Philips Research Laboratory Brussels
Av. E. Van Becelaere 2, Box 8
B-1170 Brussels, Belgium

Manuscript M106
(third version)

List Coding of Quadtrees

Christian Ronse

February 1987

Abstract: *We propose to encode quadtrees as lists, where a parent node is represented by a pair of parentheses and a leaf node is represented by a number coding its colour (0 or 1 for two-tone images). This list representation is a particular case of the edge-string [8] used by the author for the coding of adjacency trees in binary images. It allows also an easy implementation in LISP of various image processing operations applied to quadtrees, such as: raster to quadtree conversion, quadtree to raster conversion, set operations, symmetries, shifts, Euler numbers, etc., by the use of recursive functions. We give indeed the FRANZ LISP code for such operations. Thanks to its simplicity, it can be useful for prototyping of quadtree processing algorithms and for educational purposes.*

Keywords: quadtrees, recursivity, edge-string, depth-first expression, list coding, LISP coding, FRANZ LISP.

Domains: Digital image processing, LISP, didactic applications.

I. Introduction

The *quadtree* represents a well-known hierarchical representation of digital images. It is built recursively as follows. Consider a $2^n \times 2^n$ digital image I. If all pixels of I have the same colour c, then I can be represented as a single node to which we associate that colour c. Otherwise we divide I into four $2^{n-1} \times 2^{n-1}$ images I_0, I_1, I_2 and I_3, which can be represented as *children nodes* of I in a tree. For each I_i ($i = 0, \ldots, 3$), we apply the same operation: if the pixels of I_i are not all of the same colour, we subdivide I_i into four $2^{n-2} \times 2^{n-2}$ images I_{i0}, I_{i1}, I_{i2} and I_{i3}, which will be represented as the *children nodes* of I_i in the tree. We continue in this way until we obtain homogeneous regions. The corresponding terminal nodes of the tree are called *leaf nodes*. To each one of them we associate its colour.

We show this construction in Figure 1 for a two-tone 4×4 image.

$$\begin{array}{cccc} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{array}$$

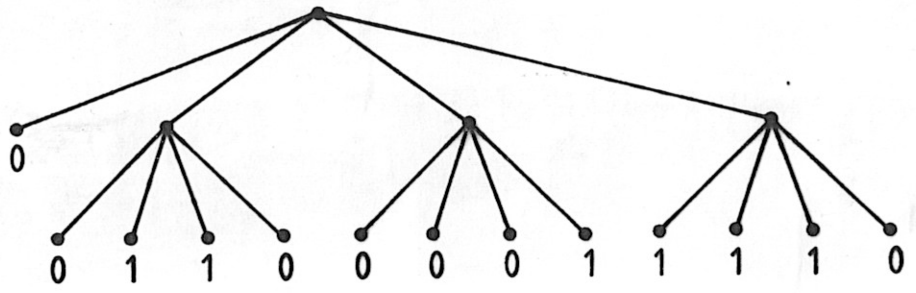

Figure 1. *An image and its quadtree*

A quadtree node which is not a leaf node is the *parent node* of four *children nodes*. Note that these four children nodes cannot be leaf nodes having all the same colour.

A survey of the quadtree has been made by Samet [11]. (Incidentally, he calls it the *region quadtree*, in order to distinguish it from a variant structure that he calls the *point quadtree*). The reader should consult that reference for an extensive bibliography on this subject.

Many papers have been devoted to efficient ways for coding quadtrees (see in particular Section 2.2 of [11] and the relevant bibliography). Each representation can be justified on three grounds: its memory cost, the computational efficiency and the simplicity of the corresponding code for the implementation of basic digital image operations.

One of the most compact coding schemes is the *depth-first* (in brief, *DF*-) expression of Kawaguchi and Endo [5,6]. As explained in [12], it often requires less memory than Gargantini's linear quadtree [4]. Here every non-leaf node is coded by a left parenthesis, while every leaf node is coded by its colour. Thus the quadtree of Figure 1 has the following DF-expression:

$$(0(0110(0001(1110 \tag{1}$$

In this paper, we propose to encode quadtrees by a similar expression, but where each left parenthesis is balanced by a right one. For example the quadtree of Figure 1 is coded as

$$(0(0110)(0001)(1110)) \tag{2}$$

We call this representation the *list coding*. It has been considered also in [7]. It was briefly described in [5]; it was called there the "context-free grammar G'", and it was considered as an intermediate form of the DF-expression. The latter is obtained from the list coding by deleting all right parantheses. As we will explain in Section II, it is also a particular case of the *edge-string* which we used in [8] in order to encode the tree structure formed by the adjacency relations between black and white connected components in a binary picture.

We will see in Section II that for an image having c distinct colours, the asymptotic memory cost of the list coding is that of the DF-expression multiplied by the factor

$$\frac{5\log(c+2)}{4\log(c+1)} \tag{3}$$

For binary images (*i.e.*, for $c = 2$), this factor is approximately 1.577.

The advantage of the list coding, compared to the DF-expression, resides in the ease of the implementation of picture processing operations. Indeed, the underlying grammar is nearly the same as that of symbolic expressions in LISP. By inserting blank spaces between the various colour numbers and sublists in (2), we get the list

```
(0 (0 1 1 0) (0 0 0 1) (1 1 1 0))                    (4)
```

which is a well-formed LISP list. We call it the *LISP coding* of the quadtree.

Moreover LISP is a language particularly well suited to recursivity, which is a basic feature of quadtrees. Thus many standard image operations can easily be implemented by recursive functions. Indeed, we give in Section III an implementation in FRANZ LISP of several basic operations applied to quadtrees of binary pictures: raster to quadtree conversion, quadtree to raster conversion, set operations, symmetries, shifts, and Euler numbers. The resulting code is simple, and these operations run theoretically in linear time, at worst in the number of pixels, and at best in the number of nodes of the quadtree. In practice, this remains true if one neglects the time taken by the LISP *garbage collector* for three of these operations: raster to quadtree conversion, quadtree to raster conversion, and Euler numbers.

We explain in Section IV that other operations can be implemented, for example the construction of Gargantini's linear quadtree, the determination of the colour of a pixel, etc.. Moreover, the methodology can also be generalized to related structures, such as octtrees or point quadtrees (see [11] for a definition).

The simplicity of the LISP code for these operations indicates that list coding can be used for educational purposes: the teaching of LISP to people having some experience in image processing, or the teaching of quadtrees to LISP programmers.

Readers interested solely in the LISP implementation can skip Section II, which is a 'cultural' background on the relations between list coding and other string-type grammars for the representation of trees.

II. From edge-strings to lists

The problem of coding trees is also treated in the detection of connected components of binary pictures. Here the graph induced by the adjacency relation between black and white connected components is a tree [9]; moreover, for every node c in the tree corresponding to a (black or white) connected component C, the children nodes of c correspond to the connected components of the opposite colour adjacent to C and surrounded by it. This graph is called the *adjacency tree* of the picture. There are several differences between this type of tree and the quadtree. A first one is that the number of children nodes of a nonleaf node can be any positive integer, not only 4. A second one, more important from our point of view, is that the ordering of the children nodes of a given node is irrelevant, since the ordering of the holes in a two-dimensional connected component has no topological meaning. This contrasts with the quadtree, where the order of the children nodes is essential.

We will thus introduce a refinement of the notion of tree. We call an *ordered tree* one in which the ordering of the children nodes of a nonleaf node is taken into account; if that order is irrelevant, one calls it an *unordered tree*. Thus quadtrees are ordered trees, while adjacency trees of binary pictures are unordered trees.

An interesting fact is the existence of two general methods [1,8] for coding ordered trees as strings of symbols, which have been found in the context of a study of the adjacency tree (which is an unordered tree). The first one, due to Buneman [1], was called in [8] the *vertex-string*; in [8] we derived from it the second one, which was then called the *edge-string*. The vertex- and edge-strings of an ordered tree $\mathcal{T}$ can be defined as follows. Define the path $\mathcal{P}$ such that

(*i*) $\mathcal{P}$ begins and ends in the top node t of $\mathcal{T}$;

(*ii*) $\mathcal{P}$ passes through every node of $\mathcal{T}$ at least once;

(*iii*) $\mathcal{P}$ is shortest possible with respect to (*i*) and (*ii*);

(*iv*) Given a parent node v, $\mathcal{P}$ passes through the children nodes of v in their order of succession.

This path $\mathcal{P}$ is *unique*. The succession of nodes of $\mathcal{T}$ in $\mathcal{P}$ forms the *vertex-string* of $\mathcal{T}$ (this is because we called in [8] a node a *vertex*). Let us label each edge of $\mathcal{T}$ by the label of the bottom node on this edge (*i.e.*, the one which is a child node of the other node). Let E be the sequence of edge labels in $\mathcal{P}$; then every node label intervenes in E, except that of the top node t of $\mathcal{T}$. We choose thus tEt as the *edge-string* of $\mathcal{T}$.

These two strings can be built recursively:

(1°) An ordered tree having a single node v has vertex-string v and edge-string $v\,v$.

(2°) Let $\mathcal{T}$ be an ordered tree having vertex-string $V_{\mathcal{T}}$ and edge-string $E_{\mathcal{T}}$. Suppose that x is a leaf node of $\mathcal{T}$ and that we modify $\mathcal{T}$ by adding to it k nodes $y_1, \ldots, y_n$ which are children nodes of x. Let $\mathcal{R}$ be the resulting ordered tree; then its vertex-string $V_{\mathcal{R}}$ and edge-string $E_{\mathcal{R}}$ are obtainde in the following way:

(a) $V_{\mathcal{R}}$ is built from $V_{\mathcal{T}}$ by replacing in it the unique occurrence of x by the string $x\, y_1\, x\, y_2\, x\, \ldots\, x\, y_k\, x$.

(b) $E_{\mathcal{R}}$ is built from $B_{\mathcal{T}}$ by replacing in it the unique occurrence of $x\,x$ by the string $x\, y_1\, y_1\, y_2\, y_2\, \ldots\, y_k\, y_k\, x$.

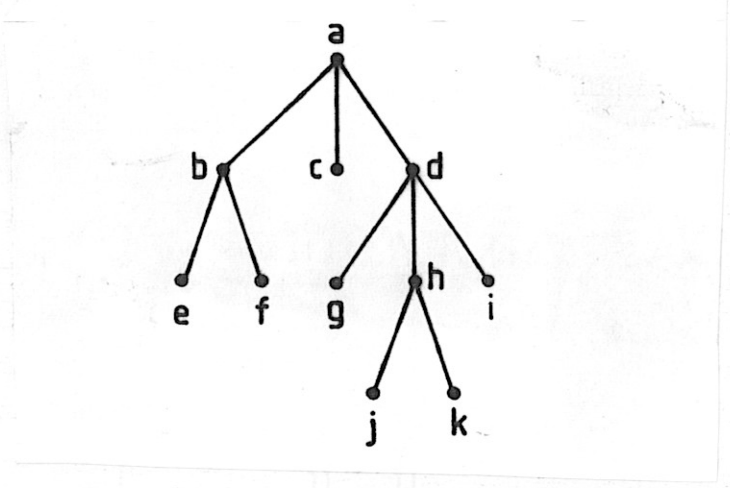

Figure 2. *A tree*

For example the tree shown in Figure 2 has vertex-string

$$a\,b\,e\,b\,f\,b\,a\,c\,a\,d\,g\,d\,h\,j\,h\,k\,h\,d\,i\,d\,a \tag{5}$$

and edge-string

$$a\,b\,e\,e\,f\,f\,b\,c\,c\,d\,g\,g\,h\,j\,j\,k\,k\,h\,i\,i\,d\,a \tag{6}$$

and it is easy to check that a node having c children occurs $c+1$ times in the vertex-string, while every node occurs 2 times in the edge-string.

For more details on vertex- and edge-strings, their properties, the conversion from one to another, their practical use for coding adjacency trees, etc., the reader should consult Section 5.3 of [8]. Let us simply note that for an *ordered* tree, its vertex- and edge-strings are *uniquely defined*.

Now there exist variants of the edge-string. In the case of the adjacency tree, if one knows the colour of the connected component corresponding to the top node of that tree (*i.e.*, the component surrounding all others), and one generally assumes that it is white, then the topological structure of the image is determined by its adjacency tree, where the labels and ordering of nodes are irrelevant. Thus one can replace in the edge-string the two occurrences of a node v by two parentheses (and). Then the edge-string (6) corresponding to the tree of Figure 2 becomes

$$(\,(\,(\,)\,(\,)\,)\,(\,)\,(\,(\,)\,(\,(\,)\,(\,)\,)\,(\,)\,)\,) \tag{7}$$

This parenthesis representation of the adjacency tree was given by Rosenfeld [9].

Let us now explain how the *list coding* of quadtrees can be derived from its edge-string. A quadtree is an ordered tree in which a colour is associated to each leaf node. Given the quadtree $\mathcal{T}$, we make the following modifications to its edge-string $E_{\mathcal{T}}$. For a nonleaf node v, its two occurences in $E_{\mathcal{T}}$ are replaced by two parantheses (and). For a leaf node w, its two occurrences in $E_{\mathcal{T}}$ appear together; we replace then that occurence of $w\,w$ by c, where c is the colour associated to w. It is then easily seen that we obtain in this way the list coding of $\mathcal{T}$.

Again, the list coding of a quadtree is uniquely defined. (In fact, the same holds for the DF-expression, see [5]).

We can finally describe the memory cost of the list coding, compared to that of the DF-expression. Given a quadtree $\mathcal{T}$ having N nonleaf nodes and L leaf nodes, it is easily seen that $L = 3N + 1$. Given the alphabet A consisting of the c colours of the image and of the parantheses (and), the list coding of $\mathcal{T}$ takes $2N + L = 5N + 1$ symbols in A, and each element of A is coded in $\log_2(c + 2)$ bits. On the other hand, the DF-expression of $\mathcal{T}$ takes $N + L = 4N + 1$ symbols in $A' = A - \{)\}$, and each element of A' requires $\log_2(c + 1)$ bits of coding. Thus the ratio between the memory cost of the list coding and the DF-expression is

$$\frac{(5N+1)\log_2(c+2)}{(4N+1)\log_2(c+1)} \approx \frac{5\log(c+2)}{4\log(c+1)},$$

as said above in (3).

III. Implementation of basic image processing operations in LISP

We said in the Introduction that by inserting blanks between the colour numbers and the sublists in the list coding of a quadtree, one obtains a well-formed LISP list; we called it the LISP coding of that quadtree. In fact, as we will explain here, this coding is the natural way to represent quadtrees in LISP.

Recall the recursive definition of the quadtree of an image I given at the beginning of this paper. If all the pixels of the image I have the same colour c, then the quadtree of I is a single node to which one associates that colour c. Here the LISP representation of I can be defined as the atom c. If this is not the case, then one divides I into four subimages I_0, I_1, I_2 and I_3, which will give the children nodes of the one representing I. The LISP representation of this subdivision of I comes naturally as the list $(I_0\ I_1\ I_2\ I_3)$. We repeat the above argument to each I_i, etc., until we get homogeneous regions. Then the corresponding LISP symbolic expression will be the LISP coding of I defined in the Introduction (*i.e.*, the list coding with blanks inserted).

Not only is this coding the natural representation for quadtrees in LISP, but the inherent recursivity of the quadtree structure is well served by the easy implementation of recursivity in that language. As we will see below, many operations on quadtrees can be implemented by recursive functions requiring a few lines of code.

Languages like LISP and PROLOG are highly popular in the AI community, and there have been some implementations of image processing operations in these languages (see for example [2,3]).

Before describing the implementation of these operations, let us make a general remark on quadtrees. Given a $2^n \times 2^n$ image I, if one replace every pixel by a $2^k \times 2^k$ array of pixels having the same colour, then the resulting $2^{n+k} \times 2^{n+k}$ image I' has the same quadtree as I; one can then consider that I and I' represent the same image, with different resolutions 2^n and 2^{n+k} respectively. Thus we will assume that all images have size 1×1, with the pixels having size $2^{-n} \times 2^{-n}$ for a resolution 2^n.

We will now describe the implementation in LISP (in fact in the FRANZ LISP dialect) of several operations for processing quadtrees of *binary* images. (We tested them on a VAX 11/780 running under UNIX 4.2BSD with a FRANZ LISP interpreter). For each operation, we give a brief description of the implementation, and then its LISP code. In order to understand it, the reader should know the rudiments of that language, say the first 4 chapters of [13].

We assume that the colour of a pixel is coded by the two numbers 0 and 1 corresponding to white and black respectively. We have implemented 7 types of operations: basic functions, raster to quadtree conversion, quadtree to raster conversion, set operations, symmetries, shifts, and Euler numbers.

III.1. Basic functions

These are some simple operations on quadtrees which will be used subsequently.

We have the function *normal*, which simplifies the list $(x\ x\ x\ x)$ into x for $x = 0, 1$. We will call this operation the *normalization* of the quadtree; it will be useful in raster to quadtree conversion, set operations or shifting of quadtrees, because one can then obtain nodes whose 4 children nodes are leafs of the same colour.

There is also the reverse function *unnorm*. It will be necessary in order to give the list coding of the quadrants of a quadtree reduced to a leaf, and it will also be used in quadtree to raster conversion.

Then we define *nwqd*, which extracts from the quadtree of an image I the quadtree of the <u>n</u>orth-<u>w</u>est <u>q</u>ua<u>d</u>rant of I. We define similarly *neqd*, *swqd* and *seqd* for the other three quadrants.

Finally, the function *depth* (taken from [13], page 82) gives the number of levels of the quadtree minus 1. It is the logarithm in base 2 of the minimal resolution of an image whose quadtree corresponds to that list.

We get the following code:

```
(defun normal (A B C D)
  (cond ((and (atom A) (eq A B) (eq A C) (eq A D)) A)
        (t (list A B C D)) ))

(defun unnorm (X)
  (cond ((atom X) (list X X X X))
        (t X) ))

(defun nwqd (X) (car (unnorm X)))

(defun neqd (X) (cadr (unnorm X)))

(defun swqd (X) (caddr (unnorm X)))

(defun seqd (X) (cadddr (unnorm X)))

(defun depth (Z)
  (cond ((null Z) 1)
        ((atom Z) 0)
        (t (add (apply 'max (mapcar 'depth Z)) 1) )))
```

III.2. Raster to quadtree conversion

In LISP, a raster image can be represented as a list of rows, where each row is a list of pixels. For example the image of Figure 1 can be represented by the following list:

```
((0 0 0 1) (0 0 1 0) (0 0 1 1) (0 1 1 0))
```

Now the raster to quadtree conversion is achieved by the function *quadtree*. It is based on a repeated application of the function *quad*, which transforms a $2^{n-k} \times 2^{n-k}$ raster whose pixels are LISP codings of quadtrees of $2^k \times 2^k$ subimages (in particular the original raster for $k = 0$) into a $2^{n-k-1} \times 2^{n-k-1}$ raster whose pixels are lists representing quadtrees of $2^{k+1} \times 2^{k+1}$ subimages. The function *quad* is applied successively until $k = n$; then *quadtree* returns the resulting quadtree list, which is the unique pixel of the resulting 1×1 raster. The function *quad* works as follows. Given the raster list $R = (R_1 \ \ldots \ R_{2u})$, it takes two lists, the empty list () and the reverse $(R_{2u} \ \ldots \ R_1)$ of R, and applies to them the function *prequad*, which transforms recursively two lists $(R'_{i+1} \ \ldots \ R'_u)$ and $(R_{2i} \ R_{2i-1} \ \ldots \ R_1)$ into $(R'_i \ \ldots \ R'_u)$ and $(R_{2i-2} \ R_{2i-3} \ \ldots \ R_1)$, until the second list is empty, returning then the first list $R' = (R'_1 \ \ldots \ R'_u)$, which is the new raster.

The transformation of the two rows R_{2i} and R_{2i-1} into the new row R'_i is achieved by the function *group*. It uses a similar process as above. It applies a recursive function *pregroup* to three lists, the empty list () and the reverses of R_{2i} and R_{2i-1}. Here *pregroup* takes the first two elements of the last two lists, combine them through the function *normal*, and puts the resulting normalised quadtree in front of the first list, until the last two lists (or simply one of them) are empty, returning then the first list which is the new row R'_i.

Defining these operations in the correct order, one obtains the following code:

```
(defun pregroup (A B C)
  (cond ((null B) A)
        (t (pregroup (cons (normal (cadr C) (car C) (cadr B) (car B)) A)
                     (cddr B)
                     (cddr C)) )))

(defun group (X Y) (pregroup nil (reverse X) (reverse Y)))

(defun prequad (X Y)
  (cond ((null Y) X)
        (t (prequad (cons (group (car Y) (cadr Y)) X)
                    (cddr Y)) )))

(defun quad (Z) (prequad nil (reverse Z)))

(defun quadtree (Z)
  (cond ((null (cdr Z)) (caar Z))
        (t (quadtree (quad Z))) ))
```

It is easy to check that the computational complexity of the operation *quad* in terms of elementary operations is in $O(N)$, where N is the number of pixels in the raster R. Thus for a $2^n \times 2^n$ image, the total computational complexity of the function *quadtree* is asymtotically linear in $4^n + 4^{n-1} + \cdots + 4^0$, in other words is in $O(4^n)$.

III.3. Quadtree to raster conversion

This operation is the reverse of the previous one, it is achieved by the function *raster*. However, there is one fact to take into account, that the size in pixels of the raster image is not uniquely determined by its quadtree. Thus *raster* admits an additional argument, a nonnegative integer N, which is the logarithm in base 2 of the resolution of the resulting raster. If one wants to take the minimal raster corresponding to that quadtree, then N must be equal to the depth of that quadtree; this leads to the function *minraster*.

The functions corresponding to *pregroup*, *group*, *prequad* and *quad* are *presplit*, *split*, *preras* and *ras* respectively. This gives the following code:

```
(defun nncar (X) (unnorm (car X)))

(defun presplit (A B C)
  (cond ((null C) (list A B))
        (t (presplit (cons (car (nncar C)) (cons (cadr (nncar C)) A))
                     (append  (cddr (nncar C)) B)
                     (cdr C)) )))

(defun split (Z) (presplit nil nil (reverse Z)))
```

```
(defun preras (X Y)
  (cond ((null Y) X)
        (t (preras (append (split (car Y)) X) (cdr Y)) )))

(defun ras (Z) (preras nil (reverse Z)))

(defun inraster (N Z)
  (cond ((zerop N) Z)
        (t (inraster (diff N 1) (ras Z))) ))

(defun raster (N Z) (inraster N (list (list Z)) ))

(defun minraster (Z) (raster (depth Z) Z))
```

Again the computational complexity of the function *raster* is linear in the number of pixels of the resulting raster.

III.4. Set operations

This is very simple: one applies a set operation to one or two quadtrees by applying it to each of its quadrants and normalizing the resulting quadtree, noting that when one of these quadtrees is reduced to a single node, the result of the operation is trivial.

We define four operations: *comp* (complement of an image), *union* (union of two images), *inter* (intersection of two images) and *delta* (symmetric difference of two images). We give here the code for *comp* and *delta*, and we leave the code of *union* and *inter* to the reader:

```
(defun comp (X)
  (cond ((eq X 1) 0)
        ((eq X 0) 1)
        (t (list (comp (car X))
                 (comp (cadr X))
                 (comp (caddr X))
                 (comp (cadddr X))) )))

(defun delta (X Y)
  (cond ((eq X 0) Y)
        ((eq X 1) (comp Y))
        ((eq Y 0) X)
        ((eq Y 1) (comp X))
        (t (normal (delta  (car X) (car Y))
                   (delta  (cadr X) (cadr Y))
                   (delta  (caddr X) (caddr Y))
                   (delta  (cadddr X) (cadddr Y))) )))
```

It is easy to see that the computational complexity of these operations is linear in the number of nodes of the quadtrees used as arguments.

III.5. Symmetries

These are the 8 symmetries of the square: the identity, the rotations by $\pm 90°$, the central symmetry, the diagonal and median symmetries. Each one is achieved by applying it to each quadrant of the quadtree and to the ordering of these 4 quadrants inside the quadtree.

We have labelled the 7 nonidentity symmetries as follows: *rotl* (left rotation, that is counterclockwise rotation of $90°$), *rotr* (right rotation, that is clockwise rotation of $90°$), *symc* (central symmetry), *symv* (symmetry about the vertical median), *symh* (symmetry about the horizontal median), *sympd* (symmetry about the principal diagonal, the one between the NW and SE corners of the image), and *symsd* (symmetry about the secondary diagonal, the one between the NE and SW corners of the image).

We give below the code for *rotl* and *sympd*; we leave to the reader the code of the 5 other non-identity symmetries:

```
(defun rotl (Z)
  (cond ((atom Z) Z)
        (t (list (rotl (cadr Z))
                 (rotl (cadddr Z))
                 (rotl (car Z))
                 (rotl (caddr Z))) )))

(defun sympd (Z)
  (cond ((atom Z) Z)
        (t (list (sympd (car Z))
                 (sympd (caddr Z))
                 (sympd (cadr Z))
                 (sympd (cadddr Z))) )))
```

Clearly the computational complexity of these operations is linear in the number of nodes of the quadtrees used as arguments.

III.6. Shifts

We recall that we assume that all images have the same size; thus the amplitude of a shift will not be expressed in terms of a number of pixels, but of a proportion of the image. A horizontal or vertical shift can thus be defined as a function of three variables: its relative amplitude, the original image to be shifted, and the image with which to fill the portion of the original image which is left vacant by the shift.

For example, a rightwards horizontal shift of an image I can be written as the expression ($rhshift\ p\ J\ I$), where p is some data giving the relative amplitude α of the shift (we will come back to it later), and J is the image whose αth right portion must fill the αth left portion of I which becomes vacant by that shift. Two particular cases are $J = I$ (circular shift) and $J = 0$ (shift with filling by white pixels).

The same can be done with vertical shifts; a downwards vertical shift can be written as (*dvshift* p J I). We can obtain a leftwards horizontal shift *lhshift* as follows: if p corresponds to the relative amplitude α (where $0 \leq \alpha \leq 1$) and p' corresponds to $1-\alpha$, then the leftwards shifted image (*lhshift* p J I) can be obtained as (*rhshift* p' I J). We can also obtain upwards vertical shift *uvshift* from a downwards one.

Let us now describe how we represent the relative amplitude, and how these shifts are computed. Suppose that we want to operate a shift whose relative amplitude is of the form $a_1 2^{-1} + \cdots + a_n 2^{-n}$ (where each $a_i = 0, 1$ and normally $a_n = 1$); then this amplitude can be represented by the list (a_1 ... a_n), with the empty list corresponding to the amplitude 0. The corresponding operations will be called *Lrhshift* and *Ldvshift* (where L stands for list). They can be defined recursively as follows. If L is the empty list *nil*, then the resulting image is I. Assume now that L is not empty. Suppose first that the first element of L is 0; then the list L' obtained from L by removing it represents the double amplitude; now the shift (by L) on J and I is equivalent to a shift of double amplitude (thus by L') on the quadrants of J and I; for example in *Lrhshift*, this will be from (*neqd* J) to (*nwqd* I), from (*nwqd* I) to (*neqd* I), etc.. Suppose finally that the first element of L is 1; one does then as above, but one must precede the application of shift by L' by a shift of amplitude 1/2 from J to I; for example in *Lrhshift* we get (*nwqd* J) instead of (*neqd* J), (*neqd* J) instead of (*nwqd* I), etc. in the formula. This gives thus the following code:

```
(defun Lrhshift (L J I)
  (cond ((null L) I)
        ((zerop (car L))
           (normal (Lrhshift (cdr L) (neqd J) (nwqd I))
                   (Lrhshift (cdr L) (nwqd I) (neqd I))
                   (Lrhshift (cdr L) (seqd J) (swqd I))
                   (Lrhshift (cdr L) (swqd I) (seqd I))) )
        (t (normal (Lrhshift (cdr L) (nwqd J) (neqd J))
                   (Lrhshift (cdr L) (neqd J) (nwqd I))
                   (Lrhshift (cdr L) (swqd J) (seqd J))
                   (Lrhshift (cdr L) (seqd J) (swqd I))) )))

(defun Ldvshift (L J I)
  (cond ((null L) I)
        ((zerop (car L))
           (normal (Ldvshift (cdr L) (swqd J) (nwqd I))
                   (Ldvshift (cdr L) (seqd J) (neqd I))
                   (Ldvshift (cdr L) (nwqd I) (swqd I))
                   (Ldvshift (cdr L) (neqd I) (seqd I))) )
        (t (normal (Ldvshift (cdr L) (nwqd J) (swqd J))
                   (Ldvshift (cdr L) (neqd J) (seqd J))
                   (Ldvshift (cdr L) (swqd J) (nwqd I))
                   (Ldvshift (cdr L) (seqd J) (neqd I))) )))
```

The computational complexity of these two operations is in $O(4^{\ell})$, where ℓ is the length of L. If the maximum of the depths of the images I and J is d (in other words if I and J can be considered as $2^d \times 2^d$ images in terms of pixels) and the shift amplitude is a multiple of the resolution of I and J, then $\ell \leq d$, and so this complexty is at most in $O(4^d)$.

III.7. Euler numbers

The Euler number of a binary image is the number of connected components of the figure (*i.e.*, the set of black pixels) minus the number of holes in it. Of course, it depends on the adjacency relation chosen for black pixels, and so there are two such numbers, corresponding to the 4- and 8-adjacencies.

There exist simple formulas for computing the Euler numbers, which are valid if one assumes that the image contains a white connected component surrounding the rest of the image. If the original image does not satisfy this constraint, then we can surround it by white pixels on all sides; this is done for quadtrees in LISP by the function *zeroborder*.

We will use a formula based on the counting of certain 2×2 configurations, which is found in [10], page 349. Consider the following 3 configurations (up to a square symmetry):

$$\begin{matrix} 0 & 0 \\ 1 & 0 \end{matrix} \qquad\qquad \begin{matrix} 1 & 0 \\ 1 & 1 \end{matrix} \qquad\qquad \begin{matrix} 0 & 1 \\ 1 & 0 \end{matrix}$$

$$(a) \qquad\qquad\qquad (b) \qquad\qquad\qquad (c)$$

Then (a) represents a convex turn of 90°, (b) a concave turn of 90°, and (c) two turns of 90°, which are convex if one considers 4-adjacency for black pixels and 8-adjacency for white ones, and concave if one chooses the reverse adjacencies. Write cvt, cct and dbt for the number of configurations (a), (b) and (c) in the image. As the sum of convex turns minus the sum of concave turns makes $+360°$ on the outer edge of a black connected component, and $-360°$ on the outer edge of a hole, the Euler number is then

$$\frac{cvt - cct \pm 2dbt}{4},$$

where $\pm$ is $+$ if one chooses the 4-adjacency for black pixels and $-$ if one chooses the 8-adjacency for them.

Thus the two Euler numbers (called *euler*4 and *euler*8 in the code) can be obtained by computing the list $(cvt\ cct\ dbt)$ (which can be considered as a vector in LISP). The function giving this vector-type list will be named *eulervec* in the code. It can be computed recursively. Every part of the image contributes to that vector. For a 2×2 configuration of 4 pixels, this contribution is computed by the function *elemvec*, which gives as result one of the vectors (0 0 0), (1 0 0), (0 1 0) and (0 0 1), which are called *nulvec*, *cvtvec*, *cctvec* and *dbtvec* respectively.

Now, for a quadtree Q, (*eulervec* Q) is either *nulvec* (if Q consists in a single node), or it can be obtained by adding the following contributions:

— The application of the function *eulervec* to the 4 quadrants of Q.

— The contribution given by the 2×2 configurations along vertical edges separating those quadrants; it is obtained by the function *vert*.

— The one given by the configurations along horizontal edges separating those quadrants; it is obtained by the function *hori*.

— The one given by the configuration at the common corner of those quadrants; it is obtained by the function *corn*.

These 3 functions *corn*, *vert* and *hori* have easy recursive decompositions. Thus the following code is obtained:

```
(defun zeroborder (Z) (list (list 0 0 0 (nwqd Z))
                            (list 0 0 (neqd Z) 0)
                            (list 0 (swqd Z) 0 0)
                            (list (seqd Z) 0 0 0)))

(setq nulvec '(0 0 0))

(setq cvtvec '(1 0 0))

(setq cctvec '(0 1 0))

(setq dbtvec '(0 0 1))

(defun elemvec (NW NE SW SE)
  (setq SUM (add NW NE SW SE))
  (cond ((eq SUM 1) cvtvec)
        ((eq SUM 3) cctvec)
        ((and (eq SUM 2) (eq NW SE)) dbtvec)
        (t nulvec) ))

(defun corn (NW NE SW SE) (cond
  ((and (atom NW) (atom NE) (atom SW) (atom SE)) (elemvec NW NE SW SE))
  (t (corn (seqd NW) (swqd NE) (neqd SW) (nwqd SE))) ))

(defun vert (W E)
  (cond ((and (atom W) (atom E)) nulvec)
        (t (mapcar 'add (vert (neqd W) (nwqd E))
                        (vert (seqd W) (swqd E))
                        (corn (neqd W) (nwqd E) (seqd W) (swqd E)) )) ))

(defun hori (N S)
  (cond ((and (atom N) (atom S)) nulvec)
        (t (mapcar 'add (hori (swqd N) (nwqd S))
                        (hori (seqd N) (neqd S))
                        (corn (swqd N) (seqd N) (nwqd S) (neqd S)) )) ))
```

```
(defun eulervec (X)
  (cond ((atom X) nulvec)
        (t (mapcar 'add (eulervec (car X))
                        (eulervec (cadr X))
                        (eulervec (caddr X))
                        (eulervec (cadddr X))
                        (hori (car X) (caddr X))
                        (hori (cadr X) (cadddr X))
                        (vert (car X) (cadr X))
                        (vert (caddr X) (cadddr X))
                        (corn (car X) (cadr X) (caddr X) (cadddr X)) )) ))

(defun euler4 (Z)
  (setq Y (eulervec Z))
  (quotient (diff (add (car Y) (times 2 (caddr Y)) ) (cadr Y) ) 4))

(defun euler8 (Z)
  (setq Y (eulervec Z))
  (quotient (diff (car Y) (add (cadr Y) (times 2 (caddr Y)) ) ) 4))
```

We will now describe the computational complexity of these functions. Let d be the depth of the quadtree. The complexity of *corn* is at most linear in d, since it depends upon the depth of its arguments. That of *vert* (or of *hori*) has an upper bound of the form $\varphi(d)$, where $\varphi(x) = 2\varphi(x-1) + ax + b$; it is easy to see that this implies that $\varphi(x)$ is of the form $u \cdot 2^x - vx - w$. Finally, the complexity of *eulervec* is bounded above by a function $\psi(d)$, where $\psi(x) = 4\psi(x-1) + 4\varphi(x) + ex + f$; one checks that this implies that $\psi(x)$ is of the form $m \cdot 4^x - n \cdot 2^x - rx - s$. Thus the computational complexity of the computation of the Euler numbers is at worst linear in 4^d.

IV. Further applications of LISP for quadtrees and conclusion

We leave open the possibility of a more efficient implementation in LISP of the quadtree processing functions described above.

Other operations on binary images represented by quadtrees can be implemented in LISP. We will only give here a few hints about this implementation.

First, one can construct Gargantini's *linear quadtree* [4]. It is a concatenated sequence of labellings of black leaves. It can be defined recursively as follows. We choose a "separator" character X (for example a blank or a carriage return). Let Q be quadtree. If Q consists of a single white node, then the linear quadtree is empty; if Q consists of a single black node, then the linear quadtree consists of the single character X. If the quadtree has more than one node, then consider the four quadrants Q_0, Q_1, Q_2 and Q_3 of the image; for each Q_i, one can construct its linear quadtree, and so each black leaf corresponding to it is coded by a string S; then the coding of that leaf in Q is iS, and the linear quadtree corresponding to

Q is the concatenation of all strings corresponding to leafs. The coding of a black leaf takes the form

$$a_1 \ldots a_n X \qquad \text{with} \qquad a_j \in \{0,1,2,3\} \quad \text{for} \quad j = 1, \ldots, n.$$

Thus the string $a_1 \ldots a_n X$ indicates that the corresponding black leaf is in the a_nth quadrant of the ... of the a_1th quadrant of Q. In particular it is possible to obtain the coordinates and dimensions of black leaves from the linear quadtree.

Such a particular recursive structure allows an easy implementation in LISP. Here the coding $a_1 \ldots a_n X$ can be translated into a list $(a_1 \; \ldots \; a_n)$, which will be called the *black leaf coding list*, and the linear quadtree consists then in a list of such lists corresponding to black leaves. Matters of efficiency (for example a linear computational complexity) may impose the use of auxiliary functions involving several arguments, for example:

— a partial list of black leaf coding lists, to which the other ones must be added;

— the quadtree Q^* under investigation;

— a list of coefficients to be added to the coding lists of black leaves in Q^* in order to obtain the corresponding coding lists in Q.

One can also determine the colour of a pixel. Given a pixel p having coordinates (i,j) in a $2^n \times 2^n$ image, we can code i and j in the same way as the relative amplitude of shifts in the previous section. Thus if

$$i = \sum_{t=1}^{n} i_t \cdot 2^{n-t},$$

then we code it by the list $L[i] = (i_1 \; \ldots \; i_n)$, and similarly for j. Then i_1 and j_1 determine the quadrant Q° in which p lies, and we have then only to perform the search in Q° for the colour of p with the lists $(cdrL[i]) = (i_2 \; \ldots \; i_n)$ and $(cdrL[j])$, and repeat the operation until the quadrant is a leaf in the quadtree. Thus that operation's running time is linear in the depth of the quadtree (if one does not take into account the garbage collector).

The LISP coding of quadtrees is not restricted to binary images. The operations described above can be generalized to grey level images (but one must take care, when grey levels are floating point numbers or "large" integers, to replace *eq* by *equal* in the code, and not to use *atom* to test whether a list represents a leaf, but rather to define a predicate *leaf* for that purpose).

One can also apply our methodology to *octtrees* (the generalization of quadtrees to 3 dimensions) or to what Samet [11] calls *point quadtrees*. This structure differs from the ordinary quadtree (or the *region quadtree* according to Samet [11]) in that the subdivision of a region into 4 subregions is not done into 4 equal squares, but into 4 rectangles determined by the point which is their common corner. Here we can code the point quadtree Q recursively as $(i \; j \; Q_0 \; Q_1 \; Q_2 \; Q_3)$, where (i,j) are the coordinates of the point determining that subdivision (relative to Q), and the Q_i's are the 4 portions of the image determined by that subdivision.

As a conclusion, we will say that the *LISP coding* of quadtrees defined in Section 1 leads to a very easy implementation in LISP of various image processing operations, with a very short source code. It is particularly suited to the case where one is more concerned over simplicity than efficiency. Indeed, for space efficiency, pointerless representations of quadtrees take less memory than LISP lists (in which two pointers correspond to each node), and in this case it is better to use for example Gargantini's linear quadtree. But there are some instances where simplicity matters more than efficiency; we give here two examples:

(1°) *Education*: As most of our code uses only elementary LISP functions, list coding can be useful for the teaching of LISP to students having a background in digital geometry. The implementation and testing of various quadtree transormations are thus interesting exercises. Conversely, it is more attractive to introduce quadtrees to experienced programmers with our simple LISP coding than with an implementation in PASCAL or C requiring long (and uninstructive) declarations in the program.

(2°) *Prototyping*: When studying and testing prototypes of image processing transformations, efficiency is not the highest concern. The most important thing is to be able to modify easily the source code. This one must thus be the simplest possible.

For the processing of images in the quadtree form, the corresponding image data could be stored in memory either under the form of the *DF-expression* or by the *list coding* (without blanks), while it would be processed in the form of LISP lists; in this respect it would not be hard to devise a program translating one form into another, providing thus a simple interface. In this way, one would satisfy both requirements of compact memory storage and computationally easy processing.

Acknowledgement

The author thanks his colleague M. Vauclair for his helpful comments on the presentation of the manuscript.

References

[1] O.P. Buneman: A grammar for the topological analysis of plane figures. In *Machine Intelligence 5.* (B. Meltzer & D. Michie, eds.), pp. 383–393, Edinburgh University Press, Edinburgh, 1969.

[2] J. Camacho Gonzalez, M.H. Williams, I.E. Aitchison: Evaluation of the Effectiveness of Prolog for a CAD Application. *IEEE CG&A*, pp. 67–75, Mar. 1984.

[3] S. Edelman, E. Shapiro: Quadtrees in Concurrent Prolog. *Report CS84-19*, Dept. of Math., Weizman Institute of Science, Aug. 1984, *revised* Jan. 1985.

[4] I. Gargantini: An effective way to represent quadtrees. *Comm. ACM*, Vol. 25, no. 12, pp. 905–910, Dec. 1982.

[5] E. Kawaguchi, T. Endo: On a Method of Binary-Picture Representation and Its Application to Data Compression. *IEEE Trans. Pattern Analysis & Machine Intelligence*, Vol. PAMI-2, no. 1, pp. 27–35, Jan. 1980.

[6] E. Kawaguchi, T. Endo, J.-I. Matsunaga: Depth-First Picture Expression Viewed from Digital Picture Processing. *IEEE Trans. Pattern Analysis & Machine Intelligence*, Vol. PAMI-5, no. 4, pp. 373–384, Jul. 1983.

[7] M.S. Parsons: Generating Lines Using Quadgraph Patterns. *Unpublished Manuscript*, University of Kent at Canterbury, May 1985.

[8] C. Ronse, P.A. Devijver: *Connected Components in Binary Images: the Detection Problem.* Research Studies Press, Letchworth, Hertfordshire, England, 1984.

[9] A. Rosenfeld: Adjacency in digital pictures. *Information and Control*, Vol. 26, pp. 24–33, 1974.

[10] A. Rosenfeld, A.C. Kak: *Digital Picture Processing.* Academic Press, New York, 1976.

[11] H. Samet: The Quadtree and Related Hierarchical Data Structures. *Computing Surveys*, Vol. 16, no. 2, pp. 187–260, Jun. 1984.

[12] M. Tamminen: Comment on Quad- and Octtrees. *Comm. ACM*, Vol. 27, no. 3, pp. 248–249, Mar. 1984.

[13] P.H. Winston, B.K.P. Horn: *Lisp*, 2nd. edition. Addison-Wesley, Reading, Mass., 1984.