CHINESE CHARACTER PROCESSING

by

Yeung Chuen-sang

(楊 荃 生)

A Master Thesis submitted in partial
fulfillment of the requirements for the degree of
Master of Philosophy
in
The Department of Electronics
The Chinese University of Hong Kong

Hong Kong
May 1987

ACKNOWLEDGEMENTS

I should like to express my deepest gratitude towards my supervisor, Dr. W.K. Cham for his patient and invaluable guidance throughout the course of this work.

Thanks are also due to Dr. H.T. Tsui for his helpful comments on the topics.

ABSTRACT

Chinese characters are characterized by complex internal structures with strokes as basic structural elements. As the number of basic strokes that can be combined to form the complete character set is small, the availability of stroke information can considerably simplify many processing problems associated with Chinese characters. A system is described that can extract stroke information from binary Chinese character patterns. Demonstration is made on utilizing this information to achieve good quality scaling and font transformation.

TABLE OF CONTENTS

1.0 INTRODUCTION

An extensive amount of works has been reported that tries to bring Chinese characters interface with digital computers practical. The large number of distinct characters and the complexity of the structure of these characters have imposed great difficulties to these attempts. These difficulties are evident from the fact that native writers typically took years to master by rote a subset, around 3000, of the characters so as to be able to communicate adequately through them; and the current effort in Mainland China to simplify the characters so that their use may be easier. Unfortunately, at this moment, even the rule of lexicographical ordering of these characters in the dictionary is not too satisfactory. These, in the context of computer processing of Chinese characters, create problems in several areas: storage, input method, recognition and generation. Numerous schemes on these have been proposed but few has yet received universal acceptance.

In this report, attempt is made to solve the problem of recognizing strokes, which are the basic structural elements of all Chinese characters, from Chinese character patterns; and demonstration is made on how this information is able to ease the problems of scaling and font conversion. A system that can extract stroke information from binary Chinese character patterns, and achieve scaling and font transformation, has been constructed. The system is divided into two parts. The first part is a three-level stroke extraction process that extracts stroke information from Chinese character patterns. The second part is on scaling and font transformation which operates on individual strokes and combines the results to form characters that are properly scaled and/or of different font type. The first part of the work is a limited form of character recognition while the second part, scaling and font conversion, belongs to the category of character generation.

A survey of published works as an introduction to the topic will start the presentation, which is followed by a general description on the objectives of the work and the approaches taken. The resulting system is then described in detail. The results are presented in the chapter that follows. Finally, discussions are made on the results and other related matters.

The chapter classification follows the above mentioned sequence with Chapter Two devotes to general introduction to Chinese character processing through a brief survey of related works. Chapter Three gives a general introduction of this work. The implementation details are given in Chapter Four to Chapter Seven. Chapter Eight and Nine presents the testing result and an overall discussion.

2.0 INTRODUCTION TO CHINESE CHARACTER PROCESSING

2.1 Introduction

This chapter has two purposes: to provide background information on Chinese character processing and to introduce the scope of our work in this context. The first section will give a survey emphasizing on Chinese character recognition, scaling and font transformation. The second section will give a brief introduction to our work to be described in this report.

2.2 A Survey

This section presents a brief survey on the published papers and texts on topics that have a direct or indirect relations with and influence on our work. The survey will be divided into two parts: topics on Chinese character recognition and topics on Chinese character generation. Works on coding and compression on Chinese characters were well summarized by Nagao [30] and therefore will not be repeated here. Some of the publications mentioned below may seem remote to our work but are included as they carry important topics which can help make a better presentation.

2.2.1 Modelling of Chinese Characters

By modelling of Chinese characters means a formal description of them from their pictorial structures. As discussed by Stallings [38], modelling of Chinese characters is very difficult, given the great deal of structure and irregularities, but is important because the knowledge of the structure of Chinese characters may contribute to their mechanization and recognition. Numerous works have been reported in this area [38]. Although these works all hoped to generate all characters in use, they faced the problem that these methods would also, as a by-product, generate non-characters. In addition, the normally accepted stroke sequence of which a character is drawn by native writers is in not predictable from most of these methods of representation.

The lack of a good formal description forces researchers to use a large number of different methods in building practical devices or developing processing systems. For example, the number of proposed searching and indexing methodology for use in digital computers are large; the number of input devices for typesetting, typewriting, or computer usage are numerous [38,44].

Clearly the problem is non-trivial, it appears that a problem of such a scale, in which tens of thousands of Chinese characters are involved, as is many other real life problems of considerable complexity, that no simple elegant model can be found to handle all the cases that can arise. Rich [33] listed a number of problems to show that to solve these problems, an often voluminous, may be difficult to characterize accurately, and in some cases constantly changing, knowledge base is often required. This can be easily extended to the case of modelling of Chinese characters. The number of Characters are large; the number of font types are increasing; and the characters are changing, most by several deliberate efforts in history. One may then argue that an expert system with considerable artificial intelligence might be needed in order to be able to handle the difficult task of generating accurate Chinese characters with a generally accepted stroke sequence.

2.2.2 Chinese Character Recognition and Analysis

It has long been recognized that to input information to a computer with Chinese character as a medium is handicapped by the lack of uniformity and the lack of a satisfactory lexicographical ordering. Before the computer era, mechanicals device to convert key strokes to printed character, a counter-part of our daily used typewriters, may have as many as thousands of keys. At this stage, numerous clever schemes, some have already been applied in some commercial products, have been proposed to use the ordinary keyboard when interfacing with a computer. But these methods are not terribly convenient. Clearly, one of the alternatives is to let the machine recognize them by optical or other means. This has become an important subject in computer processing of Chinese characters.

There was a number of approaches to solve the difficult problem of Chinese character recognition, and the work may be separated into three groups that deal with three forms of input characters: printed characters, hand-written characters, and on-line hand-written characters. In general, they all deal with characters in digital form but stroke input time-spatial sequence is also available for the on-line hand-written type of input method. The methods employed to recognize these forms of input characters vary from system to system. Some of them are described in the following. See Stallings [37] for a more detailed treatment.

One early work on Chinese character recognition is by Casey and Nagy [3] who first reported the recognition of printed characters by template matching. In their method, a given digitized sample of a printed character is compared with templates of characters stored in advance. To improve processing efficiency, a two-stage matching process is used. Some other methods also by pattern matching have also been reported [10,35]. Other reported approaches include peripheral feature [21], transformation algorithm [45] and boundary belt patterns [11]. Recently, Umeda [43] reported a very successful recognition method for multi-font printed Chinese characters. The method used a combination of mesh feature, peripheral feature, and some discrimination procedures to achieve the high recognition rate reported.

As to On-line hand-written Chinese characters, Groner [14] reported an experimental system that utilizes sequential positional information to recognize hand-written chinese characters. Another method also by time-spatial information on on-line hand-written input of Chinese characters is also reported [50].

Regarding hand-written Chinese characters recognition, a large number of groups attempted the problem by many different techniques such as by pattern matching [25,42], periphery of a character [46,47], cellular feature [32], structure feature concentration [1,18], and probabilistic modeling [22,23,24], and some others [28,52]. Most of these works divided the task into two or more levels to improve the efficiency and attempts were made to utilize features that were found to be the least variant in most conditions and were easier to extract. The features chosen, however, in many cases do not correspond to the natural basic structural units that constitute a Chinese character. In this regard, although sufficient in achieving the goal of recognition, their contribution to the mechanization of Chinese characters are limited by the lack of generality.

Other than those mentioned above, there are works that deal directly with the basic structure of Chinese characters, some of which have been applied to character recognition. [2,16,17,37,38] are examples of this kind of approaches which attempt to classify, analyze or encode Chinese characters by the inherent structural information. Stallings [38] developed a scheme based on a two-level representation of the structure of a Chinese character. A character is considered to be composed of a two-dimensional arrangement of stroke segments. A tree of graphs may thus be generated and coded to represent a character and the information is used to recognize them. This method as implemented may not be as efficient as those above but did address the area of structural information and its representation. The group led by Hsu [16,17] attempted to extract stroke information from higher resolution character patterns. The work was an extraction of the skeleton of a character which is, however, remote to its title that implies stroke classification.

To summarize, the choice of method is generally determined by processing efficiency consideration. It appears that there are a large number of features that are usually easier to deal with than strokes as basic classification elements, and a two or three level classification scheme normally is sufficient to make the approach feasible. In fact, extracting the basic structural information such as stroke from character patterns, though desirable as this information may be used for many purposes other than solely for recognition, in terms of processing efficiency and the ease of the method, is not preferred when the purpose is to recognise characters. However, stroke, as a natural basic structural element of Chinese characters, is an important information in many facets of Chinese character processing ranging from storage compression to font generation. Therefore, even though they are difficult to deal with, and may not be the best choice as a classification element, attempts have still been made to extract them out of Chinese character patterns as is evident from the papers mentioned above.

2.2.3 Scaling, Font Generation and Transformation

There exists only a few papers in the category of font generation and transformation, probably because in this level of technologies where computer with Chinese character as a practical interface is just beginning to reach the commercial sector, it is generally true, except for acedemic interest, that lower quality output characters are considered acceptable in many applications such as in game graphics and non-business oriented word processing systems; and this may be partly attributed to the fact that recognition problem brings more immediate fruits than area that partly involves aesthetic which is difficult to quantify and thus evaluate. Knuth [19] made the historic attempt and set a new direction in automatic font generation; and naturally, the principle is extended to Chinese character font. Several systems [7,15,26] based on similar idea have been reported. All of them had special routines to construct basic strokes; and characters are synthesized using these strokes. All these systems produced very high quality fonts. Similar to font generation, scaling is a less noticed area as brute force may be all that is required for a temporary solution, and larger systems can afford to store a large database of character patterns sufficient for their particular applications. Except possibly in very special situation as that reported by Casey [4,5] that the scaling problem was investigated.

Apart from those mentioned above, smaller systems employing different methods were also reported [6,13,47] but gave low quality character output. In [31], the problem of font transformation was tackled by a very simple store-the-difference approach which was of course very limited. Shiono [36] proposed to detect and change some special features to achieve the effect. The result, however, was not too satisfactory.

2.3 Scope of Our Work

Following the concept by Stalling [38] who views a character pattern as a two-dimensional graph and stroke segments of the character pattern are traced out directly from the pattern so that a graph can be constructed, a stroke extraction scheme by a similar but unrelated trace procedure with additional classification and description capability is chosen as part of the investigation, which corresponds roughly to the area of character recognition. The technique may be named as analysis-by-synthesis. The meaning of which will become clear after the detailed presentation. The other part of the investigation is on scaling and font transformation, which corresponds to the area of character generation.

3.0 GENERAL DESCRIPTION OF THE SYSTEM

3.1 Introduction

As the field is yet considered mature, and as pointed out by one author [37] that many techniques from different disciplines may be required to solve the difficult problem of Chinese character processing. There is no clear method that can claim superiority in performance and solve the large number of conditions that may appear given the multitudinous of characters and their complex structures. We have chosen here to deal with strokes in character patterns as the generality of this information makes it possible to be applied to a wide range of problems. An analysis scheme has been developed to extract stroke information from character patterns. The job involves classification as well as description. We have then used scaling and font transformation to show that stroke information can simplify the scaling algorithm, and guide the use of pre-stored patterns to form strokes so that character patterns of a different font can be obtained. The source of the character patterns is a library of binary character patterns of resolution 24x24.

3.2 Stroke Extraction

There are traditionally two main methods, according to Fu [9], that are used in recognition: decision-theoretic approach, of which statistical approach is an example; and structural approach. The former is appropriate when the problem is primarily one of classification and explicit structural information about the pattern is not considered important. The latter is required when the pattern is rich in structural information and the problem requires classification as well as description. In practice, a combination of the two may somethimes be necessary to bring about a practical system. Returning to our target of stroke extraction, with Chinese characters rich in structural contents, a system of the structural type is obviously more appropriate. Note that it is generally difficult to express the structural relation, that must be used to make decision in a recognition process, among the structural elements in a Chinese character without resorting to linguistic definition, which therefore dictates a structural approach.

In our approach, a character is viewed as a graph where the links are stroke segments and the nodes are where stroke segments start or end, or where they connect. A stroke may consist of one or more connected stroke segments. A three-level system is employed to identify strokes in a character pattern. The lower two levels involve primitive extraction and selection. The highest level involves structural analysis. This may be called an analysis-by-synthesis as the last level is mainly a synthesis process. Knowledge on internal structure of typical Chinese characters is used to guide the analysis.

The stroke extraction stage is divided into three levels. The input to the first level, Level One, is a binary Chinese character pattern; current implementation expects a resolution of 24x24. On output, this level produces a group of data describing the stroke segments. The second level, Level Two, performs data abstraction on the data from Level One. The data produced by Level Two are a list of symbollic quantities describing the stroke segments. The third level, Level Three, is an analyzer that is responsible for selectively combining these stroke segments into strokes.

The three-level stroke extraction part corresponds roughly to a subset of ordinary recognition system [9]. The system in this case is simplified by eliminating the usual noise removal stage, which is not implemented as the pattern is assumed noise free, an assumption found later to be only partially true. In order not to be overly ambitious, the implementation limits the recognition to stroke level only. To extend the recognition to a whole character, more powerful classifier would be needed.

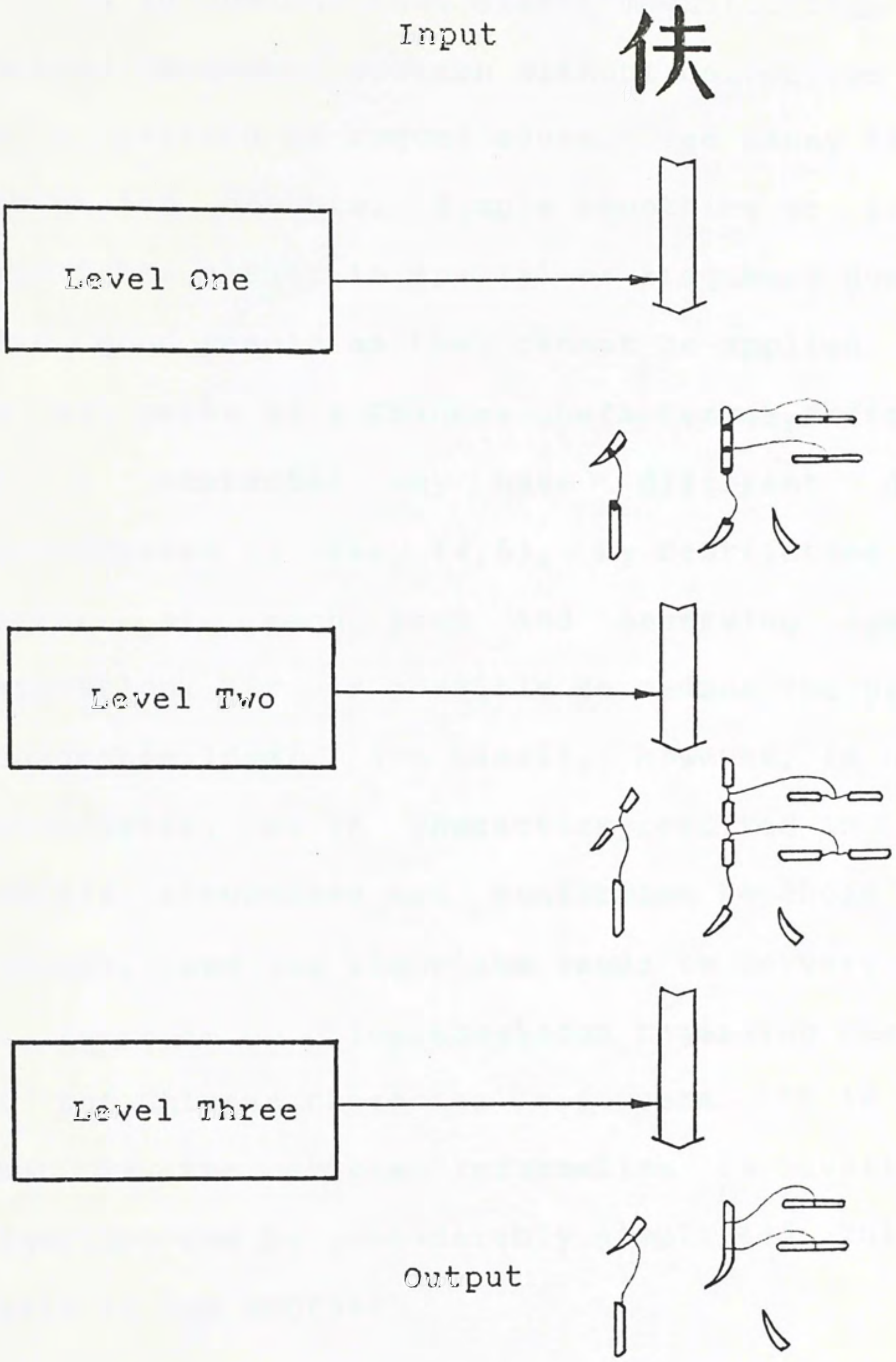


Figure 3.1 Stroke Extraction Process

3.3 Scaling

It is obvious that direct magnification of a binary Chinese character pattern without correction will result in a pattern of rugged edges, see Casey [4,5] for a discussion of this. Simple smoothing or interpolation algorithms either in spatial or frequency domains do not give good result as they cannot be applied universally to all parts of a Chinese character as different parts of a character may have different demand. As demonstrated by Casey [4,5], by restricting the scaling factor at each step and observing symmetry and separation, it is possible to reduce the problem to a manageable level. The result, however, is not entirely satisfactory as the characters resulted in some cases exhibit structures not conforming to those considered correct, and the algorithm tends to be very complicated as there is no prior knowledge regarding the structure of the Chinese character in concern. It is speculated that if the stroke information is available, the algorithm can be considerably simplified. This forms the basis of our approach.

In our approach, we utilize the stroke information to guide the scaling process. The scaling algorithm as implemented is rather simple as the stroke information supplied made such approach feasible. It works in spatial domain and is in a form of a universal operator that examines a number of the adjacent pixels to determine the outcome of a particular pixel in the scaling process. The algorithm operates on individual strokes one at a time.

3.4 Font Conversion

A very popular approach to generate high quality character is fitting and joining high order curves such as cubic splines which define the boundary curve of a stroke. This method requires reference points that may not be on the strokes. Modification of these points in order to generate a different font is difficult without human assistant as this is generally a trial-and-error process and involves aesthetic judgement of which no general guideline can be found. Another method which we have chosen may be more suitable for lower resolution application in which a set of patterns are prestored and combined when needed to produce strokes which in turn produce characters. This method is not as powerful and elegant as the equation method but the complexity of which is reduced and is generally faster as the amount of computation is much smaller as oppose to a computational intensive curve fitting approach. In addition, only a window is all that is needed to define the size and position of a stroke. The human assistance part is implicitly shifted to pattern construction and is done once and for all without further intervention to the process.

The transformation works by replacing the orginal pattern by a combination of prestored patterns selected according to stroke information extracted from the original pattern. The size and position of a stroke is determined also from the source pattern. Each stroke is constructed by one or more of the more foundamental subpatterns found common in many strokes. The list of strokes and the foundamental subpatterns will be shown in Chapter Seven.

A major limitation of the this method is that it is capable of achieving font transformation where the relative positions of strokes of a character do not require adjustment. This of course limits the number of fonts that may be produced. However, as there lacks good parameters that allow aesthetic values be quantified and thus form the basis of adjustment, it is difficult to devise method to allow transformation be extended to those fonts that require a stroke position shift, and let alone those requiring a major structural modification. For these, at this stage, human assistance seem to be the only alternative. Based on this, our system will be limited to those fonts without a need of stroke position adjustment.

4.0 STROKE EXTRACTION LEVEL ONE

4.1 Introduction

This is a primitive extraction stage. The primitives of the input Chinese character pattern are chosen as subpatterns that are portions of a stroke. They are called stroke segments here. A trace algorithm has been devised for this purpose. This chapter describes in detail this algorithm.

In finding a method to extract stroke segment information from a character pattern, several factors have been considered that determined the final approach. They are the following.

1. Each traced stroke segment should correspond to a stroke as much as possible as this would simplify the design of other levels.

2. The intersaction area should be investigated in such a way that when two segments of similar inclination are joined at an intersaction area, for example, when two straight horizontal segments are joined together later possibly to form a stroke that is both straight and horizontal at Level Three, the pixels reported for these two segments, should the connection take place, should form a

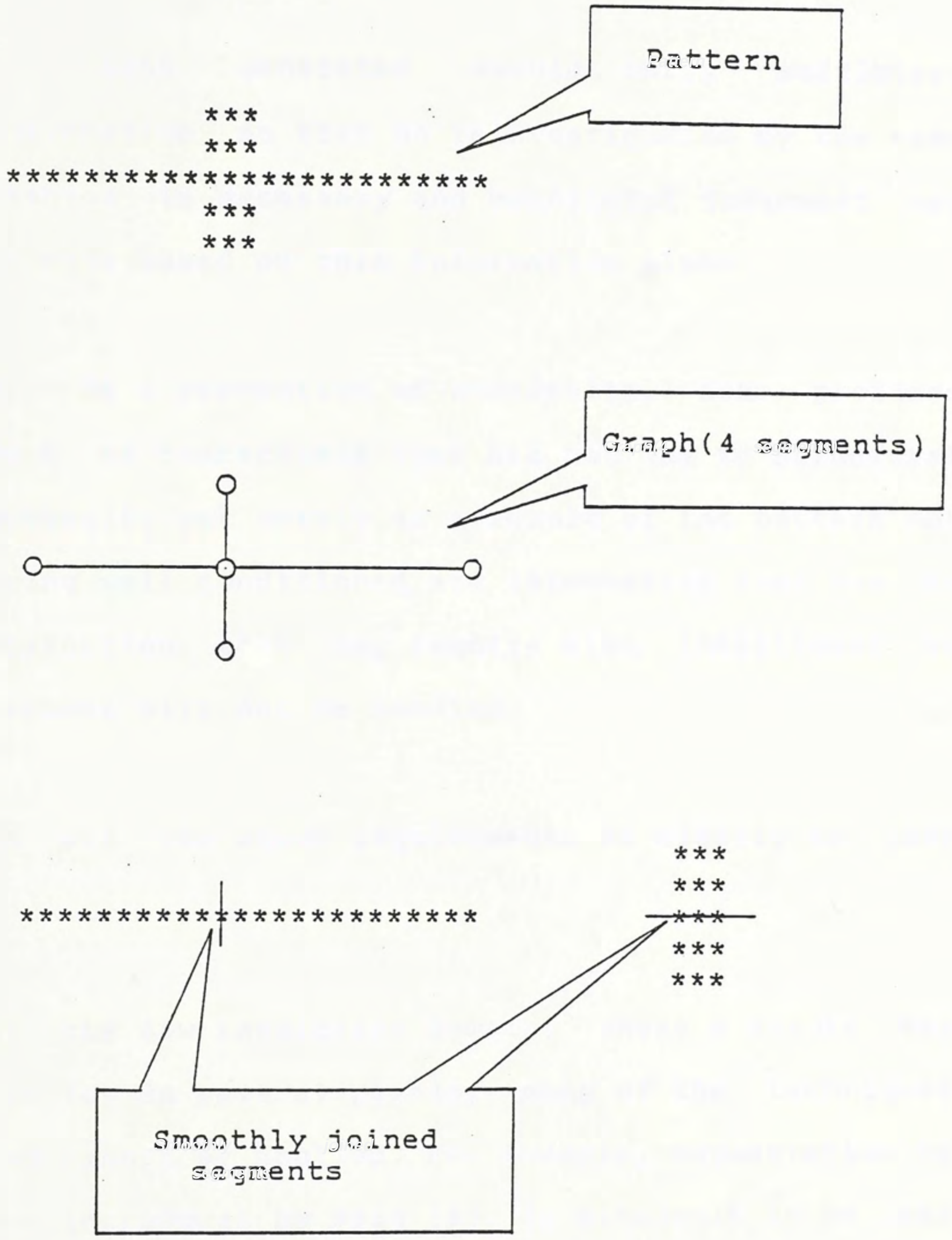


Figure 4.1 Smooth Stroke Segment Connection

good, that is, a smooth joint at this area (Figure 4.1). This is necessary so as to ensure the information generated is sufficient for completely defining the stroke shape.

3. Data generated should carry sufficient information so that no reinvestigation by the same fashion is necessary and meaningful judgement may be made based on this information alone.

4. As a prevention of overdesign, some problems such as connections that are not due to structural necessity but merely as a result of the pattern not being well conditioned and information lost due to distortion that may require high intelligent to recover will not be handled.

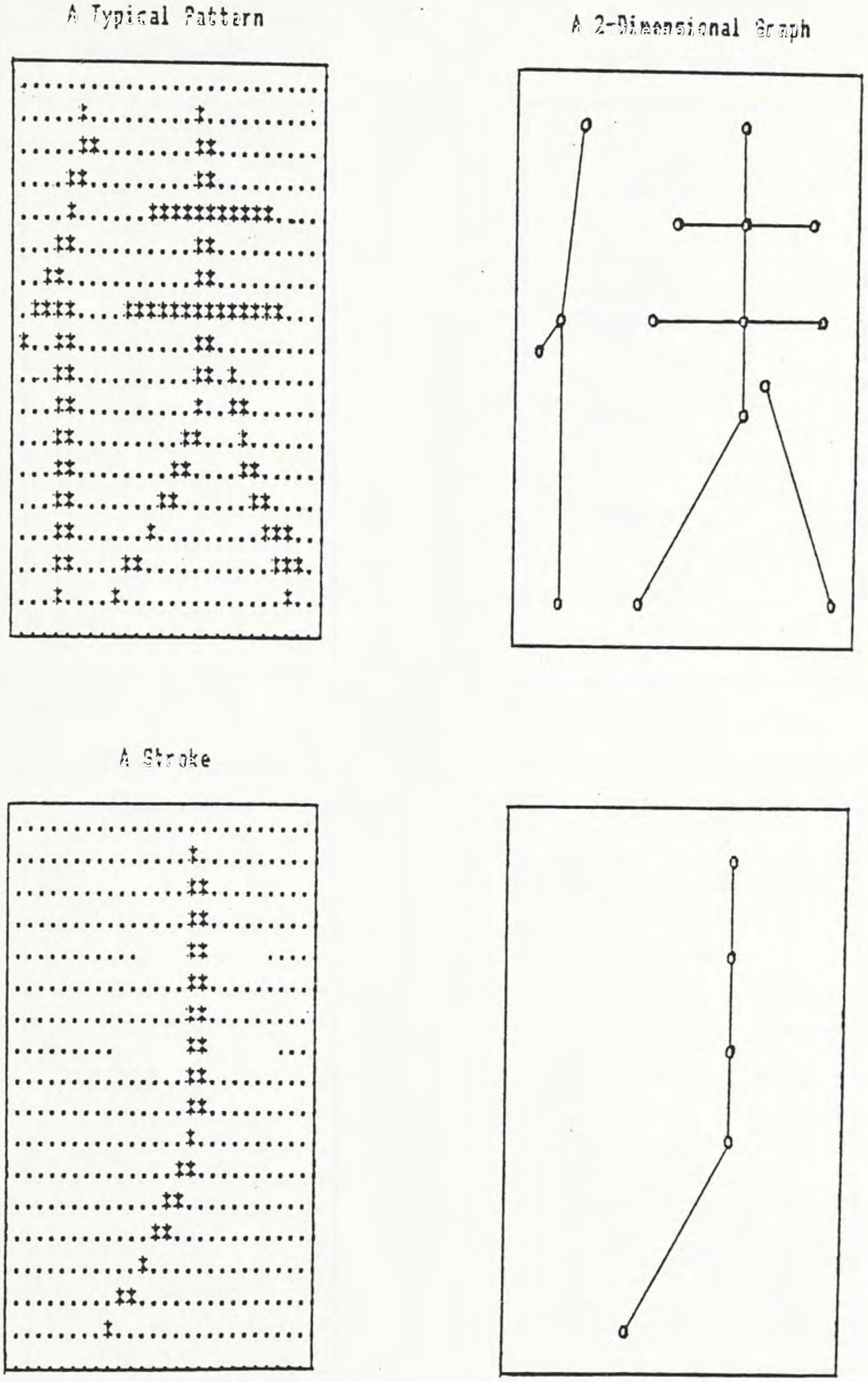
Meeting all the above requirements is clearly no easy task.

At the low resolution domain, where a stroke may have as few as several pixels, many of the techniques reported cannot be applied. For example, segmentation by polygons introduced by Feng [8], is difficult to be used here. The method reported by Stallings [37] may be useful in identifying the graph representing a character, however, the information generated is insufficient to meet requirements two and three. In general, Chinese characters have the following characteristics that, if properly utilized, might be helpful to achieve our goal: as a character is formed by strokes, and the way the stroke is written is mostly from high to low, a trace following the same path may be able to use the past to predict what is after a joint so that the trace may be continued as this will ensure requirement two be met; although some of the customally defined strokes in dictionary are quite complex and are produced with multiple brushes of different orientation, a simpler set of strokes may be defined so that this type of situation can be, to certain degree, avoided, which may then make requirement one meaningful. Note that this simplifies mainly later levels. Based on the above, the following method is devised.

4.2 Detail

The input to this part of the system is a binary matrix containing a Chinese character pattern of resolution 24x24. The pattern is assumed noise free and is positioned upright. The character represented by the pattern is viewed here as a two dimensional graph. See Figure 4.2. A node of the graph corresponds to an end point of a stroke, a cross between two or more strokes, or where a stroke changes inclination. A link of the graph corresponds to the body, or part of the body of a stroke. By this classification, a stroke segment is a link together with its corresponding two end points; a stroke therefore consists of one or more stroke segments. Arbitrarily, one end that is higher in position or is to the left if the stroke segment orients horizontally is called a Head, the other a Tail while the link, or the body, is called a Trunk. See figure 4.3. On output of this part of the system is a table called a sequence table with each entry as a sequence. A sequence is defined as a group of pixels together they form a part of a stroke, or joined strokes, having a particular orientation, as shown graphically in figure 4.3a. In general, a sequence may consist of one or more stroke segments and is obtained by a pattern following process called a trace process, to be described later.

Each pixel of the pattern on input has only one of the following two states: Zero or One, with Zero meaning not occupied by the character in a character pattern matrix, and One, also named Black, meaning the opposite. During the trace process to be described, each Black pixel may be changed into two other states: Processed or Special. These additional states are necessary because some pixels may be reprocessed and these additional states serve to prevent confusion. State Special is used for those pixels identified as belonging to nodes.



A Typical Pattern

A 2-Dimensional Graph

A Stroke

Figure 4.2 A Chinese Character Pattern as A Graph

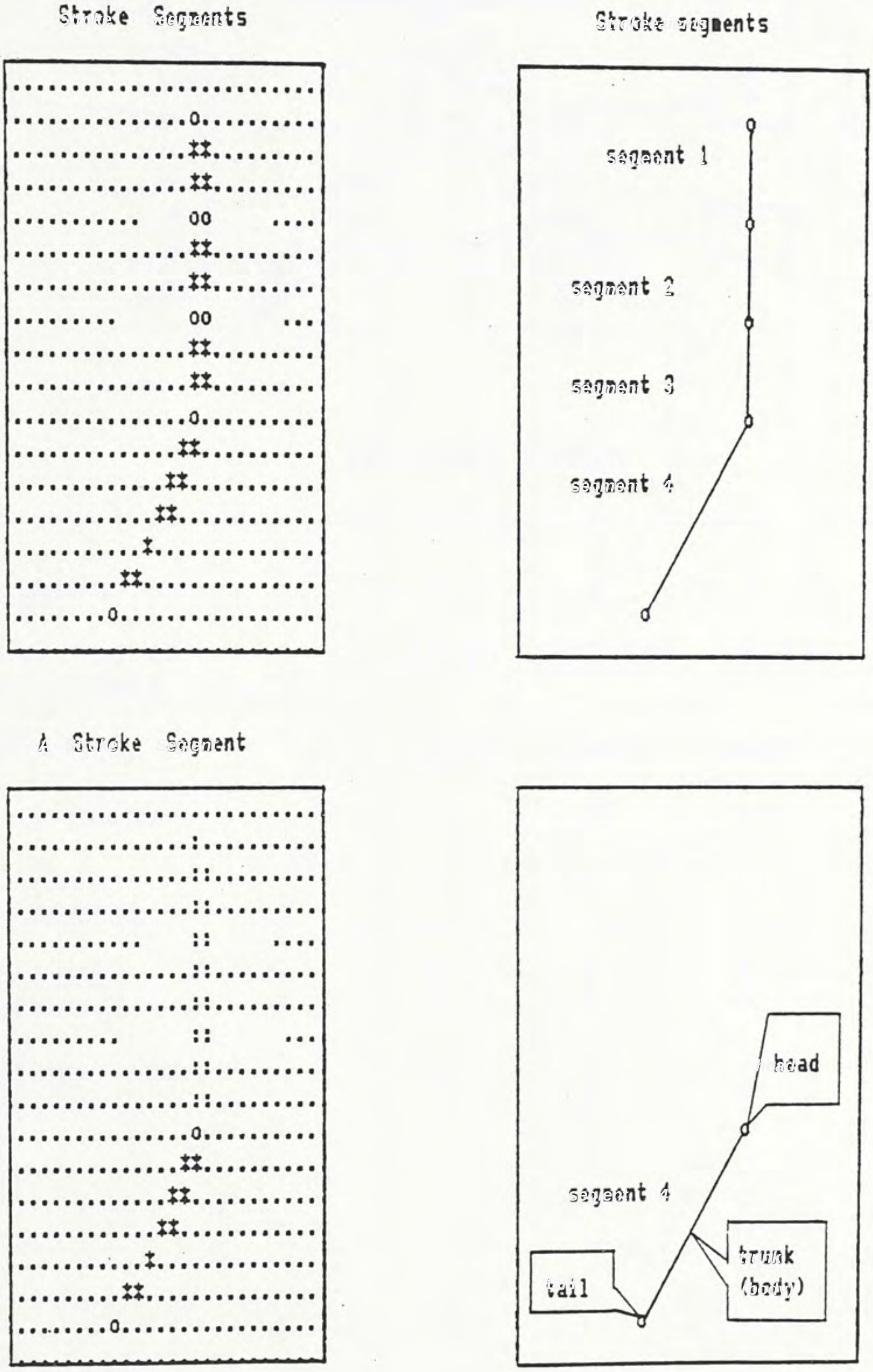
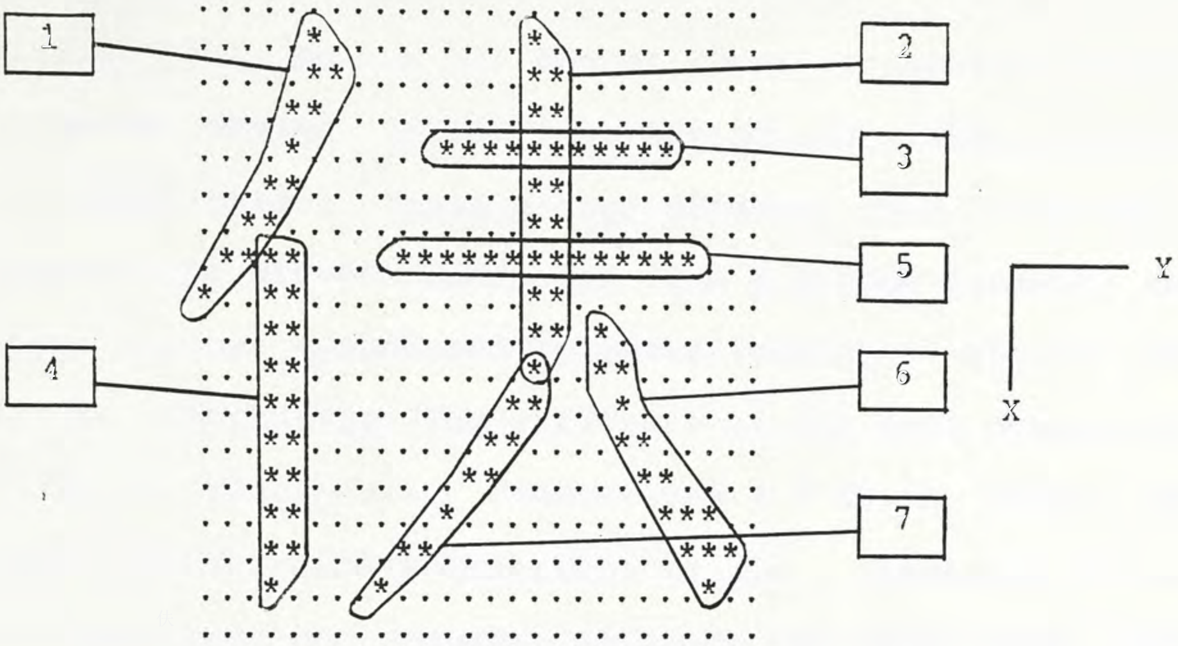


Figure 4.3 Head, Tail and Body of A Stroke



SEQUENCE TABLE

Sequence #	1	2	3	...
Location of Ref. Pixel (x,y)	(2,6)	(2,16)	(5,12)	...
Direction	south	south	east	...
# of Units	8	10	11	...
Width(1)	1	1	1	...
Offset(1)	—	—	—	...
Width(2)	2	2	1	...
Offset(2)	0	0	0	...
Width(3)	2	2	1	...
Offset(3)	-1	2	1	...
.	.	.	.	
.	.	.	.	
.	.	.	.	

Figure 4.3a Sequences and Sequence Table

Each trace process is to obtain data of a group of pixels that belong to one or more connected stroke segments having similar inclination and width variation characteristics. These groups of data, each is called a sequence to prevent confusion from a stroke segment, may then further processed by later levels to obtain the stroke information. The structure of the main program is shown in Figure 4.4. From Figure 4.5 which shows the operation of a very important module - Scanner, it can be seen that the sequence of trace may be divided into several steps.

1. Locate pixels to start tracing.
2. Determine trace direction.
3. Trace to get a data sequence.
4. Insert data to table.

The sequence is repeated until no more suitable pixels can be found. In the following, the important modules involved will be described individually.

The main module is the Scanner. It scans the binary pattern from top to bottom, left to right to locate two types of pixels, Black and Special, and reports the coordinates of which. Additional constraint must be satisfied when the pixel is of type Special: the pixel or its surround pixels of the same type must be connected to pixel of type Black for it to be accepted, as the opposite will mean this pixel is not appropriate

Program Level One

Input

A binary Chinese character pattern of resolution 24x24.

Output

A table called sequence table (Figure 4.3a) with each entry holding information of a sequence of pixel strings, each of which is called a unit (Figure 4.10), representing one or more stroke segments.

Method

```
     Definition --- Scanner : a routine, see text.

     Begin
1.     Load a Chinese character pattern.
2.     Scanner.
3.     Save sequence table.
4.   End of program.
```

Figure 4.4 Program Level One

Procedure Scanner

Input

A 24x24 Chinese character pattern with each pixel having one of the following two states: 1 and 0 with 1 (Black) for those occupied by the character.

Output

A table called a sequence table with each entry being a sequence of units of pixels collected in a trace process.

Method

```
Definition --- R : size of the two axes of the
                   character pattern matrix
               B : pixel type = Black
               P(I,J) : pixel at coordinates (I,J)
               D : direction of trace
               T : portion of a character pattern
                   corresponds to one or more stroke
                   segments, called a sequence
               A : sequence table
               S : pixel type = Special
               I,J : variables as counters for loops
               Starter, Tracer : routines, see text.
```

```
1.     For I := 1 to R do 2 to 8
         Begin
2.         For J := 1 to R do 3 to 8
             Begin
3.             If P(I,J) = B then
                 Begin
4.                 Select D by calling Starter, trace in
                     direction D by Tracer to get T.
                   Enter T into A.
                 End.
5              Else if P(I,J) = S then
                 Begin
6.                 Find pixel = B connected to P(I,J),
                     or its neighbours of same state.
7.                 If pixel = B located then
                     Begin
8.                     Do 4.
                     End.
                 End.
             End.
         End.
9.     End of Procedure.
```

Figure 4.5 Procedure Scanner

for starting a trace. A routine will confirm whether this condition is satisfied. If no more qualifying pixel can be found, the trace process is declared completion.

After Scanner reports the coordinates, a module called Starter will determine the trace direction. A trace direction is defined as the direction where further pixels are to be located. Obviously, the trace direction will exclude those going upward. The remaining question is how fine should the two quadrants, from east clockwise to west, be devided? One of the candidates is to divide the two quadrants into five parts as ranged by degrees, 90-130, 131-160, 161-200, 201-230, 231-270 (Figure 4.6). This would nicely fit with the general stroke direction. However, the two divisions at 131-160 degrees and 201-230 degrees would make the definition of stroke length and width difficult. Since there is no simple solution, the number of trace directions is arbitrarily reduced to three: south, east and west, and is found to be adequate. The possible cases of inclined stroke segments are then handled during the trace process by monitoring the positional offsets (see below). Note that the direction thus defined is also for defining the length and width of a stroke but has nothing to do with the detailed inclination. The detailed inclination will be computed in Level Two based on the positional offsets. This, therefore, simplifies the Starter.

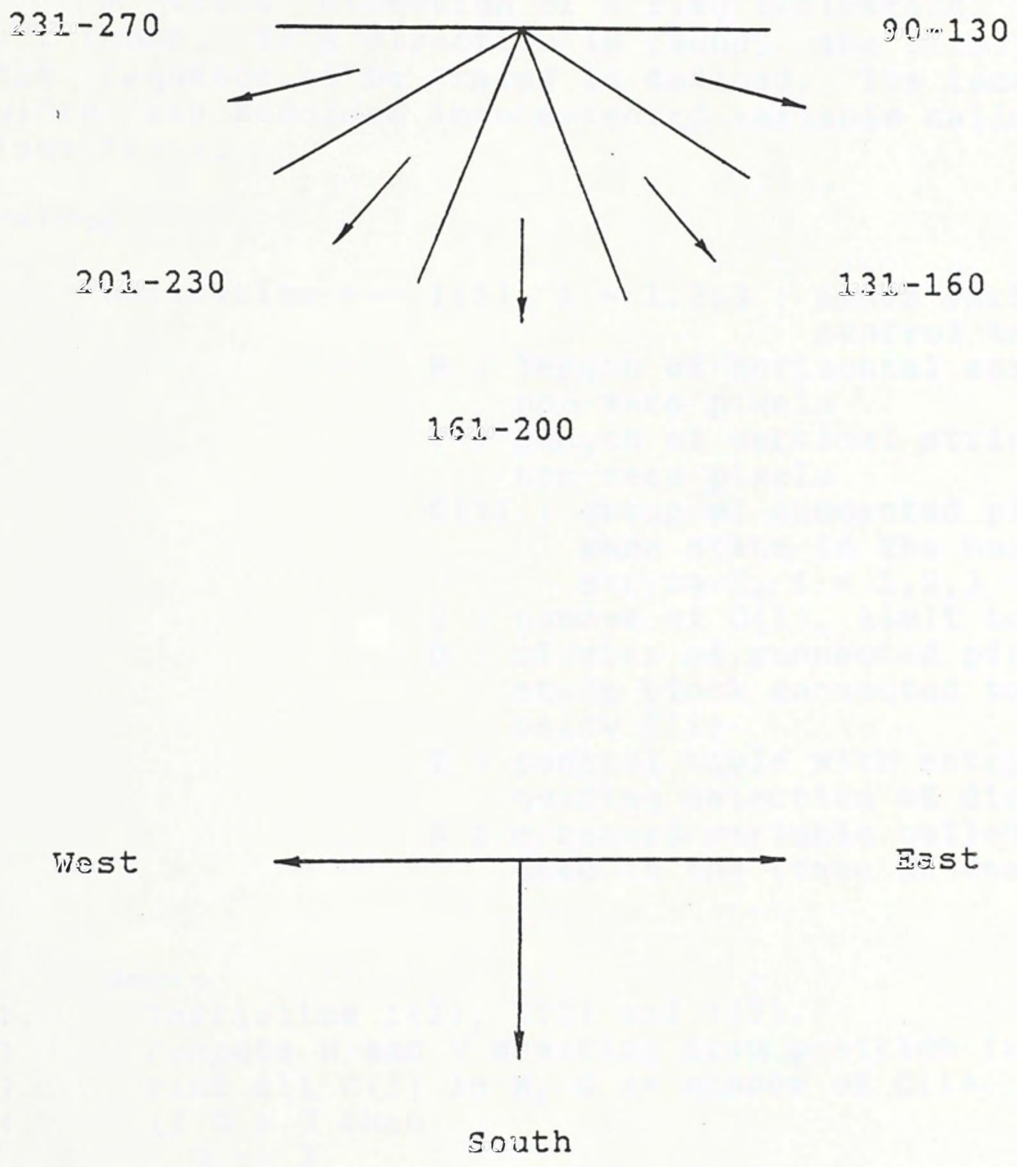


Figure 4.6 Division of Space and Direction

Procedure Starter

Input

X and y coordinates of a pixel of two possible states: Black and Special.

Output

A trace direction or a flag indicating direction not found. If a direction is found, the first unit of the sequence to be traced is defined, its location and width are recorded into a record variable called State (see text).

Method

```
Definition --- I(i), i = 1,2,3 : index variable for
                                  control table T
               H : length of horizontal string of
                   non-zero pixels
               V : length of vertical string of
                   non-zero pixels
               C(i) : group of connected pixels of
                      same state in the horizontal
                      string H, i = 1,2,3
               G : number of C(i), limit to 3
               D : cluster of connected pixels of
                   state black connected to and
                   below C(i)
               T : control table with entry
                   guiding selection of direction
               S : a record variable called State
                   used in the trace process.
```

```
     Begin
1.     Initialize I(1), I(2) and I(3).
2.     Compute H and V starting from position (x,y).
3.     Find all C(i) in H, G := number of C(i).
4.     If G > 3 then
          G := 3.
5.     For i := 1 to G do 7
          Begin
6.          Find D(i) for C(i).
7.          Compute I(i) using C(i), D(i) and V
          End.
8.     Find direction using T(I(1),I(2),I(3)).
9.     Prepare first unit in S.
10.  End of procedure.
```

Figure 4.7 Procedure Starter

Case	Condition			Direction
	index1 abc	index2 abc	index3 abc	
1	SBN	---	---	west from right end
2	SBN	dPd	ddd	east
3	SBC	ddd	---	south
4	SBC	dPd	dBd	east
5	dPN	dBN	---	west from right end
6	dPN	SBC	---	south from pixel group two
7	dPN	LBC	---	east
8	dPN	dBd	dPd	east
9	dPC	---	---	south
10	dPC	dBN	---	west from right end
11	dPC	dBN	dPd	east
12	dPC	SBC	dPd	south from pixel group two
13	dPC	LBC	dPd	east
14	LBN	---	---	west from right end
15	LBN	dPd	ddd	east
16	LBd	ddd	ddd	south if short black cluster directly below, else east or west depending on offset of black cluster

Note: a = size -- L: large
S: small
d: don't care

b = type -- B: Black
P: Special or Processed
d: don't care

c = connection -- N: no black cluster connected
C: black cluster connected
d: don't care

--- : non exsistent

Figure 4.8 Decision Table For Starter

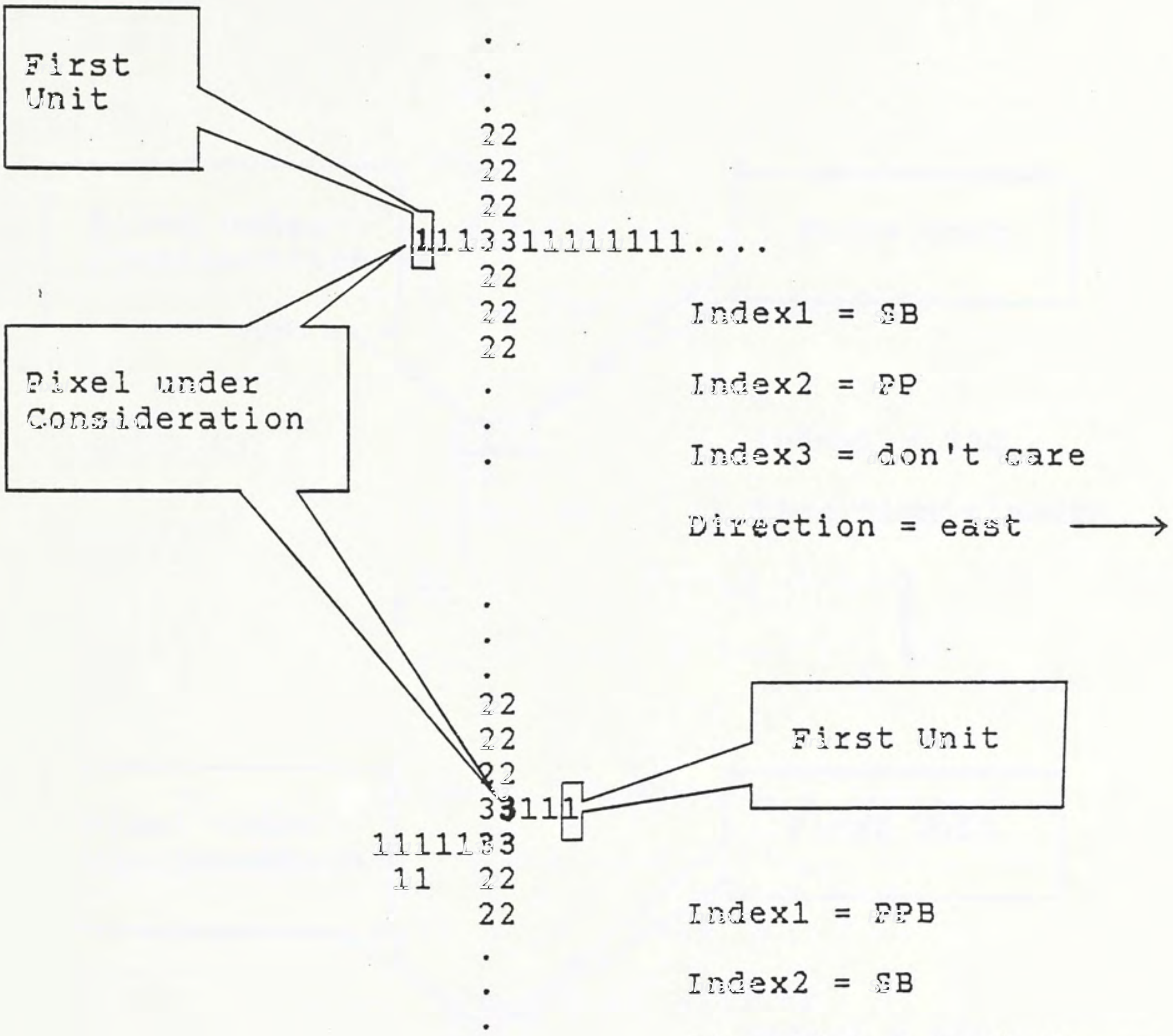
Conceptually, the decision upon which Starter will base to select a trace direction is by the relative size of the length and the width of a stroke segment. In practice, the condition is complicated by the possible multiple combinations of pixels of various states as defined above. These combinations necessitates a construction of a control table to handle the possible cases.

Upon receiving the pixel coordinates, Starter looks first at pixels at the east and those below (those at the west need no consideration as Scanner scans from left to right). As the states of the string of pixels at the east may have several values, the first three groups of pixels of same states are considered. The state of a connected group is determined by

1. type --- B (black) or P (special or processed),
2. size --- L (large) or S (small),
3. connection --- C (connected) or N (not connected).

By connection, we mean that those pixels directly below these connected groups are examined to see if clusters of Black pixels are present; and if so then this connected group is declared C (connected to black clusters), otherwise N (not connected to black clusters). By this, three indexes are resulted as shown in Figure 4.8 that for different combinations, different directions are selected.

For example, in case two as also shown in Figure 4.9a, the direction is chosen as east. In case thirteen, also see Figure 4.9a, the direction as chosen is west with the starting point changed to the right end. This can solve the problem of a possible inclined stroke segments hidden over the horizon. Note that in those cases of SBC (short black group connected to black cluster below), selecting south is a good decision as chances are high that they are the tip of a vertical or similarly inclined stroke segments. This also solves the case of an inclined stroke segments with most of



Index1 = SB

Index2 = PP

Index3 = don't care

Direction = east ⟶

Index1 = PPB

Index2 = SB

Direction = west ⟵

Note:

1 : black pixel
2 : processed pixel
3 : special pixel (node)

Figure 4.9a Starter Example

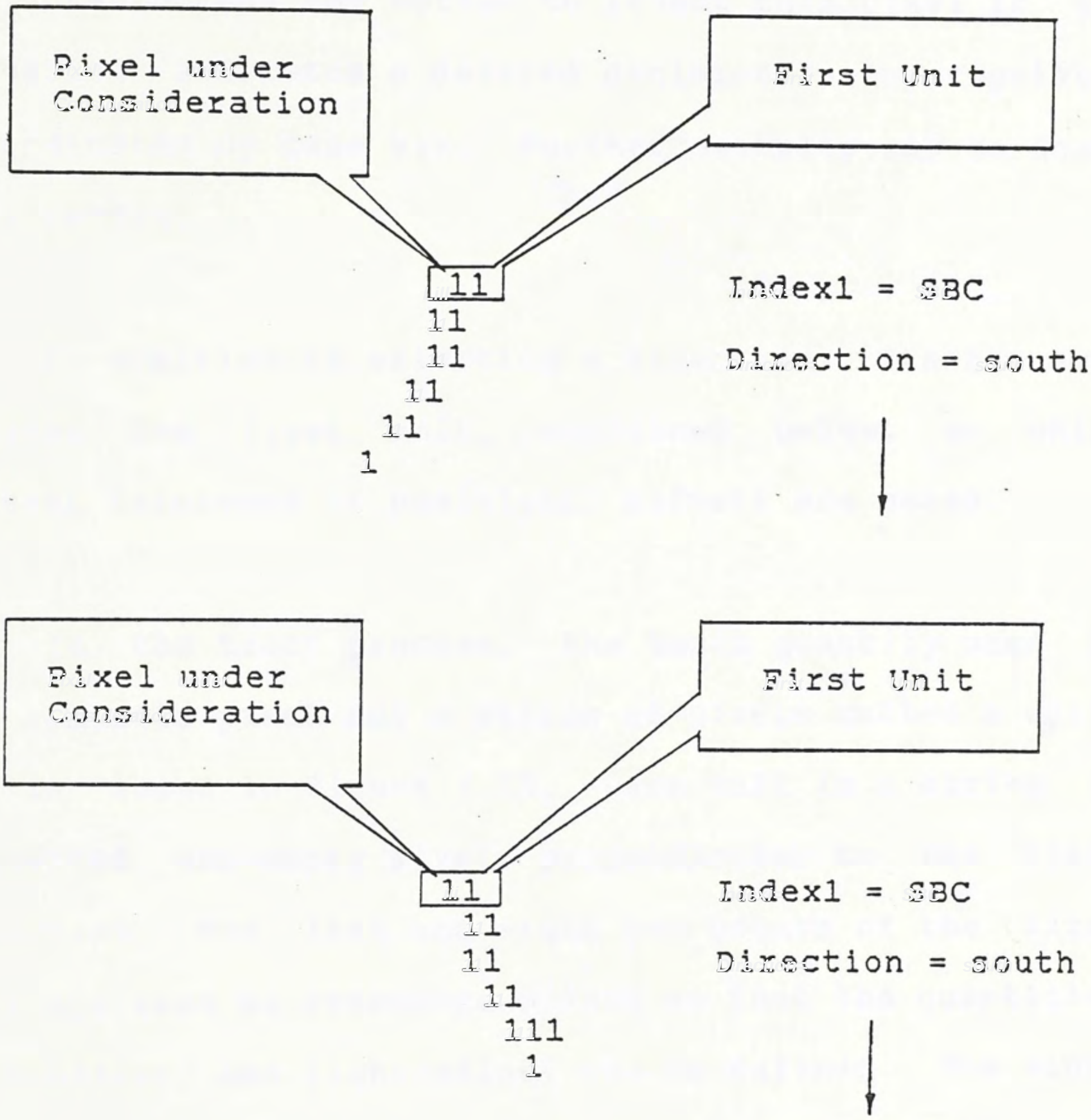


Figure 4.9b Starter Example

the data hidden because of the inclination. Therefore, graphically, Starter will be able to select direction as shown in Figure 4.9b though not through complete explicit examination. Note that a special case is that the Starter has the option to reject this pixel if the situation indicates a defered decision is appropriate, as indicated by case six. Further examples may be found in Appendix.

In addition to selecting a direction, Starter also defines the first unit, explained below, on which further reference of positional offsets are based.

In the trace process, the basic quantity used is not a single pixel but a string of pixels called a unit. As is shown in Figure 4.10, each unit is a string of connected non-empty pixels perpendicular to the trace direction. The left and right end points of the first unit are used as reference points so that the quantities left-offset and right-offset can be defined. The width of a unit is the number of pixels in the string. The length of a segment can then be defined as the number of connected units involved. This relatively unprecise definition that cannot reflect the true length of a stroke segment in the normal sense does not create problem as the length information is not critical for later processing. The final output of each trace process is thus a sequence of units representing one or more stroke segments.

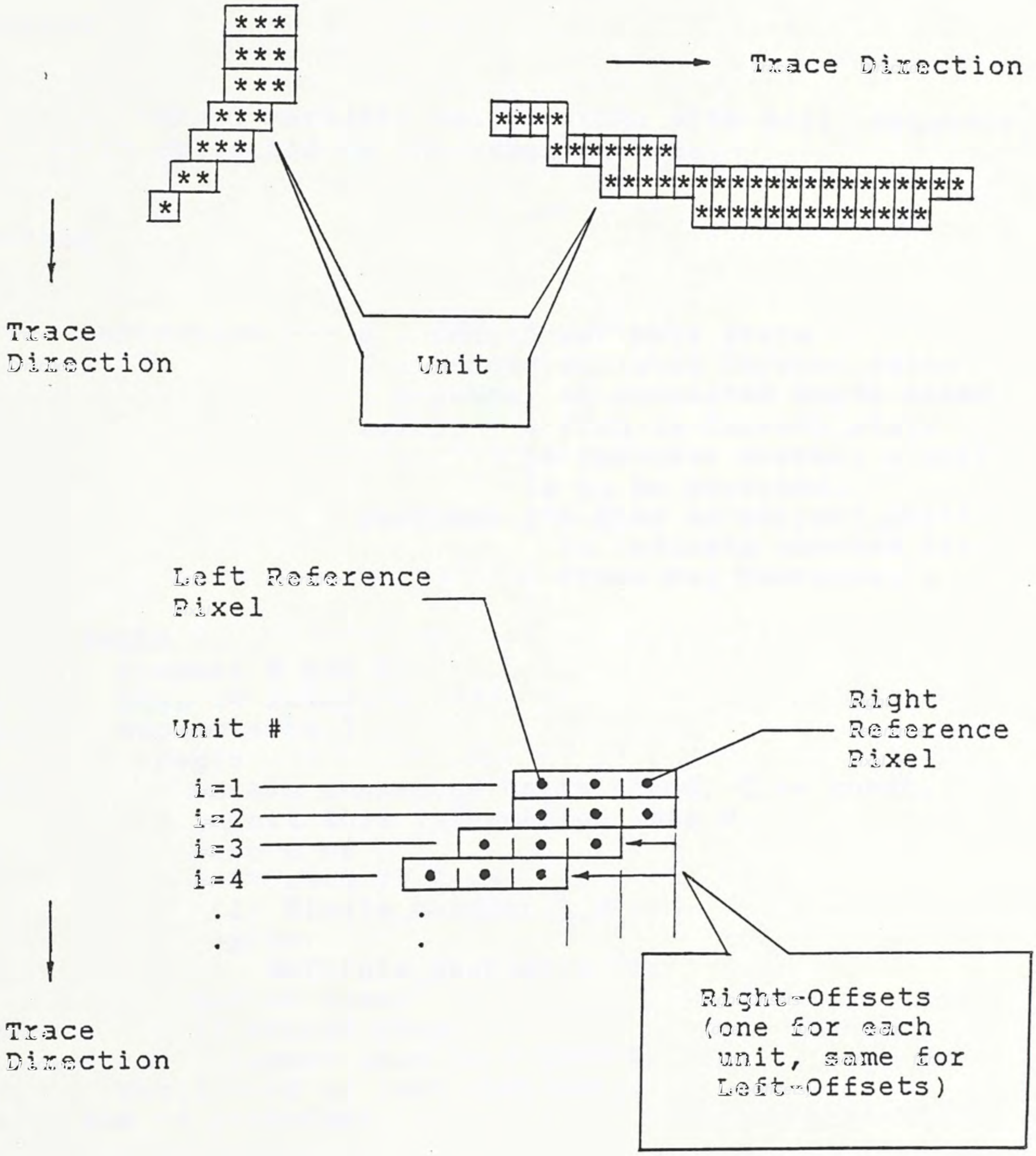


Figure 4.10 Definition of A Unit

Procedure Tracer

Input

A record variable called State in which trace direction and the first unit have been defined.

Output

A record variable called State with full sequence of units collected by the trace process.

Method

```
Definition --- S : record variable State
               U : record variable Current_state
               C : number of connected units ahead
               accept : a flag in Current_state
                        to indicate whether a unit
                        is to be accepted.
               continue : a flag in Current_state
                          to indicate whether the
                          trace may continue.

     Begin
1.     Prepare S and U.
2.     Stop := false.
3.     Repeat 4 to 7
         Begin
4.         Obtain connected units ahead, C := count,
           insert this information into U.
5.         Case C of
             0: stop := true.
             1: Single_handler(S,U).
             Else
               Multiple_handler(S,U).
           End of case.
6.         If accept then
             insert unit in U into S.
7.     Until stop or (not continue).
8.   End of procedure.
```

Figure 4.11 Procedure Tracer

Once a trace direction is known, the Tracer traces in that direction. The logic flow of the Tracer is shown in figure 4.11. The trace process is as follows. A module is called to examine situation ahead. If there is only one unit connected ahead, then module Single_handler is responsible for resolving the situation. Figures in Appendix have numerous examples of this. If there are more than one connected units, module Multiple_handler will handle the case, see figure A.1.30 for example.

Of all the main modules, to be described later, involved in the tracing process, two record variables are used during the trace process to hold the information gathered and all the necessary variables through which these modules communicate. These two records are called State and Current_state. Record State has two parts. The first part holds the width and offset information of the unit sequence that has been traced. This part is declared as an array. The second part holds the global status that indicates the overall situation such as whether the sequence is a straight one; or whether the sequence is bending towards right or towards left, and the direction of trace. Record Current_state is a buffer for holding the newest unit(s). In addition, it has variables used by the routines involved as input and output parameters. These include two important flags: accept to indicate the new unit is accepted after a test; continue to indicate that the trace may continue. By grouping data into two records and creating temporary copies when needed, the complete status of a trace can be recorded and passed to test a particular trace session conveniently and backtracking is therefore relatively easily implemented.

Backtracking is necessary as there are cases where there are multiple units ahead or the unit is not of a simple type, or the width of which indicates that a cross of multiple stroke segments may be encountered. In each of these circumstances, there is no easy way to get around but a sequencial testing for a path passing this area. By creating copies of record State and Current_state, and passing them to the handling routines, the array in State can be used as a working space that holds the information of the test run. On return, the complete path is returned as a whole. The module that initiates the testing then has the option to select the best path available. The path is judged by the length. The shortest path in most cases is chosen. Another criterion is that a good path is that the trace may still continue following the path. Figure A.1.30 has an example.

Coming back to the two modules Single_handler and Multiple_handler, it is necessary now to describe them individually. Single_handler handles only singly connected unit. If the unit is of type Black and the width of which does not exceed a threshold, then the unit is either accepted or rejected, which means a termination of the trace path. The latter case arises when the unit indicates a change in inclination. If the unit has a width over a threshold, or if the unit is not of a simple type, then a node must have been encountered, and special handling is therefore necessary. These are handled by modules Exception_handler and Special_handler.

Procedure Single_handler

Input

A singly connected unit recorded in record variable Current_state, and the past sequence of units recorded in record variable State.

Output

Result of testing and/or further trace directly recorded into record variables State and Current_state.

Method

```
Definition --- U : record variable Current_state
               S : record variable State
               accept : accept/reject the new unit
               width : width of a new unit
               type : type of the new unit
               (above three variables are in U)
               threshold : an empirical constant


Begin
  If type = Black then
    Begin
      See if unit acceptable, set/clear accept.
      If not accept then
        Begin
          If width > threshold then
            Exception_handler(S,U).
        End.
    End.
  Else
    Special_handler(S,U).
End of procedure.
```

Figure 4.12 Procedure Single_handler

Procedure Multiple_handler

Input

A set of connected units recorded in record variable Current_state, and the past sequence of units recorded in record variable State.

Output

Result of testing and/or further trace directly recorded into record variables State and Current_state.

Method

```
Definition --- found : flag to indicate if a path
                       is found
               count : number of connected units,
                       count is a variable in U
               S(I) : variable State for unit I,
                      for testing
               U(I) : variable Current_state for
                      unit I, used for testing
               S : variable State
               U : variable Current_state
               I : variable as counter of loop
               U(I).accept : accept is a variable
                             in U(I)
```

```
     Begin
1.     found := false.
2.     For I := 1 to count do 3 to 5
         Begin
3.         S(I) := S,
           U(I) := U.
4.         Modify U(I) to have only one connected unit.
5.         Single_handler(S(I),U(I)).
         End.
6.     Choose among U(I).accept = true for
         shortest path in S(I); name the pair as
         as S1 and U1; found := true.
7.     If found then
         Begin
8.         S := S1,
           U := U1.
         End.
9.   End of procedure.
```

Figure 4.13   Procedure Multiple_handler

Procedure Exception_handler

Input

A new unit known to be of type black and of a width exceeding the acceptable threshold. This unit is stored in record variable Current_state. The past sequence of units traced is stored in record variable State.

Output

Result of testing and/or further trace directly recorded into record variables State and Current_state.

Method

```
     Definition --- S : variable State
                    U : variable Current_state
                    S1 : copy of variable State for
                         testing
                    U1 : copy of Current_state for
                         testing
                    accept : accept flag in U

     Begin
1.     S1 := S,
       U1 := U.
2.     Predictor(S1,U1).
3.     If accept then
         Begin
4.         S := S1,
           U := U1.
         End.
5.   End of procedure.
```

Figure 4.14 Procedure Exception_handler

Multiple_handler handles multiple connected units. The way to resolve this is by calling Single_handler sequentially testing each connected unit one by one. Of all those paths returned, one will be selected.

Exception_handler is called when a unit is Black but its width exceeds a threshold. The situation is resolved by calling a module Predictor.

Special_handler is called when a unit is not of a simple type. That is, the unit may consist of pixels of type Black, Processed or Special. As part of the unit can be selected to reduce the range of search, a selection is made and the selected portion is passed to Predictor. The selection is based on the table in Figure 4.16. The different states of the pixel groups in the unit are converted to a maximum of three indexer quantized to B (Black), S (Special) and M (Processed).

Procedure Special_handler

Input

A new unit that is known to be not of simple type Black. This unit is recorded in record variable Current_state. The past sequence of units traced is recorded in record variable State.

Output

Result of testing and/or further trace directly recorded into record variables State and Current_state.

Method

Definition --- S : variable State
U : variable Current_state
S1 : copy of variable State for testing
U1 : copy of Current_state for testing
Index(I) : index variables for control table T
T : control table for selection portion of a unit to test
U1.accept : accept is a variable in U1

```
     Begin
1.     Based on data in U, assign values to
         Index(1), Index(2) and Index(3).
2.     S1 := S,
       U1 := U.
3.     Modify U1 as directed by
         T(Index(1),Index(2),(Index(3)).
4.     Predictor(S1,U1).
5.     If U1.accept then
         Begin
6.         S := S1,
           U := U1.
         End.
7.   End of procedure.
```

Figure 4.15 Procedure Special_handler

Case	Condition			Selected Portion
	index1	index2	index3	
1	B	M	dd	B
2	B	S	dd	B + S
3	S	--	--	S
4	S	B	--	S + B
5	S	M	--	S
6	S	B	M	S + B
7	S	B	S	S + B + S
8	M	--	--	reject
9	M	B	--	B
10	M	S	--	S
11	M	B	M	B
12	M	B	S	B + S
13	M	S	B	S + B
14	M	S	M	S

B : black portion
M : processed portion
S : special portion
-- : non existant

Figure 4.16 Selection Table for Special_handler

Input

A unit on which testing must be performed is stored in record variable Current_state. The past sequence of units traced is in record variable State.

Output

Result of testing and/or further trace directly recorded into record variables State and Current_state.

Method

```
Definition --- W : average width of units traced
               O : average offset of units traced
               S : variable State
               U : variable Current_state
               Margin : an empirical constant
               I,J : variables as counters for loops
               S(I,J) : copy of variable State
                        corresponds to I as width,
                        J as offset, used for testing
               U(I,J) : copy of variable Current_state
                        corresponds to I as width,
                        J as offset, used for testing
               straight : variable in S to show if
                          the collected units indicate
                          a straight stroke segment
```

```
     Begin
1.     Compute W and O.
2.     If straight then
         Begin
3.         Modify U with W as width, O as offset.
4.         LookAhead(S,U).
         End.
5.     Else
         Begin
6.         For I := O-Margin to O+Margin do 7 to 10
             Begin
7.             For J := W-Margin to W+Margin do 8 to 10
                 Begin
8.                 U(I,J) := U, S(I,J) := S.
9.                 Modify U(I,J) with I as offset and
                     J as width.
10.                If selected unit is acceptable then
                     LookAhead(S(I,J),U(I,J))
                 End.
             End.
11.        Choose among U(I,J).accept = true for
             shortest path in S(I,J); name the pair
             as S1 and U1; found := true.
12.        If found then
             Begin
13.            S := S1, U := U1.
             End.
14.        Else
15.          U.accept := false.
         End.
16.  End of procedure.
```

Figure 4.17 Procedure Predictor

Predictor expects an oversized unit, but smaller unit causes no trouble. Its job is to select a portion of the unit as a test unit and initiate a test run base on this new test unit. For the case in which a stroke segment has already been declared a straight segment, the expected width and offset is obvious. Otherwise, the selection is based on the average width and the average offset of those traced units plus and minus a margin. All these cases will be tested by calling a module LookAhead with this predicted unit as an input parameter. One of the reported path will be selected.

LookAhead is a subset of the module Tracer with the difference being that LookAhead will test run starting from the input unit and reports the resulting path. It is called by Predictor and is the module that initiates the recursive behaviour upon encountering unit that needs multiple testing for a good path. Examples of this can be found in figures in Appendix.

The control flow of this recursive testing procedure is given in Figure 4.19. It should not be confused with the flowchart of that of Level One.

Procedure LookAhead

Input

A selected unit in record variable Current_state. The past sequence of traced units is in record variable State.

Output

Result of further tracing is returned through record variables State and Current_state.

Method

```
Definition --- C : number of connected units ahead
               U : variable Current_state
               S : variable State


Begin
   Obtain connected units ahead,
      C := number of connected units;
      insert this information into U.
   Case C of
      Begin
         0: U.continue := false.
         1: Single_handler(S,U).
         Else
            Multiple_handler(S,U).
      End.
End of procedure.
```

Figure 4.18 Procedure Look_ahead

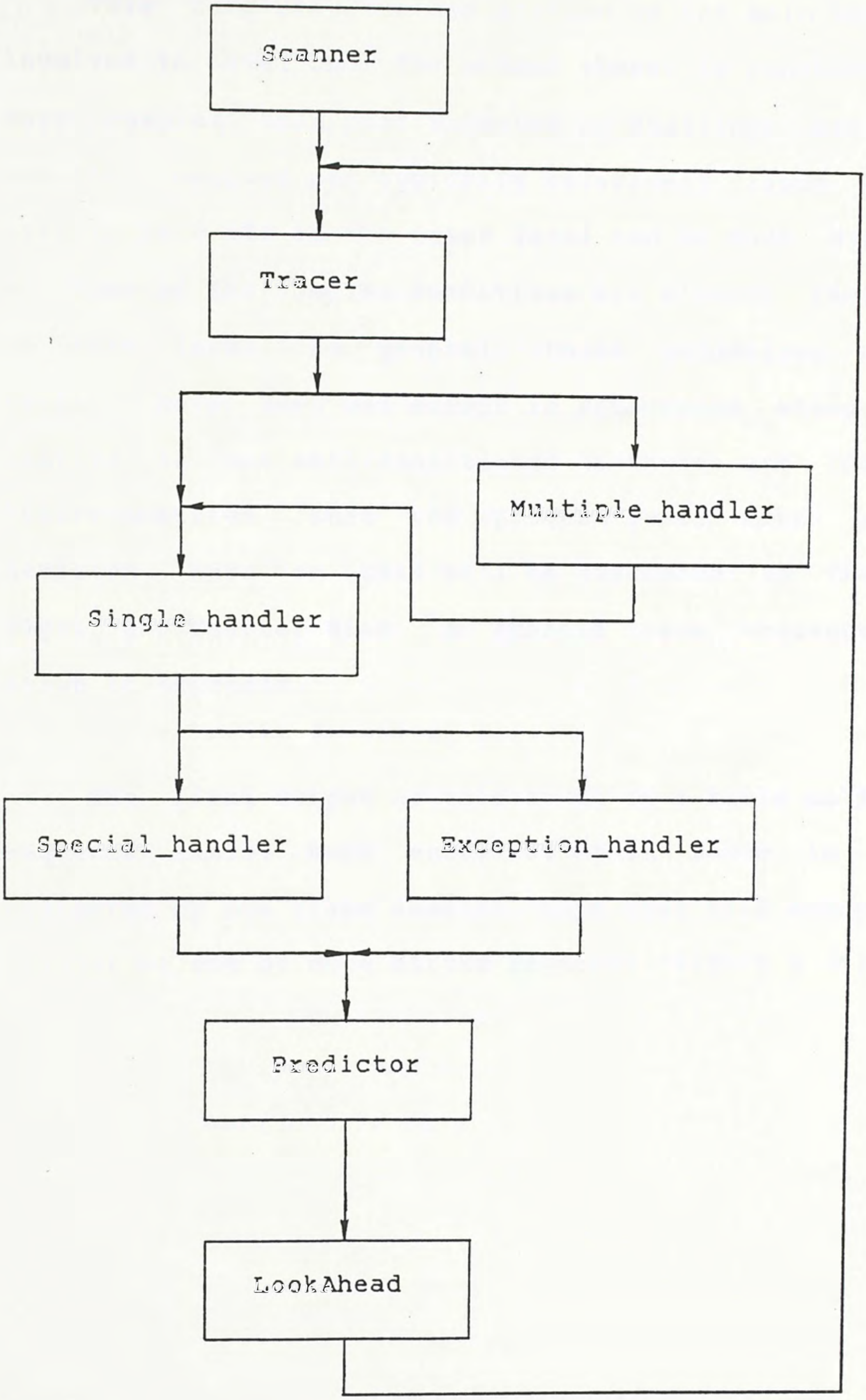


Figure 4.19 Control Flow among Main Modules

This completes the description of the main modules involved in Level One. The method chosen is considerably more complex than that reported by Stallings and the modules involved are typically relatively large. This cost is paid off as the later level can be made simpler as some of the complex conditions are already resolved in this level. In general, those guidelines given earlier have been met except in some cases where the pattern is not well conditioned or there are complex interconnection that the program will make wrong decision. More on this will be discussed in Chapter Eight and Chapter Nine. An example trace sequence is shown in Appendix.

The final output of this level is a table called a sequence table. Each entry of this table is data collected by one trace session. Note that each entry may consist of one or more stroke segments (Figure 4.3a).

5.0 STROKE EXTRACTION LEVEL TWO

5.1 Introduction

There is no theorectical reason for not incorporating this level into Level One and certainly doing so would speed up the process by eliminating the time consuming input/output in between. However, squeezing two already large programs into one would require very careful memory and data planning; and the limitation of data and code sizes of 64 k bytes each by most compilers for IBM-PC type machines places heavy constraints on data structures. Although extensive use of linked lists instead of large arrays may reduce the size of data segment, the added program complexity may be more than offset the benifit gained. As a result, a separate program is used to handle the work.

The input to this part of the system is the sequence table resulted from previous level (Figure 4.3a). The function of this level is to organize proper portions of each sequence into quantities each of which is called a token, which represents truely a stroke segment, and into a form suitable to be processed by Level Three. Part of the process can be regarded as constructing a graph of a Chinese character pattern that has been traced by Level One, and with all the nodes labelled and all the links identified. Figure 5.1 clarifies the concept.

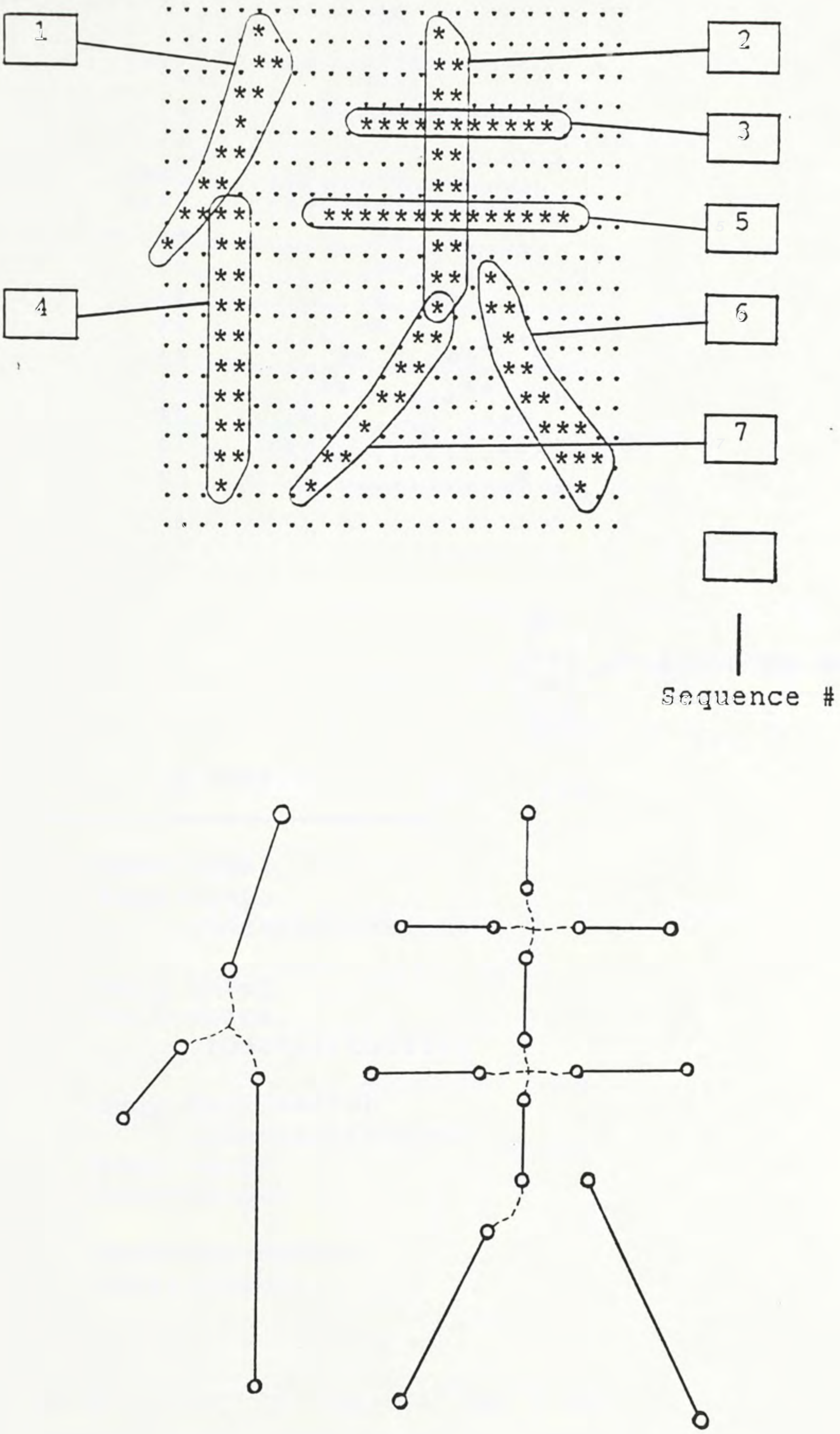
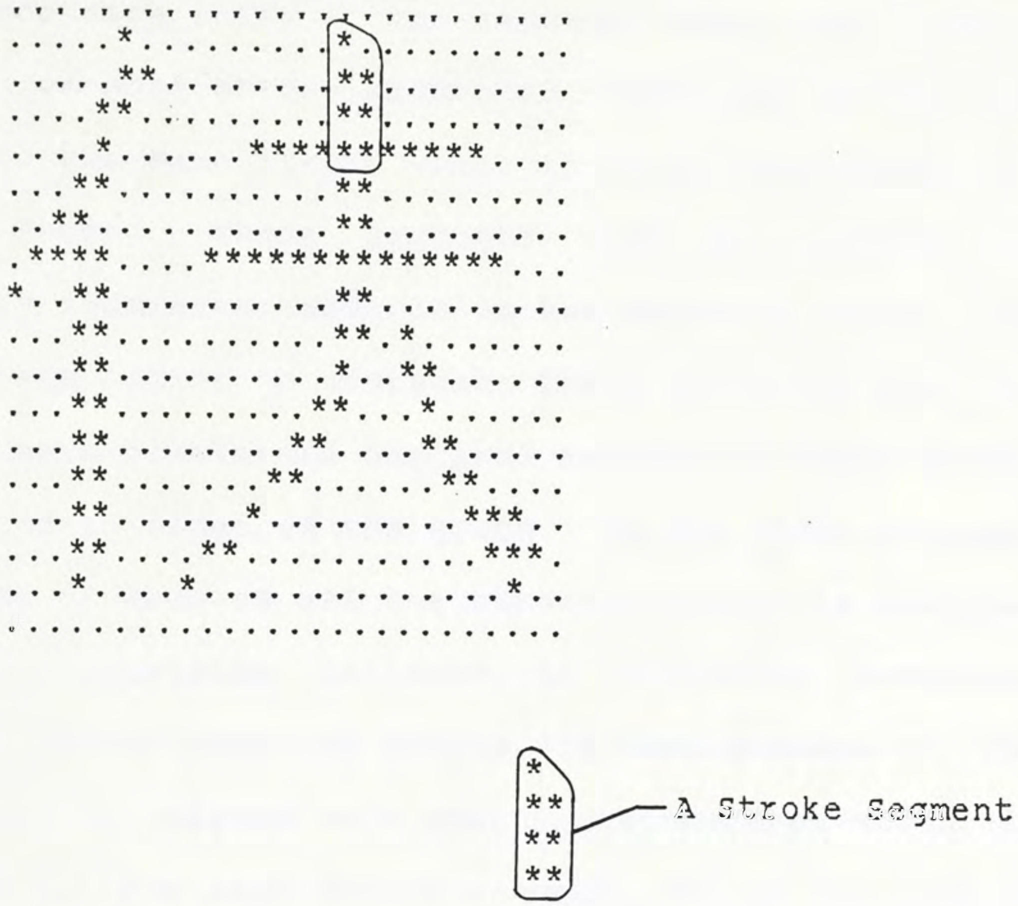


Figure 5.1 Character and Stroke Segments



A Token

Head-label
Head-shape
(identification)

Tail-label
Tail-shape
(identification)

Body-inclination
(identification)
Body-shape
Body-length

Sequence number
Token number

Note : Body is also called Trunk

Figure 5.2 A Token and A Stroke Segment

Since each entry in the sequence table may consist of one or more stroke segments, which may or may not belong to the same stroke which is to be constructed by Level Three. These segments must be singled out correctly from each sequence in the sequence table. The appropriate point to segregate these sequences are the points where pixels are declared Special as these points correspond to nodes of the graph. In the whole process, both ends of each of all the stroke segments is assigned a label (provision is made to eliminate redundant labels). These labelled points are simply nodes if the character is viewed as a graph, as has been shown in Figure 4.2. For each stroke segment, one of the node is called Head and the other Tail, see section 5.2 below.

A token, or physically a stroke segment, contains the following quantities: head-identification, head-label, tail-identification, tail-label, trunk-identification, trunk-shape, trunk-length, and the number of the sequence to which this token belongs (Figure 5.2). These quantities are not readily available and must be computed. Head-label and tail-label are simply the label assigned as discussed above. Head-identification and tail-identification are used to describe the shape of a given node computed using the corresponding end units of a stroke segment. Trunk-length is the length of a stroke segment. Trunk-shape describes whether the width of a stroke segment decreases, increases, or remains constant from head to tail. Trunk-identification describes the inclination of a stroke segment. These will be further clarified in later sections.

All together three tables are produced by this Level (Figure 5.3): a token table, a label table, and a link table. The token table contains a list of tokens. The label table contains a list of labels, with additional entries under each label as those tokens sharing that same label. The link table carries additional information describing how the tokens are connected, that is, how the graph is connected. Hence the main entry of the table is again a list of label, with the additional entries as a list of token pairs and the related connection information. Four quantities called P1, P2, P3 and P4 are used to describe the connection. This will be discussed later.

There is an additional program that converts these tables into text files as currently, there is no easy way to allow direct data transfering between Level Two, which is written with Pascal language, and Level Three, which is written with Prolog language, and which can only handle text files. Consequently, a little conversion is necessary.

```
        An Entry of The Token Table

-----------------------------------------

Token # --- Head Identification

            Head Label

            Tail Identification

            Tail Label

            Trunk Identification

            Trunk Shape

            Trunk Length

            Sequence Number

-----------------------------------------


        An Entry of The Label Table

---------------------------------------------

Label # --- List of Tokens whose Head Label

            is equal to this Label

        --- List of Tokens whose Tail Label

            is equal to this Label

---------------------------------------------


        An Entry of The Link Table

-----------------------------------------

Label # --- List of pairs of Tokens and

            their connection parameters

            P1, P2, P3 and P4

-----------------------------------------
```

Figure 5.3   Entries of The Three Tables

Program Level Two

Input

A sequence table produced by Level One.

Output

Three tables: a token table, a label table and a link table.

Method

```
Definition --- Q : sequence table
               N : number of sequnces
               S(I) : a sequence whose number = I
               T : token table
               L : label table
               K : link table


     Begin
1.     Load Q.
2.     Re_examine.
3.     Labeller.
4.     For I := 1 to N do 5
          Begin
5.          Builder(S(I)).
          End.
6.     Build_label_list.
7.     Build_link_list.
8.     Save T, L and K.
9.   End of Program.
```

Note: Re_examine, Labeller, Build_label_list, and Build_link_list are program modules, see text for details.

Figure 5.3 Program Level Two

```
     Matrix M(1..resolution,1..resolution)

-----------------------------------------------------------

 Each entry M(i,j) being a record R, i = 1..resolution,
                                     j = 1..resolution.


 Record R

    Begin

       N : number of sequences

       S : array [1..4] to hold the identity of sequences
             that has pixel at that location

       M : label number

       P : number of tokens

       T : array [1..4] to hold the identity of tokens

    End of Record.
```

Figure 5.4 Structure of The Working Matrix

Procedure Re_examine

Input

Sequence Table from Level One. A matrix M(1..24,1..24) as defined in Figure 5.4.

Output

A matrix M(1..24,1..24) with sequence part modified.

Method

```
Definition --- M : matrix M(1..24,1..24)
               N : total number of sequences
               I : variable as counter for loop
               S(I) : sequence whose number is I


     Begin
1.     Initialize M.
2.     For I := 1 to N do 3
         Begin
3.         With S(I) do
             Begin
4.             Plot all units to M.
             End.
         End.
5.   End of Procedure.
```

Figure 5.5 Procedure Re_examine

Procedure Labeller

Input

Matrix M with sequence part filled by procedure Re_examine.

Output

Matrix M with label part modified. A list called label list (not yet fully completed).

Method

```
     Definition --- R : size of one axis of a character
                        pattern ( = 24)
                    I,J : variables as counters for
                          loops, also used as indexes to
                          matrix M
                    N : number of sequences in a
                        particular position of matrix M,
                        ie., M(I,J)
                    A : label number

     Begin
1.     For I := 1 to R-1 do 2 to 6
         Begin
2.         For J := 1 to R-1 do 3 to 6
             Begin
3.             If (N of M(I,J)>1) and
                  (not yet assigned label) then
                 Begin
4.                 Assign a label A to M(I,J),
                   Insert label A to L.
                 End.
6.             Examine M(I,J+1),M(I+1,J) and M(I+1,J+1),
               assign them label A if N of them > 1.
             End.
         End.
7.   End of Procedure.
```

Figure 5.6 Procedure Labeller

Procedure Builder

Input

Sequence table from Level One. Matrix M processed by procedure Re_examine and Labeller.

Output

Matrix M with token part modified. A list called token list.

Method

```
Definition --- S : a stroke segment (a token)
               T : token list

Begin
1.    Repeat 2 to 8
2.      Obtain a part of the sequence as
        a stroke segment S.
3.      Write the stroke segment to matrix M.
4.      Compute Head parameter.
5.      Compute Body parameter.
6.      Compute Tail parameter.
7.      Insert S into T.
8.    Until all segments done.
9.  End of Procedure.
```

Figure 5.7 Procedure Builder

Procedure Build_link_list

Input

A list called label list.

Output

A list called link list with entries as defined in Figure 5.3. This list is indexed with label names.

Method

```
Definition --- N : number of labels
               L : label list
               I : variable as counter for loop,
                   also used as label number.
               J,K : variable as counter for loop,
                     also used as token numbers.
               M : number of tokens under L(I)
               P1, P2, P3 and P4 : connection
                                   parameters
               U : link list
```

```
     Begin
1.     For I := 1 to N do 2 to 6
         Begin
2.         With L(I) do 3 to 6
             Begin
3.             For J := 1 to M do 4 to 6
                 Begin
4.                 For K := J+1 to M do 5 to 6
                     Begin
5.                     Compute connection parameters
                       P1, P2, P3 and P4.
6.                     Insert J, K, P1, P2, P3 and P4
                       to U.
                     End.
                 End.
             End.
         End.
7.   End of Procedure.
```

Figure 5.8 Procedure Build_link_list

Procedure Build_label_list

Input

A list of tokens with both head_label and tail_label defined.

Output

A list called label list with entries as defined in Figure 5.3. This list is indexed with label number.

Method

```
     Definition --- N : number of tokens
                    I : variable as counter for loop,
                        also serves as identities of
                        tokens
                    L : label list

     Begin
1.     For I := 1 to N do 2 to 4
         Begin
2.        With T(I) do 3 to 4
            Begin
3.            Insert I to L(head_label).
4.            Insert I to L(tail_label).
            End.
         End.
5.   End of Procedure.
```

Figure 5.9 Procedure Build_label_list

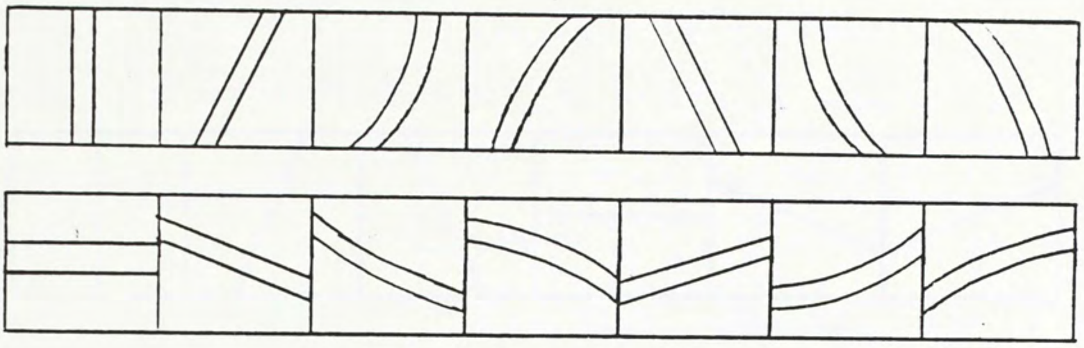
5.2 Detail

The logic of the program is shown in Figure 5.3. After obtaining the data from previous level, the data must be restructured so that data of different sequences can be more easily related. This is by redisplaying the data into a matrix with each position of the matrix corresponds to the position of a pixel of the character originally traced, and under which are entries that allow recording of what sequences are involved (Figure 5.4). This is done by a module called Re_examine (Figure 5.5).

The second module is named Labeller (Figure 5.6) which assigns label to all the nodes that are shared by multiple stroke sequences by scanning the matrix from top to bottom, and from left to right. Provision is made to make sure that all pixels that satisfy the above conditions of the same connected cluster are assigned the same label. Note that those nodes at the isolated ends of a stroke segments are not yet handled. They will be handled later (by module Builder) when encountered to keep this module simple.

After the labelling step, each stroke sequence is examined individually by a module called Builder (Figure 5.7) to compute all parameters of the stroke segments involved. These parameters are: inclination, shape, length and identification of the two ends. These are discussed in the following.

Segment Inclination



Segment Shape

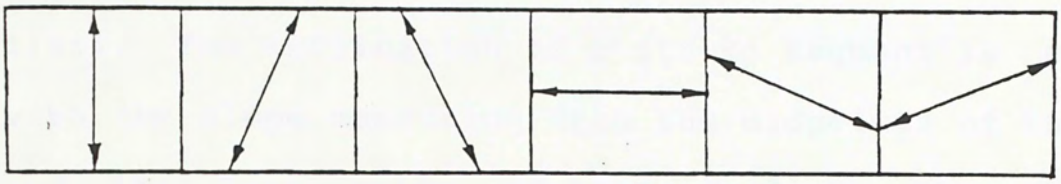
```
↓     *********      ******        ***
       *******       ******       *****
        *****        ******      *******
         ***         ******     *********

     Decreasing     Constant    Increasing
```

Figure 5.10 Segment Shape and Inclination

Slope of A Segment

Allow six ranges



Bending of A Segment

Three types

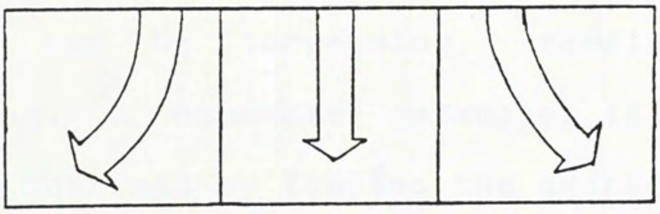
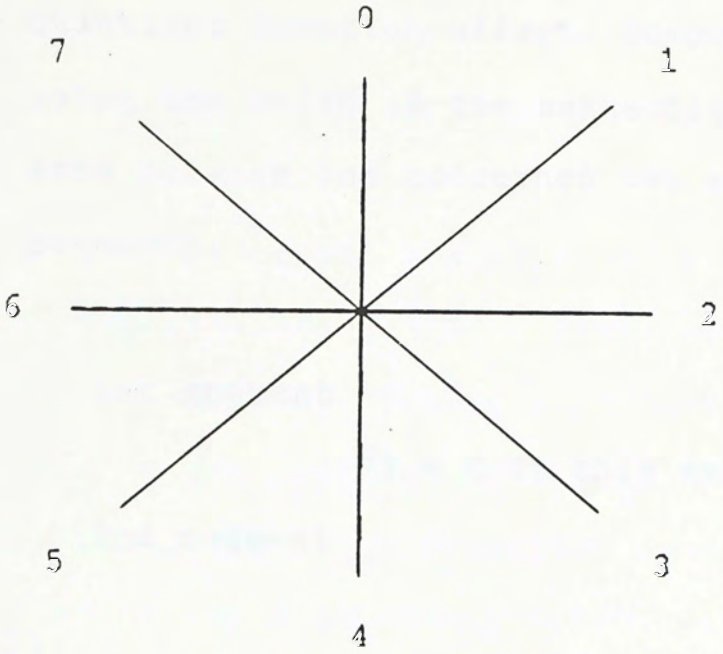


Figure 5.11 Segment Slope and Bending

Figure 5.10 lists the classification of a stroke segment according to its shape and inclination. For those stroke segments with fewer than four units, the classification is meaningless as not enough information is available and therefore they are made a separate class. The inclination of a stroke segment is computed with the slope measuring from the midpoints of the first and the last units of this segment. The slope is quantized to six values as shown in Figure 5.11. An additional parameter is used that indicates whether the segment is straignt, or bent clockwise or anticlockwise. This is computed by counting the number of pixels on the two sides of the main axis defined by the midpoints of the first and the last units of the segment. Altogether fourteen inclination is possible. As the width of a segment can be increasing, remaining constant or decreasing. A separate parameter is devoted to this which is obtained by finding the average slope in a plot of unit width versus segment length. This completes the body part of a stroke segment.

The shape of each of the two ends of a stroke segment is classified by the first two units, if Head; or last two units, if Tail; but only for those isolated Heads or Tails not connected to others. By the positional offset between these units together with widths of these units, a total of some 50 node numbers are assigned to different combinations. For those nodes which are connected to multiple stroke segments, the



Example 1.

P1 = 1

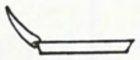
P2 = 2

Example 2.

P1 = 0

P2 = 4

Example 3.



P1 = 7

P2 = 2

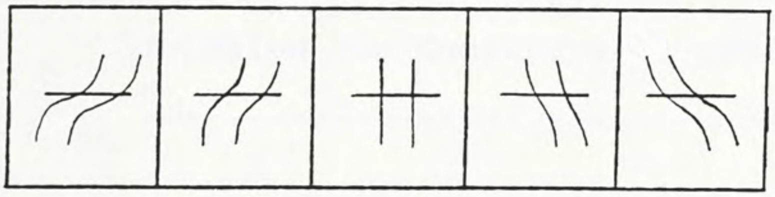
Figure 5.12 Relative Position of Two Connected Stroke Segments

P3 --- Quantized junction offset. Computed by using the units at the connection area between the concerned two stroke segments.

```
       *
      **          1st segment
      **
  ....**.....                      P3 = C in this example
      **
      **          2nd segment
 .....**......
```

(see Figure 5.2)

P3 = (A, B, C, D, E), a total of five values



P4 --- Quantized width variation at junction area

P4 = (A, B, C, D, E), a total of five values

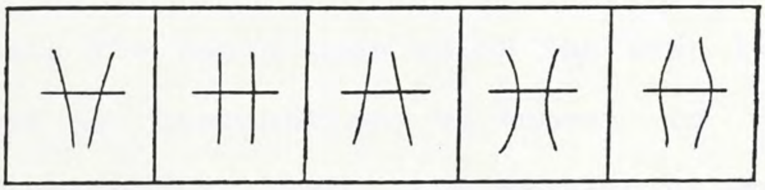


Figure 5.13 Junction Offset and Shape, P3 and P4

shape identification is of no use and instead, four parameters are computed to show how each pair of stroke segments are connected at a particular node.

The four parameters are simply named P1, P2, P3 and P4. P1 and P2 describe the relative positions of the two concerned stroke segments. Figure 5.12 demonstrates the concept. P3 and P4 are computed only when P1 and P2 show good possibility for the two concerned token to form a single quantity called a secondary-token, to be used in Level Three and will be explained at Chapter Six. The condition is

Condition For Computing P3 and P4

$$\mathrm{abs}(P1 - P2) = 3,\ 4 \text{ or } 5$$

P3 indicates the offset at the junction while P4 indicates the width variation (Figure 5.13). Together, they are the basis upon which the next level will decide whether a combination of tokens is meaningful when constructing strokes, the meaning of which will become clear when Level Three is discussed.

To summarize, a total of three sets of data are produced by this level. They include:

1. label list indicates what stroke segments are connected at what label;

2. token list with the token listed according to the order they are computed with all the aforementioned attributes;

3. link list detailing the connection attributes for each pair of stroke segments that are connected.

6.0 STROKE EXTRACTION LEVEL THREE

6.1 Introduction

This level is responsible for selectively combining the list of tokens, by using the available accompanying information, and with a set of prebuilt rules, into a list of strokes, which is the final target of this stroke extraction system (Figure 6.1). This is a synthesis process.

Prolog is chosen the programming language for this level as it is ideal for proving relations among subjects. The automatic handling of backtracking and recursive characteristics provide good program constructs for solving problems that are heavy on reasoning rather than on numerical computing, which is exactly the type of problem we are facing at this level. In addition, there are built-in database facility that allows facts and rules be used and manipulated. Section 6.2 will describe Level Three in detail.

Initial design of this level is faced with the following two options.

1. The most powerful approach will be that a Chinese character is completely defined by some relations, somewhat similar to that defined by Cheng and Chen [7], so that the resulting system

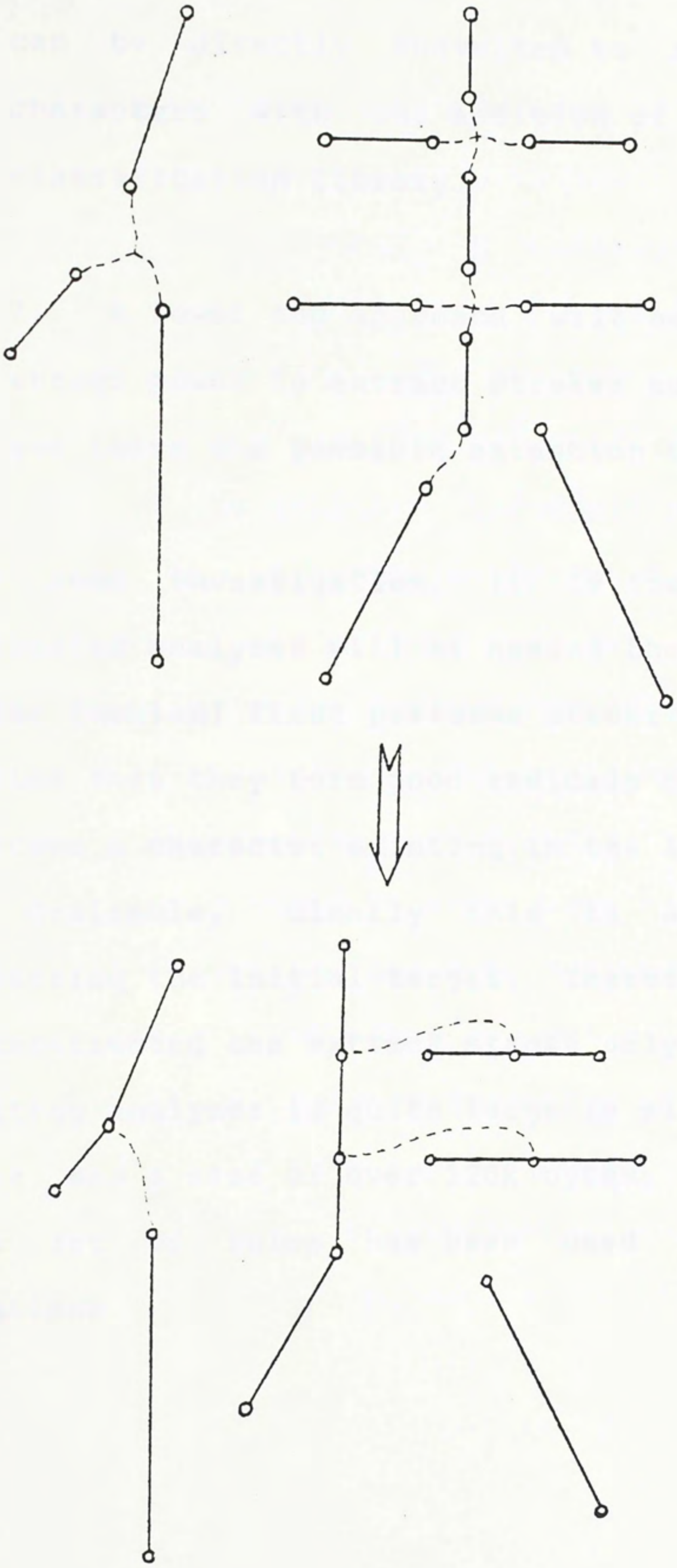


Figure 6.1 Stroke Extraction

not only can extract stroke information but also can be directly converted to recognize Chinese characters with the addition of some pre-stored classification library.

2. A lower end approach will be to provide just enough power to extract strokes out of the pattern and leave the possible extention to later.

After some investigation, it is found that a very complicated analyzer will be needed that (in a somewhat similar fashion) first performs stroke extraction, then verifies that they form good radicals that finally group to become a character existing in the library. Although very desirable, clearly this is an overkill when considering the initial target. Therefore, the analyzer as constructed can extract stroke only. Even so, the resulting analyzer is quite large in size. The compiled module has a size of over 120k bytes, partly because a large set of rules has been used to handle shape variations.

6.2 Detail

The two-dimensional nature of a typical stroke in terms of stroke segments as primitives indicates that a simple string grammer is difficult, though not impossible, in describing the relationship. More complicated grammars such as tree grammar and graph grammer [9,12] are powerful but still lack the facility to describe the semantic information, without which separate rules must be created for different strokes. With some 25 strokes (Figure 6.6) in total each of which is defined with one or more stroke segments and to allow for some variations in parameters to deal with possible variation of segment shapes, the resulting set of production rules would be very complicated. In contrast, semantic information can turn a simple grammar into a powerful descriptive machine which allows semantic information to separate different elements belonging to the same group such as stroke identity to the group of strokes. This semantic information can further be used to handle shape variations as well, as indeed be demonstrated in this case. A further advantage is inspired by Prolog's database facility that by building the analyzer in such a way that semantic information is used as reference rules to control the searching process, as opposed to directly incorporating them into the search procedure, the analyzer, once built, will need no further modification, and only the database be changed to handle new situation such as a new font with different stroke and stroke segment characteristics.

Attributed grammars were first formulated by Knuth to assign semantics or meanings to context-free languages. Formally, an attributed context-free grammar is defined as shown in figure 6.2 (closely following that given by Tsai and Fu [41]) which is the grammar chosen here to describe a stroke out of stroke segments (tokens). In this case, the starting symbol is Stk which represents a stroke. The set of non-terminals is

$$V_N = \{ Stk, Ptk, K \}$$

where Ptk and K represent partial-stroke and secondary token, to be explained later, respectively. The set of terminals has only one element

$$V_T = \{ t \}$$

which represent a token, or equivalently, a stroke segment. The set of production rules has two parts as shown in Figure 6.3.

Attributed Grammar

Definition

An attributed context-free string grammar is a 4-tuple G = (V_N, V_T, P, S) where

V_N Set of nonterminals,
V_T Set of terminals,
$S \in V_N$ Start symbol,

for each $X \in (V_N \cup V_T)$, there exists a finite set of attributes A(X), each attribute of A(X) having a set, either finite or infinitive, of possible values D ; and P is a set of productions each of which is divided into two parts: a syntactic rule and a semantic rule. The syntactic rule is of the following form

$$X_0 \dashrightarrow X_1 X_2 \cdots X_m$$

where $X_0 \in V_N$ and each $X_i \in V_N \cup V_T$ for $1 \leqslant i \leqslant m$. The semantic rule is a set of expressions of the following form

$$\begin{aligned} &\alpha_1 \dashrightarrow f_1(\alpha_{11}, \alpha_{12}, \cdots, \alpha_{1n}) \\ &\alpha_2 \dashrightarrow f_2(\alpha_{21}, \alpha_{22}, \cdots, \alpha_{2n}) \\ &\quad \vdots \\ &\alpha_n \dashrightarrow f_n(\alpha_{n1}, \alpha_{n2}, \cdots, \alpha_{nn}) \end{aligned}$$

where $\{\alpha_1, \alpha_2, \cdots, \alpha_m\} = A(X_0) \cup A(X_1) \cup \cdots \cup A(X_m)$, each $\alpha_{ij} (1 \leqslant i \leqslant n, 1 \leqslant j \leqslant n_i)$ is an attribute of some X_k for $0 \leqslant k \leqslant m$, and each $f_i (1 \leqslant i \leqslant n)$ is an operator which may be in one of the following three forms:

a) a mapping $f_i : D_{\alpha_{i1}} \times D_{\alpha_{i2}} \times D_{\alpha_{i3}} \times \cdots \times D_{\alpha_{in_i}} \dashrightarrow D_{\alpha_i}$,
b) a closed-form function, i.e., α_i may be expressed functionally in terms of the values of $\alpha_{i1}, \alpha_{i2}, \cdots, \alpha_{in}$,
c) an algorithm which takes $\alpha_{i1}, \alpha_{i2}, \cdots, \alpha_{in}$, and any other available information or data as input and α_i as output.

If $\{\alpha_1, \alpha_2, \cdots, \alpha_n\} = A(X_0)$ and each α_{ij} is an attribute of some X_k for $1 \leqslant k \leqslant m$, then all attributes defined in the semantic rules are synthesized attributes. If $\{\alpha_1, \alpha_2, \cdots, \alpha_n\} = A(X_1) \cup A(X_2) \cup \cdots \cup A(X_m)$ and each α_{ij} is an attribute of X_0, then all attributes defined are inherited attributes.

Figure 6.2 An Attributed Context Free Grammar

Syntactic Part

1. Stk(a) --> P(b)
2. Stk(a) --> P(b_1) P(b_2)
3. Stk(a) --> P(b_1) P(b_2) P(b_3)
4. P(b) --> K(c)
5. P(b) --> t(d)
6. K(c) --> t(d) K(c_1)
7. K(c) --> t(d_1) t(d_2)

Semantic Part

1. a <-- R1(b)
2. a <-- R2(b_1,b_2)
3. a <-- R3(b_1,b_2,b_3)
4. b <-- R4(c)
5. b <-- R5(d)
6. c <-- R6(d,c_1)
7. c <-- R7(d_1,d_2)

Note: see text for explanation

Figure 6.3 Production Rules

Note that alphabats a to d with possible numeric subscripts represent associated sets of attributes of the corresponding terminals or non-terminals. The details of these sets of attributes will be given later so as to prevent the rules from being crowded by large number of yet defined quantities. R1 to R7 are so named to indicate that they are in the form of rules as implemented though in a general sense they may be regarded as functions. Note that the attributes are all synthesized attributes.

Before explaining in further details, it is necessary to define the quantities partial-stroke and secondary-token. The idea of a partial-stroke is originated from the finding, after analyzing the strokes, that all of them can be defined in terms of some elemental units that are higher in hierarchy than tokens (stroke segments) and yet are conceptually of one-element structure, and will certainly exist as intermediate products during the process of combining tokens into strokes. This relationship is demonstrated in Figure 6.4. The list of partial-strokes used is shown in Figure 6.5. The list of strokes defined is shown in Figure 6.6. Secondary-token is also an intermediate product which is the results of grouping tokens of similar characteristics together forming a new quantity with the same attribute set of a token, hence as so named. Therefore, hierarchically, their order is as reflected by the production rules. Figure 6.7 gives an example that should clarify the distinctions.

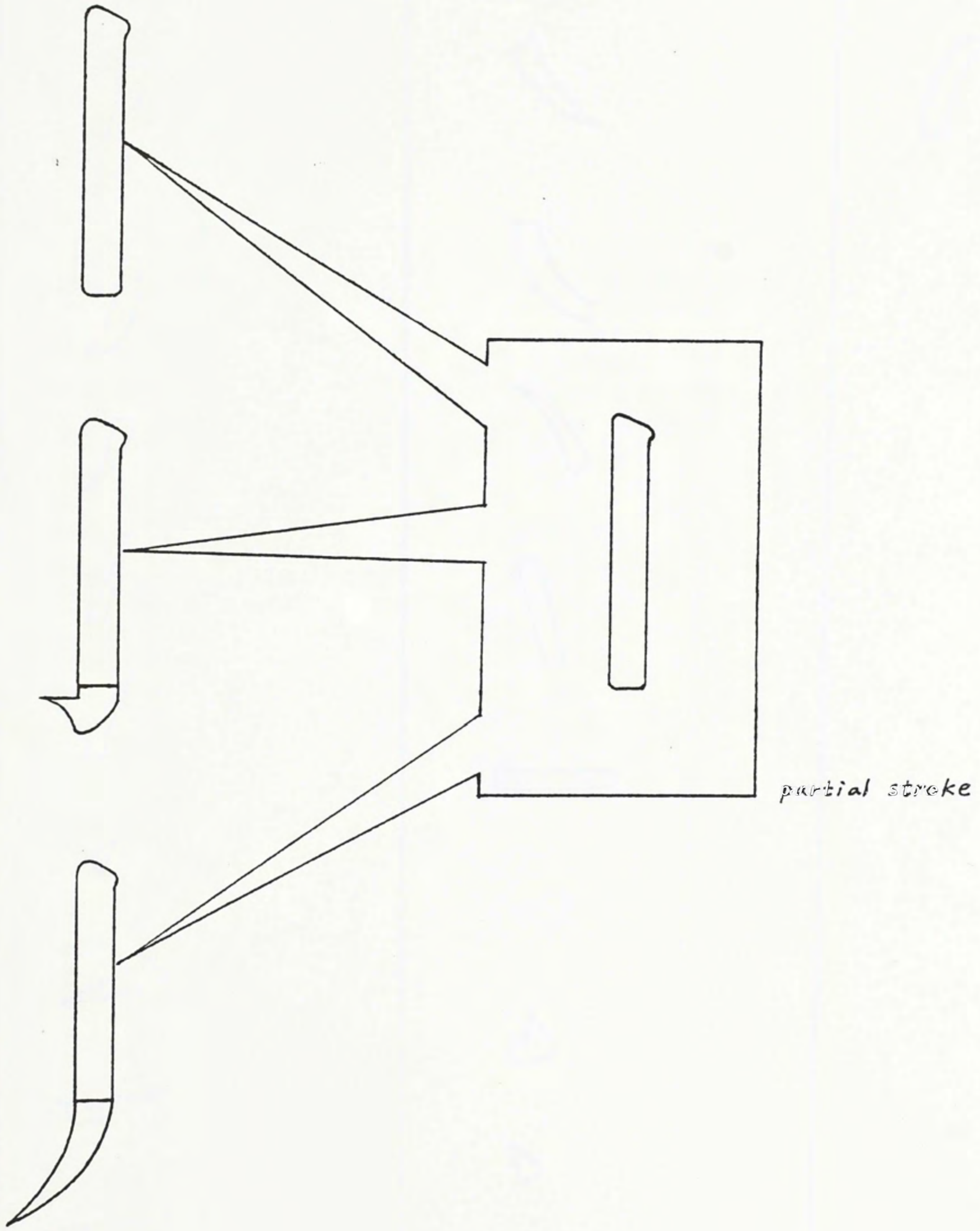


Figure 6.4 Stroke and Partial-stroke

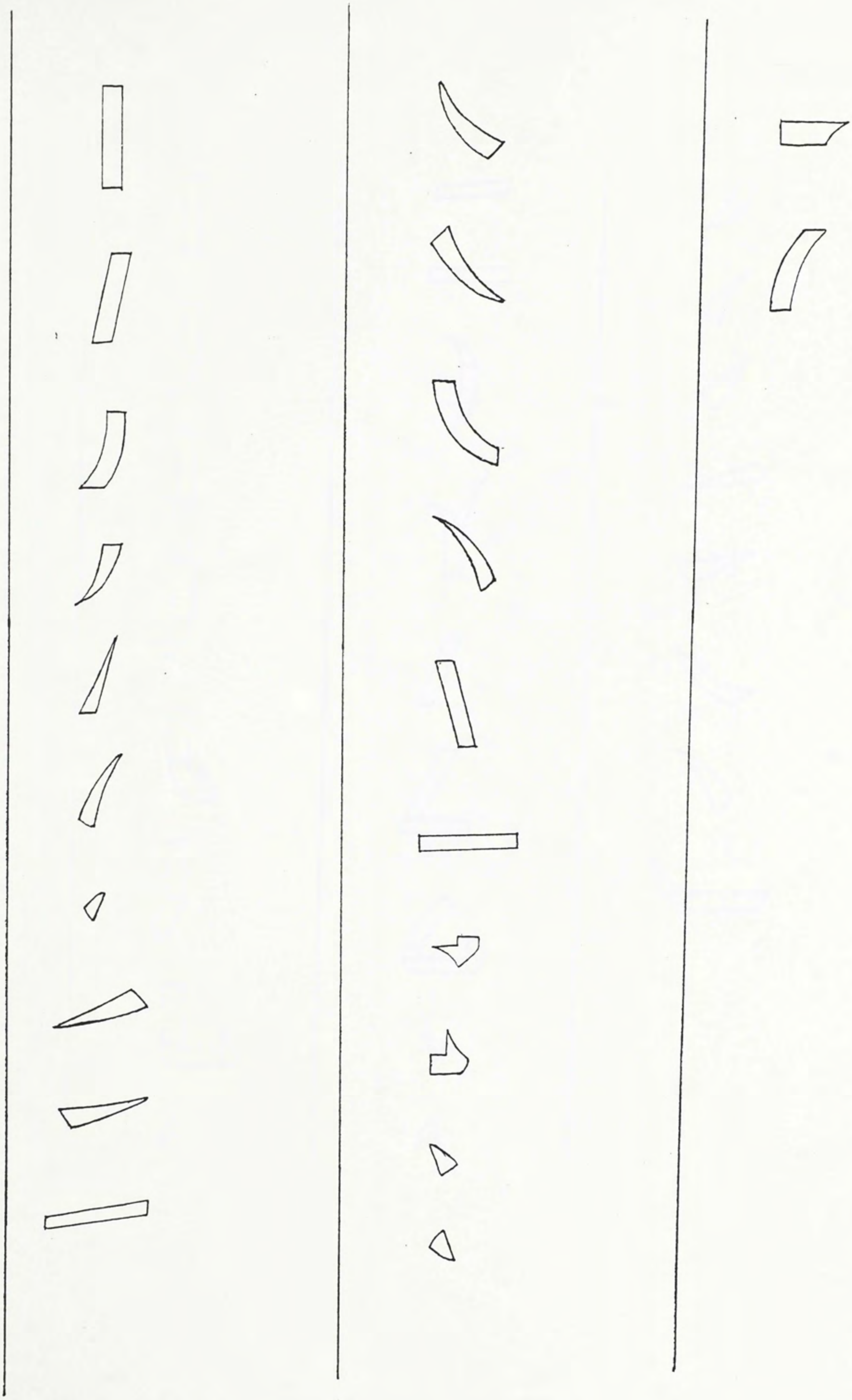


Figure 6.5 List of Partial-strokes

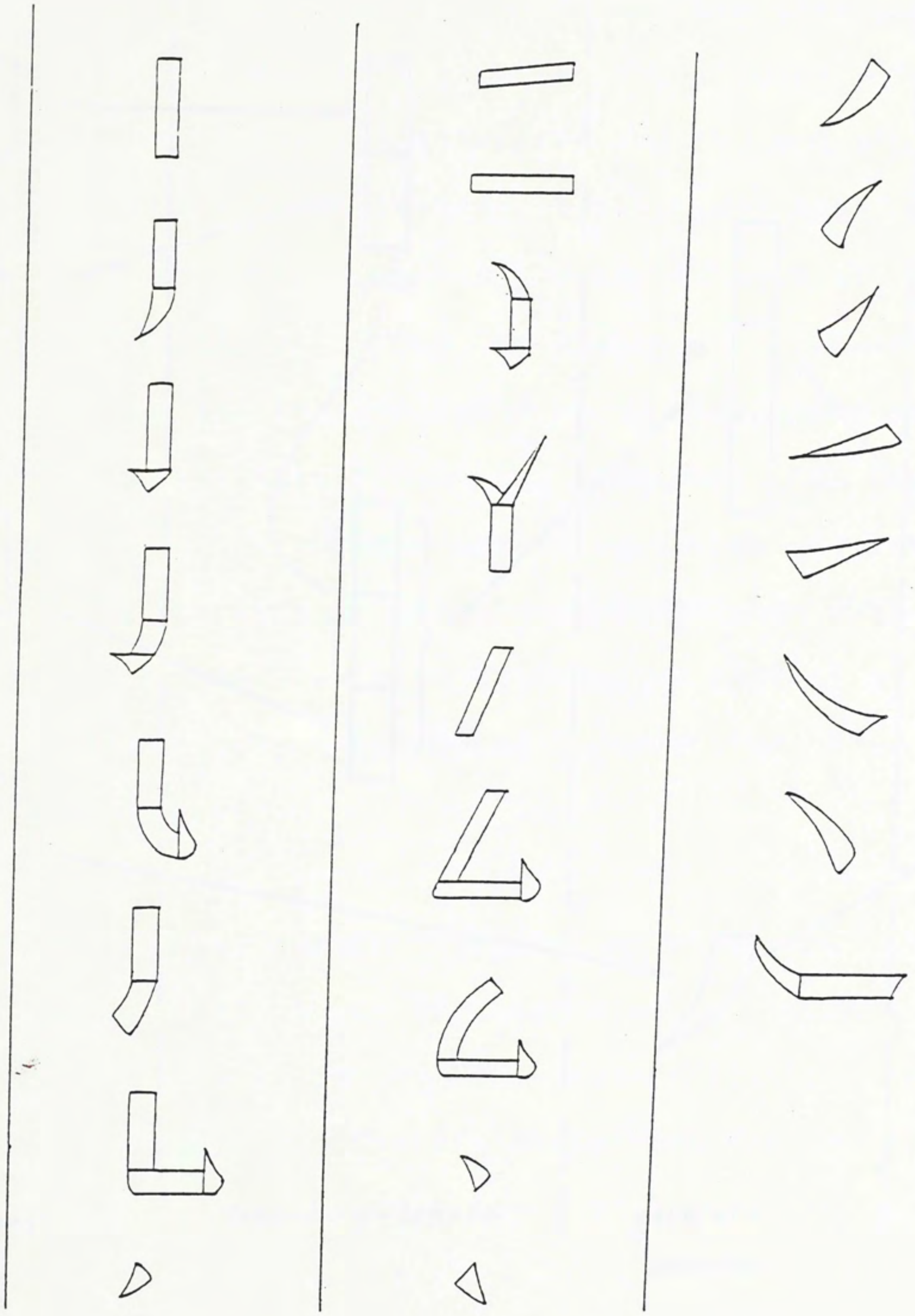


Figure 6.6 List of Strokes

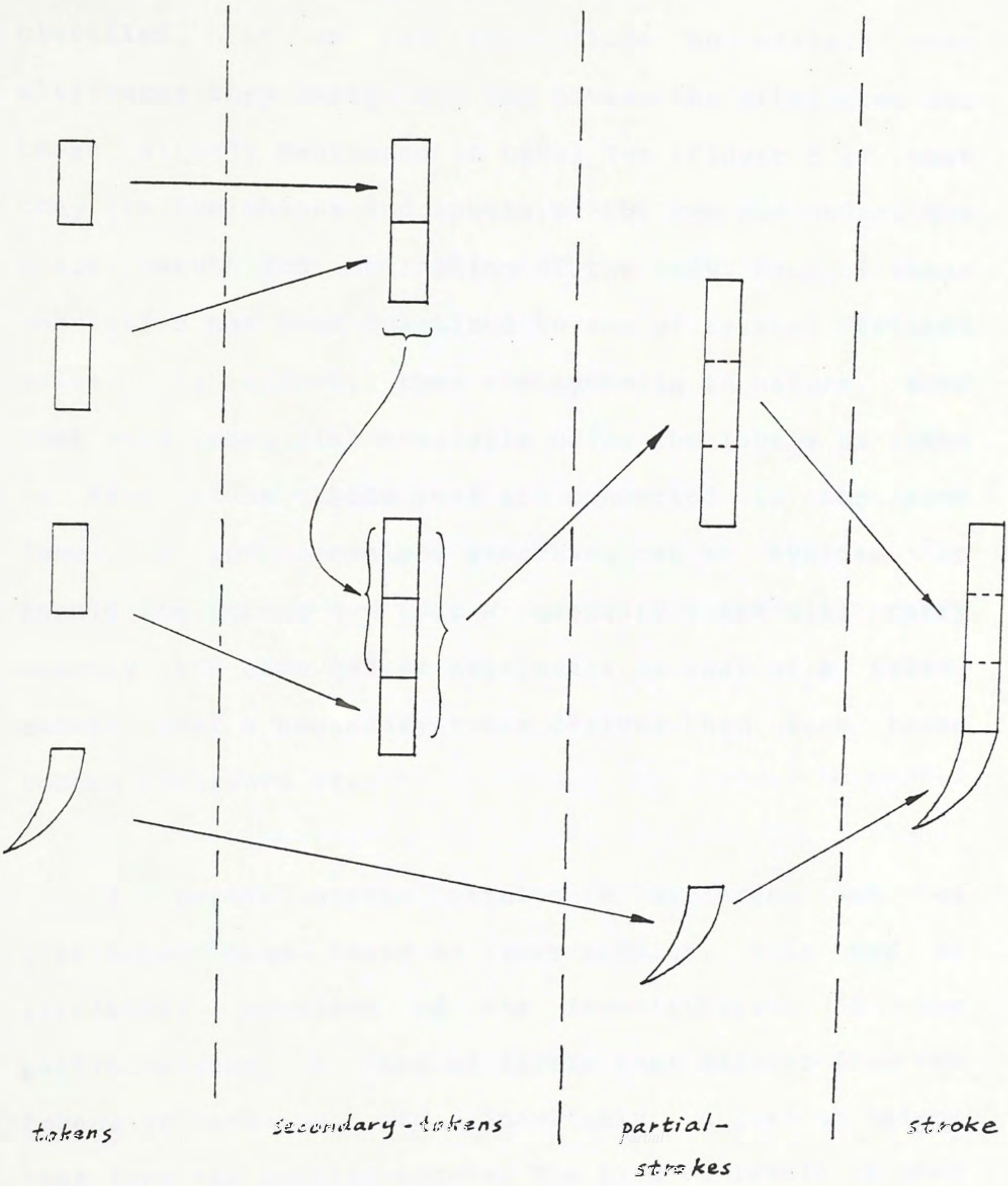


Figure 6.7 Stroke, Partial-stroke, Secondary-token and token

With both the non-terminals and terminals clarified, it is now appropriate to explain what attributes they carry. For the token, the attributes are those already mentioned in Level Two (Figure 5.2) that they are the shapes and labels of the two end nodes, the shape, length and inclination of the body. Each of these quantities has been quantized to one of several defined values. In effect, they are symbolic in nature. Note that with label list available using the labels as index to find other tokens that are connected to the same label, a lot of useless searching can be avoided. It should be clear now that a secondary-token will carry exactly the same set of attributes as that of a token, except that a secondary-token derives them from those tokens that form it.

A partial-stroke carries a different set of attributes from those of lower echelon. This set of attributes consists of the identification of the partial-stroke, a list of labels that derived from the tokens it contains, and, inevitably, a list of tokens that form the partial-stroke. The list of labels is used to check for connection with other quantities (by the use of the label list). A stroke carries a similar set of attributes as does a partial-stroke, with the identification being that of the stroke itself. Again the list of labels is used to check for connection.

The production rules can be explained in the following in the context of what is provided by the information generated by previous levels.

1. A stroke can be formed by a single partial-stroke. Rule one defines the stroke-identification given the parameters associated with the partial-stroke.

2. A stroke can be formed by two connected partial-strokes. Rule two defines the stroke-identification given the parameters associated with the two partial-strokes.

3. A stroke can be formed by three connected partial-strokes. Rule three defines the stroke-identification given the parameters associated with the three partial-strokes.

4. A partial-stroke can be formed by a secondary-token.

5. A partial-stroke can be formed by a single token.

6. A secondary-token can be formed by a token and a connected secondary-token.

7. A secondary token can be formed by two connected tokens.

The reasons for choosing these rules can be explained with the following example. Figure 6.8 is a pictorial sequence of what actually happens during the search process. In this example, the first stroke encountered is a straight stroke very common in Chinese characters. As this straight stroke is connected at multiple points to other strokes, it is separated into multiple stroke segments each becoming a token. The analyzer attempts to form a stroke starting with the first token encountered by calling a module to return all the possible partial-stroke that can be formed with this and possible together with other connected tokens (Rule one to three). In order to find a partial-stroke, the program first applies rule 5. In this case, it succeeds. The program then attempts rule 7 followed by rule 4 where both succeed and lead to a second partial-stroke. The process repeats until a total of four partial-strokes are found, all of the same type but consists of different lists of tokens. This is a breadth first process as all the possible solutions are found beginning at this node of the search graph. With these possible partial strokes, the analyzer first starts with the one with the most number of tokens and verifies that it becomes a stroke. Note that start testing first with a quantity involving the most number of tokens is a heuristic decision that will be discussed later.

A stroke consists of four tokens T1, T2, T3 and T4. Searching starts with T1...

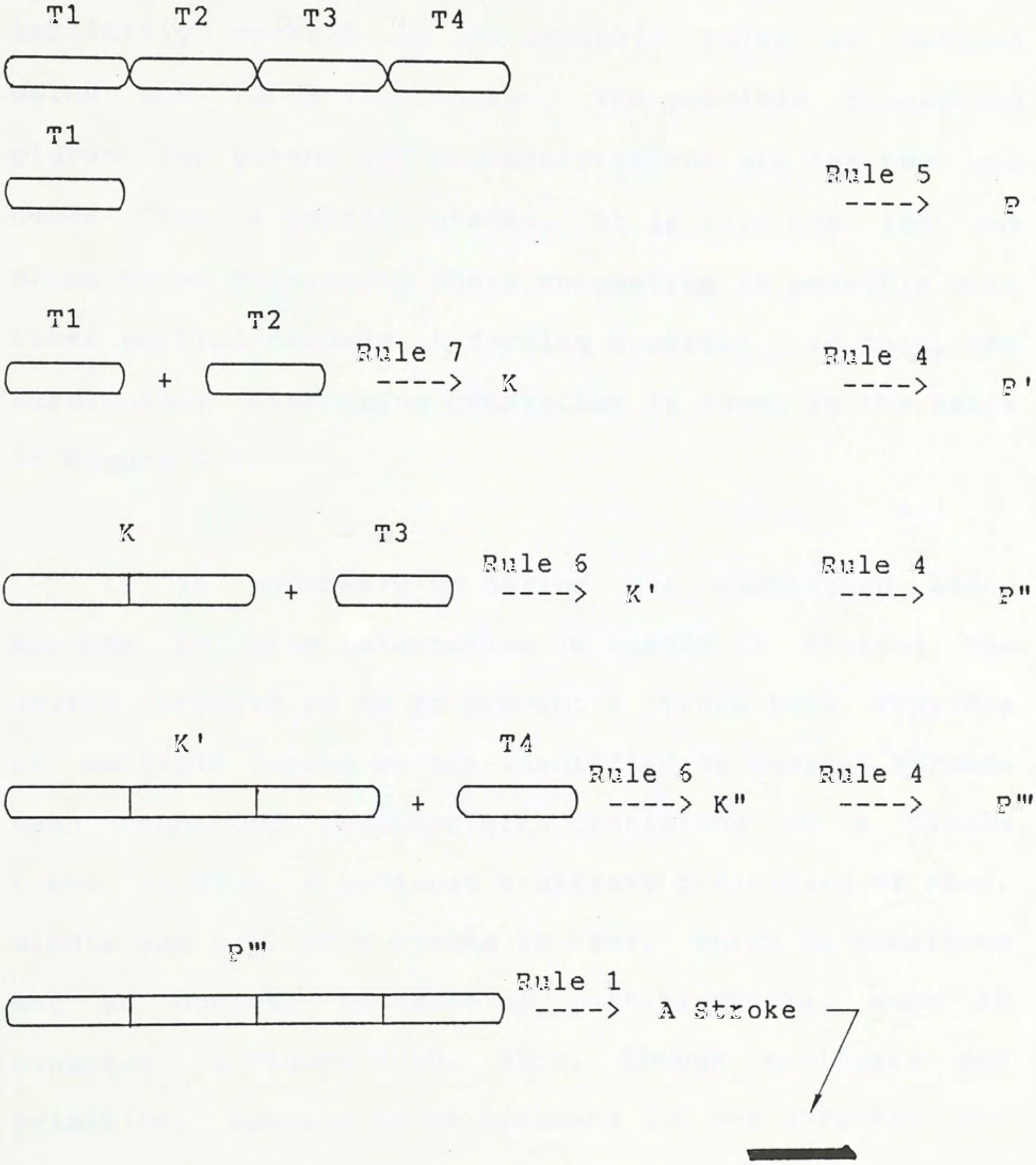


Figure 6.8 Sample Search Process

In the above, the connection detail between tokens, secondary-tokens, partial-strokes are not apparant from the production rules. Reason being that they have been implicitly covered by the semantic rules as defined using the label information. The possible connection places for tokens and secondary-tokens are the two end nodes. For a partial-stroke, it is also the two end nodes to be considered where connection is possible with other partial-strokes in forming a stroke. In this, the terminology describing connection is shown in the table in Figure 6.9.

It is necessary to define the connection among strokes as this information is needed to control the search process so as to prevent a stroke that consists of multiple tokens be mis-identified as several strokes been connected together each consisting of a single token. In this, a somewhat arbitrary definition of head, middle and tail of a stroke is used, which is sometimes not as obvious as those of partial-stroke, such as examples in Figure 6.10. This, though arbitrary and primitive, appears to be adequate for our purpose; and of course, may be further improved. We thus have the table in Figure 6.9.

Token/Secondary-token	Head	Tail
Head	h-h	h-t
Tail	t-h	t-t

Partial-stroke	Head	Tail
Head	h-h	h-t
Tail	t-h	t-t

Stroke	Head	Middle	Tail
Head	h-h	h-m	h-t
Middle	m-h	m-m	m-t
Tail	t-h	t-m	t-t

Figure 6.9 Types of Connections

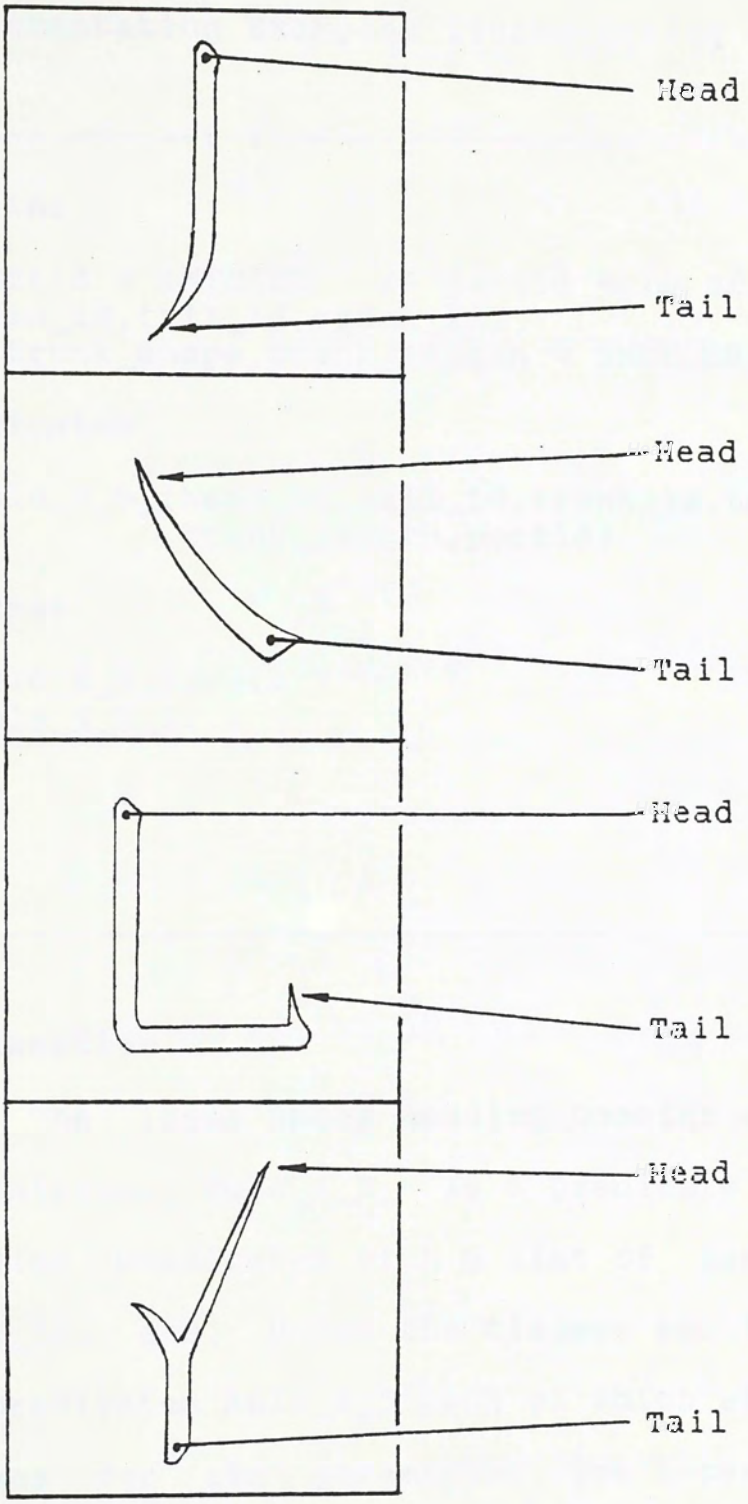


Figure 6.10 Head, Middle, Tail of Strokes

Implementation Examples (Turbo-Prolog syntax)

```
------------------------------------------------------------

Domains

  partid = INTEGER   /* partid = ID of partial-stroke */
  head_id,tail_id,trunk_id,
    trunk_shape,trunk_length = INTEGER

Predicates

  rule_4_5_(head_id,tail_id,trunk_id,trunk_shape,
            trunk_length,partid)

Clauses

  rule_4_5_(.....
  rule_4_5_(...
   .
   .
   .

------------------------------------------------------------
```

Explanation

The items under heading Domains are variable type definition. Rule_4_5_ is a predicate so declared under heading Predicates with a list of parameters head_id, tail_id, etc. Under the Clauses section will be a list of predicates Rule_4_5 each of which will have different values for its paramters. Don't-care type value is allowed in Prolog.

This particular predicate is actually an implementation of semantic rule number 4 and number 5.

Figure 6.11 Implemantation Example of Semantic Rules

As to the actual implementation of the semantic rules, often a simple list out of parameters (attributes) in the form of clause, a programming construct in Prolog, will do. The actual indexing and unification is automatically handled by Prolog. Figure 6.11 lists an implementation example of the semantic rules.

Now we turn to the attention of the search strategy that has been implemented. A what is called a constraint satisfaction procedure type algorithm is implemented. Note that the algorithm described earlier in Chapter Four about Level One is also one of this type of algorithms, though the particular implementation made the kind of presentation chosen in Chapter Four possible without resorting to a broad classification as is done here.

The name constraint satisfaction procedure is a very loose classification that in itself does not indicate the underlying search strategy that may have been applied. In general, it is defined as in Figure 6.12 following closely to that given in Rich [33], full details will not be repeated here.

Constraint Satisfaction Procedure

General Form:

```
    Begin
1.    Repeat
2.        Select an unexpanded node of the search graph.
3.        Apply the constraint inference rules to the selected node to generate all possible new constraints.
4.        If the set of constraints contains a contradiction, then report that this path is a deadend.
5.        If the set of constraints describes a complete solution, then report success.
6.        If neither a contradiction nor a complete solution has been found, then apply the problem space rules to generate new partial solutions that are consistent with the current set of constraints. Insert these partial solutions into the search graph.
      Until (a complete solution is found) or
            (all paths have led to deadends).
7.  End of Procedure.
```

Figure 6.12 A Constraint Satisfaction Procedure

It is known that every search process can be regarded as a traversal of a directed graph in which each node represents a problem state; and each arc represents a relationship between the states represented by the nodes it connects. Given the often astronomical number of problem states, it is generally true, except in some trivial cases, that constructing the entire problem space then finding the path that can lead one from the initial state to the goal state, or vice versa, is impossible or impractical. Therefore, most search algorithm involves representing the graph implicitly in the rules and generating explicitly only those when needed for testing. In our case, we are initially given a set of tokens from Level Two that are known to belong to a Chinese character, the initial state. The goal is to find out the actual number and types of strokes involved, the goal state. The rest of the states may consist of some combinations of strokes, partial-strokes, tokens or secondary-tokens. The steps of moving from state to state may sometimes involve application of the production rules defined above.

If the problem state is attached with a set of constraints that changes as the pieces of the problems are solved and the search mechanism is built as able to manipulate this list, then the number of states need to be visited can be greatly reduced. We have applied the constraint in several levels. At the highest level, those strokes identified in the search process will restrict the types of strokes that follows that have connections with them. This is a condition mentioned before. At next level, a partial-stroke identified have a set of partial-strokes as target that must be found connected so as to form a stroke. At the lowest level, partially accomplished by Level Two where linked lists of tokens according to labels have been compiled, testing of a particular token will restrict the list of token that may be tested to form a partial stroke.

The search algorithm as implemented is shown in Figure 6.13. It is written in a procedure form so that the method is easier to follow. In actual implementation, the program looks more like a list of goals with the order similar to that shown in Figure 6.14 and can be easier related to the production rules defined earlier.

Program Level Three

Input

A list called Token List from Level two.
A list called Label List from Level Two.
A list called Link List from Level Two.

Output

A list of strokes, or failure.

Method

```
Definition --- TL: token list
               BL: label list
               NL: link list
               T,T': a token belongs to TL
               S: a stroke
               SL: stroke list
               K,K': a secondary-token
               -->: to become
               ==>: to replace
               KL: list of secondary-tokens
               P,P',P": a partial-stroke
               ML: list of strokes that are
                   inter-connected
               ?C: represents a list of tokens
                   connected to the quantity
                   represented as ?.
               Error: a flag to indicate error
```

```
     Begin
1.     Load TL, BL, NL and insert them to database.
2.     Repeat
3.       Given (T, TC : T not used in ML,
                T not used by any S in database) do
           Begin
4.           If T --> K then
               Begin
5.               Insert K to KL.
               End.
6.           For all T' in TC do
               Begin
7.               If T + T' --> K' then
                   Begin
8.                   Insert K' to KL.
9.                   K' ==> T,
                     K'C ==> TC,
                     do 3.
                   End.
               End.
           End......to be continued
```

Figure 6.13a   Program Level Three in Procedure Form

Continue...

```
10.     For all K in KL do
          Begin
11.         If K --> P then
              Begin
12.             Insert P to PL.
              End.
          End.
13.     Given (P, PC : P in PL) do
          Begin
14.         If P --> S then
              Begin
15.             Insert S to SL.
              End.
16.         Using PC ==> TC,
            do 3 to 10 to obtain PL' and do
              Begin
17.             For all P' in PL' do
                  Begin
18.                 If P + P' --> S then
                      Begin
19.                     Insert S to SL.
                      End.
20.                 Using P'C ==> TC,
                    do 3 to 10 to obtain PL" and do
                      Begin
21.                     For all P" in PL" do
                          Begin
22.                         If P + P' + P" --> S then
                              Begin
23.                             Insert S to SL.
                              End.
                          End.
                      End.
                  End.
              End.
          End.
24.     Given S with the most # of T, SL : S in SL do
          Begin
25.         Repeat
              Insert S to ML.
26.           Using SC ==> TC, do 3 to 23 to get SL'.
27.           Given (S'with the most # of T,
                     SL' : S' in SL') do
                Begin
28.               If S' acceptable given ML then
                    Begin
                      Insert S' to ML.
29.                   S' ==> S,
                      S'L ==> SL,
                      do 24.
                    End....... to be continued
```

Figure 6.13b   Program Level Three in Procedure Form

Continue...

```
30.                     Else if other S' in S'L not yet
                          exhausted then
                          Begin
31.                         using new S', do 28.
                          End.
32.                     Else
                          Begin
33.                         Error := true.
                          End.
                      End.
34.               Until (all tokens used) or (Error).
                End.
35.         If not Error then
              Begin
36.             Insert ML into database.
37.             If not all tokens used then
                  Begin
38.                 do 2.
                  End.
              End.
39.       Until (all tokens used) or (Error).
40.       If not Error then
            Begin
41.           Save all S in database as result
            End.
42.  End of Program.
```

Figure 6.13c Program Level Three in Procedure Form

Program Level Three

List of Goals ---

character : to find all strokes from the list of tokens given

multi-stroke : to find a group of strokes that are inter-connected

stroke : to find a stroke

partial-stroke : to find a partial stroke

secondary-token : to find a secondary-token

token : get a token from the list of tokens given

Relations between Goals ---

1. character --> multi-stroke
2. multi-stroke --> stroke & multi-stroke
3. multi-stroke --> stroke
4. stroke --> partial-stroke
5. stroke --> partial-stroke & partial-stroke
6. stroke --> partial-stroke & partial-stroke & partial-stroke
7. partial-stroke --> secondary-token
8. partial-stroke --> token
9. secondary-token --> token & secondary-token
10. secondary-token --> token & token

Note: '&' = logical AND

Figure 6.14 Program Level three as A List of Goals

The search process implemented is bottom-up, or reasoning forward from initial state, which is also the reason for using synthesized attributes as they are computed from attributes of lower level quantities (terminals or non-terminals). From Figure 6.13, it can be seen that the program attempts to construct a stroke in the following sequence:

token --> secondary-token --> partial-stroke --> stroke.

The order of which the production rules are applied may be difficult to see in Figure 6.13 but is more apparant from Figure 6.14.

A search sequence is illustrated in Figure 6.15. Note that in intermediate steps, all possible solutions of secondary-tokens, partial-strokes and strokes are generated before further testing. These are breadth-first processes. Note also that by generating a linked list at Level two, the number of tokens to search at each node can be reduced. The choose of strokes with the most number of tokens to start testing takes advantage that well connected tokens have the highest possibilities to form a single stroke instead of several strokes, this is the only heuristic rules applied in the search process.

Although not mentioned before, it may have been obvious that this implementation does not allow a token be shared by different strokes. It should be clear by now that the program will require major modification and be more complicated in order to handle this rare occurance which happens mostly when resolution is too low so that pattern is distorted. We shall have an example of this in Chapter Eight.

Many recognition systems have explicit training mode to collect automatically the reference data needed for their algorithm. In this case, there is no explicit training process. The author has analyzed about 200 character patterns, which are chosen from a small library used previously by students working on data compression, to extract the set of control rules described above. This set of rules can be expanded if more characters are analyzed.

This completes the algorithm description as far as stroke extraction is concerned.

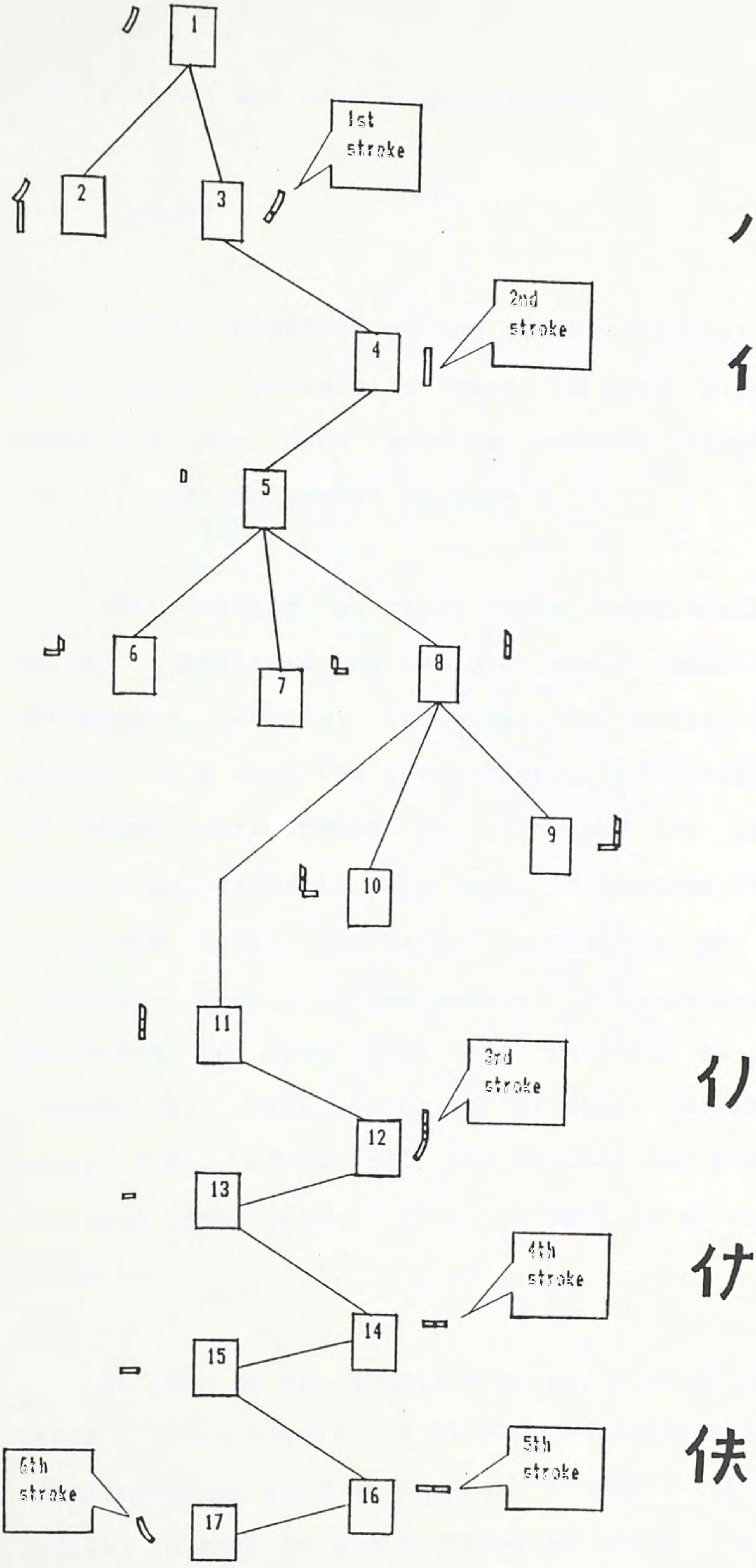


Figure 6: A Typical Search Sequence

7.0 SCALING AND FONT TRANSFORMATION

7.1 Scaling

Taking advantage of the stroke information obtained from stroke extraction level, a very simple scaling algorithm has been used to produce relatively good quality scaling result (Figure 7.1).

The scaling process works individually on each stroke identified by the previously described stroke extraction process. In order to scale a character pattern to a required resolution, individual parts each of which corresponds to a stroke in its original resolution, 24x24 in this case, is doubled in resolution in steps until the target resolution is reached or exceeded. Then, if necessary, a subsampling step is performed to fine tune the pattern to the target resolution. This doubling process is adopted from Casey [4,5]. After all the strokes are scaled to the required resolution, they are ORed to obtain a scaled character.

In each of the doubling steps, a pixel conceptually becomes four pixels. A direct multiplication without adjustment, as is done in some commercial systems, will result in some coarse patterns as shown in Figure 7.2. The amount of distortion can be severe if the base pattern is of very low resolution.

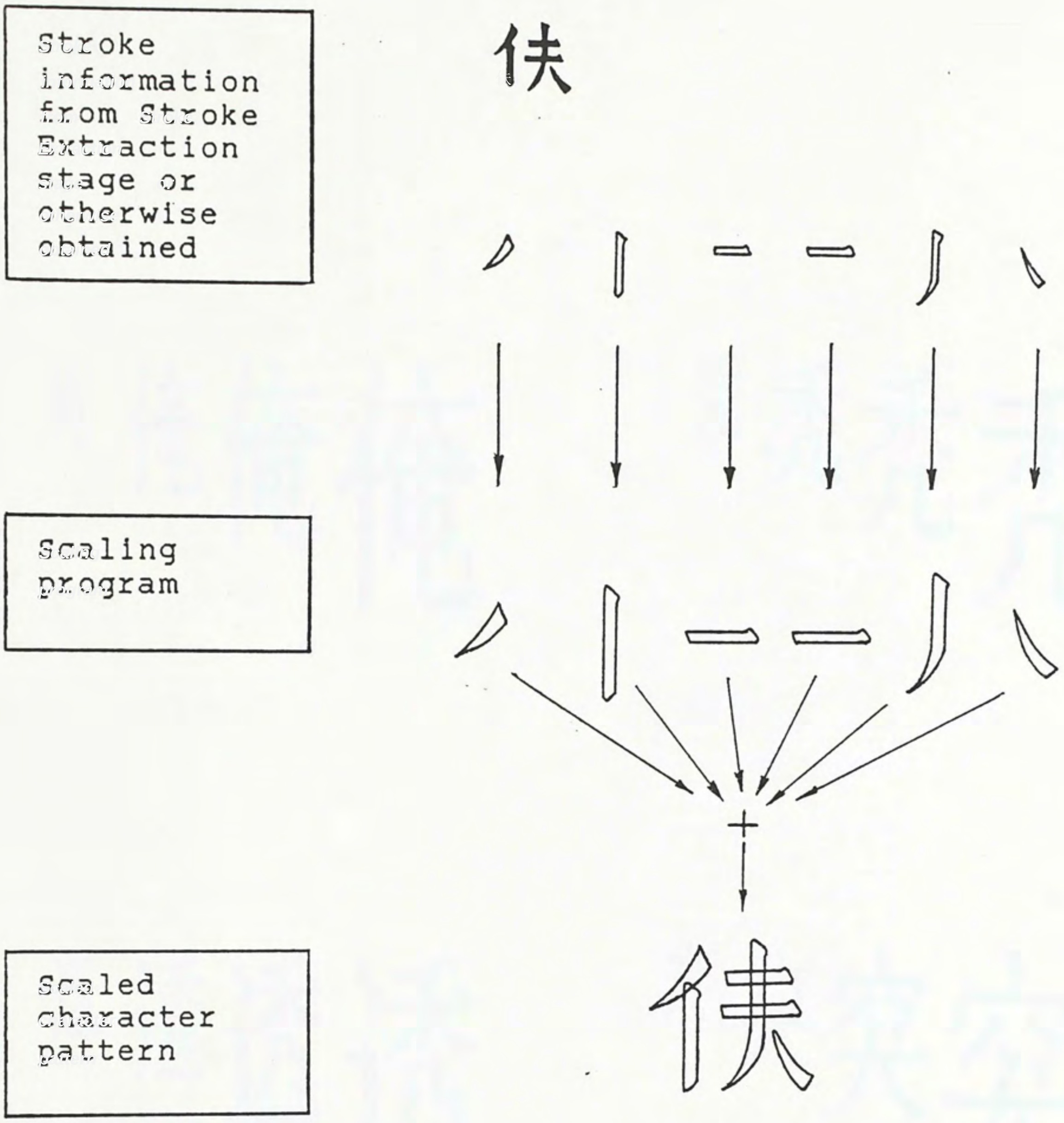


Figure 7.1 Stroke Based Scaling of A Chinese Character

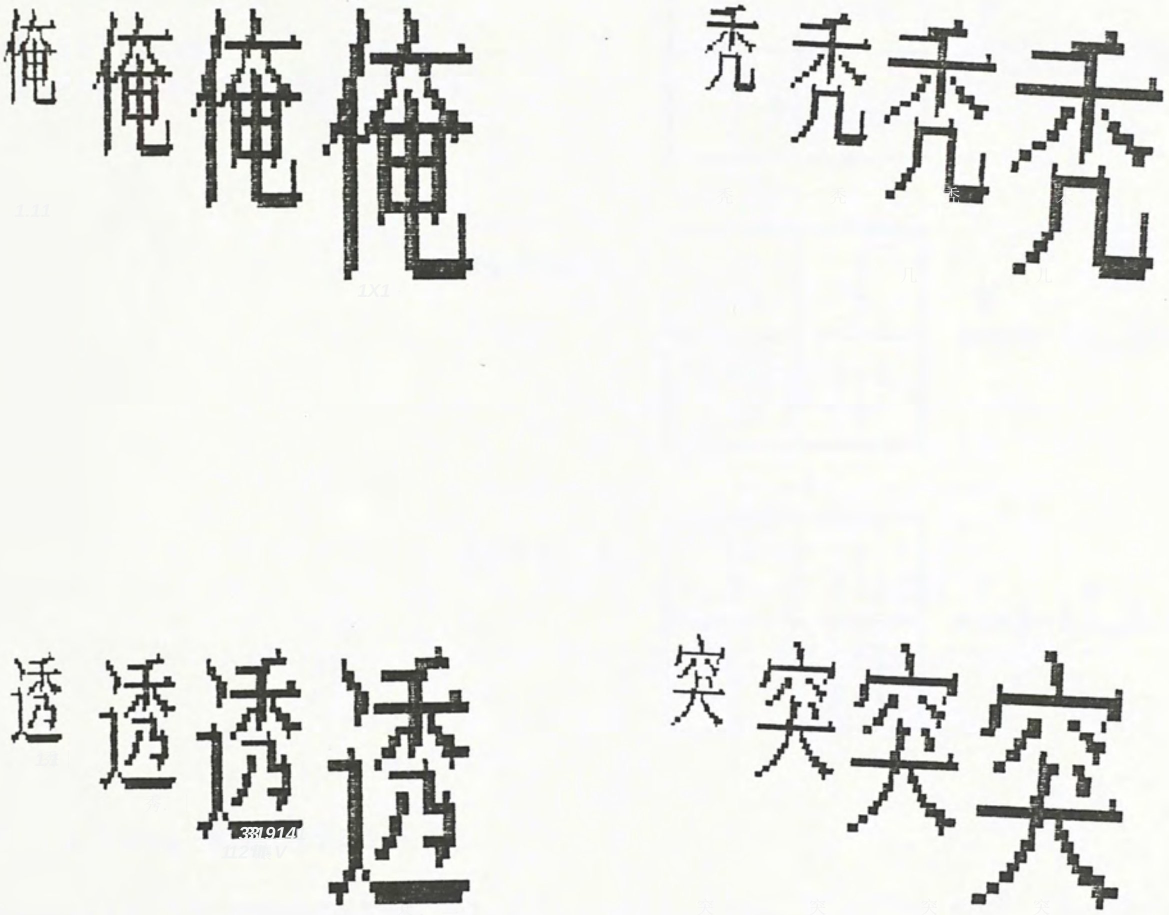


Figure 7.2   Distorted Character Pattern Example

i-1 j-1	i-1 j	i-1 j+1
i j-1	i j	i j+1
i+1 j-1	i+1 j	i+1 j+1

Pixel under consideration is (i,j)

Group 1.

i-1 j-1	i-1 j
i j-1	i j

$\xrightarrow{F} (i,j)_1$

Group 2.

i-1 j	i-1 j+1
i j	i j+1

$\xrightarrow{F} (i,j)_2$

Group 3.

i j-1	i j
i+1 j-1	i+1 j

$\xrightarrow{F} (i,j)_3$

Group 4.

i j	i j+1
i+1 j	i+1 j+1

$\xrightarrow{F} (i,j)_4$

F is some function

Pixel(i,j) generates $(i,j)_1$, $(i,j)_2$, $(i,j)_3$ and $(i,j)_4$ in each doubling step

(i,j) --->

$(i,j)_1$	$(i,j)_2$
$(i,j)_3$	$(i,j)_4$

Figure 7.3 Some Smoothing Method

i-1 j-1	i-1 j	i-1 j+1
i j-1	i j	i j+1
i+1 j-1	i+1 j	i+1 j+1

Pixel under consideration is (i,j)

Pixel(i,j) generates $(i,j)_1$, $(i,j)_2$, $(i,j)_3$ and $(i,j)_4$ in each doubling step

$$(i,j) \;\; \text{--->} \;\; \begin{array}{c|c} (i,j)_1 & (i,j)_2 \\ \hline (i,j)_3 & (i,j)_4 \end{array}$$

using the following table

Ratio (type) of (i,j) to other three surrounding pixels (Fig. 7.3) of different type	(i,j) = 1		(i,j) = 0	
	$(i,j)_1$, $(i,j)_4$	$(i,j)_2$, $(i,j)_3$	$(i,j)_1$, $(i,j)_4$	$(i,j)_2$, $(i,j)_3$
1 : 3	0	1	0	1
1 : 2	1	1	0	0
1 : 1	1	0	1	0
1 : 0	1	1	0	0

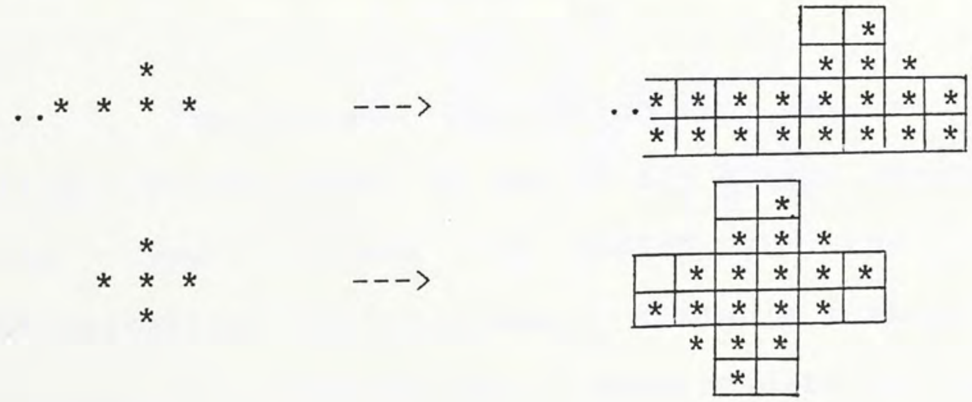


Figure 7.4 Smoothing Method Used

The simplest of the approaches to smooth the edges is by averaging. The eight neighbours of a pixel may be divided into four groups (Figure 7.3) in which each group has three adjacent members which will determine the state of one of the four new pixels to be generated. If they observed the same formula such as by majority voting, the resulting pattern would lost some features at some boundary points in which a pixel which was black and was surrounded by three or more pixels of the opposite type would be lost. However, this pixel actually carries more information than other pixels that are surrounded by pixels of the same type and thus should not be totally lost. This kind of simple approach clearly would penaltize pixels that carry more information. If the rule were relaxed a bit and favored black pixels, the resulting stroke would simply become too thick. To improve this situation, a table (Figure 7.4) is devised to determine the outcome of a multiplication. This is found to be able to preserve certain sharp edges as shown, and is used in our program.

If the target resolution is not an integral multiple by the power of two of the source resolution, a scale down process is needed at the end of multiplication. In this case, a simple subsampling is sufficient to achieve this. Some results are shown in Figure 7.5. The method described above clearly is very primitive. Ways to improve it will be discussed in Chapter 9.

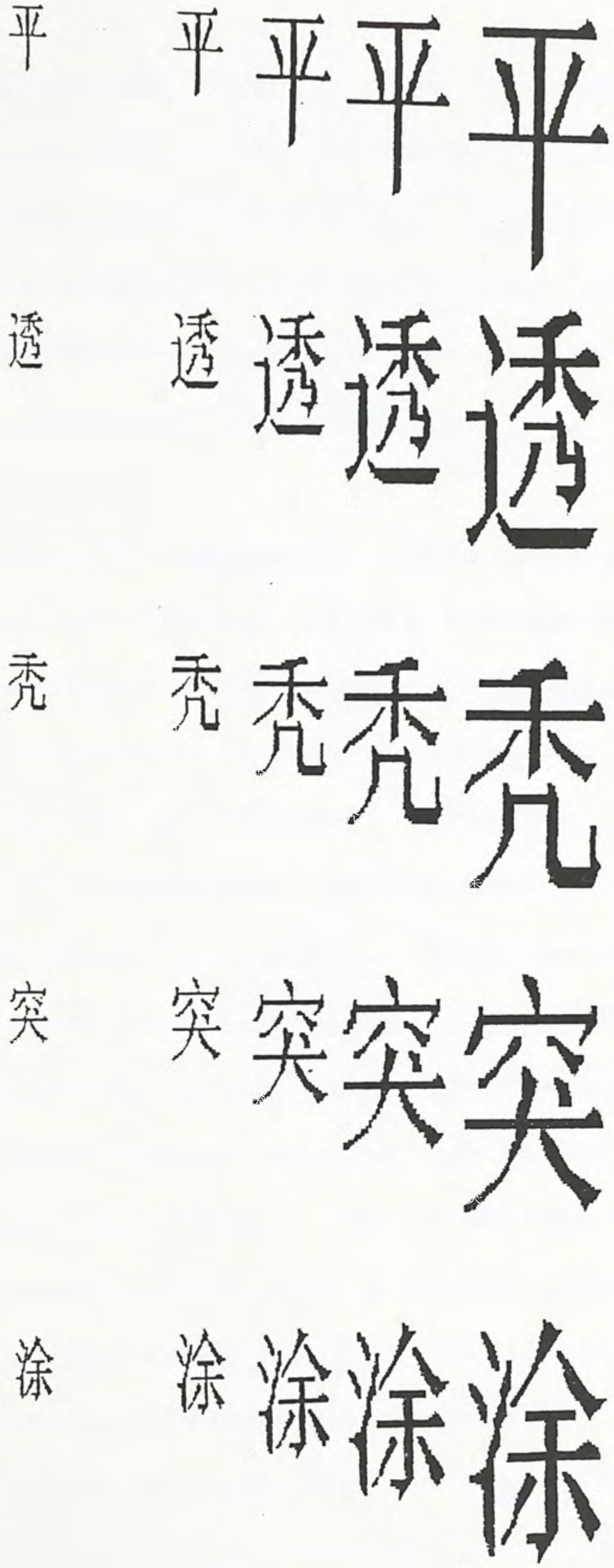


Figure 7.5 Some Scaling Result

7.2 Font Transformation

The approach here relies on a set of prestored patterns which are either a stroke or part of a stroke, and a control table that holds information as how these patterns are combined to form strokes; together with the stroke information provided by Level Three, to re-assemble a character to a new style.

A powerful approach that employs high order polynomials to define stroke boundaries may be the ultimate solution to the problem of character generation or shape alteration that are accurate, flexible and of good quality. It is, however, difficult to mechanize the selection of reference points that are required to control the shape of a curve. Human assistance appears to be the norm. The information extracted from the basic character pattern in our case provides little additional help at this respect as only the stroke size and the stroke type are known. An alternative approach is therefore used that defines a stroke appearance based on the stroke size and position. The quality of a stroke thus produced in general is inferior to the polynomial method if the pattern must be also scaled to other resolution which may destroy some of the asthetic quality of the character pattern which is exactly as designed only at the upper bound resolution. Figure 7.6 shows the principle.

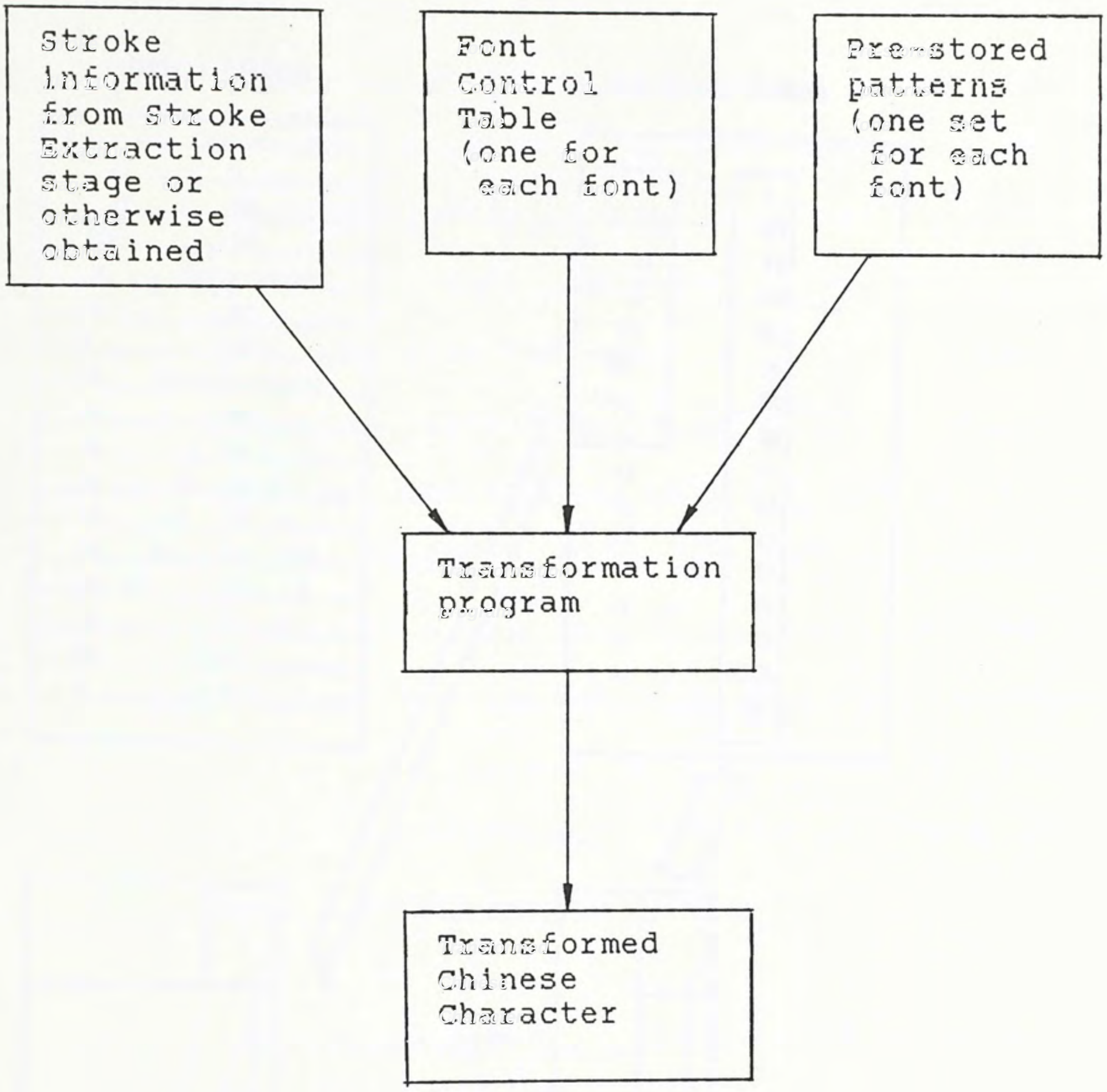


Figure 7.6 Transformation

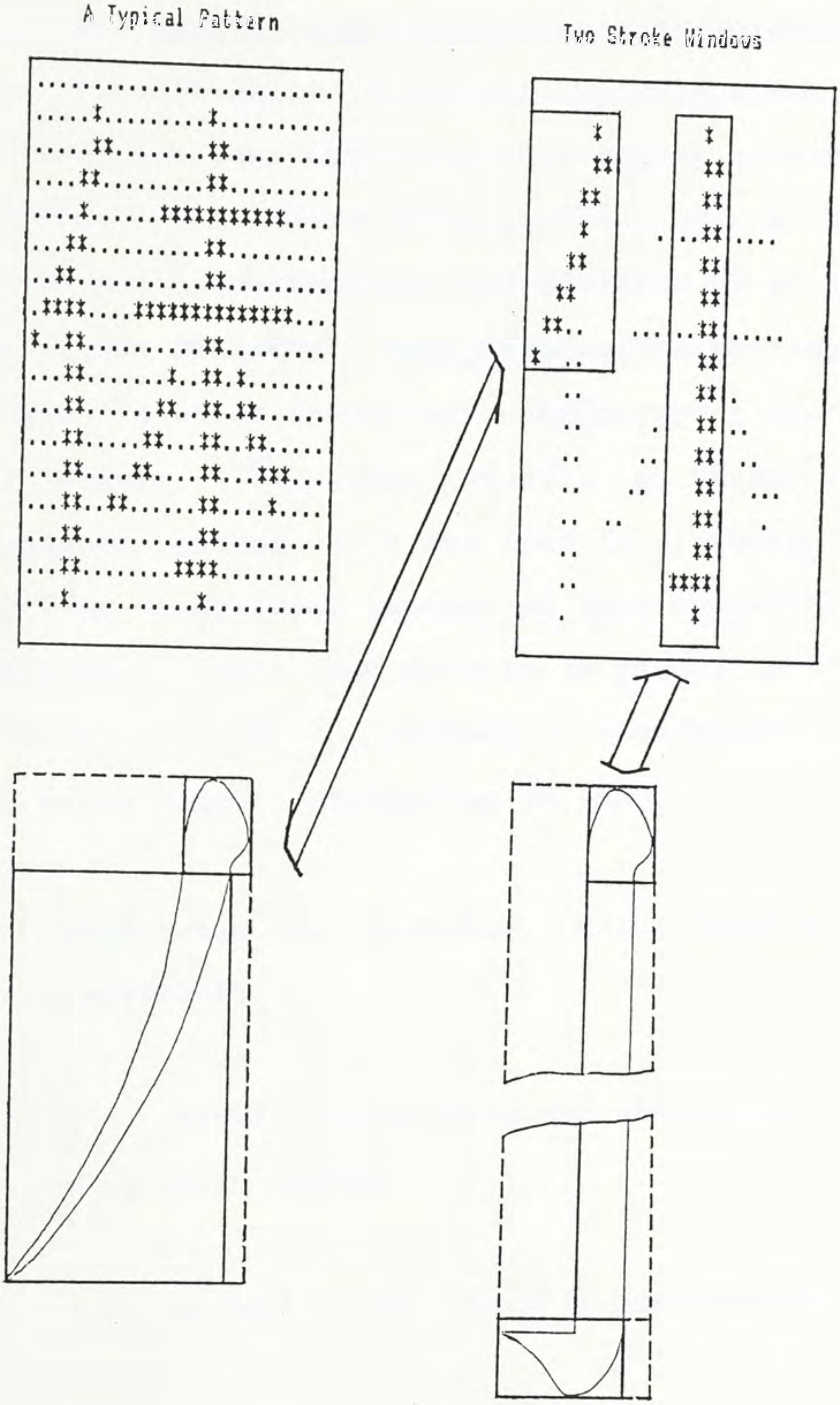


Figure 7.7 Stroke Windows

In this method, a set of patterns is stored for each font that is to be used. These patterns are normally portions of a stroke at their highest possible resolution (128x128 in current implementation). All the strokes are constructed by selectively combining these patterns. The size and position of a stroke is determined from the size and position of this stroke in the source character. This defines the working area of a stroke in which the constructed patterns are to be fit in (Figure 7.7). This we call a stroke window. A character pattern of a new font is produced by adding all the constituent strokes of this character together. Additional scale down process is needed if the target pattern is of a different resolution (smaller). Currently simple subsampling is used.

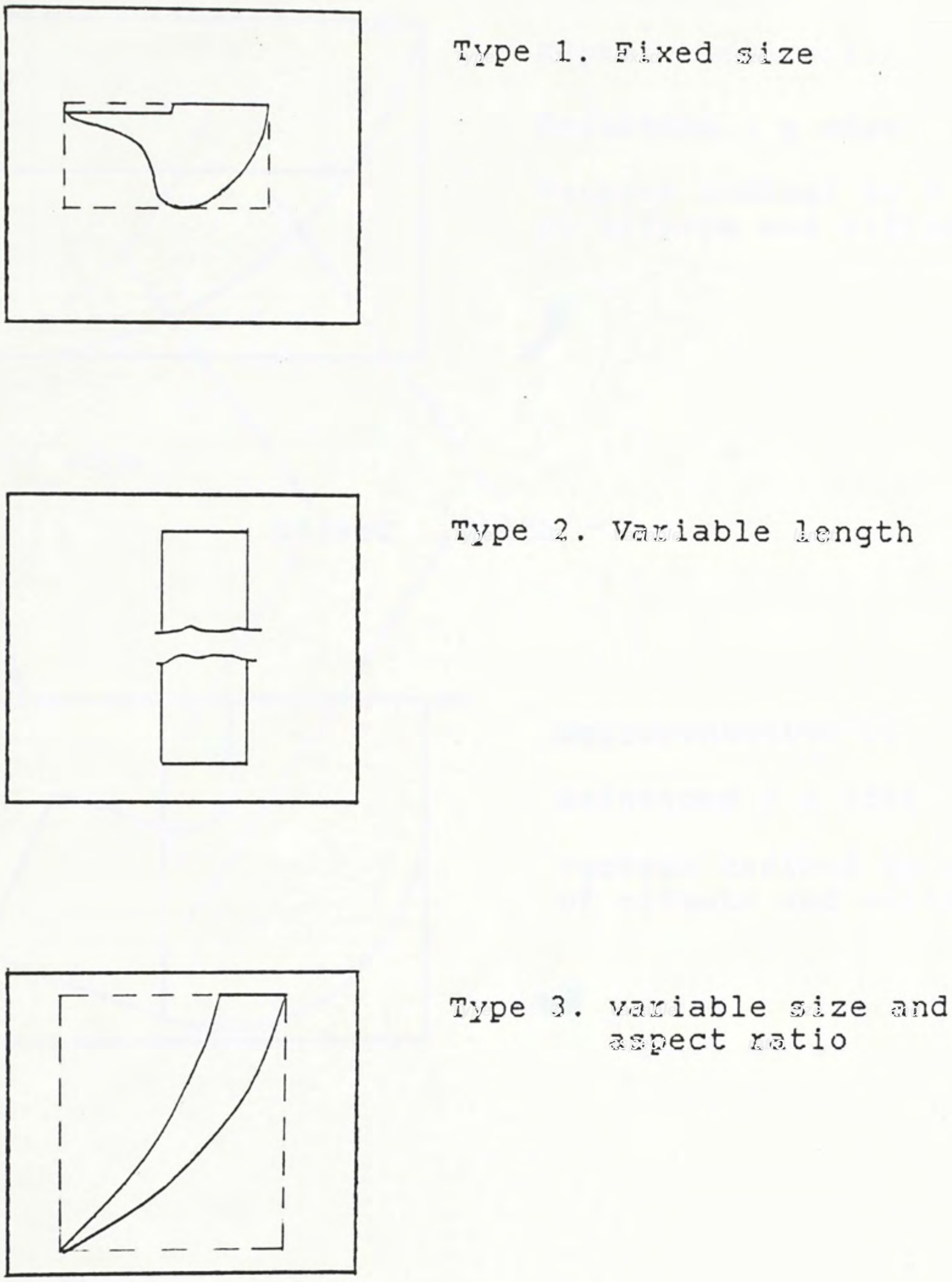
After some investigation, three types of patterns are classified:

1. pattern whose size and shape are independent of stroke windows;

2. pattern whose size is dependent on one axis only;

3. pattern whose size and shape are functions of both sides of a stroke window (Figure 7.8).

Type one are patterns that constitute small features of a stroke. Type two are patterns that are straight whose width are independent of stroke windows. Type three are patterns that are inclined which must be shaped according to different window sizes and different aspect ratios. The number of patterns thus classified that form a complete font is font dependent. Each pattern is stored as a list of offsets and widths (Figure 7.9). A total of two fonts have been produced Figure 7.10 shows one. In the following, a pattern is also called a component.



Type 1. Fixed size

Type 2. Variable length

Type 3. variable size and
aspect ratio

Figure 7.8 Three Types of Basic Patterns

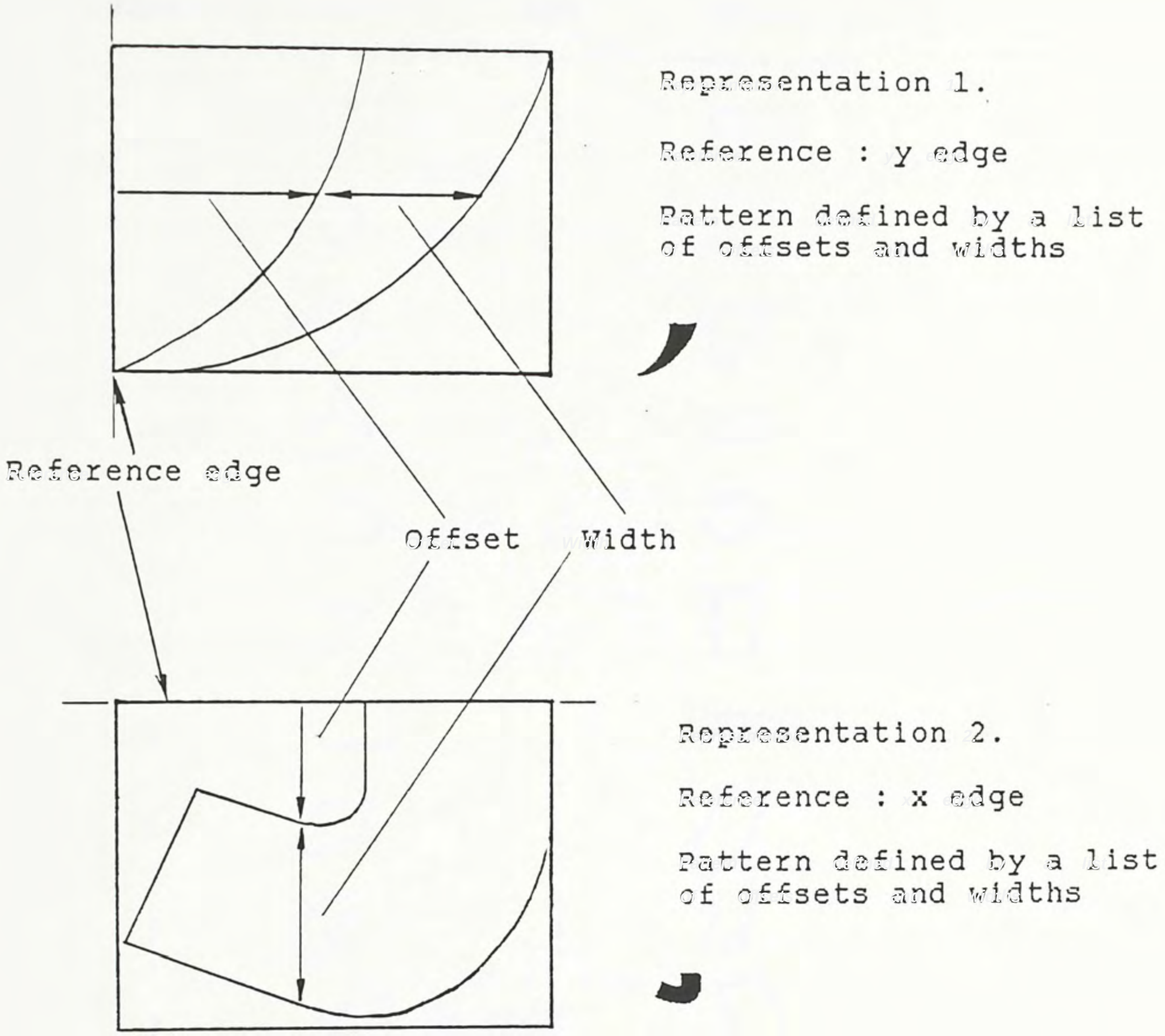


Figure 7.9 Representation of A Pattern

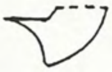
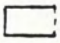
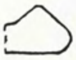
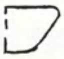
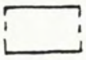
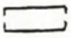
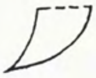
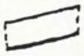
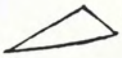
type	Pattern	type	Pattern
1		1	
1		1	
1		1	
1		1	
1		1	
1		2	
2		2	
3		3	
3		2	
2		3	
3		3	
3		3	
3		3	
3		3	
3		3	
3		3	

Figure 7.10 Patterns of A Font

```
                 Font Control Table
----------------------------------------------------

     Number of Stroke

     A list of record M for each stroke


     Record M

          Begin
            Stroke name.
            Overall size X.
            Overall size Y.
            Number of record N (pattern)
            A list of record N
            Number of record C (check list record)
            A list of record C
          End of record M.

--------------------------------------------------


     Record N

          Begin
            Pattern name.
            Reference side X1.
            Offset to X1.
            Reference side Y1.
            Offset to Y1.
            Reference side X2.
            Offset to X2.
            Reference side Y2.
            Offset to Y2.
          End of record N.


     Record C

          Begin
            Connection type.
              (one of h-h,h-m,h-t,t-h,t-m,t-t as
               in Level three)
            List of stroke to check.
            Pattern name to replace.
          End of record C.
```

Figure 7.11   Structure of A Font Control Table

The font patterns are stored as individual files and are loaded only when needed. This reduces the memory demand at run time. The information as what patterns will form a particular stroke is stored in a separate file which will be loaded to act as a control table, the structure of which is shown in Figure 7.11. The entries are:

1. a list of stroke names;

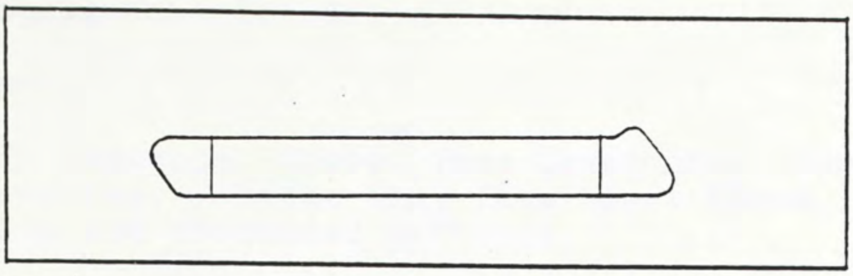
2. under each stroke name entry there is a list of component names (pattern name) that form this stroke, and a list called check list that indicates that if this stroke is connected to other strokes, then a different component may be used to replace the first or last component in the component list, depending on the type of connection (Figure 7.12);

3. under each component entry there is information that defines the location of this component inside the stroke window;

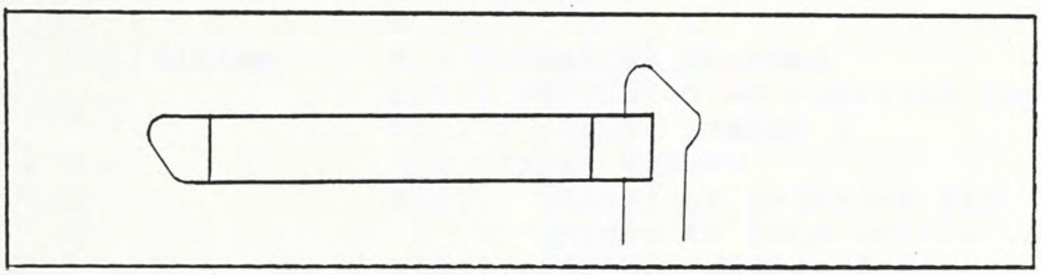
4. under each entry of the check list there defines the type of connection, the name of strokes concerned and the name of the pattern to be replaced.

The program operates in the following sequence (Figure 7.13). The stroke information is first loaded into memory. Together with the sequence table created by Level One and the token list created by Level two, the stroke window for each stroke of the character can be computed. Then based on the information provided by the font control table, component windows are defined and the corresponding component patterns are loaded into memory and plotted. Finally, after all the strokes are plotted, they are combined to form a character pattern with a specific font characteristics.

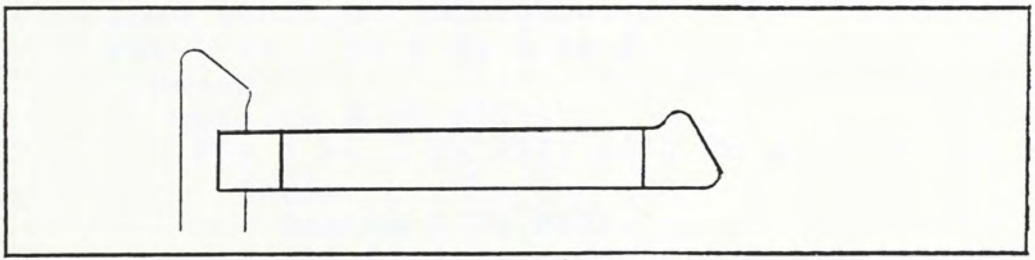
During the process, once a stroke window is defined, the next job is to outline the area and location in which each associated pattern is to be placed. The defined area is called a component window (Figure 7.13). Component windows may overlap and are positioned relative to one of the four sides of the stroke window as reference. The component type is with the component file and therefore is not included into the control file.



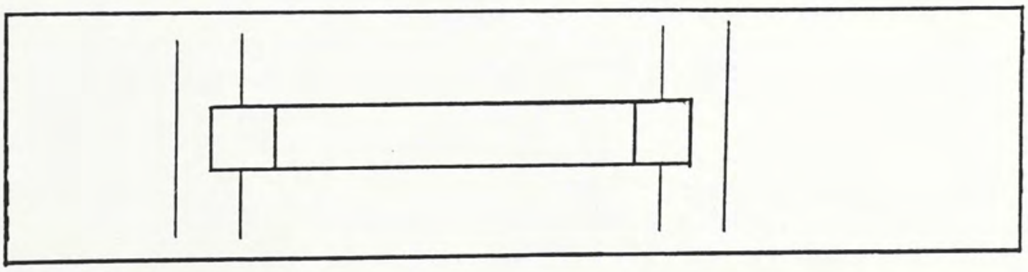
Normal Structure



If connected at the tail



If connected at the head



If connected at both ends

Figure 7.12 Component Change Example

Program Transformation

Input

Sequence table from Level One. Token list from Level two. Stroke list from Level Three. Font control table and prestored patterns.

Output

A character pattern of a specified font

Method

```
Definition --- N : number of strokes
               I,J : variables as counters for loop
               S(I) : stroke number I
               W : stroke window
               M(I) : number of patterns that are
                      needed to form stroke S(I)
               P(J) : pattern number J
               C : component window (Figure 7.13)
```

```
     Begin
1.     Load input data.
2.     Prompt for font name.
3.     Load font control table.
4.     For I := 1 to N do 4 to 8
         Begin
5.         Define W of S(I).
6.         For J := 1 to M(I) do 7 to 8
             Begin
7.             Define C of P(J).
8.             Fit P(J) into C
             End.
         End.
9.     If resolution not 128x128 then
10.      Subsampling.
11.  End of Program.
```

Figure 7.13 Program Transformation

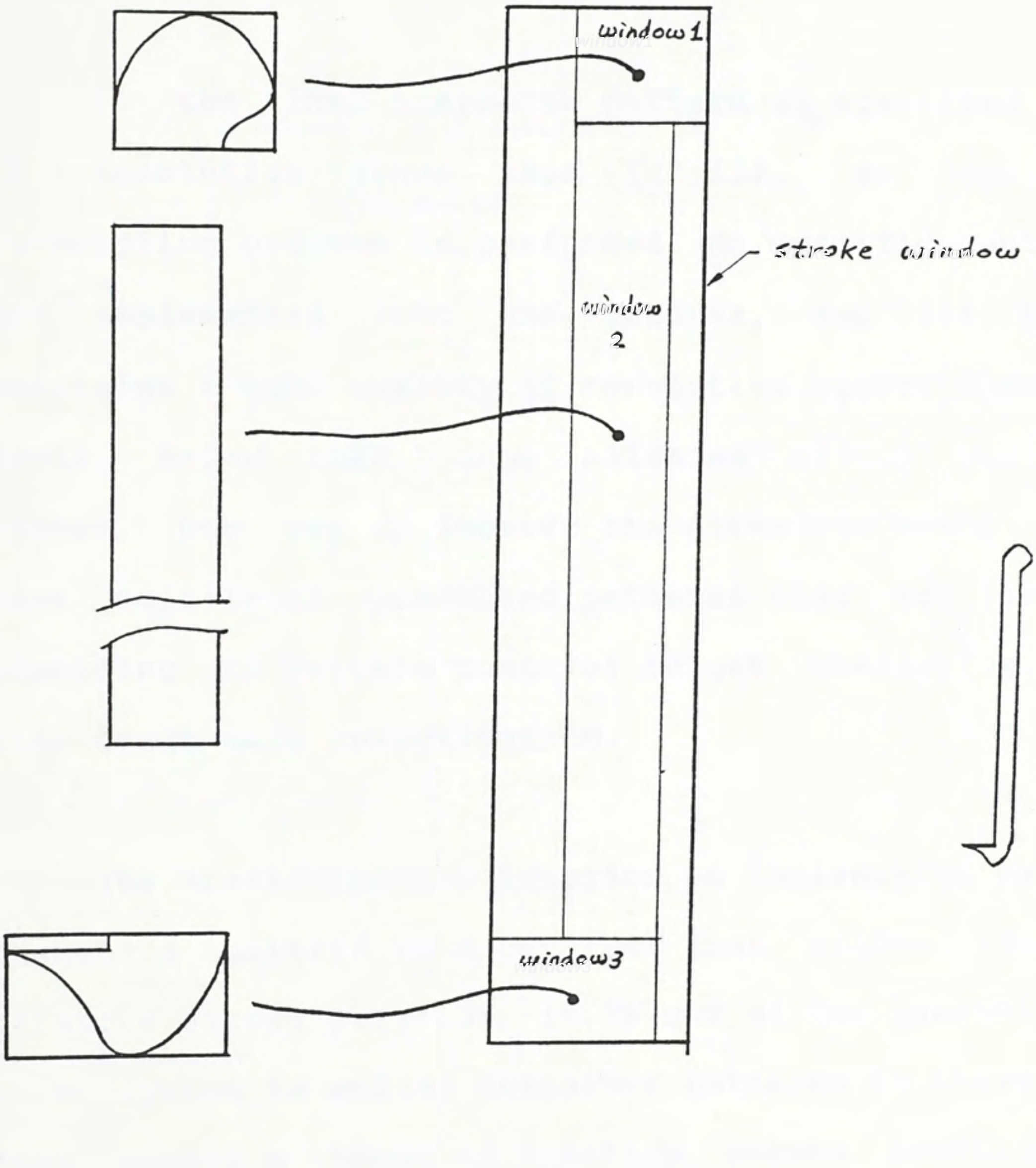


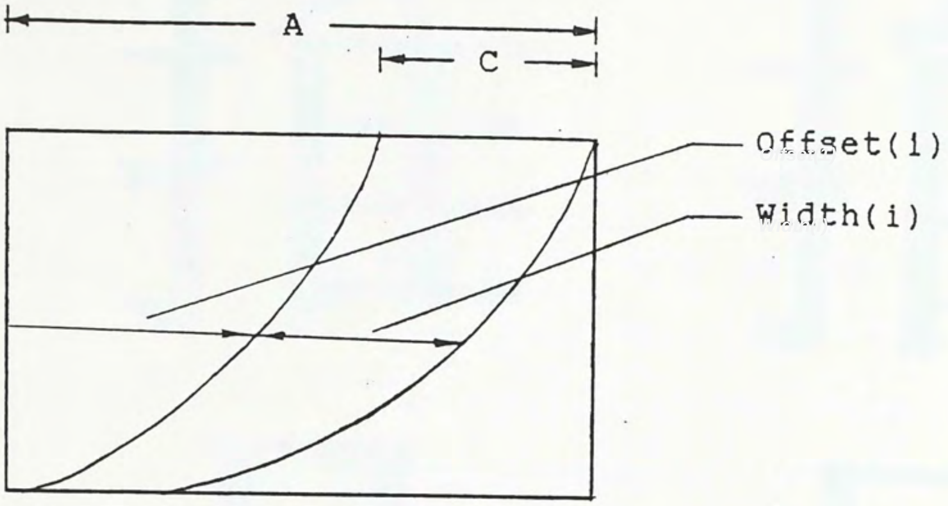
Figure 7.14 Component Windows

After a component window is defined, the pattern to be fit into the window may need adjustment. For example, if the pattern has been constructed of a size of 50x50, and the component window is of a size 50x30, then the pattern must be adjusted. The adjustment method is shown in Figure 7.15.

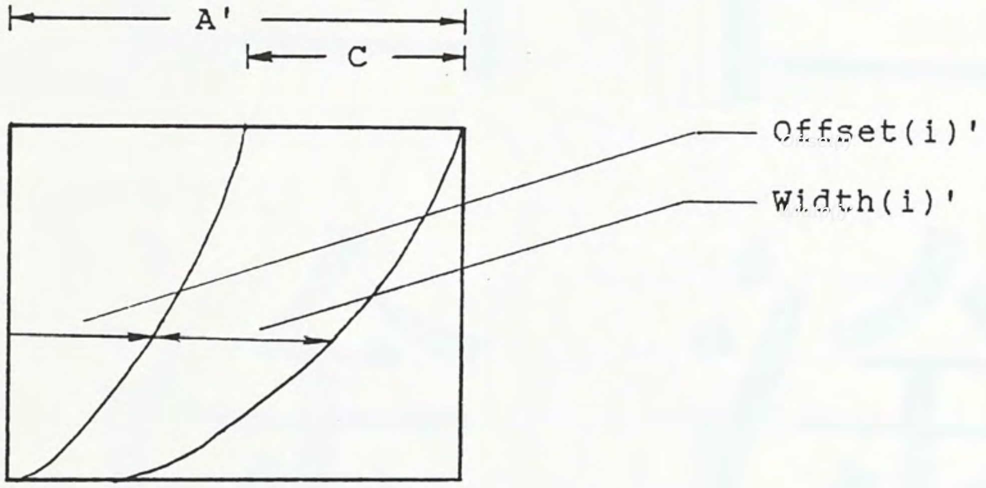
If the final character pattern is specified to be of resolution other than 128x128, an additional subsampling process is performed. No specific control is yet implemented into the process, and the pattern maintains a good quality at resolution higher than about 48x48. Below that, some aliasing effect begins to appear. One way to improve the situation would be to have multilevel prestored patterns that are selected according to certain range of target resolution. This area needs more investigation.

The transformation function so implemented may only change a pattern to a new font that allows identical relative stroke position. It is out of the capability of this system to modify character patterns to those fonts that demand a change of relative stroke position, or more radically, a change of character structure.

Some results are shown in Figure 7.16.



Pattern as Constructed



Window to be fit-in

Method

$$\text{Offset}(i)' = \text{Offset}(i)\ \frac{A' - C}{A - C}$$

$$\text{Width}(i)' = \text{Width}(i)$$

(Similar process is done to adjust on another axis)

Figure 7.15 Adjustment Method

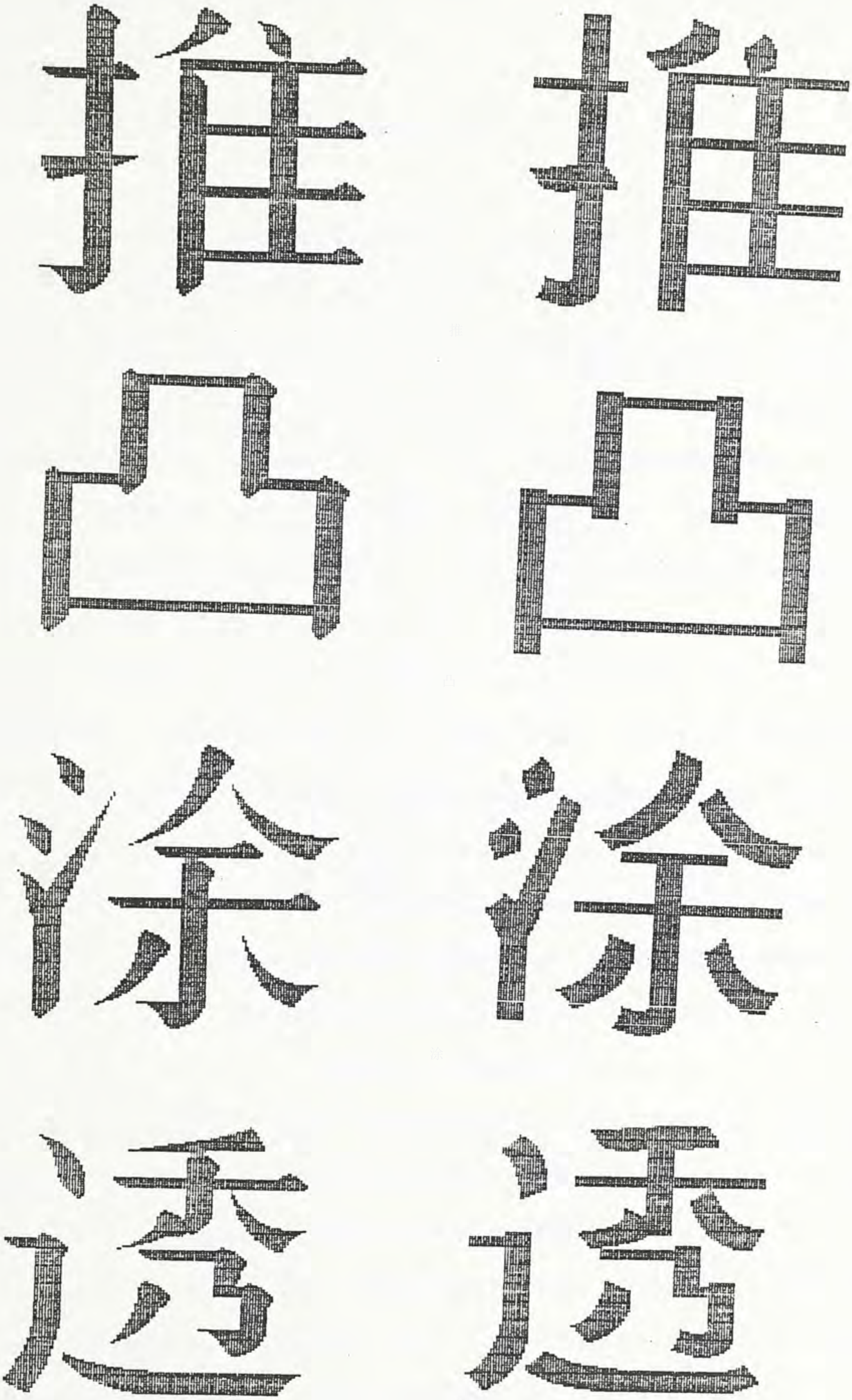


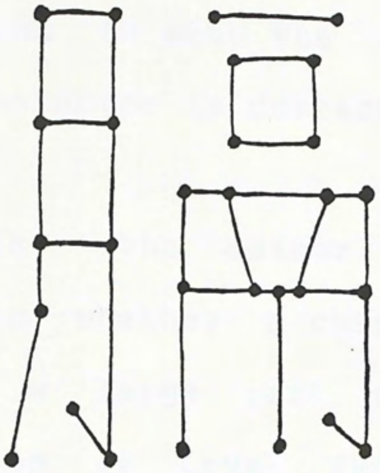
Figure 7.16 Some transformation Example

8.0 RESULT

8.1 Stroke Extraction Level One

To give a simple criteria on whether a character is correctly traced has been found to be rather difficult for the power of the analyzer at the Third Level is also a factor that must be considered. This is because the more powerful an analyzer, the less a burden is placed on the tracer at the first level and hence looser criteria may be acceptable. In ordinary pre-processing stage when the primitives chosen are sufficiently simple, there is no need of such a success or failure judgement. However, in our case that involves directly obtaining rather complex topological information from a Chinese character, there are cases that can be judged to be failures where the information produced will not be possible to allow later stages to proceed properly. Consider again the requirements given in Chapter Four. The whole idea of these requirements is that the sequences traced must consist of well formed stroke segments each of which is a part of a stroke; to this, however, no simple quantitative guidelines can be given. One would expect well formed stroke segments meaning those segments apparant to the human reader. For example, one, to those trained in Chinese character writing, may expect a graph for the character in Appendix be like that shown in Figure 8.1. However, to those not familiar with Chinese characters, the graph may be different. In practice, the program in current

Graph Expected (from knowledge)



Graph Produced (from Level One and Level Two)

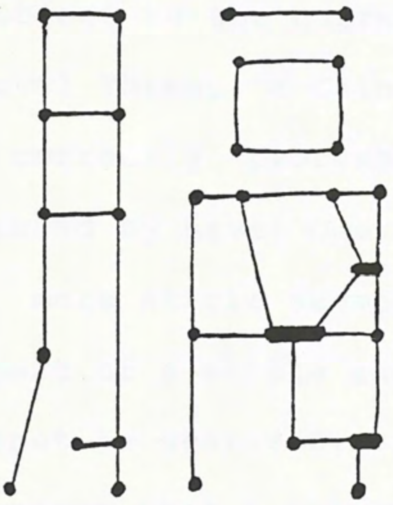


Figure 8.1 Graph of a Character

produced a more complicated graph but this does not mean the character is not correctly traced. This, probably, is sufficient to show the difficulty in determining whether a character is correctly traced.

To this, the author was forced to base his judgement on whether a character has been correctly traced in a large part on whether the current implementation of Level Two and Level Three could possibly use the information produced to find out the underlying strokes. This can be simply stated as follows: subject to the current implementation of Level Two and Level Three, a Chinese character pattern is considered correctly processed by Level One if each sequence traced by Level One from the pattern consists of one or more stroke segments each of which is the whole or a part of a stroke and only one stroke; stroke segments cannot be shared by strokes. In addition, if for some reasons that a character traced is fragmented to pieces; ie., an unnecessarily complex graph is produced, a sometimes occurance when the amount of distortion and forced connection is large; or when a case encountered had not been analyzed when the program was designed, causing the program to go astray, then the trace process is declared failure. If the graph produced corresponds well to what one may expect, and the current implementation of Level Two and Level Three can theoretically based on the information to find out the underlying strokes, then the trace is declared successful. It is to be regretted that this is rather subjective and depends heavily on one's familiarity of detailed program implementation of the three levels. A number of cases will be discussed later. More on this will be discussed in next chapter.

About one thousand characters have been tested. About 25% of them were judged to have failed this level in ways as listed below:

1. wrong sequences were produced;
2. wrong path generated at a complex node;
3. generated sequences that could not possibly lead to successful stroke extraction;
4. erred on situation not considered;
5. erred on character pattern too poorly formed.

Each of these general error modes will be individually discussed.

Case one and case two are shown in Figure 8.2 and Figure 8.3 which need no further explanation. The third case is rather interesting (Figure 8.4), a failure situation can occur when a stroke segment identified actually belongs to two connected but different strokes. The failure here is primarily due to the fact that the analyzer at Level Three is not designed to handle such a complicated case. Therefore, even though the program here did generate logical identification, the information would be insufficient to allow later level to proceed properly.

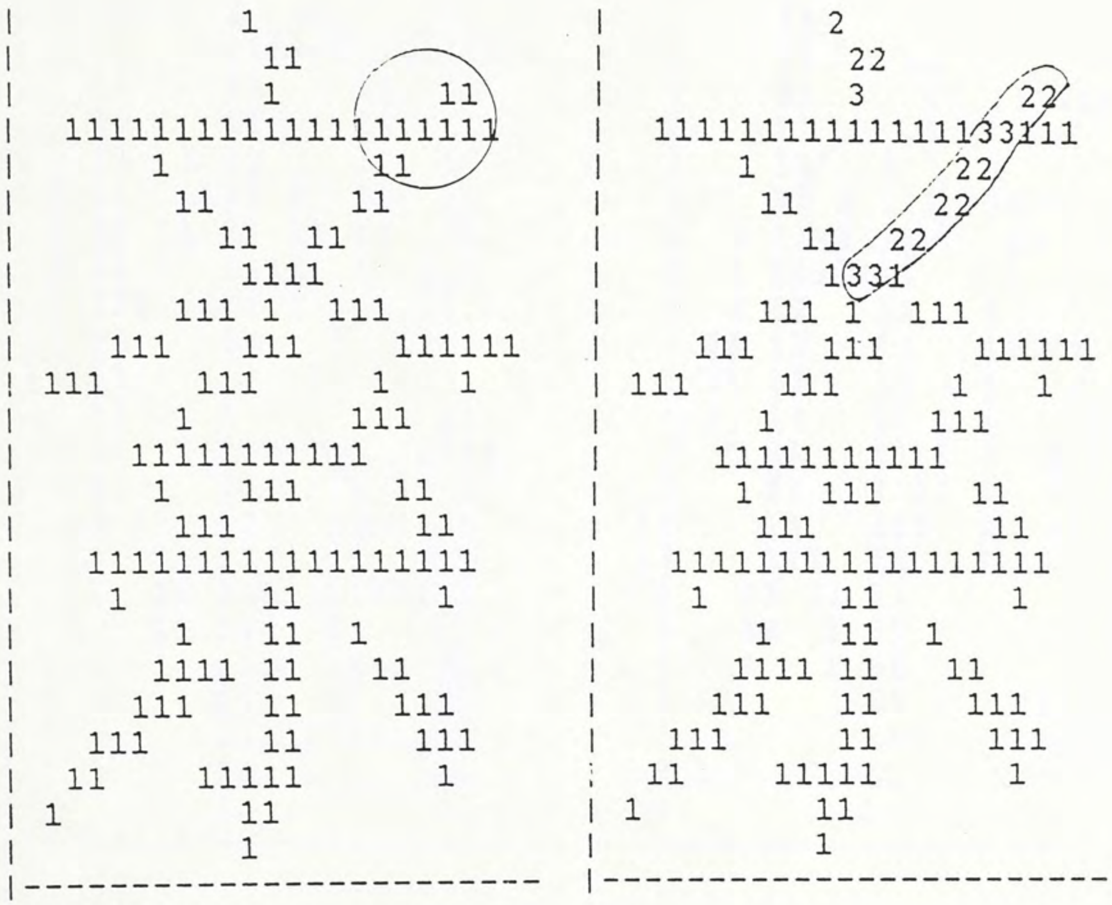
There is no example given here for case four as some of the errors as so discovered have been subsequently corrected. This may be classified as program bugs if one wishes to do so. Given the multitudinous of Chinese characters and their being of various sizes and conditions, there is always a non-zero probability that new situations might be encountered that would be out of the capability of the program currently implemented.

Another frequent failure case is when a character pattern is sufficiently complex that it must be squeezed into the small space (for a discussion on the resolution needed to faithfully represent Chinese characters, see Nagao [30]) available and hence causing forced connections and information lost; or the pattern itself is not well conditioned. The system is not designed to handle these cases, however. Figure 8.5 has examples of these. In addition, if the number of connected places are large, the program may produce wrong sequence as has been shown as case one.

Typically, a complex node does not necessarily lead to error provided sufficient prior information is collected enabling the system to make good prediction (Figure 8.6 and Figure 8.7). In the case where a node is encountered at the first few units, the program may not be able to make good guesses and hence produces wrong data (Figure 8.3). A number of cases that either causes wrong sequences be generated or the information generated will not enable later level to extract strokes correctly are shown in Figure 8.8 and Figure 8.9.

The above listed cases covered most of the errors encountered by Level One. From the examples, many character patterns only vaguely resemble the intented character shapes due to low resolutions. Without prior training on Chinese characters, it would be difficult to identify them even for a human reader.

紊



```
|          1                |          2
|           11              |           22
|           1       11      |           3       22
|  11111111111111111111     |  11111111111111133111
|      1         11         |      1         22
|       11      11          |       11      22
|         11  11            |         11  22
|          1111             |          1331
|       111 1  111          |       111 1  111
|    111   111    111111    |    111   111    111111
| 111    111     1   1      | 111    111     1   1
|       1       111         |       1       111
|     11111111111           |     11111111111
|      1   111    11        |      1   111    11
|       111        11       |       111        11
|   111111111111111111      |   111111111111111111
|    1      11      1       |    1      11      1
|       1   11  1           |       1   11  1
|      1111 11   11         |      1111 11   11
|     111   11    111       |     111   11    111
|   111     11     111      |   111     11     111
|  11    11111      1       |  11    11111      1
| 1        11               | 1        11
|          1                |          1
|------------------------   |------------------------
```

Figure 8.2   A Wrong Path

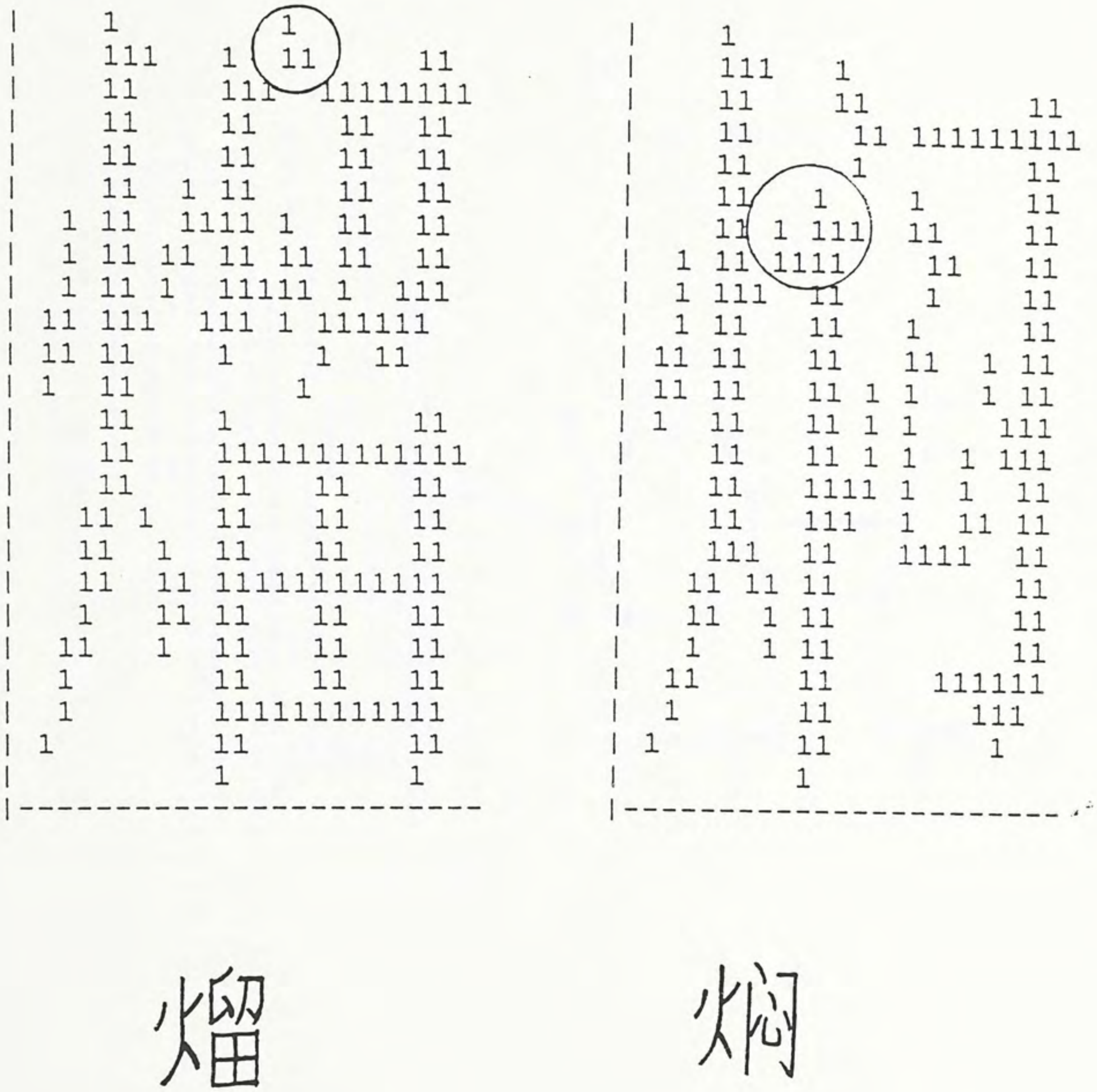
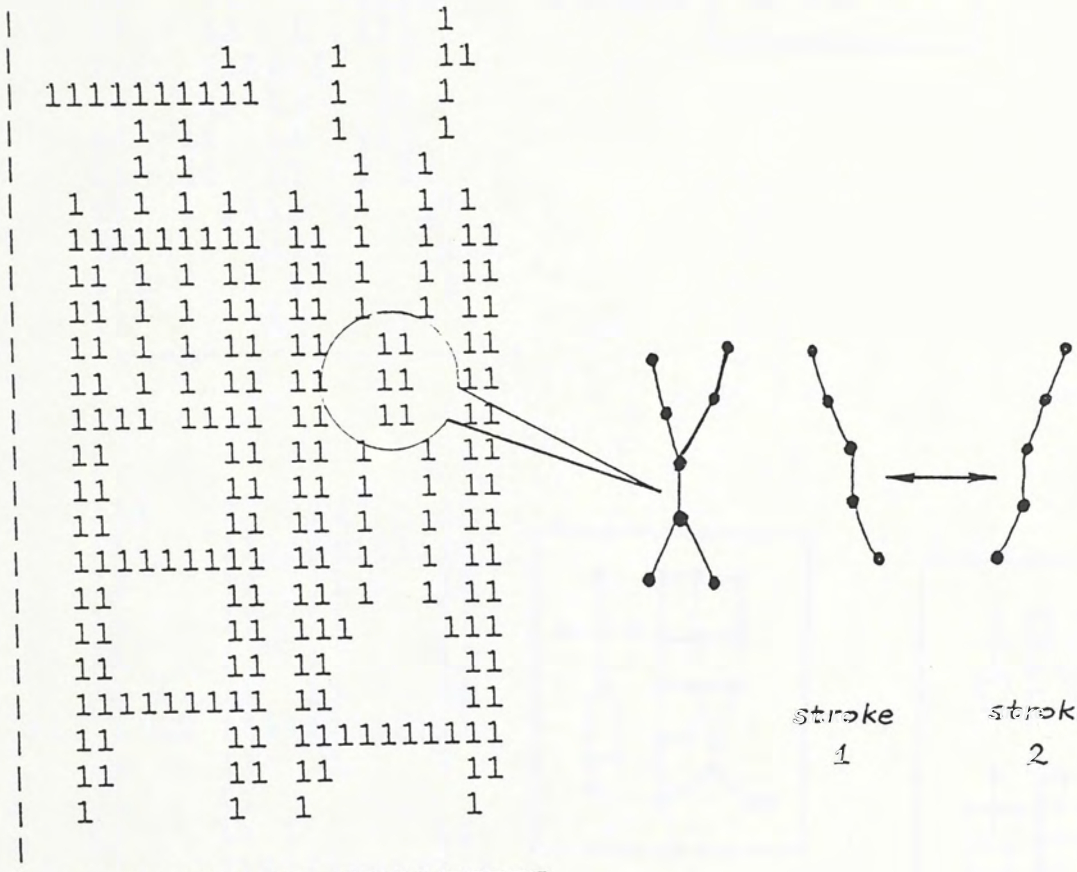


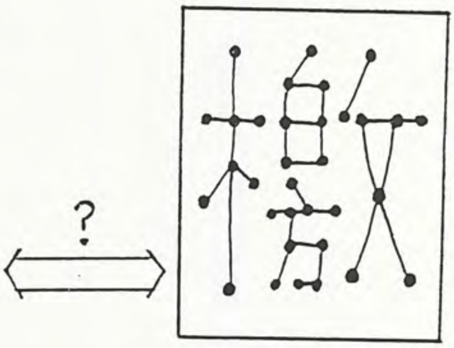
Figure 8.3 Situations That Would Lead to Error



```
|                   1
|         1    1     11
| 1111111111   1     1
|     1 1      1     1
|     1 1       1  1
|  1  1 1 1  1  1  1 1
|  111111111 11 1  1 11
|  11 1 1 11 11 1  1 11
|  11 1 1 11 11 1  1 11
|  11 1 1 11 11  11  11
|  11 1 1 11 11  11  11
|  1111 1111 11  11  11
|  11     11 11 1  1 11
|  11     11 11 1  1 11
|  11     11 11 1  1 11
|  111111111 11 1  1 11
|  11     11 11 1  1 11
|  11     11 111    111
|  11     11 11      11
|  111111111 11      11
|  11     11 1111111111
|  11     11 11      11
|  1      1  1       1
|
|------------------------
```

Figure 8.4 A Segment Shared By Two Strokes

```
|    1      1     1
|    111    11    111
|    11     1     11
|    11  1 1  11  11
|    11  11111111 11
|    11  11   11 11
|    111 11   1111    1
| 111111111111111 1111111
|    11  11   1111  11
|    11  11   1111  11
|   1111 1111111 1  11
|   111111 1     1  11
|   111 1   1  1 11 11
|  1 11 11111111111 1
|  1 11   11     1111
| 1  11   11  1  111
|    11   111111  11
|    11   11 11   11
|    11  11  11  111
|    11  11  11  1 11
|    11  1 1111 1   11
|    11 1   1111     111
|    1 1      1      1
|    1       1
|------------------------
```



```
|    1
|    111   1   11    11
|    11    1111111111111
|    11    11  11    11
|    11    11  11    11
|    11    11  11    11
|    11 1  11  11    11
| 11111111 111111111111
|    11    11   1    1
|    11    11
|    11    11 1      11
|  1 11 1  1111111111111
|  1111111 11   1    11
|  11  11  11   1   11
|  11  11  11 1  1  11
|  11  11  11111 1 11
|  11  11  11      111
|  11  11  11      11
|  11  11  11     111
|  111111  11     1 11
|  11  1   11   1   11
|  1       11 11      1111
|          111        1
|          1
|------------------------
```

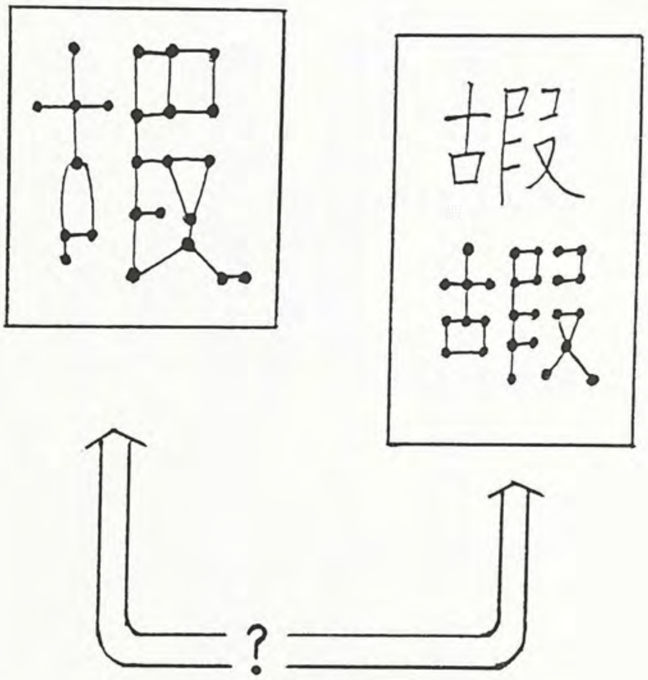


Figure 8.5 Crowded or Poorly Formed Patterns

```
|     1
|     111
|     11
|     11
|    11
|    11
|    11
|   111
|   1111
|  1111
|  1 11
| 1  11
| 1  11
|1   11
|    11
|    11
|    11
|    11
|    11
|    11
|    11
|    11
|    11
|    1
|---------------------
```

```
|     2
|     222
|     22
|     22
|    22
|    22
|    22
|   331
|   3311
|  3311
|  2 11
| 2  11
| 2  11
|3   11
|    11
|    11
|    11
|    11
|    11
|    11
|    11
|    11
|    11
|    1
|----------------
```

惟

```
|     1       1
|     11      11 11
|     111    11   11
|     11     11   11
|     11     1     1
|   1 1111  11        1
|   1 11 11 111111111111
|   1 11  1111   11
|  11 11   111   11
|  11 11  1111   11
| 11  11  1 11   11  1
|     11 1  11111111111
|     11    11   11
|     11    11   11
|     11    11   11
|     11    11   11  1
|     11    11111111111
|     11    11   11
|     11    11   11
|     11    11   11
|     11    11   11   1
|     11    111111111111
|     11    11
|     1     1
|---------------------
```

```
|     2       2
|     22      22 11
|     222    22   11
|     22     22   11
|     22     2     1
|   1 3311  22        1
|   1 22 11 311111111111
|   1 22  1331   11
|  11 22   311   11
|  11 22  3311   11
| 11  22  2 11   11  1
|     22 3  11111111111
|     22    11   11
|     22    11   11
|     22    11   11
|     22    11   11  1
|     22    11111111111
|     22    11   11
|     22    11   11
|     22    11   11
|     22    11   11   1
|     22    111111111111
|     22    11
|     3     1
|---------------------
```

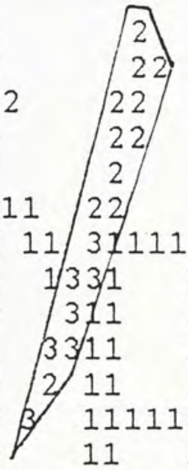


Figure 8.6 Resolved Situations

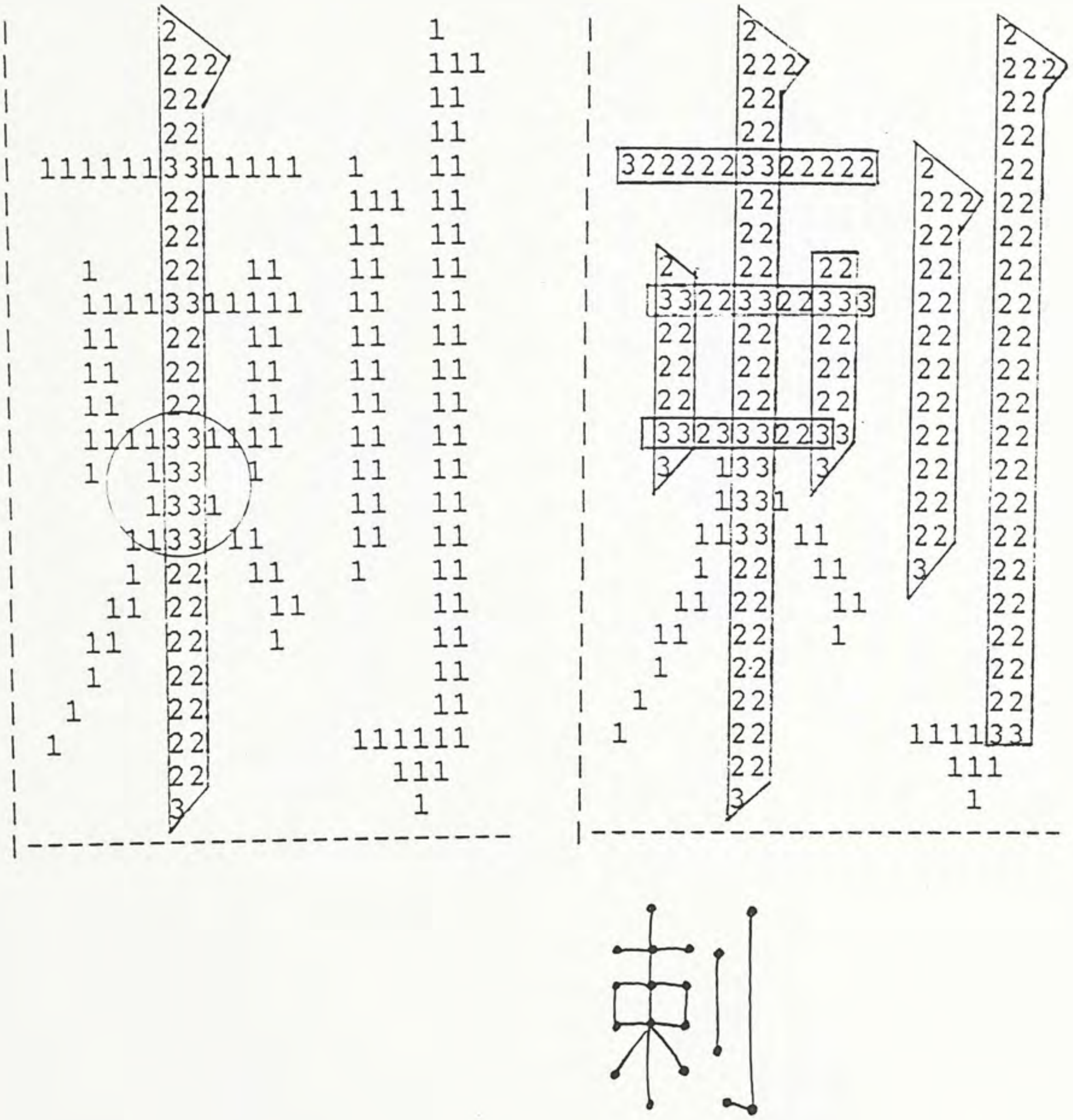


Figure 8.7 A Complex Node Resolved

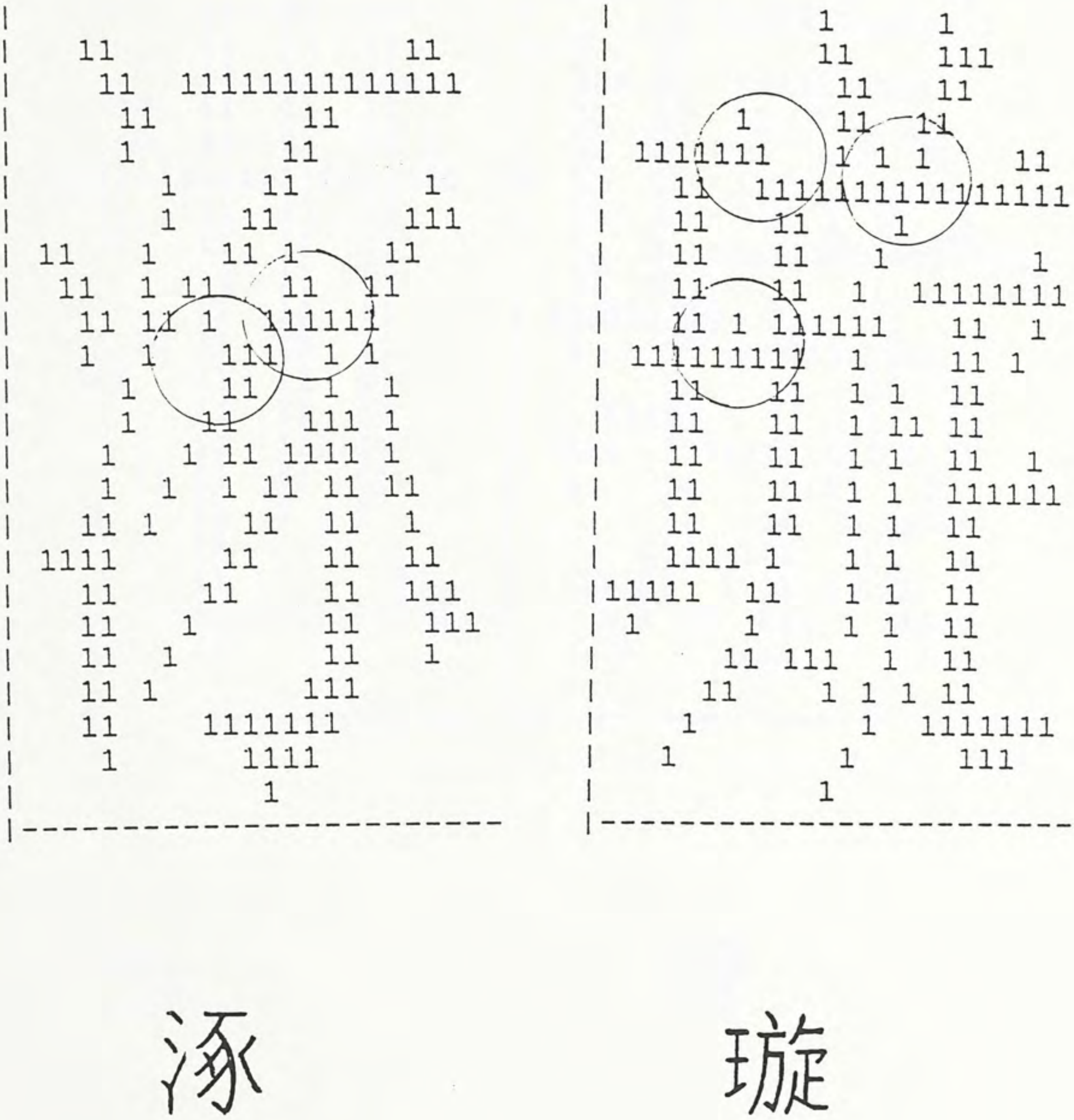


Figure 8.8 Situations That Lead to Error

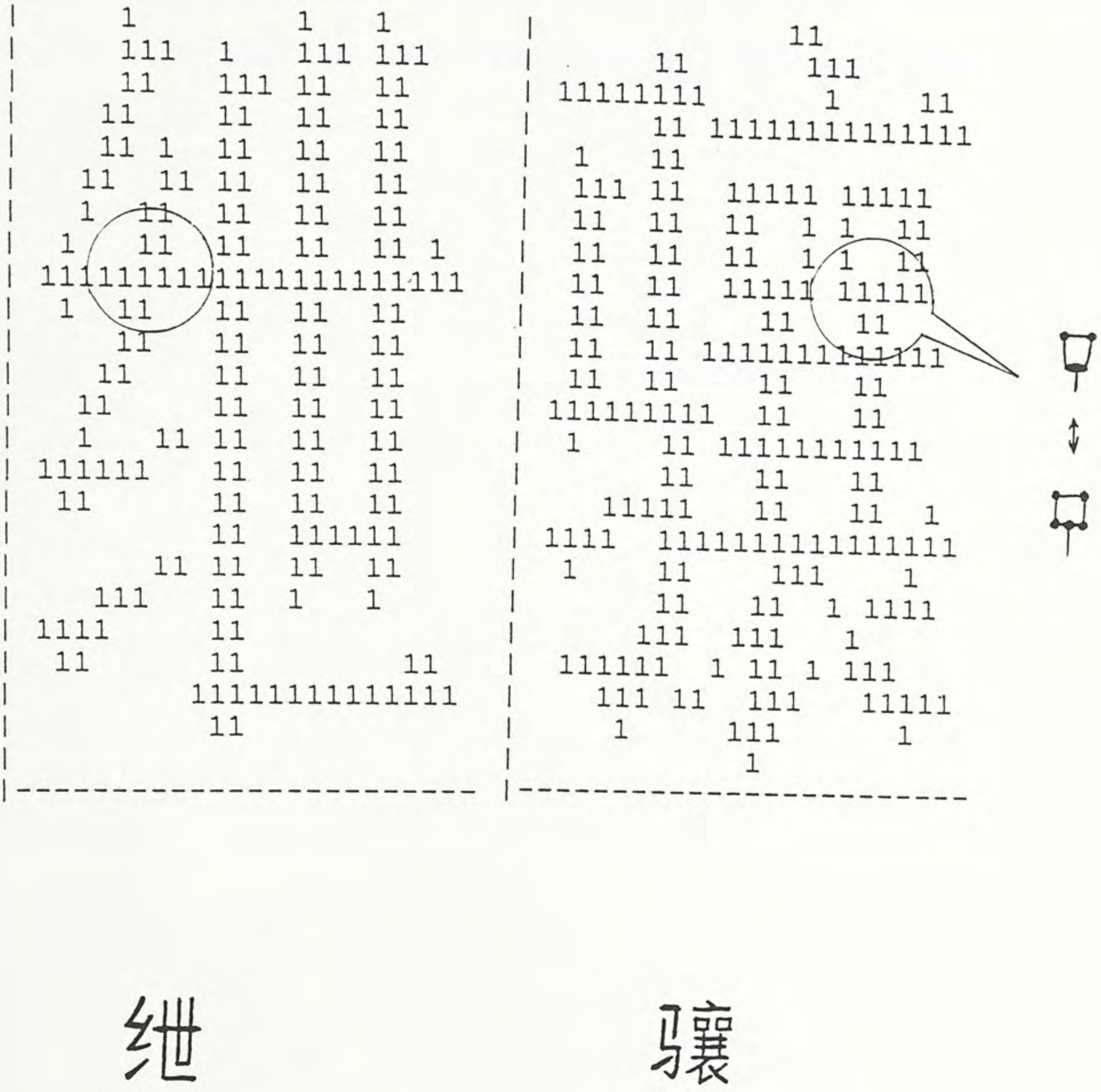


Figure 8.9 Ambiquity Cases

8.2 Stroke Extraction Level Two

The program for this level is sufficiently mechanical that no severe error was found. A total of about 300 sets of data from level one has been processed to generate data for further testing by Level Three. All the error reported were minor program bugs and have been subsequently corrected.

8.3 Stroke Extraction Level Three

Those sets of data, 300 characters, from Level Two have been tested on Level Three. About 70% were successful. The judgement of success were much easier for this level than for Level One because the strokes identified could be displayed to check for error. To find out the reason for failure is much more difficult, however.

Although the program can be run in a special mode where intermediate data are displayed for debugging purpose, the place where failure occurs typically is buried deep in recursion so that following the sequence of search is difficult. Experience has shown that many failures that have been subsequently corrected were due to situation not being inserted into the rules (Recall that these control rules were constructed by examining 200 characters). Of those failures not corrected and remained as part of the 30% that categorized as failed, many were buried so deep in the process that the author could not help but stop searching for the actual failure points for the moment. These need to be further analyzed, however. Other failure cases will be discussed in next chapter.

Since the set of control rules was based on only a small set of characters as reported in Chapter six, it must be cautious to extrapolate the 70% result to other yet processed characters. Perhaps it is fair to say that the result did show that the approach is feasible but improvement is much needed.

8.4 Scaling

Scaling algorithm used currently is very simple and the result in certain cases can be further improved especially when the original pattern is of a poor quality. In general, a simple scaling algorithm like this, with the aid of stroke information, can achieve better result than those with rather complicated algorithm with stroke information. Note that to results like this that require aesthetic judgement, there is no hard success or failure but can atmost be relatively good or relatively poor in quality.

The stroke information in the current use is simply to separate different portions of a pattern so that unwanted interpolation at the junction between two or more strokes can be eliminated. Actually, the stroke information could be further utilized to provide further error control. This, if implemented, could further demonstrate the importance of stroke information. More on this will be discussed in next chapter.

8.5 Font Transformation

Patterns of two different font types have been produced and the results are generally good. There are, however, still areas where the program must be further improved. These are not the defect of the algorithm, as the method is rather mechanical, but that some cases not yet handled. One is when the component window deviated too much to the size of the pattern, distortion results as shown in Figure 8.10. The other is when two connected strokes are reproduced, the pattern produced may not show good connection as desired. Apart from these, most of the degradation of quality comes from the scaling part which has been allowed so that resolution smaller than 128x128 may be specified.

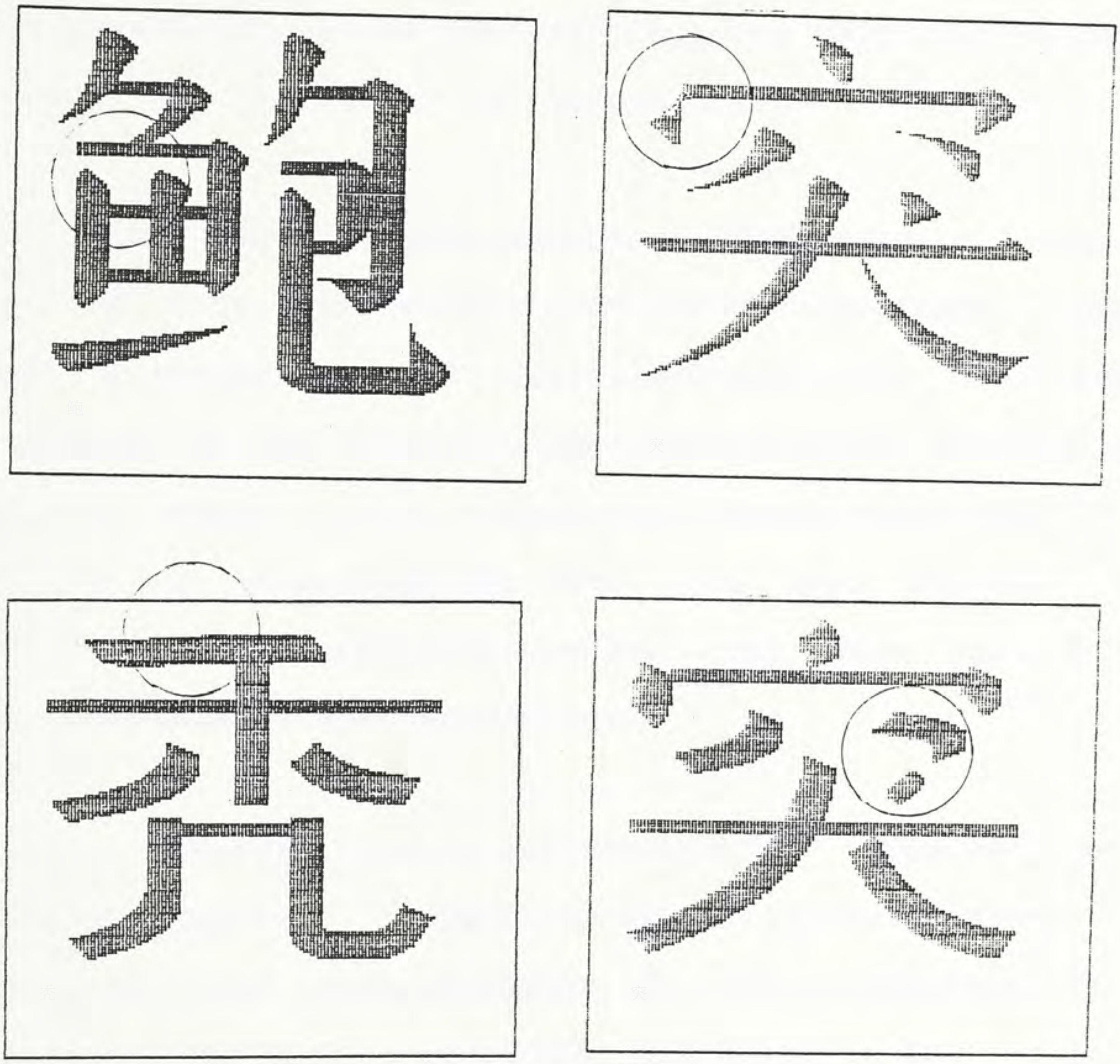


Figure 8.10 Results to be Improved

9.0 DISCUSSION

9.1 On Stroke Extraction

In the following discussion, we attempt to answer the naturally arised question that how well the method worked, and how it may be improved.

The current implementation gave only a modest success rate, as evident from the data available that only slightly over 52% overall success rate has been achieved for the first 300 characters tested. Obviously, this is remote to what a practical system would require. As can be seen from the data, the main sources of rejection came from Level One and Level Three. This will be individually discussed below.

A careful examination of data has revealed that several implicit assumptions about Chinese character patterns have been made when the trace algorithm was designed. As these assumptions were found to be invalid, errors occurred as a result.

The first assumption is that each stroke segment should observe a general and gradual change in direction and width. This assumption, however, due to low resolution, and sometimes to a character not being well conditioned, is not always true. The second assumption is that with the prediction process implemented, many complicated nodes could be resolved.

This, as the data have shown, is not the case. One prominent example is when the prediction is made when still early in a trace sequence where only one or two units have been obtained, then the prediction is made on insufficient data so that the process fails. The third assumption is that the characters in a library would be well conditioned so that no noise removal be necessary. It is found that being low resolution itself is almost equivalent to having a certain amount of noise, though the situation may not be as worse as those directly from camera. The presence of this, as non-gradual change of direction and width and as unnecessary connections by which we mean strokes are connected not because of structural necessity but the result of limited space, has created many error cases. The fourth assumption is that occational forced connection could be recovered by Level Three as long as the trace produced reasonable stroke segments. It is found that once a character is broken down into a list of tokens, it is virtually impossible to distinguish these ambiquity cases. An after-thought suggests that those we who have been trained on Chinese characters resolve these ambiquity cases partly by our knowledge, partly by the vastly superior parallel recognition process where primitive and context information are used at the same time. To the serial method as our three-level system, these kinds of ambiquity situations prove to be impossible.

In our method, a sequence is collected unit by unit. In effect, the tracer is blind over the horizon, which makes it succeptable to stroke segments not having a gradual width and direction variation. Projection profile methods [43] reported may be borrowed here that the overall trend of a stroke segment can be reported at once with the tracing process adding to the detail. This might be able to reduce the difficulty of over-the-horizon problem that we are facing here. At the resolution of 24x24, information carried by each pixel is important, which makes a pre-processing smoothing stage impossible. If the patterns were of higher resolution, a pre-processing stage might be added to remove some notches that could mislead the program.

The case of complex nodes is very difficult. Thinning has been suggested by Stalling [38], and some equivalent result can be found in [16,17], but the result does not seem to be encouraging. This should not be surprising as such a locallized method without contextual (structural) guidance should not perform too well. Our method in some cases with sufficient past data of a sequence did manage to generate a reasonable path through some complex nodes. Although restricting a trace in only going east, west or downward was an attempt to take advantage of a general Chinese character characteristics and has simplified considerable the program design, it appears that this may be too severe a restriction when the situation indicates other direction may be more appropriate. For example, if a node is encountered early in a trace sequence, it is sometimes advantageous by going backward starting again from the other end of this stroke segment where there is no complex node so that the data then collected may allow a good investigation of the complex node. The exact method deserves further investigation. The idea is that starting from a complex node is to be avoided, and the best place to start a trace is from an isolated end of a stroke segment.

Our implementation of the analyzer handles only symbollic quantities. This precludes nearness measure frequently employed in character recognition systems. Our use of attributed grammar does allow variation of shapes be handled but it still requires the character be well traced which we have defined as in the last chapter as a stroke segment belonging to only one stroke, and that the connections among stroke segments be clearly identified allowing eventual recombinations to form strokes. This therefore places heavy constraint on Level One. A wrong sequence as shown in last chapter (Figure 8.2) will guarentee a rejection at Level Three, thus vastly increased the rejection rate at Level One. Note that a case as this may be acceptable in a character recognition system where the skeleton of a character is obtained to match against a library for a recognition, but is rejected here by the limitation of our analyzer, and by our requirement of obtaining strokes exactly from the pattern.

Noise handling is an important subject. Traditionally structural approach is particularly weak in handling patterns with noise. Fu [41,51] has some papers on combining syntactic and statistical approaches. In our case, statistical decision is difficult to apply to the trace process unless a radically different approach is taken. The problem of pre-processing noise removal is handicapped by the low resolution of our patterns.

Since all those works on strokes as mentioned in Chapter Two had different objectives and hence requirements from our work here, no direct comparison can be made. For example, the group led by Hsu [16,17] satisfy themselves by extracting a skeleton of a character and stop after a somewhat equivalent of our trace procedure with compatible results. If comparison is made to systems on character recognition, we can see that in some approaches the main concerns are to generate unique descriptor for each distinct character where detailed information needed for classifying strokes simply is of no use; or the main concern is to generate data for matching against a library so that the constraint is less severe as possible variations can be handled by a distance measure.

Without a nearness measure or some form of feedback, it is suspect that a serial method like this will remain low in recognition rate.

As to Level Three, current data do not indicate clearly where major failures occur. From the experience so far obtained, many of the failures could be attributed to cases not covered by the control rules. It is a blessing that Prolog has been chosen as the programming language for this level, as the power of which allows attributes and rules be easily added or modified, and a uniformly structured program be used to handle a large number of strokes with possible shape variation. Consider otherwise if higher order grammars had been used and a traditional type parser had been implemented, the rigidity and complexity of the resulting system might have made our approach doomed. On the other hand, the recursive program structure together with backtracking did make finding out where rejection occurred difficult after the program modules had been integrated. Other than this, an infrequent error has been identified as when two strokes of same type are touching each other, our analyzer either treats them as a single stroke, or announces that this combination as unacceptable according to our preset rule, which eventually lead to a rejection. At present, there is no easy way to teach the program to judge on this kind of ambiguities.

At present, we have not yet attempted to interface Prolog directly with other languages such as Fortran, Pascal, C or assembly, that are more suited to numerical computing. Doing so would open up a new horizon as to our overall approach to the problem. This is an area where immediate improvement can be made. A paper by You and Fu [51] has stressed the importance of using production rules to guide the primitive extraction process; that is, to imitate the human recognition process. Their method has combined parsing and primitive extraction into a single process, in contrast to our current implementation that the primitive extraction and the analysis are two distinct processes. Perhaps their conclusion can be extended to that the higher level contextual information might be very useful to guide stroke segment extraction. This is an area that deserves additional examination. Please be noted that this is mainly an observation as the limited amount of our data does not allow us to extrapolate the result too far. For a discussion of the importance of context in pattern recognition, please see Toussaint [40].

The choice of parameters to describe each stroke segment is another area to be investigated. It must be admitted that most of the descriptive quantities have been chosen rather arbitrarily without theoretical or statistical support, ie., by a somewhat ad-hoc approach. This brings us back to the problem of modelling as discussed at the survey given earlier in the report. We have restricted our target to strokes so as to avoid the difficult problem of giving an adequate model of Chinese characters. Nonetheless, the problem appears when it comes to classification. We have here chosen a set of intuitively appealing parameters; we have also defined connection among strokes so as to prevent a single stroke consists of several tokens be mistaken as several strokes each of which consists of one token, but if the method is to be improved; to improve so that the parameters are not just adequate but can truely allow a practical system be based on, probably a re-investigation in the area of modelling is needed.

We have previously mildly stressed the importance of knowledge. From the list of practical A.I. works surveyed by Rich [33] that none of which can be covered by a few lines of elegant theory. All but a few require a large, carefully compiled list of facts or rules. Perhaps this is what prompted Rich to make the remark that the fast and hard fact from the first twenty years of A.I. research is that intelligence needs knowledge. Processing problems on Chinese characters appear to fit into the area where similar remarks can be made. For example, most of those character recognition systems have large tables where reference data are held. These allow them to handle variations and make good judgements even at the presence of large amount of distortions. By comparison, our method is particularly weak in this respect, as a result of insufficient investigation on modelling and insufficient provision on handling of variations and distortions. Still, this is a difficult field, as thousands of years of development still do not yield a satisfactory lexicographical ordering may indicate the level of difficulties caused by the vast number of oftern complexly structured characters. It seems that, an occational reported noval methods, including the trial implementation as this one, or those reported very high recognition rate at controlled conditions, are remote in providing a truely all-round practical system, which may only come about after a careful and successful modelling, and mechanization; and which may eventually demand a very large database to represent facts and rules; and may eventually demand massive parallelism, a mechanism that we do not yet fully understand.

9.2 On Scaling

In general, the program worked as directed and verified the simple but difficult to demonstrate concept of stroke based scaling. The quality of the result could be further improved with additional extention on the current system.

There are still rooms to improve the stepwise scaling method, though the time available does not allow further investigation. One of the obvious way to improve the method is by generating a Freeman chain of the boundary so that irregularity can be detected by a finite state machine and corrected. This method has been used by Casey. The method is not entirely applicable if the operation is on a character as a whole as some pixels that are singular but must be preserved may be killed. But we can take advantage of the fact that the operation in our case is on individual stroke and hence the situation is sufficiently simple that irregularities can be removed by this process.

A global parameter may be used to correct stroke width problem initiated from the base pattern in which strokes of the same type having unequal width. In the low resolution case where the width of a stroke is normally one or two pixels, the resulting distortion when enlarged can be severe. As the stroke information is available, a simple counting will know what space will be allowed for maneuvouring these strokes and automatic correction procedure may be devised. Similar arguments supports another parameter to provide good spacing between strokes so that they will not be crowded at some region. The pattern, though may not be always true, may have a better chance of having a better appearance.

9.3 On Font Transformation

The requirement of fixed relative stroke position is obviously an inevitable constraint in our font transformation program. To find a way to improve on this is very difficult as there is yet general guidelines to judge the asthetic arguments involved. All those reported systems rely on human assistance to adjust the controlling parameters.

Requiring that the system to handle also the fonts that require a structural change may be overly ambitious. It is suspected that in these cases, building a system with such a level of intelligence may not be worthwhile. A separate set of font as base may be more desirable and cost effective.

10.0 CONCLUSION

A three-level stroke extraction system has been implemented. The information produced has been applied to scaling and font transformation of Chinese character patterns[1].

The data obtained indicate that much improvement is needed to bring the stroke extraction system practical. Experience has shown that the lack of an in-depth study on the modeling of Chinese characters has hindered their mechanization. Modelling of Chinese character is therefore an area that deserves most of the future effort.

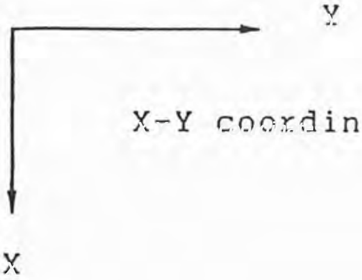
Results of scaling and font transformation are generally good, but can be further improved. This has demonstrated that stroke information can simplify considerably a number of processing problems associated with Chinese characters.

---

[1] A summary of this work has been accepted for presentation at The IEEE Asian Electronics Conference 1987 under the title "Stroke Extraction, Scaling and Font Transformation of Chinese character patterns".

APPENDIX A : A TRACE EXAMPLE OF LEVEL ONE

```
000000000000000000000000
001000110000000000011000
001111111011111111111100
001100110000000000000000
001100110000100000110000
001100110000111111111000
001100110000110000110000
001111110000110000110000
001100110000111111110000
001100110000110000110000
001100110010000000001100
001100110011111111111110
001100110011010000101100
001111110011011000111100
001100110011001101101100
001100110011001001001100
001100110011111111111100
001100110011000110001100
001100110011000110001100
001000110011000110001100
011011110011000110001100
010000110011000111111100
100000100011000110011000
000000000010000000010000
```



X-Y coordinates

Figure A.1.1 Original Pattern

```
........................
..o...11...........11...
..*******.111111111111..
..11..11................
..11..11....1.....11....
..11..11....111111111...
..11..11....11....11....
..111111....11....11....
..11..11....11111111....
..11..11....11....11....
..11..11..1.........11..
..11..11..1111111111111.
..11..11..11.1....1.11..
..111111..11.11...1111..
..11..11..11..11.11.11..
..11..11..11..1..1..11..
..11..11..111111111111..
..11..11..11...11...11..
..11..11..11...11...11..
..1...11..11...11...11..
.11.1111..11...11...11..
.1....11..11...1111111..
1.....1...11...11..11...
..........1........1....
```

```
Scanner --> x = 2, y = 3
          (pixel marked as o)
Starter --> direction = south
            width(1) = 1
Tracer  --> new width  = 7
            new offset = 0
            (left offset)
         (pixels marked as *)
```

Figure A.1.2

```
........................
..2..11............11...  Tracer -->
..3311111.111111111111..  Single_handler --> (width = 7)
..22..11................  Predictor -->
..**..11....1.....11....  LookAhead --> returns
..11..11....111111111...                width(2)  = 2
..11..11....11....11....                offset(2) = 0
..111111....11....11....                width(3)  = 2
..11..11....11111111....                offset(3) = 0
..11..11....11....11....  Back to Tracer
..11..11..1.........11..  Tracer --> new width = 2
..11..11..1111111111111.             new offset = 0
..11..11..11.1....1.11..             (as marker as *)
..111111..11.11...1111..
..11..11..11..11.11.11..
..11..11..11..1..1..11..
..11..11..111111111111..
..11..11..11...11...11..
..11..11..11...11...11..
..1...11..11...11...11..
.11.1111..11...11...11..
.1....11..11...1111111..
1.....1...11...11..11...
..........1........1....
```

Figure A.1.3

```
........................
..2...11...........11...
..3311111.111111111111..  Tracer -->
..22..11................  Single_handler --> accepts
..22..11....1.....11....                 width(4)  = 2
..**..11....111111111...                 offset(4) = 0
..11..11....11....11....  Sequence is declared straight
..111111....11....11....
..11..11....11111111....  Back to Tracer
..11..11....11....11....  Tracer --> new width = 2
..11..11..1.........11..             new offset = 0
..11..11..1111111111111.             (marked as **)
..11..11..11.1....1.11..
..111111..11.11...1111..
..11..11..11..11.11.11..
..11..11..11..1..1..11..
..11..11..111111111111..
..11..11..11...11...11..
..11..11..11...11...11..
..1...11..11...11...11..
.11.1111..11...11...11..
.1....11..11...1111111..
1.....1...11...11..11...
..........1........1....
```

Figure A.1.4

```
........................
..2...11...........11...
..3311111.111111111111..   Tracer -->
..22..11................   Single_handler --> accepts
..22..11....1.....11....                      width(5) = 2
..22..11....111111111...                     offset(5) = 0
..22..11....11....11....
..******....11....11....   Back to Tracer
..11..11....11111111....   so cycle repeats until
..11..11....11....11....                     new width = 6
..11..11..1.........11..                    new offset = 0
..11..11..1111111111111.                     (marked as *)
..11..11..11.1....1.11..
..111111..11.11...1111..
..11..11..11..11.11.11..
..11..11..11..1..1..11..
..11..11..111111111111..
..11..11..11...11...11..
..11..11..11...11...11..
..1...11..11...11...11..
.11.1111..11...11...11..
.1....11..11...1111111..
1.....1...11...11..11...
..........1........1....
```

Figure A.1.5

```
........................
..2...11...........11...
..3311111.111111111111..   Tracer -->
..22..11................   Single_handler -->
..22..11....1.....11....   Exception_handler -->
..22..11....111111111...   Predictor -->
..22..11....11....11....   LookAhead --> returns
..331111....11....11....                 width(7)  = 2
..22..11....11111111....                 offset(7) = 0
..**..11....11....11....                 width(8)  = 2
..11..11..1.........11..                 offset(8) = 0
..11..11..1111111111111.   Back to Tracer
..11..11..11.1....1.11..   Tracer --> new width = 2
..111111..11.11...1111..              new offset = 0
..11..11..11..11.11.11..              (marked as *)
..11..11..11..1..1..11..
..11..11..111111111111..
..11..11..11...11...11..
..11..11..11...11...11..
..1...11..11...11...11..
.11.1111..11...11...11..
.1....11..11...1111111..
1.....1...11...11..11...
..........1........1....
```

Figure A.1.6

```
........................
..2...11...........11...
..3311111.111111111111..
..22..11................
..22..11....1.....11....
..22..11....111111111...
..22..11....11....11....
..331111....11....11....
..22..11....11111111....
..22..11....11....11....
..22..11..1.........11..
..22..11..1111111111111.
..22..11..11.1....1.11..
..331111..11.11...1111..
..22..11..11..11.11.11..
..22..11..11..1..1..11..
..22..11..111111111111..
..22..11..11...11...11..
..22..11..11...11...11..
..*...11..11...11...11..
.11.1111..11...11...11..
.1....11..11...1111111..
1.....1...11...11..11...
..........1........1....
```

```
so cycle repeats until
          new width  = 1
          new offset = 0
Tracer -->
Single_handler -->
        width = 1 < previous
        terminates here
        must be a node.

End of first sequence.
```

Figure A.1.7

```
........................
..2...o1...........11...
..33*****.111111111111..
..22..11................
..22..11....1.....11....
..22..11....111111111...
..22..11....11....11....
..331111....11....11....
..22..11....11111111....
..22..11....11....11....
..22..11..1.........11..
..22..11..1111111111111.
..22..11..11.1....1.11..
..331111..11.11...1111..
..22..11..11..11.11.11..
..22..11..11..1..1..11..
..22..11..111111111111..
..22..11..11...11...11..
..22..11..11...11...11..
..3...11..11...11...11..
.11.1111..11...11...11..
.1....11..11...1111111..
1.....1...11...11..11...
..........1........1....
```

```
Scanner --> x = 2, y = 7
Starter --> direction = south
            width(1) = 2
Tracer --> new width = 5
           new offset = -2
           (marked as *)
```

Figure A.1.8

```
........................
..2...22...........11...
..3311331.111111111111..
..22..22................
..22..11....1.....11....
..22..11....111111111...
..22..11....11....11....
..331111....11....11....
..22..11....11111111....
..22..11....11....11....
..22..11..1.........11..
..22..11..1111111111111.
..22..11..11.1....1.11..
..331111..11.11...1111..
..22..11..11..11.11.11..
..22..11..11..1..1..11..
..22..11..111111111111..
..22..11..11...11...11..
..22..11..11...11...11..
..3...11..11...11...11..
.11.1111..11...11...11..
.1....11..11...1111111..
1.....1...11...11..11...
..........1........1....
```

```
Tracer -->
Single_handler -->
Exception_handler -->
Predictor -->
LookAhead --> returns
              width(2)  = 2
              offset(2) = 0
              width(3)  = 2
              offset(3) = 0
```

Figure A.1.9

```
........................
..2...22...........o1...
..3311331.************..
..22..22................
..22..22....1.....11....
..22..22....111111111...
..22..22....11....11....
..331133....11....11....
..22..22....11111111....
..22..22....11....11....
..22..22..1.........11..
..22..22..1111111111111.
..22..22..11.1....1.11..
..331133..11.11...1111..
..22..22..11..11.11.11..
..22..22..11..1..1..11..
..22..22..111111111111..
..22..22..11...11...11..
..22..22..11...11...11..
..3...22..11...11...11..
.11.1133..11...11...11..
.1....22..11...1111111..
1.....3...11...11..11...
..........1........1....
```

So cycle repeats till second sequence done.

```
Scanner --> x = 2, y = 20
Starter --> direction = south
            width(1) = 2
Tracer -->  new width = 12
            new offset = -9
            (marked as *)
```

Figure A.1.10

```
........................
..2...22...........22...
..3311331.111111111331..
..22..22................
..22..22....1.....11....
..22..22....111111111...
..22..22....11....11....
..331133....11....11....
..22..22....11111111....
..22..22....11....11....
..22..22..1.........11..
..22..22..1111111111111.
..22..22..11.1....1.11..
..331133..11.11...1111..
..22..22..11..11.11.11..
..22..22..11..1..1..11..
..22..22..111111111111..
..22..22..11...11...11..
..22..22..11...11...11..
..3...22..11...11...11..
.11.1133..11...11...11..
.1....22..11...1111111..
1.....3...11...11..11...
..........1........1....
```

```
Tracer -->
Single_handler -->
Exception_handler -->
Predictor -->
LookAhead --> returns
                width(2) = 2
                offset(2) = 0
          no more units ahead
          therefore end of
          third sequence.
```

Figure A.1.11

```
........................
..2...22...........22...
..o311331.111111111331..
..22..22................
..22..22....1.....11....
..22..22....111111111...
..22..22....11....11....
..331133....11....11....
..22..22....11111111....
..22..22....11....11....
..22..22..1.........11..
..22..22..1111111111111.
..22..22..11.1....1.11..
..331133..11.11...1111..
..22..22..11..11.11.11..
..22..22..11..1..1..11..
..22..22..111111111111..
..22..22..11...11...11..
..22..22..11...11...11..
..3...22..11...11...11..
.11.1133..11...11...11..
.1....22..11...1111111..
1.....3...11...11..11...
..........1........1....
```

```
Scanner --> x = 3, y = 3
Starter --> east
              width(1) = 1
```

Figure A.1.12

```
........................
..2...22...........22...
..o*11331.111111111331..
..2*..22................
..2*..22....1.....11....
..2*..22....111111111...
..2*..22....11....11....
..331133....11....11....
..22..22....11111111....
..22..22....11....11....
..22..22..1.........11..
..22..22..1111111111111.
..22..22..11.1....1.11..
..331133..11.11...1111..
..22..22..11..11.11.11..
..22..22..11..1..1..11..
..22..22..111111111111..
..22..22..11...11...11..
..22..22..11...11...11..
..3...22..11...11...11..
.11.1133..11...11...11..
.1....22..11...1111111..
1.....3...11...11..11...
..........1........1....
```

```
Tracer --> new width = 5
                new offset = -4
           complex type
           consists of two
           types of pixels:
           special and processed.
           (marked as *)
```

Figure A.1.13

```
........................
..2...22...........22...
..332*331.111111111331..
..22..22................
..22..22....1.....11....
..22..22....111111111...
..22..22....11....11....
..331133....11....11....
..22..22....11111111....
..22..22....11....11....
..22..22..1.........11..
..22..22..1111111111111.
..22..22..11.1....1.11..
..331133..11.11...1111..
..22..22..11..11.11.11..
..22..22..11..1..1..11..
..22..22..111111111111..
..22..22..11...11...11..
..22..22..11...11...11..
..3...22..11...11...11..
.11.1133..11...11...11..
.1....22..11...1111111..
1.....3...11...11..11...
..........1........1....
```

```
Tracer -->
Single_handler -->
Special_handler -->
Predictor -->
LookAhead --> returns
              width(2)  = 1
              offset(2) = 0
              width(3)  = 1
              offset(3) = 0

Returns to Tracer -->
              new width = 1
              new offset = 0
              (marked as *)
Eventually accepted.
```

Figure A.1.14

```
........................
..2...*2...........22...
..3322*31.111111111331..
..22..*2................
..22..*2....1.....11....
..22..*2....111111111...
..22..*2....11....11....
..331133....11....11....
..22..22....11111111....
..22..22....11....11....
..22..22..1.........11..
..22..22..1111111111111.
..22..22..11.1....1.11..
..331133..11.11...1111..
..22..22..11..11.11.11..
..22..22..11..1..1..11..
..22..22..111111111111..
..22..22..11...11...11..
..22..22..11...11...11..
..3...22..11...11...11..
.11.1133..11...11...11..
.1....22..11...1111111..
1.....3...11...11..11...
..........1........1....
```

```
Tracer --> new width = 6
           new offset = -4
        complex type consists
        of three groups of
        pixels.

Single_handler -->
Special_handler -->
Predictor -->
Lookahead --> temporarily
              width(5) = 1
              offset(5) = 0
        (marked as @ at next
         figure)
```

Figure A.1.15

```
........................
..2...2*...........22...
..3322@*1.111111111331..
..22..2*................
..22..2*....1.....11....
..22..2*....111111111...
..22..2*....11....11....
..331133....11....11....
..22..22....11111111....
..22..22....11....11....
..22..22..1.........11..
..22..22..1111111111111.
..22..22..11.1....1.11..
..331133..11.11...1111..
..22..22..11..11.11.11..
..22..22..11..1..1..11..
..22..22..111111111111..
..22..22..11...11...11..
..22..22..11...11...11..
..3...22..11...11...11..
.11.1133..11...11...11..
.1....22..11...1111111..
1.....3...11...11..11...
..........1........1....
```

```
LookAhead --> new width = 6
              new offset = -4
              (marked as *)
Single_handler -->
Special_handler -->
Predictor -->
LookAhead --> temporarily
              width(6) = 1
              offset(6) = 0
```

Figure A.1.16

```
........................
..2...22...........22...
..3322@@*.111111111331..
..22..22................
..22..22....1.....11....
..22..22....111111111...
..22..22....11....11....
..331133....11....11....
..22..22....11111111....
..22..22....11....11....
..22..22..1.........11..
..22..22..1111111111111.
..22..22..11.1....1.11..
..331133..11.11...1111..
..22..22..11..11.11.11..
..22..22..11..1..1..11..
..22..22..111111111111..
..22..22..11...11...11..
..22..22..11...11...11..
..3...22..11...11...11..
.11.1133..11...11...11..
.1....22..11...1111111..
1.....3...11...11..11...
..........1........1....
```

LookAhead --> new width = 1
new offset = 0
(marked as *)
accepted.

returns a whole series
width(5) offset(5)
width(6) offset(6)
width(7) offset(7)

accepted.

Figure A.1.17

```
........................
..2...22...........22...
..3322333.o*1111111331..
..22..22................
..22..22....1.....11....
..22..22....111111111...
..22..22....11....11....
..331133....11....11....
..22..22....11111111....
..22..22....11....11....
..22..22..1.........11..
..22..22..1111111111111.
..22..22..11.1....1.11..
..331133..11.11...1111..
..22..22..11..11.11.11..
..22..22..11..1..1..11..
..22..22..111111111111..
..22..22..11...11...11..
..22..22..11...11...11..
..3...22..11...11...11..
.11.1133..11...11...11..
.1....22..11...1111111..
1.....3...11...11..11...
..........1........1....
```

Eventually back to Tracer

Tracer --> no more units ahead
end of fourth sequence.
Last unit is automatically
automatically declared
as Special as it must
be a node.

Scanner --> x = 3, y = 11
Starter --> direction = east
width(1) = 1
Tracer --> new width = 1
new offset = 0
(marked as *)

Figure A.1.18

```
.......................
..2...22..........22...
..3322333.222222222333..
..22..22...............
..22..22...1.....11....
..22..22...111111111...
..22..22...11....11....
..331133...11....11....
..22..22...11111111....
..22..22...11....11....
..22..22..1........11..
..22..22..1111111111111.
..22..22..11.1....1.11..
..331133..11.11...1111..
..22..22..11..11.11.11..
..22..22..11..1..1..11..
..22..22..111111111111..
..22..22..11...11...11..
..22..22..11...11...11..
..3...22..11...11...11..
.11.1133..11...11...11..
.1....22..11...1111111..
1.....3...11...11..11...
..........1.......1....
```

So eventually sequence five is done.

Figure A.1.19

```
.......................
..2...22..........22...
..3322332.222222222333..
..22..22...............
..22..22...2.....22....
..22..22...332222333...
..22..22...22....22....
..o31133...22....22....
..22..22...33111133....
..22..22...33....33....
..22..22..1........11..
..22..22..1111111111111.
..22..22..11.1....1.11..
..331133..11.11...1111..
..22..22..11..11.11.11..
..22..22..11..1..1..11..
..22..22..111111111111..
..22..22..11...11...11..
..22..22..11...11...11..
..3...22..11...11...11..
.11.1133..11...11...11..
.1....22..11...1111111..
1.....3...11...11..11...
..........1.......1....
```

Similarly more sequences are done.
Scanner --> x = 8, y = 3
Starter --> direction = east
            width(1) = 1

Figure A.1.20

```
........................
..2...22...........22...
..3322333.222222222333..
..22..*2................
..22..*2....2.....22....
..22..*2....332222333...
..22..*2....22....22....
..3322*3....22....22....
..22..*2....33111133....
..22..*2....33....33....
..22..*2..1.........11..
..22..*2..1111111111111.
..22..*2..11.1....1.11..
..331133..11.11...1111..
..22..22..11..11.11.11..
..22..22..11..1..1..11..
..22..22..111111111111..
..22..22..11...11...11..
..22..22..11...11...11..
..3...22..11...11...11..
.11.1133..11...11...11..
.1....22..11...1111111..
1.....3...11...11..11...
..........1........1....
```

Eventually Tracer encounters a larger unit.

```
Tracer --> new width = 10
           new offset = -5
           (marked as *)
Single_handler -->
Special_handler -->
Predictor -->
LookAhead --> temporarily
              width(5) = 1
              offset(5) = 0,
              (marked as @ at
              next figure)
```

Figure A.1.21

```
........................
..2...22...........22...
..3322333.222222222333..
..22..2*................
..22..2*....2.....22....
..22..2*....332222333...
..22..2*....22....22....
..3322@*....22....22....
..22..2*....33111133....
..22..2*....33....33....
..22..2*..1.........11..
..22..2*..1111111111111.
..22..2*..11.1....1.11..
..331133..11.11...1111..
..22..22..11..11.11.11..
..22..22..11..1..1..11..
..22..22..111111111111..
..22..22..11...11...11..
..22..22..11...11...11..
..3...22..11...11...11..
.11.1133..11...11...11..
.1....22..11...1111111..
1.....3...11...11..11...
..........1........1....
```

```
LookAhead --> new width = 10
              new offset = -5
              (marked as *)
Single_handler -->
Special_handler -->
Predictor -->
LookAhead --> temporarily
              width(6) = 1
              offset(6) = 0,
              see next figure.
```

Figure A.1.22

```
........................
..2...22...........22...
..3322333.222222222333..
..22..22................
..22..22....2.....22....
..22..22....332222333...
..22..22....22....22....
..3322@@....22....22....
..22..22....33111133....
..22..22....33....33....
..22..22..1.........11..
..22..22..1111111111111.
..22..22..11.1....1.11..
..331133..11.11...1111..
..22..22..11..11.11.11..
..22..22..11..1..1..11..
..22..22..111111111111..
..22..22..11...11...11..
..22..22..11...11...11..
..3...22..11...11...11..
.11.1133..11...11...11..
.1....22..11...1111111..
1.....3...11...11..11...
..........1........1....
```

LookAhead --> no more units ahead returns whole sequence

width(5), offset(5)
width(6), offset(6)

(marked as @@)

Eventually back to Tracer.

Figure A.1.23

```
........................
..2...22...........22...
..3322333.222222222333..
..22..22................
..22..22....2.....22....
..22..22....332222333...
..22..22....22....22....
..332233....22....22....
..22..22....33111133....
..22..22....33....33....
..22..22..1.........11..
..22..22..1111111111111.
..22..22..11.1....1.11..
..331133..11.11...1111..
..22..22..11..11.11.11..
..22..22..11..1..1..11..
..22..22..111111111111..
..22..22..11...11...11..
..22..22..11...11...11..
..3...22..11...11...11..
.11.1133..11...11...11..
.1....22..11...1111111..
1.....3...11...11..11...
..........1........1....
```

Tracer --> ends sequence nine.

Figure A.1.24

```
........................
..2...22...........22...
..3322333.222222222333..
..22..22................
..22..22....2.....22....
..22..22....332222333...
..22..22....22....22....
..332233....22....22....
..22..22....33222233....
..22..22....33....33....
..22..22..2.........22..
..22..22..3323222232333.
..22..22..22.1....1.22..
..331133..22.11...1133..
..22..22..22..11.11.22..
..22..22..22..1..1..22..
..22..22..331111111133..
..22..22..22...11...22..
..22..22..22...11...22..
..3...22..22...11...22..
.11.1133..22...11...22..
.1....22..22...1111133..
1.....3...22...11..11...
..........3........1....
```

Similarly, four more
sequences are done.

Figure A.1.25

```
........................
..2...22...........22...
..3322333.222222222333..
..22..22................
..22..22....2.....22....
..22..22....332222333...
..22..22....22....22....
..332233....22....22....
..22..22....33222233....
..22..22....33....33....
..22..22..2.........22..
..22..22..332o222232333.
..22..22..22.*....1.22..
..331133..22.11...1133..
..22..22..22..11.11.22..
..22..22..22..1..1..22..
..22..22..331111111133..
..22..22..22...11...22..
..22..22..22...11...22..
..3...22..22...11...22..
.11.1133..22...11...22..
.1....22..22...1111133..
1.....3...22...11..11...
..........3........1....
```

```
Scanner --> x = 12, y = 14
Starter --> direction = south
            width(1) = 1

Tracer --> new width = 1
           new offset = 0
           (marked as *)
```

Figure A.1.26

```
........................
..2...22...........22...
..3322333.222222222333..
..22..22................
..22..22....2.....22....
..22..22....332222333...
..22..22....22....22....
..332233....22....22....
..22..22....33222233....
..22..22....33....33....
..22..22..2.........22..
..22..22..3323222232333.
..22..22..22.2....1.22..
..331133..22.**...1133..
..22..22..22..11.11.22..
..22..22..22..1..1..22..
..22..22..331111111133..
..22..22..22...11...22..
..22..22..22...11...22..
..3...22..22...11...22..
.11.1133..22...11...22..
.1....22..22...1111133..
1.....3...22...11..11...
..........3........1....
```

```
Tracer -->
Single_handler --> accepts
                   width(2) = 1
                   offset(2) = 0

Back to Tracer.

Tracer --> new width = 2
           new offset = 0
           (marked as *)
```

Figure A.1.27

```
........................
..2...22...........22...
..3322333.222222222333..
..22..22................
..22..22....2.....22....
..22..22....332222333...
..22..22....22....22....
..332233....22....22....
..22..22....33222233....
..22..22....33....33....
..22..22..2.........22..
..22..22..33232222o2333.
..22..22..22.2....*.22..
..331133..22.22...1133..
..22..22..22..22.11.22..
..22..22..22..3..1..22..
..22..22..331111111133..
..22..22..22...11...22..
..22..22..22...11...22..
..3...22..22...11...22..
.11.1133..22...11...22..
.1....22..22...1111133..
1.....3...22...11..11...
..........3........1....
```

```
Trace continues until the
fifth unit that indicates
a change in inclination,
so stops there.
Tracer --> accepts
           width(5) = 1
           offset(5) = 1

           and stops.
End of sequence twelve.


Scanner --> x = 12, y = 19
Starter --> direction = south
            width(1) = 1
Tracer --> new width = 1
           new offset = 0
           (marked as *)
```

Figure A.1.28

```
........................
..2...22...........22...
..3322333.222222222333..
..22..22................
..22..22...2.....22....
..22..22...332222333...
..22..22...22....22....
..332233...22....22....
..22..22...33222233....
..22..22...33....33....
..22..22..2.........22..
..22..22..3323222232333.
..22..22..22.2....2.22..
..331133..22.22...****..
..22..22..22..22.11.22..
..22..22..22..3..1..22..
..22..22..331111111133..
..22..22..22...11...22..
..22..22..22...11...22..
..3...22..22...11...22..
.11.1133..22...11...22..
.1....22..22...1111133..
1.....3...22...11..11...
..........3........1....
```

```
Second unit is accepted,
Tracer --> new width = 4
           new offset = 0
complex type (marked as *).

Single_handler -->
Special_handler -->
Predictor -->
LookAhead --> ...
```

Figure A.1.29

```
........................
..2...22...........22...
..3322333.222222222333..
..22..22................
..22..22...2.....22....
..22..22...332222333...
..22..22...22....22....
..332233...22....22....
..22..22...33222233....
..22..22...33....33....
..22..22..2.........22..
..22..22..3323222232333.
..22..22..22.2....2.22..
..331133..22.22...@@@@..
..22..22..22..22.**.**..
..22..22..22..3..1..22..
..22..22..331111111133..
..22..22..22...11...22..
..22..22..22...11...22..
..3...22..22...11...22..
.11.1133..22...11...22..
.1....22..22...1111133..
1.....3...22...11..11...
..........3........1....
```

```
LookAhead --> sees two units
          new width(1) = 2
          new offset(1) = -1
          new width(1) = 2
          new offset(1) = 2

Multiple_handler --> try both
paths by calling Single_handler.

The two paths are

   3       3
   2       2
   11@@    @13@
  11         22 <-- reject

so left path is chosen.
```

Figure A.1.30

```
........................
..2...22...........22...
..3322333.222222222333..
..22..22................
..22..22....2.....22....
..22..22....332222333...
..22..22....22....22....
..332233....22....22....
..22..22....33222233....
..22..22....33....33....
..22..22..2.........22..
..22..22..3323222232333.
..22..22..22.2....2.22..
..331133..22.22...3333..
..22..22..22..22.22.22..
..22..22..22..3..2..22..
..22..22..o31111131133..
..22..22..22...11...22..
..22..22..22...11...22..
..3...22..22...11...22..
.11.1133..22...11...22..
.1....22..22...1111133..
1.....3...22...11..11...
..........3........1....
```

So sequence fifteen
eventually stops.
Scanner --> x = 17, y = 11
Starter --> direction = east
            width(1) = 1

Figure A.1.31

```
........................
..2...22...........22...
..3322333.222222222333..
..22..22................
..22..22....2.....22....
..22..22....332222333...
..22..22....22....22....
..332233....22....22....
..22..22....33222233....
..22..22....33....33....
..22..22..2.........22..
..22..22..3323222232333.
..22..22..22.2....2.22..
..331133..22.22...3333..
..22..22..22..22.22.22..
..22..22..22..3..2..22..
..22..22..332233332233..
..22..22..22...11...22..
..22..22..22...11...22..
..3...22..22...11...22..
.11.1133..22...11...22..
.1....22..22...1111133..
1.....3...22...11..11...
..........3........1....
```

As before, sequence sixteen
is done.

Figure A.1.32

```
........................
..2...22...........22...
..3322333.222222222333..
..22..22................
..22..22....2.....22....
..22..22....332222333...
..22..22....22....22....
..332233....22....22....
..22..22....33222233....
..22..22....33....33....
..22..22..2.........22..
..22..22..3323222232333.
..22..22..22.2....2.22..
..332233..22.22...3333..
..22..22..22..22.22.22..
..22..22..22..3..2..22..
..22..22..332233332233..
..22..22..22...22...22..
..22..22..22...22...22..
..3...22..22...22...22..
.22.2233..22...22...22..
.2....22..22...3322333..
3.....3...22...33..22...
..........3........3....
```

The rest are similarly traced.

Figure A.1.33

APPENDIX B : SAMPLE LISTING OF PROLOG PROGRAM OF STROKE EXTRACTION LEVEL THREE

Note : program too long to be listed completely.

```
/* ------------- TURBO-PROLOG SYNTAX ------------- */

project "recogn"

code = 3000

/* program level3 ---- stroke extraction */

/* abbreviation ---
H : head, M : middle, T : tail, K : token, S : stroke,
P :  partstroke   C : connect,  L : list,
LL :  list of list, O : output, I : input   U : trunk,
A : shape, N : length, B : label
*/

include "b:global.def"  /* global definition file */


/* ----- database section ------ */

database
 sk_(strokeid,label,labellist,label,tokenidlist)
      /*  sk_ will be created in process and saved  as
          result */
 tk_(tokenid,id,label,id,label,trunkid,shape,length)
 lbl_(label,tokenidlist)
 lblk_(label,tokenlinklist)
       /* tk_,lbl_,lblk_ will be read in */
 lbls_(label,strokeidlist)
       /* lbls will be created & saved as result */

include "b:global2.def"  /* 2nd definition file */

include "b:tools.pro"    /* tool box file */


/* loop processing */

predicates

  processing(tokenidlist,string)
  loopprocessing(integer,integer,integer)
  loadtokenlist(string,tokenidlist)
  loadlabellist(string)
  loadlinklist(string)
  savestrokelist(string)
  savestrokelabellist(string)
  cleardatabase
  get_multi_(tokenidlist,integer)
```

```
clauses

 loopprocessing(_,0,_) :- !.
 loopprocessing(Start,Countdown,Countup) :-
   Charnum = Start+Countup,
   Digit2 = (Charnum div 676) mod 26 + 97,
   Digit1 = (Charnum div  26) mod 26 + 97,
   Digit0 =  Charnum          mod 26 + 97,
   char_int(C2,Digit2),
   char_int(C1,Digit1),
   char_int(C0,Digit0),
   str_char(Id2,C2),
   str_char(Id1,C1),
   str_char(Id0,C0),
   Concat(Id2,Id1,Id21),
   Concat(Id21,Id0,Id),
   concat("b:toklst3p.",Id,Filename),
   loadtokenlist(Filename, A), bound(A),nl,
   concat("b:lnklst3p.",Id,Filename1),
   loadlinklist(Filename1),nl,
   concat("b:lbllst3p.",Id,Filename2),
   loadlabellist(Filename2),nl,
   processing(A,Id),
   write("clearing databases "),nl,
   not(cleardatabase),

   Countdown2 = Countdown - 1,
   Countup2 = Countup + 1,
   loopprocessing(Start,Countdown2,Countup2).

 processing(A,Id) :-
   readdevice(keyboard),
   get_multi_(A,1),!,
   concat("b:stklst3p.",Id,Filename3),
   savestrokelist(Filename3),
   write("STROKE EXTRACTED !! saving results...."),nl,
   concat("b:lblstk3p.",Id,Filename4),
   savestrokelabellist(Filename4).

 processing(_,_) :-
   write("FAILED ...."),nl.
```

```
/* ---------- goal section ----------- */

goal
  clearwindow,
  write("input start char number:"),
  readint(Start),nl,
  write("input count:"),
  readint(Count),
  loopprocessing(Start,Count,0).



/* ---- load data section ----*/

predicates

  loadtokenloop(INTEGER,INTEGER,
                tokenidlist,tokenidlist)
  loadlabelloop(integer,integer)
  loopreadtokenid(integer,tokenidlist,tokenidlist)
  loadlinkloop(integer)
  loopreadlinkpair(integer,tokenlinklist,tokenlinklist)

clauses

/**/  loadtokenlist(Name,Lout) :-
        write("loading ",Name),
        openread(tokenlistfile,Name),
        readdevice(tokenlistfile),
        readint(Tokencount),
        loadtokenloop(Tokencount,1,[],Lout),
        closefile(tokenlistfile).

/**/
  loadtokenloop(0,_,Lin,Lin) :- !.
  loadtokenloop(Count,Countup,Lin,Lout) :-
    Countdown = Count - 1,
    Countup2 = Countup + 1,
    readint(Headid),    readint(Headlabel),
    readint(Tailid),    readint(Taillabel),
    readint(Trunkid),   readint(Shape),
    readint(Length),
    Tokenid = Countup,
    append_(Lin,[Tokenid],Lin1),
    assertz(tk_(Tokenid,Headid,Headlabel,Tailid,
                Taillabel,Trunkid,Shape,Length)),
    loadtokenloop(Countdown,Countup2,Lin1,Lout).

 /**/ loadlabellist(Name) :-
        write("loading ",Name),
        openread(labellistfile,Name),
        readdevice(labellistfile),
        readint(Labelcount),
        loadlabelloop(Labelcount,1),
        closefile(labellistfile).
```

```
/**/
  loadlabelloop(0,_) :- !.
  loadlabelloop(Count,Countup) :-
    Countdown = Count - 1,
    Countup2 = Countup + 1,
    readint(Tcount),
    Label = Countup,
    loopreadtokenid(Tcount,[],Tokenlist),
    assertz(lbl_(Label,Tokenlist)),
    loadlabelloop(Countdown,Countup2).

/**/
  loopreadtokenid(0,X,X) :- !.
  loopreadtokenid(Count,X,Y) :-
    readint(A),
    Count2 = Count - 1,
    loopreadtokenid(Count2,[A|X],Y).

/**/
  loadlinklist(Name) :-
    write("loading ",Name),
    openread(linklistfile,Name),
    readdevice(linklistfile),
    readint(Linklabelcount),
    loadlinkloop(Linklabelcount),
    closefile(linklistfile).

/**/
  loadlinkloop(0) :- !.
  loadlinkloop(Count) :-
    Countdown = Count - 1,
    readint(Labelidint),
    Labelid = Labelidint,
    readint(Linkcount),
    loopreadlinkpair(Linkcount,[],Linklist),
    assertz(lblk_(Labelid,Linklist)),
    loadlinkloop(Countdown).

/**/
  loopreadlinkpair(0,X,X) :- !.
  loopreadlinkpair(Count,X,Y) :-
    Count2 = Count - 1,
    readint(P1),
    readint(P2),
    readint(P3),
    readint(P4),
    readint(Tokenid1int),Tokenid1=Tokenid1int,
    readint(Tokenid2int),Tokenid2=Tokenid2int,
    loopreadlinkpair(Count2,[tlk_(P1,P2,P3,P4,
                         Tokenid1,Tokenid2)|X],Y).
```

```
/* ----- save data section ----- */

predicates
  savestroke
  savestrokelabel

clauses
  savestrokelist(Name) :-
    openwrite(skresultfile,Name),
    writedevice(skresultfile),
    not(savestroke),
    closefile(skresultfile),
    writedevice(screen).

  savestroke :-
    sk_(A,B,C,D,E),
    write("sk_(",A,',',B,',',C,',',D,',',E,")"),
    nl,fail.

  savestrokelabellist(Name) :-
    openwrite(lblsresultfile,Name),
    writedevice(lblsresultfile),
    not(savestrokelabel),
    closefile(lblsresultfile),
    writedevice(screen).

  savestrokelabel :-
    lbls_(A,B),
    write("lbls_(",A,',',B,')'),nl,fail.

/*----- clear database section -----*/

cleardatabase :-
  retract(tk_(_,_,_,_,_,_,_,_)),
  retract(lbl_(_,_)),
  retract(lblk_(_,_)),
  retract(sk_(_,_,_,_,_)),
  retract(lbls_(_,_)),fail.
                      /* let backtracking do the job */

/*---- character section ----*/

clauses

get_multi_([],_) :- !.
get_multi_(LK,N) :-
  multi_(LK,N,LK1,N1),write("upto ",N," done"),nl,!,
  get_multi_(LK1,N1).

/* ---- */
/* other sections of the prog'm are in separate files */




/* -------------------------------------------- */
```

REFERENCES

1. **S. Akamatsu, T. Kawatani** and **K. Komori**, Structure-Concentrated Feature for Handprinted Chinese Character Recognition, Journal of IECE of Japan, vol.65-D, 542-549, 1982.

2. **N. Babaguchi, T. Aibara, H. Sanada** and **Y. Teuka**, Extraction of Connectivity Structure of Direction Segments and Geometrical Feature Region from Character Pattern, Journal of IECE of Japan, vol.66-D, 495-502, 1983.

3. **R. Casey** and **G. Nagy**, Recognition of Printed Chinese Characters, IEEE Trans. Elect. Comput., 15, 91-101, 1966.

4. **R. G. Casey, T. D. Friedman** and **K. Y. Wong**, Automatic Scaling of Digital Print Fonts, IBM J. Res. Develop., vol.26, no.6, 657-666, 1982.

5. **R. G. Casey, T. D. Friedman** and **K. Y. Wong**, Use of Pattern Processing Techniques to Scale Digital Print Fonts, Proc. 5th Int Joint Conf. on Pattern Recognition, 872-879, 1980.

6. **S. K. Chang**, An Interactive System for Chinese Character Generation and Retrieval, IEEE Trans. on SMC., vol.3, no.3, 257-265, 1973.

7. **K. Y. Cheng** and **K. J. Chen**, A Structured Design Methodology for Chinese Character Fonts, Technical Report, Institute of Information Science, Academia Sinica, 1983.

8. **H. F. Feng** and **T. Pavlidis**, Decomposition of Polygons into Simpler Components: Feature Generation for Syntactic Pattern Recognition, IEEE Trans. on Comput., vol.24, no.6, 636-650, 1975.

9. **K. S. Fu**, Syntactic Method in Pattern Recognition, Academic Press, 1974.

10. **M. Fujisawa, Y. Nakano, Y. Kitazume** and **M. Yasuda**, Development of A Kanji OCR: An Optical Chinese Character Reader, Proc. 4th Int. Joint Conf. on Pattern Recognition, 816-820, 1978.

11. **T. Fujita, M. Nakanishi** and **K. Miyata,** The Recognition of Chinese Characters (Kanji), Using Time Variation of Peripheral Belt Pattern, Proc. 3rd Int. Conf. on Pattern Recognition, 119-121, 1976.

12. **R. C. Gonzalez** and **M. G. Thomason,** Syntactic Pattern Recognition: An Introduction, Addition-Wesley, 1978.

13. **I. Grieger, G. Wagewann** and **G. Wu,** Interactive Creation and Modification of Chinese Characters, Comput. & Graphics, vol.8, no.1, 81-92, 1984.

14. **G. F. Groner,** On-Line Computer Classification of Handprinted Chinese Characters as a Translation Aid, IEEE Trans. Elect. Comput., 856-860, 1967.

15. **J. D. Hobby** and **G. U. Guoan,** A Chinese Meta-Font, Standford Computer Science Report, 1982.

16. **W. S. Hsu, K. Takahashi, S. Ozawa** and **H. Fujita,** Ordered stroke Extraction Method for Printed Chinese Character Recognition, Journ. of IECE of Japan, vol.J65-D, no.2, 266-273, 1982.

17. **W. S. Hsu, T. Yamamoto, S. Ozawa** and **H. Fujita,** An Expansion of Pen Movement Stroke Extraction Method to Multifont Chinese Character Recognition, Journ. of IECE of Japan, vol.J65-D, no.9, 1159-1166, 1982.

18. **K. Ishii, N. Kanemaki** and **K. Komori,** Automatic Design of A Character Recognition Dictionary Based on Feature Concentration Method, Proceeding 4th Int. Joint Conf. Pattern Recognition, 804-806, 1978.

19. **D. Knuth,** Tex and Metafont, New Direction in Typesetting, Digital Press and the American Mathematical Society, 1979.

20. **P. M. Lewish, D. J. Rosenkrantz** and **R. E. Strearns,** Compiler Design Theory, Addition-Wesley, 1976.

21. **Y. Liu** and **T. Kasvand,** A New Approach to Machine Recognition of Chinese Characters. Proceeding 7th Int. Joint Conf. Pattern Recognition, 381-384, 1984.

22. **Y. L. Ma**, Chinese Character Recognition by a Stochastic Sectionalgram Method, IEEE Trans. on SMC., 575-584, 1974.

23. **Y. L. Ma**, The Pattern Recognition of Chinese Characters by Markov Chain Procedure, IEEE Trans. on SMC., 223-228, 1974.

24. **Y. L. Ma** and **J. S. Jour**, New Probabilistic Model for Chinese Character Recognition, Proc. 7th Int Joint Conf. on Pattern Recognition, 370-374, 1984.

25. **K. Maeda, Y. Kurosawa, H. Asada, K. Sakai** and **S. Watanabe**, Handprinted Kanji Recognition by Pattern Matching Method, IEEE Proceedings of Pattern Recognition, 789-792, 1982.

26. **T. Y. Mei**, LCCD, A Language for Chinese Character Design, Computer Science Department, Standford University, Standford.

27. **K. Mori** and **H. Nakano**, Computer Aided Design of Dot Matrices for Kanji Characters, Proc. 4th Int Joint Conf. on Pattern Recognition, 829-831, 1978.

28. **S. Mori** and **T. Sakakura**, Line Filtering and Its Application to Stroke Segmentation of Handprinted Chinese Characters. Proc. 7th Int Joint Conf. on Pattern Recognition, 366-370, 1984.

29. **H. Nagahashi, M. Nakatsuyama** and **N. Nishizuka**, On Effective Coding of Chinese Character Patterns and their Stroke-Sequences, Journ. of IECE of Japan, vol.J66-D, no.3, 286-293, 1983.

30. **M. Nagao**, Data Compression of Chinese Character Patterns, Proc. of IEEE, vol.68, no.7, 818-829, 1980.

31. **K. Ogawa, K. Nakane** and **H. Ikezawa**, On a Size and Type Transformation of Kanji Patterns, Journ. of IECE of Japan, vol.J65-D, no.2, 234-241, 1982.

32. **R. Oka**, Handwritten Chinese-Japanese Characters Recognition using Cellular Features, Journal of IECE of Japan, vol.J66-D, 17-24, 1983.

33. **E. Rich**, Artificial Intelligence, McGraw-Hill, 1983.

34. **Rosenfeld** and **Kak**, Digital Picture Processing, 2nd Edition, 1973.

35. **K. Sakai, S. Hirai, T. Kawada, S. Amano** and **K. Mori**, An Optical Chinese Character Reader, Proc. 3rd Int. Conf. on Pattern Recognition, 122-126, 1976.

36. **M. Shiono, H. Sanada** and **Y. Tezuka**, A Method for Transformation of Ming Type Kanji Patters into Gothic Type, Journ. of IECE of Japan, vol.J65-D, no.11, 1366-1371, 1982.

37. **W. Stallings**, Approaches to Chinese Character Recognition, Pattern Recognition, 8, 87-98, 1976.

38. **W. Stallings**, Chinese Character Recognition, Syntactic Pattern Recognition and Applications, Springer-Verlag, New York, 1977.

39. **J. W. Tai**, A Syntactic-Semantic Approach for Chinese Character Recognition, Proc. 7th Int Joint Conf. on Pattern Recognition, 374-377, 1984.

40. **G. T. Toussaint**, The Use of Context in Pattern Recognition, Pattern Recognition, Vol.10, 189-204, 1978.

41. **W. H. Tsai** and **K. S. Fu**, Attrubuted Grammar -- A Tool for Combining Syntactic and Statistical Approaches to Pattern Recognition, IEEE Trans. SMC., vol.10, no.12, 1980.

42. **M. Umeda** and **S. Meguro**, Classification of Handprinted Chinese Characters by Mesh and Peripheral Pattern Matching, Journ. of IECE of Japan, 79-26, 1979.

43. **M. Umeda**, Recognition of Multi-font Printed Chinese Characters, IEEE Proceedings on Pattern Recognition, 793-796, 1982.

44. **P. P. Wang**, The Topological Analysis, Classification, and Encoding of Chinese Characters for Digital Computer Interfacing - Part I, Proc. 1st Int. Symposium on Computers and Chinese Input/Output Systems, 183-194, 1973.

45. **P. P. Wang** and **R. C. Shiau**, Machine Recognition of Printed Chinese Characters via Transformation Algorithms, Pattern Recognition, 5, 303-321, 1973.

46. **H. Yamada**, Contour DP Matching Method and Its Application to Handprinted Chinese Character Recognition, Proc. 7th Int Joint Conf. on Pattern Recognition, 389-394, 1984.

47. **K. Yamamoto**, Recognition of Handprinted Characters by Outer-most Point Method, Pattern Recognition, 12, 229-236, 1980.

48. **Y. T. Yeow** and **T. G. Tang**, Radical-Pattern-Based Chinese Character Synthesis, Proceed. of the IEEE, vol.70, no.10, 1242-1244, 1982.

49. **E. F. Yhap**, Keyboard Method for Composing Chinese Characters, IBM J. Res. Development, 60-70, 1975.

50. **E. Yhap** and **E. Greanias**, An On-Line Chinese Character Recognition System, IBM J. Res. Develop., vol.25, 1981.

51. **K. C. You** and **K. S. Fu**, A Syntactic Approach to Shape Recognition Using Attributed Grammar, IEEE Trans. SMC., vol.9, no.6, 1979.

52. **S. Zhang**, A Chinese Character Recognition System Based on Pictorial Database Technique, IEEE Proceeding on Pattern Recognition, 780-782, 1982.