TABLE OF CONTENTS

FIGURE LIST

CHAPTER 1
INTRODUCTION

While much of computer science textbooks and classroom lectures are filled with diagrams, and much of our design activity as programmers takes place on whiteboards, we write our programs as text. With the recent proliferation of graphic user interfaces, technology exists that would allow us to draw rather than write programs. This dissertation addresses the question of whether we should be diagramming rather than coding, as well as the broader question of how the visual can be used in the process of programming.

1.1. MAJOR CONTRIBUTIONS

This thesis contributes to the field in four areas. The first area is in the analysis of diagrams. The use of diagrams in the field of computer science is thoroughly surveyed for the first time, and some underlying principles identified. The visual conventions of Adjoinment, Linking, and Enclosure are defined and illustrated.

The second area is in the creation of Visual Programming Languages. Three languages are developed - a simple programming language that encompasses shell commands, *awk* and SASL, a visual version of APL, and a visual front end for Mathematica. The visual version of APL is notable in that it presents both a program and instances of data undergoing transformation as part of one unified diagram.

The third area is the creation of system design conventions. Building on the work of R. J. A. Buhr, new visual systems designing conventions are created to handle the intricacies of facilities in the Ada 9X language. Asynchronous transfers of control, requeueing, and generic formal parameters are addressed. The asynchronous transfer of control convention is suitable for CASE representations of the language construct, and can be easily animated.

The fourth area is in the analysis of the effectiveness of graphic representations of computer programs. Some existing software metrics are modified for use in analyzing diagrams, and two new metrics are proposed: graphic token count and diagram class complexity. A graphic design measure, data density, is transformed into a computer science measure, token density. Using these metrics, graphic representations are compared to each other and to textual representations. From this, a strong set of conclusions are drawn about the relative strengths of graphic and textual representation.

1.2. AN OVERVIEW OF THE DISSERTATION

In Chapter 2, the current state of the field is surveyed. The use of diagrams in computer science is looked at first. In particular, the representation of recursion is highlighted. Visual language formalisms are discussed, including Pratt's H-graphs. Program visualization systems, programming in the large, and visual programming languages are all surveyed.

In Chapter 3, a simple visual language is developed, based on a simple two-dimensional design making use of simplicity of functional languages. A visual typing system is also suggested. The

first version of the language runs on top of the Unix shell, performing file manipulations. This language is then modified to run over AWK, and then extended to generate SASL.

A visual version of APL is created in chapter 4. By extending the simple language of chapter 2 to include intermediate output after the invocation of each function, it becomes possible to create easily understandable APL code through combining boxes and running sample data through the visual program.

In Chapter 5, a visual version of a subset of Mathematica is created, based on a generic template editor. This chapter discusses a general approach to the creation of visual languages using essentially a language design tool. Using this tool it becomes possible to easily create a front end for Mathematica by drawing a graph of the program. It is also possible to interactively create data structures that can serve as input to the Mathematica program.

System design notation is discussed in chapter 6. In particular, the two conventions created by Buhr, Buhr diagrams and MachineCharts, are reviewed, and extensions proposed to MachineCharts. Conventions for handling asynchronous control, requeueing, and generic parameters are created.

In chapter 7, textual metrics are examined and visual counterparts proposed. A new measure, Diagram Class Complexity, is created to measure the complexity of families of diagrams. Another metric, Graphic Token Complexity, is created and used together with Textual Token Complexity to analyze the representation of basic diagram types such as trees and graphs. Then the metrics are applied to many of the systems surveyed in chapter 2 and developed in subsequent chapters. From the results of this chapter, diagram cognition is discussed.

In chapter 8, a short recapitulation is performed, followed by strong conclusions about the role of the visual in programming.