USE OF THE COMPILER-WRITING TOOLS LEX AND YACC TO CONSTRUCT 3-D OBJECTS

By

JIA-PYNG HWANG

Bachelor of Science

National Cheng Kung University

Taiwan, R. O. C.

1979

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fullfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 1985

USE OF THE COMPILER-WRITING TOOLS LEX AND YACC TO CONSTRUCT 3-D OBJECTS

Thesis Approved:

M J Folk
Thesis Adviser

[illegible]

[illegible]

Norman N. Durham
Dean of the Graduate College

PREFACE

This thesis describes the procedures of using LEX and YACC to construct 3-D object images. The theory of syntactic pattern recognition is introduced. Several examples are presented to illustrate the method we use in constructing the desired images.

I would like to thank Dr. G. E. Hedrick and Dr. S. Thoreson, my committee members, for their contributions and advice, and to Dr. K. L. Davis for substituting during my oral examination. A special thanks goes to my major advisor, Dr. Michael J. Folk, for his guidance and assistance on this thesis.

A final thanks is due to my parents for their encouragement and support which made this thesis possible.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION AND LITERATURE REVIEW

Statement of Problem

The three-dimensional (3-D) object image is getting more important in the field of computer-aided design. Researchers in computer graphics have developed several approaches to representing or constructing 3-D images [23, 24, 27]. Each of these approaches has its advantages and weak points. Other effective approaches are still needed to adapt to different applications.

Lin and Fu [18, 19] have proposed a syntactic approach to describing the structures of 3-D objects. The object of this thesis is to propose a similar approach which is based on the same concept but with a different construction method.

The syntactic approach grows out of the field of compiler-writing. In the process of compiler-writing there are two important phases: lexical analysis and syntax analysis. To generate the intermediate codes there exists a scheme called "syntax-directed translation", which allows semantic actions to be attached to the production rules of a context-free grammar [1]. The form of a production rule can be represented as

A --> bc { semantic actions } ;

Semantic actions are associated with each production. The UNIX system has two powerful tools, LEX and YACC, that can handle these complicated procedures adequately. LEX generates a program designed for lexical processing of character input streams [17]. YACC provides a general tool for imposing structure on the input to a computer program [15].

The basic idea for this thesis is to use these practical compiler-writing tools to construct 3-D object images from small sets of simple patterns of 2-D primitives. We can regard the notations in this environment as similar to those used for a programming language, where the terminal symbol (token) represents each 2-D primitive which will be recognized by LEX. The whole structure of the 3-D image will be constructed via the production rules of YACC. The input should be a sentence defined by those rules. The process is analogous to that of parsing a language sentence.

In this thesis we will develop two different kinds of approaches: the static construction approach and the dynamic construction approach. The static approach is similar to the scheme proposed by Lin and Fu [18]. Corresponding to each single object there is a unique set of production rules. For constructing a different object the production rules and the semantic actions associated with these rules must be redefined. The grammatical form in this thesis is simpler than the 3-D plex-grammar proposed by Lin and Fu

[18], and can be revised very easily, if necessary.

The dynamic approach is more flexible and convenient in performing image construction. We consider a 3-D object to be composed of many small 2-D "primitive cells". By controlling the expanding directions of these cells we can construct a desired object. Only LEX is needed for this approach, and its rules are fixed. We need only input terminal symbols representing primitive cells and their directions. Cells of various sizes and shapes can be defined for designing different objects. We can even establish a "cell bank" so that diverse cells can be retrieved for many different applications.

A common attractive aspect of these approaches is the recursive property of grammars. A grammar rule can be applied any number of times for some basic structures, resulting in a very compact way of representing the infinite sets of sentences.

In this thesis we assume that all necessary descriptions of 2-D primitives have been provided, though they can be calculated via simple geometric methods. We will display several examples. All cases are implemented on the GIGI graphics terminal.

Literature Review

The syntactic pattern recognition approach applied to computer graphics and image processing has gained much attention recently. Many papers related to this field have been published. Several typical methods have been developed and discussed, and some tasks have been performed in these methods. All aspects are based on the same theory.

Rosenfeld [25] has discussed some problems about image pattern recognition. Fu [8] gives a comprehensive introduction of the syntactic pattern recognition approach, including theory and many applications. We will describe the basic theory in the next section.

Some discussions of the syntactic approach are concentrated on the applications of 2-D images. Chen [4] collects several articles describing the applications to signal processing by using the syntactic pattern recognition method. Pavlidis et al [22] describe a parser whose input is a piecewise linear encoding of a contour and whose output is a string of high-level descriptions: arcs, corners, protrusions, intrusions, etc. Such a representation can be used not only for description but also for recognition. Simple grammars are used by them for the contour description. You et al [28] propose a syntactic method used to describe the structure of a two-dimensional shape by grammatical rules and the local details by primitives. They use both semantic and syntactic information to perform the primitive extraction and syntax analysis at the same step. Another application of syntactic methods in computer graphics is also presented by Slavik [26]. He uses attributed pair grammars for syntax directed translation of picture descriptions in appropriate data structures. The attributes describe geometric relations among graphical objects. Belaid et al [3] propose a system for the interpretation of 2-D mathematical formulas based on a syntactic parser. This system is able to recognize a large class of 2-D mathematical formulas written on a graphics tablet.

Since the tendency is towards using 3-D object in computer graphics, the applications of syntactic approach to 3-D images are getting more popular. Requicha et al [23, 24] and Srihari [27] have introduced some concepts for constructing 3-D object from 2-D primitives. Gips [12] describes a syntax-directed program that performs a three-dimensional perceptual task. His program uses a fixed set of syntactic rules to analyze line drawings. He mentioned that this is the first use of formal syntactic techniques in the analysis of pictures of three-dimensional objects. Jakubowski [13] uses syntactic methods to describe rotary machine elements defined by contours. Segments are defined as intervals of straight lines or curves. A broken line constructed of segments is a contour. Similar configurations of segments are included in a class. All such classes are subsets of a language generated by the local adjunct grammar. An algorithm deciding if any contour belongs to a class has been given. Choi et al [5] describe an algorithmic procedure to identify machined surfaces for a workpiece directly from its 3-D geometric description. They define a machined surface type as a pattern of faces, and use a syntactic pattern recognition method to find the machined surface from the boundary file. A 3-D object representation scheme which uses surfaces as primitives and grammatical production rules as structural relationship descriptors is proposed by Lin et al [18]. Possible selections of surface primitives are discussed in their paper, and several examples are given to illustrate the object description method.

In this thesis we adopt concepts similar to those of Lin et al but use different implementation methods. Lin's method with a simple example will be described in the following section.

Structure of the Thesis

The next chapter of this thesis, Chapter II describes the basic theory of the syntactic pattern recognition approach, illustrating the method that Lin and Fu [18] use to represent 3-D objects, and providing a brief overview of LEX and YACC. Chapter III discusses in detail the procedures we propose, including the programs and several examples. The advantages and limitations of the approaches, and the potential of applications will be discussed in Chapter IV. And finally, in Chapter V, we will summarize the methods used in this thesis, drawing conclusions, and suggesting the future research directions.

CHAPTER II

BACKGROUND THEORY

Basic Theory

In this section we give a simple introduction to the basic theory of general syntactic pattern recognition system. This will be followed by a description of Lin's method, then an overview of LEX and YACC.

Fu [8, 9, 10] shows a block diagram of a syntactic pattern recognition system as Figure 1. The block diagram has been divided into the recognition part and the analysis part, where the recognition part consists of preprocessing, primitive extraction (including relations among primitives and subpatterns), and syntax (or structural) analysis, and the analysis part includes primitive selection and grammatical (or structural) inference.

In the syntactic approach, a pattern is represented by a sentence (a string, a tree, or a graph of pattern primitives and their relations) in a language which is specified by a grammar. This approach draws an analogy between the structure of patterns and the syntax of a language. The language which provides the structural description of patterns is sometimes called the "pattern description language". The rules governing the composition of

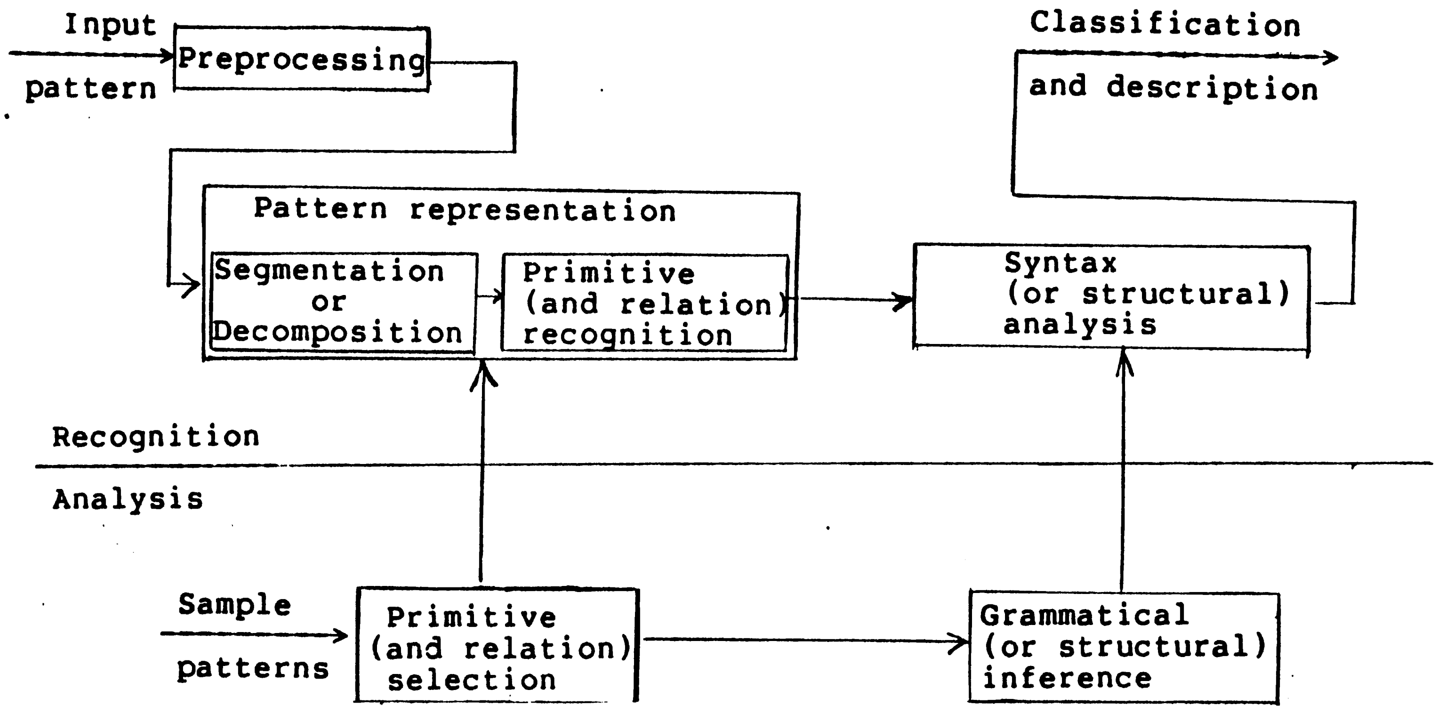
Input
pattern
Preprocessing
Pattern representation
Segmentation or Decomposition
Primitive (and relation) recognition
Syntax (or structural) analysis
Classification
and description
Recognition
Analysis
Sample
patterns
Primitive (and relation) selection
Grammatical (or structural) inference

Figure 1. Block Diagram of a Syntactic Pattern Recognition System

primitives into patterns are specified by the so-called "pattern grammar". A number of special languages have been proposed for the description of patterns such as English and Chinese characters, chromosome images, spark chamber pictures, two-dimensional mathematics, chemical structures, carotid pulse waveforms, two-dimensional airplane shapes, spoken words, and finger-print patterns.

For shape description in terms of boundary of an object, straight line segments or curve segments are often suggested as primitives. Length, slope and curvature can be used as the attributes of the primitives. The contour of an object is represented as a sequence of primitives. A set of structural or syntax rules can be inferred to characterize the structural interrelationships of these sequences (or strings of primitives) describing the object of interest.

Some higher-dimensional grammars such as web grammars, graph grammars, tree-grammars and shape grammars have been used for syntactic pattern recognition in describing high-dimensional patterns [10].

Fu [11] mentioned that a method recently proposed for syntactic shape recognition is the use of attributed grammars. In this method, a primitive is defined by a symbol and its associated attributes. The rules governing the construction of the objects from the primitives consists of syntax rules which provide the basic structural description as well as semantic or attribute rules which assign meaning to that description. This concept is very similar to the method used in this thesis.

Parsing efficiency has become a concern in syntactic recognition. Special grammars and parallel parsing algorithms have been suggested for speeding up the parsing time.

Syntactic representation of patterns such as hierarchical trees and relational graphs should also be very useful for database organization.

For more complete information on syntactic pattern recognition please refer to [8].

Description of Lin's Method

Lin and Fu [18] proposed a 3-D object description scheme using surfaces as primitives and grammatical production rules as structural relationship descriptors. They extended Feder's plex-grammar describing 2-D structures [6] to a 3-D plex-grammar. The idea is to use the attaching curve entity, considering each terminal or nonterminal symbol as a primitive or composite surface having an arbitrary number of attaching curves for joining to other surfaces. Every attaching curve has an identifier. Interconnections of entities can explicitly be made through the specified attaching curves in the grammatical production rules.

Conventionally, a grammar for a formal language is defined as a 4-tuple [2]:

G = (N, E, P, S)

where

N is a finite set of nonterminal symbols;

E is a finite set of terminal symbols, disjoint from N;

P is a finite set of (NUE)*N(NUE)*x(NUE)*; The elements in P are called productions;

S is a distinguished symbol in N called the start symbol.

A 3-D plex-grammar can be represented by a six-tuple:

G = (N, E, P, S, I, i)

where N, E, P, and S play the same roles as the formal language's, and N, E, and S represent the attaching curve entities. I is a finite set of symbols called identifiers, disjoint from (NUE). i, a member of I, is a special identifier called the null identifier.

Lin and Fu considered that an unrestricted 3-D plex-grammar is too broad to be of much practical use and then proposed the context-free 3-D plex-grammar whose productions have the form

$$A\Delta_A \text{------>} \chi\Gamma_\chi\Delta_\chi$$

It is more adequate to explain this type of productions of the 3-D context-free plex-grammar by giving an example, as below.

Figure 2 illustrates the image of a bottle to be derived. Figure 3 shows the attaching curve entities (primitives) of this image. Figure 4 is the pictorial interpretations of some production rules. The following 3-D plex-grammar generates the surfaces of this class of objects:

G = (N, E, P, S, I, i) where
N = {<BOTTLE>, <CAP>, <BODY>, <SIDE>, <BOTTOM>},
E = {<a>, <b>, <c>, <d>, <e>},
S = <BOTTLE>,
I = {0, 1, 2},
i = 0,

and P consists of the following rules:

1) <BOTTLE>{} --> <CAP><BODY><BOTTOM>{110;012}{}
2) <CAP>{1} --> <a><b>{11}{02}
3) <BODY>{2} --> <c><SIDE>{21}{10;02}
4) <SIDE>{2} --> <d><SIDE>{12}{01;20}
5) <SIDE>{2} --> <d>{}{1;2}
6) <BOTTOM>{1} --> <e>{}{1}.

The fourth production rule indicates that <SIDE> can be recursively constructed as <SIDE> attached by <d>. In this case, Δ_A= {1,2}, Δ_χ= {01;20}, and Γ_χ= {12}. $\Delta_A$indicates that <SIDE> (right-hand side) is connected to the rest of the plex by its tie curves labeled 1 and 2. The first field of Δ_χ, 01, indicates that curve 1 of <SIDE> (right-hand side) connects to the rest of the plex, while <d> is not involved in this connection. The connection is made at the curve

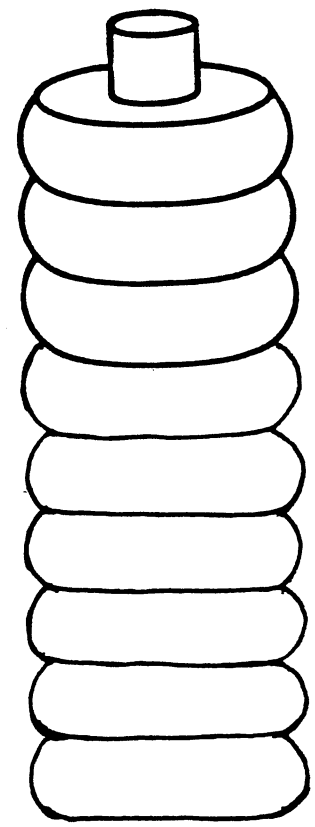

Figure 2. A Derived Bottle Image

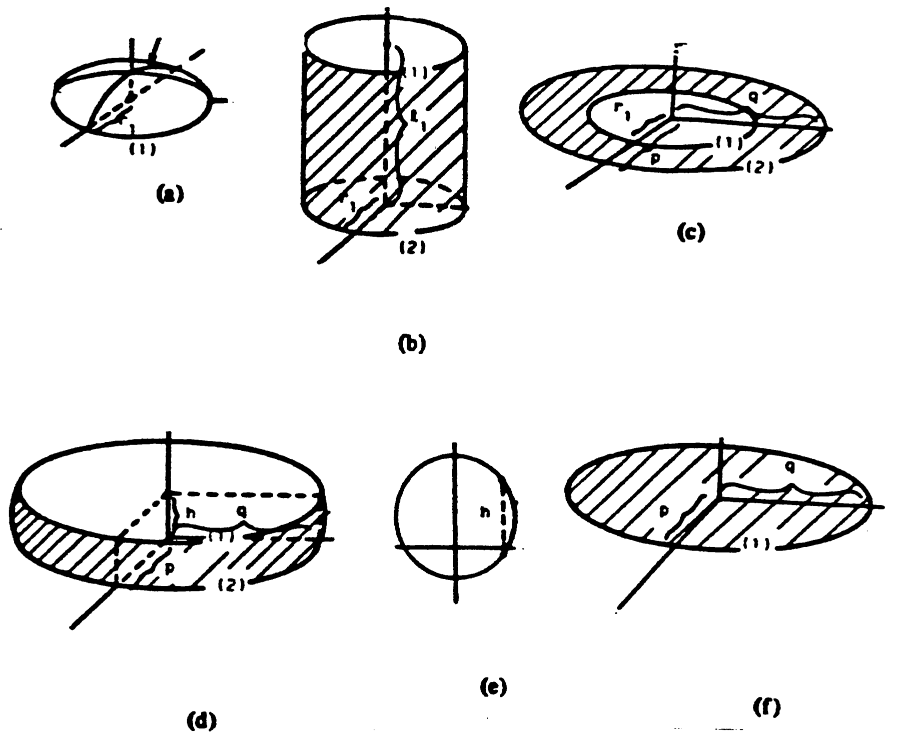

Figure 3. The Attaching Curve Entities (Primitives) of the Bottle Image

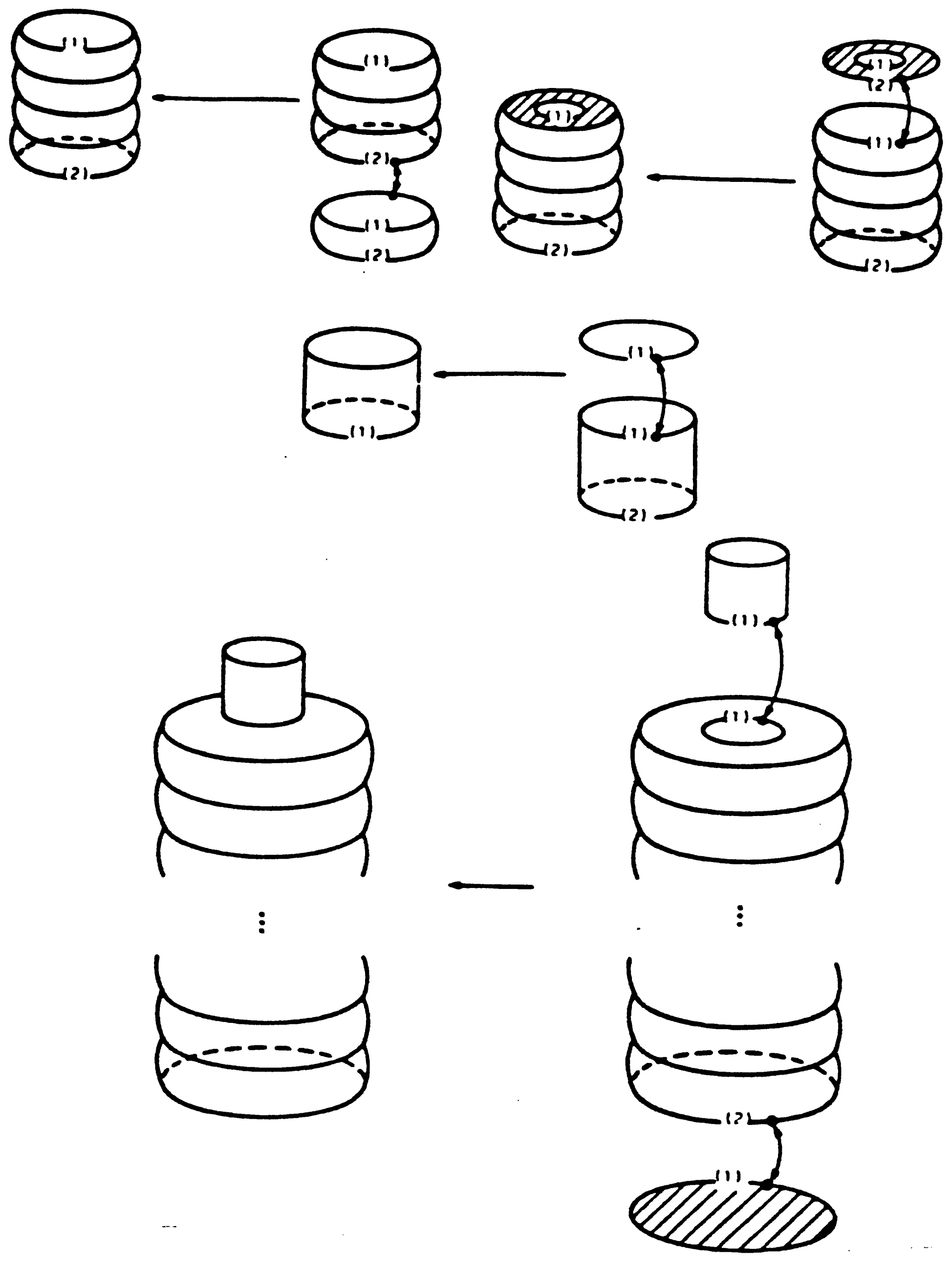

Figure 4. Pictorial Interpretations of Production Rules of the Bottle Image

corresponding to curve 1 in <SIDE> (left-hand side), as indicated by the first entry in Δ_A. That is, curve 1 of <SIDE> (left-hand side) corresponds to curve 1 of <SIDE> (right-hand side). The other field of Δ_χ, 20, indicates that curve 2 of <d> connects to the rest of the plex, while <SIDE> (right-hand side) is not involved in this connection. The connection is made at the curve corresponding to curve 2 in <SIDE> (left-hand side), as indicated by the second entry in Δ_A. That is, curve 2 of <SIDE> (left-hand side) corresponds to curve 2 of <d>. Since $\Gamma_\chi = \{21\}$, curve 2 of <SIDE> (right-hand side) is connected to curve 1 of <d>. Productions 3), 2), and 1) can be interpreted similarly.

LEX and YACC Overview

In the field of compiler-writing there are a number of tools developed specifically to help construct compilers. These tools range from scanner and parser generators to complex systems [1]. Owing to the same principle of syntactic structures we can also use some of these tools to construct 3-D object images.

In the UNIX system there are two such powerful tools called LEX [17] and YACC [15]. The use of them to construct 3-D object images is the heart of this thesis. The following sections are brief overviews of them.

LEX Description

LEX is a program generator designed for lexical processing of character input streams. It helps write programs whose control flow is directed by instances of regular expressions in the input stream, and is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.

LEX source is a table of regular expressions and corresponding program fragments. The general format of LEX source is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

where the definitions and the user subroutines are often omitted. The rules represent the user's control decisions, in which the left column contains regular expressions and the right column contains actions, program fragments to be executed when the expressions are recognized. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. The generated program is named yylex. Figure 5 is an overview of LEX.

$$\begin{tikzcd}
\text{Source} \arrow[r] & \boxed{\text{LEX}} \arrow[r] & \text{yylex} \\
\text{Input} \arrow[r] & \boxed{\text{yylex}} \arrow[r] & \text{Output}
\end{tikzcd}$$

Figure. 5 An Overview of LEX

The recognition of the expressions is performed by a deterministic finite automaton generated by LEX. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream. The lexical analysis programs written with LEX accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial lookahead is performed on the input, but the input stream will be backed up to the end of the current partition, so that the user has general freedom to manipulate it.

For the details of LEX source, regular expressions, actions, ambiguous source rules, source definition, usage, etc. please refer to [17].

YACC Description

YACC provides a general tool for imposing structure on the input to a computer program. The YACC user prepares a specification of the input process, including rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine (the lexical analyzer, LEX here) to do the basic input. The class of specification accepted is the LALR(1) grammars with disambiguating rules. The basic specification consists of three sections: the declarations, (grammar) rules, and programs. A full specification file looks like

```
declarations
%%
rules
%%
programs
```

where the declarations and programs section may be empty. The rules section is made up of one or more grammar rules. With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. A rule has the form:

```
NT : BODY { ACTIONS } ;
```

NT represents a nonterminal name, and BODY represents a sequence of zero or more names and literals.

YACC then generates a function to control the input process. This function, called yyparse, is a parser which calls the lexical analyzer (LEX here) to pick up the basic items (tokens) from the input stream. These tokens are organized according to grammar rules.

For the details of how the parser works, how it deals with ambiguity and conflicts, precedence, error handling, etc., please refer to [15].

The Combination of LEX and YACC

LEX programs recognize only regular expressions; YACC writes parsers that accept a large class of context free grammars, but require a lower level analyzer to recognize input tokens. Thus, a combination of LEX and YACC is often appropriate.

When used as a preprocessor for a later parser generator, LEX is used to partition the input stream, and the parser generator assigns structure to the resulting pieces. Figure 6 shows the flow of control in such a case.

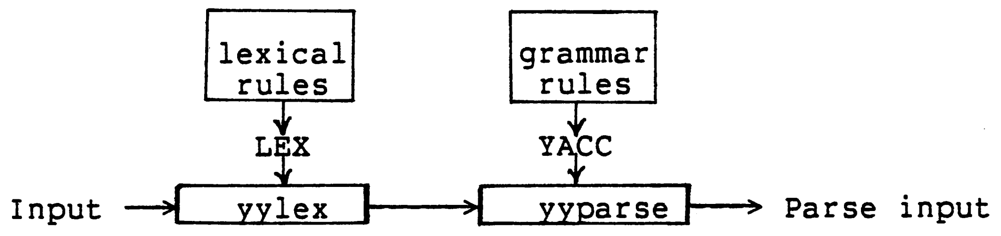

Figure 6. LEX with YACC

Normally, the default main program on the LEX library calls yylex() routine. But if YACC is loaded, and its main program is used, YACC will call yylex(). IN this case each LEX rule should end with

```
return(token);
```

where the appropriate token values is returned. Supposing the YACC source file to be yfile and the LEX source file to be lfile the UNIX command sequence can be:

```
yacc yfile
lex lfile
cc y.tab.c -ly -ll -lS
```

The YACC library (-ly) should be loaded before the LEX library (-ll) to obtain a main program which invokes the YACC parser. The generations of LEX and YACC programs can be done in either order.

CHAPTER III

CONSTRUCTION OF 3-D OBJECTS USING LEX AND YACC

Introduction

Two approaches to construct 3-D object image using LEX and YACC will be introduced in this chapter. As explained in Chapter I, these approaches are the static construction approach and the dynamic construction approach.

Detailed procedures for each method will be described, as well as the data structures of the programs, the input format, and the specification forms of LEX and YACC.

These tasks have been implemented on the GIGI graphics terminal. The line drawing and coordinates controlling functions are specific to the GIGI facilities. UNIX is used as the host system. All routines are written in the C language.

We will also show seven examples in this chapter. Example 1-6 are of the static approach, and example 7 is of the dynamic approach. For reasons of simplicity and explicitness only objects with straight line edges will be illustrated, though objects with curve edges can be constructed using the same procedures but with a little more complicated data structures and drawing functions. Integer values will be assumed as the sizes of primitives for the same reasons.

Programs Description

Data Structures and Supporting Routines

Besides the use of LEX and YACC there are several routines used to support the work. These routines and their data structures are in common for each approach. Each routine will be kept in a separated file so that it is easier to be traced and edited. These routines, together with LEX and YACC sources and their generated programs will be compiled and linked by using the program maintaining tool MAKE [7], which exists on UNIX system. The specification and usage of MAKE used in this thesis will be listed in Appendix A.

The external declarations are collected in a file called "extern.h", which will be included in the main program. Here we use a structure array

```
struct prim {
        int prix;
        int priy;
}  pmtv[]
```

to accommodate the information of 2-D primitives. If only the straight lines are used, the members of this structure simply represent the size and drawing direction of one edge of a primitive. According to the rules of the GIGI graphics terminal we take the right and down directions as positive, and the left and up directions as negative for coordinates.

The information assigned to the structure array "pmtv" includes the positive and negative values representing line drawing directions. For example, Figure 5 is a parallelogram with 5 units length in each side. The values assigned to array "pmtv" are

```
pmtv[] = {5, 0, -4, 3, -5, 0, 4, -3}
```

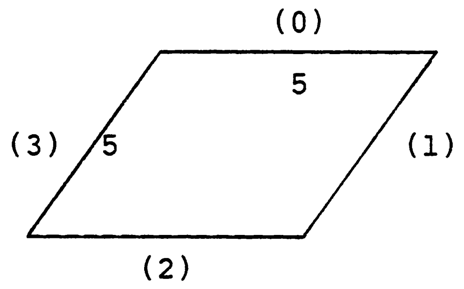

Figure 7. A Parallelogram Patch

Each pair of values represents one edge of this parallelogram. The numbers inside the parentheses represent the drawing order, starting from 0. The order can be arbitrary but is very important for the construction of the final (3-D) image. The assignment of values to the array for each edge should correspond to the edge drawing order. In Figure 5, we start from edge (0), which is to the right by 5 units horizontally and without any movement in the vertical direction. Thus the values assigned are {5, 0}. For edge (1) the movement is to the left by 4 units and down by by 3 units. Thus the values assigned are {-4, 3}, and so forth.

If there are curve edges in the primitives, e.g., quadric surfaces, the structure should include additional members to represent the curve type (vector, arc, circle, curve, etc.).

The primitive information is a data source permanently stored in array "pmtv". We have another array called "pridx" which is used to store the indexes of separated primitives. Any primitive can be retrieved any number of times via this index array when needed. If there is a lot of data, i.e., many different kinds of primitives forming a "primitive bank", which is also a more practical condition, it is more appropriate to store these data in files.

For drawing the final image the method we use is the storing of the primitive edges in a queue in the order they will be drawn. The drawing order is determined by the parameters of the function that put the order numbers into the queue and the YACC specifications (grammar rules). The queue is a large array named "plotq". If the 3-D image to be derived is a very complex one that needs too many order numbers, it is also appropriate to use a sequential file to accommodate these numbers to avoid using an extra-large array, although this makes the process run more slowly.

To put the order numbers of the primitive edges into the queue we establish several functions, named "odenq?", where the ? can be replaced by a number greater than 2. The 2-D primitives used to construct 3-D object images can be many different kinds of shapes (even for the plans with only straight line edges), e.g., triangles, rectangles, hexagons, and other regular or irregular polygons. The effects of the "odenq?" functions perform the input of the queue for these different primitive shapes. For triangles which have three edges the function "odenq3" will be used, and for rectangles the function "odenq4" will be used, and so forth. These functions are very similar in between. A typical example of "odenq4" is as

```
odenq4(dx1, dx2, dx3, dx4, n)
```

where the dx1 to dx4 are the order numbers of edges that will be put in queue, and the n represents the index in the "pridx" array. For a surface patch with five edges the queueing function is

```
odenq5(dx1, dx2, dx3, dx4, dx5, n).
```

The only difference from odenq4 is the increase of the parameter dx5 and, of course, a statement for queueing an additional edge. The stored order numbers are the indexes of edges in the "pridx" array.

There is another function named "cdcntl" which is used to control the starting point for the drawing of the next primitive patch. The form of this is

```
cdcntl(cx, cy)
```

where the parameters cx and cy are the coordinates of the next starting point. They are also stored in a queue (a one-dimensional array) to be retrieved for use.

The drawing function "tdraw" will draw the entire 3-D image. The method involves the retrieval of the order numbers stored in the queue previously which correspond to primitive edges, and the use of the GIGI commands for the line drawing of these edges. The relative coordinates capability of the GIGI terminal is quite useful in the drawing tasks.

The driving program "main" is very simple. The steps are entering the graphics mode of GIGI, clearing the screen, specifying a starting point, constructing and drawing the picture, and then escaping from the graphics mode.

The intact programs described above are listed in Appendix B - E.

LEX and YACC Sources

The LEX and YACC sources are the heart of the whole task. An overview of these tools has been shown in Chapter II. Basically, LEX performs pattern matching and YACC organizes these input patterns according to the syntactic rules provided. These are the normal procedures for a syntactic pattern recognition approach that can achieve a task. Nevertheless, as mentioned in Chapter II, both LEX and YACC have the capabilities of associating actions with their rules (regular expression rules and grammar rules). LEX is therefore powerful enough and can sometimes accomplish a job independently. In this thesis we will use LEX and YACC in the static construction approach, and use only LEX in the dynamic construction approach.

In the static construction approach there is a one to one correspondence between grammars and objects. Corresponding to each single object there is a unique set of production rules.

The LEX source for the static approach is quite simple. The regular expressions in the left-hand-side are the patterns representing 2-D primitives. They can be arbitrary symbols, or meaningful names if desired. We shall simply use one alphabet to represent each primitive. In the right-hand-side are the actions that return token numbers which will be called by YACC. A lot of patterns can be established in advance for objects that need different amounts of primitives. A source form of LEX is listed in Appendix F.

Each YACC source for the static approach contains the production rules needed for constructing an object image. This set of rules plays the role of an acceptor. As described before, in the left-hand-side are the nonterminal symbols (again they can be arbitrary symbols or meaningful names; capital letters will be used here) which derive a set of terminal symbols (the primitive patterns). Token numbers returned by LEX should be declared to correspond to the terminal symbols in the upper part of the YACC source. Associated with each production rule are semantic actions, which are the queueing functions and the coordinates control function in our project. These actions can also be put in the LEX source though, and we shall treat them this way in the dynamic approach. The YACC specifications and actions are still needed for controlling the coordinates of some discontinuous patches. Furthermore, via the grammar rules a sentence is parsed so that it is much more obvious and convenient to have an input of a sentential form for constructing the desired object. The input format will be shown embedded in the examples in the next section.

For the dynamic approach only LEX is used. The LEX source is fixed, i.e., from only one fixed source we can obtain different object images as long as the primitive symbols for the desired objects have existed in the source. This goal is achieved by controlling the extending directions of certain primitives and the switching between different primitives via the input patterns. Such primitives can be regarded as cells, and their extensions are just like the growing of cells. There may be various types of cells, i.e., the primitives with different shapes and sizes. In this approach the queueing functions and the coordinate control function are associated with the LEX's regular expression rules. The detailed image construction steps for this approach will be illustrated in the next section by a simple example.

Examples and Explanations

In this section several examples will be displayed. For each example the figures of the separated primitives and the final constructed picture will be shown. The YACC production rules and their associated actions are listed with each example, and the steps will be explained only in Example 1 for the static approach (Example 1-6). With the dynamic approach example (Example 7) the LEX source will be listed and the detailed process will be described.

Example 1

Figure 8 shows the primitives of the images in Figure 9. The YACC specification for Figure 9(a) is as below:

```
%token  a 301  b 302
%%
S   :   A E
    ;
E   :   a D  { odenq4(1, 2, 3, 0, 0); cdcntl(0,0); }
    ;
D   :   b C  { odenq4(0, 1, 2, 3, 1); cdcntl(-100,50);}
    ;
C   :   a B  { odenq4(0, 1, 2, 3, 0); cdcntl(100,0); }
    ;
B   :   b    { odenq4(0, 1, 2, 3, 1); cdcntl(-100,50);}
    ;
A   :   a    { odenq4(0, 1, 2, 3, 0); cdcntl(0,0); }
    ;
```

The input patterns for this image are

```
a a b a b
```

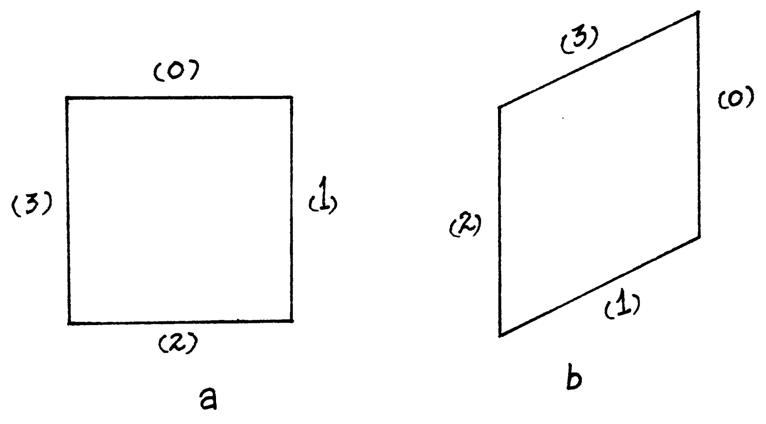

Figure 8. Primitives of Example 1

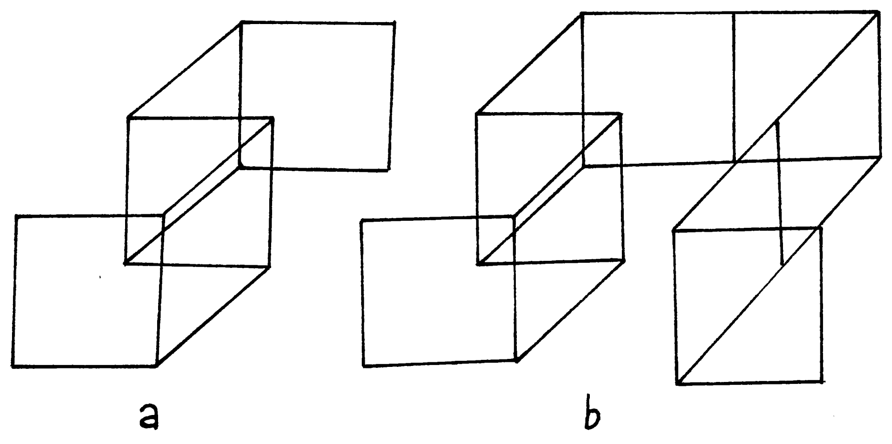

Figure 9. 3-D Images of Example 1

The YACC specification for Figure 9(b) is as below:

```
%token  a 301  b 302  c 303  d 304
%%
S   :   A K H E
    ;
E   :   a D  { odenq4(1, 2, 3, 0, 0); cdcntl(0,0); }
    ;
D   :   b C  { odenq4(0, 1, 2, 3, 1); cdcntl(-100,50); }
    ;
C   :   a B  { odenq4(0, 1, 2, 3, 0); cdcntl(100,0); }
    ;
B   :   b    { odenq4(0, 1, 2, 3, 1); cdcntl(-100,50); }
    ;
H   :   G
    |   H G
    ;
G   :   c    { odenq4(1, 2, 3, 0, 0); cdcntl(-100,0); }
    ;
K   :   J
    |   K J
    ;
J   :   d    { odenq4(1, 2, 3, 0, 1); cdcntl(100,-50); }
    ;
A   :   a    { odenq4(0, 1, 2, 3, 0); cdcntl(0,0); }
    ;
```

The input patterns for this image are

a d d c c a b a b

The parsing of the image construction sentence is a bottom-up type.

The queueing order of the edges of primitives in this example is clockwise. The order numbers (in the parentheses) are shown in Figure 8. The order can also be counterclockwise, as we shall see in other examples. Which edge in a primitive is the beginner is not crucial. It depends on the conditions that are convenient for constructing the final image. The most deeply affected part is the coordinates control for the starting point of a primitive. This task is related to both the drawing order of primitive edges and the production rules. We have to trace the last point in each stage of the construction of the picture and decide the starting point the next primitive should be drawn from. The coordinates control function "cdcntl" whose two parameters are the starting point coordinate relative to that of last point can do this job conveniently.

In this example we can see that if the last point happens to match with the starting point of the next primitive, the parameters of "cdcntl" are (0, 0), i.e., in the same position. Otherwise, the values must be filled according to the next starting point for the next starting edge of primitive. The starting edge of the next primitive is not necessarily the edge (0). The starting edge (the first one being queued) in the topmost production rule for Figure 9(a) is edge (1). It completely depends on the convenience for constructing the desired picture. Of course, the coordinates must be controlled accordingly.

The production rules for an image can also vary, as long as they can lead to the desired final 3-D picture. Sometimes the semantic action associated with a rule is only a coordinates control function for combining the partly constructed images. Some such cases can be seen in the following examples.

Example 2

Figure 10 and Figure 11 are the primitives and the 3-D object picture for this example. The YACC specification is as below:

```
%token  a 301  b 302  c 303  d 304  e 305
%token  f 306  g 307  h 308  i 309  j 310
%token  k 311  l 312
%%
STAIRS   :   UHEAD WALK DHEAD DHEAD
         ;
DHEAD    :   h f h g {odenq4(0,1,2,3,7); cdcntl(0,0);
                      odenq4(0,1,2,3,5); cdcntl(30,0);
                      odenq4(0,1,2,3,7); cdcntl(-56,83);
                      odenq4(1,2,3,0,6); cdcntl(206,-83);}
         ;
WALK     :   SEGMENT
         |   WALK SEGMENT
         ;
SEGMENT  :   SIDE1 STEP SIDE2
         ;
SIDE1    :   e d c  {odenq4(0,1,2,3,4); cdcntl(0,0);
                     odenq4(0,1,2,3,3); cdcntl(30,0);
                     odenq4(0,1,2,3,2); cdcntl(0,118);}
         ;
SIDE2    :   c d e  {odenq4(0,1,2,3,2); cdcntl(0,0);
                     odenq4(0,1,2,3,3); cdcntl(30,0);
                     odenq4(0,1,2,3,4); cdcntl(-238,51); }
         ;
STEP     :   b a  {odenq4(0,1,2,3,1); cdcntl(-28,21);
                   odenq4(0,1,2,3,0); cdcntl(178,-139);}
         ;
UHEAD    :   UBLOCK USTEP UBLOCK
                  {cdcntl(-270, -73); }
         ;
USTEP    :   l  {odenq4(0,1,2,3,11); cdcntl(150,-118); }
         ;
UBLOCK   :   k j k i  {odenq4(0,1,2,3,10); cdcntl(0,0);
                       odenq4(0,1,2,3,9); cdcntl(30,0);
                       odenq4(0,1,2,3,10); cdcntl(0,0);
                       odenq4(1,2,3,0,8); cdcntl(0,118);}
         ;
```

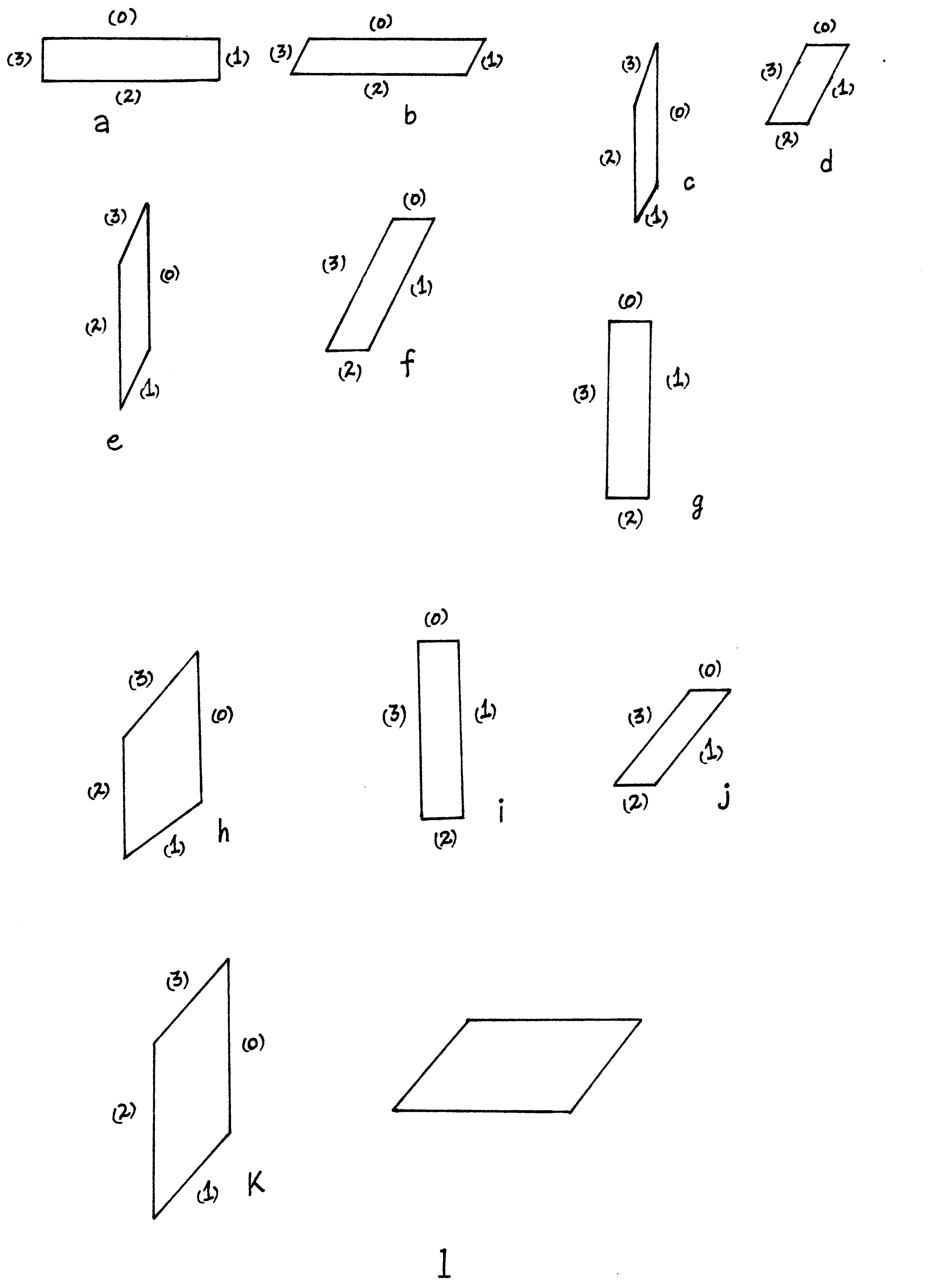

Figure 10. Primitives of Example 2

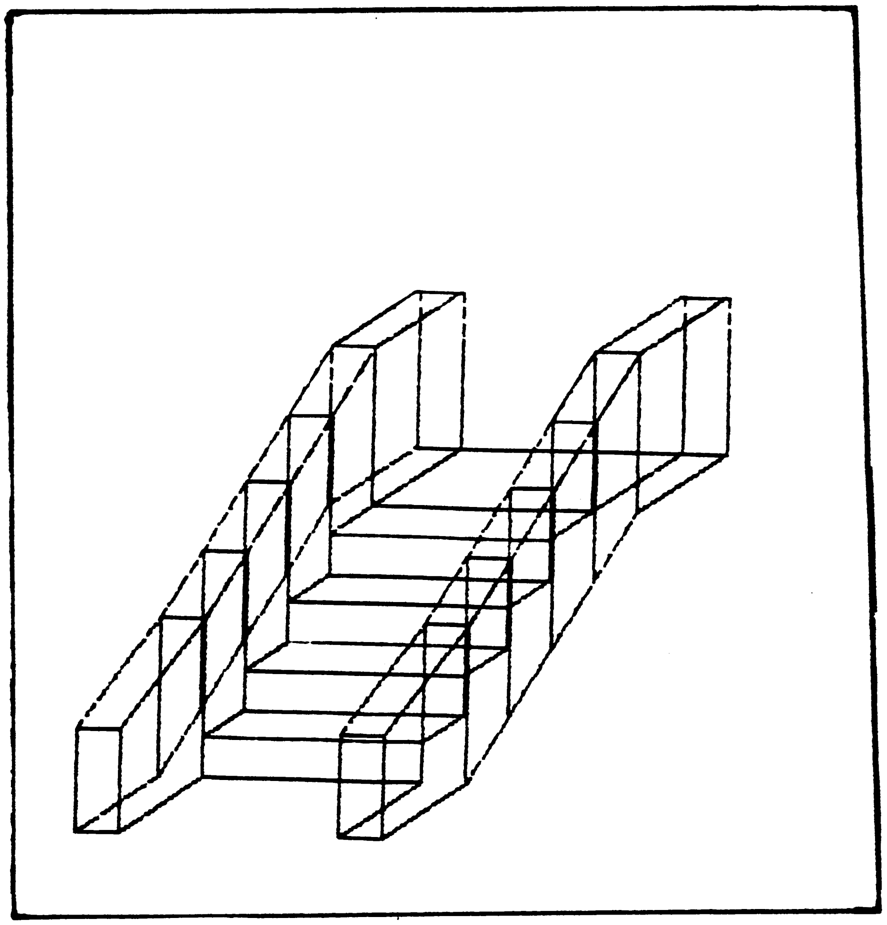

Figure 11. 3-D Picture of Example 2

The input patterns for this image are

k j k i l k j k i e d c b a c d e e

d c b a c d e e d c b a c d e e d c

b a c d e h f h g h f h g

Example 3

Figure 12 and Figure 13 are the primitives and the 3-D picture. The YACC source is as below:

```
%token  a 301  b 302  c 303  d 304  e 305
%token  f 306  g 307  h 308  i 309  j 310
%token  k 311  l 312
%%
S    :    C a  {odenq4(0,1,2,3,0); cdcntl(0,0);}
     ;
C    :    A B
     |    C A B
     ;
B    :    c b c  {odenq4(3,0,1,2,2); cdcntl(0,0);
                  odenq4(3,0,1,2,1); cdcntl(200,0);
                  odenq4(3,0,1,2,2); cdcntl(-90,-110);}
     ;
A    :    a d    {odenq4(0,1,2,3,0); cdcntl(-60,80);
                  odenq4(0,1,2,3,3); cdcntl(0,0); }
     ;
```

The input patterns for this image are

a d c b c a d c b c a d c b c a

d c b c a d c b c a

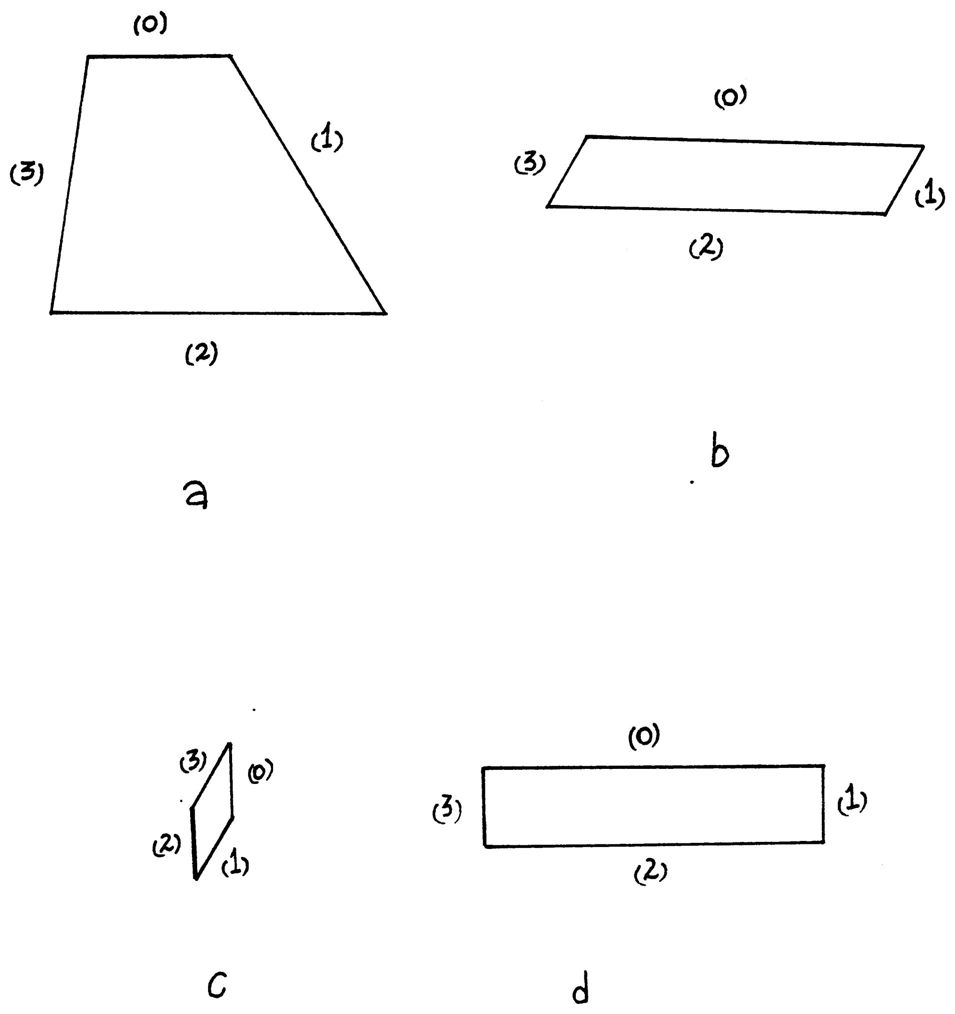

Figure 12. Primitives of Example 3

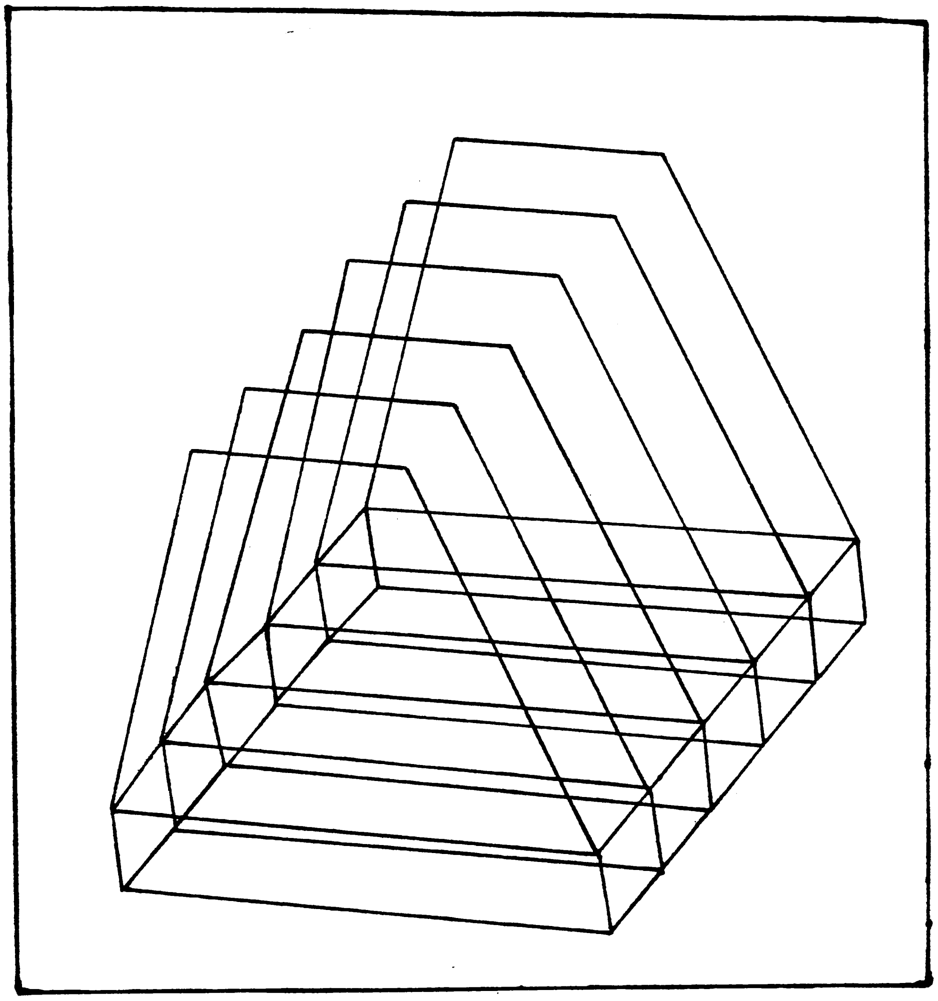

Figure 13. 3-D Picture of Example 3

Example 4

Figure 14 and 15 are the primitives and the 3-D picture. The YACC source is as below:

```
%token  a 301  b 302  c 303  d 304  e 305
%token  f 306  g 307  h 308  i 309  j 310
%token  k 311  l 312
%%
S    :   J C h  {odenq4(2,3,0,1,7); cdcntl(0,0);}
     ;
J    :   E F G I G F E  {cdcntl(-250, 30); }
     ;
I    :   i  {odenq8(1,2,3,4,5,6,7,0,8);cdcntl(140,0); }
     ;
G    :   g  {odenq4(0,1,2,3,6); cdcntl(0,0);}
     ;
F    :   f  {odenq4(0,1,2,3,5); cdcntl(80,0);}
     ;
E    :   e  {odenq5(0,1,2,3,4,4); cdcntl(0,0);}
     ;
C    :   B A
     ;
B    :   a {odenq10(0,1,2,3,4,5,6,7,8,9,0);cdcntl(160,220);}
     ;
A    :   c b d  {odenq4(0,1,2,3,2); cdcntl(0,0);
                 odenq4(0,1,2,3,1); cdcntl(140,0);
                 odenq4(0,1,2,3,3); cdcntl(-50,-140); }
     ;
```

The input patterns for this image are

```
        e f g i g f e a c b d h
```

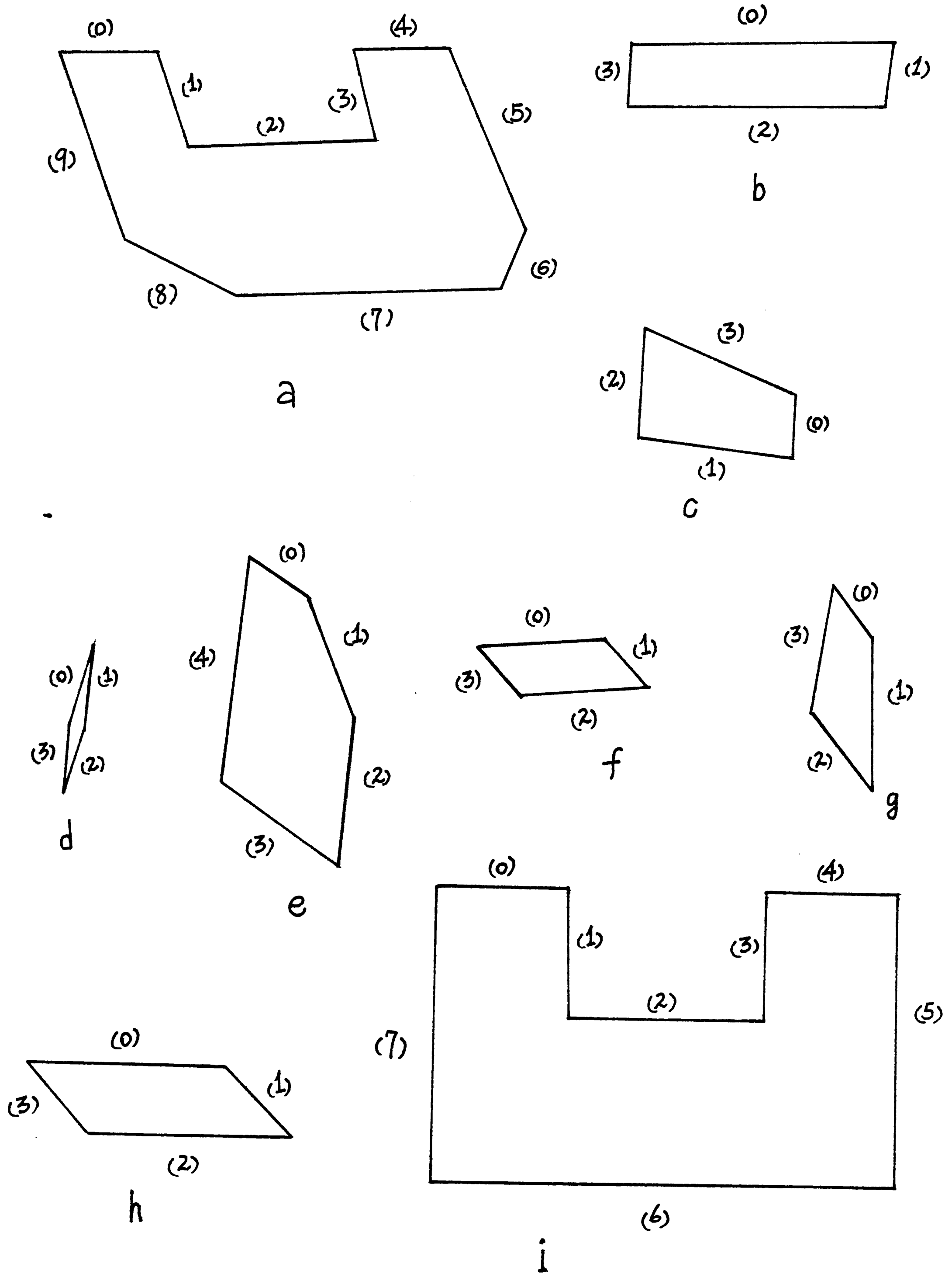

Figure 14. Primitives of Example 4

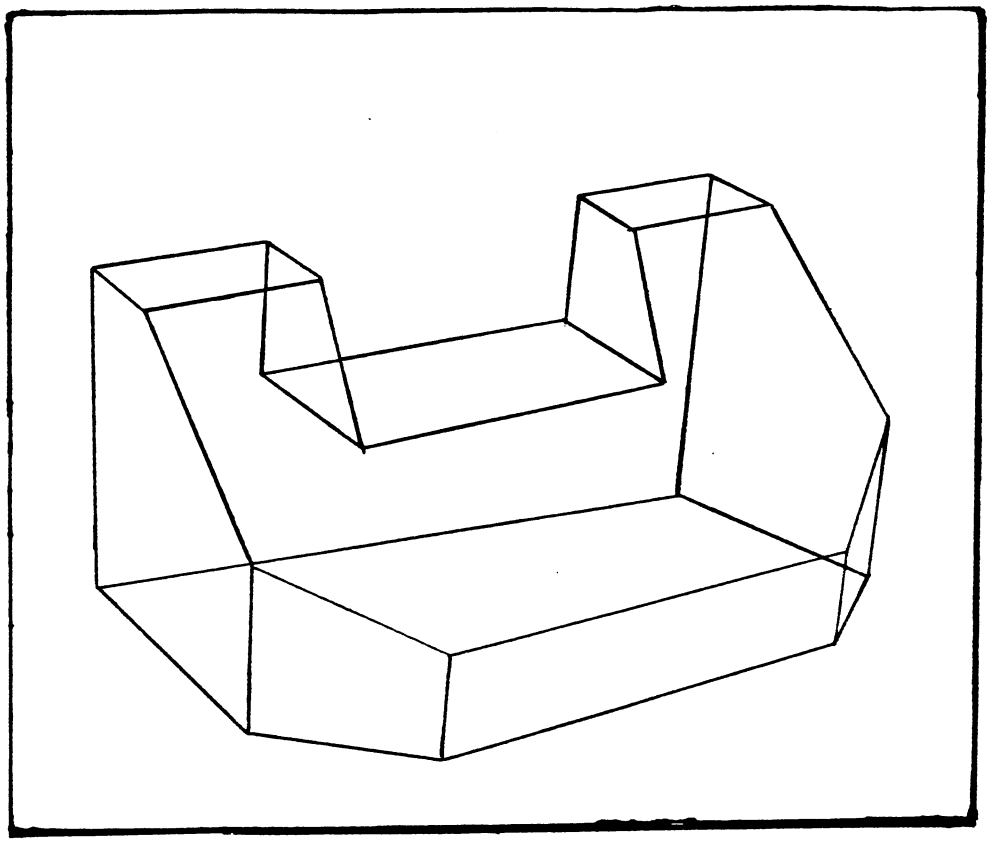

Figure 15. 3-D Picture of Example 4

Example 5

Figure 16 and Figure 17 are the primitives and the 3-D picture. The YACC source is as below:

```
%token  a 301  b 302  c 303  d 304  e 305
%token  f 306  g 307  h 308  i 309  j 310
%token  k 311  l 312
%%
S   :   U F W a  {odenq7(2,3,4,5,6,0,1,0); cdcntl(0,0); }
    ;
F   :   f  {odenq7(0,1,2,3,4,5,6,5); cdcntl(-34,118);}
    ;
W   :   V B  {cdcntl(-10, 20); }
    ;
V   :   B c d  {odenq4(0,1,2,3,2); cdcntl(100,-78);
                odenq4(0,1,2,3,3); cdcntl(80,102); }
    ;
U   :   E H I J I G H E  {cdcntl(-316, -8); }
    ;
G   :   g  {odenq8(3,4,5,6,7,0,1,2,6); cdcntl(8,-16);}
    ;
J   :   j  {odenq4(0,1,2,3,9); cdcntl(128,0);}
    ;
I   :   i  {odenq4(0,1,2,3,8); cdcntl(-8,16); }
    ;
H   :   h  {odenq4(0,1,2,3,7); cdcntl(90,12); }
    ;
E   :   e  {odenq6(0,1,2,3,4,5,4); cdcntl(0,0);}
    ;
B   :   b  {odenq4(0,1,2,3,1); cdcntl(60,8);}
    ;
```

The input patterns for this image are

```
        e h i j i g h e f b c d b a
```

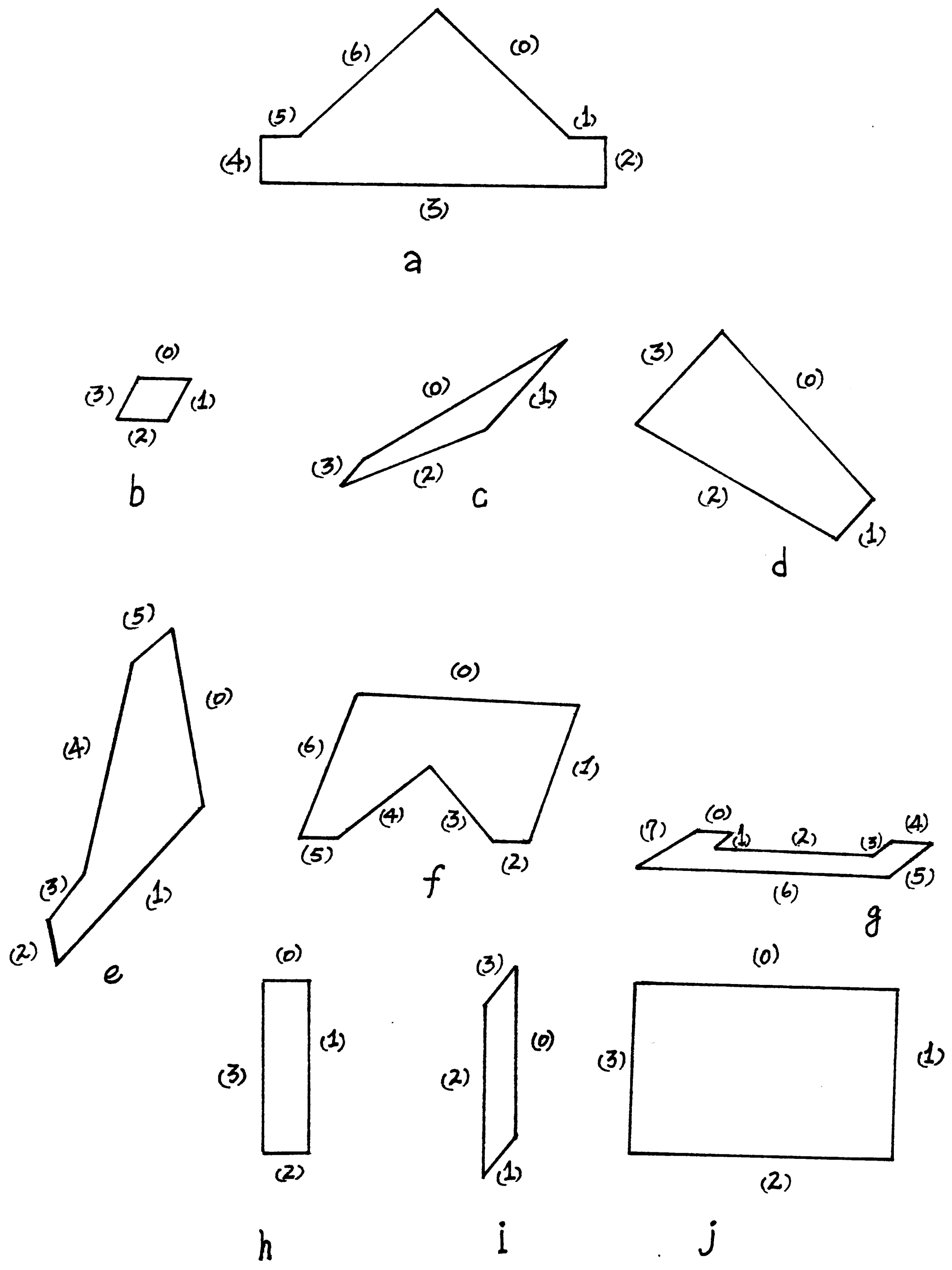

Figure 16. Primitives of Example 5

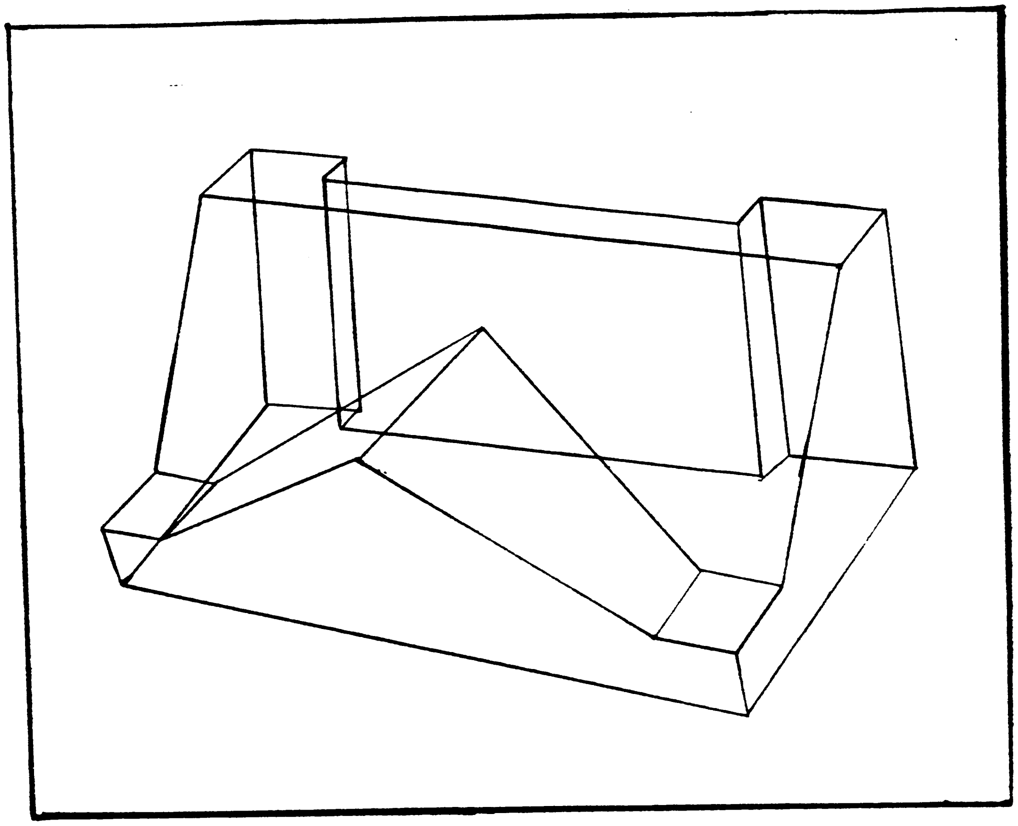

Figure 17. 3-D Picture of Example 5

Example 6

Figure 18 and Figure 19 are the primitives and the 3-D picture. The YACC source is as below:

```
%token  a 301  b 302  c 303  d 304  e 305
%token  f 306  g 307  h 308  i 309  j 310
%token  k 311  l 312
%%
S   :   C X C B A B
    ;
X   :   f d e  { odenq8(0,1,2,3,4,5,6,7,5); cdcntl(100,0);
                 odenq4(0,1,2,3,3); cdcntl(300,0);
                 odenq8(0,1,2,3,4,5,6,7,4);cdcntl(0,-150);}
    ;
C   :   c  {odenq4(0,1,2,3,2); cdcntl(0,150);}
    ;
B   :   b  {odenq4(1,2,3,0,1); cdcntl(0,-150);}
    ;
A   :   a  {odenq8(1,2,3,4,5,6,7,0,0); cdcntl(-300,150);}
    ;
```

The input patterns for this image are

```
        c f d e c b a b
```

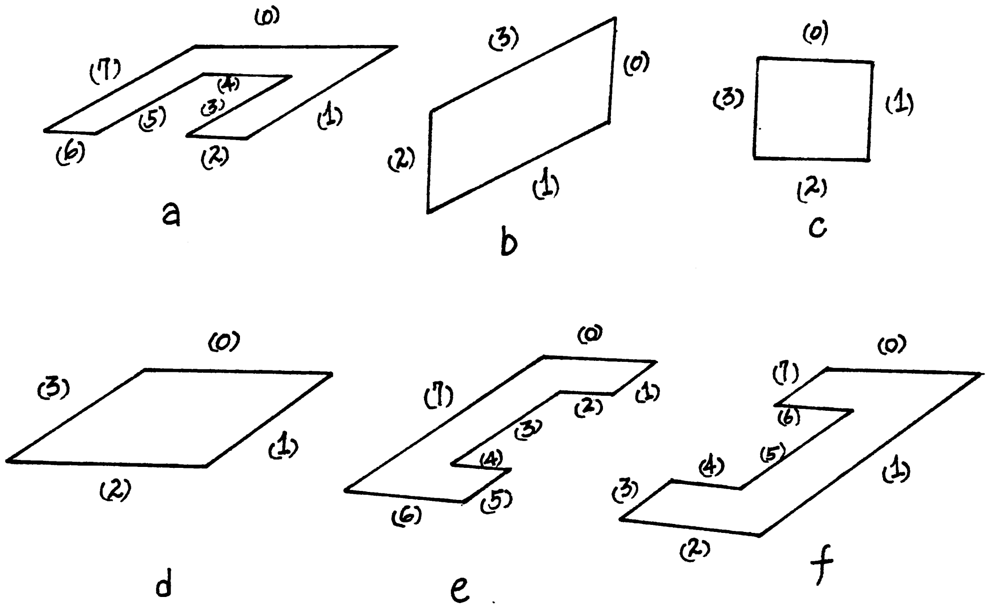

Figure 18. Primitives of Example 6

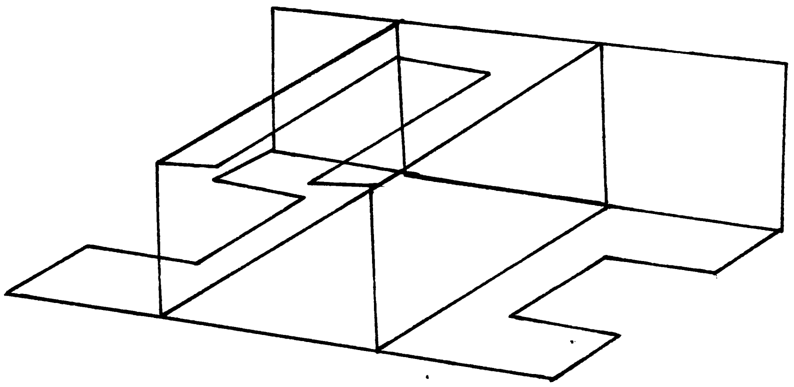

Figure 19. 3-D Picture of Example 6

Example 7

In this example the dynamic construction approach will be illustrated. Figure 20 shows a cubic block and three primitives which are decomposed from this block. The drawing order of edges is represented as before.

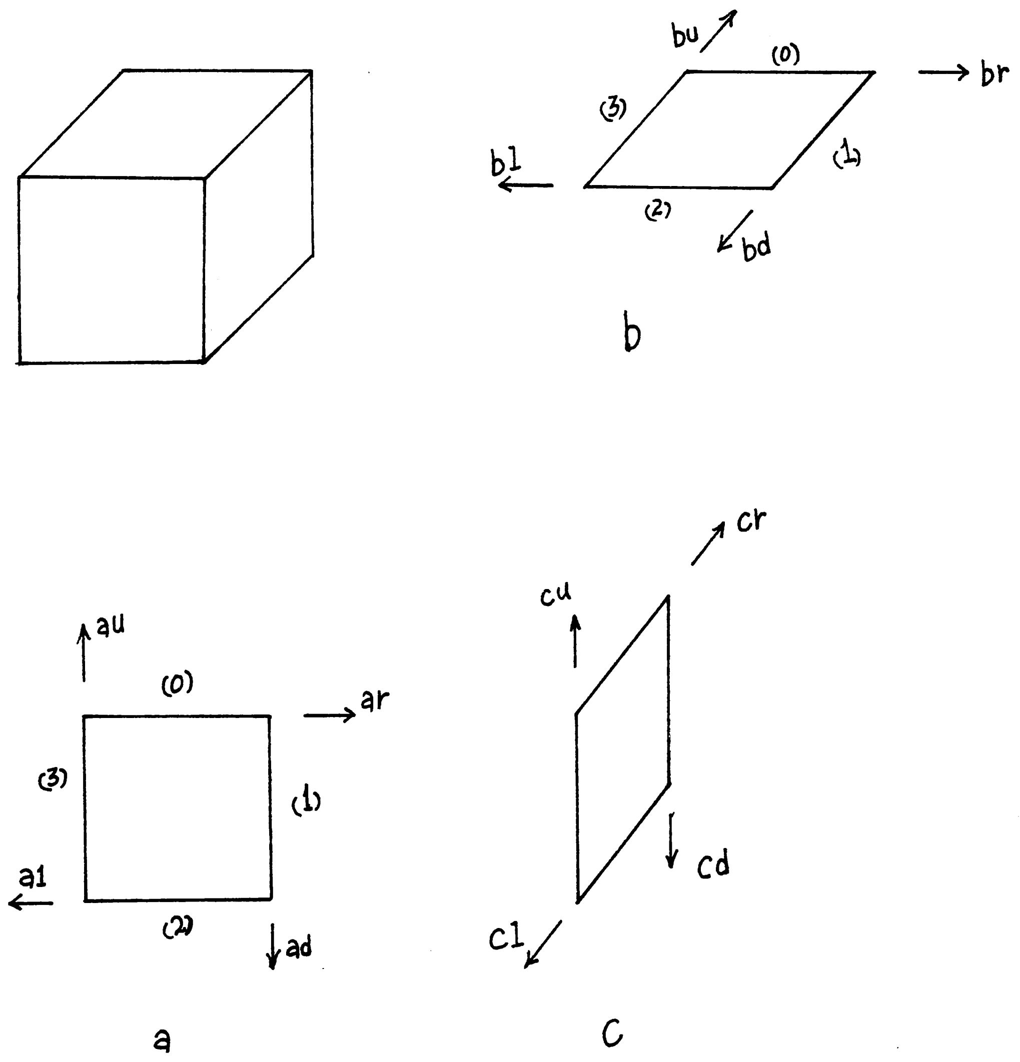

Figure 20. Primitives (Cells) of Example 7

The terminal symbols now represent not only primitives (cells) but also extending directions and switches of primitives. For example, in this case "ar" represents the primitive 'a' extending to the right direction (other symbols 'l', 'u', 'd' represent the directions of left, up, and down respectively), "ard" represents that the primitive 'a' that is now extending to the right direction will be changed to the down direction, and "arcl" represents that the primitive 'a' that is now extending to the right direction will be switched to the primitive 'c' and extended to the left direction. For the switches between different primitives (these are necessary for the construction using primitives with different shapes and orientations) some can be done directly, but some can't. If "ar" is changed to "bd", it has to be changed to "aru" or "ard" first, then to "aubd" or "adbd", and then to "bd".

The LEX source of this example is as below:

```
%{  /*  a, b, c represent the primitives.
        r, l, u, d represent the directions of right,
        left, up, and down respectively.   */
%}
%%
ar      {odenq4(0,1,2,3,0); cdcntl(100,0); }
al      {odenq4(1,2,3,0,0); cdcntl(-100,0); }
au      {odenq4(3,0,1,2,0); cdcntl(0,-100); }
ad      {odenq4(1,2,3,0,0); cdcntl(0,100); }
br      {odenq4(0,1,2,3,1); cdcntl(100,0); }
bl      {odenq4(1,2,3,0,1); cdcntl(-100,0); }
bu      {odenq4(3,0,1,2,1); cdcntl(70,-70); }
bd      {odenq4(1,2,3,0,1); cdcntl(-70,70); }
cr      {odenq4(3,0,1,2,2); cdcntl(70,-70); }
cl      {odenq4(0,1,2,3,2); cdcntl(-70,70); }
cu      {odenq4(2,3,0,1,2); cdcntl(0,-100); }
```

```
cd       {odenq4(0,1,2,3,2); cdcntl(0,100); }
aru      {cdcntl(-100,0); }
ard      {cdcntl(0,100); }
arl      {cdcntl(-100,0); }
arcr     {cdcntl(0,0); }
arcl     {cdcntl(0,0); }
alu      {cdcntl(0,0); }
ald      {cdcntl(100,100); }
alr      {cdcntl(100,0); }
alcr     {cdcntl(0,0); }
alcl     {cdcntl(0,0); }
aur      {cdcntl(100,0); }
aul      {cdcntl(0,0); }
aud      {cdcntl(100,100); }
aubu     {cdcntl(0,0); }
aubd     {cdcntl(100,0); }
adr      {cdcntl(0,-100); }
adl      {cdcntl(-100,-100); }
adu      {cdcntl(-100,-100); }
adbu     {cdcntl(-100,0); }
adbd     {cdcntl(0,0); }
bru      {cdcntl(-100,0); }
brd      {cdcntl(-70,70); }
brl      {cdcntl(-100,0); }
brcu     {cdcntl(-70,70); }
brcd     {cdcntl(0,0); }
blu      {cdcntl(0,0); }
bld      {cdcntl(30,70); }
blr      {cdcntl(100,0); }
blcu     {cdcntl(-70,70); }
blcd     {cdcntl(0,0); }
bur      {cdcntl(100,0); }
bul      {cdcntl(0,0); }
bud      {cdcntl(30,70); }
buau     {cdcntl(0,0); }
buad     {cdcntl(30,70); }
bdr      {cdcntl(70,-70); }
bdl      {cdcntl(-30,-70); }
bdu      {cdcntl(-30,-70); }
bdau     {cdcntl(-100,0); }
bdad     {cdcntl(0,0); }
cru      {cdcntl(-70,70); }
crd      {cdcntl(0,100); }
crl      {cdcntl(-70,70); }
crar     {cdcntl(0,0); }
cral     {cdcntl(0,0); }
clu      {cdcntl(0,0); }
cld      {cdcntl(70,30); }
clr      {cdcntl(70,-70); }
clar     {cdcntl(0,0); }
```

```
clal    {cdcntl(0,0); }
cur     {cdcntl(70,-70); }
cul     {cdcntl(0,0); }
cud     {cdcntl(70,30); }
cubr    {cdcntl(70,-70); }
cubl    {cdcntl(70,-70); }
cdr     {cdcntl(0,-100); }
cdl     {cdcntl(-70,-30); }
cdu     {cdcntl(-70,-30); }
cdbr    {cdcntl(0,0); }
cdbl    {cdcntl(0,0); }
```

The queueing order of edges for each primitive is determined according to the direction that the primitive is going to be extended. The changes of directions or switches between primitives are performed via the coordinates control function. The method is to find the starting point of a certain primitive in a certain direction. The desired construction can begin after the change or switch has been done.

When each input pattern is recognized its associated actions are invoked. Therefore, the construction of a picture depends on the layout of the input patterns. Figure 21 displays the steps of constructing a random picture with their corresponding input patterns. The arrows represent the route of construction.

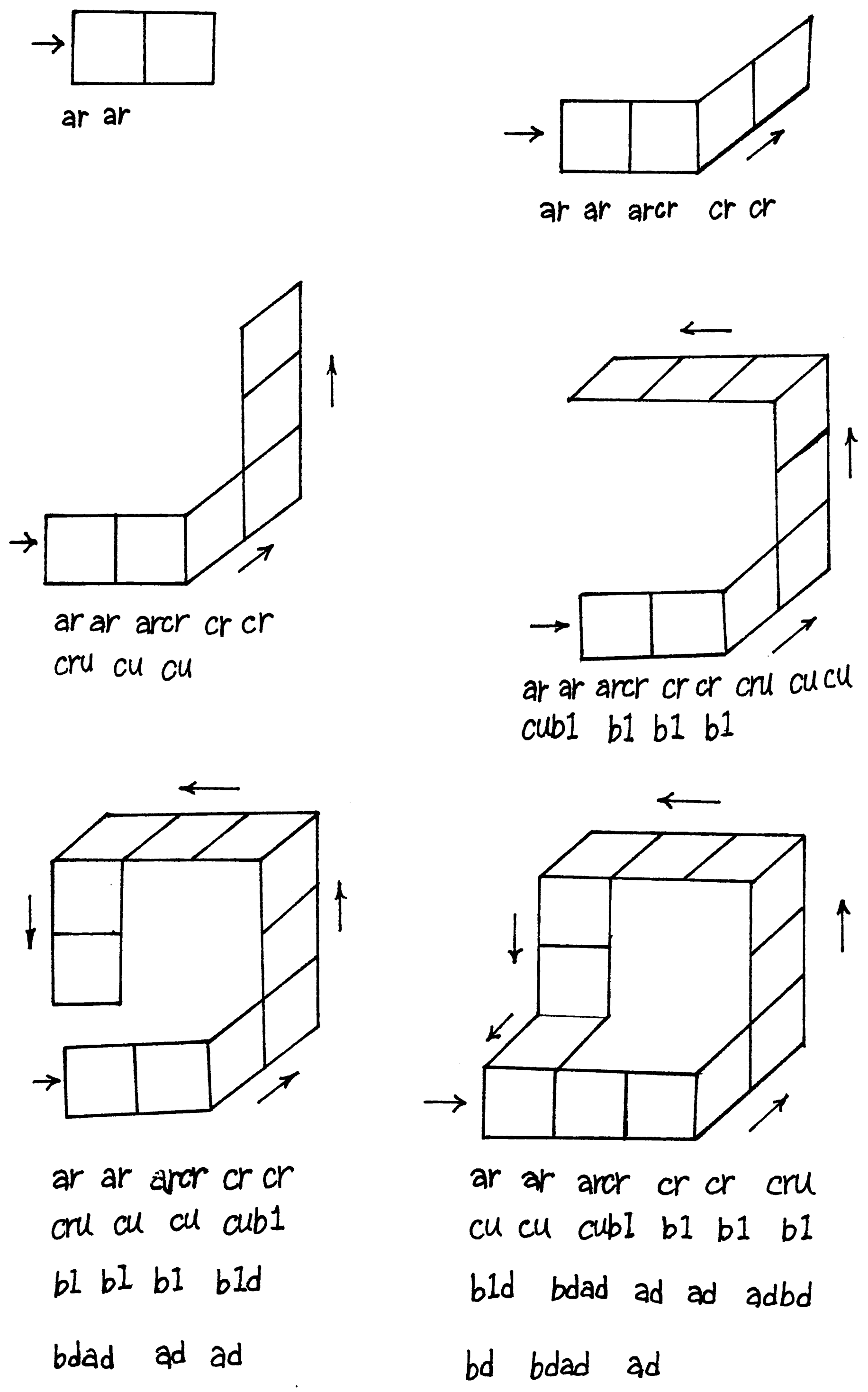

Figure 21. Construction of a Random Picture
Using Dynamic Approach

CHAPTER IV

DISCUSSION

Advantages of the Proposed Approaches

The use of LEX and YACC is an implementation of syntactic pattern recognition approach to constructing 3-D object images. Most of the advantages of the method proposed in this thesis are common to any syntactic approach to pattern recognition.

As mentioned before, these approaches are very compact and concise. Useful information can be extracted from the grammar for machine vision applications. Only a grammar is needed for describing a large number of object models due to the versatility of the grammatical production rules. If an object contains several identical primitive surface patches, only a single representation of the patch need be stored in the database.

In general, it is easier to identify the visible primitive surface patches than to recognize the object directly. This is because the primitive surface patch is simpler in shape. YACC uses a bottom-up control strategy. Its control proceeds from the identification of the visible primitive surface patches to establish the correspondence of the vertices and finally to the construction of 3-D object image.

A successful recognition also generates a structural description of the pattern. The left parse produced by YACC is a compact description of the pattern.

For the dynamic approach in this thesis only one set of LEX rules needs be established. These rules can adequately include the patterns that are very frequently used for the desired applications.

By using the grammatical production rules we can attribute the control of coordinates and order of drawing the edges of primitives to the parameters of associated functions. Therefore, it is very easy to revise the objective picture as desired.

Limitations

There are some limitations for the proposed methods:

1. The 2-D primitives are decomposed from the 3-D object to be constructed. In some cases a decomposition which does not agree with the intuitive notion has to be performed due to the limitations of the connecting rules for the surface patches. The decomposition not only creates more primitive surface patches but also adds more production rules. Thus the decomposition increases the complexity of the grammar.

2. The parsing requires an exact match between the unknown input sentence and a sentence generated by the pattern grammar. Such a rigid requirement often limits the ap-

plicability of the syntactic approach to noise-free or artificial patterns.

3. When using this approach most designers can construct the grammar only based on the a priori knowledge available and their experiences, either manually or interactively.

Potential of Applications

The syntactic pattern recognition approach is useful in many fields of applications, including character recognition, waveform analysis, speech recognition, automatic inspection, fingerprint classification and identification, geological data processing, target recognition, machine part recognition, and remote sensing [8]. Myers [21] also pointed out that pattern recognition by computer has found employment primarily in two fields: the processing of satellite or space images, and medicine.

The use of the syntactic approach in representing or constructing 3-D objects is even more important for real world applications. The world is intrinsically three dimensional. While constraints can be added to limit variability in or minimize the need for the third dimension, such information is still necessary. Very few manufactured items are two dimensional. Printed wiring boards and silicon circuits approach 2-D but still have important vertical components. Handling of objects, either manually or by robots, is intrinsically 3-D [14].

A current interesting application for 3-D object is the machine vision. Machine vision is a key to the development and use of generic parts-presentation equipment. Most industrial applications of computer vision can be categorized into two groups. They are (1) machine parts recognition and (2) visual inspection. To successfully satisfy robot vision requirements a three dimensional representation of a real scene must be provided. True 3-D vision could simplify many current robot applications that were built using less-capable 2-D vision systems. To aid in the development of 3-D vision systems representational problems must be researched.

To identify one type of machine part among many is to match it successfully against the corresponding model of those stored in memory. The model only needs enough data to identify unequivocally one part among the others that may be present. A model is a simple word description of a part which specifies the important spatial relationships between distinct components of the part. Fu [11] introduced some such methods by using a context-free grammar for building machine parts from picture primitives. Myers [21] mentioned that in industrial pattern recognition it needs to "see" only well enough to perform the task at hand.

The key point of inspection is the conformity of the part to some previously established standard. The syntactic pattern recognition approach is a way to achieve generic property verification -- a visual inspection technique. The input pattern is first extracted and processed and then represented by a string. The grammar rules are then applied to the string to detect local defects. Fu [9] described some tasks performed successfully by using this inspection technique. Apparently, such a technique can be applied successfully only when the inspection criteria can be transformed into a set of rules that can be applied equally well throughout the image. When the inspection criteria demand uneven tolerances at different places, this technique is crippled.

The advantages for industrial applications in syntactic approach are inexpensiveness, real-time processing, low error rates, and flexibility.

CHAPTER V

SUMMARY, CONCLUSIONS, AND SUGGESTIONS FOR FUTURE RESEARCH

Summary and Conclusions

The procedures of using LEX and YACC to construct 3-D object images are described in this thesis. The background theory is the same as that of the syntactic pattern recognition approach. The basic idea of syntactic pattern recognition is to represent a pattern in terms of its components and the relations among them.

LEX and YACC are the compiler-writing tools existing on UNIX system. We use these tools to implement the construction of 3-D object images from small sets of simple patterns of 2-D primitives. LEX recognizes the terminal symbol representing each 2-D primitive. The whole structures of the 3-D image will be constructed via the production rules of YACC. Some supporting functions are regarded as semantic actions associated with grammar rules or regular expression rules.

This thesis provides the programs for 3-D object construction with a hierarchical and systematic approach. It reduces the problem of identifying a 3-D object to subproblems of primitive surface patches identification and utilizes the structural relationship descriptive capability of YACC to perform structural analysis.

The method used here is also an implementation of the proposed attributed grammar for modeling 3-D object with regular shapes. It presents a framework of syntactic pattern recognition in solving 3-D object recognition problems. This approach is useful for robotic vision applications.

Suggestions for Future Research

There are some directions suitable for extending from the current work:

1. In the current scheme the primitive surface patches are fixed, both in size and orientation. The system will be more powerful and economical if functions can be developed to elongate the edges of the primitive surface patches and/or, to rotate the primitive surface patches in plane based on the existed patches. Additional condition options may be needed in programs for achieving this goal.

2. For the purpose of flexibility and applicability the research direction can be toward the implementation of error-correcting parsing which has been proposed. We can also specify the ranges for the edges of primitive surface patches to relax the restriction of dimension.

3. Ideally, it would be nice to have a grammatical inference machine which would infer a grammar or structural description from a given set of patterns. The problem of grammatical inference is concerned mainly with the pro-

cedures that can be used to infer the syntactic rules of an unknown grammar based on a finite set of sentences or strings from the language generated by this grammar. Since the use of YACC is an attributed grammar in nature, it is more difficult to perform the inference.

SELECTED BIBLIOGRAPHY

[1] Aho, A. V., J. D. Ullman. Principles of Compiler Design , Reading MA: Addison-Wesley Publishing Co., 1979.

[2] Aho, A. V., J. D. Ullman. The Theory of Parsing, Translation, and Computing (Volume I : Parsing) , Englewood Cliffs, NJ : Prentice-Hall, 1972.

[3] Belaid, A., J. Haton. "A Syntactic Approach for handwritten mathematical formula recognition." IEEE Trans. Pattern Anal. & Mach. Intell. , PAMI-6, 1 (1984), 105-111.

[4] Chen, C. H. Pattern Recognition and Signal Processing , Sijthoff & Noordhoff, 1978.

[5] Choi, B. K., M. M. Barash, D. C. Anderson. "Automatic Recognition of Machined Surfaces from a 3D Solid Model." Computer-Aided Design , 16, 2 (1984), 81-86.

[6] Feder, J. "Plex Languages." Information Science , 3 (1971), 225-247.

[7] Feldman, S. I. "Make -- A Program for Maintaining Computer Programs." Unix Programmer's Manual , 7th Edition, 1979.

[8] Fu, K. S. Syntactic Pattern Recognition and Application , Englewood Cliffs, NJ : Prentice-Hall, 1982.

[9] Fu, K. S. "Pattern Recognition for Automatic Visual Inspection." Computer , 15, 12 (1982), 34-40.

[10] Fu, K. S. "Recent Development in Pattern Recognition." IEEE Trans. Comput. , C-29, 10 (1980), 845-854.

[11] Fu, K. S. "Robot Vision for Machine Part Recognition." Proceedings of SPIE , 442, (1983), 2-14.

[12] Gips, J. "A Syntax-Directed Program That Performs a Three-Dimensional Perceptual Task." Pattern Recognition , Pergamon Press, 6 (1974), 189-199.

[13] Jakubowski, R, A. Kasprzak. "A Syntactic Description and Recognition of Rotary Machine Elements." IEEE

Trans. Comput. , C-26, 10 (1977), 1039-1043.

[14] Jarvis, J. F. "Research Directions in Industrial Machine Vision: A Workshop Summary." Computer , 15, 12 (1982), 55-61.

[15] Johnson, S. C. "YACC : Yet Another Compiler-Compiler." CSTR 32, Bell Laboratories, Murray Hill, N. J., 1975.

[16] Kernighan, B. W., D. M. Ritchie. The C Programming Language , Englewood Cliffs, NJ : Prentice-Hall, 1978.

[17] Lesk, M. E., E. Schmidt. "LEX - A Lexical Analyzer Generator." CSTR 39, Bell Laboratories, Murray Hill, N. J., 1975.

[18] Lin, W. C., K. S. Fu. "A Syntactic Approach to 3-D Object Representation." IEEE Trans. Pattern Anal. & Mach. Intell. , PAMI-6, 3 (1984), 351-364.

[19] Lin, W. C., K. S. Fu. A Syntactic Approach to 3D Object Representation and Recognition , TR-EE 84-16, School of Electrical Engineering, Purdue University, 1984.

[20] Mcfarland, W. D. "Three-Dimensional Images for Robot Vision." Proceedings of SPIE , 442, (1983), 108-116.

[21] Myers W. "Industry Begins to Use Visual Pattern Recognition." Computer , 13, 5 (1980), 21-31.

[22] Pavlidis, T., and F. Ali. "A Hierarchical Syntactic Shape Analyzer." IEEE Trans. Pattern Anal. & Mach. Intell. , PAMI-1, 1 (1979), 2-9.

[23] Requicha, A. A. G. "Representations for Rigid Solids : Theory, Methods, and Systems." Computing Surveys , 12 (1980), 437-464.

[24] Requicha, A. A. G., and H. B. Voelcker. "Solid Modeling: A Historical Summary and Contemporary Assessment." IEEE Comput. Graphics Applications , May, (1982), 9-24.

[25] Rosenfeld, A. "Image Pattern Recognition." Proc. IEEE, 69 (1981), 596-605.

[26] Slavik P. "Syntactic Methods in Computer Graphics." Eurographics'83 (1983), 133-142.

[27] Srihari, S. N. "Representation of Three-Dimensional Digital Images." Computing Surveys , 13 (1981), 399-424.

[28] You, K. C., K. S. Fu. "A Syntactic Approach to shape

Recognition Using Attributed Grammars." IEEE Trans. SMC , SMC-9, (1979), 334-345.

APPENDIX A

MAKE SPECIFICATION

Below is the MAKE specification for Example 2.

```
saml : dxenq4.o y.tab.o lex.yy.o tdraw.o main.o cdcntl.o
        cc dxenq4.o y.tab.o lex.yy.o tdraw.o main.o\
cdcntl.o -ly -ll -o saml
dxenq4.o : dxenq4.c
        cc -c dxenq4.c
lex.yy.c : exllex.l
        lex exllex.l
lex.yy.o : lex.yy.c
        cc -c lex.yy.c
y.tab.c : exlyacc.y
        yacc  exlyacc.y
y.tab.o : y.tab.c
        cc -c y.tab.c
tdraw.o : tdraw.c
        cc -c tdraw.c
main.o : main.c extern.h
        cc -c main.c
cdcntl.o : cdcntl.c
        cc -c cdcntl.c
```

APPENDIX B

CODES OF FUNCTION "odenq4"

```
odenq4(dx1, dx2, dx3, dx4, n)
int dx1, dx2, dx3, dx4, n;
{
   extern int plotqq[], pridx[], ptx;

   plotq[ptx++] = pridx[n] + dx1;
   plotq[ptx++] = pridx[n] + dx2;
   plotq[ptx++] = pridx[n] + dx3;
   plotq[ptx++] = pridx[n] + dx4;
}
```

APPENDIX C

CODES OF FUNCTION "cdcntl"

```
cdcntl(cx, cy)
int cx, cy;
{
        extern int chco[], rvco[];
        extern int cxr, cgr, ptx;
        chco[cxr++] = ptx-1;
        rvco[cgr++] = cx;
        rvco[cgr++] = cy;
}
```

APPENDIX D

CODES OF FUNCTION "tdraw"

```
#include <stdio.h>

tdraw()
{
  int ix=0, pl, xl, cr=0, cg=0, ctx, cty;
  extern int basex, basey, plotq[], rvco[], chco[];
  extern struct prim {
            int prix;
            int priy;
         } pmtv[];

  pl = chco[cr++];
  ctx = rvco[cg++];
  cty = rvco[cg++];
  while(plotq[ix]!=-1)  {
    xl = plotq[ix];
    basex = basex+pmtv[xl].prix;
    basey = basey+pmtv[xl].priy;
     if(pmtv[xl].prix >= 0) {
      if(pmtv[xl].priy >= 0)
       printf("v[+%d,+%d]",pmtv[xl].prix,pmtv[xl].priy);
      else
       printf("v[+%d,%d]",pmtv[xl].prix,pmtv[xl].priy);
     }
     else  {
      if(pmtv[xl].priy >= 0)
       printf("v[%d,+%d]",pmtv[xl].prix,pmtv[xl].priy);
      else
       printf("v[%d,%d]",pmtv[xl].prix,pmtv[xl].priy);
     }
  while(ix==pl)  {
    basex=basex+ctx;
    basey=basey+cty;
    printf("p[%d,%d]",basex,basey);
    pl = chco[cr++];
    ctx = rvco[cg++];
    cty = rvco[cg++];
  }
  ix++;
 }
}
```

APPENDIX E

CODES OF FUNCTION "main"

```
#include <stdio.h>
#include "extern.h"

main()
{
        printf(" 33Pp");
        printf("p[350, 10]");
        printf("s(E)");
        yyparse();
        plotq[ptx] = -1;
        tdraw();
        printf(" 33\");
}
```

APPENDIX F

LEX SOURCE FOR STATIC APPROACH

```
%%
a       return(301);
b       return(302);
c       return(303);
d       return(304);
e       return(305);
f       return(306);
g       return(307);
h       return(308);
i       return(309);
j       return(310);
k       return(311);
l       return(312);
```

VITA

Jia-Pyng Hwang

Candidate for the Degree of

Master of Science

Thesis: USE OF THE COMPILER-WRITING TOOLS LEX AND YACC TO CONSTRUCT 3-D OBJECTS

Major Field: Computing and Information Science

Biographical:

Personal Data: Born in Taiwan, R.O.C., November 10, 1956, The son of Chung-Yen and Ying-Huei Hwang.

Education: Graduated from Kaohsiung High School, Taiwan, R.O.C., in May, 1975; received Bachelor of Science degree in Mineral & Petroleum Engineering from National Cheng Kung University, Taiwan, R.O.C., in May, 1979; completed requirements for the Master of Science degree at Oklahoma State University in December, 1985.

Professional Experience: Mechanical engineer, Taiwan Power Company, Taiwan, R.O.C., from September, 1981 to July, 1983.