A REVIEW OF TWO-DIMENSIONAL PROGRAMMING LANGUAGES*

by

Mark B. Wells

University of California
Los Alamos Scientific Laboratory
Los Alamos, New Mexico 87544

ABSTRACT

Everyday mathematical language, as it appears on the printed page for instance, non-trivially makes use of two dimensions. This paper discusses various technical aspects of programming languages which utilize such natural symbolism. First, some terminology which is and can be useful in describing these systems is presented. Then, the man-machine interface in relation to two-dimensional languages in general is examined. This is followed by an historical survey of particular languages, including a brief description of their two-dimensional characteristics. Finally, a review of the analysis problem for such languages is given.

I. INTRODUCTION

We should occasionally remind ourselves that the raison d'etre of computers is to do computing, to carry out series of calculations of interest to a human but too long-winded, involved, or routine to be successfully carried out by him. The computer is merely a tool, built to do the bidding of people. Thus the language used for specifying these calculations should be natural to people, not necessarily to machines. Of course, the use of computers has taught us the importance of precision in our (algorithmic) language, but precision and naturalness are certainly not mutually exclusive. A well-written scientific report can make use of notation familiar to workers in a field yet still precisely describe a particular endeavor.

In the scientific community at least, mathematical symbolism has grown to be a basic and natural language. One might argue that whereas mathematics has to do with functions defined as abstract mappings, computing has to do with the detailed calculation of these mappings and therefore requires an extended symbolism. However, the fact remains that mathematics does provide us with an extremely rich and natural base language which we can extend as needed to accommodate the specification of algorithms. In the opinion of this author, much less extension is called for than many programming language designers have heretofore considered necessary.

One aspect of established mathematical notation which is overdue in finding its way into programming languages is the use of two dimensions. Of course the primary reason for our reliance on one-dimensional languages has been the lack of adequate hardware to permit input of two-dimensional data. A less important but still very real reason is inertia. "Since I already know Fortran, why should I be concerned with further language developments?", "Do you really find it difficult to write scripts in a one line form?" are questions frequently asked. However, irrespective of hardware and related economic matters, such questions reflect a twisted viewpoint. No justification whatsoever should be needed concerning the use of common

*This work partly supported by the United States Atomic Energy Commission and partly by National Science Foundation Grant AG-383, Proposal No. P2J0273.

mathematical constructions, whereas a language author should have to have very good reasons for adopting any artificial or mathematically unusual notational convention for his language.

II. WHAT IS A TWO-DIMENSIONAL LANGUAGE

What do we mean by a two-dimensional programming language? Roughly speaking it is a "programming language" which has a two-dimensional syntax, that is, one in which legal constructions make use of a plane in the placement of the tokens. In actuality, perhaps a better term might be "textbook language" to emphasize that the goal in the design of such a language is clarity of presentation for communication between humans--the written page being a prime medium for such communication. The term "graphics language", though closely related of course, does not convey well the intended meaning. The term has come to be linked particularly with languages associated with scope-display terminals, i.e. languages to assist with computer graphics, whether or not the languages themselves have a two-dimensional syntax. Note that a language or system designed to manipulate two-dimensional structures is not necessarily a two-dimensional language in the sense of this paper. For instance, editing systems or menu-type languages in which action is initiated by pointing to an existing two-dimensional display are not necessarily considered to be two-dimensional languages. Also, following Sammet [Sammet, 1969], we do not view a report generator, decision table, or coding form per se as a legitimate programming language due to its rigid format and lack of flexibility. Such a construct can appear as a data structure in a general programming language of course, and hopefully would have a natural two-dimensional representation in a two-dimensional language.

In discussing two-dimensional or textbook languages, it is convenient to distinguish between three types of information: textual, tabular, and pictorial. These correspond respectively in a textbook to normal text, tables, and figures. Of course it is possible to view tables and even text as general pictorial information, but this viewpoint is perhaps too broad for the sake of the development of practicable languages today (although Anderson does feel [Anderson, 1971a] that all three types should be studied under a general framework). Two dimensions are used in the presentation of textual information in three important ways: (1) for the display of normal mathematical expressions (e.g. scripting, displayed division), (2) for the construction of large operator symbols (Σ, $\surd$, etc.), and (3) for textual grouping (e.g. paragraph formation by line skipping and/or indentation, formula display by indentation, equation labeling, etc.). The first use we call expression display, the second symbol formation, and the third page formatting. Tabular data in a textbook language is two-dimensional by virtue of its presentation in rows and columns. This is of course a very simple use of two dimensions, but even here there can be interesting problems of interpretation when one does not require a completely fixed format or input order. For instance, is there a simple algorithm which can distinguish rows and can distinguish columns for an arbitrary hand-written matrix for which no writing order is imposed? Pictorial information is clearly the most general. It is perhaps convenient here to distinguish between pictures which are topologic (e.g. flow diagrams, trees, etc. where distance is of no concern) and pictures which are metric (e.g. scale drawings or plotted graphs where distance is important). Actually this last classification is rather academic since the inadequacy of input devices has so far kept language designers from making use of metric properties.

It should be pointed out that the classes of two-dimensional information just discussed are not mutually exclusive--matrices within expression display are essentially tabular, the boxes of a flow diagram often contain text, plotted graphs have topologic properties, etc. We introduce this terminology in order to describe better the evolution of two-dimensional languages.

III. THE MAN-MACHINE INTERFACE

Men write algorithms and machines carry them out. Thus the interface is the presentation of the algorithm by the human to the computer and the presentation of the results of the execution of the algorithm by the computer to the human. Being primarily interested in the naturalness of programming languages, we are concerned here more with the first--man to machine--than with the second. A

natural medium for a person to record the (two-dimensional) specification of the algorithm is on a sheet (or sheets) of paper. Thus, perhaps ideally, one would like to present this kind of record directly to a computer for execution. While some progress is being made toward this goal with optical scanning and scene analysis techniques [Rabinow, 1968], we are still some distance away, and system and language designers have sought alternatives. It is instructive to examine and to classify some of these alternatives.

One extreme alternative is simply to require a (human) translation from the two-dimensional form to a linear string form prior to the algorithm's preparation as paper tape, deck of cards, or terminal input. This approach was taken by GPSS [IBM, 1967] and initially by Ambit [Christensen, 1968], for example. Actually we can also view the definition of both a (two-dimensional) publication language and a (string) reference language for Algol 60 [Naur, 1963] as the adoption of this alternative. The chief trouble with this approach is of course that a human must perform the two-dimensional to one-dimensional translation. For a language which makes significant use of two dimensions this can be a non-trivial and non-productive job.

Before discussing other alternatives, it is best to introduce the matter of output. While our basic problem inherently concerns input, namely the way in which algorithms are fed to a computer, there are also questions of output involved because of computer feedback to the user, program editing, and documentation. With respect to the human translated language approach discussed in the previous paragraph for instance, there occurs the important question concerning the way in which the computer should acknowledge receipt of a source program; should it be (if at all) in two-dimensional or string form. In general, output has been less of a headache than input because hardware for producing two-dimensional output (fast line printers, plotters, etc.) was available much sooner than corresponding input hardware. In fact, the important work on Charybdis [Millen, 1968] was motivated by this discrepancy. A few computer scientists still feel that two-dimensional output, alone, for programming languages is sufficient. However, it is hard to understand how working in two dialects can be very satisfactory, particularly when a significant translation between dialects is required.

The mechanical analysis of truly two-dimensional information usually involves a two-dimensional discretization of the information so that reference to pieces can be made relative to some coordinate system. It is convenient to classify the approaches taken by various two-dimensional systems according to the manner in which this map is constructed. We say a language is a keyboard language† when preformed characters are placed on the map by striking certain keys and the point at which these characters are placed on the map is controlled by special keys such as space, backspace, linefeed, subscript, superscript, etc. which move that point relative to its old position. We say a language is a pen language, or a handwritten or tablet language, when information is placed on the map absolutely by some mechanism such as lightpen, acoustical pen, RAND tablet with stylus, vector drawing controlled by a mouse, joystick or trackball, etc. (Use of a pen merely to point at and thereby select any of a number of displayed items is considered in this paper as legitimate two-dimensional interaction only if the placement of the items in the plane is significant.) Handwritten languages are invariably associated with graphical input devices, (hence are called scope based languages), while keyboard languages may either be associated with typewriter-like devices (typewriter based languages) or with graphical devices. Once again, these classifications are not exclusive. Handwritten languages which exist today usually work in a "feedback mode". Information placed on the map is scanned and recognized, if possible, and the system displays on the scope (≡ cathode ray tube) an interpretation of this input selected from a previously established vocabulary.

For scope based languages in general, no output is necessary, although a reasonably good editing system is required so that the user can always easily verify that the computer's version of his program agrees with his original input. In reality,

---

†Such terminology is really quite inaccurate as it implies we are equating a language with its implementation, when for most cases these are and perhaps should be treated independently. However, this abuse does simplify our description and is convenient for the systems discussed in this paper.

all two-dimensional languages can benefit significantly from the existence of some way to get hardcopy output of the computer's version of this program. Typewriters and printers are generally either too slow or too inflexible. A more realistic solution is perhaps a device to make a hardcopy directly from a scope face or terminal memory. Such devices are becoming more and more prominent today.

Another factor of relevance primarily to scope based keyboard languages is the type of scope being used: storing or refreshing. Refreshing scopes permit spot editing but can suffer from flicker problems, while storing scopes have no problem with flicker but editing often requires excessive rewriting of the scope face. There is currently a healthy competition between these two approaches, both technically and economically, and it will be interesting to see developments over the next few years.

A cursory comparison of scope based keyboard languages and pen languages (next section) indicates that pen languages, though not without their problems, perhaps offer more future promise due to their flexibility. In actuality, both keyboard (typewriter and scope based) languages and pen languages lie somewhere between human translated languages and optical scanned languages on a scale which measures the degree of mechanization of the man to machine interface. Such classification really says very little about the "naturalness" of the various two-dimensional languages, but does say something about the manner and cost of their use. It is likely that truly functional programming languages will be influenced for some time to come by economic aspects of their implementability as much as by technical attributes which contribute to their naturalness, power, or elegance [Klerer, 1972].

IV. HISTORICAL SURVEY OF TWO-DIMENSIONAL LANGUAGES

Textual languages: The Algol 60 Publication language [Naur, 1963] was one of the first two-dimensional programming languages. It was of course a human translated language. Its two-dimensional capabilities were limited to subscripting and exponentiation.

The Madcap language developed at Los Alamos was an early two-dimensional programming language making use of hardware to avoid human two-dimensional to one-dimensional translation. The earliest version Madcap 3 [Wells, 1961], had only subscripting and exponentiation capabilities, but additional two-dimensional constructions have been incorporated as the language has grown and developed [Sammet, 1969]. The latest version is Madcap 6.

All versions of Madcap have been keyboard languages, and until recently, typewriter based; Madcap 6 is a scope based language. The Madcap languages have allowed representation only of textual two-dimensional information, although one form of tabular data, linear algebraic matrices, will most likely be allowed in the very near future. All three subclasses of textual display, expression formation, symbol formation, and page formatting have appeared in the Madcap languages. Expression formation has included subscripting, initially restricted and now arbitrary exponentiation, displayed division, summation and integral scripting, binomial coefficient display and Knuth's display notation for Stirling numbers of the second kind, and, recently, underlining. Underlining is used in Madcap 6, in lieu of declarations, to indicate the point of an identifier's definition. Symbol formation in earlier versions included construction of capital Σ (two less than symbols one half line apart), capital Π (two vertical bars overscored three half lines above the underscores), and various other symbols--$\forall$, $\exists$, ϵ, $\ni$, $\leqslant$, $\neq$, etc. The use of a large basic character set in Madcap 6 [Devaney and Hudgins, 1972] and the ability to plot characters in more than one size on the scope face obviates many of these constructions, but symbol formation does still exist. The two most prominent examples in the current version are the use of overprinted letters for identifier formation (e.g. $\vec{a}$, $\ddot{p}$, $\bar{R}$) and the construction of large absolute value, floor, and ceiling notations. (A recent two-dimensional language, Hal [Miller, 1972], uses such overprinting in its output to indicate the data type of the identifiers.)

Page formatting in Madcap is rather unique. Early versions used indentation to display loop and conditional bodies as well as arguments (arbitrary expressions) of input/output commands [Wells, 1963]. This use of indentation proved to be extremely natural and made Madcap source programs exceptionally readable. (Note that many published Algol programs are written in a similar manner although of course it is a begin-end pair which delimits the block.)

Madcap 6 allows a generalized use of indentation: an indentation level and a parenthesis level are entirely equivalent. Since Madcap is an "expression language" and parentheses are used to group expressions into compound expressions as well as for normal arithmetic grouping [Morris and Wells, 1972], the programmer has the capability of displaying his programs in a very readable form without extraneous parentheses pairs. For example

```
A ←
    (0, …, 19) ←
        exp ← ± 4N2
        frac ← ± 10N2
for 0 ≤ i ≤ 19 as e ∈ E as f ∈ F:
    Ai ← (e,f)
    ⋮
```

is equivalent to

```
A ← ((0, …, 19) ← (exp ← ± 4N2, frac ← ± 10N2));
for 0 ≤ i ≤ 19 as e ∈ E as f ∈ F: (Ai ← (e,f); …);
```

Note that this generalized use of indentation allows data and their declarations to be displayed a la Cobol [Sammet, 1969]. In this example, A is being defined as a 20-component sequence of ordered pairs of integers. The first "field" of the pair is a 4-bit signed integer named exp and the second is a 10-bit signed integer named frac.

Page formatting of a different (in a sense, inverse) sort has been suggested by Anderson [Anderson, 1972]. In this tablet language, the bulk of the text is kept at the left margin which is bracketed on the right by hand drawn brackets to distinguish loop and conditional bodies.

Another typewriter based language developed at Los Alamos in the early 1960's was named Colasl [Balke and Carter, 1962; Sammet, 1969]. Its use of two dimensions included expression and symbol formation. Expression formation included subscripting, exponentiation, displayed division, and the construction of large radicals, e.g.

$$\sqrt[n]{\frac{a+b}{a-b}}$$

Symbol formation included, besides common composed characters such as $\not>$, $\equiv$, and $\not\in$, the construction of fancy identifiers such as $\textcircled{n}$ and $\hat{\boxed{\gamma}}$. Colasl as did early versions of Madcap, combined fixed and free page formatting. Two fields at the beginning of a line were reserved for statement class and statement label, while the actual typing of the statement was free-field. Unfortunately, Colasl was somewhat ahead of its time and died with the demise of the expensive typewriter on which it was based.

Mirfac is a two-dimensional typewriter based language developed in England [Gawlik, 1972; Sammet, 1969]. It makes use only of simple subscripting and exponentiation. While it is considerably less sophisticated than Madcap or Colasl, it does illustrate the tremendous gain in notational clarity possible with only minimal use of two dimensions. A somewhat different approach to the preparation of input for this simple but basic use of two dimensions has been taken by the Mac language developed at the MIT Draper Laboratory [Laning and Miller, 1970]. In this language, which first appeared in 1958, a line of source program may occupy three cards of input, e.g.

Card 1	E						-			N	-	1				2								
Card 2	M						L		=	S	∪	M		(	A			+		3		B			
Card 3	S									I	=	1				I							I	,	J

Thus, simple exponents, subscripts, and other super- and sub-fields are allowed.

An important two-dimensional language was developed at the Hudson Laboratories of Columbia University during the mid-sixties [Klerer, 1972; Sammet, 1969]. The most interesting aspect of this typewriter based system is the use of specially designed keys to construct large mathematical symbols for use in expression formation. An example appears in the cited reference. While this

language is a typewriter based language, the synthesis of characters from basic strokes makes it a forerunner of several more recently developed pen languages.

One such system [Bernstein and Williams, 1968 and 1969] was developed at Systems Development Corporation primarily to illustrate the feasibility of a pen oriented, two-dimensional language using a feedback mode as discussed in Section 3. The language involves expression display and symbol formation. Expression display includes sub- and superscripting and displayed division, although the feedback linearizes the displayed division. Symbols are drawn on a RAND tablet in a fairly natural manner. Each user has a personalized dictionary of characters, each character being defined by a sequence of strokes used in its construction. Thus the feedback, which informs the user that his characters are being recognized properly, is an integral part of program preparation.

A similar, but somewhat more elaborate system was developed at The RAND Corporation [Blackwell and Anderson, 1969]. The parsing associated with this system is syntax directed [Anderson, 1968]. This allows users to modify and extend the mathematical notation available to them quite easily. Thus certain tabular (matrix) and topologic (directed graph) constructions have been made available in this system.

Another quite elaborate pen language, Tam, has been recently developed at System Development Corporation [Saylor, 1972]. The motivation for this work, as it is for nearly all such systems, is to provide more natural scientist-computer interaction. Most of the common two-dimensional mathematical constructions of expressions appear in this language including a superscript form for matrix transpose. A few special notational conventions are employed due to the ambiguities that occur in handwritten English as used in the United States (e.g. there is insufficient difference between a one, "1", and a vertical bar, "|"). Whereas the Blackwell-Anderson system combines both keyboard and pen control facilities, in this system, all input interaction involves a stylus.

Several symbolic manipulation languages, although not two-dimensional themselves, have made use of two-dimensional output in order to make the man-machine communication more natural. Among these are Aladin [Sirét, 1971], Macsyma [Martin and Fateman, 1971], Mathlab [Engleman, 1971], Reduce [Hearn, 1971], and Scratchpad [Griesmer and Jenks, 1972]. The Scratchpad language actually is a human translated two-dimensional language as its specification includes detailed rules for performing the linearization of the two-dimensional forms for the expressions the language can manipulate.

Tabular languages: Very little use has been made of two-dimensional tabular source information by programming languages. Apart from the input of matrices mentioned earlier, only the input of decision tables has been seriously pursued. Notable in this respect is the incorporation of decision tables in the language Babel [Scowen, 1971]. In this free format language, a decision table is merely another form of statement. The statement is designed in such a way, however, that a very readable table-like structure, e.g.

```
DECISIONTABLE
   if b1  T - F - ;
   if b2  - F T - ;
   if b3  T - - -
DO ( S1,  Y N Y Y ) ;
   ( S2,  Y N N N ) ;
   ( S3,  N Y Y N )
END
```

can appear in the source code. Other relevant languages are Detab [Silberg, 1971a] and Fortab [Amerding, 1962]. Source code for Detab is preprocessed into Cobol, while source code for Fortab is preprocessed into Fortran. A special issue of *SIGPLAN Notices* [Silberg, 1971b] provides an excellent starting point for the investigation of decision tables.

Topologic languages: Let us now turn to languages which involve topologically pictorial information. There have been essentially two such applications: data structure diagrams and flow diagrams. An important language of the first type is Ambit/G [Christensen, 1968]. The operands of this language are directed graphs naturally diagrammed in two dimensions. An operation on these structures consists

of a search for a subgraph that matches a specified pattern graph, followed by a replacement of the matching subgraph with a given graph. The language conveniently describes involved data structures and their transformations much as many of us do with pencil and paper (or chalk and blackboard) for communication between people. Text can appear in the boxes of a diagram, but the text itself is strictly one-dimensional. In its original form Ambit/G was a human translated language, although it has been implemented as a pen language using a tablet for input and scope for output [Rovner and Henderson, 1969]. A specialized version of Ambit for list processing is known as Ambit/L [Wolfberg, 1972]. This language also, like many others, is actually independent of its implementation, but for our purposes is a typewriter based, human translated language which makes use of teletype or printer two-dimensional feedback. Christensen [Christensen, 1971] says that the diagram sketch to encoded source translation (the two-dimensional to one-dimensional translation) is "simple and is done on the fly."

There are several simulation languages which make use of block diagrams [Shaw C., 1970]. One used for discrete simulation is GPSS [IBM, 1967]. This is a human translated language in which variously shaped boxes represent the different activities and functions. The text within or attached to a box is linear. A diagram is translated to a fixed format card form. Block diagram languages (also human translated at present) for continuous system simulation are Blodi [Karafin, 1965] and MOBSSL [Merritt and Miller, 1969]. There are plans to make MOBSSL an interactive scope based language.

Perhaps the most important system making use of flow diagrams as source material is the Grail language [Ellis et al, 1969]. This is a tablet language in which various flowchart symbols--circles, triangles, rectangles, etc. and flow lines--as well as characters from a simple alphanumeric vocabulary are drawn with a stylus. The input tablet is mounted with the output cathode ray tube so that a single writing surface is simulated. System generated (interpreted) symbols directly replace handwritten symbols. A scrubbing motion () is interpreted by the system as a signal to delete the information being overwritten. Text written within the flow diagram "boxes" is essentially one-dimensional, although in certain cases system feedback produces a two-dimensional coding form which is filled in by the programmer. Anderson [Anderson, 1972] discusses the advantages and disadvantages of the use of flow diagrams in a pen language.

A system called Two-D [Haag, 1972] recently developed at the University of San Francisco utilizes flow diagrams in a slightly different way. This is a scope based, keyboard language in which the user works in the context of a flow diagram built and modified by the user but drawn on the scope (or printer) by the system. Text is strictly one-dimensional.

A rather different tack of "two-dimensional" language development is represented by Pax [Johnston, 1970] and Compax [Narasimhan, 1966]. These are one-dimensional languages with facilities to manipulate truly two-dimensional data. The basic data are matrices of zeros and ones, called planes, which represent white and black spots of a picture. Various set- and graph-theoretic operations on these planes, which have the normal Euclidean connectivity properties, are available. The abstract machine on which Compax is based assumes both a two-dimensional memory for the storage of the planes and an input/output device for transmitting the digitized pictorial information. The advantage of these languages is of course that we think directly in terms of the two-dimensional structures. Also, there is much parallelism implied by the operations which can be exploited by properly designed hardware.

A summary chart of important two-dimensional languages appears in Table 1. For further study of two-dimensional languages, the reader is referred to the excellent bibliographies of Anderson [Anderson, 1971b] and Shaw [Shaw C., 1970].

V. RECOGNITION AND ANALYSIS

The previous section has traced the development of particular languages and their graphic features. In this section we briefly trace the evolution of associated analysis techniques. In general, the study of two-dimensional languages seems to be following a course characteristic of most scientific investigations. Initial systems are ad hoc and restrictive while later systems, as we learn and are able to generalize and unify, become more powerful and flexible. In reality, two-dimensional

TABLE 1

Summary Chart of Two-Dimensional Languages (Systems)

Features / Language (in order discussed in text)	1-D output	2-D output	Typewriter based	Scope based	(Scope based)	Scanned	Expression display	Symbol formation	Page formatting	Tabular	Topologic	Metric
	Human translated		Keyboard		Pen		Textual				Pictorial	
Algol 60 Publication	yes						scant					
Madcap 3-5			yes				yes	yes	yes			
Madcap 6				yes			yes	scant	yes			
Anderson '72					yes		yes	yes	yes			
Colasl			yes				yes	yes	scant			
Mirfac			yes				scant					
Mac			yes				scant					
Hal			yes				yes					
Klerer-May			yes				yes	yes				
Bernstein-Williams					yes		yes	yes				
Blackwell-Anderson				yes	yes		yes	yes		yes		
Tam					yes		yes	yes				
Scratchpad et al		yes					yes	yes	output			
Babel et al			yes							yes		
Ambit/G		yes			yes						yes	
Ambit/L		yes									yes	
GPSS	yes										yes	
Blodi and MOBSSL	yes				now?						yes	
Grail					yes						yes	
Two-D					yes						yes	
Pax and Compax (data)						ideal					ideal	ideal

interaction is still in its infancy--existing systems are quite primitive and we have a great deal to learn.

For several two-dimensional languages, the parsing of a two-dimensional mathematical expression involves (1) the building of a two-dimensional data structure--a matrix whose elements are blanks and characters which is an internal (to the computer) replica of the external expression, and (2) a translation of this matrix to an equivalent one-dimensional form--a string. {For keyboard languages, (1) above is often a translation from a one-dimensional form--e.g. a paper tape--to the two-dimensional matrix. However, even though the entire process is just a string to string translation, much has been gained. First, the original string is typewriter oriented while the output string is problem oriented. That is, the original string is entirely artificial while the output string is natural with respect to its eventual interpretation. It is true that Bernstein feels [Bernstein, 1971], and this author agrees, that the linearization

should be independent of any eventual semantic interpretation, but we are only aiming by this translation at a canonical string form for which parsing techniques are well known. Of course, except possibly for needing to know how to operate the keyboard, the programmer need not be aware of either string form--he works in his own language.} This approach is suitable for languages with fairly strict rules on the placement of tokens in the plane. When such is the case, the translation, (2) above, is straightforward [Wells, 1961; Balke and Carter, 1963]. The chief disadvantage of this approach is the required rigid format and/or the inability to use syntax to resolve ambiguities. For a pen language, immediate feedback to the user tends to compensate for the inability [Bernstein and Williams, 1969].

More general expression analysis methods, applicable to languages which allow symbols of arbitrary size and in less restrictive relative positions [Klerer and May, 1965; Blackwell and Anderson, 1969] have been studied. Anderson [Anderson, 1968] describes such a scheme which is syntax-directed. That is, the rules governing the analysis are read as data and the parsing algorithm is independent of the content of these rules. Particular syntactic units, expressions and characters, are described by sets of coordinates which indicate their position on the map. A syntax rule then specifies legal replacements of syntactic categories contingent on correct relative placement in two dimensions. The chief disadvantage of this approach is the large amount of time consumed by the analysis. Chang [Chang, 1969] gives a method based on operator-precedence analysis [Floyd, 1963] which when applicable, can parse an expression (more generally, a pattern) in time proportional to the square of the number of "primitive components" in the expression. It is applicable to patterns whose structure is based on a given set of "operators". A general discussion of the analysis problem for two-dimensional mathematical expressions is given by Martin [Martin, 1971].

The analysis methods mentioned so far depend on the pre-recognition of certain primitive components, i.e. the tokens of the language. For keyboard systems this is essentially automatic. In keyboard languages which have allowed symbol formation, either quite restrictive and unambiguous forms have been adopted as in Madcap's Σ constructed from "less than" characters or special characters are used for large symbol constructions as in the Klerer-May system. In either case, recognition is straightforward, though perhaps ad hoc.

For pen systems the character recognition problem is more difficult. The Grail system, for instance, uses a scheme developed at The RAND Corporation [Groner, 1966]. Here characters are recognized through an on-line analysis of the "strokes" from which they are being constructed. A character may consist of a number of distinct strokes and the order in which they are drawn is not important. For any system using this scheme, there is essentially a fixed dictionary of allowable characters; new characters may be added only by an expert. In addition, the scheme suffers slightly from an undesirable dependency on the pace at which a character is written and a difficulty in detecting overlapping text. A similar scheme was developed at Systems Development Corporation [Bernstein and Williams, 1969]. The unique feature of this scheme is that the user himself constructs, interactively, his own dictionary of characters. Such a personalized dictionary requires fewer changes of normal writing style for a given user.

The still more general question of the recognition of static (i.e. completely drawn, time independent) hand-written characters has been around for many years [Stevens, 1961; Munson, 1968; Tou and Gonzales, 1972], but perhaps is still to be satisfactorily solved for practical language systems today. Recent approaches include an attempt to better quantify the "shape" of simple characters [Morrison, 1972], and the application of linguistic methods to picture analysis [Shaw A., 1970; Narasimhan, 1970; Narasimhan and Reddy, 1971]. The aim of the linguistic approach is more generally the recognition and analysis of pictures (i.e. arbitrary two-dimensional information) rather than just that of mathematical symbols. It is possible that in the future results achieved by these general investigations will have important applications to two-dimensional languages in areas other than character recognition.

VI. CONCLUSION

What can one conclude about the future of two-dimensional languages? First, one should note that none of the commonly used programming languages of today--Fortran, Algol, PL/1, APL--are two-dimensional. The reason for this of course is that effective and economical two-dimensional input and output devices have arrived on the scene only lately. There is no doubt in this author's mind that twenty years from now linear string languages, at least from the general user's point of view, will be completely passé. The commonly used languages at that time will certainly contain standard two-dimensional mathematical expression display and many will contain some form of page formatting. (It is likely that symbol formation will no longer be a consideration as essentially unbounded hardware and/or software alphabets will be taken for granted.) Also, tabular and topologic data (e.g. decision tables, flow diagrams, data structure diagrams) will certainly play a more important role as we will have learned more about the "proper" way to express algorithmic flow of control and data relationships. It is unlikely that pictorial information which has metric properties will find its way into programming languages before general optical scanning [Balm, 1970] becomes economical.

The proliferation of scope display devices has certainly been and is an important stimulus for the development of these "natural" programming languages. This is very exciting and real progress can now be made. However, this author feels that perhaps terminals are still only a stopgap. It seems unlikely that scope display terminals will ever be as economical as telephones, say, so that they can truly substitute for paper and pencil by being available wherever work is to be done (desk or terminal room). Thus, perhaps we should still dream about a scheme using a sophisticated, hence centralized, optical scanning system with messenger service access so that man to machine communication can truly mimic man to man communication.