ \textit{overlaps-end}\ a \\
\textit{ends-contains}\ A &\equiv \textit{forall}\ (a : A)\ .\ \textit{contains}\ a \cap\ \textit{overlaps-start}\ a \\
\textit{starts-in}\ A &\equiv \textit{forall}\ (a : A)\ .\ \textit{in}\ a \cap\ \textit{overlaps-start}\ a \\
\textit{ends-in}\ A &\equiv \textit{forall}\ (a : A)\ .\ \textit{in}\ a \cap\ \textit{overlaps-end}\ a
\end{aligned}$$

These relations correspond to the lines between adjacent areas in the region set map, as shown in Figure 3.10.

Another useful relation simply tests whether regions have the same start point or end point, regardless of containment:

$$\begin{aligned}
\textit{starts}\ A &\equiv \textit{starts-contains}\ A \cup\ \textit{starts-in}\ A \\
\textit{ends}\ A &\equiv \textit{ends-contains}\ A \cup\ \textit{ends-in}\ A
\end{aligned}$$

starts A is the set of all regions that start at the same place as some region in A, regardless of which region contains which. Note that it isn't necessary to use *forall* in this case, because we're taking the union of the relations, not the intersection.

Two more intersections are also of interest:

$$\begin{aligned}
\textit{start-of}\ A &\equiv \textit{forall}\ (a : A)\ .\ \textit{before}\ a \cap\ \textit{in}\ a \\
\textit{end-of}\ A &\equiv \textit{forall}\ (a : A)\ .\ \textit{after}\ a \cap\ \textit{in}\ a
\end{aligned}$$

These operators always produce zero-length regions.

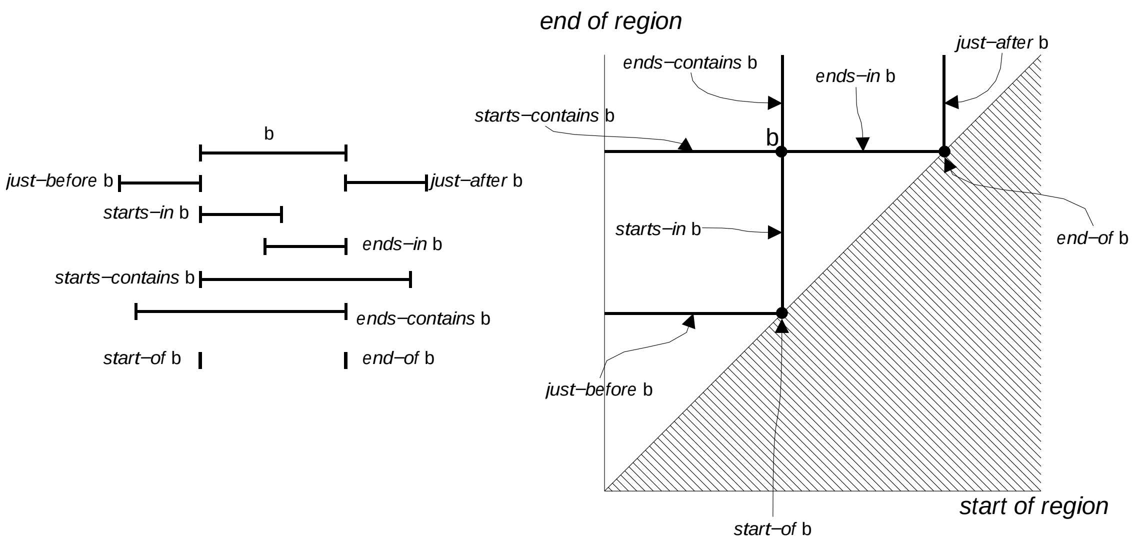


Figure 3.10: Adjacency operators.

3.6.2 Strictness

Excluding adjacency from each fundamental relation produces a set of *strict* operators:

$$\begin{aligned}
\textit{strict-before}\, A &\equiv \textit{forall}\,(a : A)\,.\ \textit{before}\, a - \textit{overlaps-start}\, a \\
\textit{strict-after}\, A &\equiv \textit{forall}\,(a : A)\,.\ \textit{after}\, a - \textit{overlaps-end}\, a \\
\textit{strict-in}\, A &\equiv \textit{forall}\,(a : A)\,.\ \textit{in}\, a - \textit{overlaps-start}\, a - \textit{overlaps-end}\, a \\
\textit{strict-contains}\, A &\equiv \textit{forall}\,(a : A)\,.\ \textit{contains}\, a - \textit{overlaps-start}\, a - \textit{overlaps-end}\, a \\
\textit{strict-overlaps-start}\, A &\equiv \textit{forall}\,(a : A)\,.\ \textit{overlaps-start}\, a - \textit{before}\, a - \textit{in}\, a - \textit{contains}\, a \\
\textit{strict-overlaps-end}\, A &\equiv \textit{forall}\,(a : A)\,.\ \textit{overlaps-end}\, a - \textit{after}\, a - \textit{in}\, a - \textit{contains}\, a
\end{aligned}$$

These relations are equivalent to using strict inequalities in the definitions of the fundamental relations.

3.6.3 Overlap

Two operators are useful for describing regions that overlap:

$$\begin{aligned}
\textit{touches}\, A &\equiv \textit{forall}\,(a : A)\,.\,(\Omega - \textit{strict-before}\, a) - \textit{strict-after}\, a \\
\textit{overlaps}\, A &\equiv \textit{forall}\,(a : A)\,.\,(\Omega - \textit{before}\, a) - \textit{after}\, a
\end{aligned}$$

Intuitively, a *touches* b if a and b touch anywhere, even if only at the endpoints. Thus, for example, a *just-before* b implies a *touches* b. In contrast, the stronger relation a *overlaps* b requires that a and b have at least one character in common, if a and b are nonzero-length regions. If either a or b has zero length, then either the regions must be identical or else one region must strictly contain the other. Figure 3.11 shows the areas for *touches* and *overlaps* in region space. *overlaps* plays an important role in many algorithms in a text-processing system. For example, a display rendering algorithm has to render every region that *overlaps* the region showing in the window.

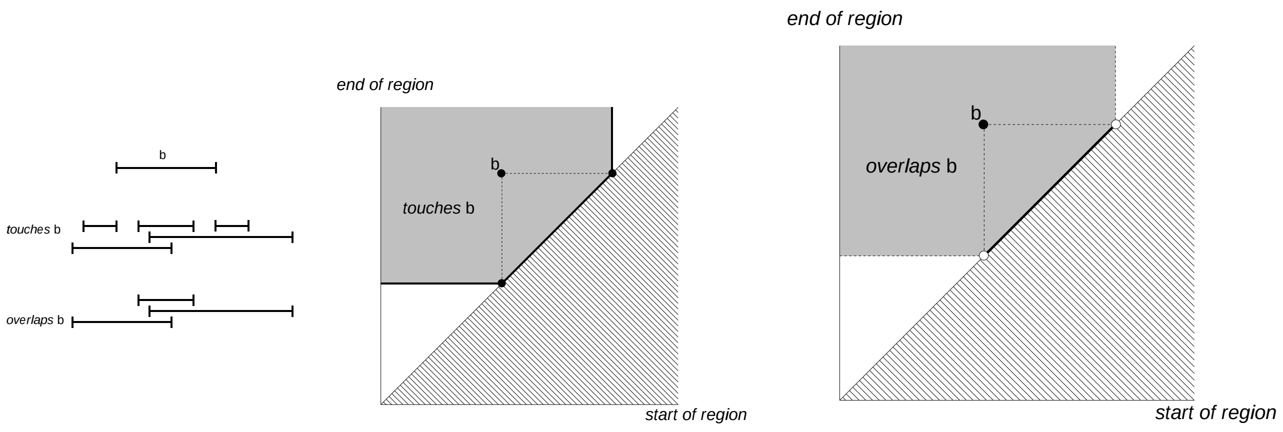


Figure 3.11: *touches* and *overlaps*.

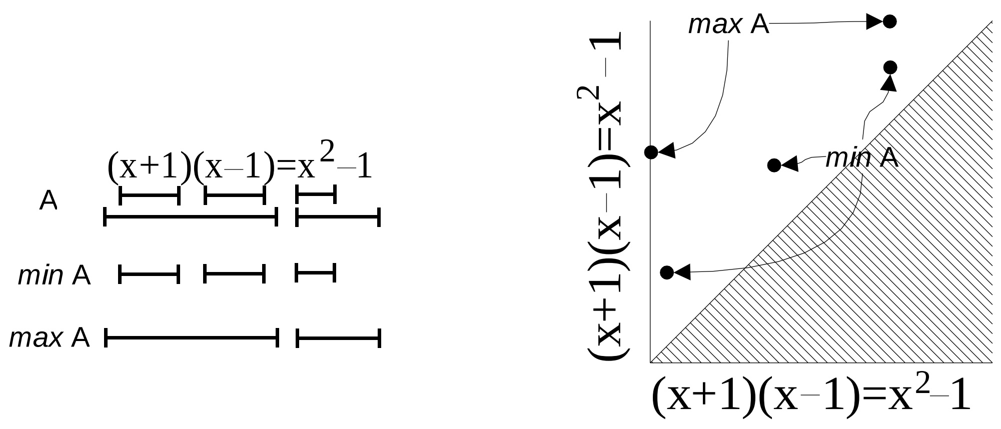


Figure 3.12: *min* and *max*.

3.6.4 Min and Max

Two operators are useful for finding the outermost and innermost regions in a region set:

$$\begin{aligned} \textit{max}\, A &\equiv \textit{forall}\,(a : A)\,.\, a - \textit{in}\,(A - a) \\ \textit{min}\, A &\equiv \textit{forall}\,(a : A)\,.\, a - \textit{contains}\,(A - a) \end{aligned}$$

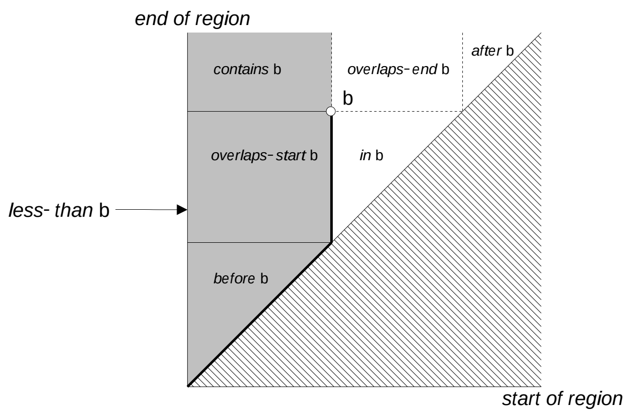
max A returns the regions in A that are not in any other region in A. Similarly, *min* A returns the regions in A that contain no other region in A. In region space, *min* A consists of the points in A closest to the diagonal, and *max* A consists of the points farthest from the diagonal (Figure 3.12).

Applied to the set of all regions Ω, *min* and *max* enable definitions of the region spanning the entire string and the set of zero-length regions:

$$\begin{aligned} \textit{entire-text} &\equiv \textit{max}\,\Omega \\ \textit{zero-length} &\equiv \textit{min}\,\Omega \end{aligned}$$

It is often useful to omit zero-length regions from a result, so the *nonzero* operator is explicitly defined for this purpose:

$$\textit{nonzero}\, A \equiv A - \textit{zero-length}$$



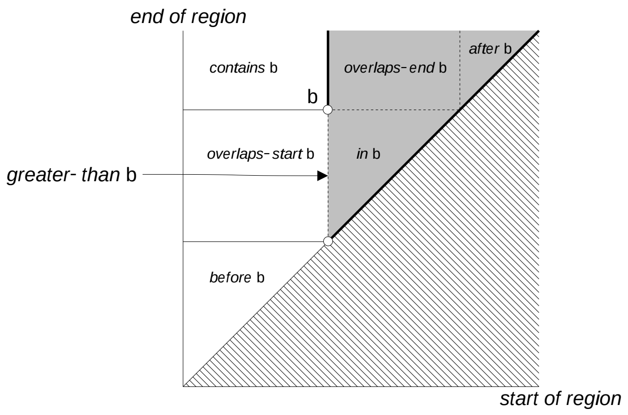


Figure 3.13: Region space areas for *less-than* and *greater-than*.

3.6.5 Counting

Counting is a common operation in structure manipulation. One may need to find the last line in a page, or the first argument of a function call.

Counting requires a total ordering on regions. We will use the conventional lexicographic ordering, so that $[s, e] < [s', e']$ if and only if either $s < s'$ or both $s = s'$ and $e < e'$. This ordering can be represented by region relations as follows:

$$\begin{aligned}\textit{less-than}\,A &\equiv \textit{forall}\,(a : A)\,.\,(\textit{before}\,a \cup \textit{overlaps-start}\,a \cup \textit{contains}\,a) - \textit{overlaps-end}\,a\\ \textit{greater-than}\,A &\equiv \textit{forall}\,(a : A)\,.\,(\textit{after}\,a \cup \textit{overlaps-end}\,a \cup \textit{in}\,a) - \textit{overlaps-start}\,a\end{aligned}$$

Figure 3.13 shows that these definitions produce the conventional lexicographic ordering, in the sense that *less-than* b corresponds to the set of points lexicographically less than point b, and *greater-than* b is the set of points lexicographically greater than b.

Now we can define *first* and *last*, a pair of operators that return the first and last region in a set by lexicographic ordering:

$$\begin{aligned}\textit{first}\,A &\equiv \textit{forall}\,(a : A)\,.\,a - \textit{greater-than}\,(A - a)\\ \textit{last}\,A &\equiv \textit{forall}\,(a : A)\,.\,a - \textit{less-than}\,(A - a)\end{aligned}$$

first A is the region that is not greater than any other region in A; *last* A region is the region that is not less than any other region in A. Unlike *min* and *max*, which may return more than one region, *first* and *last* return exactly one region (technically a singleton region set), as long as A is nonempty. If A is empty, then *first* and *last* return the empty set.

Using these operators, we can define a family of operators that return the nth region, counting either from the beginning or the end of the region set. The operators that count forward are defined recursively as follows:

$$\begin{aligned}\textit{nth}_{1}A &\equiv \textit{first}\,A\\ \textit{nth}_{n+1}A &\equiv \textit{first}\,(A\,\textit{greater-than}\ \textit{nth}_{n}A)\end{aligned}$$

Negative subscripts count backwards:

$$\begin{aligned}\textit{nth}_{-1}A &\equiv \textit{last}\,A\\ \textit{nth}_{-(n+1)}A &\equiv \textit{last}\,(A\,\textit{less-than}\ \textit{nth}_{-n}A)\end{aligned}$$

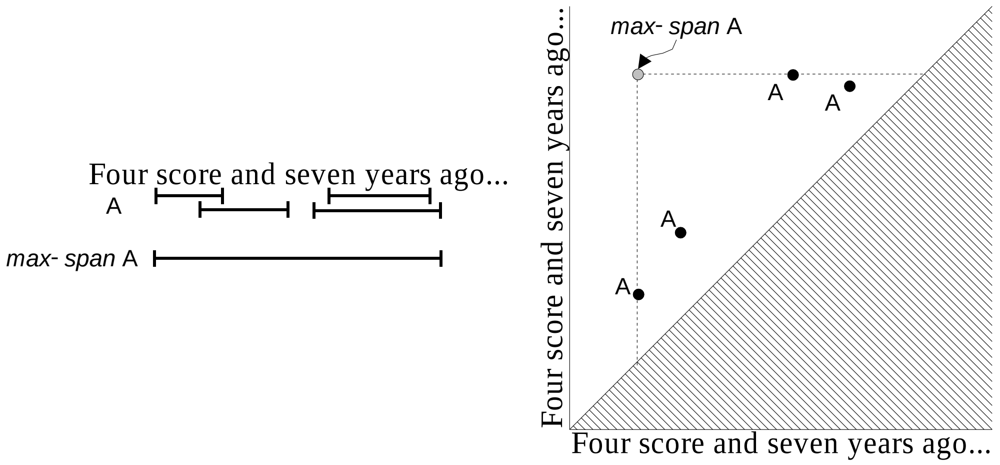

Four score and seven years ago...
A
max- span A
max- span A
A
A
A
A
Four score and seven years ago...
Four score and seven years ago...

Figure 3.14: *max-span* operator in region space.

Note that when $n > |A|$, both counting operators return the empty set.

Counting is often done relative to a context: the first word *after* a colon, or the last sentence *in* a paragraph. To represent context for counting, we can define a family of operators $\mathit{nth}_n\,\mathit{op}$, where *op* is a unary operator:

$$\mathit{nth}_n A\ \mathit{op}\ B \equiv \mathit{forall}\,(b : B)\ .\ \mathit{nth}_n(A\ \mathit{op}\ b)$$

For example, $\mathit{nth}_4\mathit{Word\ in\ Sentence}$ returns the fourth word in every sentence.

3.6.6 Span

The *span* of two regions a and b is the region *r* such that r *starts-contains* a and r *ends-contains* b:

$$A\ \mathit{span}\ B \equiv \ \mathit{starts\text{-}contains}\ A \cap \ \mathit{ends\text{-}contains}\ B$$

However, the span operator usually generates too many regions to be useful. The next few sections define more useful subsets of span.

The *max-span* operator returns the span of a set of regions with itself:

$$\mathit{max\text{-}span}\ A \equiv \ \mathit{max}\,(A\ \mathit{span}\ A)$$

max-span A returns the smallest region containing every region in A, if A is nonempty; otherwise it returns the empty set. In region space, *max-span* A corresponds to the upper left corner of the smallest bounding box containing A, as shown in Figure 3.14.

3.6.7 Concatenation

The *then* operator concatenates adjacent regions:

$$A\ \mathit{then}\ B \equiv \ \mathit{forall}\,(a : A)\ .\ \mathit{forall}\,(b : B\ \mathit{just\text{-}after}\ a)\ .\ a\ \mathit{span}\ b$$

This operator corresponds to the conventional definition of string concatenation. A region belongs to A *then* B if and only if it consists of a region in A concatenated with a region in B.

3.6.8 Delimiters

The *upto* operator returns the span of each region in A with the first region in B after it:

$$A \textit{ upto } B \equiv \textit{forall}\,(a : A)\,.\ \textit{forall}\,(b :\ \textit{first}\,(B \textit{ after } a))\,.\,a \textit{ span } b$$

For example, "/*" *upto* "*/" would match C comments.

The related operator *backto* spans each region in A with the last region in B preceding it:

$$A \textit{ backto } B \equiv \textit{forall}\,(a : A)\,.\ \textit{forall}\,(b :\ \textit{last}\,(B \textit{ before } a))\,.\,a \textit{ span } b$$

In most C programs, "*/" *backto* "/*" would match the same regions as "/*" *upto* "*/" . If some comment contained more than one occurrence of “/*”, which is permitted in C, then these expressions would return different region sets, because *backto* scans from the end delimiter.

Upto and *backto* may return overlapping sets, so neither operator is particularly useful when the start delimiter is identical to the end delimiter, as is the case for quotation marks. Suppose *QuoteMark* matches the four quotation marks in this string:

The word "zeitgeist" means "spirit of the time."

Then the expression *QuoteMark upto QuoteMark* actually matches three overlapping regions: *"zeitgeist"*, *" means "*, and *"spirit of the time."* The *backto* operator produces the same result.

Getting the region set we want, with just “zeitgeist” and “spirit of the time”, requires a left-to-right scan that alternates between starting delimiters and ending delimiters, never using the same quote mark twice. The counting operator $\textit{nth}_n$ can do such a scan, but only to a finite extent. To illustrate, suppose we had the operators *odd* and *even*, where *odd QuoteMark* returns the odd-numbered quote marks (first, third, fifth, etc.) and *even* returns the even-numbered ones. Then the expression *odd QuoteMark upto even QuoteMark* would produce the desired region set. Since *QuoteMark* is a flat set, *odd* (*QuoteMark upto QuoteMark*) would also work.

Unfortunately, defining odd and *even* in terms of $\textit{nth}_n$ requires an infinite union. *Odd* is the limit, as n goes to infinity, of the following recursive definition:

$$\begin{aligned} \textit{odd}_1 A &\equiv \textit{first}\, A \\ \textit{odd}_{n+2} &\equiv \textit{odd}_n A \cup \textit{nth}_{n+2} A \end{aligned}$$

In fact, it can be shown (Chapter 5) that the region algebra with literal string matching alone cannot generate the quoted region set we want, because a region algebra expression cannot count modulo a number. Since the hypothetical *odd* and *even* operators count modulo 2, they cannot be expressed by a (finite) region algebra expression.

LAPIS solves this problem with the pattern operator *fromto*, which is defined not by an algebra expression, but by an algorithm. The FROMTO procedure (Algorithm 3.1) takes two arguments: a region set of starting delimiters L, and a region set of ending delimiters R. The procedure iterates through L and R in left-to-right order, matching a delimiter from L with the closest following delimiter from R and returning the *span* of the two delimiters. The procedure then continues with the next L delimiter after the spanned region, so that the resulting region set is always flat.

The region algebra cannot represent balanced delimiters either, where starting delimiters are matched with ending delimiters to generate a hierarchy of nested regions. Balanced parentheses

Algorithm 3.1 FROMTO returns a flat region set by spanning from each region in L to the closest following region in R.

```
FROMTO($L, R$)
1  $C \leftarrow \emptyset$
2  $l \leftarrow$ first $L$
3  $r \leftarrow$ first $R$ after $l$
4  while $r \neq \emptyset$
5  do $C \leftarrow C \cup (l$ span $r)$
6       $l \leftarrow$ first $L$ after $r$
7       $r \leftarrow$ first $R$ after $l$
8  return $C$
```

are an example of this kind of delimiter. For balanced delimiters, LAPIS introduces the *balances* operator, which is defined by the BALANCE procedure (Algorithm 3.2). Like FROMTO, BALANCE takes a set of starting delimiters L and a set of ending delimiters R. The start delimiters in L are pushed onto the stack until an end delimiter from R is encountered. Line 10 guarantees that every end delimiter used by the procedure lies *after* its matching start delimiter, every start delimiter lies *after* its parent region's start delimiter, and every end delimiter lies *after* its children's end delimiters. As a result, the output of the entire procedure is a nested region set.

Algorithm 3.2 BALANCE returns a nested region set formed by matching opening delimiters from L with closing delimiters from R.

```
BALANCE($L, R$)
1   $C \leftarrow \emptyset$
2   $S \leftarrow$ new STACK()
3   $r \leftarrow$ first $(L \cup R)$
4   while $r \neq \emptyset$
5   do  if $r \in R$ and not EMPTY($S$)
6             then $q \leftarrow$ POP($S$)
7                  $C \leftarrow C \cup (q$ span $r)$
8           else if $r \in L$
9             then PUSH($S, r$)
10        $r \leftarrow$ first $(L \cup R)$ after $r$
11  return $C$
```

3.6.9 Hierarchies

Hierarchical structure is a particularly common type of text structure. Logical document structure, natural language phrase structure, and programming language syntax are all hierarchical. In most text-processing systems, hierarchical structure is represented by a *syntax tree* created by parsing the text with a grammar. In the region algebra, hierarchical structure is represented by a nested region set.

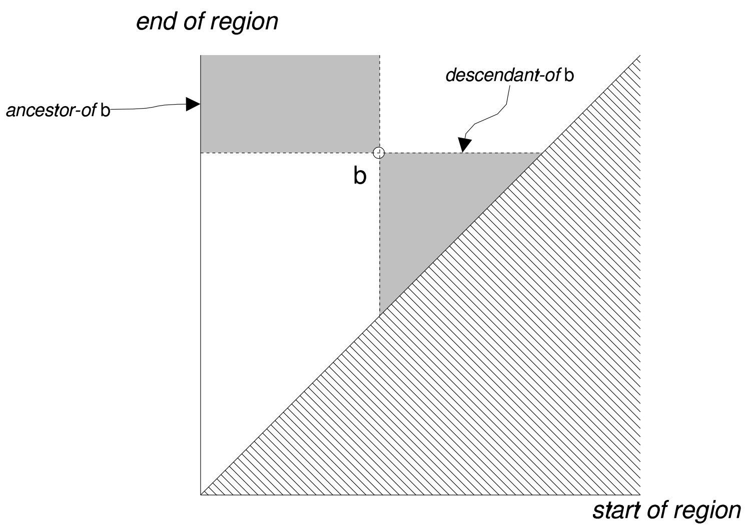


Figure 3.15: *descendant-of* and *ancestor-of*.

For capturing ancestry relations, neither *in* nor *strict-in* is quite right. For example, $a + b$ is a hierarchical structure of three expressions: a, b, and $a+b$. We would like to express the relationship that a and b are descendants of $a + b$. Both a and b are *in* $a + b$, but so is $a + b$ itself, so *in* does not describe the descendant relation. Yet neither a nor b is *strict-in* $a + b$, because the start of a (end of b) coincides with the corresponding point of $a + b$. This problem can be fixed by defining a new pair of operators:

$$\begin{aligned} \textit{descendant-of}\, B &\equiv \textit{forall}\,(b : B)\,.\,(\textit{in}\, b) - b \\ \textit{ancestor-of}\, B &\equiv \textit{forall}\,(b : B)\,.\,(\textit{contains}\, b) - b \end{aligned}$$

In region space, these operators correspond to *in* and *contains* with a corner removed, as shown in Figure 3.15. Note that *descendant-of* and *ancestor-of* do not specify the hierarchy of interest. They are defined as unary predicates, so that *ancestor-of* b matches *any* region that could be an ancestor of b. The hierarchy of interest is specified by intersection. For example, *Statement* $\cap$ *ancestor-of* b returns the statements that are ancestors of b, *Method* $\cap$ *ancestor-of* b returns the method definitions, and *Line* $\cap$ *ancestor-of* b returns the lines. Thus, each use of a hierarchy operator can refer to a different hierarchy. This is an important difference from other systems, which support one and only one syntax tree. Furthermore, b need not be an exact member of the hierarchy in question — it need not correspond exactly to a piece of Java syntax, for example, or to a line. A sloppy selection made by the user, or a partial literal string match, is a sufficient entry point into a hierarchy.

Applying *min* and *max* to the hierarchy operators, one obtains the parent, child, root, and leaf relationships:

$$\begin{aligned} A\, \textit{child-of}\, B &\equiv \textit{max}\,(A\, \textit{descendant-of}\, B) \\ A\, \textit{leaf-of}\, B &\equiv \textit{min}\,(A\, \textit{descendant-of}\, B) \\ A\, \textit{parent-of}\, B &\equiv \textit{min}\,(A\, \textit{ancestor-of}\, B) \\ A\, \textit{root-of}\, B &\equiv \textit{max}\,(A\, \textit{ancestor-of}\, B) \end{aligned}$$

Unlike *descendant-of* and *ancestor-of*, these operators must be binary. The hierarchy must be specified in order to find the *min* and *max* of ancestors and descendants.

3.6.10 Before/After with Restricted Range

One drawback of *before* and *after* as pattern-matching operators is that they match too many regions. *before* B matches anywhere in the string as long as it lies before at least one region in B. This means that all but the last region in B is irrelevant; *before* B is equivalent to *before last* B.

Frequently we want to restrict the range of the search to some region set C:

$$\begin{aligned} \textit{before-in}\,(B,C) &\equiv \textit{forall}\,(c:C)\,.\;\textit{forall}\,(b:B\;\textit{in}\,c)\,.\;\textit{before}\,b\cap\;\textit{in}\,c \\ \textit{after-in}\,(B,C) &\equiv \textit{forall}\,(c:C)\,.\;\textit{forall}\,(b:B\;\textit{in}\,c)\,.\;\textit{after}\,b\cap\;\textit{in}\,c \end{aligned}$$

before-in (B,C) matches any region that is before some B region but still in the same C region. For example, *before-in* ("@" , *EmailAddress*) would match any region preceding the “@” in an email address.

Other structured text query languages [NBY95, KM98] restrict *before* and *after* even further to return only the closest match before or after each region in B. This concept can be represented by applying *first* and *last*. For example, here is how the Proximal Nodes [NBY95] syntax and semantics for *before* and *after* would be expressed in the region algebra:

$$\begin{aligned} A\;\textit{before}\;B\,(C) &\equiv \textit{last}\,(A\;\textit{before-in}\,(B,C)) \\ A\;\textit{after}\;B\,(C) &\equiv \textit{first}\,(A\;\textit{after-in}\,(B,C)) \end{aligned}$$

3.6.11 Split

Awk [AKW88], Perl [WCS96] and other text-processing languages include a function for splitting a string into parts. In awk, for example, every line is split automatically into fields by a delimiter pattern, which is whitespace by default. Awk also provides the `split` function to split any string into pieces separated by a delimiter pattern. The standard C library includes `strtok`, which splits a string into pieces separated by one or more delimiter characters.

Splitting can be represented in the region algebra as follows. Suppose D is a set of delimiter regions. For convenience, we'll name the complement of *overlaps*:

$$\textit{separated-by}\;D \equiv \Omega - \;\textit{overlaps}\;D$$

separated-by D matches any region that doesn't overlap a delimiter in D. Figure 3.16 shows the region space areas. We can split the entire string around the delimiters D by applying *max* to the result of *separated-by*:

$$\textit{split}\;D \equiv \;\textit{max}\,(\,\textit{separated-by}\;D)$$

It is also useful to split another set of regions by the delimiters:

$$A\;\textit{split}\;D \equiv \;\textit{forall}\,(a:A)\,.\;\textit{max}\,(\,\textit{in}\;a\cap\;\textit{separated-by}\;D)$$

For example, awk's behavior of splitting lines into fields can be obtained with the expression *Line split Whitespace*, where Whitespace matches a contiguous run of space and tab characters.

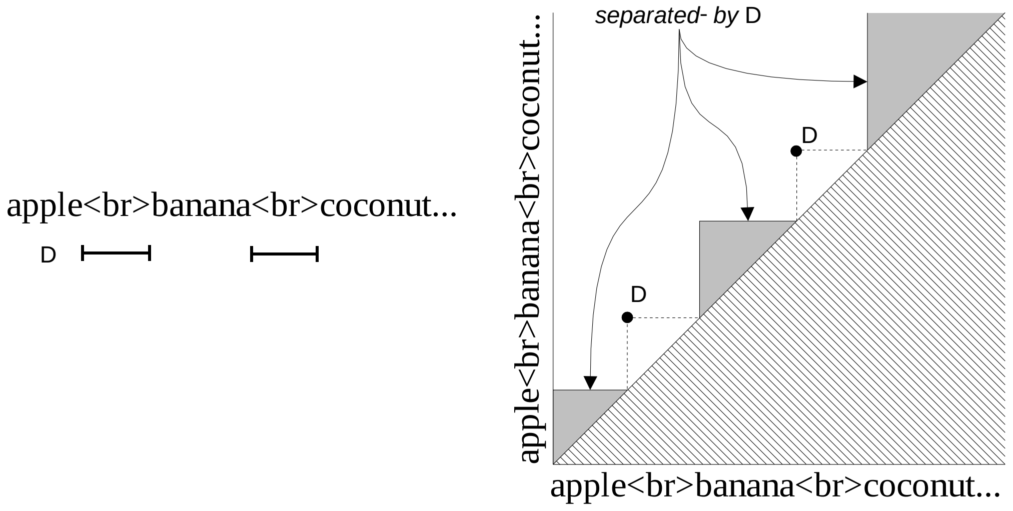

apple<br>banana<br>coconut...
D
separated- by D
D
D
apple<br>banana<br>coconut...
apple<br>banana<br>coconut...

Figure 3.16: *separated-by* D.

When two delimiters are adjacent, *split* returns the zero-length region between them. The `split` functions in Perl and awk follow the same semantics. The `strtok` function in ANSI C never returns empty strings, skipping over multiple adjacent delimiters instead. A simple way to describe this alternative semantics uses the *nonzero* operator to remove all zero-length regions from the result:

$$\textit{split-nonzero}\, D \equiv \textit{nonzero}\,(\textit{split}\, D)$$

A more subtle definition takes advantage of the fact that *overlaps* D is always a superset of D. Thus its complement, *separated-by* D, never includes an element of D, so *split* D $=$ *max* $($*separated-by* $D)$ cannot include a delimiter either. If we enlarge the set of delimiters to include the (zero-length) start points and end points of every delimiter in D, then *split* will not return any zero-length regions. Applying the *in* operator is a simple way to do this, giving the equivalent definition

$$\textit{split-nonzero}\, D = \textit{split in}\, D$$

Figure 3.17 shows how *split* and *split-nonzero* differ.

3.6.12 Flatten

Some text manipulations require a flat region set. For example, deleting or sorting a set of regions in a string is complicated if regions can nest or overlap. For such tasks, it makes sense to convert the set into a flat region set first.

Nested region sets can be flattened by applying *max* or *min*. The former operator returns the roots of the hierarchy, and the latter returns the leaves. Intermediate levels can be obtained by applying *child-of* or *parent-of*. For example, *max* $(A$ *child-of* $A)$ returns all the children of the roots.

An arbitrary region set can be flattened by replacing each group of overlapping regions with the region spanning the group. This operator, *flatten*, can be described with two applications of *split*. *split* A matches regions that span the gaps of A — characters that are not covered by any region in A. So *split split* A matches the regions between the gaps. The only problem with *split split* A

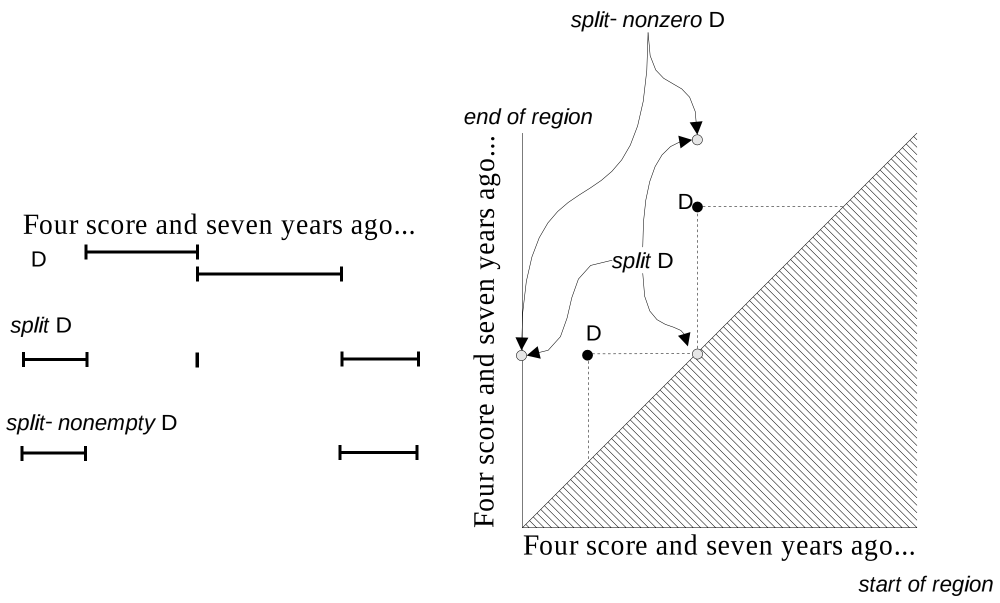

Four score and seven years ago...
D
split D
split- nonempty D
split- nonzero D
end of region
start of region
Four score and seven years ago...
Four score and seven years ago...
D
D
split D

Figure 3.17: *split* and *split-nonzero*.

is that it always includes the zero-length regions that start and end the entire string. We have to adjust the definition of *flatten* to exclude those two points if A lacks them:

$$\textit{flatten}\, A \equiv (\textit{split split}\, A) - ((\textit{start-of entire-text} \cup \textit{end-of entire-text}) - A)$$

If two regions in A are merely adjacent, not overlapping, then *flatten* A does not combine them, because *split* A leaves a zero-length region between the two regions that causes *split split* A to return the two regions separately. It is sometimes useful to coalesce adjacent regions as well as overlapping regions. To do that, we just replace the innermost application of *split* with *split-nonempty*:

$$\textit{melt}\, A \equiv (\textit{split split-nonempty}\, A) - ((\textit{start-of entire-text} \cup \textit{end-of entire-text}) - A)$$

The difference between *flatten* and *melt* can be seen in Figure 3.18.

3.6.13 Ignoring Background

It is often useful to weaken the adjacency test in such a way that two regions are considered adjacent as long as they are only separated by irrelevant characters. Call these irrelevant characters the *background*. The appropriate background depends on the pattern and the string to which it is applied. When searching for phrases in natural language, for example, a good background would be whitespace and punctuation. When searching for statements or expressions in source code, the background might be whitespace and comments. When searching an HTML or XML document, a good background might be whitespace and tags. Background can make many pattern expressions simpler.

The adjacency operators described in Section 3.6.1 can be extended to include a background parameter, which is just a set of regions W. Recalling the definition of *just-before* and *just-after*,

$$\begin{aligned}\textit{just-before}\, A &\equiv \textit{forall}\,(a : A)\,.\ \textit{before}\, a \cap \textit{overlaps-start}\, a\\ \textit{just-after}\, A &\equiv \textit{forall}\,(a : A)\,.\ \textit{after}\, a \cap \textit{overlaps-start}\, a\end{aligned}$$

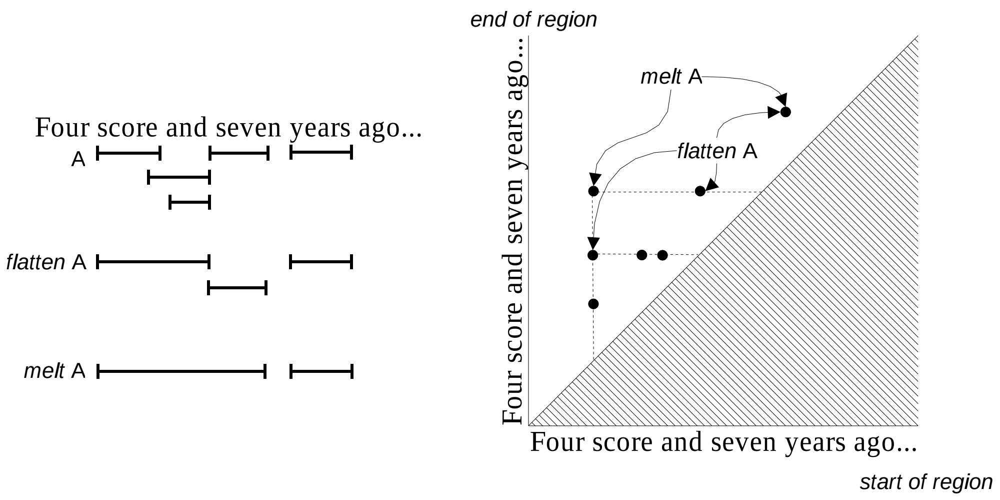

Four score and seven years ago...
A
flatten A
melt A
end of region
start of region
Four score and seven years ago...
Four score and seven years ago...
melt A
flatten A

Figure 3.18: *flatten* creates a flat set by combining overlapping regions into a single region. *melt* combines not only overlapping regions but also adjacent regions.

we can extend these definitions to ignore at most one intervening occurrence of a region from W:

$$\begin{aligned} \textit{just-before}_W A &\equiv \textit{forall}\,(a:A)\,.\ \textit{before}\,a \cap (\textit{overlaps-start}\,a \cup \textit{overlaps-start}\,(W\ \textit{overlaps-start}\,a)) \\ \textit{just-after}_W A &\equiv \textit{forall}\,(a:A)\,.\ \textit{after}\,a \cap (\textit{overlaps-start}\,a \cup \textit{overlaps-start}\,(W\ \textit{overlaps-start}\,a)) \end{aligned}$$

To allow an arbitrary number of intervening or partial background regions, we can melt the background regions (i.e. use *melt* W as the background parameter). Figure 3.19 illustrates *just-before*$_W$ and *just-after*$_W$.

Similar background-sensitive definitions can be given for other adjacency operators:

$$\begin{aligned} \textit{starts-contains}_W A &\equiv \textit{forall}\,(a:A)\,.\ \textit{contains}\,a \cap (\textit{overlaps-end}\,a \cup \textit{overlaps-end}\,(W\ \textit{overlaps-start}\,a)) \\ \textit{ends-contains}_W A &\equiv \textit{forall}\,(a:A)\,.\ \textit{contains}\,a \cap (\textit{overlaps-start}\,a \cup \textit{overlaps-start}\,(W\ \textit{overlaps-end}\,a)) \\ \textit{starts-in}_W A &\equiv \textit{forall}\,(a:A)\,.\ \textit{in}\,a \cap (\textit{overlaps-start}\,a \cup \textit{overlaps-end}\,(W\ \textit{overlaps-start}\,a)) \\ \textit{ends-in}_W A &\equiv \textit{forall}\,(a:A)\,.\ \textit{in}\,a \cap (\textit{overlaps-end}\,a \cup \textit{overlaps-start}\,(W\ \textit{overlaps-end}\,a)) \end{aligned}$$

Operators defined in terms of the adjacency operators can also be made background-sensitive:

$$\begin{aligned} \textit{starts}_W A &\equiv \textit{starts-contains}_W A \cup \textit{starts-in}_W A \\ \textit{ends}_W A &\equiv \textit{ends-contains}_W A \cup \textit{ends-in}_W A \end{aligned}$$

Background is also helpful in operators that depend on adjacency, such as *then*. Here is a background-sensitive version of *then*:

$$A\ \textit{then}_W B \equiv \textit{forall}\,(a:A)\,.\ \textit{forall}\,(b:B\ \textit{just-after}_W a)\,.\ a\ \textit{span}\ b$$

Background can also be used to weaken set intersection, so that two regions are considered equal if one can be obtained from the other by trimming or adding background characters. We start by defining *equals* $A \equiv$ *forall* $(a:A)$. *starts* $a \cap$ *ends* a, which is just the identity function.

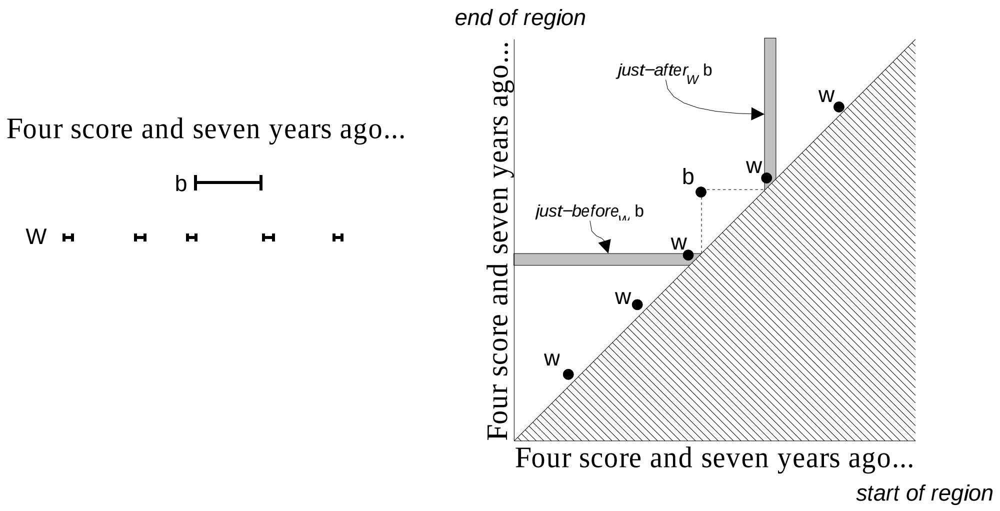

end of region
Four score and seven years ago...
b
W
just–after$_W$ b
just–before$_W$ b
w
w
b
w
w
w
Four score and seven years ago...
Four score and seven years ago...
start of region

Figure 3.19: *just-before*$_W$ and *just-after*$_W$.

But using the background-sensitive versions of *starts* and *ends* produces a background-sensitive equality test:

$$\textit{equals}_W A \equiv \textit{forall}\,(a : A)\,.\ \textit{starts}_W a \cap \textit{ends}_W a$$

Figure 3.20 shows the regions produced by *equals*$_W A$.

A pattern would use A *equals*$_W B$ instead of $A \cap B$ in order to allow weak equality between regions in A and B. Weak equality is particularly useful for comparing regions from different hierarchies, such as physical and logical layout, or regions produced by different parsers. Weak equality also improves usability. For example, weak equality allows a user's selection to match a piece of Java syntax even if it does not start and end at the precise character positions identified by the Java parser, as long as the differences are limited to background characters.

3.6.14 Trim

Many string manipulation libraries include functions that remove whitespace from the start or end of a string. One example is the `trim` method in `java.lang.String`, which trims whitespace from both ends of the string. Perl also includes `chomp`, which removes a linebreak from the end of a string. We can generalize these routines with a region algebra expression which trims the ends of a set of regions.

Trim-left and *trim-right* are binary operators that trim the left or right end of each region in A to remove at most one overlapping occurrence of a "whitespace" region from W. To trim an arbitrary number of whitespace regions, just use a melted region set for W. The trim operators are defined as follows:

$$\begin{aligned} A\,\textit{trim-right}\,W &\equiv \textit{forall}\,(a : A)\,.\,(\textit{start-of}\,a)\ \textit{upto}\ (\textit{end-of}\,a \cup W\ \textit{overlaps-end}\,a) \\ A\,\textit{trim-left}\,W &\equiv \textit{forall}\,(a : A)\,.\,(\textit{end-of}\,a)\ \textit{backto}\ (\textit{start-of}\,a \cup W\ \textit{overlaps-start}\,a) \end{aligned}$$

For example, the effect of Perl's `chomp` can be obtained by *Line trim-right Linebreak*.

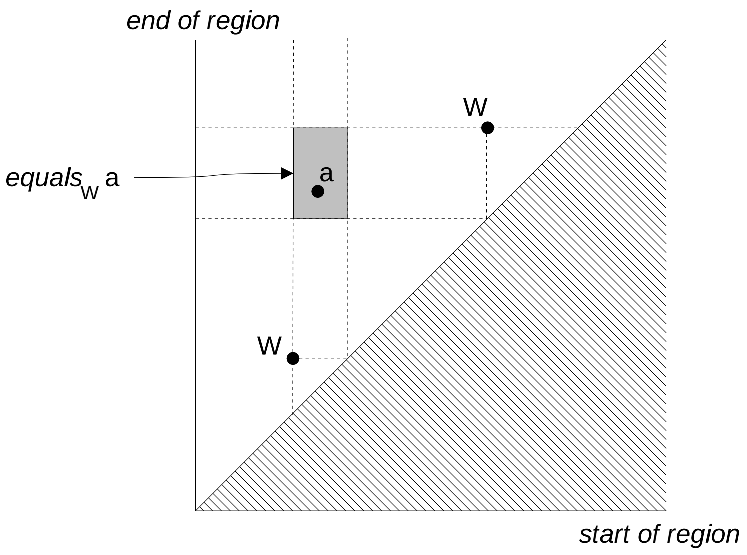


Figure 3.20: $equals_W a$ matches any region that can be obtained from a by adding or removing characters in W.

Trim applies both *trim-left* and *trim-right:*

$$A\,\textit{trim}\,W \equiv (A\,\textit{trim-right}\,W)\,\textit{trim-left}\,W$$

So the `trim` method of `java.lang.String` can be obtained by *trim Whitespace*.