The arithmetic of shapes

R Krishnamurti
Centre for Configurational Studies, The Open University, Milton Keynes, MK7 6AA, England
Received 28 November 1980, in revised form 15 December 1980

Abstract. Algorithms for the Boolean operations and relations on shapes and labelled shapes are presented.

This paper has two parts. The first part accomplishes two objectives. First, an efficient and uniform representation for shapes is presented, which is based on a linear order on the maximal lines of a shape. Second, simple and efficient algorithms for the Boolean operations on shapes (shape union, difference, and intersection) and the Boolean relations on shapes (subshape and shape equality) are presented. The second part deals with the computational aspects involved in performing the Boolean operations and relations on shapes and labelled shapes. An algorithm for the efficient decomposition of the Boolean operations and relations is presented. Last, the data structures required to implement the shape algorithms are described.

The relevant definitions and notations upon which this paper is based are given in Stiny (1980). Each shape is assumed to be initially described by a set of maximal lines; each labelled shape is given by a shape and an associated set of labelled points.

Rational shapes
A restriction is introduced to limit the class of shapes dealt with in this paper. This restriction—albeit a practical one—is necessitated by the fact that algorithms are defined with respect to some form of computing machine. In a random access machine with limited memory a real number is represented by a finite approximation which is determined by the word size of memory. This makes for inexact arithmetic. Moreover, it is usual for shapes to be drawn on some kind of graphics device such as a visual display unit or a digital plotter. On these devices only a limited number of points can be addressed, and the location of each point is given by an integral multiple or a pair of integral multiples of a unit of measurement. Since correct algorithms require exact arithmetic it is convenient to consider just those shapes which can be so described. Therefore, attention will be restricted to shapes which are, in the mathematical sense, rational. The following definition makes the notion of a 'rational shape' precise.

Definition: A point $p$ is rational if and only if each of its coordinates $x_1(p), \ldots, x_d(p)$, $d \geq 2$, can be expressed as the ratio of two integers. A labelled point $p: A$ is rational if and only if $p$ is rational. A line $l$, $l = \{p_1, p_2\}$, is rational if and only if its end points, $p_1$ and $p_2$, are rational. A shape $s$ is rational if and only if each of its maximal lines is rational. A labelled shape $\sigma$, $\sigma = (s, p)$, is rational if and only if $s$ is rational and every labelled point in point set $P$ is rational.

The ratio of two integers, $r_n/r_d$, may be expressed as the ordered pair, $(r_n, r_d)$, which, in turn, may be described by its unique primitive form. A pair of integers, $(r_n, r_d)$ is primitive if and only if the following conditions are satisfied:
(a) $r_n$ and $r_d$ are integers,
(b) \( r_d > 0 \),
(c) \( r_a \) and \( r_d \) are relatively prime—that is, there is no positive integer, say \( k \neq 1 \), such that \( k \) divides both \( r_a \) and \( r_d \).

When \( r_a < 0 \), the primitive is said to be negative.

Examples of primitives are now given. An integer \( n \) is described by the primitive \((n, 1)\); infinity \( \infty \) is described by the primitive \((1, 0)\); zero \( 0 \) is described by the primitive \((0, 1)\). A rational number \( r, r = r_a/r_d \), can always be reduced to its primitive form by applying Euclid’s greatest common denominator (gcd) algorithm (see for instance, Aho et al., 1974, pages 300-302). The following procedure outlines the steps involved in determining the primitive form for the ratio of two integers, \( r = r_a/r_d \), \( r_d > 0 \). Let \( a = \gcd(|r_a|, |r_d|) \), where, for any integer \( q \), \(|q| \) denotes the absolute value of \( q \). Let \( b = |r_d|/r_d \). Then, \( r \) is described by the primitive \((br_a)/a, |r_d|/a)\).

Primitives allow one to compare two numbers for equality. Let \( r^* = (r_1^*, r_2^*) \) denote the primitive of the number \( r \) where \( r \) may be rational, integral, or infinity. Two numbers \( r_1 \) and \( r_2 \) are equivalent if and only if their primitives \( r_1^* \) and \( r_2^* \) are equal. That is, \( r_1^* = r_2^* \) and \( r_1 = r_2 \).

Arithmetic computation involving primitives can be conveniently described by the following algorithmic notation. The expression:

\[
\text{variable } \leftarrow_p \text{ (expression)}
\]

signifies that the ‘variable’ is assigned the primitive form of the number that results from the ‘expression’. For example, let \( u, v, w \) be primitives. Then

\[
w \leftarrow_p (u + v)
\]

is equivalent to the steps:

\[
\begin{align*}
\text{set } w'_a &= u_a v_a + u_v v_u, \\
\text{set } w'_d &= u_d v_d.
\end{align*}
\]

\text{define } w' \text{ to be the ratio of integers, } w'_a/w'_d,

\text{assign to } w \text{ the primitive form of } w' \text{ by means of the procedure outlined above.}

Part 1
A representation for shapes
A good shape algorithm requires a good internal representation for shapes. It is widely accepted that by a good algorithm is meant one which has computational time-complexities and space-complexities (Aho et al., 1974, pages 12-14) which are polynomials of its input size. For shape algorithms, the inputs are essentially the maximal lines and the labelled points of the labelled shapes to which they apply. How these labelled shapes are represented, in turn, depends upon the type and nature of the computational steps involved in performing the shape operations and relations. It is clear that shape algorithms must satisfy the following two computational requirements:

1. There is an effective mechanism or method for determining whether or not two lines are colinear.
2. There is an effective mechanism or method for determining whether or not a point is coincident with a line.

Once these requirements can be satisfied it is fairly straightforward to determine whether or not two colinear lines share a common line.

A simple representation for the maximal lines of a shape will completely satisfy these requirements. The representation for two-dimensional rational labelled shapes is demonstrated, and the extensions to \( d \)-dimensional, \( d \geq 3 \), rational labelled shapes will be apparent.
**Line descriptors**

Consider an infinite line drawn in a cartesian coordinate system. Its equation can be written as

\[
\begin{align*}
y &= x(\text{slope}) + y(\text{intercept}), \quad \text{if line is nonvertical}, \\
x &= x(\text{intercept}), \quad \text{if line is vertical}.
\end{align*}
\]

Every maximal line may be viewed as a finite line segment on some infinite line whose equation takes on one of the above forms. Consequently, any two maximal lines which correspond to finite line segments on the same infinite line are **collinear**. Hence, every maximal line may be associated with a **line descriptor**, \( \psi \), which is the ordered pair given by \( \psi = \langle \mu, \nu \rangle \), where \( \mu \) is the slope, and \( \nu \) is the y-intercept if the line is nonvertical, and the x-intercept if the line is vertical. It is easy to show that for a rational line, its line descriptor may be represented by an ordered pair of primitives. Collinear lines have identical line descriptors and noncollinear lines do not.

**Coincident points and shared lines**

Let \( J \) and \( K \) be ordered pairs of primitives, where \( J = \langle j_1, j_2 \rangle \), and \( K = \langle k_1, k_2 \rangle \). Then, \( J \) is **less than** \( K \), denoted by \( J < K \), if and only if either \( j_1 < k_1 \) or \( j_1 = k_1 \) and \( j_2 < k_2 \). Here, \(<\) is an order relation.

The ordering, \(<\), on the points introduces an orientation on the maximal lines in the following manner. One may assume, without loss of generality, that \( l = \langle p_1, p_2 \rangle \) is a line the end points of which satisfy \( p_1 < p_2 \). Then, \( l \) is represented as the ordered pair given by \( l = \langle p_1, p_2 \rangle \). The endpoints \( p_1 \) and \( p_2 \) are, respectively, referred to as the **tail** and **head** of \( l \). For any line \( l \), tail of \( l \) < head of \( l \).

It follows, therefore, that a point which is coincident with a line must lie between the tail and head of the line. In other words, a point \( p \) is **coincident** with the line given by \( l = \langle p_1, p_2 \rangle \) if and only if either \( p_1 = p \), or \( p_2 = p \), or \( p \) satisfies (1) the equation of the line; and (2) the inequality \( p_1 < p < p_2 \). Since, for the shape algorithms considered in this paper, only collinear lines are compared, condition (1) is always satisfied.

The ordering of the points allows one to compare two collinear lines for overlap. This comparison, as will be seen, is important to the effectiveness of the shape algorithms. Two collinear lines \( l_1 \) and \( l_2 \) are said to **share** a common line if and only if tail of \( l_1 \) < head of \( l_2 \) and tail of \( l_2 \) < head of \( l_1 \). Their common line \( \bar{l} \), \( \bar{l} = \langle \bar{p}_1, \bar{p}_2 \rangle \), is given by \( \bar{p}_2 \), which is the **maximum** of the tails of \( l_1 \) and \( l_2 \), and \( \bar{p}_1 \) which is the **minimum** of the heads of \( l_1 \) and \( l_2 \).

Figure 1 presents examples of pairs of collinear lines, \( l_1 \) and \( l_2 \), some of which overlap, in which case their common line is indicated by a thick line. The end points of \( l_1 \) and \( l_2 \) are, respectively, denoted by \( \circ \) and \( \bullet \). End points common to \( l_1 \) and \( l_2 \) are denoted by \( \ast \). For each pair of collinear lines the conditions on their tails and heads are stated.

\[
\begin{align*}
tail(l_1) &< head(l_1) \quad \text{or} \quad tail(l_2) > head(l_2) \quad \text{or} \quad tail(l_2) < head(l_1) \quad \text{or} \quad tail(l_1) > head(l_2) \\
tail(l_1) &< head(l_1) \quad \text{or} \quad tail(l_2) > head(l_2) \quad \text{or} \quad tail(l_2) < head(l_1) \quad \text{or} \quad tail(l_1) > head(l_2) \\
tail(l_1) &< head(l_1) \quad \text{or} \quad tail(l_2) > head(l_2) \quad \text{or} \quad tail(l_2) < head(l_1) \quad \text{or} \quad tail(l_1) > head(l_2) \\
tail(l_1) &< head(l_1) \quad \text{or} \quad tail(l_2) > head(l_2) \quad \text{or} \quad tail(l_2) < head(l_1) \quad \text{or} \quad tail(l_1) > head(l_2)
\end{align*}
\]

Figure 1. Pairs of collinear lines (end points of \( l_1 \) and \( l_2 \) are, respectively, denoted by \( \circ \) and \( \bullet \); common end points are denoted by \( \ast \).

---

The arithmetic of shapes
Ordering maximal lines

It is now possible to present the representation for shapes based on a linear ordering on the maximal lines of the shape. Let $s$ be a shape described by a set of maximal lines each of which is associated with a line descriptor. The line descriptor induces a natural partition of $s$ into disjoint subshapes $s_1, ..., s_n$, where each $s_i$, $1 \leq i \leq n$, consists of multiple colinear lines. Each maximal line in $s_i$, $1 \leq i \leq n$, has the same line descriptor, $\langle \mu_i, \nu_i \rangle$. Moreover, whenever $i \neq j$, $1 \leq i, j \leq n$, the shape intersection $s_i \cap s_j$ is empty, and $s$ is described by shape union: $s = s_1 + s_2 + ... + s_n$.

Figure 2 presents a shape and its decomposition into disjoint subshapes each of which comprises colinear maximal lines.

The subshapes $s_1, ..., s_n$ of $s$ are arranged so that their line descriptors form a linearly ordered list. That is, $\langle \mu_1, \nu_1 \rangle < ... < \langle \mu_n, \nu_n \rangle$. Each subshape $s_i$, $1 \leq i \leq n$, in turn, is represented by a linearly ordered list of colinear maximal lines $L_i$, $L_i = \langle l_{i,1}, ..., l_{i,m} \rangle$ where each element in $L_i$ has the same line descriptor $\langle \mu_i, \nu_i \rangle$. (Where no confusion can arise subscripts will be omitted.) The lines in $L_i$, $L = \langle l_1, ..., l_m \rangle$, are arranged so that whenever $1 \leq i < k \leq m$, head of $l_i < \text{tail of } l_k$, which is denoted by $l_i < l_k$. Such a linear ordering of the maximal lines in $s_i$ is always possible since any pair of colinear maximal lines in the same shape cannot overlap.

The labelled points for labelled shapes are likewise treated. The labelled points are arranged into lists, each list consisting of all the labelled points which have the same label. The lists are linearly ordered according to a lexicographical ordering on the labels. The labelled points in each list may be arranged according to the order relation $\prec$.

Thus, every labelled shape $s$, $s = \langle L, P \rangle$, is represented by the ordered pair $s$ given by $\sigma = \langle L, P \rangle$. $L$ is a linearly ordered list of linearly ordered lists of maximal lines, $L = \langle L_1, ..., L_n \rangle$, where $L_i$, $1 \leq i \leq n$, contains all the maximal lines in $s$ with the same line descriptor, $\langle \mu_i, \nu_i \rangle$ and $\langle \mu_1, \nu_1 \rangle < ... < \langle \mu_n, \nu_n \rangle$. $P$ is a linearly ordered list of

![Image of Shape](image_url)

**Figure 2.** The decomposition of a shape $s$ into disjoint subshapes, each of which comprises colinear maximal lines.
linearly ordered lists of labelled points, \( P = (P_1, ..., P_m) \) where \( P_k, 1 \leq k \leq m \), contains all the labelled points of \( P \) with the same label, \( A_k \), and \( A_1 < ... < A_m \).

**List comparison properties**

The linear-order based reorganisation of labelled shapes yields the following list comparison properties which are given for the maximal lines in the shape. Similar properties hold for the labelled points of the shape. For any list \( N \), let \( \psi(N) \) denote the line descriptor for the maximal lines in \( N \). Then:

1. Let \( L = (l_1, ..., l_{n_L}) \) and \( M = (m_1, ..., m_{n_M}) \) be linear lists of colinear maximal lines such that \( \psi(L) = \psi(M) \). Then, if \( l_j, 1 \leq j \leq n_L \), is the first line in \( L \) which shares a common line with \( m_k, 1 \leq k \leq n_M \), in \( M \), then the lines \( m_{k'}, k < k' \leq n_M \), do not share a common line with the lines \( l_j, 1 \leq j \leq j \). This observation enables one, as will be proved later, to compare two lists of colinear lines in a time linear in the number of maximal lines in both lists.

2. Let \( L = (L_1, ..., L_{n_L}) \) and \( M = (M_1, ..., M_{n_M}) \) be lists of lists of colinear maximal lines. Then, if \( L_j, 1 \leq j \leq n_L \), and \( M_k, 1 \leq k \leq n_M \) are lists such that \( \psi(L_j) = \psi(M_k) \), the lists \( L_j', 1 \leq j' \leq j \), and \( M_{k'}, k < k' \leq n_M \) satisfy: \( \psi(L_j') \leq \psi(M_{k'}) \).

**Shape arithmetic**

Let \( \circ \) be a variable over the Boolean operations and relations. Then, the shape operation or relation \( o_1 \circ o_2 \) is equivalent to the ordered pair \( (L_1 \circ L_2, P_1 \circ P_2) \), where \( L_1 \circ L_2 \) represents the *shape* operation or relation, and \( P_1 \circ P_2 \) represents the corresponding *set* operation or relation. For example, let \( \circ \) be the shape union operator. Then, \( o_1 + o_2 \) is the labelled shape given by the shape represented by the shape union: \( L_1 + L_2 \), and its associated set of labelled points represented by the set union: \( P_1 + P_2 \). Again, for example, let \( \circ \) be the subshape relation, \( \leq \). Then, \( o_1 \leq o_2 \) if and only if \( L_1 \) represents a subshape of the shape represented by \( L_2 \), and \( P_1 \) represents a subset of the set of labelled points represented by \( P_2 \).

The shape operation or relation, \( L_1 \circ L_2 \), can be decomposed into a list of shape operations or relations on pairs of shapes each of which consists of colinear maximal lines. Since the Boolean operations and relations on shapes are defined in terms of overlapping lines (see Stiny, 1980), it is sufficient to compare the maximal lines in lists, one each from \( L_1 \) and \( L_2 \), which share the same line descriptor.

For convenience, let the subscripts be omitted, and let \( L_1 \) and \( L_2 \) be referred to as \( L \) and \( M \), \( L = (L_1, ..., L_{n_L}) \) and \( M = (M_1, ..., M_{n_M}) \). Let \( \emptyset \) denote the empty list of maximal lines. Suppose \( \circ \) is an operator. Then, \( L \circ M \) is the list \( N \), say, where \( N = (N_1, ..., N_{n_N}) \), and each \( N_i, 1 \leq i \leq n_N \leq n_L + n_M \), is a list of colinear maximal lines given by one of the following:

\[
N_i = \begin{cases} 
L_j \circ M_k & \text{if } L_j, 1 \leq j \leq n_L, \text{ and } M_k, 1 \leq k \leq n_M, \text{ are lists such that } \\
\psi(L_j) = \psi(M_k) \text{ and } L_j \circ M_k \neq \emptyset; \\
L_j & \text{if } \circ \text{ and } \cdot \text{ are not the same operator, and } L_j, 1 \leq j \leq n_L, \text{ is a list such that for each } k, 1 \leq k \leq n_M, \psi(L_j) \neq \psi(M_k); \\
M_k & \text{if } \circ \text{ is the operator } +, \text{ and } M_k, 1 \leq k \leq n_M, \text{ is a list such that for each } j, 1 \leq j \leq n_L, \psi(M_k) \neq \psi(L_j). 
\end{cases}
\]

Clearly, \( N \) can be arranged so that \( \psi(N_1) < ... < \psi(N_{n_N}) \).

Suppose, on the other hand, \( \circ \) is a relation. For integers \( n \) and \( m \), let \( n \circ m \) denote \( n \leq m \) in the case when \( \circ \) is the subshape relation, and \( n = m \) otherwise. Then, \( L \circ M \) if and only if \( n_L \circ n_M \) and for each list \( L_j, 1 \leq j \leq n_L \), there is a list \( M_k, 1 \leq k \leq n_M \), such that \( \psi(L_j) = \psi(M_k) \) and \( L_j \circ M_k \). Therefore, it follows that all one requires is shape algorithms for pairs of shapes each consisting of colinear
maximal lines, and that the maximal lines in the two shapes have the same line descriptor.

In a similar fashion, \( P_1 \circ P_2 \) can be decomposed into a list of set operations or relations on pairs of sets of labelled points having the same label. Again, all that is required are set algorithms for sets of identically labelled points.

Shape algorithms
Let \( \circ \) be a Boolean operator on shapes. Then, the shape expression

\[
\sigma_3 \leftarrow \sigma_1 \circ \sigma_2
\]

signifies that \( \sigma_3 \) is the labelled shape resulting from the shape operation \( \sigma_1 \circ \sigma_2 \). In the special case, where \( \sigma_3 \) is either \( \sigma_1 \) or \( \sigma_2 \), say \( \sigma_1 \), the expression

\[
\sigma_1 \leftarrow \sigma_1 \circ \sigma_2
\]

signifies that \( \sigma_1 \) is replaced by the result of the shape operation \( \sigma_1 \circ \sigma_2 \). The latter expression is particularly relevant to the shape grammar formalism. In any shape grammar there is a current shape denoted by \( \gamma \). Suppose a shape rule, denoted by \( \alpha \Rightarrow \beta \), applies to \( \gamma \). That is, there is a euclidean transformation \( \tau \) such that \( \tau(\alpha) \leq \gamma \). Then, shape rule application is described by the following two shape expressions in sequence:

\[
\gamma \leftarrow \gamma - \tau(\alpha), \quad \gamma \leftarrow \gamma + \tau(\beta).
\]

Algorithms for shape rule application are taken up in greater detail in Krishnamurti (1981).

The following shape expressions are considered:
(a) \( \sigma_1 \leftarrow \sigma_1 + \sigma_2 \),
(b) \( \sigma_1 \leftarrow \sigma_1 - \sigma_2 \),
(c) \( \sigma_2 \leftarrow \sigma_1 \circ \sigma_1 \),
(d) is \( \sigma_2 \leq \sigma_1 \)?
(e) is \( \sigma_1 = \sigma_2 \)?

The shape expressions (a) and (b) are chosen with the view to implementing the shape grammar formalism. Other shape expressions are possible. For instance, interested readers can devise their own algorithms for the shape expressions \( \sigma_3 \leftarrow \sigma_1 + \sigma_2 \). To that end, they may find the presentation in this paper for the shape expression (c) useful.

The inputs to the shape algorithms are ordered lists of colinear maximal lines, \( L \) and \( M \), both lists sharing the same line descriptor. It is assumed that \( M \) is nonempty. The proofs for the correctness of the algorithms are provided in their descriptions, and where necessary illustrations are provided to facilitate explanation.

(a) Shape union: \( L \leftarrow L + M \)

Step 0 (Is \( L \) empty?) If \( L \) is empty, go to step 8. Otherwise, set \( i \leftarrow 1 \), and select the first line, \( m_1 \), in \( M \) and copy it into \( \hat{m} \) which is referred to as the 'working line'.
Set \( j \leftarrow 1 \), and select the first line, \( l_1 \), in \( L \).

Step 1 If tail of \( l_j \leq \) head of \( \hat{m} \), go to step 3. Otherwise, \( \hat{m} \) shares no common line with any line \( l_k \geq \) \( l_j \) in \( L \), and \( \hat{m} < l_j \). This condition is illustrated in figure 3(a).
Insert \( \hat{m} \) into \( L \) as the maximal line immediately preceding \( l_j \) in the list, and continue with step 2.

Step 2 If \( M \) is not exhausted, set \( i \leftarrow i+1 \), and select the next line, \( m_i \), in \( M \) and copy it into \( \hat{m} \). (Notice that at no stage in the execution of the algorithm is the head of \( \hat{m} \) altered, and therefore, at all times \( \hat{m} < m_k \).) Go to step 1. Otherwise, \( M \) is exhausted, and go to step 9.
Step 3 (Determine if $m$ and $l$ share a common line.) If tail of $m \leq$ head of $l$, go to step 5. Otherwise, continue with step 4. (The working line $m$ shares no common line with any line preceding and including $l$ in $L$ and $m > l$. But $m$ may share a common line with line(s), $l_k > l$, in $L$. This situation is illustrated in figure 3(b).) Step 4 If $L$ is not exhausted, set $j = j + 1$, and select the next line, $l_j$, in $L$ and go to step 1. Otherwise, insert $m$ as the last line in $L$ and go to step 8. (The reader can verify that $m$ is a maximal line.)

Step 5 As $m$ and $l$ either share a common line or share an end point, they can therefore be combined to form a single line. This is illustrated in figure 3(c). Set tail of $m$ and $l$ to the minimum of the tails of $m$ and $l$.

Step 6 ($m$ and $l$ now have the same tail; one of these lines must contain the other, and the appropriate line is discarded from further consideration.) Go to step 7 if head of $l$ $\geq$ head of $m$, otherwise, delete $l$ from $L$ and go to step 4. (At this stage $m$ contains $l$. It will be assumed, for convenience, that the subscripts of the lines in $L$ are not altered. That is, at any stage of the algorithm the line currently preceding $l$ in the list $L$ may not necessarily be the line originally subscripted $l_{i-1}$. This step and the next are illustrated in figures 3(d) and 3(e).)

Step 7 As $l$ contains $m$, then $m$ is not maximal and can be ignored. Go to step 2.

Step 8 (All the lines, if any, in $L$ have been examined. There may still be some unexamined lines in $M$.) Copy all the unexamined lines in $M$ in their order, and attach them to the end of list $L$.

Step 9 (Finishing touch.) All the lines both in $L$ and in $M$ have been examined. $L$ contains the maximal lines of the shape union in their sorted order.

Figure 3. Some conditions that can arise in the algorithm for shape union (asterisks indicate the end points that are compared, and arrow heads indicate an unfixed end point).
The computational complexity of this algorithm can easily be determined as follows. Let there be $n_L$ lines in $L$ and $n_M$ lines in $M$. In the worse case, each line, $i$, in $L$ is compared once with the current working line, $m$, except when (in step 2) tail of $i > \text{head of } m$ or when (in step 7) $i$ contains $m$, that is, when $m$ is redefined (step 2). But there are $n_M$ redefinitions of $m$ once for each line in $M$. Therefore, the worst case time bound for shape union is $O(n_L + n_M)$.

(b) Shape difference: $L \leftarrow L - M$

Step 0 (It is assumed that $L$ is nonempty; otherwise the shape difference is empty.)

Set $i \leftarrow 1$, and select the first line, $m_1$, in $M$. Set $j \leftarrow 1$ and select the first line, $l_1$, in $L$.

Step 1 If head of $m_i > \text{tail of } l_j$, go to step 3. [Otherwise, $m_i$ shares no common line with $l_j$, through its successor line in $M$, $m_{i+1}$, may. See figure 4(a).]

Step 2 If $M$ is not exhausted, set $i \leftarrow i+1$, and select the next line, $m_i$, in $M$ and then go to step 1. Otherwise, the procedure is finished, and exit from algorithm.

Step 3 If head of $l_j > \text{tail of } m_i$, go to step 5. [Otherwise, $l_j$ shares no common line with $m_i$, though its successor line in $L$, $l_{j+1}$, may. See figure 4(b).]

Step 4 If $L$ is not exhausted, set $j \leftarrow j+1$, and select the next line, $l_j$, in $L$ and then go to step 1. Otherwise, the procedure is finished, and exit from algorithm.

Step 5 The lines $l_j$ and $m_i$ share a common line, and their corresponding tails and heads can now be compared. One of the following four cases must arise:

1. $l_j - m_i = l_A$, $l_A = \text{tail of } l_j$, tail of $m_i > \text{tail of } l_j$, and head of $m_i \geq \text{head of } l_j$;

2. $l_j - m_i = l_B$, $l_B = \text{head of } m_i$, head of $l_j$, when tail of $l_j \geq \text{tail of } m_i$, and head of $l_j > \text{head of } m_i$;

3. $l_j - m_i = l_A + l_B$, when tail of $m_i > \text{tail of } l_j$ and head of $l_j > \text{head of } m_i$;

4. $l_j - m_i$ is the empty line, when tail of $l_j \geq \text{tail of } m_i$ and head of $m_i \geq \text{head of } l_j$.

Compare the corresponding tails and heads of $l_j$ and $m_i$. Depending upon the four above mentioned cases, one of the following is performed:

Case (1): replace the line $l_j$ in $L$ by $l_A$ and go to step 4.

Case (2): replace the line $l_j$ in $L$ by $l_B$ and go to step 2.

Case (3): replace the line $l_j$ in $L$ by $l_A$. Insert a copy of $l_A$ to precede $l_j$ in the list. That is, $l_A < l_j$ in $L$. Go to step 2.


(Figure 5 pictorially illustrates the four cases mentioned above.)

To complete the proof of the correctness of the algorithm, the following observations are made:

1. If $l_j$ and $l_k$ are two successive lines in $L$ which share a common line with $m$ in $M$, then $l_j - m$ and $l_k - m$ are relatively maximal, and, moreover, the order $< \text{between}$ the lines in $L$ is preserved.

![Diagram](image)

(a) Step 3: head($m_i$) $\leq$ tail($l_j$)

(b) Step 3: head($l_j$) $\leq$ tail($m_i$)

Figure 4. Two conditions that can arise in the algorithm for shape difference (asterisks indicate the end points that are compared).
2. If \( m_i \) and \( m_k \) are two successive lines in \( M \) which share a common line with \( l \) in \( L \),
then \( l - m_i \) and \( l - m_k \) are relatively maximal, and, moreover, the order \(< \) between
the lines is preserved.
As in the case of shape union, the computational complexity for shape difference is
\( O(n_L + n_M) \) where \( n_L \) and \( n_M \) are, respectively, the numbers of maximal lines in \( L \) and \( M \).
(c) Shape intersection: \( N = L \cdot M \)
\( L \) may be assumed to be nonempty, otherwise the shape intersection is empty.
Shape intersection is essentially the complement of the algorithm for shape difference.
Only steps 0 and 5 are modified. In addition to those statements, in step 0 the
following initialization statement must be included:
\[ N \leftarrow \emptyset , \quad k \leftarrow 0 . \]
(That is, initialize the result list to be empty. \( k \) is an index to lines in \( N \).)
Step 5 is modified to read as:
Step 5' Corresponding to each case in step 5 for shape difference, the four cases hold:
(1) \( l_j \cdot m_i = \langle \text{tail of } m_i, \text{head of } l_j \rangle \), when \( \text{tail of } m_i > \text{tail of } l_j \) and
head of \( m_i \geq \text{head of } l_j \);
(2) \( l_j \cdot m_i = \langle \text{tail of } l_j, \text{head of } m_i \rangle \), when \( \text{tail of } l_j \geq \text{tail of } m_i \) and
head of \( m_i < \text{head of } l_j \);
(3) \( l_j \cdot m_i = \langle \text{tail of } m_i, \text{head of } m_i \rangle \), when \( \text{tail of } m_i > \text{tail of } l_j \) and
head of \( m_i < \text{head of } l_j \);
(4) \( l_j \cdot m_i = \langle \text{tail of } l_j, \text{head of } l_j \rangle \), when \( \text{tail of } l_j \geq \text{tail of } m_i \) and
head of \( m_i \geq \text{head of } l_j \).
These cases are illustrated in figure 6. The following steps simulate the cases. Set
\( k \leftarrow k + 1 \), and let \( n_k \) denote the next line in \( N \); \( n_k \) represents \( l_j \cdot m_i \) which is not
empty since \( l_j \) and \( m_i \) share a common line.
Step 5'.1 Set tail of \( n_k \) equal to the maximum of the tails of \( l_j \) and \( m_i \).
Step 5'.2 If head of \( l_j \) \( \geq \) head of \( m_i \), set head of \( n_k \) equal to the head of \( m_i \), and
then go to step 2.
Step 5'.3 Otherwise, set head of \( n_k \) equal to head of \( l_j \) and then go to step 4.

(a) \( \text{tail}(m_i) > \text{tail}(l_j) \) and \( \text{head}(m_i) \geq \text{head}(l_j) \)

(b) \( \text{tail}(l_j) \geq \text{tail}(m_i) \) and \( \text{head}(l_j) > \text{head}(m_i) \)

(c) \( \text{tail}(m_i) > \text{tail}(l_j) \) and \( \text{head}(l_j) > \text{head}(m_i) \)

(d) \( \text{tail}(l_j) \geq \text{tail}(m_i) \) and \( \text{head}(m_i) \geq \text{head}(l_j) \)

Figure 5. The four cases of step 5 in the algorithm for shape difference (asterisks indicate the end
points of the lines in the shape difference).
The proof of the correctness for the algorithm follows from the observation that the intersection of two overlapping maximal lines is a maximal line. The computational complexity for shape intersection is linear in the number of lines in \( L \) and \( M \).

(d) **Subshape and equality:** \( M \subseteq L \) and \( M = L \)?

In this case \( L \) is nonempty, otherwise the relationship does not hold (since \( M \) is assumed to be nonempty). The subshape algorithm will be considered here; the equality algorithm is equivalent to determining if the two lists are identical. A Boolean variable \( \text{flag} \) which returns a value true if \( M \subseteq L \), and false otherwise.

**Step 0** Set \( \text{flag} \leftarrow \text{true} \). Set \( j \leftarrow 1 \), and select the first line, \( l_1 \), in \( L \). Set \( i \leftarrow 1 \), and select the first line, \( m_1 \), in \( M \).

**Step 1** If tail of \( m_i \) < head of \( l_j \), go to step 3.

(Otherwise, \( l_j \) does not share a common line with \( m_i \) though its successor, \( l_{j+1} \), in \( L \) may.)

**Step 2** If \( L \) is not exhausted, set \( j \leftarrow j + 1 \), and select the next line, \( l_j \), in \( L \) and then go to step 1.

Otherwise, \( \leq \) does not hold. Set \( \text{flag} \) to false and go to step 6.

**Step 3** If tail of \( l_j \) < head of \( m_i \), go to step 4.

Otherwise, there is no line in \( L \) which contains \( m_i \). Set \( \text{flag} \) to false and go to step 6.

**Step 4** (\( l_j \) and \( m_i \) share a common line. Does \( l_j \) contain \( m_i \)?)

If tail of \( m_i \geq \) tail of \( l_j \) and head of \( m_i \leq \) head of \( l_j \), go to step 5.

Otherwise, \( l_j \) does not contain \( m_i \). Set \( \text{flag} \) to false and go to step 6.

**Step 5** (\( l_j \) contains \( m_i \). Choose next line in \( M \).)

If \( M \) is not exhausted, set \( i \leftarrow i + 1 \), and select next line, \( m_i \), in \( L \) and then go to step 1.

(a) tail\((m_i)\) > tail\((l_j)\) and head\((m_i)\) ≥ head\((l_j)\)

(b) tail\((l_j)\) ≥ tail\((m_i)\) and head\((l_j)\) > head\((m_i)\)

(c) tail\((m_i)\) > tail\((l_j)\) and head\((l_j)\) > head\((m_i)\)

(d) tail\((l_j)\) ≥ tail\((m_i)\) and head\((m_i)\) ≥ head\((l_j)\)

Figure 6. The four cases of steps 3' in the algorithm for shape intersection (asterisks indicate the end points of the common line).
Figure 7. $L \leftarrow L + M$.

Figure 8. $L \leftarrow L - M$. 
Step 6 (Finishing touch.) The procedure is finished, and flag contains the Boolean value representing whether or not the relation $M \leq L$ holds. Exit from algorithm with the value in flag.

Figures 7 through 10 present pictorial descriptions of the workings of the algorithms on sample shapes.

The corresponding operations and relations on sets of identically labelled points can be performed by means of the standard procedures for manipulating linear lists (see, for instance, Horowitz and Sahni, 1976, chapter 2).

---

Figure 9. $N \rightarrow L \cdot M$.

---

Figure 10. Is $M \leq L$?
Part 2
Efficient decomposition of the Boolean operations and relations
It has been shown that a Boolean operation or relation on labelled shapes can be
decomposed into (a) a list each element of which is the shape operation or relation
on linear lists of colinear maximal lines, and (b) a list each element of which is the
set operation or relation on linear lists of labelled points having the same label. Now
consider the following problem.
Suppose $U$ and $V$ are ordered lists of lists of elements, where $U = \langle U_1, ..., U_n \rangle$,
$V = \langle V_1, ..., V_m \rangle$, and the $U_k$s and $V_k$s are sorted according to their respective keys
$(u_1, ..., u_n)$ and $(v_1, ..., v_m)$. That is, $u_1 < ... < u_n$, and $v_1 < ... < v_m$. Let $\odot$ be an
operator. The problem is that for each list $V_j$ with key $v_j$ in $V$ to search the list $U$
for a list $U_k$ such that $u_k = v_j$. If the search is successful $U_k$ is replaced by $U_k \odot V_j$,
which may lead to the deletion of $U_k$ from $U$ in the case when $\odot$ is a difference or
intersection operator. This occurs when $U_k \odot V_j$ is empty. If the search is unsuccessful
a copy of $V_j$ is inserted into the list $U$ in the case when $\odot$ is a union operator, but at
the same time preserving the linear ordering of the elements of $U$. Here $U$ is a
dynamic list and $V$ is a static list. The analogy to the shape algorithms is obvious.
The crucial computation question related to the problem is: how effectively can
this list searching, and/or list insertion/deletion be carried out? The answer to this
depends to a large extent on the internal organization for the list. Two possible
types of data structures are considered: (a) linked lists, and (b) balanced binary trees.
Linked lists are useful when the lists are static and reasonably small in size, which is
often the case with shape rules. Balanced binary trees are useful when the lists are
dynamic and constantly increasing in size, which is generally the case with the current
shape in the shape grammar formalism. Algorithms for searching, insertion, and
deletion on linked lists are straightforward (see, for instance, Aho et al, 1974). In
the remainder of this section a search algorithm is presented when the list $U$ is
organized as a balanced binary tree. This algorithm utilizes the Brown–Tarjan (1979)
fast merge algorithm.

Organizing a linear list as a balanced binary tree
It will be assumed that the reader is familiar with binary tree terminology (see, for
instance, Knuth, 1973b) with the exception that the term 'son' is now replaced by
'offspring'. A binary tree is height-balanced if the height of the left subtree of any
node never differs by more than ±1 from the height of the right subtree of the node.
The height of a tree is the length of the longest path from the root to a leaf node.

```
      (0)
     /   \
  (0)   (0)
 /   \   /   \
(0)  (+1) (0)  (-1)
 / \  / \ / \  / \  
(0) (0) (0) (0) (0) (0)
```

Figure 11. A balanced tree representation for shapes in figure 2 (I denotes the node representing
shape s, and the numbers within the brackets are the balance factors).
A leaf node has no offsprings. The left and right subtrees of a node are, respectively, the trees rooted at the left and right offsprings of the node.

A linear list is organized as a balanced binary tree as follows. Each node in the tree represents a list element, and is referenced by its key. If \( k \) is the key presented by a node, then all nodes in its left subtree have keys \(<k\), and all nodes in its right subtree have keys \(>k\). The proper location of a node in the tree with a given key value can be determined by a standard binary tree search (Knuth, 1973a) in \(O(\log n)\) time, where \( n \) is the list size. The linear list can be reproduced in its original list order by an inorder traversal (Knuth, 1973a) of the tree in a time proportional to the list size. Figure 11 gives a balanced tree representation for the shape in figure 2.

A balanced tree search algorithm

The obvious method for comparing \( U \) and \( V \) is to take each key \( v_j \) in turn, and search the balanced tree for \( U \), for a node associated with a key which equals \( v_j \). If such a node exists, the search is successful. When \( \cap \) is a difference or intersection operator the algorithm may involve the deletion of the node from the tree. If no such node exists, the search is unsuccessful. When \( \cap \) is a union operator the algorithm involves inserting a new node with key value equal to \( v_j \) into the tree. To insert a node into or to delete a node from the balanced tree requires \(O(\log n)\) additional steps for rebalancing the tree. Rebalancing becomes necessary whenever the modified tree—as a result either of node insertion or of node deletion—fails the height balancing requirement. Rebalancing will be considered in greater detail later in this section. Knuth (1973b, algorithm 6.2.3A) provides a good description of node insertion into a tree. Deletion, however, is slightly more difficult. For a reasonable discussion of node deletion from a balanced tree the reader is referred to Crane (1972, pages 43–45, and chapter 4 for an ALGOLW description of the algorithm) and Wirth (1976, algorithm 4-64). The computational effort involved in comparing \( U \) and \( V \) is, in the worst case, \(O(m \log n)\).

When \( n > m \) the time bound can be improved. In the case of merging two disjoint lists into a single list Brown and Tarjan (1979) have demonstrated a computational time bound of \(O(m \log(n/m))\). I believe their contribution to balanced tree search can be used for other forms of list comparisons without materially changing the time complexity. The following material is adopted from their paper. As before, the first step is to search tree \( U \) for a node with its key equalling \( u_1 \). Once \( u_1 \) has been compared, some appropriate algorithmic action takes place which may result in a copy of \( u_1 \) together with the list element \( V_1 \) being inserted into the tree, or the node whose key equals \( u_1 \) being deleted from the tree, or the tree being left as it is. At the start of the general step, the keys \( u_1, \ldots, u_k \) have been compared with the nodes in tree \( U \), and a record is maintained of the nodes in the search path from the root to the last node compared with key \( u_k \), which may, in the case of insertion, be the node whose key equals \( u_k \). This path acts as a ‘finger’ into tree \( U \) moving from left to right as the keys of the lists in \( V \) are compared; the finger is useful since only nodes to the right of it have to be visited during later comparisons. This follows from list comparison property 2 (see page 467).

The general step comprises two parts. First, the finger is retreated towards the root, just far enough to the position that \( u_{k+1} \) ‘lies’ in the subtree rooted at the end of the finger. The key \( u_{k+1} \) is compared with the nodes in this subtree, and the finger is extended to follow the search path. After \( m-1 \) executions of the general step the algorithm is complete.

However, this scheme is complicated by the fact that both insertion and deletion into the tree may require rebalancing of the tree. When rebalancing takes place, it may remove a node from the finger path traced out by the search. It is possible to
update the finger to be consistent with the rearrangement, but Brown and Tarjan suggest that it is easier to just 'forget' about the part of the path which is corrupted. That is, to retract the finger to the last, and in the case of insertion the only, point of rebalancing. The algorithm takes the form shown in figure 12. At the start of the general step one now has a record on only part of the search path to the last node compared with \( v_k \). The general step proceeds as before. Notice that the first step need not be treated separately. At the start of the algorithm, the finger is initialized to the root of tree \( U \) (which is always on the path to the first comparison), and the general step is executed \( m \) times.

The algorithm can be speeded up by keeping a record of all nodes on the finger path where the finger turns left—that is, those nodes on the path whose left offspring is also in the path. It is easy to show that these are precisely the nodes on the path which have a larger key than the most recently compared item. Then, only these nodes have to be examined in the 'climbing up' phase of the general step.

![Diagram](image)

(a) Start of the general step

(b) Climbing up phase

(c) Compare \( v_{k+1} \) and extend finger

(d) Rebalancing the tree

Figure 12. The form of the balanced tree search algorithm.

Rebalancing transformations

Rebalancing is required whenever the balanced tree becomes unbalanced—that is, the tree fails to satisfy the height-balance requirement as a result of an insertion or a deletion. Each node is associated with a balance factor which is the difference between its right and left subtree heights, and can take on values from \( \text{lefttaller} (-1) \), \( \text{balanced} (0) \), and \( \text{righttaller} (1) \) which have the obvious interpretations. Consequently, insertion of a node into the left (right) subtree of a node whose balance factor is originally \( \text{lefttaller} (\text{righttaller}) \), or deletion of a node from the left (right) subtree of a node whose balance factor is originally \( \text{righttaller} (\text{lefttaller}) \) results in the entire tree becoming unbalanced.

A leaf node is always inserted or deleted. In the case of deletion, if the node to be deleted is not a leaf node, the contents of a specific leaf node is copied into the node designated to be deleted, and the leaf node is deleted instead. Furthermore, the finger is extended to follow the search path to the node to be deleted. Rebalancing can be described as follows. Let \( x \) denote the inserted or deleted node. The successive ancestors of \( x \) (denoted by \( z \)) moving up towards the root along the search
path are examined; and, if necessary, the finger is retracted to \( z \). During this climb one of the following steps is performed.

1. \( z \) is originally balanced:
   Change the balance factor of \( z \) to \( \pm 1 \) as appropriate. In the case of insertion the climb is continued; in the case of deletion the tree remains height-balanced, and the climb is stopped.

2. \( z \) is originally unbalanced:
   \( z \) becomes either balanced or doubly unbalanced. In the latter case, the subtree rooted at \( z \) is locally modified via one of two types of transformations: single and double rotation. These are shown in figure 13 wherein the symbol \( \ominus \) indicates the deleted node, and the symbol \( \oplus \) indicates the inserted node. Notice that both in single and in double rotation the mirror-image transformation is possible. In fact there is an additional special case of double rotation which occurs when \( h = 0 \), and \( x \) is an inserted node. In that case, \( x \) equals \( w \) and the subtrees \( \alpha, \beta, \gamma, \) and \( \delta \) are empty. In the case of deletion one still has to continue the climb; for insertion, the rebalancing is completed.

If the root is reached during the climb rebalancing is complete; for insertion the balanced tree has increased in height, and for deletion decreased in height.

![Figure 13. Modifying the subtree rooted at \( z \) by the rebalancing transformation; (a) single rotation, and (b) double rotation.](image)

Data structures

The following data structures serve to represent a labelled shape \( \sigma \), which is described by the ordered pair \((L, P)\), where \( L = (L_1, \ldots, L_n) \) and \( P = (P_1, \ldots, P_m) \). \( L \) is an ordered list of ordered lists of colinear maximal lines. \( P \) is an ordered list of ordered lists of labelled points having the same label. \( \sigma \) is assumed to be rational. Each data structure is described by the fields of the nodes in the data structure. The internal implementation of the data structures is given in terms of data classes. Each data class defines the fields of each data node in the class.

Each node in the data class POINT comprises two fields, and for rational shapes each field represents a pair of integers. POINT implements the data structures for the coordinates of a point and the line descriptor for a list of multiple colinear lines. The data structures are:

- coordinate node  \[ x \ | \ y \]
- line descriptor node  \[ \mu \ | \ \nu \]

The fields in the data structures have the obvious interpretation.

The data class LABEL consists of one field, \( A \), which stores the label for a list of labelled points sharing the same label.

Each node in the data class LIST consists of three fields one of which, referred to as next, is a pointer to a node in LIST. LIST serves to implement linked lists.
Linked lists are used to implement the lists \( L, P, L_1 \) and \( P_2 \). The data structures are:

\[ \text{\(L_1\)-node or \(P_2\)-node} \quad \begin{array}{|c|c|c|} \hline \text{key} & N & \text{next} \rule{0pt}{2.5ex} \hline \end{array} \]

where \( \text{key} \) is a pointer either to a node in \( \text{POINT} \) which contains the line descriptor for a list of colinear maximal lines, or to a node in \( \text{LABEL} \) which stores the label for a list of labelled points sharing the same label; \( N \) is a pointer to a node in \( \text{LIST} \) which is the first node of the linked list representing either the list of colinear multiple lines or the list of labelled points having the same label; and \( \text{next} \) points to the next node in the linked list in the sense that \( \text{key}[\text{node}] < \text{key}[\text{next}[\text{node}]] \).

\[ \text{\(\text{line-node} \)} \quad \begin{array}{|c|c|c|} \hline \text{point} & \text{mark} & \text{next} \rule{0pt}{2.5ex} \hline \end{array} \]

where \( \text{point} \) points to a node in \( \text{POINT} \) which stores the coordinates of a labelled point; \( \text{mark} \) is an integer which is not used in this paper (see Krishnamurti, 1981); and \( \text{next} \) satisfies the property \( \text{point}[\text{node}] < \text{point}[\text{next}[\text{node}]] \).

The last data class to be considered is \( \text{BTREE} \). \( \text{BTREE} \) simulates height-balanced binary trees. Each node in \( \text{BTREE} \) consists of five fields.

\[ \text{\(\text{tree-node} \)} \quad \begin{array}{|c|c|c|c|c|} \hline B & \text{left} & \text{right} & \text{key} & N \rule{0pt}{2.5ex} \hline \end{array} \]

The fields have the following interpretation. \( B \) is the balance factor of a node in a balanced tree; \( \text{left} \) and \( \text{right} \) are pointers to nodes in \( \text{BTREE} \) and represent, respectively, the left and right offspring of a node in a balanced tree; and \( \text{key} \) and \( N \) have the same interpretations as given earlier.

Each labelled shape \( \sigma \) is represented by the pair \( <\text{top}_s, \text{top}_p> \) where \( \text{top}_s \) is a pointer to a linked list or a pointer to a height-balanced tree. The subscript \( s \) refers to the data structure which maintains the maximal lines and the subscript \( P \) refers to the data structure which maintains the labelled points. A shape is described by the pair \( <\text{top}_s, \text{null}> \) where \( \text{null} \) is a special pointer which points to an empty node. In a similar fashion, a labelled shape consisting of only labelled points is represented by the pair \( <\text{null}, \text{top}_p> \). Figure 14 illustrates the internal organization of a labelled shape in which its shape is housed in a balanced tree, and its set of labelled points is housed in a linked list. Notice that the data structures are arranged according to the data classes to which they belong with only sample links indicated.

Remark: It is assumed that these data structures can be implemented without violating or mixing data types. For instance, the data classes can be regarded as collections of integer arrays, and the fields may be equivalenced via some mechanism, say the Fortran \texttt{EQUVALENCENCE} statement.

Algorithm
The data structures are incorporated into algorithms 1–4 which describe, in Algol-like notation, the algorithmic framework for decomposing the Boolean operations or relations on labelled shapes into (a) lists of shape operations or relations on lists of colinear maximal lines, or (b) lists of set operations or relations on lists of identically labelled points. The set of algorithms 1–4 is based on the modified Brown–Tarjan balanced tree search algorithm, and are called: SHAPE EXPRESSION, DELETE NODE, INSERT NODE, and ROTATE TREE. SHAPE EXPRESSION describes the algorithmic framework in which the algorithms for the shape expressions \( \sigma_1 \circ \sigma_2 \), \( \sigma_1 \circ \sigma_2 \) in the case that \( \circ \) is an operator, and \( \circ \), if \( \circ \) is otherwise, are invoked. The inputs are the labelled shapes \( \sigma_1 \) and \( \sigma_2 \); the shape operator or relation \( \circ \) which takes on values from \{\( +, -, \cdot, \leq, = \}\}; and a parameter \( c \) which takes on values from \( \{s, P\} \). SHAPE EXPRESSION invokes algorithms DELETE NODE and INSERT NODE, respectively, wherever a node is deleted from or inserted into the balanced tree rooted at \( \text{top}_e(\sigma_1) \). DELETE NODE and INSERT NODE both invoke ROTATE TREE.
Figure 14. The internal organization of a labelled shape.
Algorithm SHAPE EXPRESSION (α, β, γ, ε, ψ, μ, π, λ, ρ, ζ, ξ)

This routine describes the decomposition of the Boolean operations and relations based on the fast balanced tree search. The following shape expressions are considered as illustrative examples:

- α = α₀ ⊗ α₁ for α an operator; and α₀ = α₀ for α otherwise.
- α₁ = represented by the pair (top₁(α₁), top₁(α₁)), where top₁(α₁), c ∈ {α₀, π₁, ζ₁}, is the root of a balanced binary tree.

The following 'conventions' are adopted:

1. N denotes the list of elements represented by node u, the first element of which is pointed to by N[u].
2. ψ(u) is the key associated with node u, and is given by: (μ, π)[key[u]] for c = μ, and A(key[u]) for c otherwise.
3. The expressions:
   - DATA CLASS fetches an unused (available) node in DATA CLASS, initializes all its fields, and assigns it to node.
   - DATA CLASS node releases the node pointed to by node, and makes it available for future use.

Initialization

path: an array containing all the nodes traversed from the root to the current node.

successor: an array containing all the nodes in path whose left offsprings are also in path.

pptr, sptr: pointers to the last entry in path and successor respectively.

flag: a Boolean variable which signifies whether or not α₀ ⊗ α₁ if α is a relation, and α₀ ⊗ α₀ = 0 if α is otherwise.

ψ(u) = DATA CLASS[p] + 1 = top₁(α₁) if α₀ is initially assumed nonempty.

pptr = 0

flag = true

Balanced tree search algorithm

GETNEXT(top₁(α₁)) is a global routine which fetches, on each invocation, the 'next' node, in order, in the data structure headed by top₁(α₁), c ∈ {α₀, π₁, ζ₁}, respectively.

for u = GETNEXT(top₁(α₁)) while (u ≠ null and flag)

Climb up phase. Retract the path until top of path, u, is either the root or satisfies ψ(u) ≤ ψ(α₁). It is only necessary to examine the nodes pointed to in successor:

while (sptr > 0) and (ψ(u) > ψ(path[successor[sptr]])) do {pptr = successor[sptr];

u = path[pptr];

if balanced tree search. Compare keys and extend the path

search = true
while search

if ψ(u) = ψ(α₁)

Matching keys found. Perform the appropriate action

search = false

case α in

(+, −, −) Shape or set operation

then

Nᵦ = Nᵦ ⊗ Nᵦ

if Nᵦ = 0 then {DELETE NODE

flag = path[1] ≠ null

(≤, −) Shape or set relation

flag = Nᵦ ⊗ Nᵦ

do

if ψ(α) < ψ(u)

Branch left

then

successor[sptr + 1] = pptr

x = left[u]

do

Branch right

else

if x = null

There is no list Nᵦ with the same key value as Nᵦ.

search = false

case α in

(+) Insert Nᵦ into the tree

x = BTREE

ψ(x) = ψ(α)

Nᵦ ⊗ Nᵦ

INSERT NODE

(−, −) If successor[sptr] = pptr then sptr = − 1

(≤, −) flag = false

do

Finishing touches

case α in

(+, −, −) top₁(α₁) = path[1]

return

(≤, −) return(flag)

end SHAPE EXPRESSION

Algorithm 1.
Algorithm DELETE NODE

This routine deletes node $u$ from the tree and rebalances the tree.

```
if $right[u]$ = null
  then $x = left[u]$
    if $left[u]$ = null
      then $x = right[u]$

    if $u$ has nonempty subtrees. Delete, instead, a node from the higher subtree of $u$
    after copying its content into $u$ so that the resulting tree still represents the altered
    list in its original list order.

    if $B[u]$ = right,taller
      then
        # Let $y$ be the immediate successor of $u$ in the list
        path[$ppr + 1] = $y = right[u]
          while $left[y]$ = null do
            successor[$ppr + 1] = $ppr
            path[$ppr + 1] = $y = left[y]
            $x = right[y]$

        # Let $y$ be the immediate predecessor of $u$ in the list
        successor[$ppr - 1] = $ppr
        path[$ppr] = $y = left[u]
          while right[y] = null do
            path[$ppr] = $y = right[y]
            $x = left[y]$

      # Copy contents of $y$ into $u$ and rename $y$
      $u = y$
      $\psi[u] = \psi[y]$
      $M[u] = M[y]$
    
    if $c = s$ then $POINT = key[u]$ else $LABEL = key[u]$

    B[TREE] = $u$
    $ppr = -1$

    $x$ is either the root or becomes an offspring of the former parent of $u$.
    if $ppr = 0$
      then
        path[$ppr] = $x$

        # Deleting a node from the right (left) subtree effectively makes the left(right) subtree
        taller:
        $z = path[ppr]$
        if $u = right[z]$
          then
            $a = left[z]$
          else $a = right[z]$

        # Save the path pointer so as not to destroy path. Trace back along the path towards the
        root either adjusting the balance factors or rebalancing the tree.
        savepr = $ppr$
        while (savepr > 0) and ($B[z]$ = balanced)
          if $B[z]$ = a
            # Subtree rooted at $z$ has become balanced.
            $B[z] = balanced$
            if (savepr = -1) > 0 then $a = right[z]$ = path[savepr]
              then leftrtaller
                else righttaller

          # $z$ has become doubly unbalanced. That is, the difference in the heights of the
          right and left subtrees of $z$ equals 2. Rotate the subtree rooted at $z$ and let
          $w$ be the new root.
          do
            $y$ = (if $a = righttaller$ then right[z] else left[z])
            ROTATE TREE
            $z = x$
          end do
          else path[ppr] = savepr = $w$
          if (savepr = -1) > 0
            if $z = right[z]$ = path[savepr]
              then [right[z] = $w$
                then $a = left[z]$
                  else $a = right[z]$
          end if
          savepr = $a$

      # Rebalancing may have corrupted the path. Delete the invalidated section.
      while ($ppr > 0$) and (successor[ppr] > $ppr$) do $ppr = -1$

end DELETE NODE
```

Algorithm 2.
Algorithm INSERT NODE

1. This routine inserts a new node \( x \) as an offspring of \( u \), and rebalances the tree.
2. If \( z(u) < z(w) \) then \( \text{left}(u) \rightarrow x \) else \( \text{right}(u) \rightarrow x \).
3. \( p = \text{path}(\text{save} \text{ptr} + 1) \).
4. Save the path pointer so as not to destroy \( p \). Trace back along the path towards the root.
5. Adjusting the balance factors until the point of rebalancing is reached.
6. \( x = \text{path}(\text{save} \text{ptr} + p \text{ptr} - 1) \).
7. \( y = x \).
8. While \( \text{save} \text{ptr} > 1 \) and \( \text{balanced} \) do:
   - \( z = \text{path}(\text{save} \text{ptr} - 1) \).
   - If \( y = \text{right}(z) \) then \text{right}\text{taller} else \text{left}\text{taller}.
   - \( y \rightarrow z \).
9. \( z \) is either the root or the point of rebalancing. Determine in which subtree of \( z \) is \( x \) inserted.
10. If \( y = \text{right}(z) \) then \text{right}\text{taller} else \text{left}\text{taller}.
11. The following segment is a translation of steps 7-10 of algorithm 6.2.3A in Knuth (1973b).
12. If \( B[z] = a \), then:
   - \( B[z] \rightarrow a \).
13. \( z \) is doubly unbalanced. That is, the difference in the heights of the right and left subtrees of \( z \) equals \pm 2. Rotate the subtree rooted at \( z \), and let \( w \) be the root of the rotated subtree.
14. \text{ROTATE TREE}
15. \( \text{path}(p \text{ptr} + \text{save} \text{ptr}) \rightarrow w \).
16. If \( \text{save} \text{ptr} > 1 \) then:
   - \( z \rightarrow \text{path}(\text{save} \text{ptr} - 1) \).
17. If \( z = \text{right}(t) \) then \text{right}\text{taller} else \text{left}\text{taller}.
18. Rebalancing may have corrupted the path; delete the invalidated section.
19. While \( \text{sptr} > 0 \) and \( \text{successor}(\text{sptr}) > p \text{ptr} \) do \( \text{sptr} \rightarrow 1 \).

end INSERT NODE

Algorithm 3.

Algorithm ROTATE TREE

1. This routine rotates the subtree rooted at \( z \). \( w \) contains the new root of the subtree. \( a, w, y, \)
2. and \( z \) are global variables.
3. If \( B[y] = \rightarrow a \):
   - \( \text{Double rotation} \) (figure 13(b)).
4. If \( a = \text{right}\text{taller} \):
   - \( w \rightarrow \text{left}(y) \).
5. \( \text{left}(y) \rightarrow \text{right}(w) \).
6. \( \text{right}(w) \rightarrow y \).
7. \( \text{left}(w) \rightarrow z \).
8. \( B[z] \rightarrow \) (if \( B[w] = a \) then \( \rightarrow a \) else \text{balanced})
9. \( B[y] \rightarrow \) (if \( B[w] = \rightarrow a \) then \( a \) else \text{balanced})
10. \( B[w] \rightarrow \text{balanced} \).
11. \( \text{Single rotation} \) (figure 13(a)).
12. If \( a = \text{right}\text{taller} \):
   - \( \text{right}(z) \rightarrow \text{left}(y) \).
13. \( \text{left}(y) \rightarrow z \).
14. \( \text{left}(z) \rightarrow \text{right}(y) \).
15. \( \text{right}(y) \rightarrow z \).
16. \( B[z] \rightarrow \) (if \( B[z] = \text{balanced} \) then \( \rightarrow a \) else \text{balanced})

end ROTATE TREE

Algorithm 4.
which describes the two types of rebalancing transformation: single and double rotation which are shown in figure 13. GETNEXT which is not given here is a global routine for selecting the nodes in the data structure rooted at top (03) in order.

Acknowledgements. I am indebted to George Stiny for his support and whose suggestions considerably improved the presentation of this paper. The research reported in this paper was supported in part by a grant from the Science Research Council.

References
Horowitz E, Sahni S, 1976 Fundamentals of Data Structure (Computer Science Press, Woodland Hills, Calif.)
Knuth D E, 1973b The Art of Computer Programming, Volume 3: Sorting and Searching (Addison- Wesley, Reading, Mass)
Stiny G, 1980 "Introduction to shapes and shape grammars" Environment and Planning B 7 343-351