A Method for the Specification and Parsing of Visual Languages

Eric J. Golin
Ph.D Dissertation

Brown University
Dept. of Computer Science
Providence, Rhode Island 02912

Technical Report No. CS-90-19
May 1991

A Method for the Specification and Parsing of Visual Languages

by
Eric J. Golin
Sc. B., Brown University., 1981
Sc. M., Brown University, 1985

Thesis
Submitted in partial fulfillment of the requirements for the
Degree of Doctor of Philosophy in the Department of Computer Science
at Brown University.

May, 1991

To my parents, for teaching me the love of learning.
And to Marion, for teaching me to love.

Abstract

Visual programming languages use pictures formed from graphical elements as programs rather than strings of text. A visual program is a diagram specifying a computation. Examples of visual programming languages include many types of diagrams traditionally used within computer science, such as data flow diagrams, finite state diagrams and flowcharts; as well as graphical methodologies developed for software engineering; and new languages such as StateCharts developed to exploit the visual paradigm.

One problem that has hampered research in visual programming languages is the lack of a syntax definition mechanism. The specification of syntax and parsing of programs is well understood for textual programming languages. Both formal methods for syntax specification (e.g. context-free grammars) and practical tools (such as *yacc*) have proven valuable to language developers. Visual language syntax is more complicated because of the two-dimensional nature of the languages. Previous visual programming environments largely used ad-hoc specifications and special purpose structure editors to handle visual language syntax.

The *attributed multiset grammar* is a formal model for generating languages which are collections of objects. *Picture layout grammars* are a specification mechanism for visual language syntax based on attributed multiset grammars. Using picture layout grammars, a language designer can define both the syntactic structure and the two-dimensional layout of a visual language. A *spatial parser* is an algorithm to recover the syntactic structure of a visual program from an unstructured picture representation. This dissertation develops the attributed multiset grammar model and gives a spatial parsing algorithm for picture layout grammars. It also describes the GREEN environment, which combines an object-oriented graphics editor with the spatial parser to form a grammar-based visual programming environment.

Acknowledgements

While this dissertation is an individual effort, its magnitude and duration ensure that many others have assisted in its production. While it is difficult to recognize everyone who aided me in the last seven years, I'll try to do so. Those who are omitted should know my gratitude is no less.

There is much for which I have to thank my advisor, Steve Reiss. He introduced me to the idea of visual programming, and slowly led me to view "programming" as something more than producing lines of code. Steve was the first to suggest to me that graphical parsing might be an interesting problem. He has patiently provided support and guidance, through each step of my graduate career. Most of all, I thank him for his creativity, which has inspired me and shown me what it means to do research in software systems.

I would also like to thank the two other members of my committee, Ken Zadeck and Roberto Tamassia. Ken has played an important role as skeptic, encouraging me to examine my research from a viewpoint outside the world of visual languages. Roberto has helped to provide insight into the problems associated with the application of the geometric constraints (e.g., testing adjacency). Both have provided a thorough, close reading of this dissertation and I know it is better for their efforts.

In addition, I would like to thank my unofficial, fourth reader, Robert Rubin for all his assistance. Rob's contributions have been too numerous to detail, and on many levels, ranging from intellectual to personal. His insightful discussions helped me to develop the ideas in this thesis, and his comments on an early draft helped to clarify the presentation. Rob has served as fellow student, officemate, carpool partner, colleague, collaborator, and primarily as friend, and I thank him for all his help.

I would like to thank Peter Wegner for reviewing and providing assistance with my thesis proposal. Thanks also to Dr. William Chao of GE for comments on several chapters of my thesis. Jim Walker performed an informal analysis of the spatial parser which helped to clarify problems with the implementation and was greatly appreciated. I would also like to thank Joe Pato for many helpful discussions throughout my graduate career, and for his friendship. Thanks to Mark Post for sage advice on what not to do as a graduate student (if only I'd listened more often!), and for continually asking me if I was done yet.

Although aggregate thank you's often seem perfunctory, I'd like to express my appreciation to three groups: First, to the faculty with which I've had the pleasure to work - they have created a superb atmosphere for graduate study at Brown. Secondly,

to the staff: secretarial, administrative and technical - they all perform admirably to resolve problems, even for the lowly graduate student. Lastly, I'd like to thank my fellow graduate students, and particularly those with whom I've had the pleasure of sharing an office: Brian Dalio, Sunil Agarwal, Dhun Wadia, Sreedhar Annamalai, Dave Johnson, John Stasko and Andrea Skarra.

IBM provided generous support for this research in the form of a Graduate Fellowship. I would also like to thank GTE Laboratories and Dr. Gerry Jones for providing me with an office, workstation and salary for the summer while implementing the spatial parser.

Finally, I want to thank my wife Marion. She has been there through good times and bad, patiently providing the support I needed to persevere. My love and appreciation are with her always.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

> *Graphic representation constitutes one of the basic sign-systems conceived by the human mind for the purposes of storing, understanding, and communicating essential information. As a "language" for the eye, graphics benefits from the ubiquitous properties of visual perception. As a monosemic system, it forms the rational part of the world of images.*
>
> ...
>
> *Graphics owes its special significance to its double function as a storage mechanism and a research instrument. A rational and efficient tool when the properties of visual perception are competently utilized, graphics is one of the major "languages" applicable to information processing. Electronic displays, such as the cathod ray tube, open up an unlimited future to graphics.* [5].
>
> —Jacques Bertin, *Semiology of Graphics*, 1967.

Programming is both one of the most fundamental and one of the most difficult activities associated with computing. Its difficulty has led to the development of many different approaches and languages for programming. Traditional programming languages such as Fortran and Pascal provide programming with different syntactic constructs within a similar procedural model. New paradigms have been introduced, such as object-oriented programming, functional programming and logic programming, which change the underlying model of computation. Another approach is to change the medium used to express computations, from textual elements to visual ones. *Visual programming* is a mode of programming where graphics rather than text is used as a medium for programming.

Visual programming is based on the idea that pictures can be both more expressive and easier to understand than text. Visual programming is an approach to programming, the idea that pictures should be used to express computations. A concrete expression of a computation requires a visual programming language. A visual programming language is a programming language whose components are graphical. Programs in a

visual programming language are formed by combining the graphical elements into a picture.

This thesis develops a foundation for visual programming languages. It presents a formal model of visual language syntax along with an algorithm for performing syntax analysis of pictures. A formal model of visual language syntax is essential for discussing visual languages. It allows us to say what constitutes a picture, and what pictures comprise valid programs in a visual programming language. Within the formal model, a convenient grammar mechanism for defining visual languages is provided. These grammars are the input to the generalized parsing algorithm.

Most of the research in visual programming has focused on the development of specific visual languages. Part of the reason for this focus is the lack of formal models for specifying visual language syntax and tools for utilizing those specifications. This thesis provides a basis for research in visual languages, as well as tools to aid the visual language researcher.

To understand the motivation for this thesis, one can examine the state of affairs in traditional textual programming languages. Context-free grammars provide a formal model for the syntax of textual programming languages. This has allowed language designers to specify and discuss programming languages independent of any particular implementation. In addition, parser generators such as *yacc* [40] allow the easy creation of parsers for new languages. Complete programming environments have been generated from grammar based language specifications [65, 71].

This research may be seen as an attempt to put visual languages on a similar footing as textual languages. Only when visual languages have both a solid theoretical foundation and an array of useful tools will visual programming languages be on a par with textual ones.

This chapter gives an overview of the research. First, the discussion considers what a visual language is and why they are useful. The next section characterizes what the "problem" addressed by this thesis is and why it is important. Finally the solution and its significance are outlined. The chapter closes with a description of where the remaining chapters will take us.

1.1 Visual Programming Languages

The term *visual language* has been used in many different ways. Several distinct types of languages have been described as visual languages: languages manipulating visual information; languages for supporting visual interactions; and languages for programming with visual expressions [86]. *Visualization* systems are also often confused with visual programming. They create visual representations of structures such as data, programs or designs. The connection between these areas is that they all concern both graphics and computer programming, but they differ in three directions:

- What kind of pictures are used.
- What kinds of things are represented by the pictures.

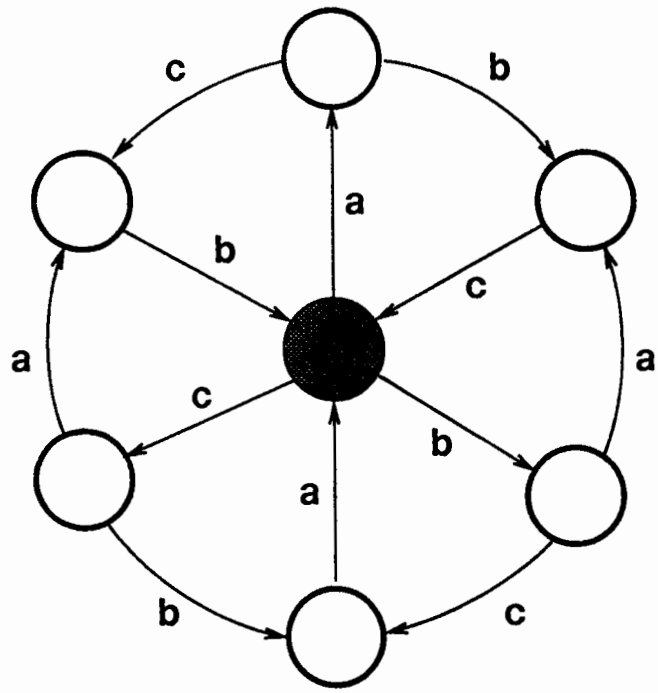


Figure 1.1: A Finite State Diagram

- How the pictures are used (i.e., are they created, edited or displayed?)

Our concern in this dissertation is largely with languages for programming with visual expressions. These are called *visual programming languages*. The term *visual* is used in the same sense as in the phrase "visual aid", meaning something graphical as distinguished from mere text. A visual programming language is a programming language where a program is formed from graphical elements, rather than textual ones.

For example, consider the picture in Figure 1.1. This picture is a finite state diagram. It is visual, in that the picture is two dimensional and composed of nontextual elements such as circles and arrows. Of course, the picture also contains some text, as text strings can be considered a type of graphical element. Thus, textual syntax can be considered a special type of graphical syntax.

In addition to being graphical, visual programming languages are designed for programming. A program is a description of a computation or action. A visual program is a picture which describes a computation. For example, the picture in Figure 1.1 can be interpreted as a finite state diagram (by considering the circles to be states and the labelled arrows to be transitions, with the marked state both the initial and final state. It describes a function mapping from Σ^* to $\{0,1\}$ (i.e., a function that accepts some strings and rejects others), where $\Sigma = \{a, b, c\}$.

Just as there is more than one textual programming language, visual programming languages take on many different forms. Visual programming languages can be characterized along three axes:

- The *problem domain* addressed by the language reflects both the type of problems that the language is suitable for and the potential audience for the language. Visual programming languages vary from the very narrow, such as the Miro language [53] for specifying access rights, to general purpose programming languages such as Prograph [19]. Visual languages have been directed towards non-programmers [21], novice programmers [10, 33] and experienced programmers [35].

- The *computational model* underlying the language determines the execution semantics. Visual programming languages are not limited to any particular semantics. In practice, a wide range of computational models have been used for visual programming languages. Examples include the traditional imperative model [33], object-oriented programming [19], data flow [44], logic [96], and programming by demonstration [76].

- The *visual syntax* of the language describes what pictures form programs in the language. The types of pictures used in a visual language can vary greatly.

Our focus is on the syntactic aspects of visual languages and my approach to syntax is independent of the problem domain or computational model. Even restricting our attention to syntax, a great variety can be found in visual languages. The differences in visual syntax are largely of function of two aspects of the language:

1. *visual symbols*: The visual symbols used to form programs dictate much of a language's syntax. These symbols correspond to such elements of textual language as keywords, identifiers and constants. Some types of visual symbols are rectangles, circles, lines, text strings, icons and images.

2. *arrangements*: The remainder of the syntax is determined by the way in which the visual symbols can be arranged to form a program.

Shu [86] has classified visual programming languages into three groups according to the type of pictures they contain. *Forms-based languages* use two dimensional arrangements of text as programs. For example, FORMAL [85] is a data management system based on filling in forms. A stylized form heading consisting of hierarchically structured text is used to represent a data structure. Data manipulation is programmed as a transformation between forms.

Icons are atomic visual symbols which relate to objects or processes. *Iconic languages* use two dimensional arrangements of icons for programming. For example, the Pict system [33] uses flowcharts constructed from icons to create programs for simple, numeric computations. Icons are used to represent variables, program operations and

control structures. Iconic languages generally use a large number of icons which are combined in a simple fashion.

Diagram languages use existing graphical design methodologies to create executable visual programs. Flowcharts, data-flow diagrams, petri nets and finite state automata have all been implemented as visual programming languages. Diagram languages may be further broken down into two groups: nested box languages and box and arrow languages. In contrast to iconic languages, diagram languages generally have a small number of visual symbols which are combined in a complex fashion.

1.1.1 The Benefits of Visual Programming

One question to be addressed is why should a visual programming be used? Textual programming language technology is quite advanced and a large number of textual programming languages already exist. What advantage is to be gained by extending these capabilities to visual languages? It is apparent from the level of interest in visual programming languages (as evidenced by the rising attendance at the IEEE Visual Languages Workshop and the start of a new journal devoted to visual languages [14]), that some advantage is perceived for a visual approach to programming.

The benefits of visual programming must be discussed in the context of the intended audience. This audience can be grouped into three classes, according to the relationship to programming: end-users; novice programmers; and experienced programmers. Visual programming will enhance the ability of all three groups to exploit the power of computers, although in different ways. What gives each of these diverse groups and advantage is the relationship of visual programming to mental models. Norman has observed that

> Conceptual models are devised as tools for the understanding or teaching of physical systems. Mental models are what people really have in their heads and what guides their use of things. Ideally, there ought to be a direct and simple relationship between the conceptual and the mental model.
>
> ... In the ideal world, when a system is constructed, the design will be based around a conceptual model. This conceptual model should govern the entire human interface with the system, so that the image of that system seen by the user is consistent, cohesive, and intelligible. [60]

Visual programming languages allow a programmer to use a conceptual model that is close to her own mental model. Reiss calls this approach *conceptual programming* [66]. Conceptual programming tailors the solution to the thought process of the user, rather than forcing her to think in terms of the programming language. Because the solution is often expressed using diagrams, a visual programming environment is well suited to conceptual programming. Each class of users will form mental models of the solution, but they differ in relating those to conceptual models. By providing a closer relationship between the conceived mental model and the implemented conceptual model, visual programming allows the user to make more effective use of the computer. The benefits of a visual approach to programming for each class of users is briefly examined next.

End-User Programming

End-users are people who either do not program at all (nonprogrammers) or people for whom programming is not a typical task (casual programmers). For some applications, visual languages are more easily manipulated that textual ones by people without special training in programming [72], and allow such users more control over their computing.

Computers are very flexible in adapting to specific problems, but using this flexibility often requires programming to customize an application. Visual programming enhances the ability of end-users to do such tailoring without the aid of a computer expert. Visual languages and interfaces provide a model of the computing process which is simple to understand and manipulate.

One example of such end-user programming can bee seen in the HI-VISUAL system [97]. HI-VISUAL is an iconic language for specifying image transformations. A program consists of a graph of icons, where the icons represent data, primitive operations and subprograms. A program is constructed by selecting icons from a menu and connecting them to form a graph. The connections represent the flow of data.

HI-VISUAL is used to create image processing programs. The icons correspond to entities within the mental model of the image scientist, such as the TV camera, basic image processing functions (e.g. thresholding, edge detection) and image displays. A complicated image processing procedure can easily be built up from the basic components. Creating a custom application such as this would otherwise require extensive programming expertise, even with the use of a library providing functions equivalent to the icons.

Novice/Learning Programmers

A second area of computing where visual languages realize considerable benefits is in use by novice or learning programmers. One way pictures are useful in learning to program is by aiding memory. Visual aspects such as shape, color and texture help in recognition of program components. Pictures also aid comprehension because they better reflect the structure of the program.

Bonar and Liffick [10] developed a visual programming language for instructional use. The language was based on the concept of *programming plans*. Plans are atomic elements, and each plan has a single icon. The focus is on the interconnection of plans. Control flow is expressed by end-to-end connection of icons. Data flow is expressed by embedding a *value icon* into a hole in another icon. The icons were shaped such that they could only be combined in syntactically valid forms. This use of shape helped the novice to learn programming both by guiding the construction of a program and reflecting that structure in an accessible manner.

The PICT system [33] is an iconic programming system oriented towards novice programmers. The language was loosely based on Pascal. The user interacted with a structure editor to choose icons and then connect them with paths. PICT used color rather than names to represent variables. While this limited the number of variables, it also increased perception of which variables were used.

Expert/Experienced Programmers

Finally, visual programming languages can increase the productivity of experienced programmers by providing new, powerful ways of expressing computations. Pictures can be more expressive than words, particularly when describing things with a two-dimensional nature, such as topological information. Research has suggested that graphical representations of algorithms are sometimes easier to comprehend than textual representations [79]. Pictures are also very adaptable to new paradigms, and visual programming languages have been for many sophisticated computational models. Three examples of this are the PiP, ThinkPad and Prograph systems.

The Programming In Pictures system [64] supports a a visual programming language based on the FP functional programming language. The system uses two-dimensional graphics to define functions. The language is designed as a *direct manipulation* interface – no names are used to identify entities. Programs are created using a series of specialized editors. A function is defined by manipulating the constituents of the input and output data structures. Functions may then be combined using a number of built-in functional forms. Each functional form has a characteristic frame which controls its display. PIP demonstrates the usefulness of a visual language for functional programming.

ThinkPad [76] is an environment for graphical programming by demonstration, with a language based on logic programming. The user defines graphical representations of her data structures using the data editor. Functions are then defined as mappings from inputs to outputs by manipulating the input and output data structures. Multiple cases with different conditions assigned can be defined for a function. The function definitions are mapped to Prolog clauses for execution.

Prograph [19] is a pictorial programming environment based on an object-oriented dataflow language. The environment provides the capabilities to graphically define classes and methods. Methods are programmed visually by constructing a picture that depicts the dataflow implementation of the method. Execution of Prograph programs is provided by an interpreter and interactive debugging tool.

Visual programming languages are also well suited for use in concurrent programming systems. Because pictures are two dimensional, they express process interconnections in a natural manner. Several languages have been developed to exploit this feature. [CITE Pigsty, etc. here]

Specialized visual languages can be used to express particular aspects of a system. The Miro project [53] is designing a visual language for specifying properties of large software systems. They have developed a visual notation for describing security properties based on the access-rights matrix model. The visual language uses labelled boxes to represent both the universe of users and files to be accessed. Labelled arrows indicate access rights. The visual language allows precise, convenient security definitions and is an improvement over previous textual approaches.

Lastly, pictures are used extensively within software engineering methodologies and tools. Design methodologies such as SADT [74] use graphical representations

as "blueprints" of software designs. Numerous commercial applications for Computer-Aided Software Engineering are available which visual programming of design or documentation. One example is the *Software through Pictures* modular tool set from Interactive Development Environments [92]. This system provides editors for constructing several types of diagrams, as well as programs to check diagrams for design-rules of supported methodologies.

1.2 Specifying Visual Syntax

1.2.1 The Problem

Languages are sets: textual languages are sets of strings; graph languages are sets of graphs; and visual languages are sets of pictures. The purpose of a language syntax specification is to define the set constituting the language. The problem addressed by this dissertation is to develop a method for specifying the syntax of visual programming languages. The requirements on the solution to this problem are that:

- The specification mechanism must be applicable to a wide range of visual programming languages, and easily extensible to new languages.
- A language specification be a natural expression of the syntax of a language and easily understood.
- The specification of a language's syntax must provide a feasible basis for efficient syntax analysis.
- A language specification should provide a foundation for the processing of visual programming languages, such as the compilation of visual programs and the construction of visual programming environments.

Syntactic analysis of visual programs is central to my approach to visual programming. Parsers have proven to be very valuable in the development of textual programming languages. Parsers separate the manipulation of programs from their interpretation, allowing the visual programmer to work in terms of the concrete syntax of the visual language. Thus, the second problem addressed is the development of a method for parsing visual programs based on the specification of the visual language syntax.

My approach to syntax specification is grammar-based. A grammar is a finite set of rules which specify the syntax of a language. The nature of a grammar depends on structure of the elements of the language. String grammars are used to specify the syntax of textual languages; graph languages are specified by graph grammars, and visual languages are specified with picture grammars.

A grammar approach to syntax specification has several advantages:

- Grammars provide a formal definition of the language syntax, which allows reasoning about language elements (e.g. determining membership in the language; proving correctness of parsing algorithms).

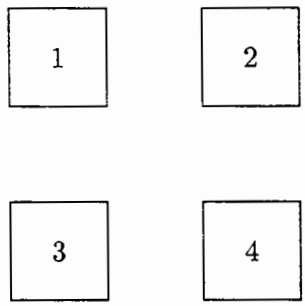


Figure 1.2: A Two-dimensional Collection of Objects

- Grammars are general tools for defining syntax and can be extended to new languages by writing new rules.
- The grammar rules provide a structure to the elements of the language.
- A grammar provides a structured mechanism for defining the translation from abstract to concrete syntax.
- A grammar specification can form the basis for a parsing algorithm.

1.2.2 Why is Visual Language Syntax Difficult?

Formal languages whose elements are strings have been widely studied. Context-free grammars are used as a basis for the syntax of textual programming languages. Parsing of context-free languages is well understood, and is widely used within traditional compilers. So the question arises, why is visual language syntax difficult? The answer to this question is that visual languages differ from context-free languages in one key aspect - the elements of a visual language are two dimensional collections, rather than one dimensional sequences. This difference is reflected in three ways:

- The symbols in a visual program are not represented by a linear ordering.
- A wide variety of relationships between symbols is possible, rather than simple one dimensional adjacency.
- The underlying structure of a visual program is a directed graph, rather than a tree.

No Linear Ordering

Textual programs are strings. A string is merely a sequence of symbols. This sequence reflects a linear ordering of the symbols comprising a program. This ordering is inherent in the program and is reflected in any representation of it. Traditional parsing

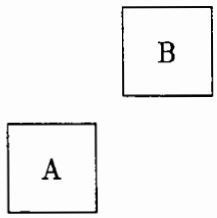


Figure 1.3: An Indeterminate Composition

algorithms make use of this fact.

A picture, on the other hand, is a two dimensional object. A representation of the picture should embody all the important relationships within the picture. A linear ordering of the components of a picture would not preserve all the adjacency relationships. For example, consider the picture formed from four boxes shown in Figure 1.2. The structure of this picture could have either the boxes labelled 1 and 2 or the boxes labelled 1 and 3 related, depending on the definition of the language. Thus the elements of the picture cannot be simply linearized, while preserving all the relationships between components.

Many Compositions

In a textual language, concatenation is the only composition operation used. The relationship between elements of a sequence of symbols is fixed. In a visual language, many types of composition operators are possible. Several different operators could apply to the same picture. For example, in Figure 1.3, is the box labelled B *above* the box labelled A, *above and right* of A, or just *right* of A. The interpretation of this relationship is dependent on the syntax of the language. The representation of pictures must sufficiently general to express all these possibilities.

Underlying Graph Structure

The structure underlying a textual program is a tree. Like strings, trees are essentially linear objects. Visual programs are two dimensional, so they often do not have a natural representation as a tree. For example, consider the finite state diagram shown in Figure 1.1. This visual program is two dimensional - it can't be properly drawn in one dimension.

Furthermore, a tree structure will not correctly express the relationships between the components of the picture. An arrow in the picture is clearly related to the circles at both ends, since that relationship is significant to the meaning of the picture. The cycles present in the FSA graph prohibit a tree-based representation of these relationships, however. Representing a cycle requires some object (circle or arrow) to appear twice in the same tree. Thus, the two dimensional nature of the finite state diagram reflects an underlying structure that is graph-based rather than tree-based.

1.2.3 Overview of the Solution

To model the syntax of visual languages, I have developed a new kind of grammar, called the *attributed multiset grammar.* Multiset grammars are similar to context-free grammars, except that the right hand side of a production is considered to be an unordered collection of symbols, rather than a string. The language generated by a multiset grammar is a set of multisets.[1] An attributed multiset grammar is a multiset grammar which has been augmented with *parsing attributes.*

Attributed multiset grammars are similar to traditional attribute grammars [45], but differ in two key respects: First, because they generate multisets, attributed multiset grammars remove the notion of ordering implicit in a string grammar. Instead, an attributed multiset grammar represents the logical structure of a program. Another difference is that the attributes in an attributed multiset grammar are an integral part of the parsing of an input multiset. The attributes in a grammar are used to express ordering relations within the structure of a program.

The mechanism developed for specifying visual syntax is a specific form of attributed multiset grammar called a *picture layout grammar (PLG).* A picture layout grammar is used to specify both the structure and the two dimensional syntax of a visual programming language. The symbols in a picture layout grammar are graphical elements, and the productions specify how those elements are combined to form pictures. A picture is represented as an attributed multiset of graphical symbols. The attributes in a picture layout grammar represent the geometric information about the elements of a picture.

I have developed an algorithm for parsing visual programs. The parsing algorithm takes as input a picture and a grammar and constructs a directed acyclic graph expressing the derivation of the input picture. Syntax analysis is done in a bottom-up fashion, with all possible parses computed. The parse graph can serve as the basis for processing of a visual program.

Finally, I have implemented a visual programming environment, called Green, based on picture layout grammars and the parsing algorithm. Green extends the attribute mechanism of picture layout grammars to allow the specification of language semantics similar to those found in attribute grammar based programming environments [71].

1.3 Contributions

This thesis solves the two problems stated earlier – development of methods for specification and parsing of visual languages. These developments are of significance from both a theoretical and practical point of view. They advance the understanding of the structure of visual syntax and also provide useful tools for use in research and development of visual programming languages.

The primary contributions of this dissertation are:

- A new formal model, the attributed multiset grammar, for describing computations of collections of objects. Attributed multiset grammars provide a basis for

[1] A *multiset* is a set which may have repeated elements, i.e. an unordered collection.

the syntax of visual programming languages.

- A concrete mechanism for the specification of the syntax of visual programming languages. Picture Layout Grammars present a method of easily and naturally describing visual syntax.
- A general algorithm for the parsing of visual programs. My algorithm is capable of parsing programs in any visual language described by a well-formed picture layout grammar.
- A design for a visual programming system based on objects, picture layout grammars and spatial parsing. This architecture allows the quick prototyping of visual languages using picture layout grammars, and programming in a visual language using the parser to analyze and process visual programs.

1.4 Thesis Organization

Chapter 2 reviews other work related to visual language syntax, concentrating on other research in visual programming languages that has attempted to provide syntax specifications. Research from syntactic pattern recognition, on using grammars to specify pictures, is also discussed.

Chapter 3 discusses visual language syntax; examines the various alternatives in visual syntax; and develops a conceptual framework for pictures and visual languages. It outlines how grammars may be used to specify visual syntax and illustrates this with an example.

Chapters 4 and 5 develop the method for specifying visual language syntax. The formal definition of the model, the attributed multiset grammar is given in Chapter 4. Chapter 5 elaborates on this model, describing how picture layout grammars are used to specify visual syntax.

Chapter 6 gives the algorithm for parsing visual programs. The algorithm is presented, along with an analysis of the complexity and a proof that the parsing algorithm operates correctly.

Chapter 7 describes the architecture and implementation of the GREEN visual programming environment. Chapter 8 presents conclusions, evaluating my method for specifying visual languages and the specific implementation. Finally, it discusses how this approach could be extended and gives directions for future research utilizing the model of visual syntax.

Finally, Appendix A describes the exact format used to write a picture layout grammer. Appendix B gives a detailed discussion of the GREEN Editor interface. Appendix C gives three sample picture layout grammars defining simple visual programming languages.

Chapter 2

Previous Work

This chapter examines previous research done on the specification of visual syntax. A number of other researchers have looked at the question of specifying and parsing visual languages. Several researchers have addressed the question of producing a language independent visual programming environment. The previous work related to these issumes are discussed.

There is also a large body of research into the problem of using grammars to specify and recognize patterns. This area is called syntactic (or structural) pattern recognition [28]. Pictures are a typical application for syntactic pattern recognition. This area of research has much in common with the objectives of this dissertation, although they are not identical. The research most closely related to my goals is examined.

In addition to work in these two areas, there is a broad range of previous work that deals pictures for purposes of generating images. These include systems oriented towards computer graphics such as PHIGS [39]; page description languages such as PostScript [38]; program visualizations systems [58]; constraint systems [11]; and languages for specifying diagrams within documents, such as PIC [42] and Ideal [93]. These areas of research, are related in that they contain models for images. They differ from this thesis in several respects: the types of images modelled; the structure of the models (e.g., procedural representations); and most importantly, in that the goal is the synthesis of a picture. Because they are not directly relevant, these areas are not considered in more depth.

The next section covers the related research in visual programming languages. Following that is a discussion of the relevant previous work from syntactic pattern recognition. Finally, I briefly relate my new approach to the previous work.

2.1 Visual Programming Environments

A *visual programming environment* is a software tool or collection of tools which provides support for programming in a visual language. The role of the visual programming environment is analogous to the role filled by a traditional, text-based programming

support environment [65]. Visual programming environments must support two primary tasks: the manipulation of programs (i.e., the ability to create and edit visual programs); and the processing of programs by analyzing and executing them.

The approach to building a visual programming environment presented here is based on spatial parsing. A general visual programming environment is customized for a particular visual language using a grammar specification of the language. This approach parallels developments in textual programming support environments, where specifications have been used to generate language specific environments [65, 71].

Typically, visual programming environments are not language independent. Rather, most of the research in visual languages has concentrated on the development of new visual languages along with dedicated environments for those languages. A single language environment such as this generally contains a special purpose interface for creating and editing visual programs. A representation of the underlying structure is constructed along with the picture and used to drive any processing.

ThinkPad [76] is typical in this respect. The ThinkPad environment provided a visual interface for creating ThinkPad programs. This interface contained distinct editors for the various components of a ThinkPad program (e.g., data structures, conditions and operations). These editors were all restricted to editing operations based on the underlying representation of the programs. The editor (and environment) was completely specific to the ThinkPad language. Other examples of systems using a dedicated, special purpose editor include Pegasys [57], PiP [64], PICT [33], I-Pigs [63], InterCONS [87], Fabrik [51] and Prograph [19]. These systems do not address the problems of syntax specification and parsing and are not considered further.

A number of other visual programming environments have been designed to be language independent and use syntax specifications. The Garden system [67] is a language independent environment for graphical programming, but is not based on parsing. The SIL/ICON Compiler [17] project and the work of Lakin [48] both develop language independent environments based on parsers. Show and Tell [44], G-LOTOS [9] and Ambit/G [18] all exhibit single language environments that use parsers. The remainder of this section examines each of these systems in more detail.

2.1.1 Garden

The Garden system [68, 67] is a language-independent environment that supports a wide variety of different graphical program views in a consistent manner. Garden is intended as an environment for prototyping visual languages and its design allows for the easy definition of a new language. Another objective of Garden is to permit programs in different visual languages to interact within a common framework.

Garden is implemented as an object-oriented environment. There is an underlying object system called HUMUS which provides the basic capabilities of the environment. An object-oriented database is used to store the objects. All components of the environment, including the language definitions, visual programs, and environment information are represented using objects.

Garden takes a three-tiered approach to defining a visual language. First the abstract structure of the language is specified by defining a set of object types corresponding to the components of the language. A concrete graphical syntax for the language is defined from these types. The semantics of the language is specified by defining evaluation routines for the object types. An editor for visual programs is provided which can be customized for a specific language.

The visual syntax of a language is specified using the GELO layout package [69]. A mapping is defined from the program objects into GELO graphical objects. GELO automatically creates a display from the graphical object structure. The display of a visual program is a two-step process. The first step maps from a structural representation of the program (as Garden objects) to a structural representation of the picture (as GELO objects). The second step then maps the structural representation of the picture into an actual display.

GELO provides four types of graphical objects:

1. Basic objects consist of a simple shape enclosing a text string. Figure 2.1a shows an example of a basic object.

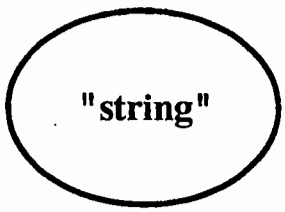
2. Tiled objects are used for hierarchical structures. A tiled object consists of a rectangle split into non-overlapping rectangular regions or tiles, as shown in Figure 2.1b. Each tile can contain another GELO object. The relationships between the tiles form a set of constraints on the contained objects, which are resolved by GELO during picture generation.

3. Layout objects generate displays of graph structures. Layout objects consist of a rectangular region into which nodes and edges are placed. The nodes are either basic or tiled objects. The edges can either be explicit arc objects or simple connections between nodes. The resulting graph is automatically laid out by GELO using built-in heuristics.

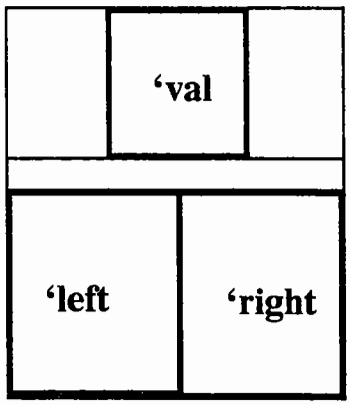
4. Arc objects are used to control the display of arcs within a layout. They can specify parameters such as the type and location of arrowheads and a text label to be place on the arc.

Figure 2.1c shows a sample Garden display. The display shows a Finite State Automaton program. The overall program is displayed as a tiled object with two fields. The top field contains the name of the FSA. The second field contains a layout object which is used to display the FSA. Within the layout object, states are translated to basic objects and displayed as ellipses enclosing the state name. Transitions are displayed as labelled arrows between the states.

The APPLE editor is used to define the mappings from the program objects to GELO objects. For each program object type, a set of stylized display examples are constructed. Each example consists of prototypical GELO objects which indicate how actual GELO objects are to be created from a program object of the given type. Conditions are used to select which of the examples are used for each program object.



a) GELO basic object



b) GELO tiled object

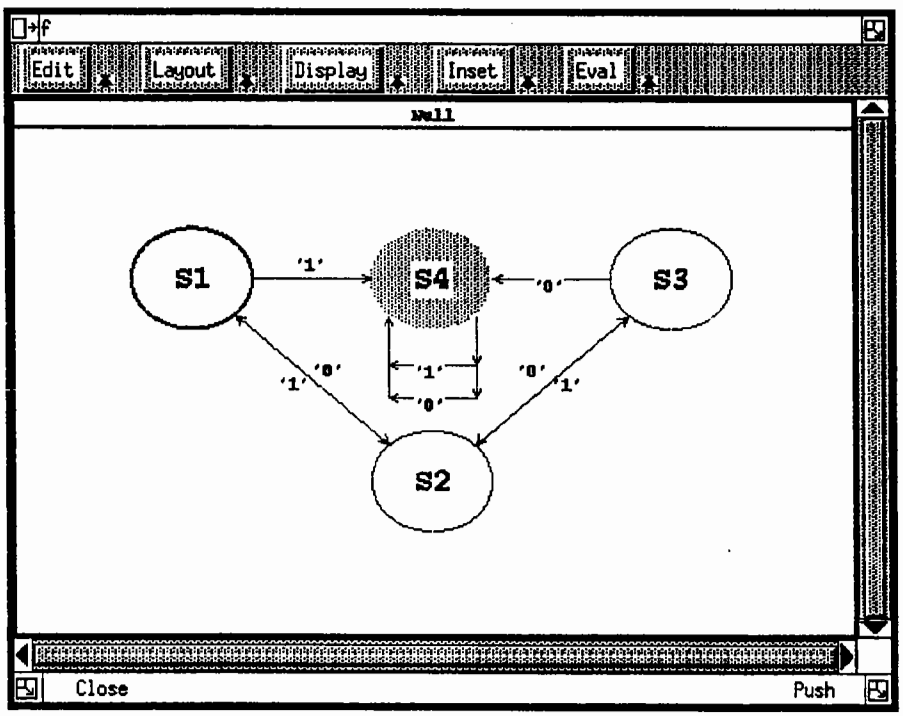


Figure 2.1: a) a GELO basic object; b) a tiled object; c) a GELO display

The approach to visual language syntax taken by Garden is directed towards generation of visual programs from abstract structures, rather than analysis of programs to recover structures. Garden does not use a grammar to define visual syntax, but its approach can be viewed in a grammar framework. The aggregate GELO objects (i.e., tiled and layout objects) correspond to composition operations. The Garden object types can be thought of as nonterminal symbols, and the display examples are like productions.

Although the Garden/GELO approach can be viewed as similar to a grammar, it differs in several key respects. First of all, since the Garden object types serve as "nonterminals", the structure of the picture must closely follow the abstract structure of the language. The translation process also forces the GELO objects into a hierarchical organization.

Another problem is that the compositions are limited to the tiling and layout aggregates. These compositions do not cover all the desirable picture structures. For example, two dimensional arrays cannot be described. Furthermore, fine tuning of a composition (e.g., relative positioning of objects within a layout) is limited to the builtin heuristics.

Garden attempts to simplify the definition of visual syntax by making the process as automatic as possible. It succeeds at making it exceptionally easy to define a visual syntax for a language, but at the loss of some flexibility in controlling the syntax. This can make it difficult to define the syntax of a language exactly as desired.

Another distinction between Garden's approach and mine is the method of creating visual programs. The language definitions used by Garden are not suitable for syntax analysis. The visual syntax is defined by three elements: the structure of the Garden objects; the mappings of those objects to GELO objects; and the rendering of the GELO objects into display entities. Reversing the translation process requires that all three of these elements be unified, and that this translation be invertible. One problem with inverting these translations is that the mappings can use arbitrary conditions for selection. Another problem is that the contents of a layout object can be specified as the closure of a set of objects, giving rise to cyclical references between "productions".

Instead of an editor/parser combination, programs are created in Garden using an interface to the PEAR structure editor. The editor uses GELO to display a picture of the user data structure (i.e. program). Editing requests are initiated by user actions, and are translated into requests to change values or to add or delete elements from a layout. The implementation of these changes in the underlying data structure is provided by the language definer in the form of methods for the paramount object type. GELO is used to update the display to reflect any changes in the program objects.

Annamalai [4] formalized Garden's translation process and attempted to make it invertible. Garden object definitions were mapped to grammar rules where the left hand side is a Garden type and the right hand side consists of the fields of that type. He expressed the translation to GELO objects as a distinguished synthesized attribute *picture*. The attribute grammar was inverted using an extension of Yellin's algorithm [95]. The resulting grammar translated from the GELO structure of a picture to the Garden

objects forming the abstract syntax. This approach required that the user explicitly specify the hierarchical structure of the picture. The interface provided by this approach was more of a modified structure editor than a general picture editor. A further problem was that the inverted grammar could be ambiguous and not amenable to analysis.

2.1.2 SIL

The SIL-ICON Compiler [17] developed at the University of Pittsburgh comes closest to my own research. The SIL-ICON compiler supports the specification, interpretation, prototyping and generation of icon-oriented systems. A formal specification of an icon system is used by an icon interpreter to understand and evaluate a visual sentence.

The design of the system is based on the concept of a *generalized icon*. A generalized icon has a dual representation

$$(X_m, X_i)$$

where X_m is the logical part (meaning) and X_i is the physical part (image). An icon system definition consists of four parts:

- G - the *Icon System*, a formal, syntactic specification of the icon system.
- ID - the *Icon Dictionary* a set of elementary icons. The elementary icons are divided into two disjoint sets, the process icons and the object icons.
- OD - the *Operator Dictionary*, a set of icon composition operators.
- ETAG - the *Extended Task Action Grammar*, a description of the tasks in the icon system.

An iconic sentence is a combination of predefined icons, forming a composite icon. An *icon world* is the set of all the composite icons that can be constructed from the elementary icons using the operators in OD. An *icon-oriented system* is a subset of an icon world that is meaningful for an application domain.

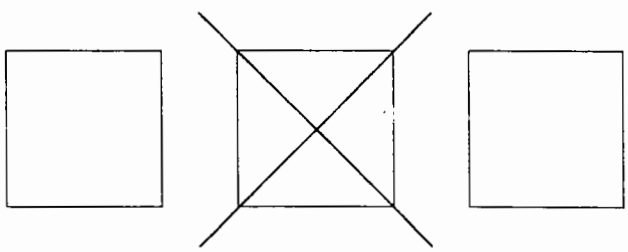
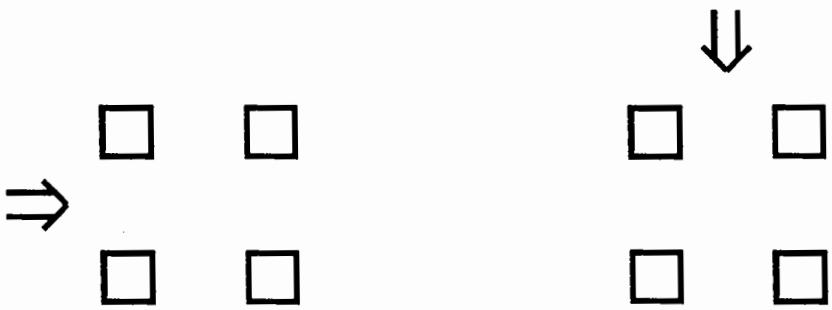


Figure 2.2: Iconic Sentence



PICTURE → right & ROWS
PICTURE → down ∧ COLS
ROWS → ROW ∧ ROW
COLS → COL & COL
ROW → box & box
COL → box ∧ box

Figure 2.3: Two Iconic Sentences and a Picture Grammar

The SIL-ICON system contains several tools: a tool for creating and manipulating icon system definitions; an editor for iconic sentences; an icon interpreter; and an icon-oriented system generator. An icon system definition is essentially the definition of a visual language. An iconic sentence corresponds to the notion of a visual program. The icon interpreter parses an iconic sentence and evaluates it, producing a complex icon structure in the form of a conceptual graph. The icon-oriented system generator builds an executable version of a set of complex icons.

The physical part of an icon (i.e., the concrete syntax of a visual program) is specified using a picture grammar. A picture grammar is a context free grammar where the terminal symbols include both primitive picture elements and spatial image operators. The operators describe compositions of the physical parts of icons. SIL provides three operators: horizontal concatenation (represented with the character '&'), vertical concatenation ('∧'), and spatial overlay ('+'). Using these operators, a string can describe a complex physical icon. For example, the iconic sentence in Figure 2.2 can be represented by the string "(box & box& box) + cross".

Parsing of an iconic sentence is a multistep process. First the spatial operators are inserted between icons in the picture. Each operator has an associated division rule [15] which specifies the regions occupied by the arguments of the operator. The division rules are used to compute operator dominance and precedence relationships for the picture. Using these relationships, the operators in the picture are ordered and the picture transformed into a pattern string. The pattern string is then parsed according

to the grammar, using a parsing algorithm for context free grammars.

The structural analysis method used by the SIL-ICON compiler can only handle a limited range of visual languages. The problem is that when picture is converted to a pattern string, only the precedence and dominance of the spatial operators is used. Thus identical subpictures must be resolved identically, regardless of context.

For example, consider the pictures shown in Figure 2.3 with the picture grammar given. The first picture corresponds to the pattern string

"right & ((box&box)∧(box&box))"

and the second to the string

"down ∧ ((box∧box)&(box&box))".

The subpicture containing the four boxes needs to be transformed into different pattern strings.

Furthermore, because only a small set of spatial operators is provided, more complex syntactic constructions cannot be described. For example, a diagonal concatenation is not possible. A richer set of spatial operators could be introduced, however this complicates the transformation into a pattern string even further.

2.1.3 Visual Grammars

Lakin [48] has developed a method for specifying visual languages. His philosophy is very similar to mine. Lakin defines a visual language as "a set of spatial arrangements of text-graphic symbols with a semantic interpretation that is used in carrying out communicative actions in the world." A text-graphic symbol is a visual symbol consisting of either a fragment of text or a primitive graphical element.

Lakin advocates using a general purpose graphics editor to create a picture consisting of atomic text-graphic symbols with an associated spatial arrangement. He coined the term *spatial parsing* for the process of recovering the underlying syntactic structure of the picture from its spatial arrangement. He has constructed a visual programming environment that combines a picture editor with a spatial parser.

Lakin uses a *visual grammar* to define the syntax of a visual language [49]. A visual grammar consists of a context-free grammar which has been visually annotated to indicate the spatial arrangement of the elements of productions. Figure 2.4 shows a sample grammar defining a simple family of bar charts. The right-hand side of each rule is a spatial template indicating where the members of the expression are located. The connections between the line over the template and the elements of the right-hand side describes the resulting parse tree structure. The elements of the right-hand side can be visual literals (i.e., terminals), predicates testing for an acceptable literal (e.g., textlinep), or nonterminals (e.g., "*bar-list*").

A weakness in Lakin's visual grammars is that the spatial relationships within a production are not formally specified. The spatial relationships must be precisely defined to correctly specify visual language syntax. For example, consider whether the picture shown in Figure 2.5a is an element of the language defined by the grammar in Figure 2.5b. The correct interpretation depends on the intent of the language designer. In Lakin's system, the exact relationship expressed by a production is determined by

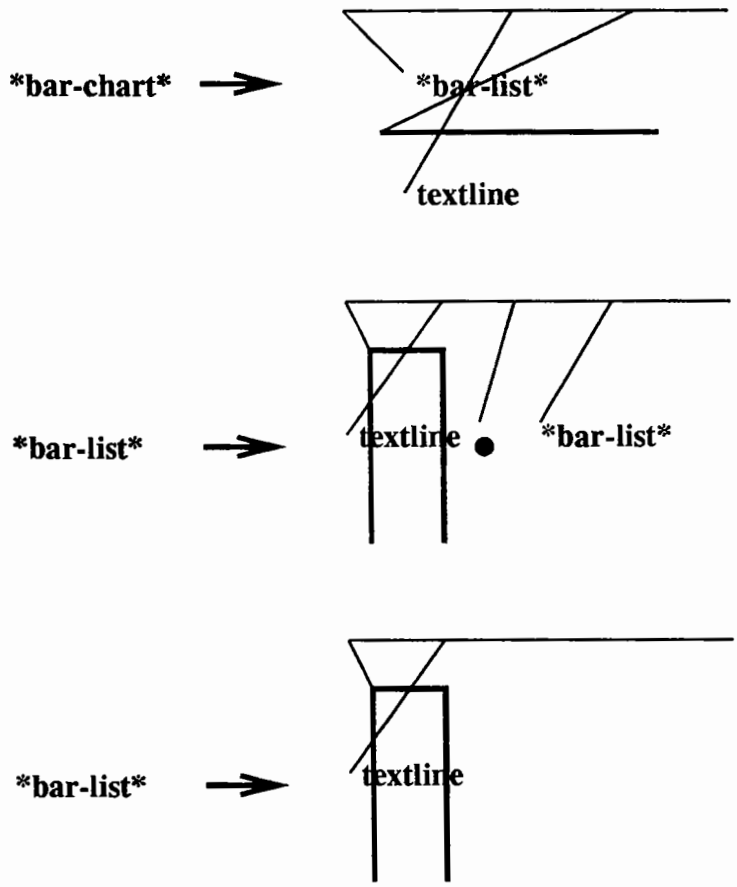


Figure 2.4: A Visual Grammar for Bar Charts

how the system interprets the production.

Lakin developed a spatial parser which takes as input a region of space and a visual grammar. It works in a top-down fashion. Given a target region and a desired element, the parser selects a rule with that element in the left-hand side. The spatial template is used to determine the subregions to search for each of the members of the right-hand side. Visual literals and predicates are checked for by a pattern matcher. Nonterminals cause the parser to be called recursively on the subregion. If a rule cannot be correctly applied, the parser backtracks and tries another rule.

This approach to parsing requires that the visual language be deterministic. That is, the parser must be able to determine the appropriate division of the input region from the current rule. If this is not the case, the parser will have to backtrack not only over the rules, but also over the application of a rule. It is not apparent that Lakin's parser was capable of handling this, and even so it could require exponential time for

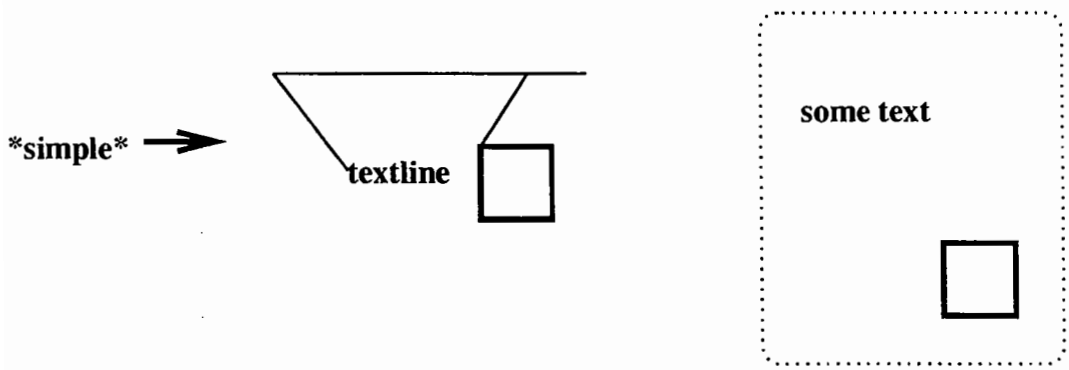


Figure 2.5: An Imprecise Grammar and Ambiguous Picture

backtracking.

2.1.4 Show and Tell

The Show and Tell project [44] designed STL, an icon based visual programming language for school children and novice computer users. The main goal of the language was 'Keyboardless Programming'. The Show and Tell project shares two features with this thesis. First, they held that a visual program should be created with a graphics editor, and coined the term "programming in MacDraw" to refer to this. Secondly, although the primary interface for STL was a language specific editor and environment, they also developed a grammar definition of the STL syntax and a parser for STL programs.

The model of computing is based on two concepts: *dataflow* and *completion* [43]. An STL program (called a puzzle) is comprised of boxes connected by arrows. The arrows indicate data flow within the program. Some of the boxes will contain values and some will be empty. An STL program computes by *completing* the puzzle - that is, filling in the empty boxes.

An STL puzzle is a two-dimensional program built from components. The shape and type of components is significant, but the size and layout (location) of them is not. The components used to form STL programs include various types of boxes, arrows and ports (shapes marking where arrows enter boxes). A group of nonoverlapping boxes connected by a set of arrows combine to form a *boxgraph*. A box can be empty or it can contain a data object, an operation icon, or another boxgraph. Arrows begin and end at boxes, and may cross through, into or out of a complex box. A boxgraph must be acyclic.

The Show and Tell project developed an environment for programming in the STL language. This environment utilized a special purpose editor for constructing puzzles. The editor is based on the internal structure of the language. It provided commands

G_{DFN} = ⟨ {B,A,BATB,EATB}, {DFN}, DFN, {R1,R2} ⟩
where the productions R1, R2 are given by:

(R1) DFN[h1,i1,t1,a1] → B[h2]

indices.1: $h1 = h2 \wedge i1 = \{\} \wedge t1 = \{h2\} \wedge a1 = \{\}$

(R2) DFN[h1,i1,t1,a1] → DFN[h2,i2,t2,a2] BATB[ar3,n3] A[ar4] EATB[ar5,n5] DFN[h6,i6,t6,a6]

indices.2:
$$\begin{aligned} h1 &= h2 \wedge \\ i1 &= i2 \cup i6 \cup (\{n6\} - t6) \wedge \\ t1 &= ((t2 - \{h2\}) \cup t6) - i1 \wedge \\ a1 &= a2 \cup a6 \cup \{ar4\} \wedge \\ h2 &= n3 \wedge \\ ar3 &= ar4 = ar5 \wedge \\ h6 &= n5 \end{aligned}$$

nocycle.2: $h2 \notin \{h6\} \cup (t2 - \{h2\}) \cup i2 \cup i6 \cup t6$

bigger.2: $a2 \neq a1 \wedge i2 \neq i1 \wedge t2 \neq t1$

Figure 2.6: Example of an Index Set Grammar

that corresponded to operations on the internal structure, such as adding an arrow or connecting an arrow to a box.

In addition to the structure editor, a grammar-based approach was investigated. A new type of grammar, called an *Index Set Grammar*, is introduced as a formalism capable of defining two-dimensional syntax [30]. An Index Set Grammar is defined by $G = \langle T, N, S, R \rangle$, where:

- T is a finite set of terminal classes
- N is a finite set of nonterminal classes
- s is the starting symbol, $s \in N$
- R is a finite set of rewriting rule classes
- V is the vocabulary, $V = T + N$

Elements in R have the form $\langle A \rightarrow \alpha, S \rangle$, where $A \in N, \alpha \in V^*$ and S is a set of first-order predicate calculus expressions. Each $v \in V$ has a fixed number of indices, which serve a function similar to attributes. The predicates in S are semantic constraints on the production and a rewriting rule can only be applied if all the semantics in S are true. The formalism is similar to an indexed grammar [2] where the RHS of productions are considered to be unordered. Figure 2.6 shows a grammar for a much simplified version of the STL language.

Before parsing an STL program, the picture is analyzed and transformed into lexical tokens. One token is generated for each elementary object in the puzzle and for every relationship of interest between elementary objects. The indices are used to relate the

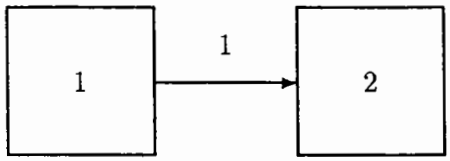


Figure 2.7: Sample Picture

relationship tokens to the object tokens. For example, the picture in Figure 2.7 would produce the tokens:

B[1]	box 1
B[2]	box 2
A[1]	arrow 1
BATB[1, 1]	beginning of arrow 1 touches box 1
EATB[1, 2]	end of arrow 1 touches box 2

Parsing of an index set grammar is accomplished using an expert system. The grammar is translated into a program in the expert system language OPS83. The grammar symbols become types, with a field for each index. The productions are instantiated as rules. The resulting expert system program is capable of both relating the indices of the objects in productions and applying the semantic constraints.

The index set grammar approach has several shortcomings. The first deficiency is the cumbersome nature of index set grammars. The use of indices and explicit relationship tokens make the index set grammar very nonintuitive. From reading an index set grammar, it is difficult to get a sense of what the pictures look like.

Another problem is the explicit representation of relationships between objects. This assumes that all the relationships of interest can be determined before parsing. In the simplified version of STL described, the only relationships are arrows touching boxes. In visual languages in general, other relationships, such as spatial adjacency, are possible. Considering the picture in Figure 2.3, it is clear that the important relationships cannot be determined by only considering the individual boxes. Since the relationships are explicitly represented as input tokens, computing all valid relationships would fill the input with tokens that are not actually part of the picture.

Lastly, no general purpose parser for index set grammars is given. Instead, the grammar is translated into a set of rules for a production system, which is then used as a parser. This approach is inadequate because the translation is not done automatically. Every visual language would essentially require the creation of a special purpose parser for that language.

Despite the weaknesses of the index set grammar approach, it is interesting for several reasons. First of all, it embraces the notion of having a formal two-dimensional grammar to specify the syntax. Secondly, index set grammars are designed to generate unordered collections of objects, much the same as attributed multiset grammars. Finally, index set grammars use constraints to both control the parse and enforce semantic conditions.

Although there is a surface similarity between my approach to visual syntax and that of Show and Tell, there is an important distinction in the use of attributes and indices. With index set grammars, the indices are used to identify the various objects in the picture. The relationships between objects are explicitly represented as tokens whose indices indicate which objects are involved. By contrast, the approach taken in this thesis uses attributes to represent geometric information. Relationships between objects are expressed as constraints in the productions. The relationships between objects do not have to be precomputed, because they are represented implicitly in the geometric attributes.

2.1.5 AMBIT/G

AMBIT/G [18] is a graphical programming language for the manipulation of directed graphs. The language is designed for visually expressing algorithms to build and manipulate data structures. The basis of the language is pattern matching. Programs are specified by giving diagrams of the data before and after the manipulation. An AMBIT/G program operates on a single data object, the *data graph.*

An AMBIT/G statement specifies an operation to be performed. Shapes are used to designate types in AMBIT and edges between shapes correspond to pointers between data objects. Names within shapes are used for identifying constants and variables. A statement consists of an input subgraph to be matched against the data graph and a resulting graph with changes to the links (edges). A statement has two exits, selected by whether or not the match was successful. A program in AMBIT/G may be viewed as a control-flow graph, where each node in the graph contains either a statement or a subroutine call.

The interface for creating an AMBIT/G program [75] used a graphics editor that incorporated a simple type of spatial parsing. A program was created by selecting shapes from a menu and positioning them on the screen with a light pen. Drawing a line between two objects indicated a connection. The editor represented a programs as a picture, rather than the underlying graph. Thus, the user could create pictures which were incomplete and incorrect in intermediate steps.

After a picture was created, the **Accept** button invoked the parser. The parser checked the picture for syntax errors such as a line which did not link to a shape. The picture was processed by the parser to yield the underlying graph structure. Although AMBIT/G used a spatial parser. it was implemented in an ad-hoc fashion and specific to the AMBIT/G language.

2.1.6 G-LOTOS

G-LOTOS is a graphical syntax for the LOTOS specification language. Bolognesi and Latella have examined the problem of developing a formal definition for G-LOTOS [9, 8]. The emphasis in their work is on developing a complete and correct specification of the syntax and semantics of the G-LOTOS language. They do not fully address the problem of recognition and have not (to date) constructed a programming environment

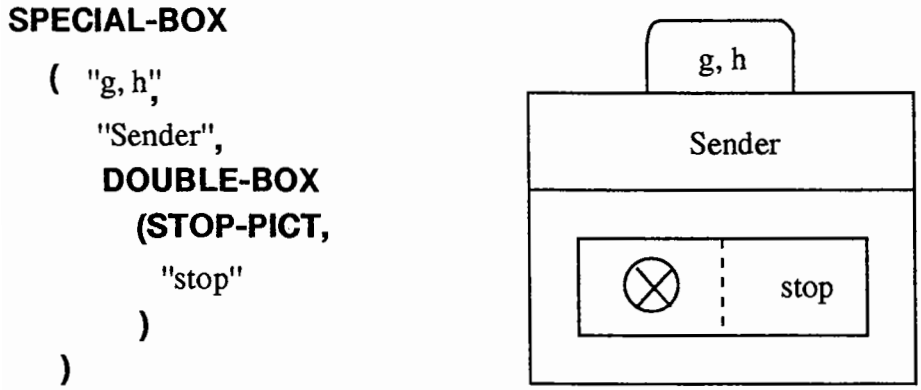


Figure 2.8: A Pictor Term and its Graphic Interpretation

for G-LOTOS.

The approach developed by Bolognesi and Latella is based on a formal model of pictures. They give a textual "picture definition language" called *Pictor*. The strings of this language have an *intended graphic interpretation*. Figure 2.8 shows a term of the Pictor language and its intended graphic interpretation. This language defines an algebra of pictures, where nullary operators (constants) provide atomic pictures and n-ary operators provide complex pictures by combining (i.e., composing) pictures.

Two methods of specifying GLOTOS based on the Pictor language are described. The first technique is to extend the context-free grammar for the textual LOTOS language with additional productions and symbols corresponding to the Pictor operators. The extended grammar defines a set of Pictor terms which constitute the graphical syntax of G-LOTOS. This approach also allows the mixing of textual and graphical LOTOS elements.

In the second approach, an abstract syntax for LOTOS is defined. The syntax of G-LOTOS is given as a mapping between terms of Abstract LOTOS and terms of Pictor. The mapping is defined with the predicate *pict*, which associates an abstract LOTOS term with its pictorial representation (expressed as a Pictor term). The language G-LOTOS is defined as the interpretation of all those Pictor terms which are representations of abstract LOTOS terms.

The formal definition of G-LOTOS provides a precise specification of the graphical language. It can also serve as the basis for the operations semantics of G-LOTOS. The syntax specification mechanisms used do not provide a basis for parsing input pictures, however. The extended grammar approach would require first transforming a picture to the appropriate Pictor string. In general, this requires a knowledge of the hierarchical structure of the picture (which is what the parsing is supposed to compute). The abstract syntax approach could form the basis of a top-down, backtracking parse. However, such a parser would have an exponential complexity for analyzing pictures.

2.2 Syntactic Pattern Recognition

This section briefly surveys other approaches to specifying pictures with grammars. Most of this research has been in the field of *syntactic pattern recognition*. Syntactic pattern recognition is a grammar-based approach to the problem of pattern recognition. In the syntactic approach, a formal grammar is used to describe a pattern in terms of its structure. Although the patterns described are not necessarily images, picture recognition and scene analysis are typical applications of syntactic pattern recognition. A good overview of syntactic pattern recognition can be found in Fu [28].

Even when dealing with images as patterns, the motivation for syntactic pattern recognition differs somewhat from the motivation for visual language syntax. For most syntactic pattern recognition applications, the pictures are real world images, rather than elements of formal visual languages. For example, a typical application would be the identification of an aircraft from an outline extracted from an image [98]. The linguistic element is introduced only to describe the structure of the object.

There are a number of approaches to syntactic pattern recognition, depending on how the patterns (i.e. pictures) are represented and the way the grammar is extended to handle two (or more) dimensions. They can be grouped according to pattern representations:

- String grammars encode the picture as a string using primitives to indicate two dimensional composition. Picture Description Language and MIRABELLE are examples of string based systems.
- Coordinate based systems use symbols augmented with coordinate values within a grammar to describe pictures. Anderson's mathematical expression system and Chang's Picture Processing Grammar are both coordinate based systems. Picture layout grammars can be thought of as a generalization of coordinate based systems.
- Besides strings, several types of higher order structures have been used as the basis of syntactic pattern recognition. These include tree grammars, Plex grammars, array grammars, graph grammars and shape grammars.
- Two other approaches are the ESP^3 system and Hierarchical Constraint Processes.

2.2.1 Picture Description Language

Shaw formalized previous attempts at describing pictures with string grammars in his thesis [83]. Shaw defined the *Picture Description Language*, a formal linguistic scheme for pictures. A picture consists of a number of primitive patterns related in a meaningful way. Each primitive is a member of a pattern class. In PDL, a primitive is a two dimensional pattern with two distinguished points, the *tail* and *head*. The abstraction of a primitive is a directed arrow going from the tail to the head.

$$
\begin{aligned}
S &\rightarrow p \mid (S\Phi S) \mid (\neg S) \mid SL \mid (/SL) \\
SL &\rightarrow S^l \mid (SL\Phi SL) \mid (\neg SL) \mid (/SL) \\
\Phi &\rightarrow + \mid \times \mid - \mid \star
\end{aligned}
$$

Figure 2.9: Picture Description Language Syntax

The tail and head are the points where a primitive can be concatenated (or connected) to another primitive. PDL allows primitives to be blank (invisible) or don't care (unspecified). A special null primitive is provided that has the same tail and head. The *primitive structure* of a picture consists of a directed graph where the edges are labelled by primitive pattern classes and the nodes correspond to the connecting points of the primitives. PDL requires that all pictures be connected (but the connections may be made with blank primitives). All pictures must have a well-defined origin from which the PDL description starts.

The primitive structure of a two dimensional picture is described by a string consisting of primitive pattern classes and composition operators. The string forms a PDL expression. The syntax of a PDL expression is given in Figure 2.9. The four binary operators specify the concatenation between two primitives, as shown in Figure 2.10. The + operator connects the head of one primitive to the tail of another. The − operator connects the heads of two primitives and the × operator connects the tails. The ⋆ operator connects both the head and tail of one primitive to the head and tail, respectively, of another primitive. The unary operator ¬ serves to reverse the head and tail of a primitive.

A superscript on an expression (as in S^l) is a label designator. A label designator is used to allow cross-references to an expression with the / operator. The / is a *superposition* operator, and each primitive within its scope refers to an identical labelled primitive outside its scope. The / allows additional concatenations to a primitive to be specified.

PDL expressions are extended to a context-free grammar by allowing non-terminal symbols as replacements for primitive class names. A PDL grammar has the operators, primitive class names and label designators as terminal symbols. A grammar will define a class of pictures. For a particular picture, the *hierarchic structural* description is the parse of the picture according to the grammar. Each production rule in the grammar has a corresponding semantic rule. This can be used to express additional semantics, but no mechanism is provided.

Recognition of a picture proceeds as follows: first the picture is segmented into primitive elements. Then the connections of the primitives are determined and the primitive structural description string is created. Finally, the primitive structural description is parsed according to the grammar, using a general parsing technique (e.g. Early's method).

PDL is capable of describing a wide range of pictures, but has several limitations and

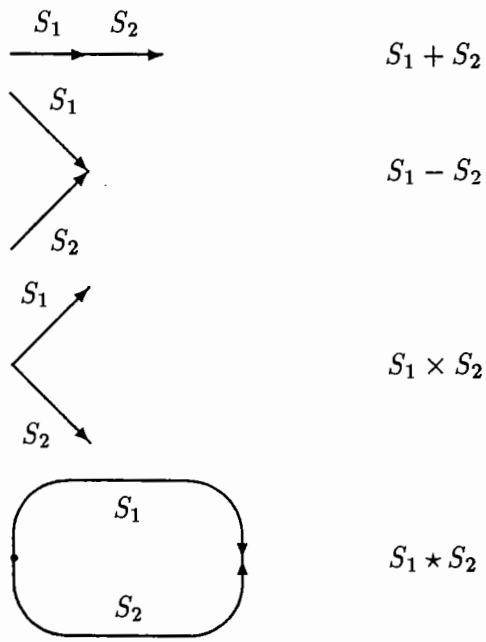


Figure 2.10: PDL Concatenation Operators

shortcomings. One shortcoming is that concatenation of picture elements is the only explicit relation. Other relations, such as geometric relations among disjoint elements, are desirable. The use of blank primitives and semantic rules could allow the expression of some relations. However, this points out a more serious shortcoming, that the picture must first be resolved into a string. This initial resolution requires not only that the picture be uniquely segmented into terminal symbols, but that the connections be uniquely determined by the assignment of terminal symbols.

While this is not a problem for pictures consisting of connected lines or boundaries, pictures consisting of many disjoint components (e.g. tiled objects) generally do not have uniquely determined connections. Resolving which relationships are part of the picture (i.e. what connections are to be made) is most of the problem in recovering the structure of such pictures. The approach presented in this thesis, by contrast, only requires that the picture be uniquely segmented into terminals.

2.2.2 MIRABELLE

Mirabelle is a syntax based drawing recognition and interpretation system [54]. Like PDL, Mirabelle is an attempt to describe pictures using a form of context-free grammar. Mirabelle adopts Shaw's view of a drawing as composed of primitive elements, each of which has two distinguished points, the head and tail. The primitive elements are grouped into two classes: straight line segments and arcs of circles. The primitives are further grouped according to the angle with the horizontal, characterized as a range. Later work extended the notion of primitive to include arbitrary lists of connection

points and subpattern joints [56].

Mirabelle extended the grammar of PDL by introducing operators to express topological relationships. In addition to the five coincidence operators of PDL, thirteen binary *topographic operators* which describe relations of position were introduced. For example, S *above* S' is the shape composed of S and S' with S placed above S'. The topographic operators refer to the rectangular extents of the shapes.

Mirabelle contained a new algorithm for parsing pictures. In contrast to parsing algorithms which proceed left-to-right or right-to-left, Mirabelle's parser is capable of starting at any point in the picture and parsing outwards. This obviates the need to define and locate an origin for each picture as in PDL.

The parser alternates between bottom-up and top-down approaches. Each region is initially parsed bottom-up, by choosing a characteristic primitive from within the region as a starting point. An appropriate production generating that primitive is then chosen based on the context. The remainder of the RHS of the production is processed and the parser moves up to the LHS of the production. If a coincidence operator is used to connect expressions, the parser proceeds top down. For a topographic operator, the appropriate region is computed and processed bottom-up.

The choice of which primitive or rule to use when parsing is done based on the context. For a coincidence operator, the context is the set of primitives which can be connected to its head (tail). For a topographic operator, choosing a primitive to parse a particular nonterminal in a region is done by finding a primitive which either guarantees that the desired symbol is present or finding a primitive which is always present when the symbol is present. If the grammar is ambiguous, the choice will be nondeterministic and backtracking may be required. The grammar is processed to produce all the information required for checking contexts ahead of time, similar to the production of parsing tables for an LL or LR parser.

Mohr [56] developed an algebraic model of pictures based on fixed point solutions to sets of equations. This model treated both types of operators uniformly. The model allows the demonstration of properties used to compute the information for parsing.

Mirabelle improves on PDL both in terms of expressivity of the grammars and sophistication of the parsing algorithm. Two disadvantages remain to using Mirabelle's grammar/parser mechanisms for visual language syntax. First, for efficient parsing, the algorithm requires the appropriate production to be determinable from the current context. This is often not the case for visual languages, where the topological relationships are more important than the coincidence relationships. Secondly, the grammar mechanism is not extensible by the language designer. That is, the language designer cannot specialize the operators, and introduction of new operators requires modifying the parser.

2.2.3 Anderson

Anderson developed a method for analyzing the syntax of two dimensional mathematical expressions [3]. He represented a picture as a set of characters. Each character had an associated syntactic category and six positional coordinates: *xmin*, *ymin*, *xmax*,

ymax, *xcenter* and *ycenter*. Syntactic units (i.e. aggregates) were also assigned these attributes. The two dimensional syntax was specified using syntax rules (i.e. productions) that consisted of the following:

1. the syntactic category of the left hand side.
2. directions for partitioning a character set into subsets.
3. syntactic goals associated with each of these subsets.
4. relations to be tested among syntactic subunits.
5. functions mapping the coordinates of the right hand side subunits into the coordinates of the left hand side.

Items one and three form the rewrite rule for a production. The relations among the subunits specify the conditions for a particular composition, with the coordinate functions giving the resulting aggregate.

Parsing of a mathematical expression proceeded in a top-down fashion. Given a syntactic category as a goal, the parser finds an appropriate rule and attempts to apply it. The character set is partitioned according to the replacement rule and the corresponding goals for each of the subsets are analyzed. If the subgoals are successfully parsed, the relationship between the subunits is checked. If the rule is successful, the coordinates of the aggregate unit are computed and the character set replaced by the left hand side.

The key step in the parsing is the partition of the picture. Two different techniques are used, depending on the type of replacement rule. The first type of rule contains a terminal symbol in the right hand side which is used to divide the character set. It must be possible to delineate the other (nonterminal) subunits of the rule based on their position relative to the terminal symbol. If more than one of the appropriate symbol is found in the character set, they are ordered and each is tried as a partitioning element until either one is successful or all have been tried.

The other type of rule used contains exactly two nonterminal symbols in the right hand side with the restriction that it be possible to partition the characters by a straight line. The equation for the line is used to order the characters and all possible partitions are checked until one succeeds.

Anderson successfully applied his method to the parsing of mathematical expressions, developing a grammar to describe them. However, the form of grammar he developed is too restrictive in the types of relationships which it can describe, so his approach does not generalize to more interesting visual languages. In addition, when rules without terminals are used, the parsing algorithm could potentially use exponential time in backtracking.

2.2.4 Picture Processing Grammar

Chang developed a method for describing the hierarchical structure of two dimensional pictures called a *picture processing grammar* [16]. A picture is represented as a finite

set of (symbol, vector) pairs. The vector associates coordinates with the symbol. A production consists of a rewrite rule and a partially computable function from the associate-vectors of the right hand side to the associate-vector of the left hand side.

A grammar is called *hierarchical* if the productions can be partitioned into blocks $R_1, ..., R_n, n > 1$ such that if α appears as the left-hand symbol of a rule in R_i, then it will never appear as a right-hand symbol of any rule in any $R_j, j < i$. A hierarchical grammar can be separated into levels, each of which can be viewed as a picture transformation. This reflects the hierarchical structure of many pictures.

A grammar is called *nonretracing* when for any picture U, if $\alpha \stackrel{*}{\Rightarrow} U$ and $V \Rightarrow U$, then $\alpha \stackrel{*}{\Rightarrow} V$. This implies that the grammar may be parsed deterministically in a bottom up fashion. Chang gives an algorithm for parsing pictures which are unambiguous and nonretracing. When retracing is required, exhaustive search is used to find a correct parse.

A restricted type of picture processing grammar is given for describing two dimensional mathematical expressions [15]. As with Anderson's approach, the associate-vector for each symbol contains six coordinates. Two concepts are used for ordering the symbols in a picture: precedence and dominance. After analysis, the picture is converted into a string, which can be used to recover the syntactic structure of the picture. The special purpose method is more efficient for recognition, but is restricted in the class of languages that can be specified. This approach has been applied to icon languages in the SIL system [17], discussed in Section 2.1.2.

2.2.5 Tree Grammars

One attempt to expand the power of the grammar is to use trees instead of strings [29, 27]. Trees are a natural extension of one-dimensional strings for describing patterns. Trees are used to describe pictures as follows: let Σ be a ranked alphabet, and let T be a tree with the nodes labelled by symbols from Σ. Let \$ be the label of the root of the tree and the number of children at each node be the rank of the symbol labelling that node. Now associate a picture primitive with every symbol other than \$. As with the string grammars, each primitive has a distinguished head and tail. A tree can describes a pattern where \$ represents the origin of the tree, and the children of each node indicate primitives with their tails connected to the father primitives head. For example, Figure 2.11 shows a picture and the tree describing it, using the primitives $\stackrel{a}{\rightarrow}$ and $\stackrel{b}{\rightarrow}$.

A class of pictures can be described using a *tree grammar*. A regular tree grammar is defined over a ranked alphabet V, where productions are of the form $\Phi \rightarrow \Psi$, where Φ and Ψ are trees over V. This system generates trees as follows: If α is a tree with subtree Φ, then $\alpha \rightarrow \beta$ where $\Phi \rightarrow \Psi$ is a production and β is α with Φ replaced by Ψ. The language generated by a tree grammar can be recognized by a tree automaton [28]. Tree grammars eliminate the need to linearize the picture into a string, but they are still dependent on the model of the primitive as an object with two connecting points, and thus too restrictive for the specification of visual languages.

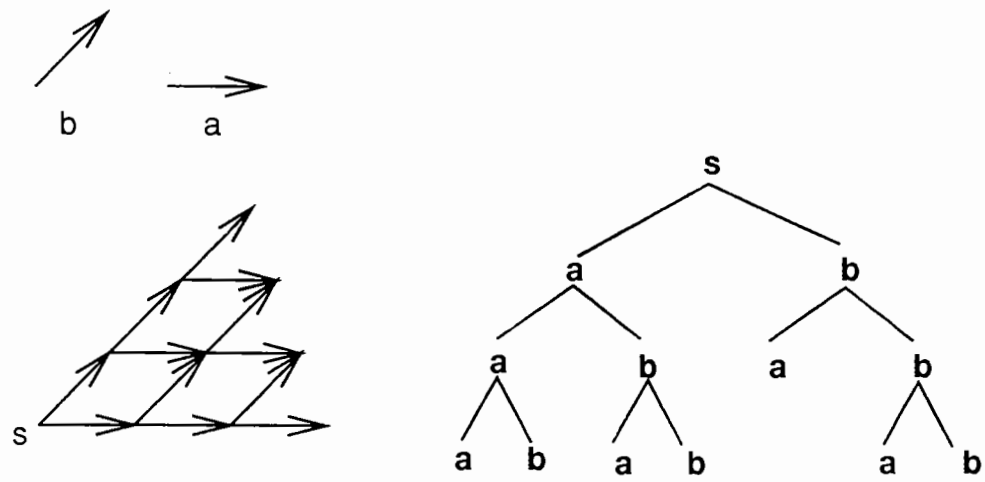


Figure 2.11: A Picture and The Tree Describing It

2.2.6 Plex Grammars

String and tree grammars used primitives which had two connecting points each. Feder generalized this notion to primitives which have and arbitrary number n of attaching points for joining to other symbols [25]. A symbol of this type is called an *n attaching-pint entity* (NAPE). A structure formed by connecting a set of NAPEs is called a *plex structure.* A *plex grammar* is a specification for a language formed of plex structures. A plex structure can be encoded as three strings: an ordered list of components, a list of joints and a list of tie-points. Each component is a NAPE. A joint is a connection of the attaching points of two or more NAPEs. The tie-points are the attaching points of components which are not connected to any joints.

Plex grammars have been used to describe structures such as rubber molecules, shift registers and flowcharts [25]. As a means of describing visual languages, Plex grammars have three shortcomings. First. although more general than string concatenation, Plex grammars are still based on connection and do not express geometric relationships well. Secondly, Plex grammars are difficult to understand and specify (i.e. they are a very non-intuitive representation). Lastly, there is no efficient mechanism for parsing Plex grammars.

2.2.7 Array Grammars

One generalization of string grammars is to extend the grammars to generate two-dimensional arrays. An *array grammar* [55, 73] operates by replacement of subarrays. An array grammar is a five-tupie $AG =< V, \Sigma, R, \#, S >$, where

- V is a finite set of symbols, the *vocabulary.*

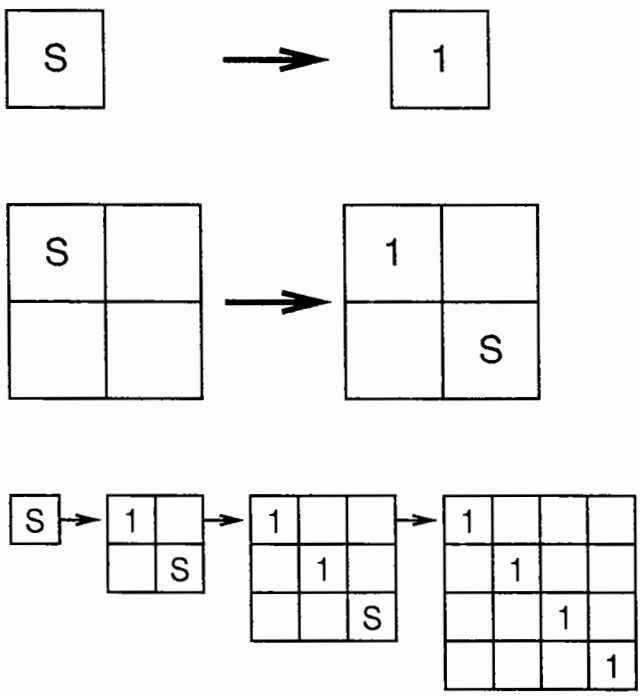


Figure 2.12: Two Array Productions and the Generation of an Array

- $\Sigma \subset V$ are the terminal symbols.
- R is a finite set of rewriting rules, each of which map from one finite, connected, nonempty array of symbols in V to another.
- $\#$ is a *blank* symbol.
- S is the initial symbol.

The initial array is an infinite array containing exactly one nonblank symbol, the initial symbol S. This array is rewritten by matching a subarray that occurs in the left-hand side of a production and replacing it with the subarray on the right-hand side. The two arrays in a production are required to be the same size. An array grammar is *monotonic* if it cannot create blanks (i.e., any nonblank element of the left-hand side must be nonblank in the right-hand side). Figure 2.12 shows two productions of an array grammar that generates identity matrices. Array grammars have been used to describe a variety of geometric patterns, but they are too low level (i.e., describing the picture at the pixel level) to be used for visual languages.

2.2.8 Web and Graph Grammars

String grammars and Plex grammars encode images with one dimensional strings, using special operators or structure to represent the two dimensional aspects of the picture. Tree grammars extended string grammars to use trees to represent images. Graphs are a direct extension of trees. Graphs may be used to represent images in two ways. First, as with string and tree grammars, a picture can be viewed as connected primitives, which maps directly onto graphs with nodes for primitives and edges for connections. A second representation of a pattern is the *relational graph*, where nodes in the graph represent either subpatterns or primitives, and edges represent the relation between two nodes.

One type of two-dimensional grammar is the *Web Grammar* [62]. A web is a directed graph with labels on the nodes [28]. A web grammar is a four-tuple $G = (N, T, P, S)$ where N and T are the nonterminals and terminals, respectively, S is a set of initial webs and P is a set of web productions. A web production is defined as

$$\alpha \rightarrow \beta,\ E$$

where E specifies how the nodes of β are connected to the neighbors of α. If all the labels are the same, the web is an unlabelled graph, and we have a graph grammar. Graph grammars have been used by several authors to represent pictures [28]. The major difficulty is the inefficiency associated with parsing graph grammars [91].

One approach is to define a subclass of graph grammars which are more tractable for parsing. Sanfeliu and Fu have suggested *tree-graph grammars* as a class for parsing [78]. Tree-graphs are graphs which can be overlaid with a tree to infer a hierarchical structure. By utilizing the tree structure mapped onto the graph, tree-graph grammars can be parsed in $O(n^4)$ time. Another approach is to map graphs into strings or trees for parsing. Shi and Fu suggested a class of graph grammars called (attributed) expansive graph grammars which generate languages contained in a graph family Ω [84]. Graphs in this family can be mapped into a string representation for syntax analysis. None of these approachs are well-suited to the specification of visual languages, both because of the expressiveness (i.e. what visual languages can be described) and ease of use (i.e. how natural it is to specify visual languages).

2.2.9 Programmed Graph Grammars

Bunke suggested the use of *attributed programmed graph grammars* for interpreting diagrams [13]. He extended graph grammars in two ways:

- The grammar is programmed by a control diagram.
- The graphs are augmented with attributes.

The control diagram is a directed graph with the nodes labelled by productions and the edges labelled by $\{Y, N\}$. It also has distinguished initial and final nodes labelled I and F, with no edges entering or leaving them, respectively. The control diagram

acts as a finite state machine to control the application of productions. A derivation begins at a node which is a direct successor to the initial node. The label of the node indicates a production to apply. If the application is successful, the Y edge is followed, otherwise the N edge indicates the next state in the control diagram. The derivation is complete when the final node is reached.

Bunke applied attributed programmed graph grammars to the understanding of schematic drawings such as circuit diagrams or flowcharts. A drawing consisting of line segments is transformed into an *input graph*, where vertices (intersections or ends of line segments) are nodes and the line segments are edges. The nodes are labelled by their degree. Rather than parsing the graph, the grammar is used as a generative tool to transform the input graph into a symbolic representation where the nodes correspond to components (e.g. a resistor) and the edges indicate the connectivity. The control diagram directs the application of productions to the input graph until the final node (in the control diagram) is reached.

This technique transforms the input graph, rather than recovering structure from the diagram. The technique is useful when the diagrams consist of discrete connected components. The processing will extract the components and the topology of the connections from the picture. Attributed programmed graph grammars are not general enough to use for specifying visual language syntax.

2.2.10 Shape Grammars

Shape grammars are a syntactic method of defining pictures that were developed originally to generate non-representational, geometric paintings [32]. Shape grammars have also been used to analyze two-dimensional line drawings of three dimensional structures [31]. A shape grammar is a four-tuple $SG =< V_t, V_m, R, I >$, where

- V_t is a finite set of terminal shapes.
- V_m is a finite set of nonterminal *marker* shapes.
- $R = \{(u, v) | u \in V_t^* \times V_m^+, v \in V_t^* \times V_m^*\}$ is a set of *shape rules*. u and v are shapes formed from the terminal and marker symbols, with u containing at least one marker.
- $I \in V_t^* \times V_m^*$ is an initial shape.

A shape is generated from a shape grammar by starting with the initial shape and applying shape rules. A shape rule specifies a rewrite, where u is matched against the current shape and replaced by v. A derivation is complete when the current shape does not contain any marker shapes. Figure 2.13 contains a shape grammar and some shapes it can generate.

Gips [31] applied shape grammars to the analysis of line drawings comparing three-dimensional objects. The analysis was syntax-directed (i.e. driven by a shape grammar), but not a general recognition procedure. An initial input picture was transformed into a labelled graph, which was rewritten by repeated application of productions until the entire input was reduced to an empty picture.

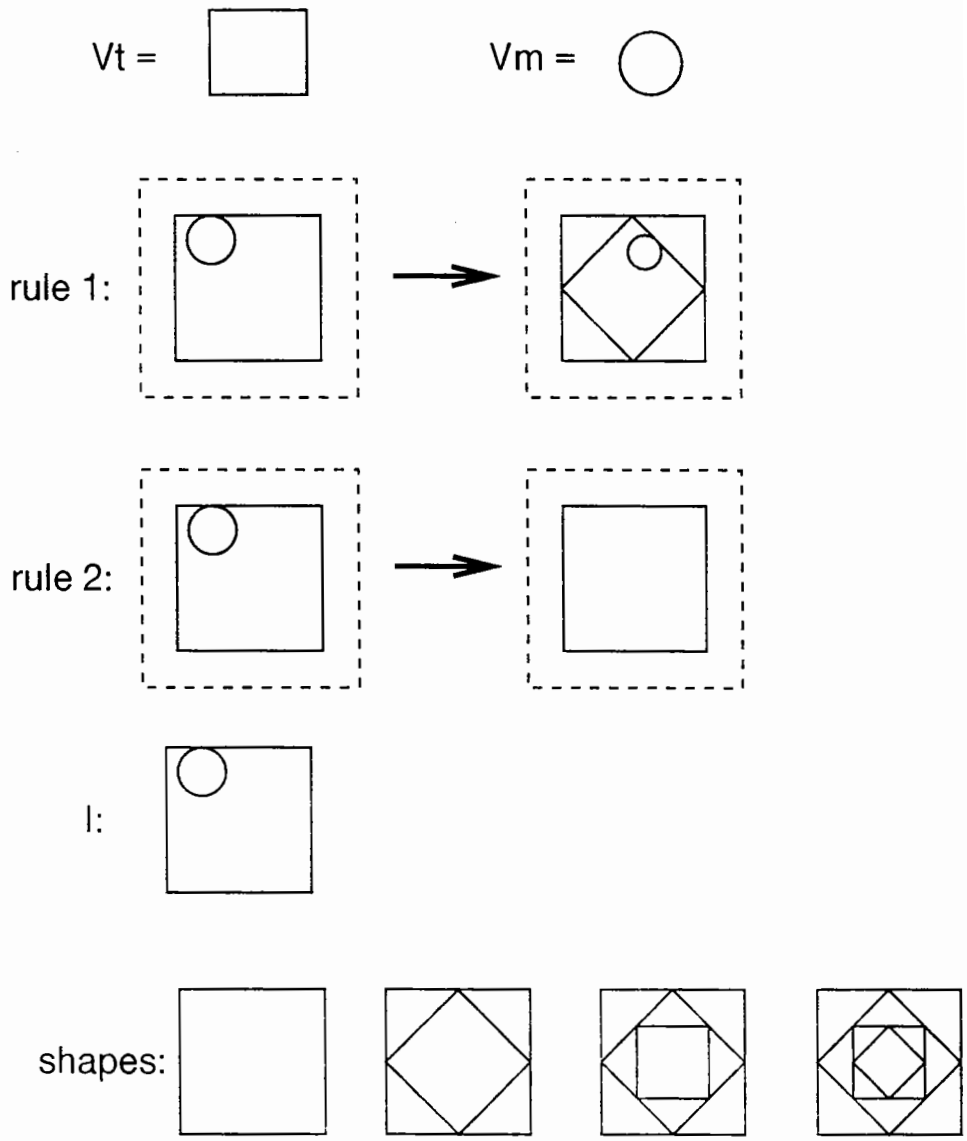


Figure 2.13: A Shape Grammar and some Generated Shapes

2.2.11 ESP^3

The extended Snobol picture pattern processor (ESP^3) was both a programming language and a pattern recognition system which was designed for generating, recognizing and manipulating two-dimensional line drawings [82]. ESP^3 is an attempt to extend the Snobol4 language from strings and patterns of strings to pictures and patterns of pictures.

A picture in ESP^3 is a two-dimensional line drawing composed of primitives such as LINE, ARC, CIRCLE, etc. A picture pattern consists of a sequence of pattern components related by either alternation or composition. A pattern component is either a primitive, a builtin predicate or a *such-that* field. Each pattern component controls how the pattern matches the data. A primitive will match a picture of the given class. A built-in predicate allows the user to apply constraints to the matching by testing the basic attributes of and relationships among pictures. The such-that field is similar to a built-in predicate, but is used to control where the pattern matcher searches (i.e. the constraints are applied before pattern matching, rather than after).

ESP^3 performs pattern matching by searching the subject picture for an occurrence of a subpicture which matches the picture pattern. The search is an ordered scan of the subject picture and is controlled by the pattern and any such-that clauses. The subject picture is scanned in a top-to-bottom, left-to-right order. When a match fails, the pattern matcher backtracks and tries a new match for some primitive.

The search strategy in ESP^3 suffers from two inherent problems. Because it uses a top-down, backtracking approach, ESP^3 effectively performs an exhaustive search of all possible pictures while matching against the input. A further complication is the need to create a new instance for every matched subpicture. These problems lead to slow searches within the pattern matcher.

2.2.12 Hierarchical Constraint Processes

Davis and Henderson developed a technique for the analysis of two-dimensional shapes called the *hierarchical constraint process* (HCP) [20]. Their technique combines a constraint propagation with a syntactic representation of the picture. The goal of HCP is image recognition. It seeks to deal with ambiguity in both the segmentation of the picture and in assignment of terminal symbols to picture primitives.

The basic approach is to allow the shape to be decomposed into many, possibly overlapping, primitives and to label each primitive with all plausible terminal symbol names. Each combination of primitive and terminal label is called a *level-0 hypothesis*. The goal of the analysis is to find all sets of assignments of terminals to primitives which can be used to form a parse of the shape.

The analysis procedure works in a bottom-up fashion. Constraint propagation is applied to eliminate superfluous hypotheses at a local level. The parser is driven by a *stratified shape grammar*. The grammars serves as both a source of local constraints and an organizing structure for the picture. A stratified shape grammar is similar to a context-free grammar, but with the following extensions:

- Every vocabulary symbol v has a level number $\ln(v)$, with $\ln(v) = 0$ for terminal symbols v, $\ln(S) = n$ for the start symbol S. If $\ln(v) = k$ for nonterminal v and $v \rightarrow v_1 v_2 ... v_r$ is a production, then $\ln(v_i) = k - 1$ for $1 \leq i \leq r$.
- Every vocabulary symbol is composed of a name, a list of attachment points and a set of predicates describing the properties of the symbol.
- Each production consists of five parts:
  1. a rewrite rule.
  2. a specification of the attachments between the elements of the RHS.
  3. semantic constraints on the RHS.
  4. the attachment part of the LHS.
  5. the semantic part of the LHS.

Two types of constraints are used. Syntactic constraints are specified by part 2 of the productions and describe the possible neighbors a symbol may have at any attachment point. Semantic constraints are the relationships between the semantic features of the symbols as specified by part 3 of the productions. The hierarchical constraint process computes a bottom-up parse of the shape by constructing a layered network of hypotheses. Constraints are applied at each level to remove hypotheses which are unsupported. Both the syntactic and semantic constraints can be compiled into tables to speed the analysis [36].

The hierarchical constraint process approach is interesting because it combines constraint propagation with a bottom-up parser. However, stratified shape grammars are not general enough to be used for specifying language syntax. The stratification (i.e. the level numbers) restricts the structure of the picture and disallows recursive productions. While this is acceptable for describing particular images, it does not suffice for visual languages.

Chapter 3

Visual Language Syntax

3.1 Introduction

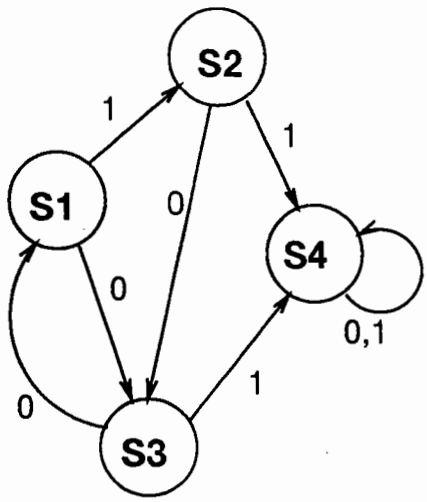
> *To analyze graphic representation precisely, it is helpful to distinguish it from musical, verbal and mathematical notations, all of which are perceived in a linear or temporal sequence. The graphic image also differs from figurative representation, essentially polysemic, and from the animated image, governed by the laws of cinematographic time. Within the boundaries of graphics fall the fields of networks, diagrams, and maps.* [5]
>
> —Jacques Bertin, *Semiology of Graphics*, 1967.

A textual language is a set of strings. A textual program is a string which is an element of a particular textual language. A textual program is composed of symbols which are combined by concatenation into a string. The union of all the symbols used in all strings of a language form the alphabet for that language.

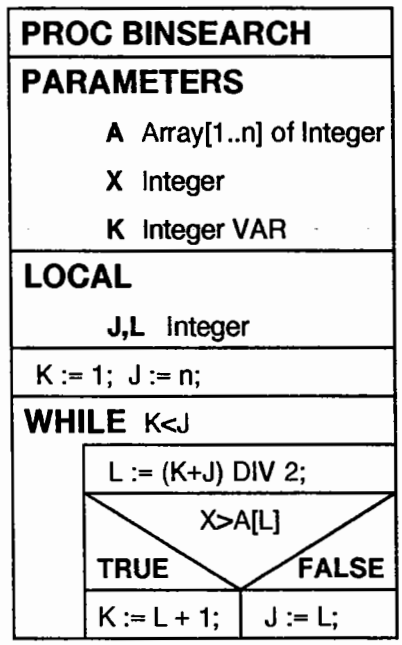
A visual language is a set of pictures. A visual program is a picture which is an element of a particular visual language. A visual program is composed of visual symbols drawn from a visual alphabet. For a textual language, the symbols are lexical tokens such as keywords, identifiers and special characters. Visual symbols are primitive graphical objects such as rectangles or lines. While the symbols in textual languages are only composed using the one dimensional concatenation operator, visual symbols are composed in a two dimensional fashion to form pictures.

Of course, both visual and textual languages may be viewed at a lower level - a picture is simply an array of pixels and a text program is merely a string of characters. Mapping from this lowest level into the elements of the syntax is the realm of *lexical processing*. This dissertation does not address the issue of lexical processing for visual languages, for two reasons. First, at the lowest level, the lexical processing for a visual program consists of finding primitives (e.g., lines) in a picture representation such as an array of intensity values. The problems associated with this are related to the fields of image processing and computer vision, rather than visual language syntax.

Just above this level, however, is the problem of mapping from the recognizable primitives into the constituents of the visual syntax. For example, determining that



a) An FSA Program



b) A Nassi-Schneiderman Diagram

Figure 3.1: Examples from Two Visual Languages

four lines constitute a rectangle. This problem is closer to the textual problem of grouping characters into tokens. Appropriately specifying and extracting these visual lexical units from a lower-level representation is an interesting problem, and is left for future research. It is avoided here for two reasons. First, Chapter 7 describes a visual programming environment based on an editor which will produces the language primitives directly, so the mapping is avoided. Secondly, this mapping can be specified using the method given in this thesis, by viewing the mapping as part of the syntax of the language. An analogy in the textual world is to include the definitions of the lexical tokens in the context-free grammar. Thus, the primary motivation for lexical processing is one of efficiency, and not expressiveness.

The *syntax* of a textual language determines which strings are elements of the language. The syntactic distinctions between different textual languages (e.g., FORTRAN and Pascal) arise from both the symbols found in the language and how those symbols may be organized to form strings in the language.

Similarly, visual languages are distinguished both by what visual symbols are used and by how those symbols are arranged to form a picture. For example, consider the two visual programs shown in Figure 3.1. The first program is a finite state diagram.

It belongs to the language FSA consisting of all pictures which are valid finite state diagrams. The symbols used to form a finite state diagram are circles, arrows and fragments of text. These symbols may be arranged to form a valid FSA program according to some simple rules, intuitively expressed as:

1. each circle has exactly one text fragment within it.
2. each arrow begins and ends on a circle.
3. each arrow has exactly one text fragment labelling it.
4. every text fragment is either within a circle or labelling an arrow.
5. no symbols may overlap (other than a text fragment within a circle).

The program in Figure 3.1b is a structured flowchart. It belongs to the visual language consisting of Nassi-Schneiderman (NS) diagrams [59]. The symbols for this language consist of boxes, lines and text fragments. These symbols are combined according to rules for forming valid NS diagrams. Although the elements of both visual languages are pictures, the two languages are distinct. The picture in Figure 3.1b is not a valid finite state diagram and therefore is not an element of the FSA visual language (just as a FORTRAN program is not an element of the Pascal language).

What is needed, then, is a method of saying which pictures belong to which visual languages. The aim of this work is to develop a model for the specification of visual language syntax which will provide such a method. My approach is similar to the approach used to define the syntax of textual languages, specifying the precise language syntax with a grammar.

This chapter discusses the syntax of visual languages. The next section develops the model of visual syntax. Section 3 describes a simple approach to specifying visual syntax with grammars, and Section 4 illustrates my approach with an extended example. Finally Section 5 examines the example and summarizes.

3.2 The Picture Model

Textual languages can be examined at three levels. At the lowest level are the symbols used in the language. This symbols are atomic tokens; their internal structure is unimportant to the language, except for certain attributes. The next level organizes these symbols into a language element. For a textual language, the elements of the language are organized as strings of symbols. At the highest level these strings are organized into languages (i.e., determine the set of strings which form a language).

Visual languages can also be viewed on three levels. At the lowest level is the *picture element*, which is a primitive graphical object such as a line, shape or text fragment. Again, the picture elements constitute atomic objects in the visual language. A collection of picture elements can be arranged on a plane to form a (two-dimensional) *picture*. Thus a picture is comparable to a string, except that a picture has a two-dimensional organization. Finally, a *visual language* is a set of pictures.

A visual language specification defines the set of pictures which belong to a particular visual language. To develop such a specification requires an understanding of how picture elements are combined to form a picture. This section describes a model of pictures, first discussing the primitive graphical objects which can form visual alphabets, and then describing how these picture elements can be combined to form a picture.

The model presented here is somewhat loose. It is intended to illustrate the important issues in visual syntax. For example, Section 3.2.2 discusses compositions in order to establish a vocabulary of possible interesting compositions. The model in this chapter is meant to provide insight into the structure of visual languages, but is not intended to serve as the specification mechanism itself. Chapters 4 and 5 develop a method for specifying visual languages that uniformly handles the various aspects of visual languages discussed in this chapter.

3.2.1 Picture Elements

The picture elements form the alphabet over which visual languages are defined. They are the basic building blocks from which pictures are formed. The range of possible picture elements is as varied as the visual languages that use them. Thus, exactly what constitutes a picture element depends on the particular visual language being defined. What is needed is a framework to talk about picture elements.

Selker [81] grouped graphical components into three types:

- *images* are real world pictures, such as digitized photographs.
- *icons* are stylized representations with real-world referents.
- *symbols* are drawings of arbitrary designs.

Selker also notes that individual components are distinguished by surface characteristics such as color or texture. The distinctions between the three groups given above are largely based on appearance (although the source of the picture element would distinguish between an "image" bitmap and a "symbol" bitmap).

A model for picture elements should succinctly express the unique properties of picture elements in such a way to distinguish different elements. One approach would be to uniquely identify every possible picture element. The problem with this approach is that it doesn't tell us anything about the picture elements. For example, intuition tells us that a line from $(3,2)$ to $(7,5)$ and a line from $(3,12)$ to $(7,15)$ have more in common with each other than with a circle of radius 4 centered at $(12,6)$. Thus, our intuitive notion of a picture element is expressed as "a line from $(3,2)$ to $(7,15)$. The following definition embodies this intuition.

Definition 3.1 *A* picture element *is a pair,* (s, A)*, where* s *is the* symbol class*, and* A *is a finite set of* characteristic attributes*. A* picture element class *is a pair* (s, I)*, where* s *is a symbol class and* I *is a finite set of attribute identifiers. A picture element* (s_p, A) *is an* instance *of a picture element class* (s_c, I) *if it has the same symbol (i.e.,* $s_p = s_c$*) and for every attribute identifier* $i \in I$*, there is a corresponding attribute* $a \in A$*.*

The symbol class in a picture element specifies what kind of graphical component it is. The characteristic attributes contain all the relevant information about the appearance of a specific picture element. A picture element class serves to group together similar picture elements. For example, all line segments could be grouped together into picture elements with the symbol **line** and attributes giving the endpoints.

In practice, this definition is too general to be used directly in describing compositions of picture elements. There are too many possible picture element classes, each of which can have its own set of attributes. Compositions would need to be specified for every allowable combination of picture element classes. To solve this, the framework is extended. Just as picture elements were grouped into classes, so are picture element classes grouped into *abstract element classes.*

Definition 3.2 *An* abstract element class *is a a set of attribute identifiers I_{abs} and associated interpretations (e.g., that attribute i_{foo} expresses the foo property of the picture element). A picture element class (s, I_s) is a member of an abstract element class if $I_{abs} \subseteq I_s$ and the attributes are interpreted in the same way.*

An abstract element class groups together picture element classes that share a subset of their attributes. For example, a visual language might have three kinds of line segments: solid lines, arrows and dashed lines, each of which has a corresponding picture element class. These three classes have in common attributes for the endpoints of the segment, but the arrow class might have an additional attribute indicating at which end the arrowhead is, and the dashed line might have an attribute containing a pattern for the dash sequence. Two abstract element classes are used in the model of visual syntax: *the abstract line*, and *the abstract shape.*

Abstract lines are defined by $AL = \{l_1, l_2\}$, where l_1 and l_2 are interpreted as locations specifying the starting and ending points of the line segment. This is the class described in the previous paragraph.

Abstract shapes are defined by $AS = \{ll, ur\}$, where ll and ur are interpreted as locations specifying the *lower-left* and *upper-right* corners of the rectangular extent of the shape. Abstracts shapes include many types of atomic graphical entities such as rectangles, circles and polygons. Other abstract shapes include text strings, icons and images (i.e., small bitmaps). All of these graphical objects are bounded by a rectangular extent, and the associated primitive element classes contain the ll and ur attributes.

Text fragments are a special type of abstract shape, consisting of a rectangular extent and a piece of text. The term *text fragment* (or simply text) is used to refer to a piece of text which is an atomic picture element (to distinguish it from a text string which is part of a textual language). Even in a visual languages, text is often the best way to represent primitive data such as numbers or names.

Shapes and lines were selected as the basis for visual language syntax because they are both natural spatial concepts. Both are needed since neither appropriately models the other: representing lines by extents is ambiguous since both diagonals refer to the same extent; using a line to represent a shape removes the implicit structure of the attributes of the shape (i.e., that left<right) which is used in expressing the compositions.

Shapes and lines together can be used to describe many different graphical primitives, although some primitives may require attributes for additional properties. For example, a spline could be abstracted to its extent (for interaction as a shape) or to its endpoints (for interaction as a line). The type of spline (e.g., cubic B-spline) could be represented by the symbol class. To fully describe the spline, however, the control points must also be specified.

The model of abstract shapes adopted here considers all shapes to be ***orthogonal.*** That is, shapes are viewed as rectangles with horizontal and vertical sides. While this suffices for many visual languages, it is not the only possible definition of shapes. For example, other convex polygons, arbitrary polygons, or even curves could be used as abstract shapes. Choosing an alternative (or additional) class of abstract shape would have two effects within the framework presented. First, the new abstract element class would have a different set of attributes associated with the class. Secondly, new compositions would have to be specified for the new class that are meaningful for the new type of shape. While such changes complicate the details of the language specification mechanism (and may increase the cost of parsing, as discussed in Chapter 6), they fit naturally within the framework described.

The characteristic attributes describe the additional properties that distinguish picture elements of the same symbol class. Characteristic attributes allow the abstract element classes to be specialized to the primitive element classes for a particular visual language. Some useful characteristic attributes include:

- the text value of a text fragment.
- color
- fill-style (or texture)
- line-style (e.g., thickness, solid/dotted, etc.)

A specific visual language uses a finite set of different primitive element classes. This set of primitive element classes is called the *visual alphabet.* In the model of visual languages, a visual alphabet is defined as follows:

Definition 3.3 *A* visual alphabet *is a pair, (C, L), where C is a finite set of picture element classes such that every class $c \in C$ is either an abstract shape or an abstract line, and L is a set of locations.*

The locations provide the values for the attributes of the abstract shapes and lines (e.g., l_1). A *two dimensional visual alphabet* is defined as a visual alphabet where $L = \{(x, y) \mid x, y \in Z\}$. Two dimensional visual alphabets provide a picture model that is strictly two dimensional, with no depth information. Thus when two objects overlap, no distinction is made as to which is on top. For example, the two pictures shown in Figure 3.2 would not be distinguished.

Two dimensions are enough to express all the compositions described below, which suffice for a wide range of visual languages. Thus we will restrict our attention to

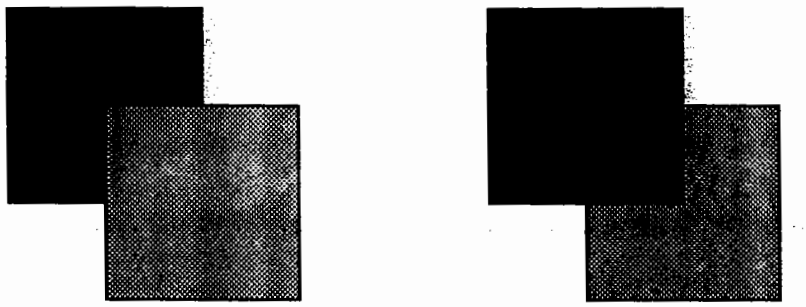


Figure 3.2: Overlapping objects

two dimensional visual languages (i.e., languages defined over two dimensional visual alphabets) for the remainder of this thesis. The model is easily and naturally extended to $2\frac{1}{2}$, 3 or more dimensions (but with a concomitant increase in the complexity of parsing such languages). For example, extending the model to distinguish between the two pictures in Figure 3.2 (i.e., to $2\frac{1}{2}$ dimensions) is easily done by including a *priority* attribute in the abstract element classes. This attribute could be used to order the elements of a picture such that the uppermost object always has lower priority [52].

Finally, a picture is defined:

Definition 3.4 *Given a visual alphabet $\Sigma = (C, L)$, a* picture *is a finite set of picture elements, $P = \{o\}$, such that o is an instance of a class in C and the location attributes of o are taken from L.*

This definition is analogous to the definition of a string for a textual language. The picture is a physical representation, corresponding to what the user sees, in the same way that a string of symbols is how the user sees a textual language element. The deeper structure of the picture is implicit in the attributes of the picture elements, just as the deeper structure of a string is implicit in the positions of the symbols.

This model of pictures is similar to that found in object-based graphical editors, such as MacDraw,[1] where a picture is constructed from a set of basic shapes. These shapes have attributes which may be manipulated to vary the style of the shape. The picture itself consists of a $2\frac{1}{2}$ dimensional arrangement of the shapes. This similarity is intentional, as one motivation for visual syntax specification is to allow the use of an ordinary graphics editor for visual program construction. The model given here differs from the typical graphics editor model in that hierarchy is not explicitly represented. This is because the hierarchy found in a graphics editor (e.g., through the GROUP operation in MacDraw) reflects the user's view of the program structure, which may not correspond directly to the syntactic structure of the picture.

[1] MacDraw is a trademark of Claris Corporation.

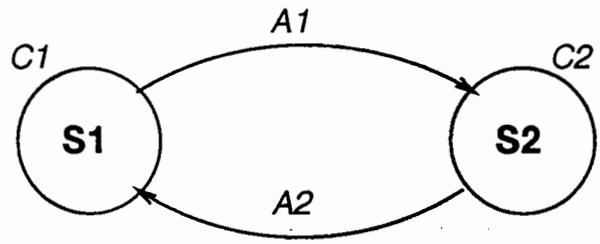


Figure 3.3: An FSA Program

3.2.2 Compositions

The fundamental syntax of a visual language is a set of picture elements. A set based representation of programs is unstructured, in that the structure of the program is not explicit. This is similar to viewing a textual program as a collection of identifiers, operators, keywords and constants. An intuitive understanding of a program overlays it with a structure by grouping the basic elements into aggregates. That is, smaller elements are composed to form larger elements, until the entire program is formed. The composition process provides a syntactic structure to the program.

In a (context-free) textual language,[2] composition combines elements which are adjacent in the string. In this manner, substrings are combined to form expressions, statements, blocks, subprograms and so on. The compositions reflect a relationship (adjacency) between components, and produce aggregates. Visual languages also use compositions to build up the syntactic structure of a picture from its component elements. For example, consider the FSA language described above. A state is drawn as a text string enclosed by a circle. The state is an aggregate object formed by combining the text string with the circle containing it.

The composition operation used in textual languages is concatenation of adjacent strings. Visual compositions are more complicated because they are two dimensional, and many different types of compositions are possible. Compositions can be characterized by the *composition operator* employed. Intuitively, the composition operator is a rule specifying how components are combined to form an aggregate. The operator determines the number of components, the relationship between the components and how the composite shape is formed. String concatenation and two dimensional containment are both examples of composition operators.

[2]In discussing the syntactic structure of a textual language, it is necessary to define the class of string languages considered. For our purposes (i.e., programming languages), context-free string languages are the natural class to use. Without restrictions on the class of textual language, an arbitrary mapping from sets to strings could be used to represent pictures as strings. The "visual compositions" in the resulting "visual language" would not relate elements adjacent in the string. This definition is not useful, however, for specifying and parsing visual languages. Thus, the discussion of textual languages may be taken to apply to context-free string languages.

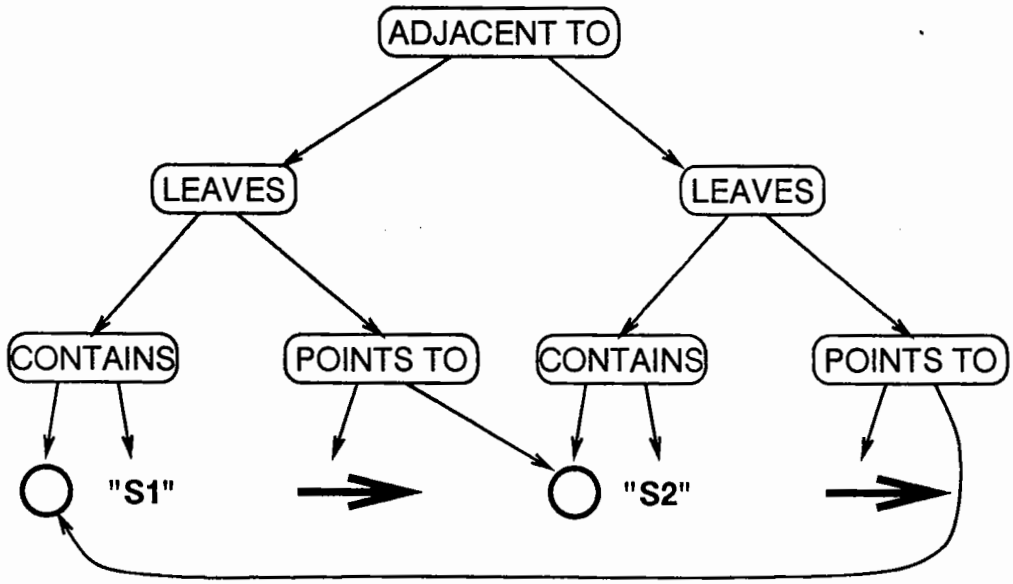


Figure 3.4: Composition Graph for FSA

The decomposition of a textual program yields a tree structure, where the leaves of the tree are the symbols forming the program. Because visual programs are two dimensional, the situation is somewhat more complicated. Consider the FSA language described in Section 3.1 above. Rule two states that every arrow begins and ends on a circle. This implies that the arrows should be composed with the circles at either end to form a valid picture. Now consider the picture of an FSA shown in Figure 3.3. If the arrows A_1 and A_2 are each composed with both the circles C_1 and C_2, then the compositions form a directed graph, rather than a tree.

The definition of this graph is in terms of the aggregates formed by the compositions.

Definition 3.5 *A* composite object *is a pair, (op, A), where op is a composition operator, and A is a finite set of characteristic attributes. Given a picture P, a* composition graph *of P is a pair $CG = (O_c \cup P, E)$, where O_c is a set of composite objects and $E \subset O_c \times (O_c \cup P)$ is a set of edges, such that $(O_c \cup P, E)$ forms a directed acyclic graph. CG is said to* cover *P if $(O_c \cup P, E)$ is connected and has a root node.*

Figure 3.4 shows one possible composition graph representing the FSA in Figure 3.3. The directed graph $(O_c \cup P, E)$ is visualized as a set of nodes $(O_c \cup P)$ labelled by either the symbol class (for $n \in P$) or the composition operator (for $n \in O_c$), and edges between the nodes according to E.

The composition graph expresses the structure of the picture. The definitions of the composition operators determine how the elements of a picture can be combined to form

composite objects. Thus, just as picture elements form the basis of the unstructured elements of visual languages (i.e., pictures), so composition operators form the basis for the structural representation of visual languages (i.e., visual syntax). Representing a picture as a composition graph is similar to a constructive solid geometry (CSG) representation of a three-dimensional object. In CSG, an object is stored as a tree, where the leaves are simple three-dimensional primitive objects, and the internal nodes are operators [26]. The operators can perform either graphical transformations (e.g., translation, scaling), or Boolean set operations which serve to combine objects.

As with the picture element classes, an infinite variety of composition operators are possible. Rather than listing composition operators , the remainder of this section discusses the characteristics that differentiate compositions. Chapters 4 and 5 present a formal framework for defining both picture elements and composition operators, as well as a canonical set of compositions operators.

The question of what constitutes a particular composition is central to the specification of visual syntax. Several aspects must be considered in the definition of a composition operator:

- The number of objects involved.
- The type of objects involved.
- The relationship between the objects.
- The degree of specificity in the relationship.

Picture elements can be considered as nullary (i.e., constant) composition operators. The significance of this view is that composite objects and picture elements are both considered entities within the picture. This leads naturally to viewing composite objects as members of abstract element classes. In other words, the composite object formed by combining the circle C_1 and text **"S1"** is a form of abstract shape, and has characteristic attributes appropriate to an abstract shape (i.e., ll and ur). Picture element classes are extended to composite objects as follows:

Definition 3.6 *A* picture object *is either a picture element* (s, A) *or a composite object* (op, A). *A* picture object class *is a pair* (c, I), *where* c *is either a symbol class or a composition operator and* I *is a finite set of attribute identifiers. A picture object* (o_p, A) *is an* instance *of a picture object class* (o_c, I) *if* $o_p = o_c$, *and for every attribute identifier* $i \in I$, *there is a corresponding attribute* $a \in A$. *A picture object* (o_p, A) *is a member of an abstract element class if* $I_{abs} \subseteq A$ *and the attributes are interpreted in the same way.*

This definition extends the class notion to apply to composite objects as well as picture elements. The term class is used to refer to the picture object class (which is equivalent to the picture element class for a picture element). The definitions above distinguish between picture elements and composite reasons for one important reason:

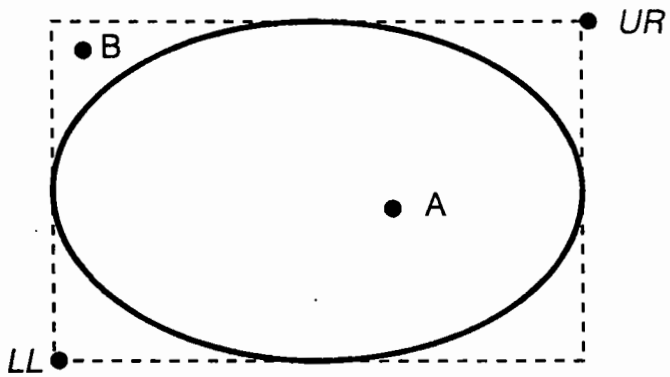


Figure 3.5: An *ellipse containment* Composition

the picture elements are given by the underlying representation of the picture, the composite objects must be computed from the picture using the syntax specification.

Extending the definition of "class" to composite objects provides a basis for composition operators which are applicable to only certain classes of objects. This restriction could be made on the basis of the picture object class or membership in an abstract element class. Restricting a composition to a certain class of picture objects accords with the natural interpretation of compositions. For example, the containment composition used above cannot be sensibly applied if a line is used in place of a circle.

Returning to the "arity" of composition operators, unary relationships are used for compositions which transform the characteristic attributes of an object, such as reversing the endpoints of a "line" object. Another use of unary relationships is to restrict an object class by enforcing a condition on an object. For example, an *empty* composition operator might only be applicable to abstract shapes which do not contain other objects within them.

Binary relationships are the most common, corresponding to the combination of two objects. The two previous examples (concatenation and containment) are both typically used as binary compositions. Compositions involving more than two objects are also possible. For example, a composition could group three abstract shapes which have collinear centers (or corners).

Lastly, instead of combining a fixed number of objects, a composition might relate an arbitrary number of components. For instance, the containment composition could be extended to combine together all objects within a shape. Compositions involving more than two elements can typically be broken down into a series of binary compositions, allowing a more general set of compositions to be defined, with the further detail expressed in the language syntax.

Using the class of the components to restrict which objects can be combined is useful for compositions with more than one component as well. On example mentioned

before was the containment operator, which required an abstract shape rather than an abstract line as the containing object. To further extend this example, an *ellipse containment* composition operator can be defined that requires the contained object to lie within the ellipse bounded by *ll* and *ur*. Figure 3.5 illustrates what is meant by this. Point A is within the ellipse; point B is within the extent, but not within the ellipse. This progression is also an example of the varying degree of specificity in composition operators.

The most important aspect of a composition operator is the relationship between the component objects. Selker [81] identified four categories of spatial structures used to form visual syntax: Positional syntax refers to spatial relationships; Size syntax refers to the relative size of objects; Temporal syntax concerns the use of time (e.g., the ordering of interactions by the user); and Rule syntax describes visual relationships algorithmically.

The composition operators of interest to us fall into the positional and size categories. Temporal relationships are explicitly ignored, as it is a goal for a visual program to be understandable by inspection of its representation as a picture, without any knowledge of the construction of the picture. What Selker terms "rule syntax" is reflected in my approach by the grammar rules, as outlined in the next section and discussed in Chapter 5. The relationships of interest can be grouped into four categories: spatial relationships, coincidence relationships, metrical relationships and non-geometric equivalences.

Spatial Relationships

Spatial relationships in composition operators relate picture objects by their (two dimensional) positions. As with all compositions, spatial relationships are dependent on the types of picture objects. Since each visual language can use its own set of specific object classes, spatial relationships are considered in terms of the abstract classes defined earlier: shapes and lines.

Shape compositions combine two more more shapes which satisfy some spatial relationship. Many different spatial relationships between shapes are possible. Masini and Mohr [54] called this type of composition *topographic operators*, and identified thirteen different spatial relationships. They specified the operators by dividing the plane into nine regions according to the extent of one object and considering the regions occupied by the other object, as shown in Figure 3.6. For example, the *above* relationship corresponds to the region labelled two.

A different *above* relationship could refer to the union of regions one, two and three. This approach can be extended to describe any binary spatial relationship between rectangles. Let A and B be two objects which are abstract shapes and determine the nine topographical regions around the extent of A. Then the spatial relationship of B to A is given by the pair (i, j), where i is the region containing the lower-left corner of B and j is the region containing the upper-right corner of B. Thirty-six different shape compositions are determined by this method, as shown in Figure 3.7. A similar approach could be taken for other shapes (e.g., circles), expressing spatial relationships

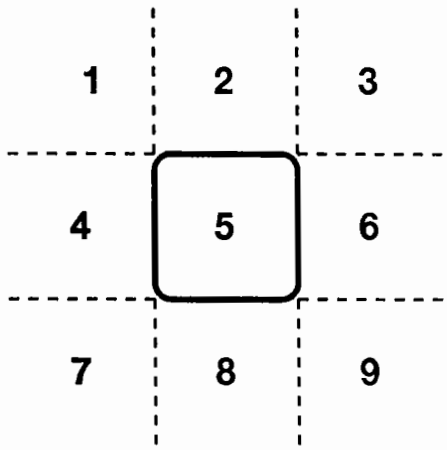


Figure 3.6: Topographical Regions

in terms of the geometry of the shape.

Spatial relationships involving more than two objects can often be formed from several binary spatial relationships. Another way of specifying multi-object spatial relationships is with a *tiling*. A tiling is a rectangular region which is divided into non-overlapping rectangular regions. The Garden system [67] provides a tiling for defining defining spatial relationships. Except for those describing overlapping objects, the binary spatial relationships in Figure 3.7 can be modeled using appropriate tilings.

In addition to objects which are abstract shapes, abstract line objects can also be composed with spatial relationships. The interpretation of spatial relationships like above and below is slightly different for lines than for shapes. Figure 3.8a shows an example of a *below* adjacency relationship between two lines. Line B is parallel to and below line A. This relationship differs from the *below* adjacency for shapes. For example, in Figure 3.8b, line B might still be considered "below" line A, which is certainly not the case if A and B were interpreted as abstract shapes.

In addition, there can be spatial relationships where one (or more) of the objects is an abstract shape and the other(s) are abstract lines. This type of relationship is commonly used to associate a label with a line, as in Figure 3.8c.

Coincidence Relationships

Coincidence relationships relate two (or more) objects based on a common location. For example, a typical composition operator combines two lines where the second endpoint of one line coincides with (i.e., is at the same location as) the first endpoint of the other line. Coincidence can be considered a special kind of spatial relationship. Coincidence is distinguished in the informal characterization of composition operators because much previous research has been based on coincidence relationships [25, 83].

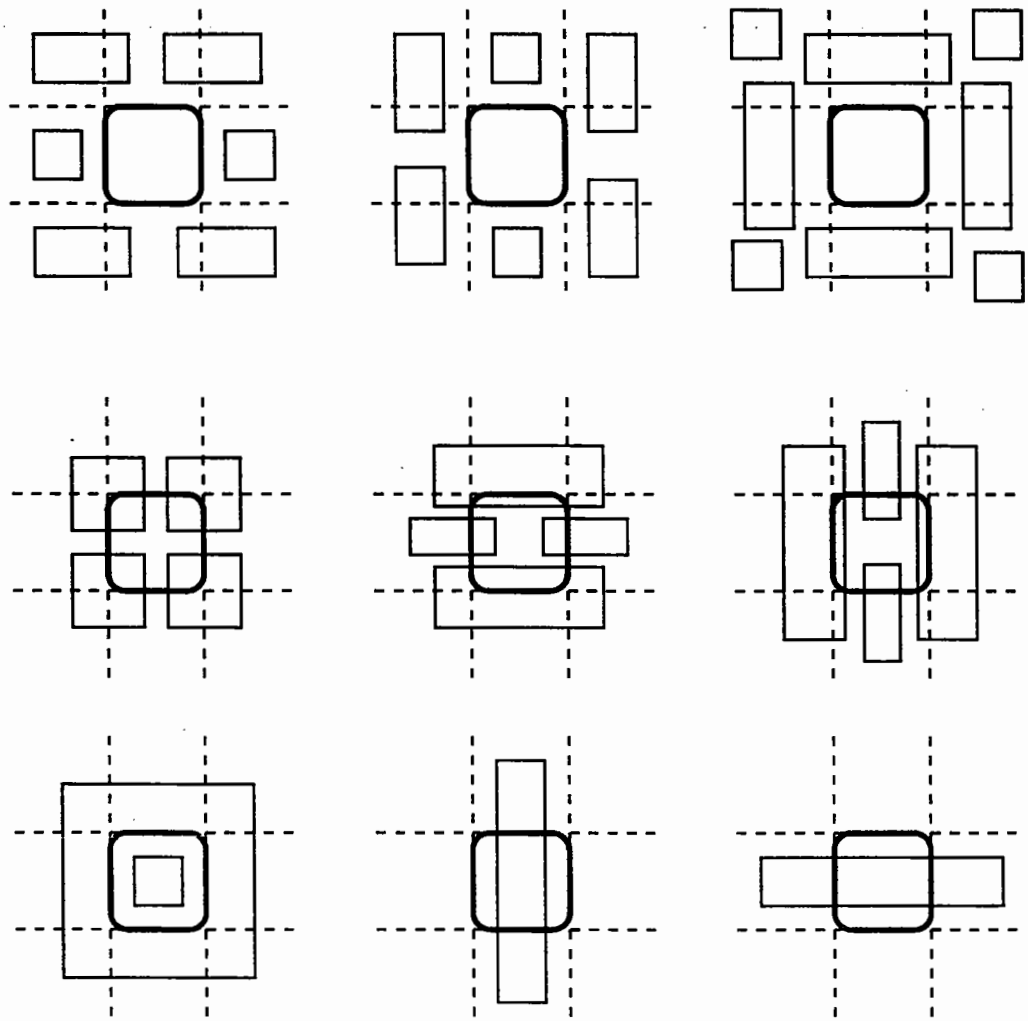


Figure 3.7: Thirty-six Spatial Relationships between Shapes.

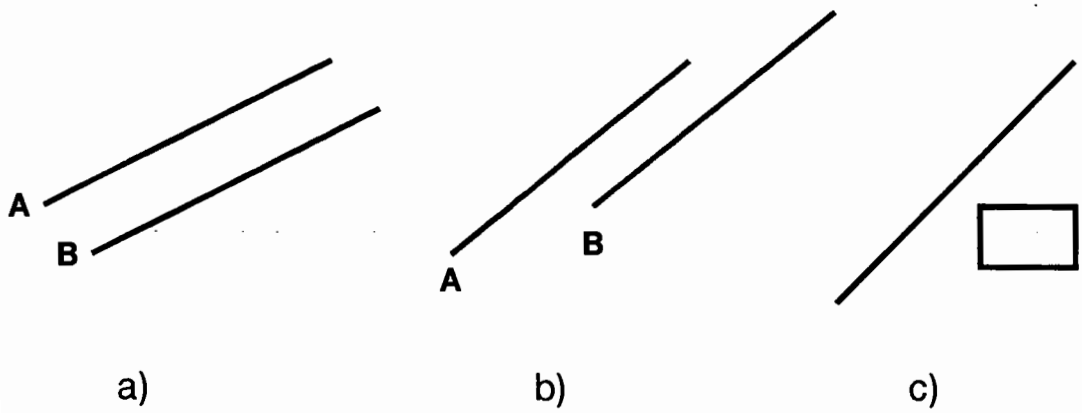


Figure 3.8: Spatial Relationships between Lines: line A *below* line B

The basic composition between two lines is based on the coincidence of the endpoints. In Shaw's [83] Picture Description Language (PDL), he identified four different types of endpoint coincidence, as discussed in Chapter 2. Composition operators based on these coincidences are shown in Figure 2.10.

Coincidence relationships can also be used to define composition operators for combining lines and shapes. The simplest such relationship is for an endpoint of the line to coincide with one of the corners of the shape. This can be extended to allow an endpoint to fall anywhere along the boundary of the shape, as shown in Figure 3.9a. Many different shape/line coincidence relationships are possible: the relationship can be completely constrained (touching in a specific place), unconstrained (touching anywhere), or something in between (e.g., touching anywhere along the bottom). Also, the composition can be varied according to which endpoint of the line is used and how the properties of the aggregate are computed.

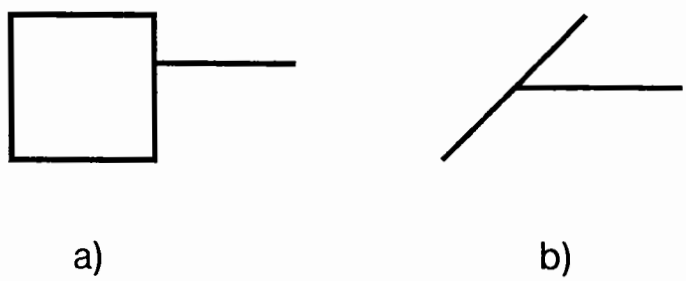


Figure 3.9: Shape and Line Coincidence

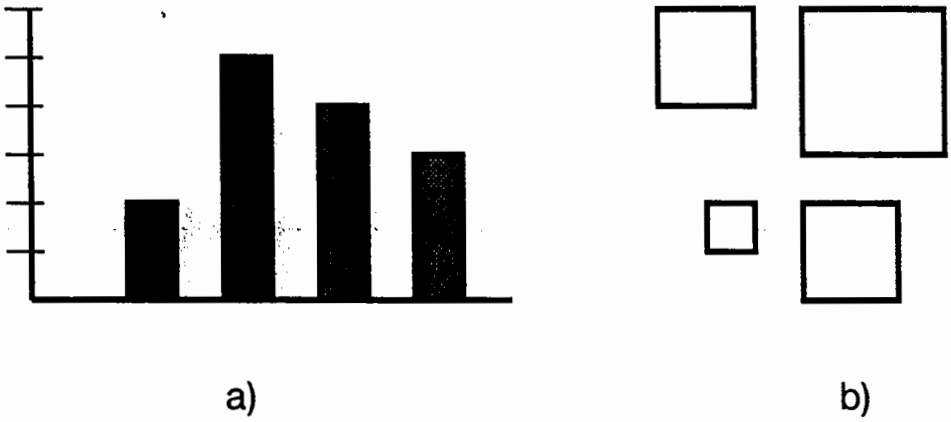


Figure 3.10: Two Examples of Metrical Relationships

A similar type of coincidence relationship exists for lines. Two lines can be composed when the endpoint of one line lies anywhere along the other line segment, as in Figure 3.9b.

Metrical Relationships

A metrical relationship relates objects in some fashion that is dependent on a measured value. The spatial relationships shown in Figure 3.7 are based on the relative positions of the two shapes, without considering the absolute dimensions or locations of the objects. A metrical relationship might take these properties into account. Figure 3.10 shows two examples of metrical relationships. In (a), the filled rectangles are related to the markings to the left by their height; and in (b) a composition might combine the two squares of the same size.

Non-geometric Equivalences

Lastly, besides the various relationships based on the geometric aspects of the picture, compositions can be based on other, non-geometric aspects. For example, a composition might combine two (or more) objects of the same class, or two strings containing the same text, or all objects of the same color. Alone, these relationships are not widely used in visual languages, however they can come into play in conjunction with other relationships. For example, combining a dotted line with other dotted lines which are crossed.

3.3 Grammar-Based Picture Specification

A visual alphabet and a set of composition operators provide the basis for describing pictures. Together they define a universe of possible pictures: the set of pictures formed from primitive elements in the alphabet that are covered by a composition graph over the set of operators. Thus an alphabet and operator set can be said to define a visual language.

Using the entire universe of pictures to define a visual language has two problems, however. First, a picture may have more than one composition graph which describes it. Although a certain amount of ambiguity might be allowed, processing of a visual program (e.g., compiling it) requires that a well-defined interpretation of the program be extracted.

More importantly, the universe of pictures is too broad. A visual language is generally limited to a subset of all possible pictures. Suppose a simple language consisting of a single picture containing two rectangles, one to the right of the other, is to be defined. Using rectangles as a visual alphabet and horizontal concatenation as a composition operator would specify not only the intended picture, but also the pictures with three rectangles, four rectangles, and so on.

A syntax specification must determine exactly which pictures belong to a visual language and how those pictures are constructed by composition of the picture elements. The approach taken here to syntax specification is grammar-based. A grammar is a finite set of rules which specify the syntax of a language. A grammar is based over an alphabet, divided into terminals and nonterminals. The terminals are the symbols that form the language elements. For a visual language, the terminal symbols correspond to the picture elements.

The rules (or productions) of a grammar serve to relate the pieces of a language element, just as the compositions relate pieces of a picture. In a grammar specification of a visual language, the application of a production corresponds to the application of composition operator. The nonterminal symbol on the left-hand side of the production refers to the composite object formed by the production. The derivation tree for a picture is equivalent to its composition graph.

The following informal definition of a picture grammar expresses this meaning:

Definition 3.7 *A* simple picture grammar *is a four-tuple* (N, Σ, s, P) *where* N *is a set of nonterminal symbols,* Σ *is a set of terminal symbols,* $s \in N$ *is an initial symbol and* P *is a set of productions of one of the following forms:*

- $X \rightarrow z$, $X \in N$, $z \in N \cup \Sigma$.
- $Y \rightarrow op(X_1, ..., X_p)$, $X_0 \in N$, $X_i \in N \cup \Sigma$, *where op is a composition operator.*

For a given visual language, the rules of the grammar specify which pictures belong to the language. If the grammar is unambiguous, then for each picture there will only be one derivation tree, corresponding to the intended structure of the picture. The next section illustrates how a grammar, compositions and a visual alphabet are used to specify the syntax of a visual language.

3.4 An Example - The StateCharts Language

In this section, an example is given of how picture grammars are used to specify the syntax of a visual language. The language defined is based on the StateCharts language [35].[3] StateCharts are an extension of finite state automata. StateCharts are interesting because they were developed as a visual programming language and are not based on any textual language. They exploit the two-dimensional nature of pictures to express concurrency in a natural fashion. StateCharts have been used for programming real-time reactive systems, such as embedded control systems.

The StateChart model is based on finite state automata. A finite state automaton (FSA) consists of a set of states and a set of transitions. Each state is identified by a label. A transition may be thought of as a triple $< s_0, e, s_f >$, meaning that in state s_0, if the next input is event e, move to state s_f. One state in the FSA is designated as the initial state, and one or more states are designated as final states. Finite state automata also form a visual language (e.g., by drawing states as circles and transitions as labelled arcs).

StateCharts enrich finite state automata both semantically and syntactically (i.e., visually). The basic model of computation is similar to that of finite state automata. A StateChart has a current state which changes in response to input events. StateCharts enhance the notion of state with two types of structure: depth and orthogonality. The StateChart language is developed below by specifying it with a simple picture grammar.

StateCharts use five different types of picture elements: text fragments, rectangles, circles, arrows and lines. Each class of picture element has an equivalent terminal symbol, so the grammar starts with:

$$\Sigma = \{\textbf{text}, \textbf{rectangle}, \textbf{circle}, \textbf{arrow}, \textbf{line}\}$$

Rather than listing N now, we will take it to be the union of the left hand sides of all the productions given, and show all nonterminal symbols in CAPITAL letters, while terminals are shown in **lowercase bold** letters.

The basic entity in a StateChart is a state. The simplest version of a state is a rectangle containing a text string, as shown in Figure 3.11. The picture is formed from two picture elements (the extent of the text string is shown as a dashed box). This is a simple atomic state, similar to a state in an FSA. An atomic state may also represent a more complex state which is left unspecified. An atomic state is described by the two productions:

(1) STATE → ATOMIC_STATE
(2) ATOMIC_STATE → contains(**rectangle**, **text**)

The second production says that an atomic state is formed by combining a **rectangle** with a piece of **text** using the *contains* composition operator. The *contains* operator composes two elements where the extent of the second element is contained

[3] I have simplified and present only a subset of the language defined by Harel.

Figure 3.11: An Atomic State

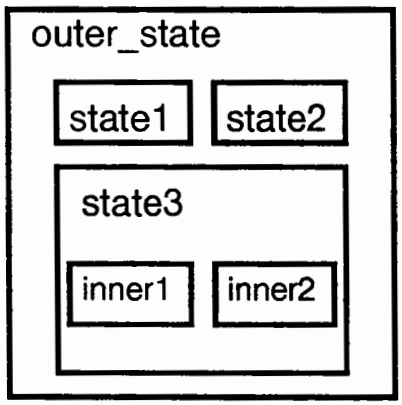


Figure 3.12: XOR-union of States

within the extent of the first.[4] The aggregate object will have the same extent as the outer element.

StateCharts extend atomic states with *depth*. A state can be formed from the union of a group of states. The containing state is the exclusive-OR (XOR) of the states within it. An FSA consists of a single XOR-union of atomic states. StateCharts extend this notion to allow states to be defined as unions to arbitrary depth.

This hierarchy of states is shown graphically by nesting the group of states within the containing state. Figure 3.12 shows a StateChart consisting of a state labelled *outer_state* which contains the XOR of three states: *state1*, *state2* and *state3*. *state3* in turn consists of the states *inner1* and *inner2*. The only spatial relationship required between states in an XOR-union is that they are non-overlapping. The label for the containing state must be above all the states in the XOR-union.

The productions defining the syntax of an XOR-union are:

(3) STATE → XOR_STATE
(4) XOR_STATE → contains(**rectangle**,XOR_INSIDE)
(5) XOR_INSIDE → over(**text**,XOR_UNION)
(6) XOR_UNION → STATE
(7) XOR_UNION → adjacent_to(XOR_UNION_1,XOR_UNION_2)

Productions six and seven form a group of STATEs using the *adjacent_to* operator, which composes two adjacent elements. The extent of the aggregate object is defined to be the smallest extent enclosing both of the constituents. The *over* operator in production five composes two adjacent elements where the bottom of the first is above the top of the second. This grammar does not necessarily specify a unique decomposition

[4] A further restriction of the contains operator is that there is nothing between the inner and containing elements. This is true any time we are composing objects which are adjacent in some form (but this is ignored for now). The specifics of this restriction are discussed in Chapter 5.

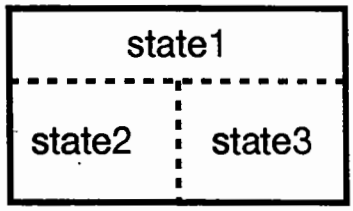


Figure 3.13: Orthogonal Product

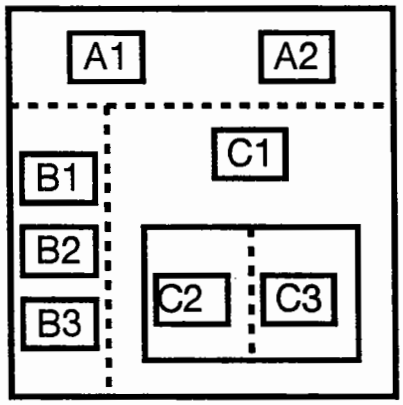


Figure 3.14: A Complex State

of the xor-union, but the decomposition is unimportant, since it is only the group that is interesting.

The second extension made by StateCharts is to allow a state to be decomposed into orthogonal components. This composition corresponds to the Cartesian product of states. When the StateChart is in the product state, it is also simultaneously in each of the contained states. Just as the previous grouping can be viewed as the OR of states, this product can be seen as the AND of a group of states.

The visual notation for the orthogonal product is to divide the containing state by dashed lines, either vertically or horizontally. Figure 3.13 shows an example of an orthogonal product of states. When in this state, the machine is simultaneously in *state1*, *state2* and *state3*. Note that no label is given to the product state. Orthogonal products can be combined with XOR-unions, as shown in Figure 3.14. The state defined by Figure 3.14 is an element of $(A1 \cup A2) \times (B1 \cup B2 \cup B3) \times (C1 \cup (C2 \times C3))$.

This visual syntax is defined by the following productions:

(8) STATE → PRODUCT_STATE
(9) PRODUCT_STATE → contains(**rectangle**, PRODUCT_INSIDE)
(10) PRODUCT_INSIDE → horiz(PRODUCT_INSIDE$_1$,VBAR,PRODUCT_INSIDE$_2$)
(11) PRODUCT_INSIDE → vert(PRODUCT_INSIDE$_1$,HBAR,PRODUCT_INSIDE$_2$)
(12) PRODUCT_INSIDE → **text**
(13) PRODUCT_INSIDE → XOR_UNION
(14) VBAR → dashed_vertical(**line**)
(15) HBAR → dashed_horizontal(**line**)

The interpretations of productions ten and eleven are shown graphically in Figure 3.15. Production ten composes PRODUCT_INSIDEs which are horizontally aligned and separated by a vertical dashed line. Similarly, production eleven composes PRODUCT_INSIDEs which are vertically aligned and separated by a horizontal dashed line. In this fashion, an orthogonal product is built up from horizontal and vertical compositions of states. Again, these productions do not necessarily give a unique decomposition

Figure 3.15: Productions for an Orthogonal Product State

of the PRODUCT_STATE, but they suffice since only the group is important. Productions 14 and 15 use unary compositions to specify the syntax for vertical and horizontal dashed lines.

Similar to finite state automata, StateCharts have transitions defined on input events. The visual notation for a transition is a labelled arc between states, as shown in Figure 3.16. The label is a text string specifying the input event. The natural way to fit a transition into the picture is to compose it with the state it leaves. This syntax for transitions is specified by the productions:

(16) TRANSITION → labels(**text**,**arrow**)
(17) STATE → leaves(TRANSITION, STATE_1)

Production 16 defines a TRANSITION as a composition of an **arrow** and the **text** labelling it. The *labels* operator corresponds to spatial adjacency of a shape and a line. Production 15 relates the TRANSITION to the state it leaves. The *leaves* operator combines a line and a shape, where the first endpoint of the line is somewhere on the boundary of the shape. The aggregate will have the extent of the shape rather than the endpoints of the line.

Finally, StateCharts makes two extensions to allow transitions to work with the hierarchy. For example, consider the statechart shown in Figure 3.17, which has a transition from state A to state B on event $E1$. Since state B is the XOR-union of states B_1 and B_2, which of these states is to be entered must be specified. The first extension is the use of *default states*. The default state specifies which state is to be entered when a group is entered. It is indicated visually by a unlabelled arrow starting

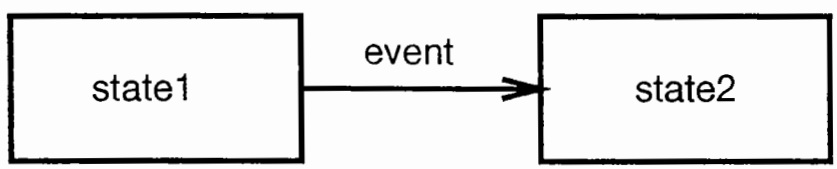


Figure 3.16: A StateChart Transition

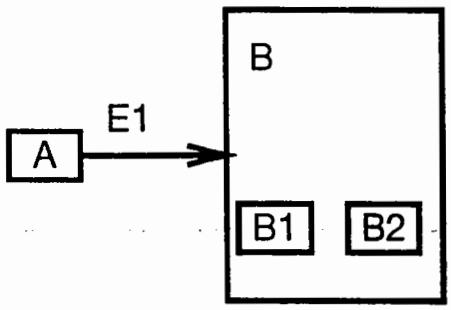


Figure 3.17: Underspecified Transition

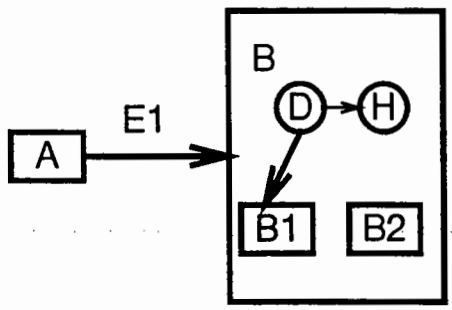


Figure 3.18: Default and History States

at an circle containing the letter 'D' and pointing to the default state. A default state is analogous to the start state of a finite state automata.

The second extension is to add memory to an XOR-union. A circle containing the letter 'H' represents the most recently visited state. A transition ending at the history symbol signifies a transition to whatever state was most recently visited. Figure 3.18 shows a StateChart with both history and default states. The default state has two arrows - one to the history and the other to state $B1$. This signifies that the default state is initially state $B1$ and then is the most recently visited state.

The productions to define the visual syntax of default and history states are:

(18) DEFAULT → DEFAULT_STATE
(19) DEFAULT → leaves(**arrow**,DEFAULT_STATE)
(20) DEFAULT_STATE → leaves(**arrow**,DEFAULT_SYMBOL)
(21) DEFAULT_SYMBOL → contains(**circle**,**text**)
(22) HISTORY → contains(**circle**,**text**)
(23) XOR_UNION → DEFAULT
(24) XOR_UNION → HISTORY

A DEFAULT is specified as an **circle** containing a 'D' with one or two arrows leaving it. A HISTORY is specified as a **circle** containing an 'H'. Productions 23 and 24 are used to include the default and history symbols into the XOR-union.

The specification of the StateChart language is completed with the production,

(25) CHART → STATE

which defines a StateChart to be a STATE (i.e., CHART is the start symbol). Figure 3.19 shows an example of StateChart. The derivation tree (or equivalently, the composition graph) for this StateChart is given in Figure 3.20. The interior nodes of the parse tree are labelled with nonterminal symbols and correspond to the aggregates formed by applying the compositions specified by the productions. The leaf nodes are labelled by the terminal symbols forming the picture.

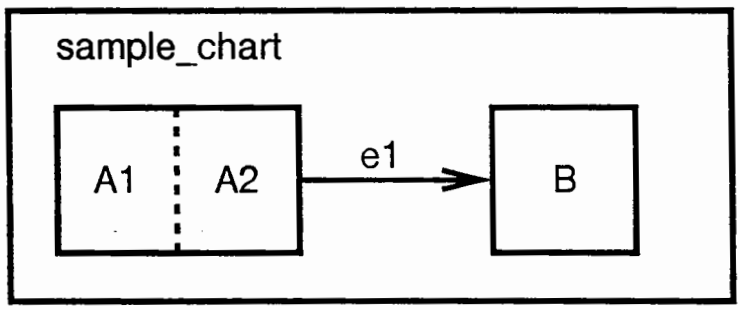


Figure 3.19: A Sample StateChart

3.5 Summary

The grammar given in the previous section illustrates how visual language syntax can be defined, but it has several problems. First of all, some productions needed to be more specific about the terminal symbols involved. For example, the text in Production 21 must be restricted to the string "D". Also, in Productions 10 and 11, lines which were dashed and vertical or horizontal are required. This was accomplished by using special unary composition operators to enforce the restriction. Similarly, operators could be defined which only apply to the text strings "D" and "H". This approach requires extending the set of production operators for every new language, however. Alternatively, the problem could be solved by introducing additional terminal classes (e.g., **vert_dashed_line**). Doing so, however, embeds the structure of the language into the representation of the pictures.

Another problem is that an xor-union should contain at most one default state. One solution would be to introduce extra nonterminals and productions to distinguish whether an XOR_UNION contains a default state. This approach leads to very cumbersome grammars, however. For example, when history states are added, distinctions must be made between XOR_UNIONS which contain a default state, a history state, both or neither. A better approach to problems like this is described in Chapter 5.

A third problem is demonstrated by the transitions. The informal description of the language said that a transition had to begin and end at a state. This grammar only composes a TRANSITION with the STATE at which it begins. The other end of a TRANSITION is not restricted to point to a STATE, as the language definition requires. A more subtle problem is the fact that the order that the transitions leaving a state are composed with the state is not specified.

These problems are dealt with by the mechanism developed in the next two chapters. In Chapter 4, the *attributed multiset grammar* is defined to provide a formal model for specifying the syntax of visual languages. Chapter 5 describes the particular class of attributed multiset grammars used to define visual languages, the *Picture Layout Grammar*.

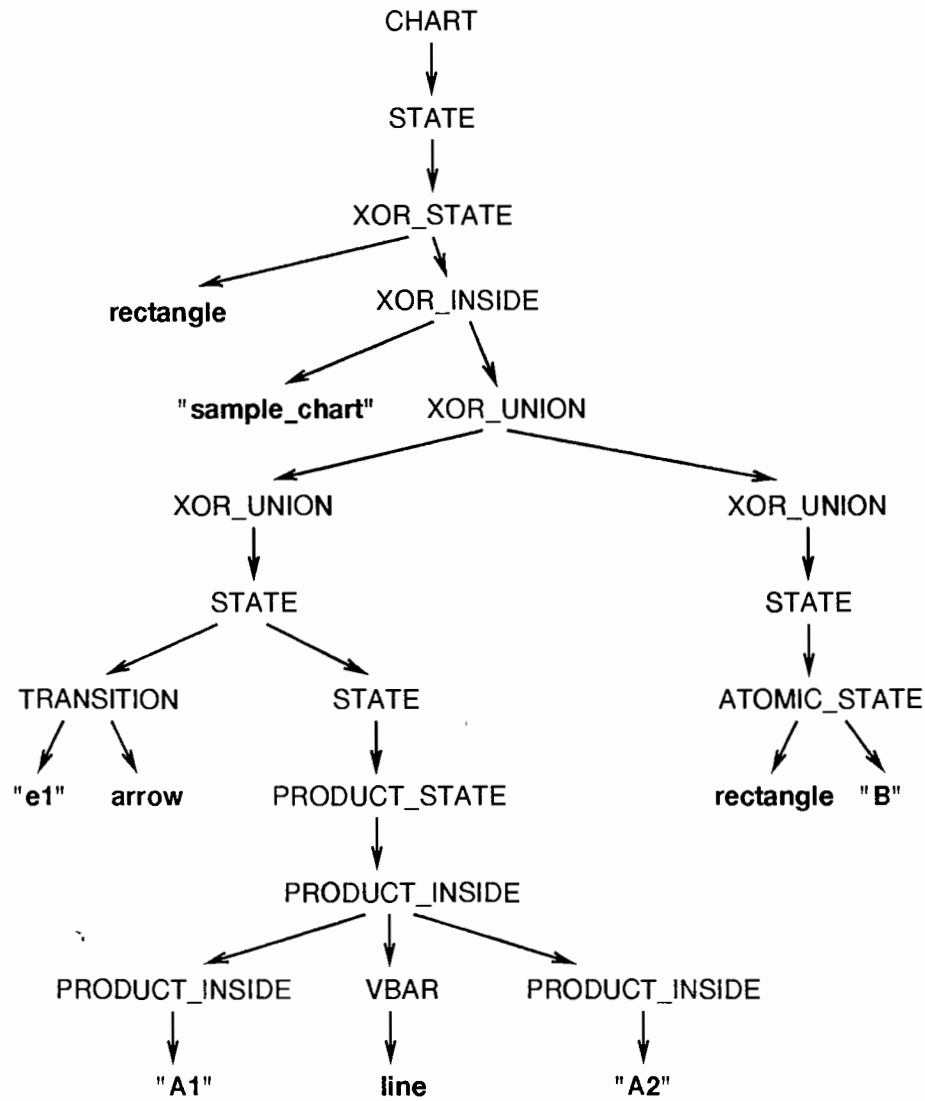


Figure 3.20: Parse structure for StateChart

Chapter 4

The Grammar Model

4.1 Introduction

This chapter describes the syntactic theory of visual languages. A syntactic theory describes a family of languages with similar characteristics. The family of languages considered here are visual languages.

A language is a set of structures called words formed from symbols. Words are typically viewed as strings of symbols, and languages are sets of strings. A string has two properties of interest: it is a collection of symbols and the symbols are ordered. The elements of a visual language are two dimensional pictures rather than strings, so I have developed a model of language whose elements are unordered collections of symbols.

A language is often characterized by means of a grammar. A grammar is a type of rewriting system, and consists of a set of symbols V, a set F of ordered pairs (P, Q) of words over V and rules indicating how rewriting is done [77]. For example, a context-free grammar is a rewrite system where V consists of two distinct sets V_t and V_N and each of the pairs has $P \in V_N$ and Q a string of elements of V. The language characterized by a context-free grammar is taken to be all those strings over V_t which can be generated from a distinguished symbol $S \in V_N$ by repeated rewriting.

Grammars have several advantages as a language definition mechanism. First, a grammar allows a finite specification of an infinite language. Secondly, grammars can serve as the basis of a recognition tool. Lastly, grammars also provide a structural characterization of a language, which can be used as a basis for the semantics of the language.

Grammars may be considered from either an analytical or generative point of view.

This chapter describes the theory of visual language syntax. First, the unordered collections which form the components of visual languages are described, and then the grammars used to specify these languages are defined.

4.2 Symbols, Words and Attributes

A language is defined over a finite set of symbols called an *alphabet.* The next chapter demonstrates how these symbols correspond to the primitive graphical elements of a visual language. For the purpose of this discussion it suffices to merely consider them as symbols.

An *attributed alphabet* is a triple

$$\Sigma = (\Sigma', A, D)$$

where Σ' is a finite set of symbols, A is a finite set of *attributes*, and D is a set of *attribute domains.* The attributes are used to describe properties of the symbols. The attributes are partitioned into disjoint sets A_σ for each $\sigma \in \Sigma'$, with

$$A = \bigcup_{\sigma \in \Sigma'} A_\sigma \text{ and } A_{\sigma_1} \cap A_{\sigma_2} = \emptyset \text{ for } \sigma_1 \neq \sigma_2$$

For each symbol $\sigma \in \Sigma'$, *the attributes of* σ, refers to the attributes in A_σ unless it is specifically indicated that all the attributes in A are referred to.

The attribute domain set $D = \{D_i\}$ is a set of domains for the values of the attributes. D_i is the domain for attribute $a_i \in A$, and $|D| = |A|$. Where the domain set is clear or unimportant, it will be left unspecified and an attributed alphabet is treated as the pair (Σ', A).

Given an attributed alphabet Σ as described above, an *attributed symbol* is a pair (σ, V) where $\sigma \in \Sigma'$ is a symbol and $V = \{v_i\}$ $\forall a_i \in A$ is an attribute vector. If $A_\sigma = \{A_{i_1}, A_{i_2}, ..., A_{i_m}\}$ are the attributes of σ, then $v_i = \perp$ if $a_i \notin A_\sigma$ and $v_i \in D_i \cup \perp$ $\forall a_i \in A_\sigma$. The attribute vector V contains values for all the attributes, with $\perp$ signifying an undefined value (i.e. an attribute not defined for σ). The function PROJ is defined by $\text{PROJ}((\sigma, V), a_i) = v_i$ as given above.

The domain of the attribute vector is given as $D^* = \prod_{a_i \in A}(D_i \cup \perp)$. The notation $\sigma[a_1 : v_1, ..., a_m : v_m]$ is used for an attributed symbol, which may be shortened to only include those attributes $a_i \in A_\sigma$. The attribute names (a_i :) may also be left off. If the attributes are not of concern, the symbol may be referred to just as σ.

Example 4.1 *The triple* $\Sigma = (\{a, b, c\}, \{i, j, k\}, \{Z, Z.Z\})$ *is an attributed alphabet, with* $A_a = \{i\}, A_b = \{j\}$, *and* $A_c = \{k\}$. *An example of an element of* Σ *would be* $a[i : 3, j : \perp, k : \perp]$. *Other notations for this element would be* $a[i : 3]$, $a[3]$ *or* a.

The structures forming the elements of the family of languages are unordered collections of symbols. The concept of a multiset [46] is used for a collections. Intuitively, a multiset is like a set, except that it may contain identical elements which are repeated. Another way of thinking of a multiset is as a set of objects, where each object is a unique instance of a symbol class. The formal definition of a multiset is given by:

Definition 4.1 *A* multiset M *defined over an alphabet* Σ *is a function mapping* Σ *to the non-negative integers. For* $x \in \Sigma, M(x)$ *is the number of occurences of* x *in the multiset* M.

Although formally defined in terms of a function, a multiset will usually be specified by listing its members as $\{m_1, m_2, ..., m_n\}$. (The distinction between a set and a multiset will always be implied by the context). Given a set S, S^* is defined to be the set of all multisets over S.

Example 4.2 *Let $\Sigma = \{a, b, c\}$ be an alphabet. Then an example of a multiset over Σ would be $M = \{a, b, a, b, c, b\}$ (which is the same as the multiset $M' = \{a, a, b, b, b, c\}$). This multiset would be defined by the function $M(a) = 2$, $M(b) = 3$, $M(c) = 1$.*

Given two multisets A and B over the set S, define the following operations $\forall x \in S$:

$A \uplus B = M$, where $M(x) = A(x) + B(x)$.

$A \cup B = M$, where $M(x) = \max(A(x), B(x))$.

$A \cap B = M$, where $M(x) = \min(A(x), B(x))$.

$A \ominus B = M$, where $M(x) = \max((A(x) - B(x)), 0)$.

$\alpha \in A$ if and only if $A(\alpha) > 0$.

$\emptyset$ is the multiset given by $\emptyset(x) = 0$.

$\text{rank}(A) = \max_{x \in S}(A(X))$.

The relation $A = B$ holds if $A(x) = B(x)$.

The relation $A \subseteq B$ holds if $A(x) \leq B(x)$.

Now the notion of a multiset is extended to include attributed symbols. Intuitively, an attributed multiset is an unordered collection of attributed symbols. An attributed multiset may be viewed as a collection of objects, where the symbol is the *class* of the object, and the attribute values are the instance variables of the object.

Definition 4.2 *An* attributed multiset *M defined over an attributed alphabet $\Sigma = (\Sigma', A, D)$, is a function mapping $\Sigma \times D^*$ to the non-negative integers. For an attributed symbol (σ, V), $M(\sigma, V)$ is the number of occurences of (σ, V) in the attributed multiset M.*

As with an unattributed multiset, an attributed multiset is represented by listing the elements (rather than as a function). An example of an attributed multiset would be M_1 = {a[1],a[2],b[1],b[2],b[2],c[3]}.

The operations defined above for multisets are extended to attributed multisets in the natural manner. For an attributed multiset M, the underlying unattributed multiset M' will be called the *base multiset.* The base multiset is defined by

$$M'(\sigma) = \sum_{V \in D^*} M(\sigma, V)$$

For the attributed multset M_1 given above, the base multiset is $\{a, a, b, b, b, c\}$.

The term *(attributed) multiset language* is used to mean any set of (attributed) multisets over a specific (attributed) alphabet. Multiset languages may be referred to as languages when it is clear from context that they are multiset languages. *Word* and *sentence* are also used as synonyms for multiset (either attributed or not) when referring to language elements.

Example 4.3 *Let $\Sigma = \{a, b\}$ be an alphabet and let L be the (unattributed) multiset language $L = \{M \mid M(a) = k, M(b) = 2k, k > 0\}$. L includes $\{a, b, b\}$, $\{a, a, b, b, b, b\}$ and all multisets consisting of twice as many b's as a's. Now let $\Sigma' = (\Sigma, \{a_1, a_2\})$ be an attributed alphabet and let L' be the attributed multiset language over Σ' given by $L' = \{M' \mid M'(a, (i, \perp)) = 1, M'(b, (\perp, j) = 1 \forall 1 \leq i \leq k, 1 \leq j \leq 2k, k > 0$ and $M = 0$ elsewhere. L' contains $\{a[1], b[1], b[2]\}$, $\{a[1], a[2], b[1], b[2], b[3], b[4]\}$ and so on.*

4.3 Multiset Grammars

Specifying languages as sets is not very useful. Grammars provide both a means of finitely defining infinite sets and a structural characterization of the language elements. Grammars also form the basis of a analysis technique. This section first defines grammars for (unattributed) multiset languages, and then extends these definitions to attributed multiset languages. Now the first grammar definition:

Definition 4.3 *A* multiset grammar *is a four-tuple,*

$$G = (N, \Sigma, S, P)$$

where

- *N is a finite set of non-terminal symbols.*
- *Σ is a finite set of terminal symbols.*
- *$V = N \cup \Sigma$ is an alphabet called the vocabulary.*
- *$S \in N$ is a start symbol.*
- *P is a set of productions of the form $\mathrm{A} \rightarrow M$, where $M = \{X_1, ..., X_r\}$, with $X_i \in V$, $A \in N$ and $M \neq \emptyset$.*

This is similar to the definition of a context-free grammar, except that the elements of the right hand side of a production are considered a multiset rather than a string.

Although the right-hand side of a production is an unordered multiset, for any grammar G there is a fixed *canonical ordering* for each production $p \in P$ which orders the elements of the right-hand side. The canonical ordering is used to distinguish between elements of the RHS (e.g. for relating a particular element of the RHS to specific child in a derivation tree, as discussed later). The actual ordering used is

arbitrary and is unimportant as long as it is fixed for a grammar. By convention, the right hand side of a production is listed from left to right in canonical order.

NOTATION
The following conventions regarding grammars are used:

- The capital letters A, B, C, D, E and S will denote non-terminal symbols.
- The lower-case letters a, b, c, d and e, and boldface strings such as **box** will denote terminal symbols.
- The capital letters W, X, Y and Z will denote symbols which may be either terminals or non-terminals.
- The capital letters M and N, will denote multisets of terminal and non-terminal symbols.
- The lower-case letters u, v, w, x, y and z will denote either multisets or strings of terminal symbols, depending on the context.

Derivation and the language generated by a grammar G are defined as follows. If $A \rightarrow \alpha$ is a production in P, and $M \in V^*$ is a multiset with $M(A) = i_A, i_A > 0$ (i.e. M contains at least one occurence of A), then $M \Rightarrow_G M'$ (i.e. M *directly derives* M'), where $M' = (M \ominus \{A\}) \uplus \alpha$. That is, the production $A \rightarrow \alpha$ applied to the multiset M (where $A \in M$), derives a new multiset where one occurence of A has been replaced by all the occurences of symbols in α.

If $M_1, M_2, \ldots, M_n$ are multisets in V^*, with $n \geq 1$, and

$$M_1 \Rightarrow_G M_2, M_2 \Rightarrow_G M_3, \ldots, M_{n-1} \Rightarrow_G M_n$$

then $M_1 \overset{*}{\Rightarrow}_G M_n$, or simply M_1 *derives* M_n. That is, $\overset{*}{\Rightarrow}_G$ is the reflexive and transitive closure of $\Rightarrow_G$.

The *multiset language generated* by G (denoted $ML(G)$) is $\{w \mid w \in \Sigma^*$ and $S \overset{*}{\Rightarrow}_G w\}$. That is, $ML(G)$ is the set of all multisets containing only terminal symbols that can be derived from S.

Example 4.4 *Let G be the grammar with $N = \{S\}$, $\Sigma = \{a, b\}$ and P given by $S \rightarrow \{a, b\}$, $S \rightarrow \{S, a, b\}$. This grammar generates the multisets $\{a, b\}$, $\{a, a, b, b\}$ and so on. The multiset language generated by this grammar is $ML(G) = \{M \in \Sigma^* \mid M(a) = M(b) = k, k > 0\}$. That is, G generates multisets with equal numbers of a's and b's.*

The derivation of a multiset may be viewed as a tree where the leaf nodes are labelled by the members of the multiset and the interior nodes are labelled by the non-terminals which appeared on the left-hand side of productions used in the derivation.

Definition 4.4 *Let V be an alphabet. A* labelled, ordered tree *over V (or more simply* tree*) is a triple (N, E, L) where N is a set of nodes, L is a labelling function mapping $N \rightarrow V$ and E is an ordered list of edges $E = e_1, ..., e_r \mid e_i \in N \times N$ such that the following hold:*

1. *There is exactly one node $r \in N$, called the* root, *such that there are no edges $(w, r) \in E$.*
2. *For all nodes $v \in R, v \neq r$, there is exactly one edge $(w, v) \in E$.*
3. *A* path *(from v_1 to v_s) is defined to be a set of edges $(v_1, v_2), (v_2, v_3), ..., (v_{s-1}, v_s)$. There is no path in E with $v_s = v_1$ (i.e. there are no cycles in the edges).*

For any node $n \in N$, if there are no edges $(n, w) \in E$, then n is a leaf node. *Otherwise n is an interior node. Let $e_{i_1}, ..., e_{i_m}$ be all the edges $e_i = (n, w_i) \in E$, with $i_1 < i_2 < ... < i_m$ (i.e. the edges are ordered by the list E). Then $w_{i_1}, ..., w_{i_m}$ are the* children *of n, and the* j'th child *of n is w_{i_j}.*

Given a tree $T = (N, E, L)$ and two nodes $v, w \in N$, if there is a path from v to w, then v is an *ancestor* of w and w is a *descendant* of v. A path from v to v is called a *cycle*. A tree $S = (N', E', L')$ is a *subtree* of tree $T = (N, E, L)$ (written $S \subseteq T$) if $N' \subseteq S, E' \subseteq E$ and $L'(v) = L(v)\ \forall v \in N'$. Now a definition for the set of derivation trees for a multiset grammar:

Definition 4.5 *Given a grammar $G = (N, \Sigma, s, P)$ as defined above. Then $J_G = \{P \cup \Sigma\}$ is the* label set *for G. For a label $l \in J_G$, the* label symbol *for l is defined by*

$$lsymb(l) = \begin{cases} l & \text{if } l \in \Sigma \\ A & \text{if } l \in P \wedge l = A \rightarrow M \end{cases}$$

Now let $\sigma \in V$ be a symbol of the grammar. Then a derivation tree *for σ in G is a labelled, ordered tree $T = (N, E, L)$ over the label set J_G, where:*

1. *Every node $v \in N$ has a label $l \in J_G$.*
2. *The label of the root of T is l, where $lsymb(l) = \sigma$.*
3. *If n is an interior node with label $l = A \rightarrow \{X_1, X_2, ..., X_k\}$ (where the numbering $1, ..., k$ corresponds to the canonical ordering of the production), then n has children $n_1, n_2, ..., n_k$ with labels $l_1, l_2, ..., l_k$, where $lsymb(l_i) = X_i$, respectively.*

The set of all derivation trees for σ is defined to be

$$\mathcal{T}_\sigma^G = \{T \mid T \text{is a derivation tree for } \sigma\}$$

The *yield* of a derivation tree T is the multiset formed from the leaf nodes of T. The *string yield* of a derivation tree T is the string formed by taking the leaves of T from left to right, where left to right corresponds to the canonical ordering within the productions labelling the nodes. This is also called the *frontier* function [89] ($fr(T)$).

4.3.1 Context-Free Multiset Languages

Definition 4.6 *A multiset language L is called a* context-free multiset language *if there is a multiset grammar G such that $L = ML(G)$.*

The reason for calling the languages generated by multiset grammars context-free is made clear by the following discussion of the relationship between multiset grammars and context-free grammars.

Let Φ be the homomorphism from strings to multisets defined by

$$\Phi(w) = M : \Sigma \rightarrow N \text{ given by } M(a) = \text{ the number of occurences of } a \text{ in } w$$

This homomorphism removes the ordering information from the string, leaving a multiset.

Theorem 4.1 *Let L_M be a context-free multiset language. Then L_M is the homomorphic image of some context-free string language L_S.*

Proof: Let G be a multiset grammar for L_M. Simply use the canonical ordering for every production to order the symbols on the RHS into a string, resulting in a context-free (string) grammar G_S generating a context-free string language L_S. Since every derivation tree over G has a corresponding derivation tree over G_S, it is clear that any multiset $M \in L_M$ has a corresponding string $S \in L_S$ for each derivation tree of M. Similarly, for any string in L_S, there is a multiset in L_M with the corresponding derivation tree. Thus,

$$\begin{aligned} \forall M \in L_M \quad \exists \quad S \in L_S \text{such that} M = \Phi(S) \\ \forall S \in L_S \quad \exists \quad M \in L_M \text{such that} M = \Phi(S) \end{aligned}$$

□

The frontier function $fr(T)$ is clearly a homomorphism from (labelled, ordered) trees to strings. This is used to reverse the relationship in Theorem 4.1

Definition 4.7 *The* ordered string set *generated by a multiset grammar G is given by $OSS(G) = \{x \in \Sigma^* \mid x = fr(T) \text{for some} T \in \mathcal{T}_s^G\}$.*

Theorem 4.2 *Let L_S be a context-free string language. Then there is a multiset grammar G for which L_S is the ordered string set generated by G.*

Proof: Let G be a context-free (string) grammar generating L_S. Now construct the grammar G_M from G by viewing the right-hand sides of productions as multisets and using the string ordering of the RHS to determine the canonical ordering. Clearly $L_S = OSS(G_M)$. □

These two theorems show that, as expected, multiset grammars and context-free string grammars are closely related. Removing the ordering from a context-free string language results in a context-free multiset language. By restoring the ordering implied by the syntactic structure to a context-free multiset language, a context-free string language is created.

4.3.2 Extended Multiset Grammars

As noted in Chapter 3, a hierarchical structure such as a derivation tree do not preserve all the two-dimensional adjacency relationships present in an element of a visual language. It is desirable to extend multiset grammars to more fully reflect these relationships.

The approach taken here is to allow a *context* to be specified for each production. This context consists of a multiset of terminals which must be present for the application of a production to be valid within a derivation tree. When attributed multiset grammars are developed, it will be apparent how to restrict the context to a particular terminal. For now it will suffice to allow any instance of the specified symbol as the context.

Providing a context for a production can be viewed as restricting the structure of the derivation trees. Any derivation tree containing a node labelled with that production must also contain leaf nodes labelled by the symbols in the context. A context can also be viewed as augmenting the derivation tree, by adding an additional edge for each symbol in a context, from the node labelled by the production to an (arbitrarily chosen) nodes labelled by the context symbol.

Adding context to productions has two advantages:

1. The context will allow us to more precisely specify visual language syntax. For example, one can specify that an arrow must be pointing at a box. One could also use the context to distinguish between an arrow pointing to a box and an arrow pointing to a circle.

2. The additional edges in the parse tree will allow attribute information to pass from one part of the tree to another.

Now a formal definition of the extension is given:

Definition 4.8 *An* extended multiset grammar *is a four-tuple,* $G = (N, \Sigma, S, P)$ *where*

- N *is a finite set of non-terminal symbols.*
- Σ *is a finite set of terminal symbols.*
- $V = N \cup \Sigma$ *is an alphabet called the vocabulary.*
- $S \in N$ *is a start symbol.*
- P *is a set of productions of the form* $\mathrm{A} \rightarrow M/\alpha$, *where* $M = \{X_1, ..., X_r\}$ *and* $\alpha = \{a_1, ..., a_s\}$, *with* $X_i \in V, a_i \in \Sigma$ *and* $A \in N$. α *is the* context *of the production.*

Given an extended multiset grammar $G = (N, \Sigma, S, P)$, the underlying multiset grammar G' is given by (N, Σ, S, P'), where P' are the productions of P with the contexts removed and redundant productions eliminated. Next the definition of tree is extended to include the extra edges:

Definition 4.9 *Let V be an alphabet. A* labelled, ordered tree-DAG *over V (or simply tree-DAG) is a quadruple $TD = (N, E, L, F)$, where $T = (N, E, L)$ forms a labelled, ordered tree, and F is an ordered list of edges $F = f_1, ..., f_s \mid f_i \in N \times N$ and such that if $(v, w) \in F$, then w is a leaf node.*

The tree T is the *basis tree* of the tree-DAG. The edges in E are called *tree-edges (or Tedges)* and the edges in F are called *graph-edges (or Gedges).* For a node v, the nodes $w_1, ..., w_k$ such that $(v, w_i) \in E$ are the *tree-children* of v and the nodes $z_1, ..., z_k$ such that $(v, z_i) \in F$ are the *graph-children* of v.

The tree-DAG definition is used to extend the notion of a derivation tree to handle contexts.

Definition 4.10 *Let $G = (N, \Sigma, s, P)$ be an extended multiset grammar with G' the underlying multiset grammar. A multiset M is* analyzable *over G if there exists a tree-DAG TD over the label set of G with a basis tree T such that the following hold:*

- $T \in \mathcal{T}_s^{G'}$.
- *M is the yield of T*
- *If v is an interior node in TD labelled by $A \rightarrow \{X_1, ..., X_k\}/a_1, ..., a_j$, then v has tree children $w_1, ..., w_k$ and graph children $z_1, ..., z_j$, where tree child w_i is labelled l_{w_i} with $lsymb(l_{w_i}) = X_i$ and graph child z_i is labelled a_i.*

The tree-DAG TD is called an analysis *of M.*

Now the language specified by an extended multiset grammar can be defined in terms of the analyzable multisets. Given an extended multiset grammar $G(N, \Sigma, s, P)$, the multiset language recognized by G ($\mathcal{A}(G)$) is given by

$$\mathcal{A}(G) = \{M \in \Sigma^* \mid M \text{ is analyzable over } G\}$$

Example 4.5 *Let $G = (\{S, A, B\}, \{a, b\}, S, P)$ where P contains the following three productions:*

$$\begin{aligned} S &\rightarrow \{A, B\}/\emptyset \\ A &\rightarrow \{a\}/\{b\} \\ B &\rightarrow \{b\}/\{a\} \end{aligned}$$

This extended multiset grammar defines the language consisting of the single multiset $M = \{a, b\}$.

Note that this definition of the language specified by an extended multiset grammar is an analytical one, and not a generative one (thus the use of the term analysis rather than extended derivation tree). To see the distinction, consider extending the generative approach used for multiset languages by saying that $M \Rightarrow_G M'$ only when

$M(a) > 0\ \forall a \in$ the context for the production applied. Using this definition, the language generated by the grammar in Example 4.5 is empty, so $ML(G) \neq \mathcal{A}(G)$.

An interesting question is how the languages recognized by an extended multiset grammar relate to the context-free multiset languages. A similar question for string grammars is the relationship between the languages recognized (not generated) by context-sensitive grammars and context-free (string) grammars. Peters and Ritchie [61] have shown that any language that is recognized by a context-sensitive grammar is context-free. The following analogous result for multiset grammars and extended multiset grammars is proven.

Theorem 4.3 *If $G = (N, \Sigma, S, P)$ is an extended multiset grammar and $L = \mathcal{A}(G)$ is the multiset language recognized by G, then L is a context-free multiset language.*

Proof: We will construct a context-free multiset grammar $H = (N', \Sigma, \sigma, P')$ which generates the language $\mathcal{A}(G)$. First, number the elements of 2^{Σ} from 0 to n, with $\emptyset$ numbered 0 and set i called Σ_i. For each nonterminal symbol $A \in N$, some new nonterminals $A_{i,j}$ are constructed, where $0 \leq i, j \leq n$. The indices on the nonterminal are going to refer to subsets of Σ. Theses indices are used to remember two sets during parsing: the set of terminal symbols generated by the current nonterminal, and the union of all the contexts encountered in that generation. New productions are constructed that "compute" these sets for the left hand side from the right hand side.

First, set $N' = \cup_{A \in N} A_{0,0}$, and set $P' = \emptyset$. Next, for every production of the form $A \rightarrow a$ (where $a \in \Sigma$), let i be the number of the set $\{a\}$, then add a nonterminal $A_{i,0}$ to N' and the production $A_{i,0} \rightarrow a$ to P'. Continue to add nonterminals and productions using the following procedure:

1. for each production $p \in P$ do the following:
   (a) let $p = C \rightarrow \{X_1, ..., X_k\}/\{b_1, ..., b_l\}a$.
   (b) let $A_{o_1}, ..., A_{o_s}$ be the nonterminals in $X_1, ..., X_k$
   (c) let $a_{q_1}, ..., a_{q_t}$ bet the terminaals in $X_1, ..., X_k$
   (d) for each A_o, let $A'_o = \{(A_o)_{i,j} \in N'\}$
   (e) for every combination selecting one element from each A'_o:
      i. create the nonterminal $C_{x,y}$. Here x is the number of the set formed by combining all the sets specified by the first index of each of the selected $(A_o)_{i,j}$'s and unioning that with $\{a_q\}$. Similarly y is the number of the set formed by combining all the sets specified by the second index of the selected $(A_o)_{i,j}$'s and unioning that with $\{b_1, ..., b_l\}$.
      ii. add $C_{x,y}$ to the set N'.
      iii. add the production $C_{x,y} \rightarrow \{X'_1, ..., X'_k\}$ to the set P', where $X'_i = X_i$ifX_i is a terminal and X'_i is the selected $(A_o)_{i,j}$if$X_i = A_o$.
2. Repeat 1 until no new productions are added in step 1(e)iii.

3. Add a new unique nonterminal σ to N'

4. for each nonterminal $S_{i,j} \in N'$, such that $\Sigma_j \subseteq \Sigma_i$, add a production $\sigma \rightarrow S_{i,j}$ to P'.

Clearly there can be at most n^2 new nonterminals for each existing nonterminal and at most n^{2k} new productions for each existing production of length at most k. Thus this is an effective procedure for computing a grammar. Our claim is that $ML(H) = \mathcal{G}$. This is proven in two parts.

First, let $M \in ML(H)$. Then there is a derivation tree T_H for M. Now construct a new tree T by relabelling each node in T_H labelled by production $p' \in P'$ by any of the productions $p \in P$ that caused p' to be added. After all the nodes have been relabelled, delete the root node σ and the edge leaving it. Clearly if G' is the underlying multiset grammar for G, then $T \in \mathcal{T}_S^{G'}$ (ignoring the context in each production), M is the yield of T. So it is merely neccessary to extend T into a tree-DAG by adding edges from each node with a nonempty context to the terminal symbols in the context. This can be done as long as M contains all of the symbols found in contexts.

Let $a \in \Sigma$ be mentioned in the context of production p at a node n in T. Let n' be the corresponding node in T_H, labelled by production p' with $A_{i,j}$ the left-hand side of p'. From step 1(e)i, we know that $a \in \Sigma_j$. Furthermore, since at each step the second index of the parent node is formed from the union of all its children, all ancestors of n (except σ) will have $a \in \Sigma_j$, where j is the second index of the ancestor node. Now let $S_{i,j}$ be the child of the root σ. Since $a \in \Sigma_j$ and $\Sigma_j \subseteq \Sigma_i$, we know $a \in \Sigma_i$. But, at each step, the first index is formed from the union of all the children of a node, so either $S_{i,j}$ has a child which the terminal symbol a, or it has a nonterminal child $A_{i',j'}$ with $a \in \Sigma_{i'}$. By induction on the height of the tree T_H, we see that somewhere there must be a node which has the terminal symbol a as its offspring. Thus we know that M will contain every terminal occuring in a context of a production in T, and T can be extended to become an analysis for M. Therefore $M \in \mathcal{A}(G)$.

Now assume $M \in \mathcal{A}(G)$. Then there is a tree-DAG TD that is an analysis of M. We will construct a new tree T_H that is a derivation tree for M over H. First, remove all the graph-edges from TD, leaving a tree. Next, do a bottom-up traversal of the tree. At every node, two things are computed: the set of terminal symbols forming the yield of the subtree rooted at the node; and the set of terminal symbols occuring in contexts in the subtree. Finally, add a new node labelled σ with a single edge to the root of TD.

Clearly, if all the productions found in T_H are in P', then T_H is a derivation tree for M. We can show that all the productions labelling nodes in T_H were added to P' by induction on the height of node n in T_H. First let n be any node of height 1, with the corresponding node in TD labelled by production $A \rightarrow \{a_1, ..., a_k\}/\{b_1, ..., b_l\}$. Clearly, n is labelled by the production $A_{i,j} \rightarrow \{a_1, ..., a_k\}$, where $\Sigma_i = \{a_1, ..., a_k\}$ and $\Sigma_j = \{b_1, ..., b_l\}$. Furthermore, it is clear that this production will be added to P' by step 1(e)iii.

Now suppose that for every node $m, h(m) < h_0$, the production labelling m was added to P' and that $h(n) = h_0$. If n is labelled by $p = \sigma \rightarrow S_{i,j}$ then clearly p was

added to P', since we must have $\Sigma_j \subseteq \Sigma_i$. Otherwise n is labelled by production $p = A_{i,j} \rightarrow \{X_1, ..., X_k\}$ with the corresponding production $A \rightarrow \{X_1, ..., X_k\}/\{b_1, ..., b_l\}$ in TD. Since the productions labelling each of the children of n were added to P', step 1e must eventually select the symbols labelling the children of n. Thus the production p will also be added to P'. By induction, we see that all the productions labelling nodes in T_H will be added to P'. Therefore T_H is a valid derivation tree for M and $M \in ML(H)$. □

Example 4.6 *Consider the grammar G in Example 4.5. Our procedure would define a new grammar $H = (N', \{a, b\}, \sigma, P')$ where*

- *We number the elements of 2^Σ as follows: $\Sigma_0 = \emptyset, \Sigma_1 = \{a\}, \Sigma_2 = \{b\}, \Sigma_3 = \{a, b\}$.*
- $N' = \{A_{0,0}, B_{0,0}, S_{0,0}, A_{1,2}, B_{2,1}, S_{3,3}, \sigma\}$.
- *P' contains the following productions:*

$$\begin{aligned} \sigma &\rightarrow \{S_{3,3}\} \\ \sigma &\rightarrow \{S_{0,0}\} \\ S_{3,3} &\rightarrow \{A_{1,2}, B_{2,1}\} \\ A_{1,2} &\rightarrow \{a\} \\ B_{2,1} &\rightarrow \{b\} \end{aligned}$$

Clearly $ML(H) = \{\{a, b\}\} = \mathcal{A}(G)$.

4.4 Attributed Multiset Grammars

In this section, the definition of multiset grammars is augmented to generate languages over attributed alphabets. Words in these languages will be attributed multisets defined over an attributed terminal alphabet. An attributed alphabet will also be used for nonterminal symbols, so each symbol in a grammar will have a set of attributes.

Each production of the grammar will have associated with it both a set of *semantic functions* and a set of *constraints*. The semantic functions specify the values of the attributes of the symbol on the left-hand of the production in terms of the values of the attributes of the symbols on the right-hand side. The constraints are predicates over the attributes of the right-hand side indicating when a production is validly applied.

To see the usefulness of attributes, observe that a multiset M which is an element of a context-free multiset language L_M fails to express two important characteristics. First of all, there is a syntactic structure implied for M by a grammar generating L_M. Secondly, there is an ordering defined over the elements of M by this syntactic structure which is not reflected in M. Attributes can be used both to restrict the syntactic structures allowed and to capture the *ordering* specified by the grammar in a general notion.

Definition 4.11 *An* attributed multiset grammar *is a six-tuple,*

$$G = (N, \Sigma, s, I, \mathcal{D}, P)$$

where

- N *is a finite set of non-terminal symbols.*
- Σ *is a finite set of terminal symbols.*
- $s \in N$ *is a start symbol.*
- I *is a finite set of attribute identifiers, called the* parsing attributes.
- $\mathcal{D}$ *is a set of attribute domains.*
- P *is a set of productions of the form* (R, SF, C)*, where*
  - R *is a rewrite rule of the form* $A \rightarrow M_1/\gamma$*, where* $M_1 \in V^*$ *and* $\gamma \in \Sigma^*$*.*
  - SF *is a set of semantic functions.*
  - C *is a set of constraints.*
- $V = N \cup \Sigma$ *is the vocabulary.*
- $(V, I, \mathcal{D})$ *form an attributed alphabet.*
- $(N, \Sigma, s, \{R_i \forall (R_i, SF_i, C_i) \in P\})$ *forms the underlying extended multiset grammar for* G*.*

Since $(V, I, \mathcal{D})$ form an attributed alphabet, the attribute identifiers consist of $|V|$ distinct subsets $I_X \forall X \in V$. Each vocabulary symbol X has an associated set of attributes, $\text{attr(X)} = \text{I}_\text{X}$. attr is extended to an attributed multiset by defining $\text{attr}^*(M) = \bigcup_{X_i \in M} \text{attr(X}_\text{i}\text{)}$.

Consider the production $p = (R, SF, C)$, where R is

$$A \rightarrow \{X_1, ..., X_k\}/\{b_1, ..., b_l\}$$

For each symbol Z in the rewrite rule (either A, X_i or b_i), there is a set of attribute occurances, one for each attribute $a_i \in \text{attr(Z)}$, given as $\text{occur}(Z) = \{Z.a_i \mid a_i \in \text{attr(Z)}\}$. occur is extended to the set of occurances in the entire production, defined by

$$\text{occur}(p) = \bigcup_{Z_i \in \{A, X_1, ..., X_k, b_1, ..., b_l\}} \text{occur}(Z_i)$$

The attribute occurances for different symbols in the rewrite rule are considered distinct. The attribute occurances for a production are divided into two exclusive sets:

- the *defined attributes* are $(\cup_{X_i \in M_1} \text{occur}(X_i)) \cup (\cup_{b_i \in \gamma} \text{occur}(b_i))$.
- the *computed attributes* are $\text{occur}(A)$.

Because the computed attributes are limited to the attributes of the left-hand side, attributed multiset grammars are restricted to *synthesized attributes.*[1] The semantic functions are mappings from the defined attributes to the computed attributes. For each attribute $a \in \text{attr(A)}$, there is a function f which defines the attribute. SF is defined by $SF = \bigcup_{a_j \in \text{attr(A)}} f_j$. The constraints C for production p are a finite set of functions c_j from the defined attributes to {True, False}. There are no restrictions on the f_j or c_j except that they be effectively computable (i.e. a recursive function). Generally, productions are displayed in the form

$A \rightarrow \{X_1, ..., X_k\}/\{b_1, ..., b_l\}$
$A.\alpha_j = f_j(\text{<defined attributes>})$
$\vdots$
more attribute equations
Where
$constraint_1(\text{<defined attributes>})$
$\vdots$
$constraint_n(\text{<defined attributes>})$

Semantic functions may be written as expressions when it is clear that an effectively computable function could be defined for that expression. Similarly, constraints are typically written as a relation of expressions (e.g. $A.a < B.b$). The semantic functions (attribute equations) or constraints will not be displayed if empty. Also, if γ is equal to $\emptyset$ then it will be left off the rewrite rule.

Example 4.7 *Let* $N = \{S\}, \Sigma = \{a, b\}$ *and* $I = \{i_L, i_R, i_a, i_b\}$ *with* $\mathcal{D}_i = \mathcal{Z}$. *Let* P *be the following set of productions:*

$S \rightarrow \{a, b\}$
$S.i_L = a.i_a$
$S.i_R = b.i_b$
Where
$a.i_a = b.i_b - 1$

$S \rightarrow \{S_1, a, b\}$
$S.i_L = a.i_a$
$S.i_R = b.i_b$
Where
$S_1.i_L = a.i_a + 1$
$S_1.i_R = b.i_b - 1$

(Here the subscript 1 on S *is meant to differentiate between the* S *on the LHS and the* S *on the RHS of the production).* $(N, \Sigma, S, I, \mathcal{D}, P)$ *is an attributed multiset grammar.*

Now the notion of tree-DAG is augmented to include attributes.

[1] Chapter 7 discusses how this restriction is relaxed to allow inherited attributes to be used in defining language semantics.

Definition 4.12 *A* semantic tree *over a grammar G is a (labelled, ordered) tree-DAG over $K_G \times D_{av}$, where D_{av} is the domain of attribute vectors over $(V, I, \mathcal{D})$.*

In other words, the labelling function in a semantic tree maps from the tree nodes to pairs (p, AV), where $p \in P \cup \Sigma$ and AV is a vector of attribute values for the symbol $lsymb(p)$. The concept of label symbols is extended to semantic tree labels by defining $lsymb((p, AV)) = lsymb(p)$. In addition, a *label attribute* is the attribute vector in a label, i.e. $lattr((p, AV)) = AV$.

The function STRIP maps from a semantic tree to the underlying unattributed tree-DAG as follows: Let $T = (N, E, L, F)$ be a semantic tree over G. Then $\text{STRIP}(T) = (N, E, L', F)$, where L' is given by $L' : N \rightarrow lsymb(l)$ whenever $L : N \rightarrow l$. In other words, $\text{STRIP}(T)$ is the tree-DAG which is only labelled by the symbols, and not the attributes.

It is not possible to define an analysis for an attributed multiset grammar.

Definition 4.13 *Let $G = (N, \Sigma, s, I, \mathcal{D}, P)$ be an attributed multiset grammar with G' the underlying extended multiset grammar. Let M be an attributed multiset with M' the base (unattributed) multiset. Then M is* analyzable *over G if there exists a semantic tree T such that M is the yield of T, $\text{STRIP}(T)$ is an analysis of M' over G' and the following holds for all interior nodes v of T: Let v be labelled by (p, AV) where $p = A \rightarrow \{X_1, ..., X_k\}, \{b_1, ..., b_j\}$ with semantic function SF and constraint C.*

- *v has tree children $n_1, ..., n_k$ and graph children $m_1, ..., m_j$ with n_i (m_i) labelled by (p_{n_i}, AV_{n_i}) ((p_{m_i}, AV_{m_i})), where $X_i = lsymb(p_{n_i})$ ($b_i = lsymb(p_{m_i})$).*
- *for each $\alpha \in I_A$, there is a function $f_\alpha \in SF$ such that f_α evaluates to $\text{PROJ}((A, AV), \alpha)$.*
- *every constraint $c_i \in C$ evaluates to True.*

The semantic tree T is called an analysis *of M over G.*

The language specified by an attributed multiset grammar is defined in terms of the analyzable attributed multisets. Given an attributed multiset grammar $G = (N, \Sigma, s, I, \mathcal{D}, P)$, the multiset language recognized by G ($\mathcal{A}(G)$) is given by

$$\mathcal{A}(G) = \{M \mid base(M) \in \Sigma^* \text{ and } M \text{ is analyzable over } G\}$$

Next are two definitions related to the size of the attribute domains (i.e. the number of different values the attributes may take). This is an important consideration when trying to parse attributed multiset languages, and is discussed in Chapter 6. Given an attributed multiset grammar G as defined above. Let $S \subseteq \mathcal{A}(G)$ be a set of attributed multisets in the language defined by G, and $T_S = \{t \mid t \text{ is an analysis of some } M \in S\}$.

Definition 4.14 *The* cardinality *of an attribute $a \in I$ with respect to a single analysis $t \in T_S$ ($card_a(t)$) is the number of different values found in occurances of a at any node in t. The cardinality of a with respect to the set S is given by $card_a(S) = \max_{t \in T_S}(card_a(t))$.*

Definition 4.15 *The grammar G is* finitely representable *if* $card_a(\mathcal{A}(G))$ *is finite* $\forall a \in I$.

Definition 4.16 *Let* $S^n = \{M \in \mathcal{A}(G) \mid |M| \leq n\}$. *The grammar G is* polynomially bounded *if there are constants* k, c *such that* $card_a(t) \leq cn^k \forall a \in I, t \in T_{S^n}$.

Let L_M be a multiset language. If $\forall M \in L_M, \text{rank}(M) \leq 1$, then L_M is said to be *non-overlapping*. A(n) (attributed) multiset grammar is non-overlapping if the language it generates (recognizes) is non-overlapping. A non-overlapping attributed multiset grammar recognizes multisets where no two elements are exactly the same (i.e. same symbol and attributes).

4.4.1 Analysis of Attribute Multiset Grammars

The *membership problem* is one important question about attributed multiset grammars. That is, for an attributed multiset grammar G and an attributed multiset M, is $M \in \mathcal{A}(G)$. For general attributed multiset grammars, this is difficult to answer, as the following theorem shows.

Theorem 4.4 *Let G be an attributed multiset grammar as defined above with G′ the underlying extended multiset grammar and G″ the underlying multiset grammar of G′, where the following restrictions hold:*

- *There is no symbol A such that* $A \stackrel{*}{\Rightarrow}_{G''} \{A\}$.
- *The semantic functions and constraints can all be evaluated in polynomial time and space.*

Let $M \in \Sigma^*$ *be an attributed multiset, then determining whether* $M \in \mathcal{A}(G)$ *is NP-complete.*

Proof:

This problem is in NP, because a machine can nondeterministically choose an order for the elements of M that is $fr(T)$ for some semantic tree T which is an analysis of M wrt G. The productions in G'' can be interpreted as a context-free grammar based on the canonical ordering. For each node in the tree T labelled by a production containing a context, one can nondeterministically choose the appropriate terminal node for the context. This allows the tree-DAG TD to be constructed verified that it is an analysis.

To show that this problem is NP-complete, a reduction is made from 3SAT. Given an instance of 3 Satisfiability consisting of a set of variables $\Sigma = \{\alpha_k\}$ and a set of N clauses $C_i = \{v_{i1}, v_{i2}, v_{i3}\}$, where each v_{ij} is either α or $\overline{\alpha}$ for some $\alpha_k \in \Sigma$, a grammar $3SAT$ is constructed as follows:

(1) S → {ClauseList}
Where
$$ClauseList.true \cap ClauseList.false = \emptyset$$

(2) ClauseList $\rightarrow$ {Clause$_1$, Clause$_2$, ..., Clause$_N$}
$ClauseList.true = \bigcup_i Clause_i.true$
$ClauseList.false = \bigcup_i Clause_i.false$

For each clause $C_i = \{v_{i1}, v_{i2}, v_{i3}\}$, where either $v_{ij} = \alpha_{k_{ij}}$ or $v_{ij} = \overline{\alpha}_{k_{ij}}$, there is a production

(3) Clause$_i$ $\rightarrow \{A_{i1}, A_{i2}, A_{i3}\}$
$Clause_i.true = A_{i1}.true \cup A_{i2}.true \cup A_{i3}.true$
$Clause_i.false = A_{i1}.false \cup A_{i2}.false \cup A_{i3}.false$
Where
$A_{i1}.sat \vee A_{i2}.sat \vee A_{i3}.sat$

and for each of j=1,2,3, the following two productions:

(4) $A_{ij} \rightarrow \{\alpha_{k_{ij}}\}$
$A_{ij}.true = \{\alpha_{k_{ij}}\}$
$A_{ij}.false = \emptyset$
$$A_{ij}.sat = \begin{cases} true & \text{if } v_{ij} = \alpha_{k_{ij}} \\ false & \text{if } v_{ij} = \overline{\alpha}_{k_{ij}} \end{cases}$$

(5) $A_{ij} \rightarrow \{\alpha_{k_{ij}}\}$
$A_{ij}.true = \emptyset$
$A_{ij}.false = \{\alpha_{k_{ij}}\}$
$$A_{ij}.sat = \begin{cases} false & \text{if } v_{ij} = \alpha_{k_{ij}} \\ true & \text{if } v_{ij} = \overline{\alpha}_{k_{ij}} \end{cases}$$

Σ is the set of terminals, and the input multiset M is given by $M = \{\alpha_k\}$, such that $M(\alpha_k)$ = the total number of occurences of α_k (or $\overline{\alpha}_k$) in all the clauses C_i. That is, M is the multiset of all the variables from the clauses with the negations removed. The negations are remembered by the (4) and (5) productions. This grammar is ambiguous because there are two productions $A_{i,j} \rightarrow \alpha_{k_{ij}}$. Resolving this ambiguity chooses a value for $\alpha_{k_{ij}}$. The constraint of production (3) insures that the clause is satisfied. The *true* and *false* attributes build lists of the variables which are assigned True and False values respectively. The constraint of production (1) insures that the variable assignments for all the clauses are consistent (i.e that no α_k is assigned True in one case and False in another).

If $M \in ML(3SAT)$, then there is a derivation tree for M. This derivation assigns a value to every variable such that every clause is satisfied. If the clauses can be satisfied, then a derivation tree can be constructed by choosing the appropriate production (4 or 5) for each term of each clause. □

Theorem 4.4 shows that attributed multiset grammars are too powerful in general to use for recognition. However, the fact that attributed multiset grammars can express powerful computations is not a reason for not using them. It is this very power that makes AMGs an attractive mechanism for specifying the syntax of two dimensional

languages. This power can be utilized by restricting the ways in which the attributes can be used to realize subclasses of attributed multiset languages which can be efficiently recognized. For example, the class of finitely representable, non-overlapping attributed multiset grammars is such a class.

Chapter 5

Picture Layout Grammars

5.1 Introduction

The previous chapters laid out a framework for the specification of visual language syntax. They developed a model of visual languages; motivated the need for syntax specifications; and outlined a grammar-based approach to visual language definition. Finally, in Chapter 4, a computational model was developed for visual languages, based on attributed multisets.

This chapter discusses the actual mechanism used to specify visual language syntax, the *picture layout grammar*. The language designer uses this mechanism to define a new visual language. The language definition is a text file containing a grammar written in the *Grammar Definition Language*. The GREEN programming environment reads a grammar file and uses the language definition to process visual programs. The grammar file contains the following:

- The definitions (i.e. name and type) of the attributes.
- The definitions of the terminal and nonterminal alphabets. For each symbol, the symbol class name and all the attributes are given.
- The names and entry points for external functions used in the grammar, along with an external object library to be dynamically loaded.
- The start symbol.
- A list of productions.

The precise syntax of the grammar definition language and the format of the grammar file are described in Appendix A.

Picture layout grammars are a restricted form of attributed multiset grammar, specifically designed to describe two dimensional syntax. A picture is represented as an attributed multiset. The primitive picture elements form the terminal alphabet of the grammar, over which the attributed multiset is defined. The attributes of the symbols contain both the locations and the characteristics of the picture elements.

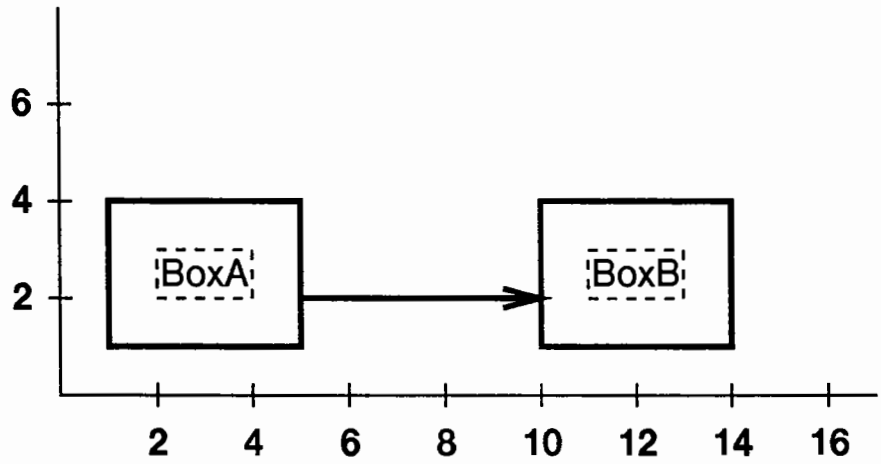


Figure 5.1: A Sample Picture

Every symbol (terminal and nonterminal) will have four *geometric attributes: lx, by, rx, ty,* which represent the geometry of the corresponding graphical entity. The geometric attributes are specified in a coordinate system where X is the horizontal axis and increases from left to right; and Y is the vertical axis and increases from bottom to top (as shown in Figure 5.1). These attributes can be interpreted in two ways:

- If the object is a *shape*, then the attributes define the *extent* of the shape, that is, a rectangle with (lx, by) as the lower-left corner and (rx, ty) as the upper-right corner. In this case, the relationships $lx \leq rx$ and $by \leq ty$ must hold.
- If the object is a *line*, then (lx, by) is the *left* endpoint and (rx, ty) is the *right* endpoint. Although the endpoints are conventionally called *left* and *right*, there is no requirement that $lx \leq rx$ (or $by \leq ty$) in this case.

The interpretation of a symbol as a shape or a line depends on the context in which it is used (i.e. how it is used in a production). Symbols may have other attributes in addition to the four required ones. For terminal symbols, these attributes express other characteristics of the graphical primitive (e.g. the string value of a text primitive). The specific terminal symbols are not fixed (they are defined as part of the grammar), but the GREEN system provides four shapes (**rectangle**, **circle**, *octagon* and *text*) and two lines (**line** and **arrow**).

Example 5.1 *Consider the picture shown in Figure 5.1. The terminal alphabet for this language consists of three symbols:* **rectangle**, **text** *and* **arrow**. *This picture is represented by an attributed multiset with five elements.*

picture = { **rectangle**[1,1,5,4], **rectangle**[10,1,14,4], **text**[2,2,4,3,"BoxA"], **text**[11,2,13,3,"BoxB"], **arrow**[5,2,10,2] }

The syntactic structure of the program parallels the compositional structure of the picture. Each composition in the picture corresponds to some production in the grammar. Some productions (e.g. those which merely rename nonterminals), however, merely control the syntactic structure and have no corresponding composition. The nonterminal symbols correspond roughly to the aggregate objects.

Formally, a picture layout grammar is defined as follows:

Definition 5.1 *A* picture layout grammar *is a non-overlapping, finitely representable attributed multiset grammar* $G = (N, \Sigma, s, I, \mathcal{D}, P)$ *where the following are true:*

1. *Every symbol* $x \in N \cup \Sigma$ *has (at least) the following four distinguished parsing attributes,* lx, by, rx, ty.

2. *If* $p = (R, SF, C) \in P$, *with* $R = A \rightarrow X/Y$, *then for all* $\alpha \in \{lx, rx, by, ty\}$, *the semantic function* SF *contains an assignment*

$$A.\alpha = x.\beta \text{ where } x \in X \cup Y \text{ and } \beta \in \{lx, rx, by, ty\}$$

In other words, p copies the attributes lx, rx, by, ty *of the left hand side from similar (but not necessarily corresponding) attributes of elements of the right-hand side.*

While picture layout grammars are formally viewed as attributed multiset grammars, they have a slightly different syntax. This is simply a more convenient way of writing the grammars. It is important to note that the grammar being defined is a specialized attributed multiset grammar and that the notation for productions in a picture layout grammar is merely a shorthand for the underlying attributed multiset grammar productions.

As with attributed multiset grammars, a production in a picture layout grammar consists of three parts: the rewrite rule, the semantic function and the constraint. The notation used for the rewrite rule is similar to that used for the simple picture grammars in Chapter 3. The rewrite rule is one of the following forms:

1. $N \rightarrow X$

2. $N \rightarrow op(X_1, ..., X_m)$

Here N is a nonterminal and X is a terminal or nonterminal. The first form merely renames a symbol (though possibly enforcing some constraint or modifying some attributes). The second form of production represents a composition. The *op* is a built-in composition operator, from the list in Table 5.1.

When more than one instance of a symbol occurs in a rewrite rule, they are distinguished with subscripts. Within the right-hand side of the rewrite rule, any terminal symbol may be marked as *remote* by underlining it. The remote symbols in a production form the context multiset for the underlying attributed multiset production. The other symbols form the regular multiset.

shape compositions	over left_of contains adjacent_to tiling
line compositions	follow join fork parallel reverse
shape/line compositions	touches_L touches_R points_to points_from labels

Table 5.1: Picture Layout Grammar Production Operators

For example, the PLG rule

FIGURE → over(TOP,BOTTOM)

corresponds to the AMG rule

FIGURE → {TOP,BOTTOM}/{}.

If the symbol TOP were marked as remote, as in

FIGURE → over(TOP,BOTTOM),

the corresponding AMG rule would be

FIGURE → {BOTTOM}/{TOP}.

In the examples below, no terminals are marked as remote (except where demonstrating its use), but this does not imply that they could not be.

The Constraint

The constraints in a production express the required relationship between the components of a composition. Each built-in production operator has an associated constraint specifying the two dimensional syntax of that composition. For example, the constraint associated with the *over* operator says that the bottom of the first argument must be above the top of the second argument. These constraints are expressed in terms of the attributes of the symbols. The PLG production

FIGURE → over(TOP,BOTTOM)

has the constraint TOP.by $\geq$ BOTTOM.ty.

This example also illustrates how the productions determine the interpretation of the shapes. That is, a component in a visual language (i.e. a terminal or nonterminal symbol) can have its four geometric attributes interpreted as either the extent or the endpoints of the component. In the shape compositions, the components of the operator are treated as shapes, and the attributes interpreted accordingly. Similarly, the line

Operator Name	meaning
$< \leq == \geq > != $	integer relations
&&	AND of two relations
||	OR of two relations
Function Name	**meaning**
$+ - * /$	integer arithmetic
max(integer$_1$,integer$_2$) $\rightarrow$ integer$_3$	maximum of two integers
min(integer$_1$,integer$_2$) $\rightarrow$ integer$_3$	minimum of two integers
concat(string$_1$,string$_2$) $\rightarrow$ string$_3$	concatenate two strings
strcmp(string$_1$,string$_2$) $\rightarrow$ integer	compare two strings
cons(list,elem) $\rightarrow$ list	add elem to front of list
append(list$_1$,list$_2$) $\rightarrow$ list	append list$_2$ to end of list$_1$
list(elem$_1$,...) $\rightarrow$ list	make new list of elements
length(list) $\rightarrow$ integer	return length of list
cdr(list) $\rightarrow$ elem	return CDR of list
car(list) $\rightarrow$ list	return first element of list
test_adjacent(x,y,z,w) $\rightarrow \{0,1\}$	do adjacency testing

Table 5.2: Built-in Operators and Functions

compositions treat the components as lines (i.e. (lx, by) and (rx, ty) are taken to be endpoints of the line); and the shape/line compositions treat one component as a shape and the other as a line.

In addition to the constraints associated with the built-in production operators, a production may contain other constraints specified by the language designer. The actual constraint for a production is the conjunction of the built-in constraint and all user specified constraints for the production.

The user defined constraints are also defined in terms of the attributes of the elements of the right-hand side. These attribute references, along with integer and string constants, form the terms of expressions. The terms are combined using the functions shown in Table 5.2. Additional functions can be defined in the grammar file and dynamically loaded. A constraint is formed by comparing two integer expressions using the comparison operators in Table 5.2. The constraints can be further combined using the two logical operations. If more than one user defined constraint is given in a production, they are ANDed together to form the complete user defined constraint.

Example 5.2 *Consider the picture shown in Figure 5.2, consisting of two rectangles. This language could be defined by the production*

FIGURE $\rightarrow$ over(**rectangle**$_1$, **rectangle**$_2$)
Where
rectangle$_1$.lx == **rectangle**$_2$.lx
rectangle$_1$.rx == **rectangle**$_2$.rx
rectangle$_1$.by == **rectangle**$_2$.ty

This production augments the over operator with three additional constraints. These

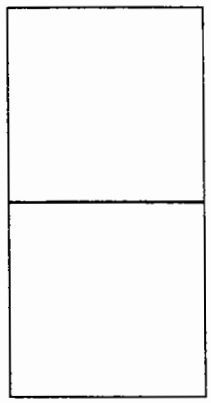


Figure 5.2: A Language Using User Defined Constraints

user defined constraints specify that the two rectangles forming the picture must be aligned on the left and right sides, as well as along the border between them.

The Semantic Functions

The semantic function in a production computes the attributes of the aggregate object resulting from the composition. As with the constraints, each built-in production operator has a semantic function which implements the associated composition. The built-in semantic functions will compute the values of the four attributes lx, by, rx, ty that are appropriate for the aggregate object. For example, the rule

FIGURE → over(TOP,BOTTOM)

sets the attributes to the extent which encloses both the TOP and the BOTTOM (i.e. to the smallest rectilinear rectangle which encloses both TOP and BOTTOM). The semantic function of this production is equivalent to

FIGURE → {TOP,BOTTOM}
FIGURE.lx = min(TOP.lx,BOTTOM.lx)
FIGURE.rx = max(TOP.rx,BOTTOM.rx)
FIGURE.by = BOTTOM.by
FIGURE.ty = TOP.ty

Again, the language designer can include additional semantic functions in a production. During parsing, the built-in semantic functions are evaluated first, followed by the user specified semantic functions. Thus the additional semantic functions can be used to override the values computed by the built-in semantic functions, as long as they continue to select from the geometric attribute instances of the right hand side and keep $lx \leq rx, by \leq ty$ for shapes. (care must be taken to ensure that the resulting language is correct) A more typical use of the additional semantic functions is to compute values for additional attributes, which are then used in subsequent constraints or functions.

Example 5.3 *Consider the picture shown in Figure 5.3, consisting of four rectangles. The following productions define a language of similar tilings:*

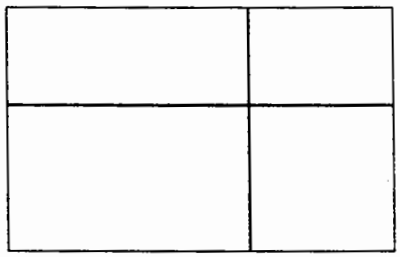


Figure 5.3: A Language Using Additional Attributes

HORIZ → left_of(**rectangle**$_1$,**rectangle**$_2$)
HORIZ.center = **rectangle**$_1$.rx
Where
rectangle$_1$.rx == **rectangle**$_2$.lx

FIGURE → over(HORIZ$_1$,HORIZ$_2$)
Where
HORIZ$_1$.by == HORIZ$_2$.ty
HORIZ$_1$.center == HORIZ$_2$.center

Here the center *attribute is used to align the splitting point on the top and bottom of the figure.*

The remainder of this chapter discusses the built-in production operators. The next section describes a mechanism for specifying adjacency constraints. The following three sections cover the operators, grouped according to the type of composition.

5.2 Adjacency Testing and Regions

As noted earlier, compositions have constraints associated with them that specify the required spatial relationship. For some compositions (e.g. coincidence of line endpoints), this relationship consists only of the relative positions of the components. Other compositions (e.g. *over*) require not only that the components have a given relative position, but that the components be *adjacent* objects, in that there be no intervening objects. For example, in Figure 5.4, although the box labelled '**a**' is to the left of the box labelled '**c**', they would not be considered horizontally adjacent because of the box labelled '**b**' between them. These restrictions are called *adjacency*

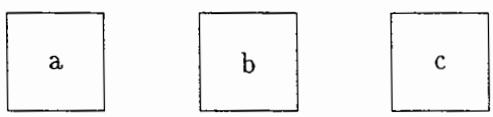


Figure 5.4: Some Adjacent Boxes

constraints. This section describes the mechanism for formulating and evaluating adjacency constraints.

There are four types of adjacency defined: horizontal, vertical, containment and label. All four types of adjacency are based on a common framework. An intuitive definition of adjacency is to say that two entities are adjacent if there is nothing between them. This definition can be made precise by specifying what it means for two objects to have "nothing between them".

The approach taken here is to identify a particular area of the picture and enforce the constraint that this area must not contain any terminal symbols. The area identified depends on both the composition operator and the actual locations (i.e. extents) of the objects being composed. The area comprises a region according to the following definition.

Definition 5.2 *A* region *is either a simple region or a complex region. A* simple region *is a rectangular area specified by four integers* lx, by, rx, ty. *A* complex region *is a union of simple regions.*

To determine whether two objects meet an adjacency constraint, the region corresponding to the area "between" the two objects is formed. The actual area for each composition is discussed with the composition below. The terminal symbols are searched to see if any intersect the region. If the region is complex, it is checked by examining the component regions individually.

There are two reasons for only considering terminal symbols when defining adjacency. First, it corresponds to the intuitive notion of adjacency. If two things are not adjacent, this should be reflected by something intervening in the picture. Aggregate objects (i.e. nonterminals) are a result of the interpretation of the picture. The "invisible" parts of an aggregate (e.g. the area between two composed symbols) aren't directly in the picture, and so are not considered for adjacency. A second reason is that it fits with the bottom-up analysis scheme in the next chapter.

Restricting our attention to terminal symbols for adjacency testing would seem to produce some counterintuitive results. Consider the picture in Figure 5.5, for example. Looking only at either the two circles or the two squares, the pairs would be considered to be adjacent (vertically and horizontally, respectively). Allowing each to be composed would seem to be incorrect, as one of the aggregates must be "between" the components of the other. This is a limitation of the model of adjacency. If the overall picture is considered, however, then Figure 5.5 forms a valid picture only if the two overlapping aggregates can be composed.

Sometimes it is necessary to relax the adjacency constraint in a production. For instance, consider the fsa program shown in Figure 3.3. In this program, adjacent states in the fsa are composed to form an aggregate of all the states. The states would not be considered adjacent however, because the terminal symbols for the transition arrows and labels would be intervening.

This problem is solved by allowing the language designer to designate a class of terminal symbols to be ignored in adjacency testing for a production. The definition

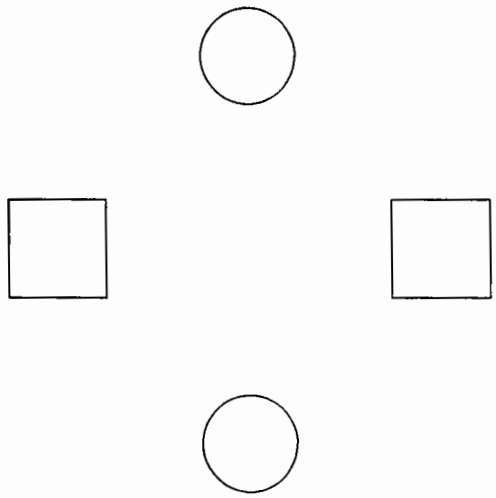


Figure 5.5: Intervening Aggregate Objects

of a terminal symbol lists the classes to which it belongs. The class is then made up of all the terminal symbols listing it. In a production, the *ignore class* is specified by a positive integer, as in

$$A \rightarrow \text{over(B,C)[1]}$$

If no ignore class is specified in the production, it defaults to 0, indicating that no terminals should be ignoreed. The ignore class has no effect for a production operator which does not test for adjacency.

An adjacency constraint may be explicitly coded using the `test_adjacent` built-in function. The implementation of the adjacency testing is discussed in the following section. Access to the underlying adjacency mechanism is also available from within an external user function for defining new types of adjacency.

5.2.1 Implementing Adjacency Constraints

The adjacency constraints are implemented by the `test_adjacent` routine. This routine takes four arguments: the type of adjacency, the two nodes involved and the class of terminals to ignore. It may be called from an explicit constraint as

`test_adjacent(`*type*`, @`*node1*`, @`*node2*`,` *class*`)`[1]

where *type* indicates the kind of adjacency and is one of ADJACENT_X, ADJACENT_Y, ADJACENT_ANY, ADJACENT_CONTAINS or ADJACENT_LABEL. ADJACENT_ANY signifies either X or Y adjacency. The node arguments are ordered, with the *node1* argument being the left, bottom, inside or line object and the *node2*

[1]The notation '@*node1* indicates that the entire nonterminal symbol is the argument, rather than a specific attribute.

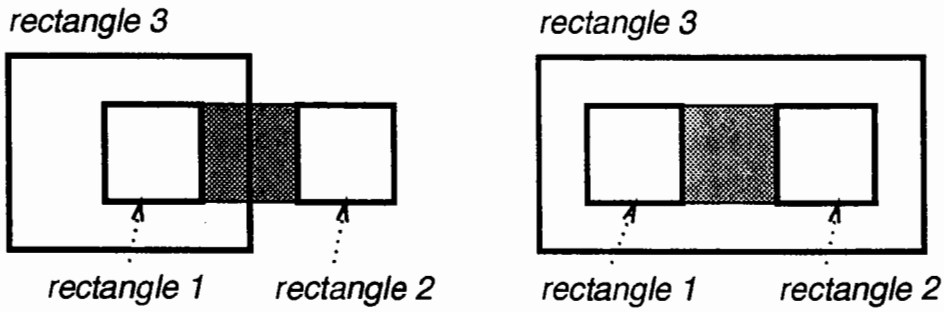


Figure 5.6: Overlapping Regions and Region of Interest

argument being the right, top, outside or label object as appropriate for the type of adjacency (the nodes do not have to be ordered for ADJACENT_ANY). `test_adjacent` returns 1 if the specified adjacency constraint is satisfied, or 0 if not.

For the X, Y and ANY adjacency, the two objects are first checked to see that they do not overlap. In the case of ADJACENT_ANY, the two objects are compared to determine the appropriate relationship. The first check determines whether *node1* is to the left of *node1*; then the opposite relationship is checked. Next, *node1* over *node2* or vice-versa are checked. Since the nodes do not overlap, at least one of these four possibilities must be true. The first condition that is true is taken as the relationship, and the constraint is treated as the appropriate (X or Y) adjacency with the nodes in the order determined.

Based on the type of adjacency and the geometry of the two nodes, `test_adjacent` constructs two regions: the adjacency region and the region of interest. The adjacency region is the area that will be searched for terminal symbols. The areas composing the adjacency region are described with the specific production operators later in this chapter. The region of interest is the overall region being examined, and is always a simple region (i.e. a single rectangular extent). For X and Y (and ANY) adjacency, the region of interest is the extent containing both of the objects. For CONTAINS adjacency, the region of interest is the extent of the outer object.

Next, the input set is searched for any terminals which overlap the adjacency region. If the adjacency region is a union of regions, a terminal is checked against each of the areas in the union separately and is considered to overlap if it overlaps any subarea. For adjacency testing, all terminals are treated as rectangles. A rectangle overlaps a region if

1. Any of the corners of the rectangle are within the region.
2. Any of the edges of the rectangle intersect the region.
3. The rectangle contains the region, but does not contain the entire region of interest.

The first two checks do not include the boundary of the region.

The third check finds rectangles which overlap the region completely. The region of interest serves to eliminate terminals which are outside the scope of the adjacency. For example, consider the drawings in Figure 5.6 where the X adjacency of rectangles 1 and 2 is being tested. The adjacency region is the shaded rectangle between rectangles 1 and 2, and the region of interest is the extent containing both rectangles. In the drawing on the left, rectangle 3 is intervening between rectangles 1 and 2, because it contains the adjacency region but not the region of interest. In the right-hand drawing, rectangle 3 contains the entire region of interest and is not considered intervening.

Label adjacency is handled slightly differently, and is described with the *labels* operator.

5.3 Renaming Productions

The simplest productions are those which merely rename a symbol, as in

LABEL → **text**
STATE → ATOMIC_STATE

Both of these productions have no built-in constraints associated with them. The built-in semantic function copies the geometric attributes of the right hand side to the nonterminal on the left hand side. That is, the first production is equivalent to

LABEL → {**text**}
    LABEL.lx = **text**.lx
    LABEL.rx = **text**.rx
    LABEL.by = **text**.by
    LABEL.ty = **text**.ty

In addition to expressing syntactic structure, these productions can transform attributes, compute new attribute values, or enforce constraints. This is accomplished using addition semantic functions and/or constraints. For example, the production

VERT_BAR → **line**
Where
    line.lx == **line**.rx
    line.style == DASHED_STYLE

is used to specify a vertical, dashed line, such as used for the orthogonal product in StateCharts.

5.4 Shape Composition Operators

Shape compositions combine two elements which are abstracted as rectangular regions. Five different shape composition operators are provided: *over, left_of, adjacent_to, contains* and *tiling*. The first four operators each represent a different type of relationship between two objects. These four compositions, together with additional constraints, are sufficient for expressing most spatial relationships between shapes. An example of a relationship which cannot be expressed is shown in Figure 5.7, where two objects overlap, but neither contains the other. The *tiling* operator provides for expressing

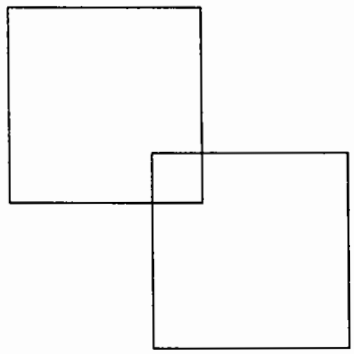


Figure 5.7: Overlapping Objects

arbitrary relationships between objects. We now examine the individual shape composition operators, describing the exact relationship between constituents, the adjacency constraint and how the aggregate is computed.

5.4.1 Over

The *over* production operator is used for vertical composition of objects. The production

$$A \rightarrow \text{over(B,C)}[i]$$

is a notation for the underlying attributed multiset grammar production

$A \rightarrow \{B,C\}$
A.lx = min(B.lx,C.lx)
A.rx = max(B.rx,C.rx)
A.by = C.by
A.ty = B.ty
Where
B.by $\geq$ C.ty
test_adjacent(ADJACENT_Y,@B,@C,i)

The explicit relationship between objects is the most general that could be considered "over"; namely that the bottom of B be at or above the top of C. No other condition (except the adjacency constraint) is placed on the components. Figure 5.8 shows five different examples of the *over* relationship. In the last two examples, the objects B and C hove no vertical overlap but are still considered as B over C.

A more specific relationship can be achieved by using additional constraints. For example, the production

$A \rightarrow$ over(B,C)
Where
B.rx > C.lx
B.lx < C.rx

specifies that B is over C and that the two objects overlap vertically. The pictures 5.8d and 5.8e would not satisfy this constraint.

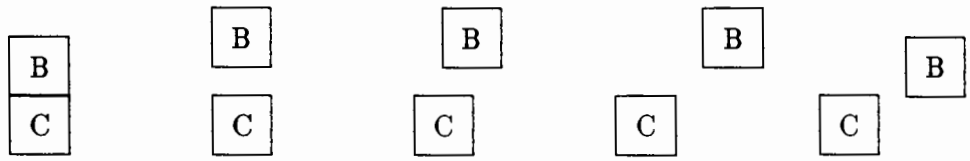


Figure 5.8: Examples of over(B,C)

The aggregate object resulting from an *over* composition is also a shape. Its extent is the rectangle containing the two components. Figure 5.9 gives the same examples with the aggregate object shown in dashed lines.

The adjacency constraint is expressed as a call to `test_adjacent`, specifying adjacency is in the vertical direction. The region associated with vertical adjacency consists of the extent of the aggregate object with the extents of the two components removed. In other words, the area "in between" the two components is the part of the containing extent which is not covered by the components. Any terminals within the components are assumed to already be taken into account. There are five simple regions from which the adjacency region is constituted, with at most three present at any one time. The subregions possible are to the left and right of each component and between the two components, as shown in Figure 5.10.

To see how the *over* operator is used, consider the StateChart language defined in Chapter 3. Production 11 of the grammar given there was

PRODUCT_INSIDE → vertical(PRODUCT_INSIDE$_1$,**line**,PRODUCT_INSIDE$_2$)

This production is intended to compose two PRODUCT_INSIDEs which are vertically aligned and separated by a horizontal dashed line. This can be expressed using the following picture layout grammar productions:

PRODUCT_INSIDE → over(PRODUCT_INSIDE$_1$,BOTTOM_PR_INSIDE)

BOTTOM_PR_INSIDE → over(HORIZ_BAR,PRODUCT_INSIDE)

HORIZ_BAR → **line**

Where

line.by == **line**.ty

line.style == DASHED_STYLE

The first two productions specify the vertical composition of three elements. The third production defines a horizontal, dashed line.

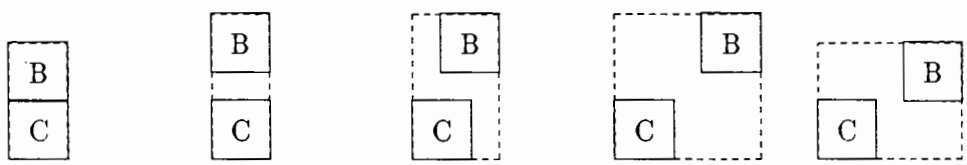


Figure 5.9: Aggregate objects for over(B,C)

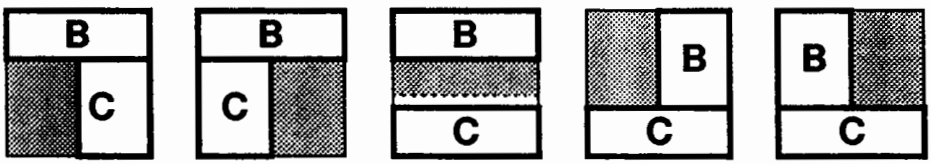


Figure 5.10: Regions for *over* Adjacency

5.4.2 Left_Of

The *left_of* production operator is used for horizontal composition of objects. It is analogous to the *over* operator rotated by ninety degrees. The production

$$A \rightarrow \text{left\_of(B,C)}[i]$$

is a notation for the underlying attributed multiset grammar production

$A \rightarrow \{B,C\}$
A.lx = B.lx
A.rx = C.rx
A.by = min(B.by,C.by)
A.ty = max(B.ty,C.ty)
Where
B.rx $\leq$ C.lx
test_adjacent(ADJACENT_X,@B,@C,i)

Again, a general explicit relationship between objects is used. In this case, the constraint is that the right edge of B must be on or to the left of the right edge of C. The aggregate object resulting from a *left_of* composition is also a shape with the rectangle containing the two components as its extent. Figure 5.11 shows five different examples of the *left_of* relationship, with the aggregate object shown in dashed lines. Note that the last two examples could be either *left_of* or *over*. As with the *over* operator, the spatial relationship of the components can be made more precise using additional constraints.

The adjacency constraint is similar to the *over* constraint except that adjacency is in the horizontal direction. The region associated with horizontal adjacency again consists of the extent of the aggregate object without the extents of the two components. The five subregions from which the adjacency region can be formed are above and below each component and between the two components, as shown in Figure 5.12.

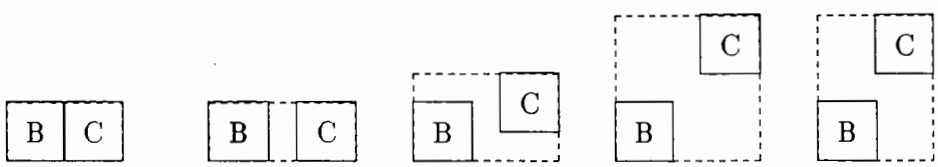


Figure 5.11: Examples of left_of(B,C) with Aggregates Shown.

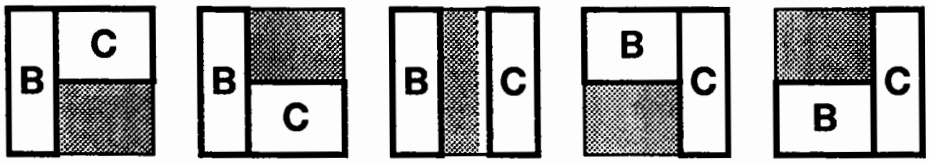


Figure 5.12: Regions for *left_of* Adjacency

5.4.3 Adjacent_To

The *adjacent_to* production operator is used for nonspecific spatial composition. It is equivalent to saying that two objects are either *over* or *left_of* each other (with the order unspecified). That is, the production

$$A \rightarrow \text{adjacent\_to}(B,C)[i]$$

may be thought of as shorthand for the four productions

$$A \rightarrow \text{over}(B,C)[i]$$
$$A \rightarrow \text{over}(C,B)[i]$$
$$A \rightarrow \text{left\_of}(B,C)[i]$$
$$A \rightarrow \text{left\_of}(C,B)[i]$$

The *adjacent_to* production is equivalent to the underlying attributed multiset grammar production

$A \rightarrow \{B,C\}$
A.lx = min(B.lx,C.lx)
A.rx = max(B.rx,C.rx)
A.by = min(B.by,C.by)
A.ty = max(B.ty,C.ty)
Where
(B.by $\geq$ C.ty && test_adjacent(ADJACENT_Y,@B,@C,i)) ||
(C.by $\geq$ B.ty && test_adjacent(ADJACENT_Y,@C,@B,i)) ||
(B.rx $\leq$ C.lx && test_adjacent(ADJACENT_X,@B,@C,i)) ||
(C.rx $\leq$ B.lx && test_adjacent(ADJACENT_X,@C,@B,i))

5.4.4 Contains

The *contains* production operator composes two objects where one is nested within the other. The production

$$A \rightarrow \text{contains}(B,C)[i]$$

is a notation for the underlying attributed multiset grammar production

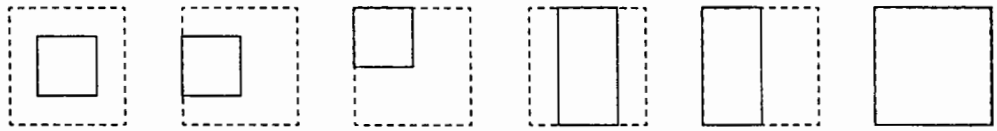


Figure 5.13: Examples of contains(B,C)

A → {B,C}
A.lx = B.lx
A.rx = B.rx
A.by = B.by
A.ty = B.ty
Where
B.lx ≤ C.lx
B.rx ≥ C.rx
B.by ≤ C.by
B.ty ≥ C.ty
test_adjacent(ADJACENT_CONTAINS,@B,@C,*i*)

The explicit relationship between the objects states that C is completely contained within B. C may abut on some, none or all of the edges of B. Figure 5.13 shows some examples of the *contains* relationship, with B drawn in dotted lines and C drawn in solid lines. As with the previous operators, the spatial relationship can be further specified using additional constraints. The aggregate object resulting from a *contains* composition has the same extent as the enclosing object.

The adjacency constraint for *contains* enforces the notion of nested objects. The area for contains adjacency is the interior of the containing object exclusive of the extent of the contained object. In other words, it is the area outside the inner object and inside the outer object. This area is formed from four possible regions: to the left, right, above and below the inner object, as shown in Figure 5.14.

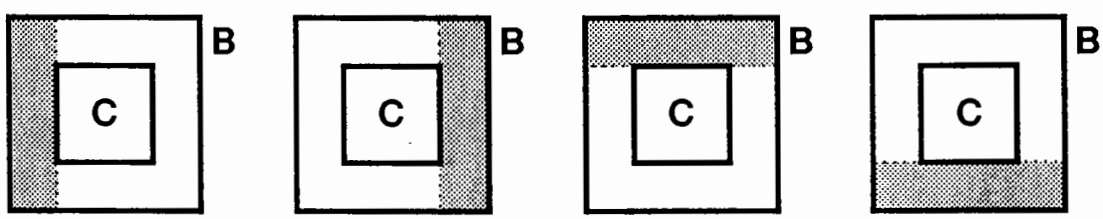


Figure 5.14: Regions for *contains* Adjacency

5.4.5 Tiling

The *tiling* operator is an unspecified composition. It takes an arbitrary number (one or more) component arguments. The built-in semantic function and constraint are both null, so the language designer must supply addition functions and constraints for any production using the *tiling* operator. The *tiling* operator is not, strictly speaking, a shape composition, since it has no fixed composition associated with it. It serves two purposes, however:

- The *tiling* operator provides a mechanism for the language designer to create new kinds of productions (i.e. new production operators). For example, the relationship in Figure 5.7 could be described by the production

  TWO → tiling(UP_LEFT,DOWN_RIGHT)
  TWO.lx = UP_LEFT.lx
  TWO.rx = DOWN_RIGHT.rx
  TWO.by = DOWN_RIGHT.by! TWO.ty = UP_LEFT.ty
  Where
  (DOWN_RIGHT.lx > UP_LEFT.lx && DOWN_RIGHT.ty < UP_LEFT.ty)
  (UP_LEFT.rx > DOWN_RIGHT.lx && UP_LEFT.by < DOWN_RIGHT.ty)
  (DOWN_RIGHT.rx > UP_LEFT.rx && DOWN_RIGHT.by < UP_LEFT.by)

- The *tiling* operator allows the composition of more than two objects at a time. Chapter 6 describes a shape composition which cannot be broken down into a series of simple *over* and *left_of* productions, but which can be specified using the *tiling* operator.

Since no built-in semantic function is provided for *tiling* productions, it is essential that the grammar writer provide a semantic function which properly defines the geometric attributes of the left hand side.

5.5 Line Composition Operators

The other abstract primitive is the directed line segment, which is represented by two points, *left* and *right* endpoints. These points are encoded as four attributes, (lx, by) for the left endpoint and (rx, ty) for the right endpoint.

Shapes interact through spatial relationships; line segments interact by means of coincidence of endpoints. GREEN provides five production operators for composing lines, patterned after the operators proposed by Shaw in his PDL system [83]. In PDL, the left endpoint is called the *tail* and the right endpoint the *head*, and we adopt this terminology in describing the operators. Each operator combines a different combination of coincident endpoints.

5.5.1 Follow

The *follow* operator composes two lines where the head (i.e. right endpoint) of the first coincides with the tail (i.e. left endpoint) of the second. The aggregate is a line from

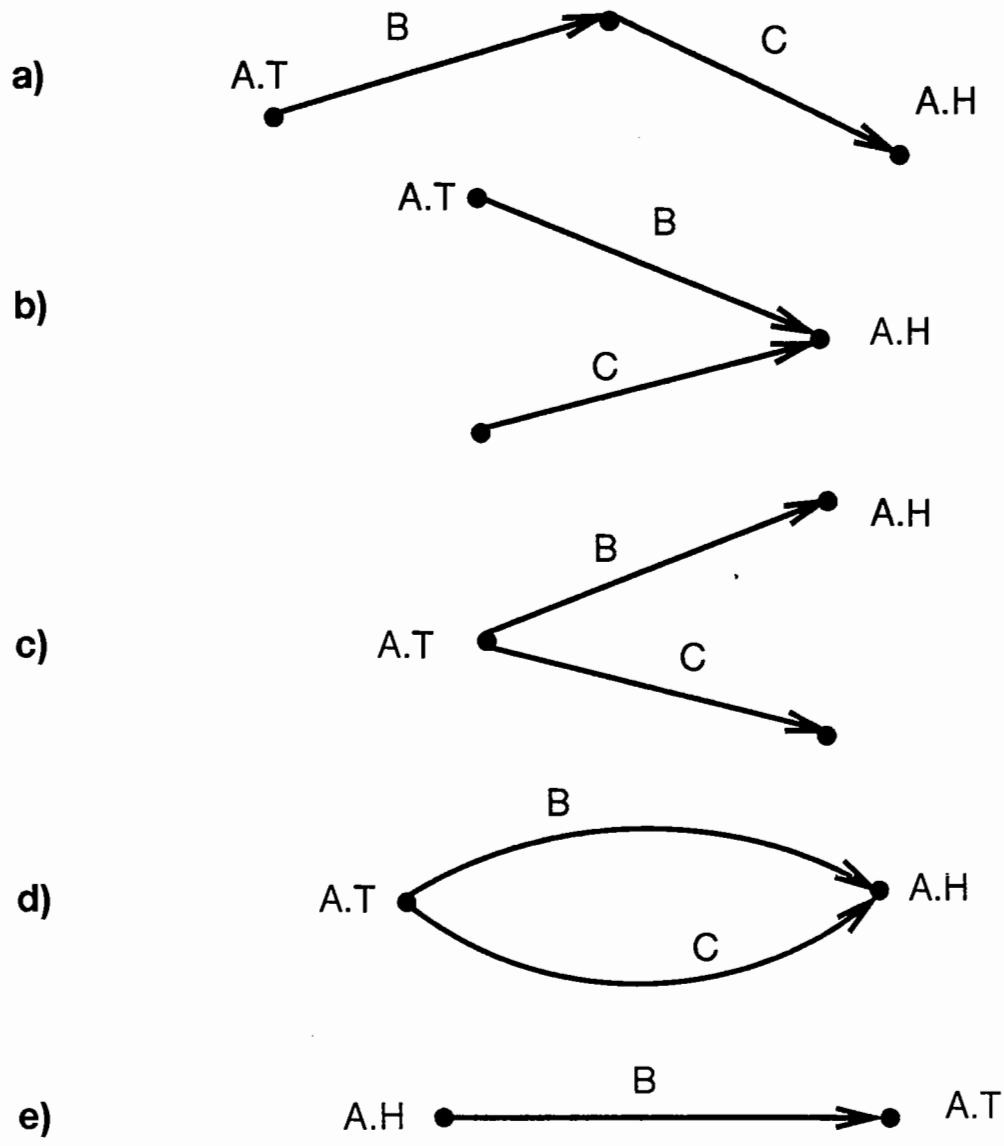


Figure 5.15: Line Composition Operators: a) follows(B,C); b) join(B,C); c) fork(B,C); d) parallel(B,C); e) reverse(B)

the tail of the first to the head of the second. Figure 5.15a illustrates the situation. Intuitively, this operator concatenates one line onto the end of another. It corresponds to the + operator in PDL. The production

$$A \rightarrow \text{follow(B,C)}$$

is equivalent to the attributed multiset grammar production

A → {B,C}
A.lx = B.lx
A.by = B.by
A.rx = C.rx
A.ry = C.ry
Where
B.rx == C.lx
B.ry == C.by

5.5.2 Join

The *join* operator composes two lines with coinciding heads, forming an aggregate that is the same as the first line, as shown in Figure 5.15b. The *join* operator corresponds to the − operator in PDL. The production

$$A \rightarrow \text{join(B,C)}$$

is equivalent to the attributed multiset grammar production

A → {B,C}
A.lx = B.lx
A.by = B.by
A.rx = B.rx
A.ry = B.ry
Where
B.rx == C.rx
B.ry == C.ry

5.5.3 Fork

The *fork* operator composes two lines with coinciding tails, again building an aggregate the same as the first line. The *fork* operator is shown in Figure 5.15c. *fork* corresponds to the × operator in PDL. The production

$$A \rightarrow \text{fork(B,C)}$$

is equivalent to the attributed multiset grammar production

A — {B,C}
A.lx = B.lx
A.by = B.by
A.rx = B.rx
A.ry = B.ry
Where
B.lx == C.lx
B.by == C.by

5.5.4 Parallel

The *parallel* operator composes two lines with both coinciding heads and tails. The aggregate will have the same head and tail as the lines, as shown in Figure 5.15d. The *parallel* corresponds to the $*$ operator in PDL. The production

$$A \rightarrow \text{parallel(B,C)}$$

is equivalent to the attributed multiset grammar production

$A \rightarrow \{B,C\}$
A.lx = B.lx
A.by = B.by
A.rx = B.rx
A.ry = B.ry
Where
B.lx == C.lx
B.by == C.by
B.rx == C.rx
B.ry == C.ry

Since the *parallel operator* requires both operands to have the same geometric attributes, the two components must be distinguished in some other manner (e.g. different symbols or additional attributes).

5.5.5 Reverse

The *reverse* operator takes a single line and reverses the head and tail to form an aggregate, as in Figure 5.15e. It corresponds to the $\neg$ operator in PDL. The production

$$A \rightarrow \text{reverse(B)}$$

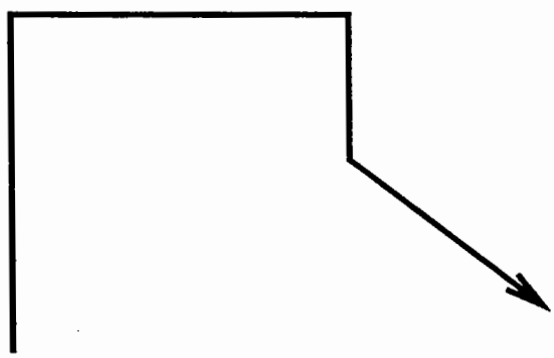
is equivalent to the attributed multiset grammar production

$A \rightarrow \{B\}$
A.lx = B.rx
A.by = B.ry
A.rx = B.lx
A.ry = B.by

5.5.6 Line Composition Example

Figure 5.16 contains an example to illustrate the use of the line composition operators. The grammar contained in the figure describes graphical objects consisting of any number of **line** components connected end-to-end (ignoring orientation), followed by an **arrow**. Figure 5.16 contains an object with this structure.

In the grammar, the LONGARC is built backwards from the **arrow** at one end. Production 2 add segments one at a time to the existing LONGARC. Productions 3 and 4 allow the orientation of the **lines** (i.e. which endpoint is the tail or head) to be ignored. The *lastX* and *lastY* attributes prevent the same line segment from being added twice with opposite orientations.



(1) LONGARC → **arrow**
LONGARC.lastX = arrow.lx
LONGARC.lastY = arrow.by

(2) LONGARC → follow(LINE,LONGARC$_1$)
LONGARC.lastX = LINE.lx
LONGARC.lastY = LINE.by
Where
(LINE.lx != LONGARC$_1$.lastX) || (LINE.by != LONGARC$_1$.lastY)

(3) LINE → **line**

(4) LINE → reverse(**line**)

Figure 5.16: An Example of Line Compositions - a multisegment arc

5.6 Shape/Line Composition Operators

The class of composition operators discussed in this section serve to combine lines with shapes. That is, in applying these operators, one argument symbols is interpreted as a line and the other as a shape. The first four operators extend coincidence to a line and a shape. This coincidence occurs when a line has an endpoint that is somewhere along the boundary (i.e. extent) of a shape. The four operators differ on which endpoint of the line is involved and which object forms the aggregate. The last operator extends the notion of spatial adjacency to cover the case where one object is a line.

5.6.1 Touches_L, Touches_R

The *touches_L* and *touches_R* operators combines a shape and a line where an endpoint of the line lies anywhere on the boundary (i.e. the extent) of the shape. The aggregate object formed will copy the geometric attributes from the shape argument. These productions are used for combining lines and shapes, where the aggregate will then interact as a shape. For example, the StateCharts grammar in Appendix C.1 uses a production of the form:

$$\text{STATE} \rightarrow \text{touches\_L(ARC, STATE}_1\text{)}$$

to combine outgoing ARCS with the STATE they are leaving. The STATEs are later combined using shape compositions.

The *touches_L* operator requires the Left end of the line (i.e. the point (lx, by)) to be on the shape boundary. The production

$$\text{A} \rightarrow \text{touches\_L(L,S)}[i]$$

is equivalent to the attributed multiset grammar production

A → {L,S}
A.lx = S.lx
A.rx = S.rx
A.by = S.by
A.ty = S.ty
Where
((L.lx == S.lx || L.lx == S.rx) && S.by ≤ L.by ≤ S.ty)
|| ((L.by == S.by || L.by == S.ty) && S.lx ≤ L.lx ≤ S.rx)

Conversely, the *touches_R* operator requires the Right end of the line (i.e. the point (rx, ty)) to be on the shape boundary. The production

$$\text{A} \rightarrow \text{touches\_R(L,S)}[i]$$

is equivalent to the attributed multiset grammar production

A → {L,S}
A.lx = S.lx
A.rx = S.rx
A.by = S.by
A.ty = S.ty
Where
((L.rx == S.lx || L.rx == S.rx) && S.by ≤ L.ty ≤ S.ty)
|| ((L.ty == S.by || L.ty == S.ty) && S.lx ≤ L.rx ≤ S.rx)

5.6.2 Points_from, Points_to

The *points_from* and *points_to* operators also combine a shape and a line where an endpoint of the line lies anywhere on the boundary of the shape. For these two operators, however, the aggregate formed copies the geometric attributes from the line argument. This corresponds to the situation where a line and shape are combined and the resultant object then interacts as a line. For instance, in the expression tree grammar given in Appendix C.2, the production

$$\text{CHILD} \rightarrow \text{points\_to}(\textbf{arrow}, \text{NODE})$$

is used to combine an **arrow** with the NODE to which it points. The aggregate CHILD is subsequently combined with the originating NODE using the *touches_L* operator.

The *points_from* operator requires the left end of the line (i.e. (lx, by)) to be on the shape boundary.[2] The production

$$\text{A} \rightarrow \text{points\_from(L,S)}[i]$$

is equivalent to the attributed multiset grammar production

A → {L,S}
A.lx = L.lx
A.rx = L.rx
A.by = L.by
A.ty = L.ty
Where
((L.lx == S.lx || L.lx == S.rx) && S.by ≤ L.by ≤ S.ty)
|| ((L.by == S.by || L.by == S.ty) && S.lx ≤ L.lx ≤ S.rx)

The *points_to* operator requires the right end (i.e. (rx, ty)) of the line to be on the shape boundary. The production

$$\text{A} \rightarrow \text{points\_to(L,S)}[i]$$

is equivalent to the attributed multiset grammar production

A → {L,S}
A.lx = L.lx
A.rx = L.rx
A.by = L.by
A.ty = L.ty
Where
((L.rx == S.lx || L.rx == S.rx) && S.by ≤ L.ty ≤ S.ty)
|| ((L.ty == S.by || L.ty == S.ty) && S.lx ≤ L.rx ≤ S.rx)

5.6.3 Labels

The *labels* operator is used to combine a line with an adjacent shape. It uses a special type of adjacency, called *label adjacency*. Unlike the other types of adjacency, where both objects were considered shapes, in label adjacency one of the objects is treated as a line. The distinction is illustrated in Figure 5.17. When referring to shapes, adjacency

[2] *points_from* and *points_to* are named according to the convention observed by Green, that arrows point from the left end to the right end. There is no requirement that this be true (or even that the operators be used with **arrow** objects rather than any other type of line). *points_from* and *points_to* could just as easily be named *points_L* and *points_R* respectively.

Figure 5.17: a) Adjacent as Shapes; b) Shape Adjacent to Line

relies on the extents of the objects. However, a shape is adjacent to a line when it is along the length of the line, as shown in Figure 5.17b.

Labels adjacency is defined based on two conditions. The first condition specifies the spatial relationship between the line and the shape. For a shape object to be label adjacent to a line object, the shape must fall within the area bounded by two lines which are perpendicular to the line object and pass through its endpoints. At least one of the corners of the (extent of) the label object must lie within this area. Figure 5.18 demonstrates the situation. Shapes A, B and C all meet the relationship condition, but shapes D and E do not.

The second condition relates to intervening objects. Intuitively, if a shape is labelling a line, there should be nothing else between them. As with the previous types of adjacency, the labels adjacency constraint is defined by identifying an area where terminals are excluded. For labels adjacency, there are three cases, as shown in Figure 5.19. If the line is horizontal or vertical, the effected area is the region formed by the overlapping parts of the line and the closest parallel side of the shape. Figures 5.19a and 5.19b show these cases.

If the line is neither horizontal nor vertical, the region is formed between a corner of the shape and a point on the line. The closest corner which is within the label area described above is used. The other point used is the intersection point of the line object and a line perpendicular to the line object which passes through the selected corner. This region is illustrated in Figure 5.19c.

To build the adjacency region, the four corners of the label object (i.e., the points $(lx, by), (lx, ty), (rx, by), (rx, ty)$) are examined. The tangent from each corner to the line is computed, along with the point where the tangent intersects the line, as shown in Figure 5.20a. If a corner is within the label area, the tangent intersection point will be somewhere in the line segment corresponding to the line object.

The corner with the shortest tangent that intersects within the line object is selected, and the rectangle from the corner to the intersection used for the adjacency region. If the adjacency region overlaps the border of the rectangular extent of the line, it must be split into two regions, as shown in Figure 5.20b (if it overlaps both borders, it will be split into four regions). These regions are then checked separately for overlapping terminals. Splitting the region avoids finding an overlap with the terminals forming the line.

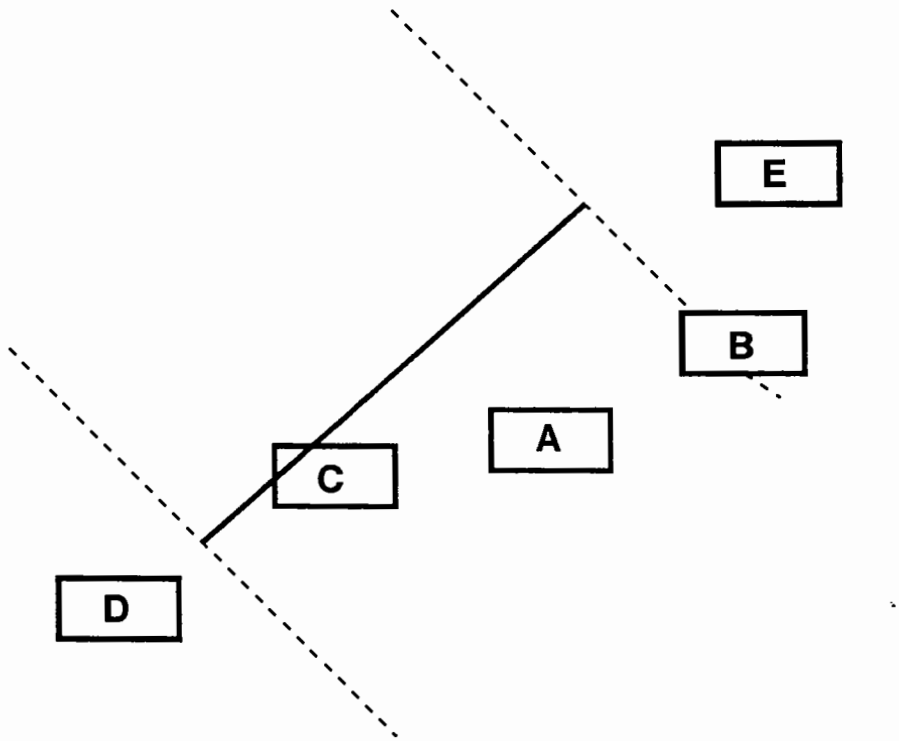


Figure 5.18: Spatial Relationship for Labels Adjacency

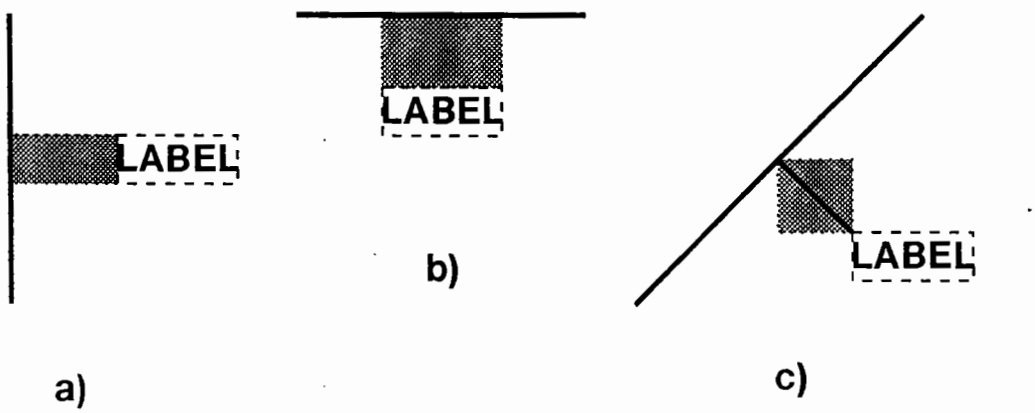


Figure 5.19: Adjacency Regions for Labels

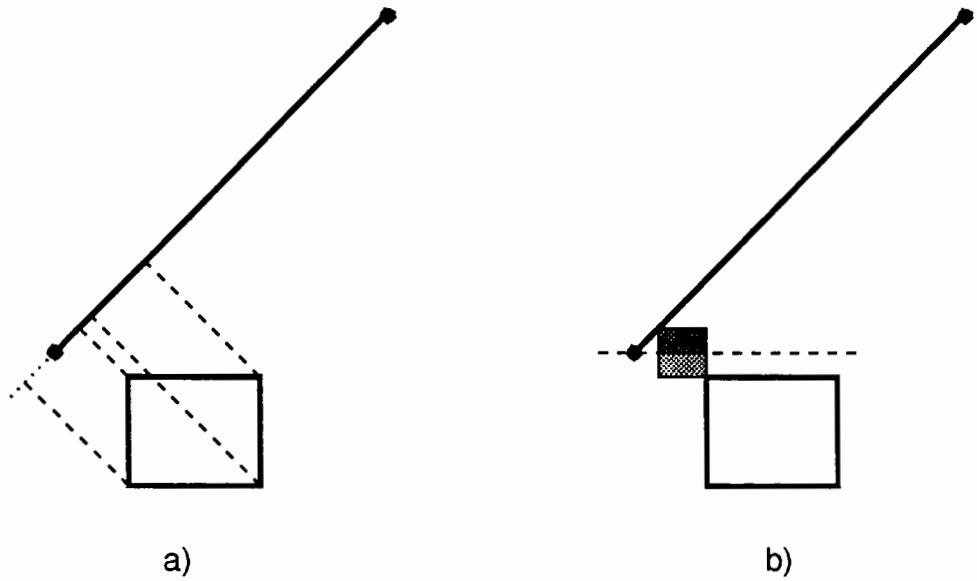


Figure 5.20: a) tangent lines; b) adjacency region; c) region splitting

The aggregate formed by the labels operator is a line object with the same endpoints as the original line. Intuitively, this corresponds to building a "labelled line". The production

$$A \rightarrow \text{labels(S,L)}[i]$$

is notation for the attributed multiset grammar production

A → {S,L}
A.lx = L.lx
A.rx = L.rx
A.by = L.by
A.ty = L.ty
Where
test_adjacent(ADJACENT_LABELS,@L,@S,i)

When called with the ADJACENT_LABELS type, `test_adjacent` checks both of the labels adjacency conditions.

Chapter 6

Spatial Parsing

6.1 Overview

The utility of a language specification derives from the processing that it allows. The specification mechanism described here has been designed to permit a particular type of processing: syntax analysis of programs. The advantages of using a language specification to analyze the syntactic structure of a program were discussed in Chapter 3. The syntax analysis of a visual program is called *spatial parsing*.

The approach to parsing works in several phases as shown in Figure 6.1. To begin with, there is a set of primitive graphical elements. These must be preprocessed to transform the primitive elements into an attributed multiset of terminal symbols. In the system developed, this mapping is performed by the GREEN environment. Normally each primitive element will map to a single terminal symbol.

The first phase of the parser is the *Build* phase. The Build phase analyzes the picture and constructs a *factored multiple derivation (FMD)* structure. The FMD structure is essentially a type of factored parse tree [94, 24]. It represents the set of all possible analyses of the program. The building phase works bottom-up, so the FMD structure may also contain analyses which are not valid. The second phase verifies that the FMD contains a correct parse and extracts a single analysis from the FMD.

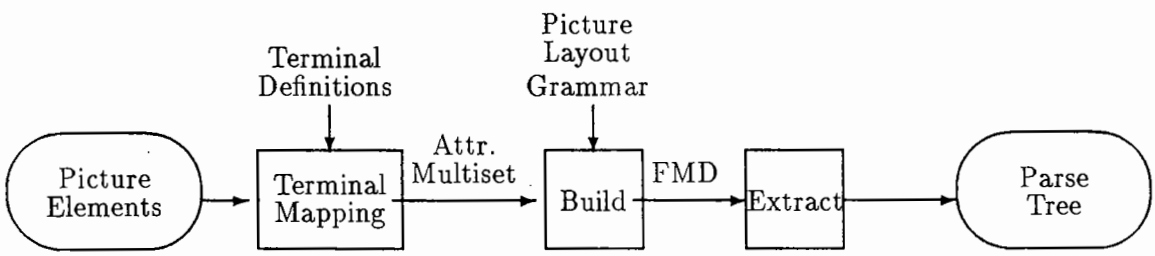


Figure 6.1: Parser Architecture

This chapter discusses the spatial parsing algorithm and its implementation. Section 2 defines the FMD structure used to represent a parse. Section 3 describes the Build phase and Section 4 discusses the implementation of the attribute and constraint mechanism. Section 5 covers the extraction phase. Finally, Sections 7 and 8 analyze the complexity and correctness of the parsing algorithm.

6.2 The Factored Multiple Derivation Structure

The basic idea of the factored multiple derivation structure is to create a parse tree where the leaves are labeled by terminal symbols and the interior nodes are labeled by productions. Each non-terminal node will have multiple sets of pointers, called *child lists*, associated with it. A child list is an ordered list of nodes corresponding to the elements of the RHS of the production labelling the node.

In a traditional factored parse tree, these child lists form an interleaved set of trees. In other words, if each node had only one child list, the resulting structure would be one or more disjoint parse trees. In a picture layout grammar, one or more terminal symbols on the right-hand side may be marked as remote. These remote nodes form the context of the production, as described in Chapter 4.

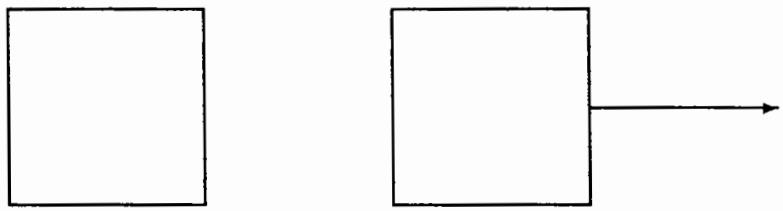
The edges corresponding to these elements are not part of the tree structure of the parse – they are additional, nontree edges. Thus the structure formed by the FMD is an interleaved set of tree-DAGS. The FMD structure could also be termed a *factored parse tree-DAG*. We will continue to use the terms "factored parse tree" and "factored parse tree node", with the understanding that they actually refer to tree-DAG structures.

In addition to a terminal symbol or production, a node in the FMD is labelled by an attribute vector. The attribute vector contains the values of the attributes of the symbol labelling that node (i.e. either the terminal symbol or the LHS of the production). These attributes will always include the four values *lx, by, rx, ty*, but may also have other values as defined by the grammar.

The FMD structure is considered to be *factored* because all parse tree nodes which are labelled with the same production (or terminal symbol) and attribute values are grouped together into a single node. Thus no two nodes in the FMD structure will have exactly the same label. Each child list under a node represents one "unfactored" parse tree node which has been grouped together.

To see how this works, consider the fragment of a visual program shown in Figure 6.2a. The productions in Figure 6.2b might specify this part of the program. The rectangle on the right could be parsed using either of the first two productions. The parse tree nodes for these would have the same attributes, since both productions copy the attributes of the rectangle. The TWO_BOXES production could be applied in two cases, depending on which production was used to analyze the box on the right. Figure 6.2c shows the FMD structure for this fragment. The TWO_BOXES node has two child lists, one for each application of the production.

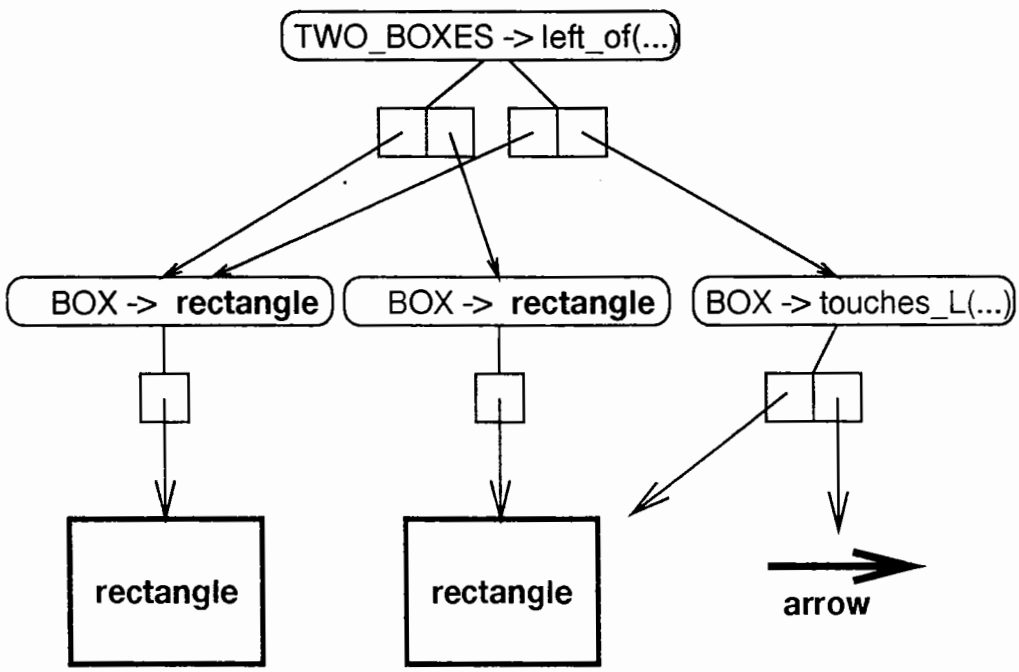
A formal definition of the FMD structure is now given, beginning with the definition of a factored parse tree node.



a) A visual program fragment

(1) BOX → **rectangle**
(2) BOX → touches_L(**rectangle**, **arrow**)
(3) TWO_BOXES → left_of(BOX_1, BOX_2)

b) PLG productions for picture above



c) FMD nodes for picture

Figure 6.2: A Visual Program Fragment

Definition 6.1 *Let $G = (N, \Sigma, s, I, \mathcal{D}, P)$ be a picture layout grammar. A* factored parse tree node (FPTN) *is a triple $F = [L, E, C]$, where*

- *$L \in \Sigma \cup P$ is a label. If $L \in \Sigma$, then F is called a terminal node, otherwise it is called a non-terminal node.*
- *$symb(F) = X_F \in \Sigma \cup N$ is the* symbol class *of the node. For $L \in \Sigma$ (terminal node), $X = L$. Otherwise $L \in P$ (nonterminal node), and X is the symbol on the left-hand side of the production L.*
- *E is an attribute vector such that (X_F, E) form an attributed symbol. E is the* attribute index *of the node.*
- *If F is a terminal node, then $C = \emptyset$. For a nonterminal node, $C = \{C_1, C_2, ..., C_t\}$ is a set of* child lists. *Each child list $C_j = \langle r_1, r_2, ..., r_m \rangle$ is a list of references (i.e. pointers to factored parse tree nodes). Here m is the number of elements of the RHS of the production.*

The symbol class of a parse tree node ($symb(n)$) is the symbol associated with the node. It is the syntactic entity that the node represents. The attribute index of a node is a list of values for the attributes of the symbol class of the node. It corresponds to the physical properties of the specific syntactic entity represented by the node.

Given a non-terminal factored parse tree node F, a *selector* is a pair (C_i, r_j) selecting a child list and a reference within the child list to determine a target node. An FPTN F' is a child of an FPTN F if F' is the target of some selector of F. A *path* from FPTN F_0 to node F_n is a (nonempty) sequence of selectors $S_0 S_1 ... S_{n-1}$ such that F_1 is a child of F_0 with selector S_0, F_2 is a child of F_1 with selector S_1, ..., and F_n is a child of F_{n-1} with selector S_{n-1}. If there is a path from F_0 to F_n, then F_0 is an ancestor of F_n and F_n is a descendant of F_0.

Definition 6.2 *A* factored parse tree *is a set of factored parse tree nodes T such that the following is true:*

1. *If $F = [L, E, C] \in T$ is a non-terminal node, then for every child list $C_j \in C$, the reference r_i is a pointer to a factored parse tree node $F' \in T$ where the symbol class of F' is the same as the i'th element of the RHS of L, the production labelling F.*
2. *There is one distinguished node $ROOT \in T$ which is not pointed to by any reference of another node in T.*
3. *Every node in T other than $ROOT$ is pointed to by at least one reference of another node in T.*
4. *There are no cycles in T. That is, there is no node $F \in T$ such that there is a path from F to itself.*

5. *If $F = [L, E, C]$ and $F' = [L', E', C']$ are distinct nodes in T, then it cannot be that $L = L'$ and $E = E'$. In other words, no two nodes may be labelled by both the same production and the same attribute index.*

It is clear that, because of condition (1), a factored parse tree consists of terminal symbols at the leaves and nonterminal symbols (with productions) at the interior nodes. The yield of a factored parse tree is the multiset of terminal symbols found at its leaves.

A factored parse tree represents a set of possible derivations. Each child list is an alternative derivation which uses the same production and synthesizes the same attribute values. The spatial parsing algorithm constructs factored parse trees for all the possible analyses of the input multiset. The final definition describes the data structure constructed by the parser, a collection of interleaved factored parse trees.

Definition 6.3 *A* factored multiple derivation structure (FMD) *is a set of factored parse tree nodes T such that every node $F \in T$ is the root of a factored parse tree in T. That is, for every node $F \in T$, there exists a subset $T_F \subseteq T$ such that T_F forms a factored parse tree with F as its root.*

6.3 Building the FMD

The first phase of the parser takes an attributed multiset as input and constructs a factored multiple derivation structure representing all possible valid analyses of the input. Intuitively, the algorithm builds the FMD structure bottom-up, operating directly on the factored parse tree nodes. The leaves of the FMD structure contain the input terminal symbols and the interior nodes of the FMD structure correspond to applications of productions. Construction of the FMD structure proceeds by starting at the bottom; finding nodes which form the right-hand side of a production; and applying that production to add a new node (or child list) to the FMD.

The building algorithm is similar in spirit to the Cocke-Younger-Kasami algorithm for parsing arbitrary context free (string) languages [1]. The CYK algorithm parses a string by first parsing all substrings of length one, then all substrings of length two and so on. The parses are represented by a triangular parse table, where each element is a set of nonterminals corresponding to possible analyses of a particular substring. To parse a string of length n, all possible ways of dividing the string into two shorter substrings are considered. For each division, the parse table gives the possible nonterminals for the two pieces, and every production which could be applied is discovered and added to the table entry for the string.

My algorithm differs from the CYK algorithm in two respects. First of all, we represent the possible parses using an FMD structure, rather than a table (although CYK can be modified to use a factored parse tree). More importantly, the CYK algorithm is designed for strings, rather than two-dimensional pictures. The structure of the parse table reflects both the linear nature of the input and the tree structure of the analysis.

The basic algorithm for building the FMD structure is examined first. For now, we assume that the picture layout grammar does not use the *tiling* production operator, so every production has either one or two arguments on the right-hand side. Extensions to handle *tiling* productions are described below.

The basic algorithm is shown in Figure 6.3. It works as follows. The FMD structure will contain all the factored parse tree nodes generated. The nodes in FMD are divided into two queues: *done* and *todo*. The *done* queue contains the nodes which have already been processed, and the *todo* queue contains the as yet unprocessed nodes. When a new node is created, it is initially put on the *todo* queue and will be moved to the *done* queue by the BUILD procedure.

The build algorithm begins by creating a factored parse tree node for each terminal in the input multiset. These nodes form the leaves of the FMD structure. The main loop of the algorithm is driven by the *todo* worklist. Each node is taken off the *todo* queue, processed, and added to the *done* queue. The processing may generate new factored parse tree nodes which are added to the worklist. It is shown below that the total number of nodes which are generated is limited, so the worklist will eventually be emptied, at which point the algorithm terminates.

The processing for each node searches through the productions to find those which could have the node in the right-hand side. If the right-hand side only has one operand (i.e. a terminal, renaming or **reverse** production), then the algorithm checks that the symbol class of the node is correct and that all constraints for the production are satisfied. When these conditions are met, the production which can be applied to this node.

If the right-hand side of the production has two operands, then the symbol class of the current node must be compared with the desired class for both of the operands. When the current node is the correct class for one or both of the operands, the list of processed nodes is searched for the other operand. Each node in the *done* queue is examined for the appropriate symbol class. A candidate pair is checked against all constraints for the production. Again, when these conditions are met, an applicable production has been found.

Once an applicable production has been found, the algorithm performs the *add a new node for p to todo and FMD* step. This step can actually add either a new node or a new child list, depending on the production applied and the state of the FMD. First a new factored parse tree node is created for the production. The semantic functions of the production are used to compute the attributes of the node. The entire FMD structure is then searched for an existing node labelled by the same production and attribute index. If none is found, the new node is added to both the FMD structure and the *todo* queue. Otherwise a new childlist is added to the existing node and the new node is discarded. The new childlist will point to the node(s) to which the new node had pointed.

When all applicable productions have been considered, the (original) node is moved from the unprocessed queue (*todo*) to the processed queue (*done*). When all nodes have been moved to the processed queue the algorithm is completed. Because the nodes are factored (that is, all nodes with the same production and attributes are grouped

```
INPUT: An attributed multiset grammar G = (N, Σ, I, S, P) and an attributed
    multiset M.
OUTPUT: a factored multiple derivation structure FMD, such that every deriva-
    tion of M has a corresponding complete parse tree in FMD.
INITIALIZATION: set FMD= ∅, todo = ∅ and done = ∅.

for each b ∈ M do
    add a terminal node for b to todo and FMD
endfor
MAIN LOOP:
while todo not empty do
    let next = an element of todo, with X = symb(X)
    for each p ∈ P such that X occurs in the RHS of p do
        if (p = A → {X})
         and (all constraints are satisfied) then
            add a node for p to todo and FMD
        endif
        else do
            /* here p = A → M, with X ∈ M */
            for each occurance of X in RHS of p do
                let Y be the other symbol in p
                for each old in done such that Y = symb(old) do
                    if all constraints are satisfied then
                        add a node for p to todo and FMD
                    endif
                endfor
            endfor
        endelse
    endfor
    remove next from todo and add it to done
repeat
```

Figure 6.3: Basic Build Algorithm

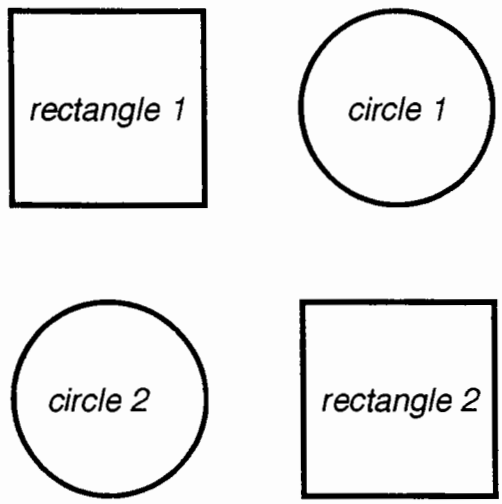


Figure 6.4: A Sample Visual Program

together as a single node with multiple child lists), the number of nodes which can be added is limited whenever the range of attribute values is finite (i.e. the grammar is finitely representable). This is discussed further in Section 6.6.

6.3.1 Examples of the Build Phase

Here are two examples to illustrate how the build procedure works. Consider the following grammar,

(1) PICTURE → PICT_LEFT
(2) PICTURE → PICT_RIGHT
(3) PICT_LEFT → SQ_LEFT
(4) PICT_LEFT → over(SQ_LEFT,PICT_RIGHT)
(5) SQ_LEFT → left_of(**rectangle**,**circle**)
(6) PICT_RIGHT → SQ_RIGHT
(7) PICT_RIGHT → over(SQ_RIGHT,PICT_LEFT)
(8) SQ_RIGHT → left_of(**circle**,**rectangle**)

This grammar defines the visual language SIMPLE1, where the terminal symbols are **rectangle**s and **circle**s. The language SIMPLE1 consists of pictures containing vertically composed rows alternating between a row with a **rectangle** to the left of a **circle** and a row with a **circle** to the left of a **rectangle**.

Figure 6.4 shows a picture which is an element of SIMPLE1. This program consists of four picture elements: $\mathbf{rectangle}_1$, $\mathbf{rectangle}_2$, $\mathbf{circle}_1$ and $\mathbf{circle}_2$. The build phase begins with a factored parse tree node for each terminal symbol, as shown in Figure 6.5a. When the first node (say $\mathbf{rectangle}_1$) is processed, no new nodes will be

created, since there are no productions which apply only to a **rectangle** node and no other processed nodes.

Suppose that the **circle**$_1$ node is processed next. Build will find Production 5, which can be applied to **circle**$_1$ and the previously processed **rectangle**$_1$ node. A new factored parse tree node will be created. The new node will be labelled by Production 5 with symbol SQ_LEFT, and will have a single child list pointing to both **rectangle**$_1$ and **circle**$_1$.

Note that Production 8 will also be found, since its operands consist of a **rectangle** and a **circle**. However, the constraint associated with the *left_of* production operator will not be satisfied, so Production 8 cannot be applied and no factored parse tree node is generated. The FMD structure after the first two nodes are processed is shown in Figure 6.5b, with the processed nodes (i.e. those on the *done* queue) shaded.

The next two nodes are processed similarly. This time only Production 8 is applicable, so a node labelled SQ_RIGHT is added, leaving the situation shown in Figure 6.5c.

Now the previously added SQ_LEFT node is processed, adding a node for Production 3. Production 4 cannot be applied, since there is no PICT_RIGHT node in the *done* queue. The SQ_RIGHT node is handled similarly, leaving the FMD structure as shown in Figure 6.5d.

The PICT_LEFT node is now processed and Production 1 applied, generating a PICTURE node. The PICTURE node signifies that the upper **circle** and **rectangle** form a valid picture by themselves. Because the analysis is bottom-up and carries forward all possibilities, this is found as one possible picture. In the larger context, the parser will discover that this PICTURE is not valid because it does not generate all the input.

The PICT_RIGHT node is similarly processed and a PICTURE node created. In addition, Production 4 can now be applied, generating a new PICT_LEFT node. As you can see, the order in which the nodes are taken from the *todo* queue and processed is not important. The new PICT_LEFT will be processed as before, yielding a PICTURE node. Since PICTURE does not appear on the right-hand side of any productions, no new nodes are generated when the PICTURE nodes are processed. The FMD structure after the building phase is shown in Figure 6.6.

Now an example demonstrating factoring. Consider the productions:

(1) BOX → **rectangle**
(2) BOX → contains(**rectangle**, **circle**)
(3) TWO_BOXES → left_of(BOX$_1$, BOX$_2$)

This grammar defines the language SIMPLE2, containing pictures consisting of two horizontally adjacent BOXes, where each BOX is a **rectangle**, possibly containing a **circle**. The picture in Figure 6.7 is an element of SIMPLE2, consisting of three terminal symbols.

When this picture is analyzed, the build phase will apply Production 1 to both **rectangle**$_1$ and **rectangle**$_2$, creating two BOX nodes (BOX$_2$ and BOX$_3$ in Figure 6.8). The parser will also apply Production 2 to **rectangle**$_1$ and **circle**$_1$, creating node BOX$_1$. Figure 6.8 shows the FMD structure after the three terminal nodes have been

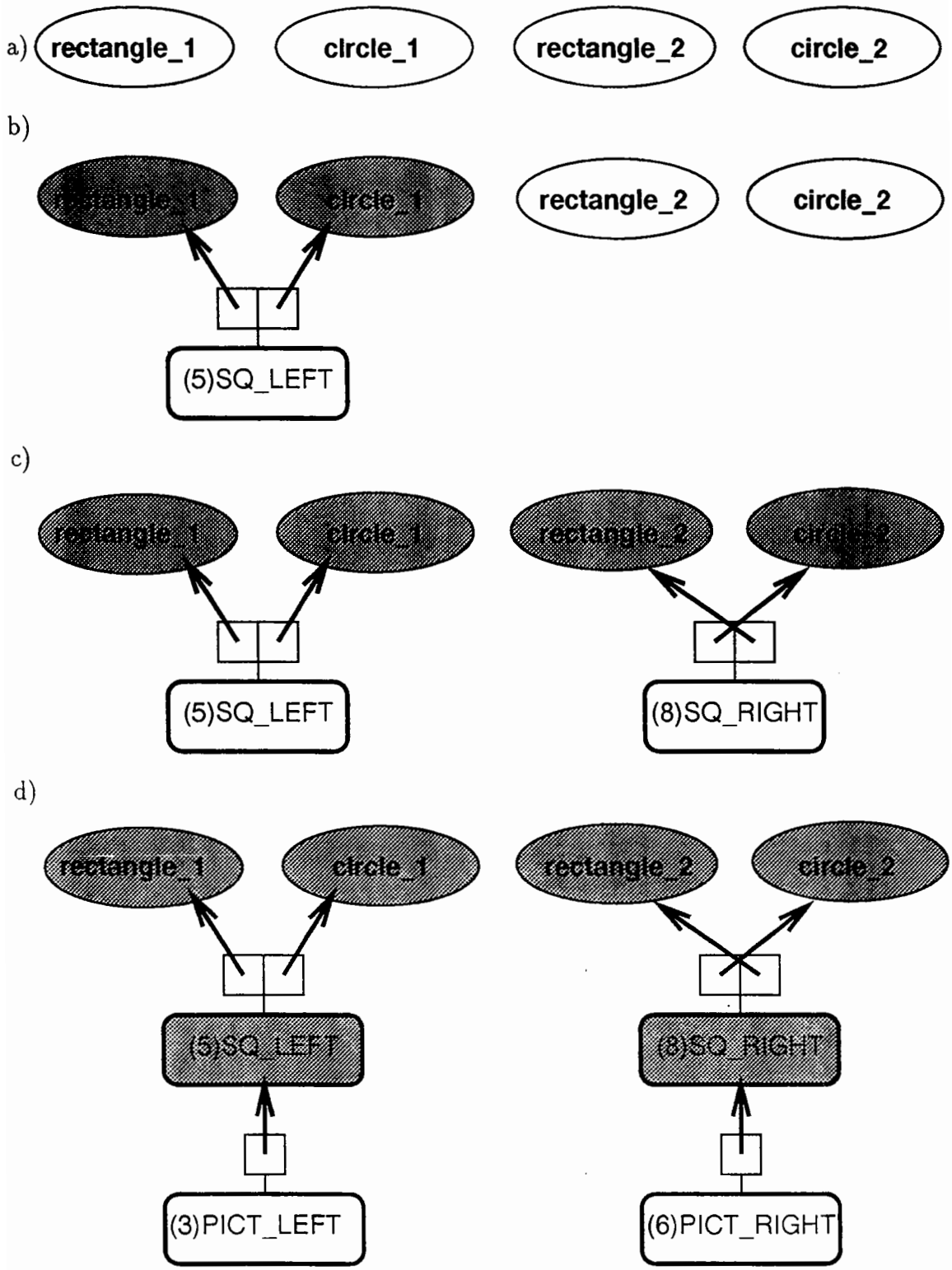


Figure 6.5: Building the FMD Structure

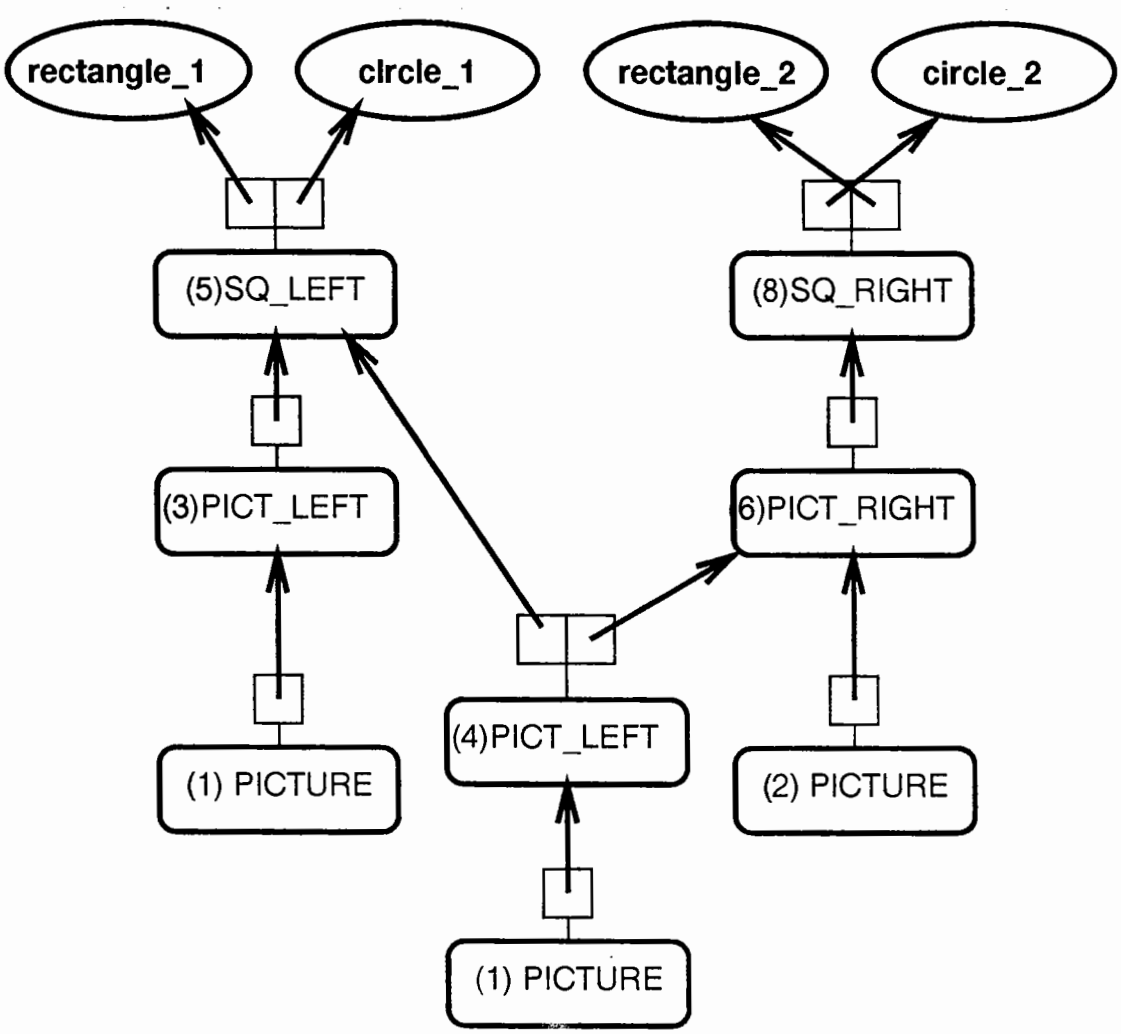


Figure 6.6: The Complete FMD Structure

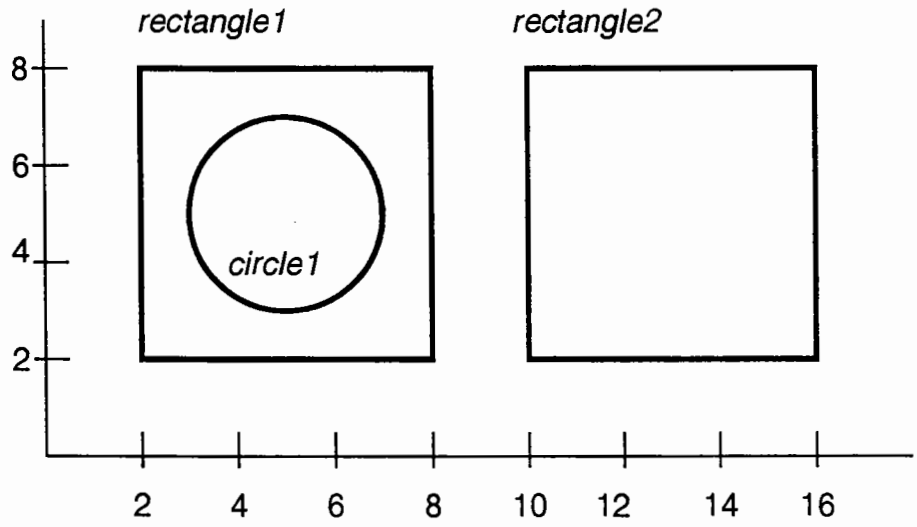


Figure 6.7: Another Simple Visual Program

processed. This figure shows the attributes ($[lx, by, rx, ty]$) associated with each node along with the symbol/production labelling it. Note that BOX_1 and BOX_2 have the same attribute index (i.e. the same extent), but are not combined because they are labelled by different productions.

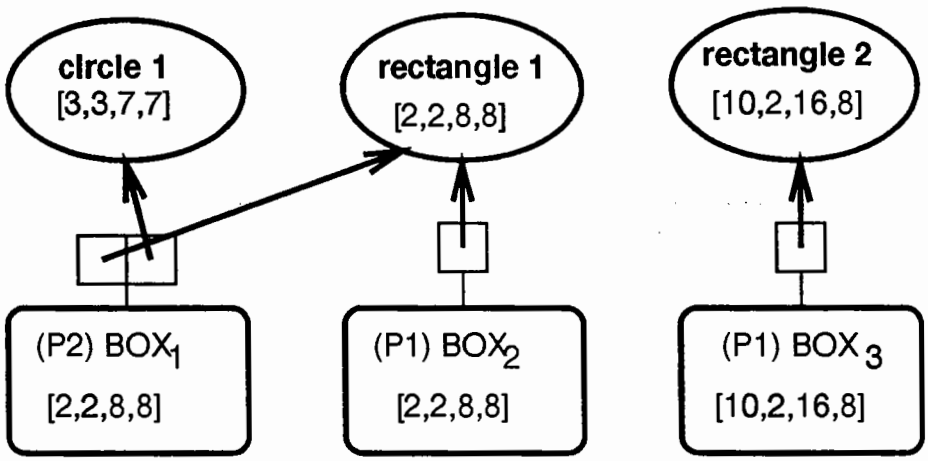
After processing the nodes BOX_1 and BOX_2 and adding them to the *done* queue, the build procedure processes the node BOX_3. First Production 3 is applied to nodes BOX_1 and BOX_3, creating a TWO_BOXES node with extent $[2, 2, 16, 8]$. This production can also be applied to the nodes BOX_2 and BOX_3, resulting in another node labelled by Production 3 and the attributes $[2, 2, 16, 8]$. Rather than adding this new node to the FMD and *todo* queue, a new child list is added to the existing TWO_BOXES node, pointing at nodes BOX_2 and BOX_3. The two TWO_BOXES nodes have been factored by the label. The FMD structure after the build phase is shown in Figure 6.8b. Later phases of the parser must determine which child list of the TWO_BOXES node is the correct one.

6.3.2 Parsing with *tiling* Productions

The discussion above assumed that the right-hand side of the production had at most two operands. While most languages can be described using binary production operators, some constructs can not be conveniently expressed in this manner. For example, consider the irreducible tiling shown in Figure 6.9. This picture cannot be easily described using the binary operators defined in Chapter 5.

The *tiling* operator provides a means of specifying productions with an arbitrary number of operands. Only those constraints and semantic functions explicitly specified

a)



b)

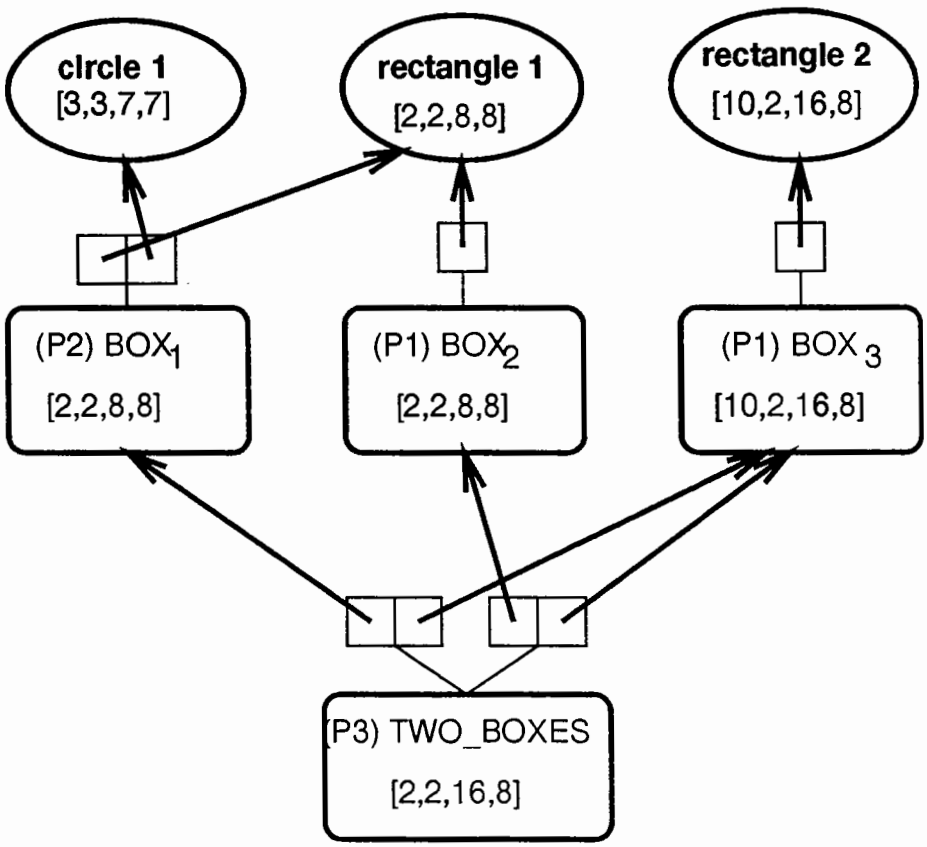


Figure 6.8: An FMD with Factoring

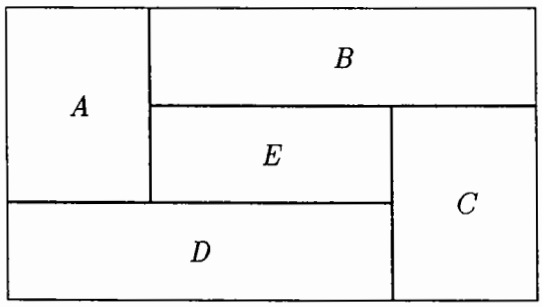


Figure 6.9: An Irreducible Tiling

by the language designer are used. The picture in Figure 6.9 is described by the production

PICTURE → tiling(BLOCK_A,BLOCK_B,BLOCK_C,BLOCK_D,BLOCK_E)
PICTURE.lx = BLOCK_A.lx
PICTURE.rx = BLOCK_C.rx
PICTURE.by = BLOCK_D.by
PICTURE.ty = BLOCK_B.ty
Where
BLOCK_A.lx == BLOCK_D.lx && BLOCK_A.by == BLOCK_D.ty
&& BLOCK_A.ty == BLOCK_B.ty && BLOCK_A.rx == BLOCK_D.lx
&& BLOCK_B.rx == BLOCK_C.rx && BLOCK_B.by == BLOCK_C.ty
&& BLOCK_C.by == BLOCK_D.by && BLOCK_C.lx == BLOCK_D.rx
&& BLOCK_E.lx == BLOCK_A.rx && BLOCK_E.ty == BLOCK_B.by
&& BLOCK_E.rx == BLOCK_C.lx && BLOCK_E.by == BLOCK_D.ty

Although this picture cannot be decomposed using the binary operators defined in Chapter 5, it can be broken down into productions combining two elements at a time. The production above can be rewritten as four productions as shown in Figure 6.10. These productions use four additional attributes Arx, Aby, Bby, Clx to "remember" the exact position required of subsequent pieces of the tiling. These attributes are used in constraints that correspond to clauses in the constraint of the original production.

Requiring the language designer to specify his language in this form has two problems. First of all, it introduces many unnecessary nonterminal symbols into the grammar and obscures the actual structure of the language. The four productions in Figure 6.10 are somewhat harder to understand than the single production (particularly if the constraint in the single production could be expressed graphically). A second problem is the need to introduce additional attributes to represent the characteristics of the aggregate objects, again making the grammar harder to read.

Extending the building algorithm described above provides for right-hand sides with more than two operands. The modified algorithm is shown in Figure 6.11. The modified parser automatically performs the decomposition shown in Figure 6.10. Intuitively, the extension can be thought of as defining new nonterminals to represent pieces of the right-hand side, and then combining those pieces to form the entire production.

The modified algorithm uses the following mechanism to implement this extension.

```
PICT1 → tiling(A,B)
        PICT1.lx = BLOCK_A.lx
        PICT1.ty = BLOCK_A.ty
        PICT1.rx = BLOCK_B.rx
        PICT1.by = BLOCK_A.by
        PICT1.Aby = BLOCK_A.by
        PICT1.Arx = BLOCK_A.rx
        PICT1.Bby = BLOCK_B.by
Where
        BLOCK_A.ty == BLOCK_B.ty & & BLOCK_A.rx == BLOCK_B.lx

PICT2 → tiling(C,PICT1)
        PICT2.lx = PICT1.lx
        PICT2.rx = PICT1.rx
        PICT2.ty = PICT1.ty
        PICT2.by = BLOCK_C.by
        PICT2.Aby = PICT1.Aby
        PICT2.Arx = PICT1.Arx
        PICT2.Bby = PICT1.Bby
        PICT2.Clx = BLOCK_C.lx
Where
        BLOCK_C.rx == PICT1.rx && BLOCK_C.ty == PICT1.Bby

PICT3 → tiling(D,PICT2)
        PICT3.lx = PICT2.lx
        PICT3.rx = PICT2.rx
        PICT3.ty = PICT2.ty
        PICT3.by = PICT2.by
        PICT3.Aby = PICT2.Aby
        PICT3.Arx = PICT2.Arx
        PICT3.Bby = PICT2.Bby
        PICT3.Clx = PICT2.Clx
Where
        BLOCK_D.lx == PICT2.lx && BLOCK_D.by == PICT2.by
        && BLOCK_D.rx == PICT2.Clx && BLOCK_D.ty == PICT2.Aby

PICTURE → tiling(BLOCK_E,PICT3)
        PICTURE.lx = PICT3.lx
        PICTURE.rx = PICT3.rx
        PICTURE.ty = PICT3.ty
        PICTURE.by = PICT3.by
Where
        BLOCK_E.lx == PICT3.Arx && BLOCK_E.by == PICT3.Aby
        && BLOCK_E.rx == PICT3.Clx && BLOCK_E.ty == PICT3.Bby
```

Figure 6.10: Decomposing a *tiling* Production

INPUT: An attributed multiset grammar $G = (N, \Sigma, I, S, P)$ and an attributed multiset M.
OUTPUT: a factored multiple derivation structure FMD, such that every derivation of M has a corresponding complete parse tree in FMD.
INITIALIZATION: set FMD= $\emptyset$, *todo* = $\emptyset$ and *done* = $\emptyset$.

```
for each a ∈ M do
    add a terminal node for a to todo and FMD
endfor
MAIN LOOP:
while todo not empty do
    let next = an element of todo, with X =symb(X)
    if next is a complete node then
        for each p ∈ P such that X occurs in the RHS of p do
            if (p = A → {X})
             and (all constraints are satisfied) then
                add a node for p to todo and FMD
            endif
            else do
                /* here p = A → M, with X ∈ M */
                for each occurance of X in RHS of p do
                    add an incomplete node for p and X
                endfor
            endelse
        endfor
    endif
    else do
        /* here next is incomplete */
        for all incomplete nodes old ∈ done such that old is labelled by p do
            if next and old can be combined then
                let new be the combination of next and old
                if all constraints are satisfied then
                    add new to todo and FMD
                endif
            endif
        endfor
    endelse
    remove next from todo and add it to done
repeat
```

Figure 6.11: Modified Build Algorithm

Each factored parse tree node has a *bit mask*, with a bit for each element of the right-hand side. This mask will be used to indicate which elements of the right hand side are actually present in a node, with a 1 bit signifying the corresponding operand is present.

A factored parse tree node with all the bits in its mask set to 1 is called a *complete* node. A node with some bits set to 0 is an *incomplete* node, and represents a partial derivation of a production. Incomplete nodes are not factored, so each incomplete node will have exactly one child list. The incomplete nodes can be thought of as the new "nonterminal" symbols added to the grammar, decomposing the production into binary productions.

The modified build algorithm operates in generally the same manner as the basic algorithm. It begins by creating nodes for all the terminal symbols (these nodes are also considered complete) and then uses a worklist of nodes to be processed. Both the incomplete and complete nodes will be placed on the *todo* queue, then (at some point) removed, processed and placed on the *done* queue.

What is different in the new build algorithm is the processing of the nodes. Incomplete and complete nodes are handled as seperate cases. The processing for a complete node still searches for all productions which could have the node on the right-hand side. Any productions which only have a single element on the right-hand side (e.g. a renaming production $A \rightarrow B$) are processed as before (i.e. a new node is generated and added to *todo* and FMD).

The new processing is for productions with two or more elements on the right-hand. Instead of searching the *done* queue for the remaining elements of the right-hand side, an incomplete node is formed and added to the worklist. The bit mask will have a one bit corresponding to the element of the right-hand side which matches the current node. If the current node matches more than one element of the right-hand side, several incomplete nodes may be added.

Incomplete nodes are handled differently. Instead of looking for productions using this node, the queue of processed nodes is searched for other incomplete nodes labelled by the same production. If a matching node can be combined with the current node and the constraints are satisfied, then the two nodes are combined and the combined node is added to the *todo* queue. Any constraints referencing elements of the right-hand side not present in the combined node are assumed to be satisfied. The two incomplete nodes are combined by creating a new node and merging the child lists of the two incomplete nodes. (Thus incomplete nodes are never pointed to by child lists). The new node can be either complete or incomplete, depending on whether all elements of the right-hand side are present.

One question remaining is when is it possible to combine two incomplete nodes? Clearly the two masks cannot have any 1-bits in common, since the two nodes must refer to disjoint pieces of the right-hand side. Using this as the only restriction would lead to generating all possible partial nodes. For a node with more than two elements in the right-hand side, this approach would generate the same node for every possible partition of the right-hand side. For example $A \rightarrow \{B, C, D\}$ would generate the same A node three times, depending on which two nodes were grouped first.

To avoid this, the canonical ordering from the right-hand side of the production is used. Incomplete nodes are only combined when the combined node is a prefex of the canonical ordering. That is, the first and second elements can be combined, then those two can be combined with the third, and so on. This approach essentially models the decomposition given in Figure 6.10.

For example, the production

PICTURE → tiling(**rectangle**$_1$,**rectangle**$_2$,**rectangle**$_3$,**rectangle**$_4$)
Where
rectangle$_1$.rx == **rectangle**$_2$.lx
&& **rectangle**$_2$.rx == **rectangle**$_3$.lx
&& **rectangle**$_3$.rx == **rectangle**$_4$.lx

specifies the picture shown in Figure 6.12a. The parse tree nodes with child lists are shown in Figure 6.12b. The incomplete nodes are shaded. Note that since the analysis is done bottom-up, each rectangle could potentially occupy any position in the right-hand side of the production (so each **rectangle** node should have three other incomplete PICTURE nodes above it with the **rectangle** in the other three positions - these nodes are omitted from Figure 6.12b). It is incumbent on the language designer to specify sufficient constraints in the production to limit the ambiguity of the *tiling* construct.

The incomplete node/bit mask approach is somewhat more general than is needed. For example, productions with only two elements in the right hand side could be processed without creating incomplete nodes. This would improve performance slightly, but complicates the presentation and discussion of the algorithm by introducing more special cases.

In addition, a counter could be used in place of a bit mask to indicate what part of the right-hand side is contained in an incomplete node. The advantage of a bit mask is that it can also be used for other, more general groupings of the elements of the right-hand side elements. For example, multiple compositions could be permitted in a single production, as in

A → over(B,left_of(C,D))

The bit mask mechanism can be used to group together the C and D elements first, then combine them with the B element.

6.4 Attribute Evaluation and Constraint Checking

The attributes in a picture layout grammar are an integral part of the parsing. The constraints associated with a production are based on the attributes of the right-hand side of the production. The attributes are also used in factoring the parse structure. Thus the attributes must be evaluated as the FMD structure is built. The build phase evaluates the semantic functions and applies the constraints for all productions. Because the parsing attributes are all synthesized, they are available for evaluation of constraints.

The input to the build phase is a set of attributed terminal symbols, and the attribute values for each terminal are stored in the attribute index (i.e. attribute

vector) of a factored parse tree node. Similarly, the attribute index of a nonterminal factored parse tree node stores the attribute values of the nonterminal symbol on the left-hand side of the associated production. The attributes of the right-hand symbols are stored in the attribute indices of the factored parse tree nodes for those symbols, and are accessed by following pointers from a child list.

The implicit constraints (i.e. the constraints asociated with the builtin production operators) are implemented directly by the parser. Each production operator *op* has two associated routines *op_semfun()* and *op_constraint()*. The *op_semfun* routine computes the appropriate attributes for a node with the given production operator. The *op_constraint* routine accesses the attributes of the right-hand side to check the appropriate condition.

The explicit user-specified constraints and semantic functions are stored with the production. A semantic function consists of a reference to an attribute of the left-hand side and an attribute expression. A semantic function is evaluated by evaluating the attribute expression and storing the result in the attribute index entry for the specified attribute.

A constraint is either a simple constraint or a logical expression. A simple constraint consists of two attribute expressions and a relational operator. It is evaluated by first evaluating both of the expressions and then performing the indicated comparison. The result of simple constraint is a Boolean value (zero or one). A logical expression is formed from two constraints combined by a logical operation. It is evaluated by checking the two constraints and combining the results using the specified operation.

Attribute expressions come in seven varieties. The two simplest are integer and string constants, which simply evaluate to themselves. An attribute reference is just that - a reference to a specific attribute instance in the right-hand side of the production. For example, in the production

$$\text{A} \rightarrow \text{left\_of}(\textbf{rectangle}_1, \textbf{rectangle}_2)$$

$\textbf{rectangle}_1$.rx is a reference to the *rx* attribute of the $\textbf{rectangle}_1$ element of the right-hand side. An attribute reference is evaluated by first finding the factored parse tree node for the appropriate element of the right-hand side and then retrieving the indicated attribute value from the attribute index. The retrieved value is the result.

These terms can be formed into three types of complex expressions:

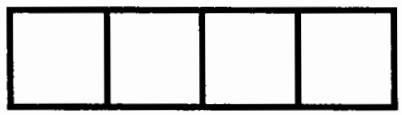
1. unary expressions, using the unary minus operator.

2. binary expressions, using one of the four binary integer arithmetic operators: '+','−','*','/'.

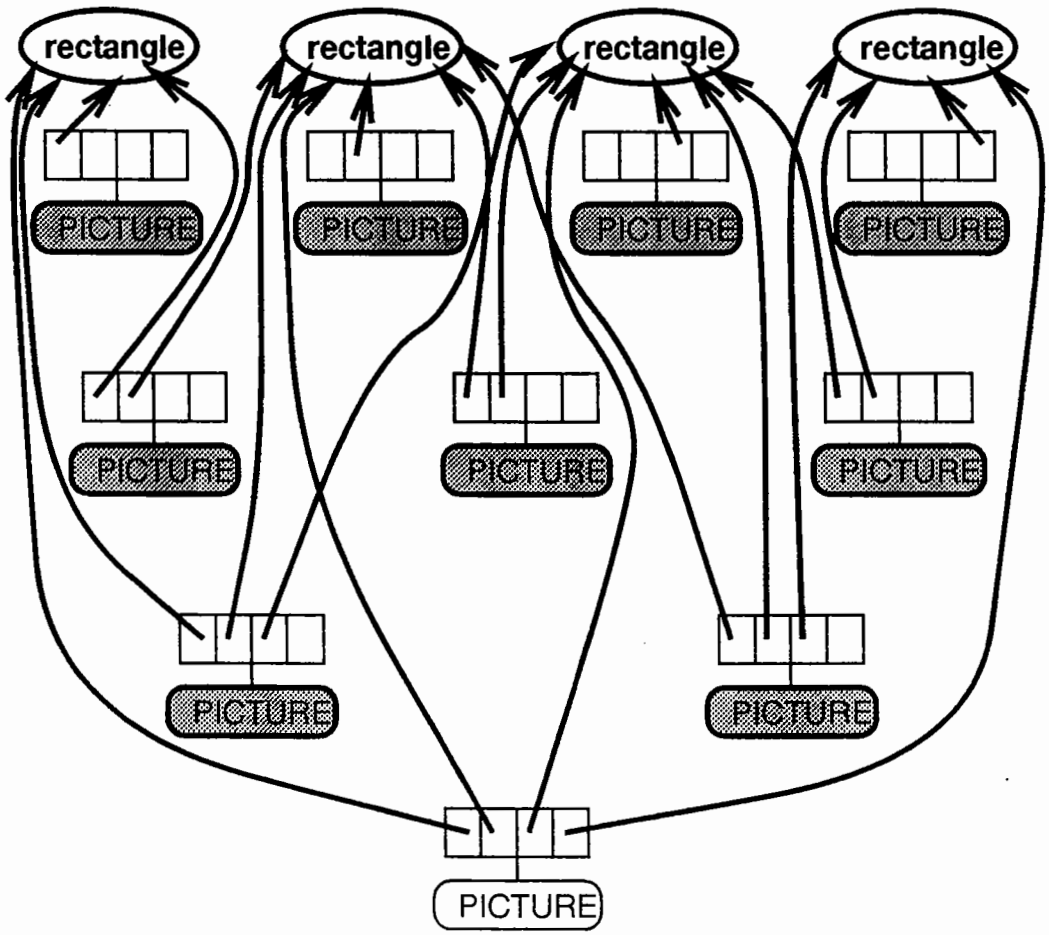
3. a call to a named function.

A complex expression is evaluated by first evaluating all the argument expressions and then performing the specified operation or calling the function named. The result of the operation becomes the value of the expression.

Function calls can be made with either builtin or user defined functions. User defined functions are declared in the grammar file with the `%function` and `%external`



a) The *tiling*



b) FMD Structure with Incomplete Nodes

Figure 6.12: A *tiled* Picture and Corresponding FMD Structure

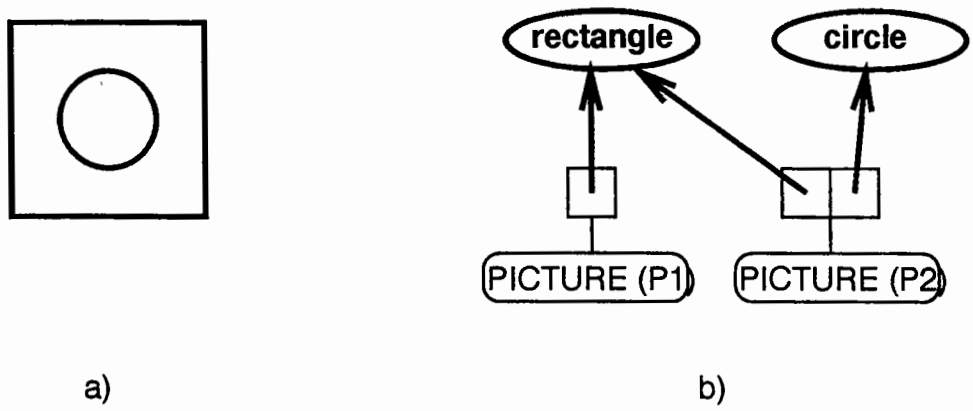


Figure 6.13: a) Simple Visual Program; b) FMD Structure for Program

(1) PICTURE → **rectangle**
(2) PICTURE → contains(**rectangle**, **circle**)

The picture in Figure 6.13a is a valid program in this language. Figure 6.13b shows the FMD structure after the build phase. Two nodes have the start symbol (PICTURE) as their symbol class and represent possible analyses of the picture. However, the node labelled by Production 1 does not represent a valid analysis, since the parse tree under it does not include the **circle** terminal from the input. Thus, when a tree-DAG is extracted from the FMD, one must ensure that it has the entire input as its yield.

Lastly, for the extracted tree-DAG to be well-formed, the tree edges must form a tree. In other words, no node should appear more than once in the tree (again, disregarding the Gedges). Another way of putting this is to say that there can not be two paths through the tree to the same terminal. To see how this could occur, consider the productions:

(1) PICTURE → left_of(LEFT,RIGHT)[1]
(2) LEFT → touches_L(**arrow**, **rectangle**)
(3) RIGHT → touches_R(**arrow**, **rectangle**)

This grammar could be used to analyze the picture shown in Figure 6.14a, resulting in the FMD structure shown in Figure 6.14b. In this case both the productions for LEFT and RIGHT are using the **arrow** terminal. This "sharing" of a terminal violates the hierarchical structure implied by the grammar. In the Attributed Multiset Grammar model, "sharing" is only allowed for symbols in contexts. The extraction procedure must check for this "overlapping" condition and disallow it.

In addition to any valid parses and the errors described above, the FMD structure will generally contain nodes which are unused, in that they cannot be reached from any node labelled by the start symbol. These nodes represent partial analyses. That is, an analysis that was locally correct, in that its constituents satisfied the appropriate

statements, as described in Appendix A. The `%function` declaration specifies the name of the function (as used in attribute expressions), the name of the routine implementing the function and the expected number of parameters. The `%external` declaration specifies an object file containing routines for user defined functions. The routines within the file are dynamically loaded by the parser at runtime. When a user defined function is first invoked, the function name is bound to the address of the dynamically loaded routine.

The last type of attribute expression is a *node reference*. A node reference is valid only as an argument in a function call. A node reference evaluates to a pointer to the factored parse tree node for the indicated element of the right-hand side. Node references are used as arguments to the `test_adjacent` builtin function, which then accesses the attributes of the nodes directly. They could also be used as arguments to a user defined function which requires direct access to the factored parse tree nodes.

6.5 Extracting a Single Parse

After the build phase has completed, the FMD structure contains all the possible analyses of the input program. All that remains is to extract a single analysis from the FMD. Intuitively, this extraction can be done by starting at the root of an analysis and walking the tree (ignoring the graph edges). Since some nodes in the FMD may have more than one child list, a child list must be selected at every node.

As the tree is walked, a tree-DAG is formed from the set of nodes visited and Tedges traversed, together with the Gedges from remote references to the appropriate terminal nodes. This tree-DAG will be an analysis (i.e. parse) for the input multiset as long as four conditions are met:

1. All the constraints are satisfied.
2. The root of the tree-DAG is labelled by the start symbol of the grammar.
3. The input multiset is the yield of the tree-DAG.
4. It is a valid tree-DAG.

The constraints are applied as the FMD structure is constructed (with one exception noted below), so every node in the FMD will always have its constraints satisfied. The second condition is actually implicit in the notion that it is possible to start at the root of an analysis. Although after building the FMD, it is not known which nodes are the root of a valid analysis, all of the nodes which could be the root of an analysis (i.e. all the nodes labelled by the start symbol) can be found. Each of these can then be used as the starting point for extracting an analysis.

Condition three states that the parse tree-DAG must include the entire input picture. This may not be the case for all the trees contained with FMD structure. Some trees may represent parses of only part of the input. For example, consider the simple language defined by the productions:

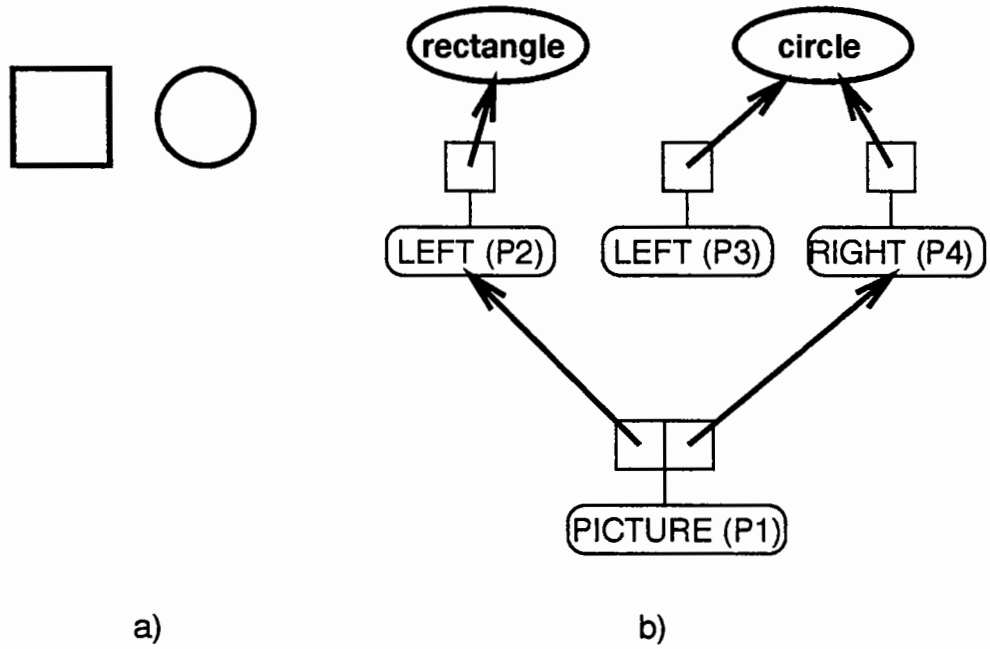


Figure 6.15: a) Visual Program; b) FMD Structure with Unreachable Nodes

LABELS operator, the appropriate child list is selected. Conceptually, this step is actually part of the tree construction, but is most conveniently described here.

The extraction works from the root of the FMD, which is a node labelled by the start symbol. If there is more than one root in the FMD structure, all roots need to be checked to find the correct one(s). Rather than looping over the roots, the tree walking mechanism is used. To do this, a new, "dummy" root node is added to the FMD structure, with a one element child list for each real root node. This corresponds to grouping all the START nodes under a node labelled by the production

$$\text{START} \rightarrow \text{START}$$

The routine *FindRootNode* finds all the root nodes, constructs a dummy root if needed, and returns a single root node.

The routines *ComputeSpanning*, *CheckSpanning* and *CheckOverlapping* enforce the conditions given above. *SelectTree* performs the final tree walk and selection. These routines are described in the following sections. Finally, *ActualRoot* will discard the dummy root node, if one was created, and return the root of the parse.

6.5.1 Extracting a Spanning Parse

The first condition to ensure is that the tree-DAG extracted yields the input multiset. This is done in a two step process. In the first step, the tree is traversed and the set

example, when adding a child list to a LABELS node, the label in the new child list can be compared with the label in the existing child list and only keep that child list which is closer.

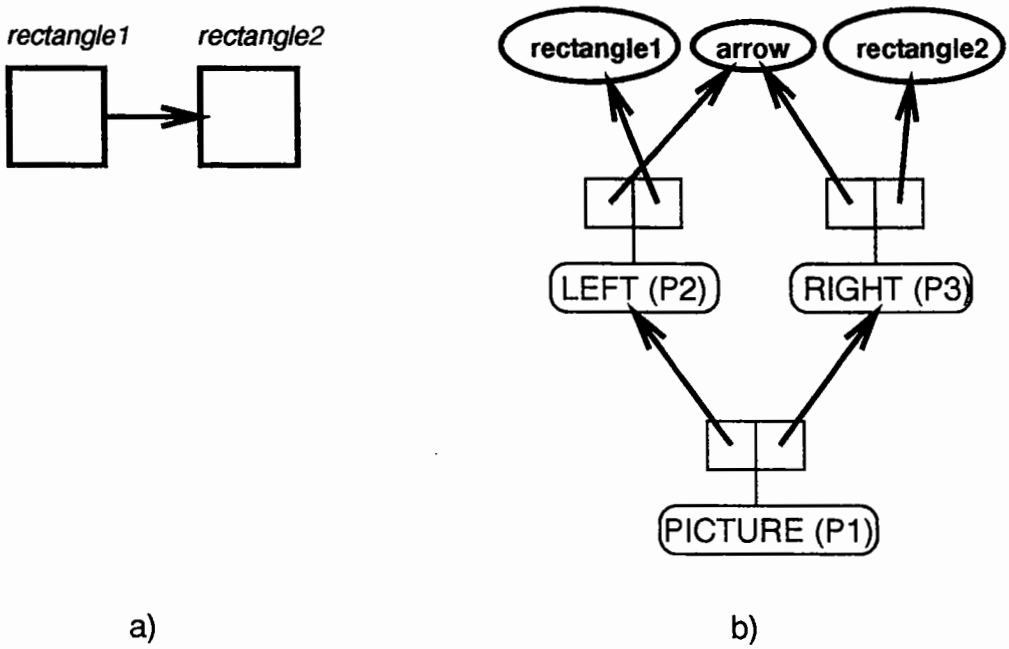


Figure 6.14: a) Overlapping Visual Program; b) FMD Structure for Program

criteria, but was not correct in the larger context of the picture. For example, consider the picture in Figure 6.15a, defined by the grammar

(1) PICTURE → left_of(LEFT,RIGHT)
(2) LEFT → **rectangle**
(3) LEFT → **circle**
(4) RIGHT → **circle**

The FMD structure that is constructed by the build phase is shown in Figure 6.15b. The node labelled by Production 3 cannot be reached from the PICTURE node. When considering only the **circle**, it cannot be determined whether the circle is on the left or right. The build phase considers all possible analyses, so it constructs this node. Since the tree extraction works by walking the tree from the possible root nodes, any unreachable nodes will be ignored.

The basic algorithm for extracting a parse from the FMD structure is given in Figure 6.16. Before extracting a tree, some cleanup from the build phase must be done related to the LABELS operator. Recall that the LABELS operator contains the constraint that the label object must be the closest label to the target. However, since the label can itself be a nonterminal symbol, the closest label cannot be determined until all candidates are found.Thus the extraction of a parse begins by applying this constraint to all the label nodes.[1] For every node labelled by a production using the

[1]Note that this could have been done using some special processing during the build phase. For

```
ComputeSpanning(node)
if !node.computed then
    node.computed = TRUE
    if node is a terminal then
        node.reachable = {node}
    endif
    else do
        node.reachable = ∅
        for cl ∈ node.childlists s.t. cl.deleted = FALSE do
            ComputeSpanningChildlist(cl)
            node.reachable = node.reachable ∪ cl.reachable
        endfor
    endelse
endif
ComputeSpanningChildlist(clist)
clist.reachable = ∅
for node ∈ clist.children s.t. node is a tree child do
    ComputeSpanning(node)
    clist.reachable = clist.reachable ∪ node.reachable
endfor
```

Figure 6.17: Computing Sets of Reachable Terminals

```
ExtractTree()
for each node n using the LABELS operator do
    select the child list with the closest label, and mark all others deleted
endfor
root = FindRootNode()
ComputeSpanning(root)
CheckSpanning(root)
CheckOverlapping(root)
if !root.valid||root.reachable ≠ all terminals then
    signal an error - invalid parse
endif
SelectTree(root)
return ActualRoot(root)
```

Figure 6.16: Basic Pruning Algorithm

of terminals which are *reachable* along tree paths is computed for every node and child list. The algorithm to do this is shown in Figure 6.17.

This algorithm proceeds in a straightforward fashion, traversing the tree edges and computing the *reachable* sets bottom-up. At the leaves of the tree, the *reachable* set consists of just the leaf node. At an interior node, the reachable set for each child list is computed as that child list is traversed, by combining the *reachable* sets of all its tree children. The *reachable* set for the parent node is then formed by combining the sets from all its child lists. The *computed* flag is used to avoid recomputing the *reachable* set, since a node may be in more than one tree.

After computing the *reachable* sets, any non-spanning child lists are detected. A child list is *non-spanning* if it does not generate all the terminals below its parent node. That is, a child list is non-spanning if its *reachable* set does not include all the terminals in its parent's *reachable* set.

The tree is traversed again, checking for non-spanning child lists in a bottom-up fashion. Any non-spanning child lists found are marked as *deleted.* In addition to marking the non-spanning child lists, the nodes and child lists with spanning subtrees below them are determined and marked as *valid.* A node is valid either if it is a leaf node or if it has at least one valid child list. A child list is valid if it is spanning and all its children are valid. The algorithm for this check is given in Figure 6.18. As with computing the reachable sets, a flag (*checked*) is used to avoid checking a node (and its subtree) more than once.

6.5.2 Overlapping Child Lists

Next the parse must be checked for overlapping child lists, which requires a definition of exactly what is meant by "overlapping".

```
CheckOverlapping(node)
if !node.ckover then
    node.ckover = TRUE
    if node is not a terminal then
        for cl ∈ node.childlists s.t. cl.valid do
            CheckOverlappingChildlist(cl)
        endfor
    endif
endif
CheckOverlappingChildlist(clist)
previous = ∅
for node ∈ clist.children s.t. node is a tree child do
    CheckOverlapping(node)
    if node.reachable ∩ previous ≠ ∅ then
        SIGNAL OVERLAPPING ERROR
    endif
    previous = previous ∪ node.reachable
endfor
```

Figure 6.19: Checking for Overlapping Child Lists

Definition 6.4 *A child list C in an FMD structure is* overlapping *if there is a terminal node s ∈ FMD such that there is a tree path (i.e. a path only along tree edges) to s, through two distinct elements of C. A valid node is overlapping if it has a valid child list which is overlapping. A factored parse tree is overlapping if it contains an overlapping valid node. The FMD structure is overlapping if the factored parse tree starting at the root of the FMD is overlapping.*

In other words, the FMD is overlapping if two tree children of a valid child list can both reach the same terminal symbol. Clearly this is only the case when both nodes contain the same terminal node in their *reachable* sets. The non-overlapping condition is enforced by traversing the tree formed by the VALID nodes and child lists and checking each child list to see if it is overlapping. The algorithm for doing this is given in Figure 6.19. Again, use of a flag (*ckover*) avoids checking a node more than once.

The check for overlapping child lists is used to enforce the definition of a PLG as a non-overlapping attributed multiset grammar. The two "overlapping" concepts (of grammars and FMD structures) are related, but not identical. Clearly, if the underlying attributed multiset grammar is overlapping, the picture layout grammar will produce an overlapping FMD structure for some input. The converse is not true, however, since the overlapping FMD structure can reflect an unresolved ambiguity that does not correspond to a valid analysis. However, this condition is rare and does not arise with actual visual languages, so it suffices to use the non-overlapping condition

```
CheckSpanning(node)
if !node.checked then
    node.checked = TRUE
    if node is a terminal then
        node.valid = TRUE
    endif
    else do
        found = FALSE
        for cl ∈ node.childlists s.t. cl.deleted = FALSE do
            CheckSpanningChildlist(cl)
            if cl.valid then
                found = TRUE
            endif
        endfor
        if found then
            node.valid = TRUE
        endif
    endelse
endif
CheckSpanningChildlist(clist)
valid = TRUE
for node ∈ clist.children s.t. node is a tree child do
    CheckSpanning(node)
    if !node.valid then
        valid = FALSE
    endif
endfor
if valid and clist.reachable = clist.parent.reachable then
    clist.valid = TRUE
endif
```

Figure 6.18: Checking for Non-spanning Childlists

shown. Repeatedly composing the same object occurs because the aggregate object (BOX_1 in this case) does not use the attributes of the repeated object (i.e., the **arrow**).

What is required is to ensure that each **arrow** is composed at most once with the aggregate. One solution to this is to impose an arbitrary ordering on the **arrow** objects, and then use the ordering to avoid reusing an object within the same aggregate. The following productions demonstrate how to do this, where $ORDER$(**arrow**) returns a positive integer value such that no two **arrow** objects have the same value (this can be computed by hashing the attributes of the **arrow** object).

BOX → **rectangle**
    BOX.order = 0

BOX_1 → touches_L(BOX_2,**arrow**)
    BOX_1.order = ORDER(**arrow**)
Where
    BOX_2.order ¡ ORDER(**arrow**)

6.5.3 Selecting a Unique Parse Tree

Finally, the parse tree is walked, extracting a single parse from the FMD. Any node which has more than one valid child list reflects some ambiguity in the underlying grammar. Each of the valid child lists are part of some analysis. For example, consider the productions

(1) PICTURE → TWO_RECTANGLES
(2) TWO_RECTANGLES → over(RECT, **rectangle**)
(3) TWO_RECTANGLES → over(**rectangle**, RECT)
(4) RECT → **rectangle**

which define a picture consisting of two rectangles, one over the other, as shown in Figure 6.20a. Clearly two parses of this picture are possible, using either Production 2 or Production 3. In the FMD structure given in Figure 6.20b, the PICTURE node has two valid child lists, one for each possible parse.

To extract a single parse, one of valid child lists must be selected. Since all valid child lists belong to an analysis, an arbitrary choice of child list can be made. The first child list of the node is always selected. The algorithm for extracting the parse is given in Figure 6.21. Note that because only one child list is followed from each node, no flag is needed to prevent reevaluation of a node.

6.6 Parser Complexity

6.6.1 Preliminaries

This section examines the worst-case running time of the parser. The input multiset M to be parsed is assumed to contain n elements. The analysis below will use two derived values based on n:

based on the FMD structure as equivalent to the non-overlapping grammar condition. Section 6.7 uses the non-overlappingness of the FMD structure for all inputs in proving the correctness of the parsing algorithm.

When an overlapping child list is detected, the parser indicates the error and aborts the processing. The language designer may then correct his grammar and try again. A better action might be to try and correct the error by disambiguating the FMD below the overlapping child list. This could be done either automatically or with user assistance. Automatic error correction requires analyzing the paths from the overlapping child list to the terminal in question to determine an appropriate child list to remove. The current version of the parser does not address the problem of error correction.

A better approach entirely would be to check the grammar for the overlapping condition statically, rather than at parse time. Checking for this condition is equivalent to asking whether any multiset in the language contains two identical elements. It does not appear possible to verify this condition, short of generating all elements of the language. The problem is threefold: first, the analysis requires a complete knowledge of how each production manipulates the attributes (which is complicated because the attributes may be extended arbitrarily); secondly, it must be shown not only that identical terminals may be used within two productions, but also that this occurs within a valid analysis; lastly, the overlapping condition may only occur in inputs which are larger than some arbitrary size.

It is important, then, to consider where the overlapping condition can occur and what can be done about it. Overlapping arises from a grammar which is not sufficiently constrained to ensure that all terminals are distinct. Production operators such as *over*, *left_of* and *contains* are area-preserving, in that the area of the aggregate formed contains the areas of the constituents. They are also non-intersecting, in that the constituents are distinct areas (with *contains* the outer region has a hole where the inner region is). When only these types of productions are used, the area of the picture is divided into a hierarchy of non-intersecting regions, and overlapping FMD structures cannot occur.

Overlapping can occur with productions which are either non-area preserving or compose intersecting objects. This is typically the case with productions using the operators based on connectivity rather than spatial relationships. There are two ways for overlapping to occur: either an object is composed with the same object more than once; or an object is composed with more than one object. Figure 6.14 is an example of the latter of these. The former condition is exhibited by the production,

$$\mathrm{BOX}_1 \rightarrow \text{touches\_L}(\mathrm{BOX}_2, \textbf{arrow})$$

which allows an **arrow** to be composed with the same BOX repeatedly.[2]

The problem in Figure 6.14 is a genuine flaw in the grammar. Fixing the grammar depends on the intended definition of the language, but changing either the LEFT or RIGHT production to use a remote reference to the **arrow** will specify the picture

[2]Some variation on this example is the cause of all overlapping constructions I have encountered in the development of actual grammars.

- K_n is the number of complete factored parse tree nodes that are added to the FMD structure during the build phase.
- K_C is the number of child lists added to the complete factored parse tree nodes in the FMD structure during the build phase.

Theorem 4.4.1 shows that attributed multiset grammars cannot, in general, be recognized efficiently. Thus, it is important to characterize those picture layout grammars which can be efficiently parsed. For example, a production of the form

$$A_0 \rightarrow A_1$$
$$A_0.a = A_1.a + 1$$

could cause the build phase of the parser to loop forever (or at least until the value of attribute a overflowed the representation of its data type). For the purpose of analyzing the run-time performance of the parser, we restrict our attention to picture layout grammars which are *polynomially bounded* (see defn. 4.16). These languages will be tractably parsable, in the following sense.

Lemma 6.1 *Let G be a polynomially bounded picture layout grammar. Then $K_n = O(n^k)$ for some constant k.*

Proof: The number of nodes is limited by the number of different node labellings possible. Each node is labelled by a production and an attribute vector. Since the grammar is polynomially bounded, for each attribute $a \in I$, the number of possible values for the attribute is bounded by $O(n^{k_a})$ for some constant k_a. Clearly, the total number of different node labellings possible is of the order

$$O(\rho \prod_{a \in I} n^{k_a}) = O(\rho n^{\sum_{a \in I} k_a}) = O(n^k),$$

where ρ is the number of productions (fixed for a given grammar G). □

A more exact characterization of K_n requires more specific knowledge of the grammar and how it uses the attributes. The following definition leads to a tighter bound on the parsing of languages defined by picture layout grammars.

Definition 6.5 *A picture layout grammar $G = (N, \Sigma, I, S, P)$ is* pure *if*

- *Each symbol in $N \cup \Sigma$ has only the four attributes lx, by, rx, ty.*
- *Each production $p \in P$ uses only the implicit constraints and semantic function associated with production operators.*
- *No productions use the tiling production operator.*

A *pure* picture layout grammar is simply a picture layout grammar that only makes use of the built-in features of the language (other than the *tiling* operator). Since these features are known, it is possible to reason about how the attributes will be used. This leads to the following corollary.

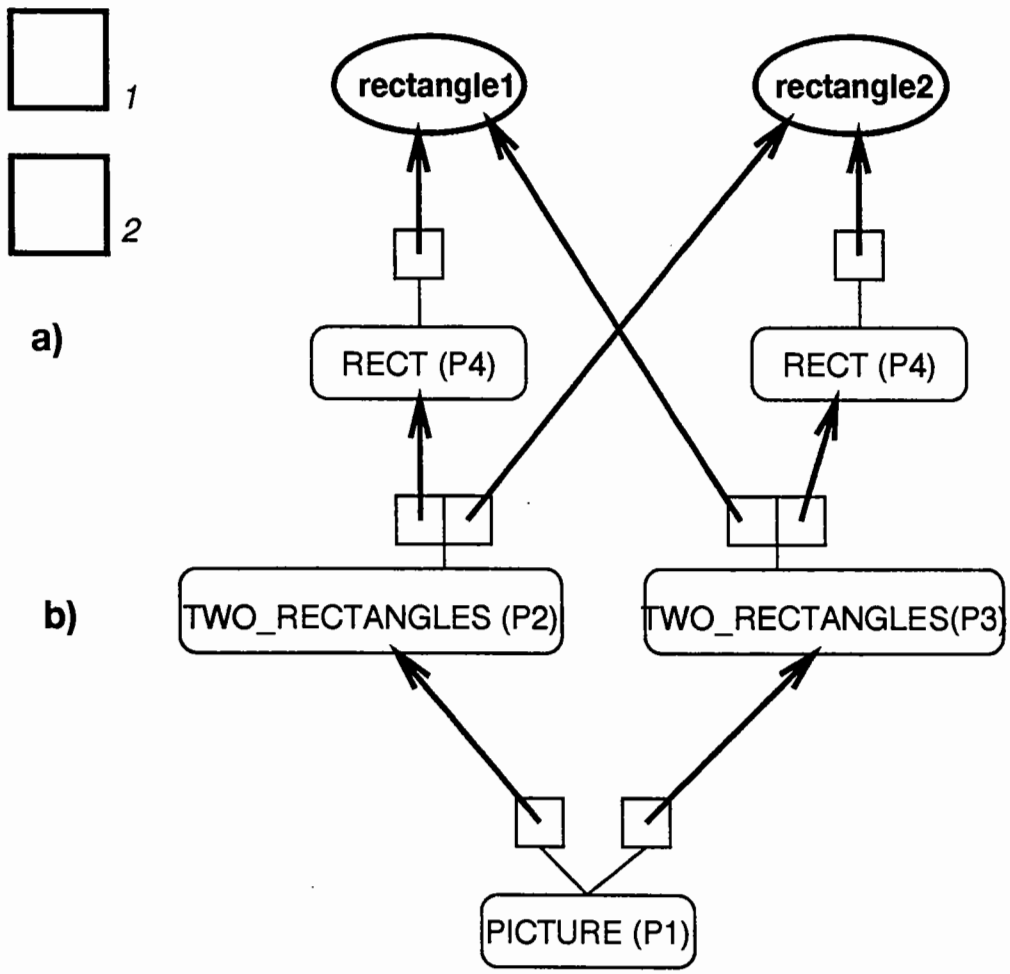


Figure 6.20: a) An Ambiguous Visual Program; 3) The Resulting FMD Structure

SelectTree(node)
if *node* is not a terminal **then**
    let $cl \in node.childlists$ *s.t.* $cl.valid$
    $node.TreeClist = cl$
    for $child \in cl.children$ *s.t.* *node* is a tree child **do**
        $SelectTree(node)$
    endfor
endif

Figure 6.21: Extracting a Single Parse

productions will be executed ρ times, which is constant for the grammar. The innermost two loops each loop over the nodes in the *done* queue, which is clearly at most length K_n. Therefore the innermost statements, evaluating the constraints of the production (and possibly adding a new node) is executed at most ρK_n^2 times. □

Note that the proof of Lemma 6.3 also confirms Lemma 6.2. Each child list is added within one of the two inner loops of the build algorithm, so at most $O(K_n^2)$ child lists could be added.

Lemma 6.4 *The time to extract a spanning tree-DAG from the FMD (i.e. the time in* **ExtractTree()** *is bounded by* $T_{\text{extract}} = O(nK_C)$.

Proof: The algorithm in Figure 6.16 first disambiguates any nodes using the LABELS operator. Since computing the distance for each label takes constant time, this process can take at most $O(\#\text{child lists of LABELS nodes})$ time, which is clearly $O(K_C)$.

The next step is to find the root node (or build a dummy root node). The build phase can construct a list of the nodes labelled by the start symbol, so finding/constructing the root node will take no more than $O(K_n)$ time.

The main part of *ExtractTree* performs three tasks - computing the reachable sets, computing the valid flags (*CheckSpanning*) and checking for overlapping child lists. Each of these is done by walking the tree edges within the FMD structure, processing each node and child list encountered. A node can be visited more than once, but a flag is used to ensure that each node is only processed once. The time for the other visits to a node can be included in the time to process the child list traversed to reach the node. Since each node is processed at most once, each child is visited at most once.

First consider computing the reachable sets, as described in Figure 6.17. The routine *ComputeSpanningChildlist()* is called once for each reachable child list. Since the number of children is fixed for a given grammar, the time to process each child list is the sum of the time to compute the reachable sets for each child, plus the time to combine the reachable sets, plus a constant for the overhead of calling *ComputeSpanning* for each node. Using bit arrays of size n for the reachable sets, they can be combined in time $O(n)$. Thus, the time to process all the child lists (ignoring the time to process the nodes) is at most $O(nK_C)$.

A terminal node takes constant time to process. The time to process an interior node is the sum of the time to process each child list, plus the time to combine the reachable sets of the child lists, plus a small constant. Again, by using bit arrays the reachable sets can be combined in time $O(n)$. Thus, the time to process all the nodes, excluding the time to process the child lists, is at most $O(K_n + nK_C) = O(nK_C)$. Therefore the total time to compute all the reachable sets is bounded by $O(nK_C)$.

Checking for spanning child lists is similar. Each child list is processed once, and the processing (testing for equality of two reachable sets) takes time $O(n)$ plus the time to process the children nodes. Each node is processed once and takes a constant time for each child list, plus the time to process the child list. Thus the total time is bounded by $O(nK_C)$.

The check for overlapping child lists performs the same walk over the tree. Here the processing at each child list is to intersect and union the reachable sets of each of

Corollary 6.1 *For a pure picture layout grammar G, the number of complete nodes added in the build phase is limited by $K_n = O(n^4)$.*

Proof: Since the implicit semantic functions always copy attributes from one of elements of the right-hand side, the value of an attribute at any node must be one of the values of the same attribute for a terminal symbol. There are at most n different values for each attribute. Since each node is labelled by four attribute values, the number of different node labellings is $O(n^4)$. □

The running time for the second phase of the parsing algorithm is more easily expressed in terms of the number of child lists added to the FMD structure. Clearly, the number of child lists added is related to the number of nodes. Each node must have at least one child list, so $K_C \geq K_n$. As an upper bound on the number of child lists is given by the following lemma.

Lemma 6.2 *For a pure picture layout grammar G, the number of child lists added in the build phase is limited by $K_C = O(K_n^2) = O(n^8)$.*

Proof: Each child list is the application of a production to one or two nodes. Thus there are at most ρK_n^2 possible different child lists. Since the application of the same production to the same two nodes will always produce the same attribute values, each child list can only be placed under one factored parse tree node. (Otherwise two nodes would have the same label). □

Lastly, consider what happens if the *tiling* production operator is included. To use the *tiling* operator, the grammar writer must supply explicit semantic functions and constraints. If the semantic functions are restricted so that each attribute of the LHS (e.g. lx) can only be copied from the corresponding attribute of on the RHS elements, then the *tiling* productions will not increase the number of possible node labellings. Thus K_n will not be affected.

A production using the *tiling* operator can have any number of elements in the right-hand side. A production with i elements in the right-hand side could have up to $O(K_n^i)$ child lists. This demonstrates that, in theory, it is more efficient to factor a long tiling production into several shorter productions. In practice, the efficiency depends on the properties of the entities being combined.

6.6.2 Worst-case Analysis

This section analyzes the worst case running time of the parsing algorithm. The analysis is done in two parts: the amount of time spent building the FMD structure; and how much time is spent in extracting a single parse tree-DAG from the FMD.

Lemma 6.3 *The time spent in the build phase is bounded by $T_{\text{build}} = O(K_n^2 T_{\text{eval}})$, where T_{eval} is the time to evaluate the constraints and semantic functions at a node.*

Proof: Looking at the build algorithm given in Figure 6.3, one can see that the main loop is executed once for each factored parse tree node, or K_n times. The loop over

N - the number of input terminals.

K_N - the number of nodes created by the build phase.

K_C - the number of child lists created by build phase.

T_B - the number of times the innermost loop in the build algorithm is executed.

T_E - total number of nodes and child lists visited during one tree walk in the extraction phase.

K_{pt} - the number of nodes in the extracted parse tree.

Three groups of data are presented in the table. The first, called Spiral, is an example of how the worst case for K_N os achieved. The picture corresponding to the case for $N = 8$ is shown in Figure 6.22a and the grammar used to analyze the picture in Figure 6.22c. This grammar describes pictures consisting of a collection of non-overlapping rectangles. The "spiral" input consists of two crossed lines of rectangles.

The second case uses the same grammar, but with an input consisting of rectangles uniformly laid out on grid, as shown in Figure 6.22b. Comparing the Square picture with the Spiral illustrates how the density of a picture can effect parser performance. For a given grid size NxN, the spiral picture will have $2N - 2$ terminals and the square picture N^2 input terminals, but they will both have the same cardinality (i.e. the same number of attribute values).

Finally some performance data for an actual visual language (the StateCharts grammar given in Appendix C.1) are given. The numbers for this grammar show a performance bounded by $O(N^3)$.

6.7 Parser Correctness

This section discusses the correctness of the parsing algorithm. Intuitively, it shows that given a "correct" picture layout grammar G, the parser will parse an input multiset M if and only if M is analyzable over G. In other words, if the parser finds a parse, then that parse is correct; and if there is a correct parse of M, then the parser will find it. First some formal definitions of terms are given, then several lemmas, leading up to Theorem 6.1, which expresses the main result.

6.7.1 Definitions

Exactly what is meant by a "correct" picture layout grammar is defined first:

Definition 6.6 *A picture layout grammar G is* well-formed *if the following two conditions hold for all input multisets $M \in \mathcal{A}(G)$:*

- *(boundedness) If M is finite, then all semantic trees for M over G are finite.*
- *(nonoverlapping) The FMD constructed by the build phase is non-overlapping.*

Grammar	N	K_N	K_C	T_B	T_E	K_{pt}
Spiral	4	124	138	2356	95	12
	8	748	1132	87,920	845	24
	12	2472	4750	969,252	3779	36
Square	1	6	5	7	3	2
	4	54	50	424	35	12
	9	234	249	8199	169	27
	16	696	868	74,272	613	48
	25	1650	2405	423,175	1781	75
	36	3366	5670	1,775,592	4375	108
	49	6174	11,865	6,004,999	9465	147
	64	10,464	22,664	17,309,824	18,569	192
StateCharts	4	43	39	107	24	14
	8	92	84	344	53	29
	12	170	168	864	82	37
	18	297	309	2576	158	60
	41	566	555	7286	260	123
	76	1067	1069	27,617	490	231

Table 6.1: Summary of Actual Performance Data

the children. These operations again take at most $O(n)$ time for each child list. There is no processing at the nodes (other than visiting the child lists). Thus the total time to check for overlapping child lists is again bounded by $O(nK_C)$.

Finally, consider the routine *SelectTree* given in Figure 6.21. This routine visits every node that is in the extracted parse tree and exactly one child list of each node. Thus the total time to select the tree is $O(N_{tree})$, where N_{tree} is the number of nodes in the extracted tree. Clearly $N_{tree} \leq K_C$, so the time at every step in the tree extraction is bounded by $O(nK_C)$. □

6.6.3 Practical Experience

Although the bounds derived above are polynomial, they are still too high to be practical ($50^8 \approx 4 \times 10^{13}$, which means that $N = 50$ might take $O(100$ hours) on a 100 million instructions per second processor!). Two things should be noted, however. First, the results derived above are *upper bounds*, and no grammar encountered so far actually exhibits this peformance.

Second, and more importantly, in grammars for actual visual languages, the performance is much better. This is because most real grammars do not exhibit a high degree of local ambiguity. That is, when considering only a subarea of the picture, there are only a small number of possible analyses for the subarea.

Table 6.1 shows data collected from actual use of the Green environment. The table gives the following data:

The first condition restricts the use of cycles in the grammar (i.e. $A \Longrightarrow A$). Because of the role of attributes and remote nodes in the parsing, it is preferable not to prohibit grammar cycles entirely. This restriction eliminates uncontrolled cycles from the grammar.

The definition of a picture layout grammar given in Chapter 5 states that a PLG is a nonoverlapping attributed multiset grammar. The nonoverlapping condition for a well-formed PLG is a stronger condition, since it requires that the FMD structure be nonoverlapping. Clearly, if underlying attributed multiset grammar is overlapping, the picture layout grammar will produce an overlapping FMD structure for some input. The converse is not true however (the overlapping FMD structure can reflect an unresolved ambiguity).

Because Theorem 6.1 relates the theoretical properties of a grammar to the algorithm for analysis, some definitions are provided for relating semantic trees and factored parse trees.

Definition 6.7 *If S is a factored parse tree, then S is* singular *if* $\forall n \in S$, where $n = [L_n, E_n, C_n], |C_n| = 1$.

Intuitively, a factored parse tree is singular if there is no ambiguity left in the tree. That is, each node in the parse tree has only a single child list (in which case the parse tree is not really "factored" at all). This definition is now extended to a factored multiple derivation structure.

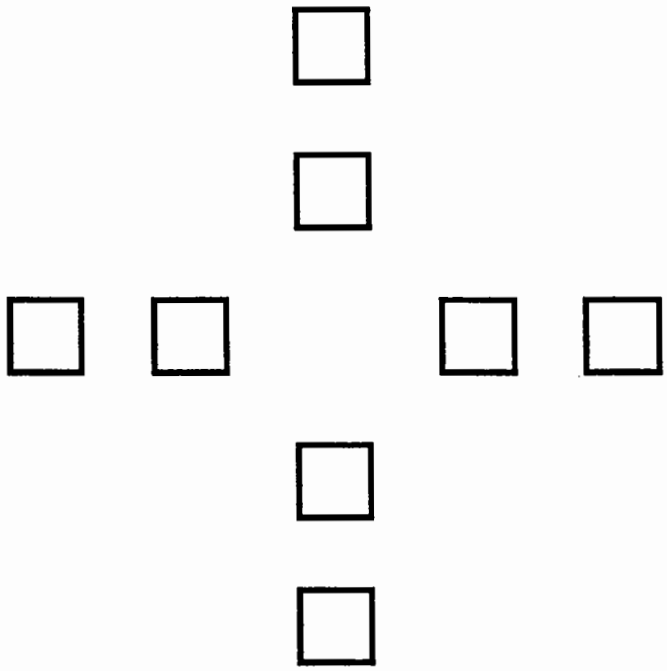
Definition 6.8 *If D is a factored multiple derivation structure, then D is* rooted *if there is an FPTN $N \in D$ such that D forms a factored parse tree with N as its root. D is said to be* singular *if it is rooted and singular.*

An FMD structure is really a forest of factored parse trees. The definition of rooted merely says that this forest contains a single tree with root N. If this tree is singular, then the FMD is singular. Clearly an FMD structure which is rooted may be thought of merely as a factored parse tree. The definition below treats the single element list of child lists as if it was the child list itself.

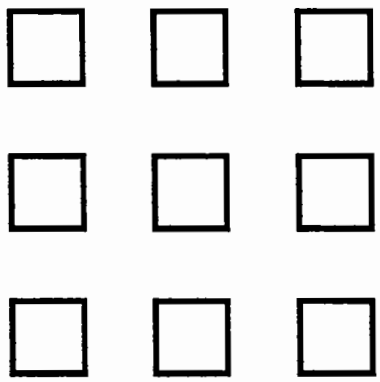
Definition 6.9 (tree congruence) *Let S be a singular factored parse tree and $T = (N_T, E_T, L_T, F_T)$ be a semantic tree. The relation $\cong$ is defined over singular factored parse trees and semantic trees as follows. $S \cong T$ (S is congruent to T), if there is a mapping $\sigma : S \rightarrow N_T$ which is one-to-one and onto, and such that the following holds for all $p \in S$:*

Let $p = [L_p, E_p, C_p]$, and let $n \in N_T$ be the element such that $\sigma(p) = n$. Let (α, V) be the label on node n (i.e. $L_T(n) = (\alpha, V)$); and let $E_n = \{(n, m) \in E_T\}$ be the set of outgoing tree edges at node n; and let $F_n = \{(n, m) \in F_T\}$ be the set of outgoing graph edges at node n.

- *$symb(p) = \alpha$ and $E_p = V$, which says that the symbols and attribute vectors at the nodes match.*



a) The 8-element "Spiral" data.



b) The 3x3 "Squares" data.

(1) PICT → rectangle
(2) PICT → over(PICT$_1$,PICT$_2$)
(3) PICT → left_of(PICT$_1$,PICT$_2$)

c) The Picture Layout Grammar used.

Figure 6.22: Two Performance Data Samples

be constructed which duplicate the nodes $a, n_1, ..., n_j$, except that they point along the new path. Now form a new tree T' by inserting these nodes above a (so that n'_j points to a instead of b and anything that pointed to a in T will point to a' in T'). This new tree is also a semantic tree for M (although it may be overlapping). Since this can be repeated, the tree T' can grow to be infinite. Since G is well-formed (and therefore bounded), we conclude that $n_a \neq n_b$. □

6.7.2 The Build Phase

The first thing to show is that the FMD constructed during the build phase represents all possible valid solutions. This is formulated in the following lemma. The intuition behind this lemma and proof is that it models the bottom-up nature of the algorithm. In other words, the algorithm works by first building all the smallest subtrees, then combining these to form larger subtrees, and so on until all possible parses have been considered.

Lemma 6.6 *Let G be a well-formed picture layout grammar; M be an attributed multiset such that $M \in \mathcal{A}(G)$; and T_s be a semantic tree which is an analysis of M over G. If T is a subtree of T, then after Algorithm 6.11 completes,* FMD $\sqsubseteq T$.

Proof: This is proven by induction on the height of T ($h(T)$). The height is the length of the longest path from the root of T to a leaf node.

To start with, assume $h(T) = 0$. Thus T consists of a single node, labelled by a terminal symbol α and attributes V_α, such that $(\alpha, V_\alpha) \in M$. Clearly the build algorithm will add a node to FMD labelled by this symbol and attributes, since it adds such a node for each terminal in M. Furthermore, this node forms a parse tree which is congruent to T.

Now assume that the lemma is true for all subtrees of height up to k and let $h(T) = k + 1$. Let $T = (N_T, E_T, L_T, F_T)$ and let r be the root of T. Since r is not a leaf node, we know that r is labelled by a production, given by $L_T(r) = p_r$. Now let $O_r = \{m \in N_T \mid (r, m) \in E_T \vee (r, m) \in F_T\}$ be the set of all offspring nodes of r. Clearly, $h(o) \leq k \; \forall o \in O_r$, so if T_o is the subtree rooted at o, we know that FMD $\sqsubseteq T_o \forall o \in O_r$. That is, each of the subtrees of r is contained in the FMD structure, meaning that there is a subset of the FMD nodes forming a factored parse tree which is congruent to the subtree. So the inductive hypothesis implies that there will be nodes in the FMD corresponding to all the children of r. Now it merely needs to be shown that p_r will be applied to these nodes to create a node for r.

For each subtree T_o, let f_o be the root of the factored parse tree congruent to T_o (that is, f_o is the FPTN corresponding to node o). In general, there may be more than one node in the FMD which is the root of a factored parse tree congruent to T_o, any of which may be selected. Clearly, when Algorithm 6.11 processes node f_o, it will add a node for production p_r with f_o in the appropriate position. If the RHS of p_r has only one element, then this node is the root of a parse tree which is congruent to T.

If the RHS of p_r has k elements (and thus r has k offspring) then k incomplete nodes $i_1, ..., i_k$ will be added for the nodes f_o. It must be shown that Algorithm 6.11

Now let $C_p = [q_1, q_2, ..., q_r]$ *be the child list of* p*, and let* $[q_{t_1}, q_{t_2}, ..., q_{t_k}]$ *be the subsequence of tree edges in* C_p *and* $[q_{g_1}, q_{g_2}, ..., q_{g_l}]$ *be the subsequence of graph edges in* C_p*.*

- $\sigma(q_{t_i}) \in E_n$ *and* $\sigma(q_{g_i}) \in F_n$*.*

This definition merely says that a factored parse tree is congruent to a semantic tree if they have the same structure and the same labelings (symbols and attributes) at the nodes. Next is the important definition, which relates semantic trees to general (i.e. non-singular) FMD structures.

Definition 6.10 *Let* $G = (N, \Sigma, s, I, \mathcal{D}, P)$ *be a picture layout grammar;* S *be a factored multiple derivation structure over* G*; and* T *be a semantic tree over* $(\Sigma, I, \mathcal{D})$*. Then we say that* S contains T *(*$S \sqsubseteq T$*) if there exists a subset* A *of* S *such that*

- A *is a factored parse tree.*
- $\forall n \in A$*, with* $n = [L_n, E_n, C_n]$ *and* $L_n \notin \Sigma$*, one can choose one child list* $C_{n_i} \in C_n$ *and form a node* $n' = [L_n, E_n, \{C_{n_i}\}]$*.*
- *Let* $A' = \cup_n \{n'\}$*, then* $A' \cong T$*.*

The factored parse tree A' *is called the* embedding *of* T *in FMD.*

This last definition tells us that an FMD structure contains a semantic tree if there is a factored parse tree embedded in the FMD structure which is congruent to the semantic tree.

Finally, the following lemma is needed:

Lemma 6.5 *Let* G *be a well-formed picture layout grammar;* M *be an attributed multiset such that* $M \in \mathcal{A}(G)$*; and* $T = (N_T, E_T, L_T, F_T)$ *be a semantic tree which is an analysis of* M *over* G*. Let* a *and* b *be distinct nodes in* T *(i.e.* $a, b \in N_T$ *and* $a \neq b$*), with* T_a *and* T_b *the sub-DAG's rooted at* a *and* b*, respectively. If FMD is a factored multiple derivation structure, and* FMD $\sqsubseteq T_a, T_b$*, then* $\exists$nodes$n_a, n_b \in$ FMD$, n_a \neq n_b$*, such that* n_a *and* n_b *are the roots of factored parse trees congruent to* a *and* b*.*

Proof: Since FMD contains both T_a and T_b, it is known that nodes n_a and n_b exist. All that remains is to show that they are distinct.

Consider first the case where neither a nor b is an ancestor of the other. Suppose that $n_a = n_b$. Then a tree T' can be created by replacing the offspring of a with the offspring of b. This new tree T' would also be a valid semantic tree for M, but it would be overlapping, and G would not be a well-formed picture layout grammar. Therefore $n_a \neq n_b$.

Now consider the case where one of a, b is an ancestor of the other. It is assumed w.l.g. that a is the ancestor and b is the decendant. Again, assume that $n_a = n_b$. Now consider the nodes $n_1, n_2, ..., n_j$ on the path from a to b. New nodes $a', n'_1, ..., n'_j$ can

will combine these nodes into a node labelled by p_r. Since T is a subtree of a semantic tree for M, the constraints in p_r are satisfied by the nodes f_o. What remains to be shown is that the the nodes i can be processed in any order.

To combine node i_j, first form the combined node for all the nodes $i_1, ..., i_{j-1}$, which is called $c_{1,j-1}$. Suppose when processing i_j, node $c_{1,j-1}$ has already been processed. Clearly, they will be combined to form $c_{1,j}$. On the other hand, suppose that i_j is processed first. Then when $c_{1,j-1}$ is processed, node i_j will be found and combined to form $c_{1,j}$.

Thus, the algorithm will construct a new node f_r labelled by p_r and the attributes of r, pointing to the nodes $f_o \forall o \in O_r$. Now construct a new subset A_T of FMD consisting of f_r and the union of all the sets used to embed the subtrees T_o into FMD. Since each of the subtrees A_{T_o} was a factored parse tree, A_T is a factored parse tree. We need to show that we can choose one child list for every node $a \in A_T$ such that the resulting singular factored parse tree is congruent to T. There are two parts to this: the congruency mapping σ_T and the choice of child lists.

First look at the mapping. A mapping σ_T from T to A_T is constructed which maps r to f_r, and any node $a \in T_o$ is mapped to $\sigma_{T_o}(a)$. By Lemma 6.5, we know that $\forall a, b$ such that a is in some subtree T_{o_i} and b is in some subtree T_{o_j}, if $a \neq b$ then $f_a \neq f_b$ (where f_a is the FMD node corresponding to a in the mapping $\sigma_{T_{o_i}}$). Clearly, if $a = b$ then $f_a = f_b$. Therefore the mapping σ_T is one-to-one and onto.

Now consider the choice of child lists. For node f_r, there is only one child list. For any node a which is in only one of the subtrees T_o, the child list used to embed that subtree is chosen. Using these selections, the tree A'_T is formed.

Each of the subtrees of f_r is congruent to the corresponding subtree of T, and the node f_r is labelled by the symbol and attributes of r, so the tree A'_T is congruent to T. Therefore FMD $\sqsubseteq T$. By induction, we conclude that the lemma is true. □

6.7.3 Extracting the Parse Tree

Having shown that the build phase will find all possible parses of the input, we would like to show that the extraction phase will retrieve one of them. The extraction phase has three principle parts - computing the reachable sets, setting the valid flags and selecting the tree. The proof of the extraction phase will similarly be broken down into three parts, dealing with the action in each of the subphases.

Lemma 6.7 *Let G be a well-formed picture layout grammar; M be an attributed multiset such that $M \in \mathcal{A}(G)$; and FMD the factored multiple derivation structure constructed by Algorithm 6.11 and possibly supplemented by the addition of a dummy root node. Assume that ComputeSpanning has completed processing on the root. If n is any node in FMD and a is a terminal node in FMD, then $a \in n.reachable$ iff there is a (possibly empty) path from n to a in FMD. Furthermore, if n is an interior node and cl is a child list of n, then $a \in cl.reachable$ iff there is a path from n to a through cl.*

Proof: This is proven by induction on $h(n)$, the height of n (i.e. length of the longest path from n to a terminal node). If $h(n) = 0$, then n is a terminal node and $n.reachable = \{n\}$ by definition. Thus $a \in n.reachable$ implies $a = n$, and there is a zero-length path from n to n.

Now suppose $h(n) = 1$, in which case n is an interior node whose child lists point only at terminal nodes. Clearly a child list cl will have $a \in cl.reachable$ iff $a \in cl.children$ (and thus there is a path through cl from n to a). Furthermore, since $n.reachable$ is the union of $cl.reachable$ for all the child lists of n, it is clear that $a \in n.reachable$ iff one of the childlists cl point to a (forming a path from n to a). Thus the lemma is true for n where $h(n) = 0$ or 1.

Now assume that the lemma is true for all nodes m where $h(m) < k$ and $h(n) = k$. Let cl be a childlist of n. If $a \in cl.reachable$, then there must be some node $m \in cl.children$ such that $a \in m.reachable$. But this node must have $h(m) < k$, so by the inductive hypothesis, there must be a path from m to a and thus a path from n to a through cl. Similarly, if there is a path from n to a through cl, then let m be the first node on the path (i.e. the node pointed to by cl). Again, $h(m) < k$, so $a \in m.reachable$ and therefore $a \in cl.reachable$. Thus $a \in cl.reachable$ iff there is a path from n to a through cl.

Clearly, if $a \in n.reachable$, then $a \in cl.reachable$ for some childlist cl of n, and thus there is a path from n to a. Similarly, if there is a path from n to a, it must go through some childlist cl, which means that $a \in cl.reachable$ and therefore $a \in n.reachable$. Thus, the lemma holds for node n of height k as well. By induction, we can conclude that the lemma is true for all nodes n. □

This lemma has shown that *ComputeSpanning* correctly computes the *reachable* sets for every node and child list. Next, turn to *CheckSpanning*, which sets the *valid* flag for nodes.

Lemma 6.8 *Let G be a well-formed picture layout grammar; M be an attributed multiset such that $M \in \mathcal{A}(G)$; T_s be a semantic tree which is an analysis of M over G; FMD be the factored multiple derivation sturcture and EM_{T_S} be the embedding of T_S in FMD. If n is a node in EM_{T_S}, then CheckSpanning will mark n as valid.*

Proof: This is again proven by induction on the height of n. Clearly, *CheckSpanning* will visit every node in EM_{T_S} as it walks the tree. Suppose $h(n) = 0$, then n is a terminal node and $n.valid$ will be set TRUE.

Now suppose that the lemma is true for all nodes m with $h(m) < k$ and $h(n) = k$. Since n is in EM_{T_S}, there is some child list cl of n that is also in EM_{T_S}. Clearly, every node m which is a tree-child of cl will be in EM_{T_S} and will have $h(m) < k$. Therefore all the tree-children of m will be marked as valid. Thus cl will be marked as valid if it is spanning.

Suppose that $cl.reachable \neq n.reachable$. This means that there is a path from n through another childlist to a terminal a that is not reachable from cl. But since both n and cl are in EM_{T_S}, there must be a path in EM_{T_S} from the root to a that does not go through n. Let LCA be the child list which is the least common ancestor of n and

a. Clearly there are two paths from LCA to a, one through n and one not through n, and therefore FMD is overlapping. Since G is well-formed, it cannot be overlapping and thus we conclude that $cl.reachable = n.reachable$. Therefore n will be marked valid. By induction, we conclude the lemma is true for all n in EM_{T_S}. □

The following two lemmas express the results of the selection phase.

Lemma 6.9 *Let G be a well-formed picture layout grammar; M be an attributed multiset and FMD be the factored multiple derivation structure constructed by Algorithm 6.11 with root node R. If $M \in \mathcal{A}(G)$, then CheckSpanning will mark node R as valid and compute $R.reachable = M$.*

Proof: Since $M \in \mathcal{A}(G)$, there is a semantic tree T_S which is an analysis of M. By Lemma 6.6, there is an embedding EM_{T_S} of T_S in FMD. By Lemma 6.8, we know that all the nodes n in EM_{T_S} will be marked valid, so R will be marked valid. Furthermore, since there is a path in EM_{T_S} from R to every terminal node $a \in M$, we know by Lemma 6.7 that $a \in R.reachable$ for all $a \in M$, so $R.reachable = M$. □

Lemma 6.10 *Let G be a well-formed picture layout grammar; M be an attributed multiset and FMD be the factored multiple derivation structure constructed by Algorithm 6.11 with root node R. Given that CheckSpanning marks R as valid and $R.reachable = M$, let EM be the set of nodes and child lists selected by SelectTree. Then EM is the embedding of a semantic tree T that is a valid analysis of M.*

Proof: A semantic tree T can be constructed from EM as follows: for each node $n \in EM$, create a node n' in T labelled by the same label (production or terminal symbol) as n, along with the attributes of n. For the selected child list of n, tree and graph edges are added to T for every tree and graph child in the child list. Clearly T is a semantic tree with EM as its embedding and M as its yield.

What is left to show is that T is an analysis for M. Let m be a node in T that is labelled by (p, AV), with n the corresponding node in EM. Clearly, since the children of m in T correspond to the children of n in EM, and n was generated by applying production p, the following will be true:

- The children of m will have label symbols equal to the corresponding symbols in the right-hand side of p.
- All the constraints will evaluate to true.
- The attributes of m will be equal to those computed by the semantic functions applied to the attributes of the children of m.

Thus T will be a valid analysis for M. □

These results are summarized in the following theorem. (Here extracting a valid parse refers to the action of $SelectTree$).

Theorem 6.1 *Let G be a well-formed picture layout grammar and M an attributed multiset. The parsing algorithm will extract a valid parse iff M is analyzable over G.*

Proof: Lemmas 6.6, 6.9 and 6.10. □

Chapter 7

A Visual Programming Environment

7.1 Introduction

The previous chapters described a means of defining the syntax of visual languages and an algorithm for spatial parsing. This chapter briefly discusses how picture layout grammars and the spatial parser can be utilized to build a language-independent environment for visual programming. The construction of such an environment is one of the motivations for the development of a grammar/parser based specification mechanism.

One approach to building a visual programming environment is to construct a group of separate components. The traditional approach to programming in textual languages is to use a collection of separate tools such as a text editor, a compiler, a cross-referencer, etc. The tools communicate through an external medium such as ASCII files. This approach can be taken to building a visual programming environment as well, provided that a syntax analysis mechanism for the language is available. Picture layout grammars and the spatial parsing algorithm described in Chapter 6 can serve this function. The *visual programmers workbench* project [34] takes such an approach to building a visual programming environment, and uses a version of the spatial parser.

Another approach which has been used extensively for textual programming is to combine tools such as an editor and a compiler into an integrated programming environment. I have taken this approach to visual programming and have built an integrated *GRaphical Editing ENvironment*, called **GREEN**. The GREEN environment allows the programmer to create and manipulate a visual program, and then recover its structure by parsing and process the program. The language syntax is defined by a Picture Layout Grammar. The attribute mechanism of picture layout grammars is extended to allow the specification of language semantics as well. Attribute grammars have also been used as a basis of integrated programming environments for textual languages [71].

The overall architecture of the environment is shown in Figure 7.1. The environment consists of three major components. The GREEN Editor provides the user interface, overall control and picture editing capabilities. The GDL component reads a grammar

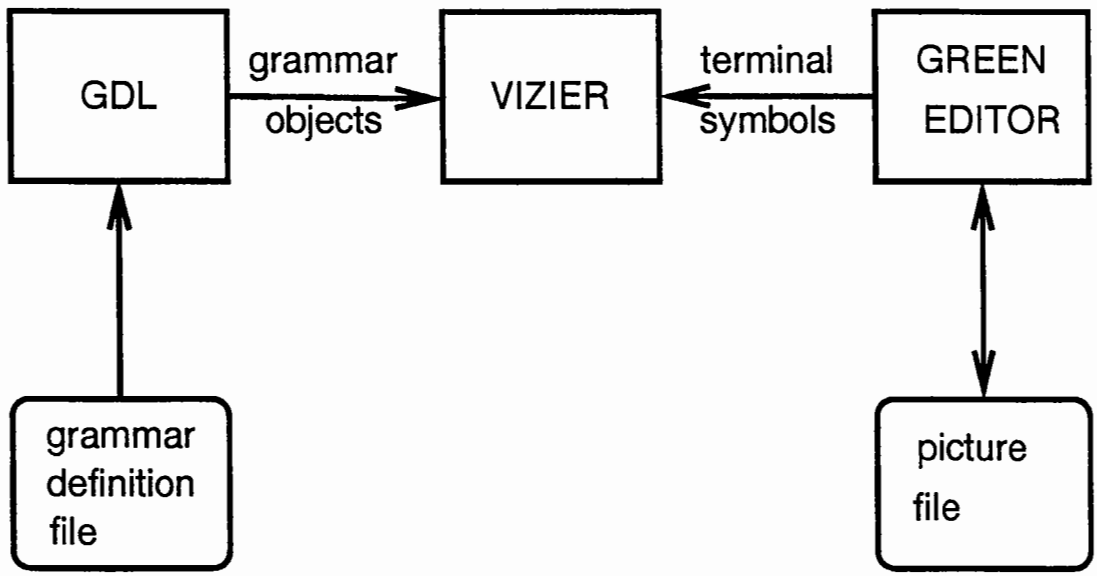


Figure 7.1: Architecture of GRaphical Editing ENvironment

definition file and creates the objects representing the language definition. Vizier is the spatial parser implementation and also provides the semantic processing capabilities. Vizier is called by the GREEN Editor to process the current program.

This chapter discusses the implementation of the GREEN environment. Section 2 gives a brief description of the GREEN Editor. A detailed description of the Editor interface is found in Appendix B. Section 3 describes how the GDL and Vizier components are integrated into the environment. Section 4 discusses the extensions to the attribute mechanism used for semantic processing, and Section 5 gives a simple example of hwo the attribute mechanism is used. Finally, Section 6 discusses how the (semantic) attributes are evaluated.

7.2 The Editor

The GREEN Editor (hereafter called, simply, the Editor) is an editor for visual programs. In other words, the Editor is a tool for creating and manipulating pictures. It is a general-purpose graphics editor[1], and treats the picture as a graphical entity rather than a language entity. All the editing operations know only about pictures, and not about any visual languages.

The basic unit that is edited is the *buffer*. A buffer contains a single picture. Equivalently, we could say that a picture is the contents of a buffer. Each buffer contains

[1]While the GREEN Editor is in the spirit of a general-purpose graphics editor, the current implementation is more of a prototype than a full-functioning graphics editor.

a collection of *objects* which form the picture. Each object in a picture is an instance of a primitive graphical shape. The Editor is object-based, rather than bitmapped based, and is more suitable to producing pictures which are visual programs than pictures which are pieces of art.

A buffer is edited by means of *actions*. An action results either from selecting a menu button or a click of a mouse button. The actions include creating (adding) an object; deleting an object; selecting an object; moving an object; and resizing an object.

The Editor itself does not contain anything new or significantly different from other graphics editors. What is important is the architecture combining a graphics editor with the spatial parser to form a visual programming environment. Two previous visual programming environments took a similar approach: Lakin's spatial parser [48] and the SIL Compiler [17].

7.3 Integrating the Spatial Parser

The GREEN environment contains two components in addition to the Editor: GDL and Vizier. These two components are used for the processing of visual programs and are integral to the GREEN environment. They provide the mechanisms for handling language definitions, spatial parsing and semantic processing.

GREEN maintains the notion of a *current language definition*. The GDL component is called to load a language definition from a grammar definition file. The grammar definition file contains a Picture Layout Grammar defining a visual language. GDL processes the grammar definition file and creates the data structures describing the language syntax, which become the current language definition. An initial grammar definition file can be specified when GREEN is invoked. If none is given, the current language definition is initially null.

Interpretation of visual programs is always with respect to the current language definition. The Vizier component performs the actual processing of visual programs. Vizier implements the spatial parsing algorithm described in Chapter 6. Vizier also provides for the semantic processing of the language, as described in the next section. Vizier is called to process a set of objects from the current buffer, using the current language definition loaded by the GDL component.

7.4 Adding Language Semantics

GREEN extends the attribute mechanism in picture layout grammars to provide for semantic processing of visual languages. Semantic processing refers to the evaluation of computations expressed by programs. This processing can either a direct evaluation of a program (in essence constructing an interpreter for the visual language) or a translation of the program into an executable form (i.e. compiling the visual language). Both of these approaches can be specified using an attribute grammar [71].

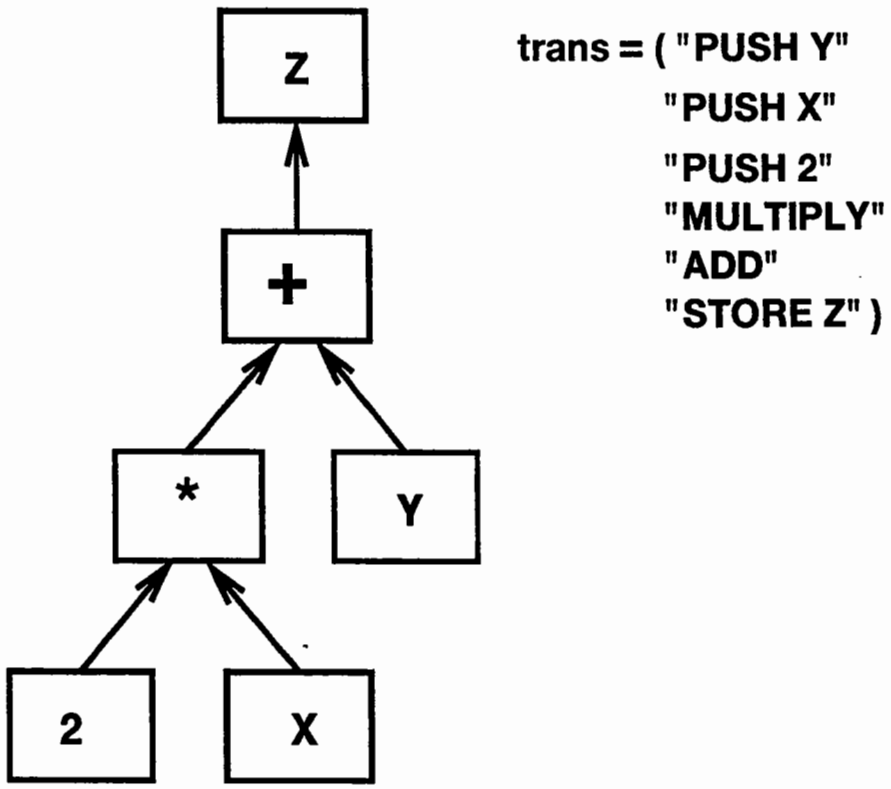


Figure 7.2: Expression Tree Program with Semantics

For example, consider the drawing on the left in Figure 7.2. This is a program in a visual language for describing arithmetic expressions. The terminals of the language are **rectangles**, **arrows** and **text**. The boxes and arrows form a tree. The text labels the leaf boxes with either integer constants or alphabetic variable names; the interior boxes with arithmetic operations; and the root node with a variable name. This is not meant to be a complete language, but could be an "assignment statement" in a larger visual language.

One type of processing for this language would be to compute the value of the expression and assign it to the variable at the root. In the example shown, if $X = 11$ and $Y = 7$, then Z would get the value 29. The other form of processing is to compile the visual program into an object program which can be executed. The visual compiler would generate "machine instructions" to load the values of X and Y from memory locations, compute the expression and store the result in Z.

The approach to language semantics is based on attributes. Attribute grammars have been successfully used to define the semantics of languages for textual programming environments [71]. Attributes serve well for defining static semantics and for

specifying both direct evaluations and translations. Because of the static nature of attributes, they are not well suited to specifying interactive aspects such as environmental or interpretive semantics [41]. The motivation is to demonstrate the utility of constructing a visual programming environment based on spatial parsing, so an attribute mechanism is sufficiently powerful for this purpose.

Semantics is added to a language through an extension to the attribute mechanism used for parsing. In addition to the parsing attributes, each symbol has a set of *processing attributes*. The processing attributes are distinguished from the parsing attributes (by declarations in the grammar), and may not be used either in constraints or in semantic functions which compute parsing attributes. Productions may contain additional semantic functions which compute the values of the processing attributes (and which may use both the parsing and processing attributes to compute the values). During parsing, the processing attributes are not evaluated and the semantic functions computing them are ignored. The processing attributes are evaluated in a separate evaluation phase after the parse has been determined, as explained below.

When a semantic function defines an attribute of the left hand side of a production, that attributed is *synthesized*. An *inherited* attribute is defined in a production where the attribute occurs on the right hand side. An attribute of a symbol either must always be inherited or must always be synthesized. The parsing attributes were restricted to attributes which were totally synthesized (i.e. never having any inherited components). The processing attributes may either be synthesized or inherited, as long as there are no cycles in the computations of the attributes.

7.5 A Simple Example

Now an example showing how the attribute mechanism is used to define the semantics of a language by means of a simple example. The language we are defining is the expression trees described above. The syntax is quite simple: A NODE is a rectangle containing a text string. A CONSTANT node contains an integer. A VARIABLE node contains an identifier. A BINARY operator node contains an arithmetic operation: +, -, * or /, and has two arrows leaving the bottom side of the rectangle, ordered left to right. Each arrow points at another NODE. An ASSIGNMENT is the root node, and gives the variable to be set.

Semantics is provided for this language by translating an expression tree program into a program for a simple stack machine. The translation is specified by computing an attributed called *trans* for the nonterminal EXPRESSION_TREE, that will contain the stack machine program. A picture layout grammar to compute this translation is given in Figure 7.3.

Each nonterminal is given a synthesized processing attribute called *trans*, which will contain a list of strings (each string will be a stack instruction). For CONSTANT and VARIABLE nodes, the translation is the single string ''`PUSH` *x*'', where *x* is either the integer or the identifier. The ASSIGNMENT node generates the string ''`ASSIGN` *x*''.

(1) EXPRESSION_TREE → touches_R(CHILD, ASSIGNMENT)
EXPRESSION_TREE.trans = append(CHILD.trans, ASSIGNMENT.trans)

(2) ASSIGNMENT → contains(**rectangle**, **text**)
ASSIGNMENT.trans = list(concat("ASSIGN ", **text**.string))
Where
is_identifier(**text**.string)

(3) NODE → CONSTANT
NODE.trans = CONSTANT.trans

(4) NODE → VARIABLE
NODE.trans = VARIABLE.trans

(5) NODE → BINARY_OP
NODE.trans = BINARY_OP.trans

(6) CONSTANT → contains(**rectangle**, **text**)
CONSTANT.trans = list(concat("PUSH ", **text**.string))
Where
is_integer(**text**.string)

(7) VARIABLE → contains(**rectangle**, **text**)
VARIABLE.trans = list(concat("PUSH ", **text**.string))
Where
is_identifier(**text**.string)

(8) BINARY_OP → touches_R(CHILD, B1)
B1.trans_right = CHILD.trans
Where
CHILD.ly == B1.by
B1.savedX < CHILD.lx

(9) B1 → touches_R(CHILD, B0)
B1.savedX = CHILD.lx
B1.trans = append(B1.trans_right, append(CHILD.trans, B0.trans))
Where
CHILD.ly == B1.by

(10) B0 → contains(**rectangle**, "+")
B0.trans = list("ADD")

(11) B0 → contains(**rectangle**, "-")
B0.trans = list("SUBTRACT")

(12) B0 → contains(**rectangle**, "*")
B0.trans = list("MULTIPLY")

(13) B0 → contains(**rectangle**, "/")
B0.trans = list("DIVIDE")

(14) CHILD → points_from(**arrow**, NODE)
CHILD.trans = NODE.trans

Figure 7.3: Expression Tree Semantics

The translation for binary operations is somewhat more complicated. A binary operation uses three nonterminals: B0 is the rectangle and operation symbol; B1 is B0 combined with the left-hand operand; BINARY_OP is the complete operation. In addition, each of the operands will have a corresponding CHILD operand. To compute the translation for a binary operation, the B1 nonterminal is given an additional processing attribute *trans_right*. This is an inherited attribute, and is set by Production 7 to contain the translation of the right operand. Production 8 combines the inherited trans_right with the synthesized translations of the left operand and the operator. The translation of the expression tree is show on the right in Figure 7.2.

7.6 Attribute Evaluation Mechanism

The processing attributes are evaluated in a separate phase, after the parsing has completed. When the parsing is done, the selected parse nodes will form a directed acyclic graph. The attribute evaluation phase operates on this structure, computing values for the processing attributes of each node. As with the parsing attributes, the values of the processing attributes are stored with the parse node associated with that symbol.

A simple evaluation strategy, similar to the left to right evaluation method of Bochmann [7], is used by Vizier. The parse structure is traversed in a depth-first manner. At each node, any semantic functions which compute inherited attributes of the children of this node are evaluated first. Then each child is visited in turn to have its attributes evaluated. Finally, any synthesized attributes of this node are computed. Because the parse structure is a DAG rather than a tree, a flag is kept indicating whether a node has been visited. When a node is revisited (in the same traversal), no new evaluation is done.

Because the dependencies between attributes may not be ordered left to right, it may not be possible to compute all attribute values in a single pass. For example, an inherited attribute of an element of the right hand side in a production may use a synthesized attribute of the left hand side in its computation. The synthesized attribute cannot be available until after the attributes of the children of the node have been evaluated.

To overcome this problem, each parse node contains a flag for each of its attribute values indicating whether that value has been computed yet. When a semantic function is evaluated, all attributes referenced are checked to see that they have been computed. If any attribute is not yet defined, evaluation of the semantic function is not done. Otherwise the computed value will be stored in the appropriate node and marked as defined.

During the traversal of the parse structure, it is noted whether any attributes remain to be evaluated. If there are unevaluated attributes, another traversal is made until all attributes have been evaluated. Because there can be no cycles in the dependencies of the attributes (by definition), eventually all the attributes must be evaluated.

To see that this is the case consider the graph with the attribute instances as nodes

and the dependencies as edges (i.e. there is an edge from an instance to all those other instances which use it in their computation). Clearly this graph is a DAG, since there are no cycles in the dependencies. Eliminating the nodes corresponding to previously computed attributes (along with any edges leaving them) leaves a DAG representing the dependencies among the uncomputed attributes. At least one of these attribute instances cannot be dependent on any other instance, and so is computable. Thus each traversal must compute at least one new attribute instance, and eventually all the attributes are computed.

The processing attribute evaluation phase is called automatically after the parsing has completed in response to the user selecting the PARSE BUFFER command. The computed attributes can be retrieved in two ways. First, the GET ATTRIBUTE command can be used within the GREEN environment to examine an attribute of the root node of the parse structure. Another method is to use an external function mechanism to process the attribute value in an appropriate way. For example, the production

```
PROGRAM → EXPRESSION_TREE
        PROGRAM.trans = write_object_file(EXPRESSION_TREE.trans)
```

could be added to the grammar in Figure 7.3, where *write_object_file* is an external function which will write the attribute value to a file as successive lines of text.

Chapter 8

Conclusions

Visual languages are rapidly emerging as a new approach to programming. This dissertation has laid a foundation for a formal theory of visual programming languages. The approach described here builds on techniques successfully used for textual languages (e.g. grammar based syntax specification, parsing, attributes) while extending them to encompass the two dimensional nature of visual languages. The basic methods for understanding and manipulating visual languages I have developed will allow visual languages to grow and evolve on a par with textual languages.

My model for visual languages is natural, general and useful. By natural, I mean that the model corresponds to an intuitive notion of the structure of pictures and visual languages. The model is general in that it can be applied to a wide variety of visual languages, encompassing several different visual styles. The utility of the model is evident by the applications it makes possible, such as spatial parsing.

The practicality of my approach is further demonstrated by the specification mechanism for visual languages that I have developed. Picture layout grammars provide the means of quickly and unambiguously describing the syntax of a visual language. This capability is essential for the design of visual languages. Picture layout grammars provide a valuable framework for understanding and discussing visual syntax. One observation is that an attribute based formalism has proved to be a very flexible approach. This view has recently come to be shared by other researchers in visual language syntax [9, 90].

The benefits of this model are enhanced considerably by the existence of an algorithm for spatial parsing. Syntax analysis a basic function and plays an important role in many textual programming tools. In particular, a general purpose parser, driven by a grammar, is a valuable tool for researchers in visual languages. The power of a grammar driven parser has been demonstrated by parser generators for textual languages such as yacc and bison.

This chapter evaluates my results and then outlines both extensions and directions for further research.

8.1 Evaluations

8.1.1 Expressiveness of the Model

A natural question to ask is how expressive are picture layout grammars? Unfortunately there is no good answer to this question. In asking about the expressiveness of the grammar model, one seeks to understand what languages it can be used for. This question is subject to several interpretations. Are we limited to the predefined attributes and operators of picture layout grammars, or do we consider the use of additional attributes? Are we concerned with what languages may be described, or only with those languages which may also be efficiently analyzed? The answer to these depends on whether one is motivated by theory or practice.

From a theoretical point of view, one would like to characterize the class of languages that can be described by a picture layout grammar. For example, the expressive power of context free (string) grammars is characterized by push-down automata, and numerous results [37] have been obtained regarding context free (string) languages. A similar understanding of picture layout grammars would be desirable. Further, string languages may are divided into four hierarchically related classes. A similar insight into the structure of classes of visual languages will require further study.

A more practical question relates to the actual use of picture layout grammars to specify visual languages. From a practical viewpoint, one is interested in understanding how broadly applicable the grammars are. This means that we are interested only in languages which can be easily and naturally defined by a picture layout grammar which can be used for efficient analysis. While there is no formal characterization of what visual languages picture layout grammars are well-suited for describing, some empirical observations can be made.

- The largest class of visual programming languages exhibited so far in the literature consists of *box and arrow* languages. Examples include finite state automata, data flow diagrams, flowcharts and petri nets. These languages are specialized versions of directed graphs. Picture layout grammars are an effective means of describing these languages. The distinguishing features of each language, such as how to draw the individual nodes and arcs, are easily expressed with picture layout grammars.

- Another style of visual language uses a combination of nesting and relative spatial positions to form programs. Examples of this style are Nassi-Schneiderman diagrams, tiled objects in GARDEN and the Boxer [21] language. Picture layout grammars prove to be a good approach for specifying these languages as well. The ability to specialize the relationship between objects in a production using additional constraints eliminates the need for a large number of specialized production operators.

- One problem with picture layout grammars is that the structure imposed by the grammar may not correspond to the abstract structure envisioned by the language

designer. In other words, while it may be possible to write a grammar to define a class of pictures, it may not be possible for this grammar to structure the picture exactly as desired. For example, in writing a grammar for an FSA language, the parse structure cannot directly follow the graph structure of the FSA, since the FSA may be a cyclic graph.

- Another limitation to picture layout grammars is the inability to express some kinds of "global constraints" on the structure of a visual language. For example, while it is possible to write a grammar defining a language of diagrams of directed graphs, picture layout grammars cannot specify the language of diagrams of directed graphs containing Hamiltonian circuits. Interestingly, an attributed multiset grammar for this language can be written, but it is not a valid picture layout grammar because it is overlapping. The overlapping condition for this grammar reflects the need to resolve which outgoing edge from each vertex is part of the circuit.

- The previous point demonstrates that the limiting factor for describing a language is often the ability to write a non-overlapping grammar. A better theoretical understanding of the non-overlapping restriction is called for. In particular, it would be beneficial to be able to determine whether a grammar is overlapping by examining the grammar. An analysis of this type requires a knowledge of the attribute domains, semantic functions and constraints.

- Another problem arises when trying to describe undirected graph structures. The problem results from using a recursive production containing a remote reference (i.e a production of the form $A \rightarrow foo(A,\underline{b})$), where the attributes of left hand side are copied from the similar symbol on the right hand side. This production introduces no new symbols into the parse structure. However, it does create a new non-tree link between nodes, which may reflect the desired underlying structure. Unfortunately, it is difficult to force the parser to include this link in the extracted parse DAG.

8.1.2 Shortcomings of GREEN

As with any prototype system, GREEN falls short of the ideal. Several shortcomings which arise from the implementation (of the editor or parser) rather than the model are noted here.

- The set of graphical primitives is restricted to boxes, octagons, circles, text, lines and arrows. Adding a new graphical primitive is relatively easy (because an object-oriented approach was taken to implementing the editor), but requires modification of the editor code. This restriction lies solely in the editor, and the parser implementation can take any set of input object classes (i.e. terminal classes).

- The attributes which are generated for each class of graphical object are fixed. If a new attribute (e.g. *fill style* for a rectangle) was desired, the editor would have to modified to compute this attribute and pass it to the parser. Again, this restriction arises from the editor and not the parser implementation.
- The grid alignment mechanism provided by the editor is lacking in two ways. First, because the grids are maintained in physical device coordinates, objects will not line up with the grids from one session to another if the size of the editing window is changed. More importantly, grid alignment is not sufficient for visual programming. A better approach would be to allow the existing objects in the picture to have *gravity* and use that for alignment, as in the Gargoyle system [6].
- The performance of the system is generally too slow to be practical for use as a program development system. The problem here lies not with the algorithm for parsing, but with its implementation, and in particular the bookkeeping that is done within parts of the algorithm. In other words, even though the number of nodes produced in the building phase and the number of operations required to prune those nodes may not be excessive, the slowness of the performing those operations can increase the overall time by a large (constant) factor. I believe that this problem can be addressed by the use of more appropriate data structures (lists are currently used for most structures) within the analysis.

8.2 Future Work

8.2.1 Extensions

A number of research topics are suggested as direct extensions to the work described in this thesis. Grouped together here are a number of improvements that should be incorporated into a new implementation of the parser. Some of them are designed to address the problems noted above.

- Separate policies from capabilities. Capabilities include the geometry processing, the adjacency checking and the special processing for the *group_of* operator. Policies relate to the specifics of the language being defined, such as what production operators are provided. Implementing the parser as policy-free and providing a rich set of capabilities promotes the orthogonality and language independence of the environment. The parser is already implemented in a largely policy free manner, and one goal is to heighten this.
- Allow the grammar writer to define new production operators. When the parser provides a policy-free set of capabilities, the language designer provides the "policies" by specifying the types of compositions to be used. This is currently possible using the attribute mechanism, but providing a macro-like facility for defining new operators would be desirable. The current set of built-in operators could be provided as a library.

- Certain overlapping constructs arise not from a context-sensitivity, but from a local ambiguity that can be resolved. For example, in the StateChart grammar, the arcs leaving a state were ordered arbitrarily. This was required in order to avoid introducing a simple cycle into FMD structure (resulting from a child list of a node pointing to the node itself). This is a common occurrence, and some special means of describing a group of arcs should be provided. The parser could then resolve the appropriate group by analysis.

- Compile the grammars. Preprocessing the grammars would allow the use of efficient structures and access within the parser. It would also improve the efficiency of computing semantic functions and constraints, which are currently interpreted. Compiling the grammars will improve the performance of the parser considerably. It will make it easier to allow the grammar writer to define his own operators without a performance penalty.

- Incorporate improved geometric search techniques. Two areas would benefit from this. First, by processing the input terminals to produce a good two-dimensional representation (e.g. segment trees), the adjacency processing could be improved. Another area that could be improved is in searching for a match to a production.

- Many improvements could be made to the editor, both to improve the functionality provided the user and to better integrate with the parser. Some key improvements include: alignment objects [6], a history facility for undoing[47], programmability, filled objects, rotation of objects and selectable fonts.

8.2.2 Further Research

The work done in this thesis is only a beginning for research into visual languages. It provides a framework for visual languages within which research may continue. I list here some of the significant questions which are suggested. Some of these questions may be viewed as finding new applications for picture layout grammars, while others look for a better understanding of visual languages.

Incremental Parsing Algorithm

The advantage of an incremental algorithm is two-fold. First, it may provide improved performance, at least in the context where the user is making small changes to a program. Another advantage is the ability to use structural information about the program to assist/provide feedback to the user.

Several problems must be solved to develop an incremental version of the parsing algorithm. First, incomplete programs must be handled for the algorithm to be useful. Secondly, the parser must be able to recognize constraints (such as adjacencies) which become false. Lastly, the pruning phase must be reversible. For example, currently a node which is discovered to be unreachable is pruned and this information propagated. A truly incremental algorithm must handle this node becoming reachable again and undoing the implications of its deletion.

A Visual Structure Editor

My approach has been to view pictures as source programs edited in an unstructured way. Another approach to building a visual programming environment is to use picture layout grammars as the basis of a language independent structure editor. A syntax-directed editor for visual programs could be built which is driven by a picture layout grammar. A similar approach has been taken towards textual languages by several researchers [65, 71]. The GARDEN [67] system provides the capability to define structure editors, though not based on a grammar definition

As with the incremental parser, a structure editor must be capable of handling incomplete programs. Another problem for a structure editor is the ability to "unparse" a visual program. That is, it must be possible to generate a picture corresponding to a parse structure. The picture must satisfy all the constraints in the parse structure. In general, the constraints do not completely specify the picture, so an automatic layout must be generated for the remainder of the picture. A typical unconstrained picture is a drawing of some form of graph, where the connectivity is important but not the relative position. Automatically generating such a picture is closely related to the problem of *graph layout* [22, 23, 88].

Picture Generation Systems

Another research topic is how to use picture layout grammars as the basis of a picture generation system. For example, suppose GREEN is used to implement a graphical query language. The result of a query should also be displayed graphically. The question is whether a picture layout grammar can be used to specify this visualization. In a more general context, many visualizations of structures (data, programs, etc) can be viewed as visual languages. One approach to producing these visualizations is through a grammar.

The problems associated with a grammar based approach to visualization may be grouped into two areas. First, it must be possible to map between the abstract structures to be visualized and a parse structure. This mapping may not be part of the grammar specification but should be related to it. In other words, the grammar specifies what the pictures should look like, and one must somehow say which picture belongs to a particular structure. As with the structure editor, it must also be possible to generate a concrete picture from the parse DAG.

Specialized Parsing Algorithms

The spatial parsing algorithm presented here is for general picture layout grammars. An interesting question is to search for restricted classes of picture layout grammars for which more efficient parsing algorithms exist. This is analogous to the situation for string languages, where classes such as LL and LR(1) have specialized parsing algorithms.

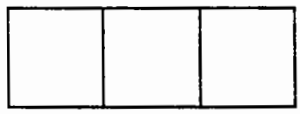


Figure 8.1: Ambiguous Primitive Objects

Resolution of Primitives

My approach assumes that the picture is already represented as a set of primitive object. By using an object-based graphical editor, the problem of segmenting the picture and extracting the primitive objects is avoided. Sometimes this seems to violate the notion that you should be able to understand the picture by looking at it. For example, looking at the picture in Figure 8.1, it cannot be determined whether it was drawn with two or three boxes (or one box with two additional lines).

An interesting question is whether this requirement on the primitive objects can be relaxed to handle such an ambiguous resolution. The parser would then have to determine the correct segmentation.

Formal Properties of Visual Languages

Much room exists for further study of the formal properties of visual languages in general, and of attributed multiset grammars and picture layout grammars in particular. A topic mentioned earlier is to examine the nature of the overlapping condition in grammars. Other topics include relationships between visual languages, closure properties and the relationship of visual languages to string languages. The applications of picture layout grammars justify additional exploration of the theory behind the model of visual languages.

Appendix A

Grammar Definition File Format

The grammar is defined using an ASCII file containing the grammar, similar to the input file for the yacc program. The definition file consists of three sections seperated by a line containing "%%" (with no white space before/after), as shown:

declarations
%%
productions
%%
objects

The file may contain blank lines, white space or comments except where specified (such as on the seperator lines). Two types of comments are available. The string "--" introduces and end of line comment, and everything following it, up to the end of the line, is ignored. Inline comments are surrounded by "/*" and "*/" and may contain any text except "*/" or a newline (i.e. comments can not span lines). Identifiers in the grammar definition may contain letters, numbers and the $, - or _ characters. An identifier must begin with a letter, $ or _ character.

A.1 The Declarations Section

The declarations section consists of a series of declarations statements. Each declaration statement is introduced by a keyword indicating the type of declaration. The declaration keyword must begin a line and be followed by at least one blank.

`%type` *type_name1* { *type_name2* ... }

The `%type` declaration defines the types that can be used for attributes. Three builtin types are provided by the system: `String`, `Integer` and `Sequence`.

`%attribute` ['@' | '^'] *attr_name* ':' *type_name*

The %attribute declaration defines a new attribute. The symbol before the attribute name indicates the class of attribute. By default, attributes are considered to be totally synthesized. If the name is preceded by a '@', the attribute is an ordinary synthesized attribute. A preceding '^' indicates an inherited attribute. The *type_name* specifies the type of the attribute and must be previously defined. No two attributes may have the same name. The system defines four builtin attributes L1, R1, L2 and R2 all of type Integer.

%terminal *name* '{' { *attr_name* ',' ... } '}' ['[' *class_list* ']']

The %terminal declaration defines a new terminal symbol. Any attributes specified in the declaration must have been previously defined by a %attribute declaration. In addition to the attributes specified in the declaration, every terminal symbol will have the four builtin attributes. The optional *class_list* is a comma-seperated list of integers specifying to which terminal classes the symbol belongs. The terminal classes are used by the *group* production and the *test_adjacent* function.

%nonterminal *name* '{' { *attr_name* ',' ... } '}'

The %nonterminal declaration defines a new nonterminal symbol. Any attributes specified in the declaration must have been previously defined by a %attribute declaration. In addition to the attributes specified in the declaration, every nonterminal symbol will have the four builtin attributes. No two symbols (terminal or nonterminal) may have the same name.

%start *nonterminal_name*

The %start declaration specifies which nonterminal is the start symbol for the grammar. Only one start symbol may be specified. The *nonterminal_name* must be defined prior to the start declaration.

%external "file_name"

The %external declaration specifies a file to be dynamically loaded.

%function *function_name* "*routine_name*" *num_parms*

The %function declaration defines a new function. The *function_name* is the name used within the grammar to refer to the function (i.e. in an expression). The *routine_name* is the name of the external symbol for the entry point of the actual function. *num_parms* is an integer constant specifying the number of (32-bit) parameters to the function. If *num_parms* is negative, the function takes a variable number of parameters.

The routine is generally loaded using an %external declaration, which may come before or after any %function declarations (GREEN provides an alternative method for loading external functions). An error will occur during parsing if a function is evaluated which has not been loaded.

operator	meaning
`over(B,C)`	B is above C
`left_of(B,C)`	B is to the left of C
`tiling(B1,B2,...)`	an arbitrary tiling
`contains(B,C)`	B contains C
`adjacent_to(B,C)`	B is next to C
`group_of(B)`	a group of B
`touches_L(B,C)`	the left end of B is on C
`touches_R(B,C)`	the right end of B is on C
`points_to(B,C)`	the right end of B is on C
`points_from(B,C)`	the left end of B is on C
`labels(B,C)`	B is adjacent to the line C
`follow(B,C)`	the line C follows the line B
`join(B,C)`	lines B and C end together
`fork(B,C)`	lines B and C start together
`parallel(B,C)`	lines B and C start/end together
`reverse(B)`	reverse direction of line B

Table A.1: Production Operators

A.2 The Productions

The production section consists of a list of productions. Each production (including the last) must be followed by the symbol "`//`". A production consists of three parts: a rewrite rule, a list of semantic functions and a list of constraints. A production has the form:

lhs '=>' *rhs* [*semantic_functions*] [`where:` *constraints*] '`//`'

The *lhs* of a rewrite rule is a reference to a nonterminal symbol. Within a production, symbol references consist of a symbol name optionally followed by an index, given as *symbol_name* '`#`' *i* where *i* is an integer. This allows us to distinguish between two occurances of the same symbol in a production. If no index is given for a symbol reference, it defaults to zero (0).

Within the *rhs* of a rewrite rule, a terminal symbol reference may optionally be preceded by the '~' character to indicate that the symbol reference is a remote or non-tree reference. If the symbol reference is not preceded by the '~' character, the reference is taken to be a local or tree reference.

The *rhs* of a rewrite rule is either simply a symbol reference or is an application of a production operator:

lhs '=>' *operator* '(' *symbol_list* ')' [*ignore*]

The production operators are given in Table A.1. The *symbol_list* is a comma-seperated list of symbol references. The *ignore* argument is an integer specifing the class of terminal symbols which should be ignored in adjacency testing.

integer
"string"
symbol ['#' *i*] '.' *attribute*
'(' *expression* ')'
'-' *int_expression*
int_expression$_1$ *op* *int_expression*$_2$
function_name '(' *argument_list* ')'

Table A.2: Attribute Expressions

The semantic functions is a possibly empty list of statements of the form:

symbol ['#' *i*] '.' *attribute* '=' *expression* ';'

The *symbol* (with optional index) must refer to a symbol from the rewrite rule. The *attribute* is the name of an attribute of the symbol which is computed in the rule. (NOTE - no checking is done to see that the attribute is computed appropriately). The *expression* is defined below. Each semantic function statment is followed by a ';' seperator character.

The `Where:` keyword introduces the optional constraint list. It is case-insensitive and must have no spaces before the following ':'. The constraint list consists of one or more constraint statements, each followed by a ';' character. Constraints have three forms:

expression$_1$ *relation_op* *expression*$_2$
'(' *constraint* ')'
constraint$_1$ *logical_op* *constraint*$_2$

The first form is a simple constraint where *relation_op* is one of '<', '<=', '==', '>=', '>', or '!='. The other two forms are used to create logical expressions of constraints, using '||' for logical or and '&&' for logical and. If parentheses are not used, the expressions associate left to right.

The expressions allowed are shown in Table A.2. The first two are integer and string constants. The third expression is a reference to an attribute of one of the symbols of production. The next three forms allow the construction of simple integer expressions. The operators allowed are '+', '-', '*' and '/'.

The final form of expression is a function call. The *function_name* is either one of the built-in functions shown in Table A.3, or a function declared with the `%function` declaration. The *argument_list* is a (possibly empty) comma-seperated list of arguments. Each argument is either an expression or a special symbol reference of the form:

'@' *symbol_name* ['#']

This reference refers to a symbol in the production. It is used by certain functions (e.g. `test_adjacent`) which operate on the entire parse tree node, rather than the individual attributes.

function name	meaning
`test_adjacent(x,y,z,w)`	do adjacency testing
`concat(string`$_1$`,string`$_2$`)`	concatenate two strings
`strcmp(string`$_1$`,string`$_2$`)`	compare two strings
`cons(list,elem)`	add elem to front of list
`append(list`$_1$`,list`$_2$`)`	append list$_2$ to end of list$_1$
`list(elem`$_1$`,...)`	make new list of elements
`length(list)`	return length of list
`cdr(list)`	return CDR of list
`car(list)`	return first element of list

Table A.3: Built-in Functions

A.3 Objects

The final section of the input contains an optional list of input objects. The input objects are used for the stand-alone version of the parser. Each input object is of the form

terminal_name '`[`' *value_list* '`]`'

The *value_list* is a comma-seperated list of integer and string constants. The values correspond to the synthesized attributes of the terminal symbol, beginning with the four builtin attributes and then any attributes specified in the `%terminal` declaration.

Appendix B

The GREEN Editor

The GREEN Editor (hereafter called, simply, the Editor) is an editor for visual programs. The basic unit that is edited is the *buffer*. A buffer contains a single picture. A buffer may be stored in a file, just as a text file may be stored. An open buffer is a buffer which is currently in memory and may be viewed or edited. More than one buffer may be open at a time (i.e. several pictures may be edited at once). One of the open buffers is the *current buffer*, which is the buffer currently shown on the screen and editable.

Each buffer contains a collection of *objects* which form the picture. Each object in a picture is an instance of a primitive graphical shape, as described below. A buffer is edited by means of *actions*. An action results either from selecting a menu button or a click of a mouse button. The actions include creating (adding) an object; deleting an object; selecting an object; moving an object; and resizing an object. Actions all effect the current buffer. This appendix describes the GREEN Editor interface.

B.1 The GREEN Screen

The beginning screen of the GREEN Environment is shown in Figure B.1. Across the top are the *command menus*. Each of these is a pull-down menu which is activated by pressing a mouse button while the cursor is over the command button for that menu. A command is selected by moving the cursor over the appropriate button within the menu and releasing the mouse button. The command menus can be *torn off* by selecting the round tack at the top of the menu, and then left on the screen as long as desired.

Under the command menus is the current buffer window. Down the left side of the buffer window is the *shape palette*. This is a menu of the shapes which can be used in drawing. The shapes currently supported are rectangles, circles, octagons, text fragments, lines and arcs. The currently *selected* shape is shown by highlighting. Initially, the rectangle is selected. A new shape may be selected by clicking (pressing and releasing) any of the mouse buttons while the cursor is positioned over one of the shapes in the palette.

The remainder of the buffer window consists of three parts. The *buffer name* is

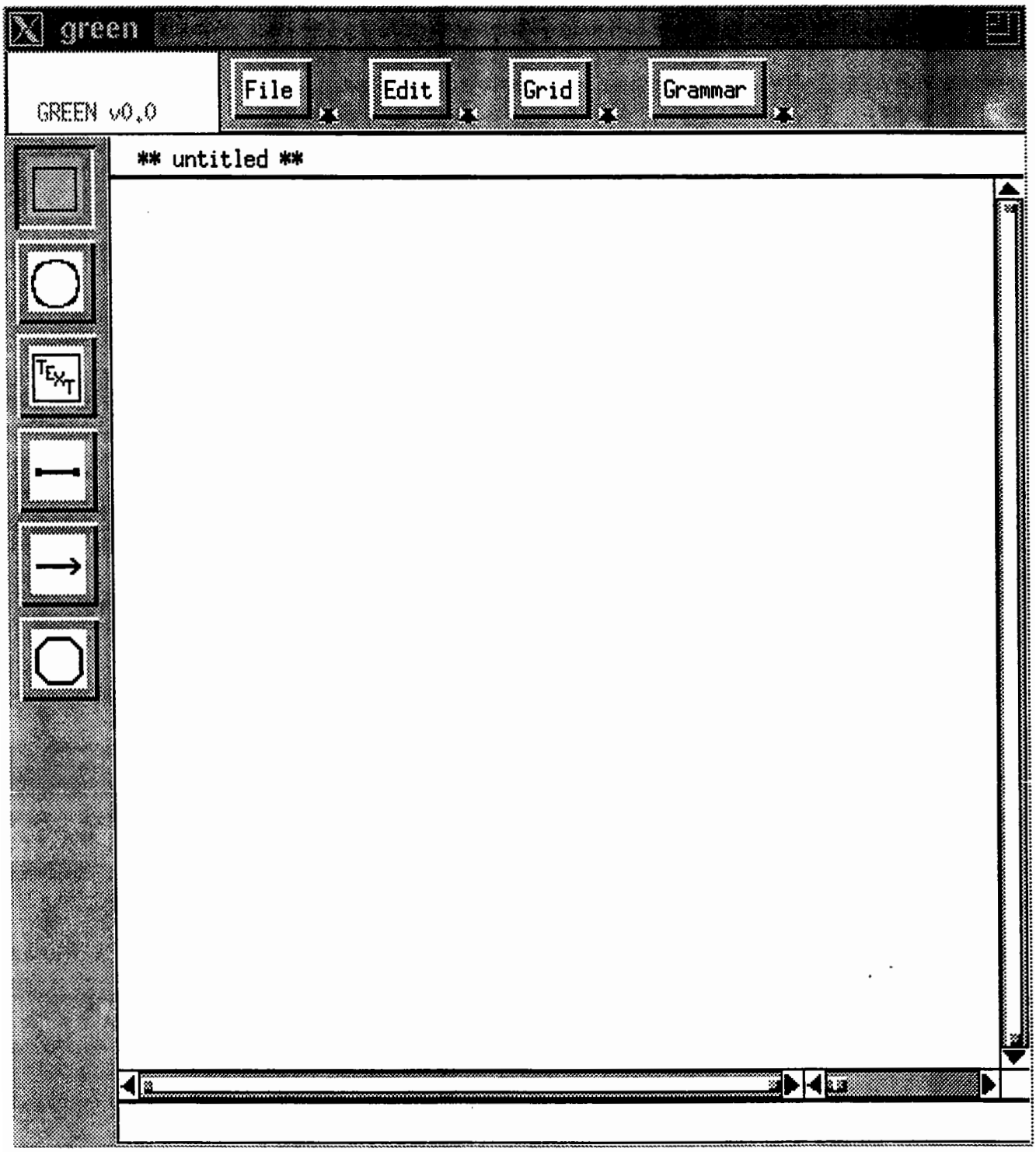


Figure B.1: The Initial GREEN Editor

shown along the top of the buffer window and a *status message* along the bottom. The buffer is initially unnamed, as shown. The large window between them is the *sketchpad*, where the current picture is displayed. The sketchpad has three scrollbars around it. The one on the bottom right is used for zooming in and out on the picture and the other two are used for panning around the picture. They are operatored by clicking a mouse button with the cursor over the scroll bar.

B.2 Drawing - Creating Objects

The sketchpad is empty to start. The user begins her drawing by selecting a shape from the palette. The mouse is moved to locate the cursor within the sketchpad, at the desired location for a corner of the object (or the first point of an arrow or line). She then presses the *leftmost* button on the mouse and holds it down while dragging the mouse to the desired location of the opposite corner (or point) of the shape. Feedback is provided by rubberbanding a bounding box (or a line, as appropriate).

When the mouse button is released, a *create* action is taken. An object of the currently selected shape is added to the picture at the specified location. Figure B.2a shows the editor screen after a rectangle has been added. Another rectangle is added to the picture by again positioning the cursor, pressing the left mouse button, dragging the mouse and releasing the button. Figure B.2b shows the screen after three rectangles have been added.

Now the user wants to draw text within the rectangles. She first selects TEXT by clicking any mouse button over the TEXT icon in the shape palette. The selected icon becomes the current shape. The user now adds a text string in the same manner as the rectangle, by moving the mouse with the left mouse button held down for a create action. The box formed by dragging the mouse specifies the *extent* of the text string.

The user is prompted for the text value of the string, and she enters the string by typing `WAIT`, followed either by RETURN or by clicking the mouse on the ACCEPT button. The text string is added to the picture. The text will be scaled so that the entire string is drawn within the extent of the text object. The extent is not normally displayed, except when the text object as been selected as described below. Clicking the mouse on the CANCEL button cancels the command without adding a string object.

Adding an arrow or a line is done in the same way. The user first selects the arrow (or line) icon from the shape palette. She then moves the cursor within the sketchpad and presses the leftmost mouse button. The point where the mouse button is pressed will be the first endpoint of the arrow (line). The mouse is then dragged to the second endpoint of the arrow (line) and the button released. Feedback is given to the user by drawing a line from the first endpoint to the current cursor position as the mouse is moved. When the button is released, an arrow (line) is drawn from the first to the second endpoint. Figure B.2c shows the picture after several text items, lines and arrows have been added.

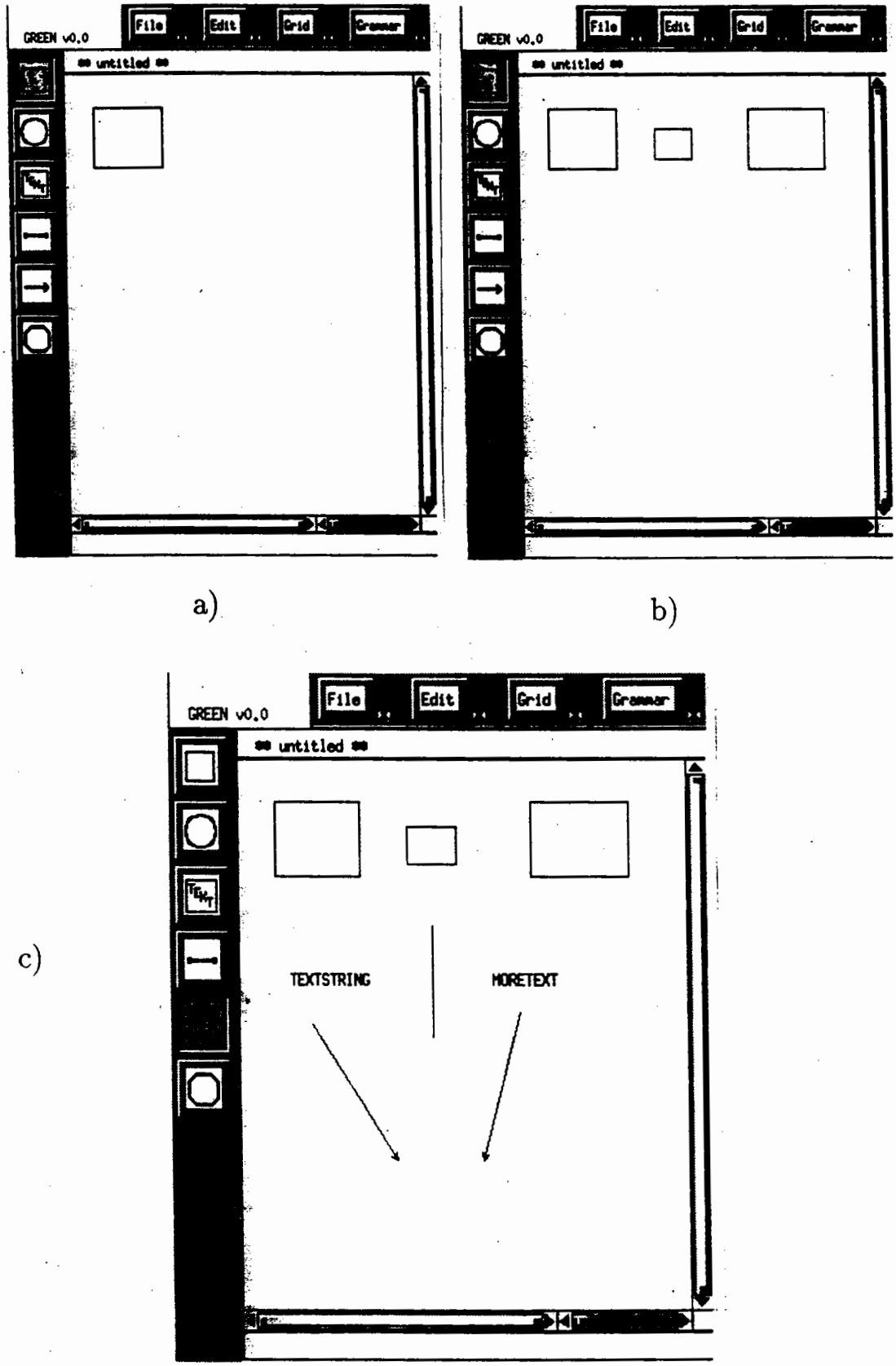


Figure B.2: Adding Objects to the Picture

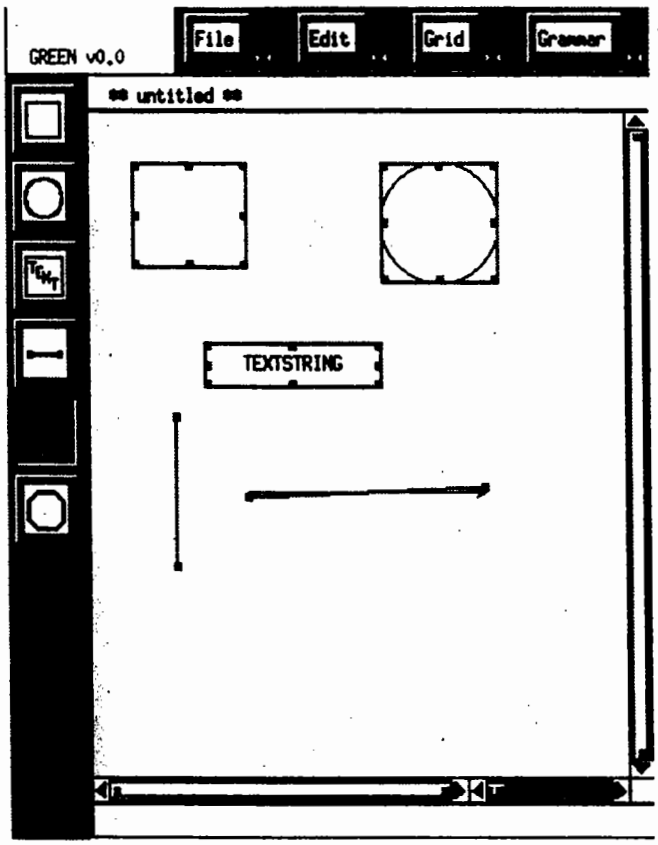


Figure B.3: Highlighted Display of Shapes

B.3 Editing the Picture

The middle mouse button is used for *selecting* objects. The user selects an individual object by pressing and releasing the middle mouse button while the cursor is positioned over the object. A group of objects may be selected by holding down the middle mouse button while moving the mouse. A box will be drawn from the point where the mouse button was pressed to the current cursor location. When the mouse button is released, all objects within the box are selected.

A selected object is highlighted in several ways. First, it is drawn with dashed lines rather than solid ones. For a text object, the extent is drawn in dashed lines. Secondly, on a color display, the object is drawn in a different color. Lastly, *handles* are drawn around the extent of the object. The handles are small solid squares. For a rectangular shape, eight are drawn: four in the corners of the extent and four in the centers of the sides of the extent. For a line or arc, three are drawn: one at each endpoint and one in the center. Figure B.3 shows examples of each type of object when selected.

The rightmost mouse button may be used to either *move* or *resize* a selected object. Moving is accomplished by pressing the right mouse button when the cursor is within the extent of a selected object. Moving the mouse will then drag the selected object to a new position, where the mouse button is released. Resizing is done similarly, except that the button must be pressed when the cursor is over one of the handles.

Selection is also used for the EDIT commands. By clicking on the EDIT button, the EDIT command menu is pulled down. This menu has five commands: CUT, COPY, PASTE, ROTATE and CLEAR SELECTIONS. The first four commands all manipulate a structure called the *kill ring*. The kill ring is a list that has lists of objects as its elements.

The CUT and COPY commands both make copies of all the currently selected objects, put them in a list, and add the list to the front of the kill ring. CUT then *deletes* the objects from the current buffer. PASTE takes the list at the head of the kill ring, makes copies of all the objects in the list and adds them to the current buffer. ROTATE moves the head of the kill ring to the end. CLEAR SELECTIONS deselects all objects currently selected.

B.4 Manipulating Buffers

The BUFFERS command menu contains commands to control buffers. The NEW command will create a new buffer, The new buffer will be unnamed (that is, the name of the buffer will be **unnamed**) and empty. The new buffer will become the current buffer. If no initial buffer file name is given on the command line when GREEN is started up, a NEW buffer is used.

The OPEN command will open an existing buffer. After selecting the OPEN command, the user is prompted for the name of the file containing the buffer. The buffer name will be the same as the name of the file. If the file does not exist, GREEN will complain and then open a new (empty) buffer with the specified name. The newly opened buffer becomes the current buffer. If an initial buffer file name is given when GREEN is started, that buffer will be OPENed.

The RENAME command is used to change the name of the current buffer. This effects only the buffer in memory and not any saved version of the buffer. The SAVE command writes the current buffer into a file with the same name as the buffer. If the buffer is unnamed, the user is prompted for a name before saving. The file that is created contains an ascii representation of the objects in the buffer in an internal format.

The CLOSE command will close the current buffer. The buffer is not saved before closing. If the buffer has been altered since opening, the user will be asked to confirm closing the buffer. If there are other open buffers, then the previous current buffer becomes the current buffer again.

The SELECT command is used to switch between buffers. The user is presented with a menu of all open buffers. Any buffer which has been changed since being opened or created will be marked with the character '>'. The current buffer will be selected by default. The user chooses one of the buffers from the menu and that buffer becomes the current buffer.

B.5 The Alignment Grid

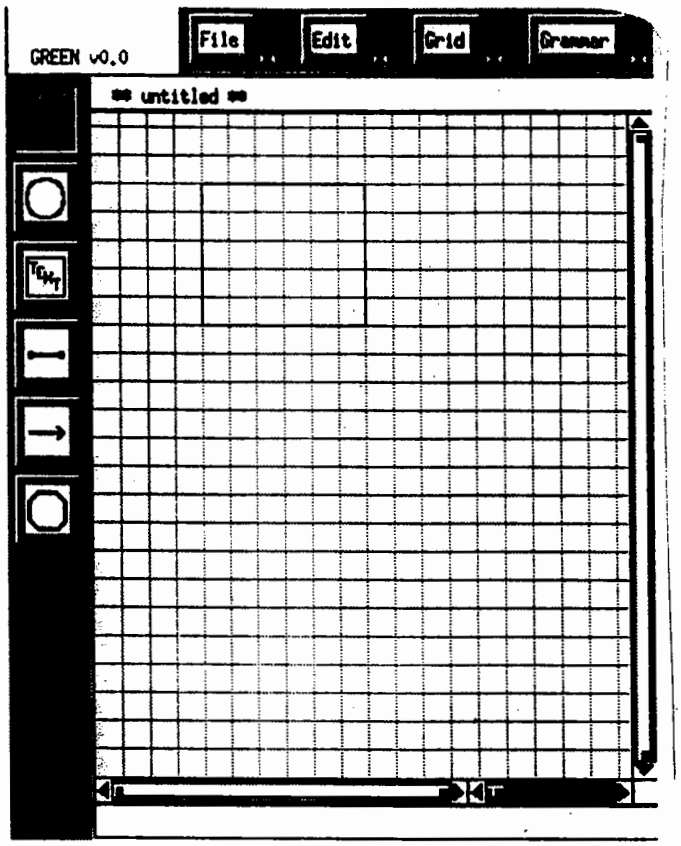


Figure B.4: Using The Alignment Grid

The GREEN Editor provides a simple mechanism for aligning of objects. Object alignment can be very important to the spatial parsing of a picture. For example, the production

$$A \rightarrow \text{points\_to}(\textbf{arrow}, \textbf{rectangle})$$

requires that the right endpoint of the **arrow** must lie exactly on the boundary of the **rectangle**, a difficult feat to perform unassisted with a mouse.

The mechanism used to help with this problem is an *alignment grid*. The grid is composed of evenly spaced horizontal and vertical lines a single pixel wide. When the grid is active, any points are mapped to the closest grid lines. That is, when the left mouse button is pressed to begin creating an object, the initial point is moved from the cursor location during the button press to the nearest intersection of grid lines. When the mouse is moved, the cursor will track the mouse, but the feedback (i.e. the line or box being rubberbanded) will always be aligned to the grid in both directions. After the button is released, the object created will be aligned with the grid. Figure B.4 shows an example of a rectangle created using the alignment grid.

The SHOW GRID/HIDE GRID commands control whether the grid is displayed. When grid display is turned on, the grid is drawn with light dotted lines, as in Figure B.4. The ENABLE GRID/DISABLE GRID commands are used to control whether the grid is active. Points are not aligned to the grid when it is inactive. The SIZE

GRID command is used to change the spacing between the grid lines. The grid display, active and size information is a property of the buffer. The information is saved and loaded with the buffer. When a new buffer is created, the information is copied from the current buffer. The default (when GREEN is started) is for the grid to be active and not displayed, with a spacing of 21 pixels.

B.6 Implementation of the Editor

GREEN is implemented on top of the Brown Workstation Environment [12, 70], which in turn utilizes the X Window System [80]. It currently runs on Sun Microsystems Workstations (3, 4 or SparcStations) and Digital Vaxstation 3100 computers. The environment is written in the C programming language, with extensions from BWE. The code for GREEN (exclusive of GDL, Vizier or BWE) is approximately 5100 lines of C text (including comments) containing about 132,000 bytes. The GDL code is about 5400 lines containing 95,000 bytes. GDL is implemented using the lex [50] and yacc [40] packages for parsing grammar definition files. The Vizier implementation consists of approximately 8200 lines of C containing about 320,000 bytes.

Within the implementation of GREEN, the ASH package is used for creating and managing windows within the Editor. The STEM package provides the facilities for the command menus and shape palette. The OAK package is an experimental system for graphics editors. It forms the substrate for the sketchpad part of the Editor, providing both output and input management. Object management and object-oriented extensions to C are provided by the WORM package. Saving and restoring of buffers from files is done with the AUXD package.

B.7 Calling the Spatial Parser

The GDL and Vizier components are called by commands in the GRAMMAR menu. The READ GRAMMAR FILE command is used to call GDL. After selecting this command, the user is prompted for the name of the grammar definition file to load. The current language definition is discarded and the new language definition loaded from the file. If there is an error in the grammar definition file, the current language definition will be set to null.

The PARSE BUFFER command is the interface to the Vizier spatial parser. When this command is selected, the Editor converts the objects in the current buffer into attributed terminal symbols for the grammar. The position of an object is used to generate the location attributes of the terminal symbol. Vizier is then passed the set of terminals and asked to parse the input. If the parse is unsuccessful, an error message is printed and the terminal symbol causing the error (if any) is highlighted. After a successful parse, the attributed parse structure is retained in memory until either a new language definition is loaded or the parser is called again.

Appendix C

Sample Grammars

C.1 StateCharts

This section contains the complete grammar definition file used to define our subset of the StateChart language.

```
--
-- This file implements a simple version of Harel's STATECHARTS
--

-- DECLARATIONS SECTION
--
-- First we declare the attributes we will use
--
%attribute ival:Integer
%attribute sval:String
%attribute name:String
%attribute default:Integer
%attribute history:Integer
%attribute pos:Integer

--
-- Now we declare the terminal and nonterminal symbols
--

%terminal rectangle{}
%terminal text{sval}[2,3] -- note terminal classes 2 and 3
%terminal line{}
%terminal arrow{}[2,3]
%terminal circle{}

%nonterminal CHART{}
%nonterminal STATE{pos}
%nonterminal INSIDE{}
%nonterminal RIGHT_INSIDE{}
```

```
%nonterminal BOTTOM_INSIDE{}
%nonterminal LABEL{name}
%nonterminal XOR_UNION{default,history}
%nonterminal LABELLED_ARC{pos}
%nonterminal HISTORY{}
%nonterminal HIST{}
%nonterminal DEFAULT{}
%nonterminal DEF_STATE{}
%nonterminal DEF_SYMBOL{}
%nonterminal UNLBLD_ARC_HIST{}
%nonterminal UNLBLD_ARC_STATE{}
%nonterminal LABELLED_ARROW{}
%nonterminal HBAR{}
%nonterminal VBAR{}
%nonterminal LINE{}

--
-- Now we declare our external function entry points and object file.
--

%function disp_int "EXTdisplay_int" 1
%function disp_str "EXTdisplay_string" 1
%function atoi "EXTstr_to_int" 1
%function is_integer "EXTis_integer" 1
%function max "EXTint_maximum" 2
%function min "EXTint_minimum" 2

%external "extlib.o"

%start CHART

--
-- PRODUCTIONS SECTION
--

%%
--
-- STATE PRODUCTIONS
--

CHART => STATE
//

STATE => contains(rectangle,INSIDE)[3]
STATE.pos = 0;
//
STATE => touches_L(LABELLED_ARC,STATE#1)
STATE.pos = LABELLED_ARC.pos;
Where:
```

```
STATE#1.pos < LABELLED_ARC.pos;
//

INSIDE => LABEL
//

INSIDE => over(LABEL,XOR_UNION)[2]
//

LABEL => text
LABEL.name = text.sval;
//



--
-- XOR UNION PRODUCTIONS
--

XOR_UNION => STATE
XOR_UNION.default = 0;
XOR_UNION.history = 0;
//

XOR_UNION => DEFAULT
XOR_UNION.default = 1;
XOR_UNION.history = 0;
//

XOR_UNION => HISTORY
XOR_UNION.default = 0;
XOR_UNION.history = 1;
//

XOR_UNION => adjacent_to(XOR_UNION#1,XOR_UNION#2)[2]
XOR_UNION.default = XOR_UNION#1.default + XOR_UNION#2.default;
XOR_UNION.history = XOR_UNION#1.history + XOR_UNION#2.history;
Where:
XOR_UNION#1.default + XOR_UNION#2.default <= 1;
XOR_UNION#1.history + XOR_UNION#2.history <= 1;
//



--
-- ORTHOGONAL PRODUCT PRODUCTIONS
--
```

```
INSIDE => left_of(INSIDE#1,RIGHT_INSIDE)[2]
//

RIGHT_INSIDE => left_of(VBAR,INSIDE)[2]
//


INSIDE => over(INSIDE#1,BOTTOM_INSIDE)[2]
//

BOTTOM_INSIDE => over(HBAR,INSIDE)[2]
//

HBAR => LINE
Where:
LINE.L2 == LINE.R2;
//

VBAR => LINE
Where:
LINE.L1 == LINE.R1;
//



--
-- TRANSITION PRODUCTIONS
--

LABELLED_ARC => points_to(LABELLED_ARROW,~rectangle)
LABELLED_ARC.pos = LABELLED_ARROW.L1 + 1000*LABELLED_ARROW.L2;
//

LABELLED_ARROW => labels(text,arrow)
//




--
-- DEFAULT and HISTORY PRODUCTIONS
--

DEFAULT => DEF_STATE
//
DEFAULT => touches_L(UNLBLD_ARC_HIST,DEF_STATE)
//
DEF_STATE => touches_L(UNLBLD_ARC_STATE,DEF_SYMBOL)
//
```

```
DEF_SYMBOL => contains(circle,text)
Where:
strcmp(text.sval,"D") == 0;
//

HISTORY => HIST
//
HISTORY => adjacent_to(HIST,text)
Where:
strcmp(text.sval,"*") == 0;
//

HIST => contains(circle,text)
Where:
strcmp(text.sval,"H") == 0;
//


UNLBLD_ARC_HIST => points_to(arrow,~circle)
//

UNLBLD_ARC_STATE => points_to(arrow,~rectangle)
//


--
-- The following four productions are used to normalize lines
-- so that they will have L1 <= R1 and L2 <= R2.
--

LINE => line
Where:
line.L1 <= line.R1;
line.L2 <= line.R2;
//
LINE => line
LINE.L1 = line.R1;
LINE.R1 = line.L1;
Where:
line.L1 > line.R1;
line.L2 <= line.R2;
//
LINE => line
LINE.L2 = line.R2;
LINE.R2 = line.L2;
Where:
line.L1 <= line.R1;
line.L2 > line.R2;
```

```
//
LINE => line
LINE.L1 = line.R1;
LINE.R1 = line.L1;
LINE.L2 = line.R2;
LINE.R2 = line.L2;
Where:
line.L1 > line.R1;
line.L2 > line.R2;
//


--
-- OBJECTS SECTION
--
%%

-- no objects, since this grammar is for GREEN.
```

C.2 The Expression Tree Language

This section contains the complete grammar definition file used to define our expression tree language. This grammar includes semantics that both translates the expression tree program into a stack program and computes its value (actually computing values would require a symbol table to store variable values, though).

```
--
-- This file implements a simple expression tree language.
--

-- DECLARATIONS SECTION
--
-- First we declare the attributes we will use
--
%attribute ival:Integer
%attribute sval:String
%attribute @trans:String
%attribute right_pos:Integer
%attribute @value:Integer
%attribute ^left_value:Integer
%attribute ^right_value:Integer
%attribute @dummy:Integer

--
-- Now we declare the terminal and nonterminal symbols
--

%terminal rectangle{}
```

```
%terminal text{sval}
%terminal arrow{}

%nonterminal EXPR_TREE{trans,value,dummy}
%nonterminal ASSIGNMENT{trans}
%nonterminal VARIABLE{trans,value}
%nonterminal CONSTANT_NODE{ival,trans}
%nonterminal NODE{value,trans}
%nonterminal NODE1{value,right_value,right_pos,trans}
%nonterminal NODE0{value,right_value,left_value,trans}
%nonterminal PLUS_NODE{value,right_value,left_value}
%nonterminal MINUS_NODE{value,right_value,left_value}
%nonterminal TIMES_NODE{value,right_value,left_value}
%nonterminal DIVIDE_NODE{value,right_value,left_value}
%nonterminal CHILD{value,trans}

--
-- Now we declare our external function entry points and object file.
--

%function disp_int "EXTdisplay_int" 1
%function disp_str "EXTdisplay_string" 1
%function atoi "EXTstr_to_int" 1
%function is_integer "EXTis_integer" 1

--
-- write_tree(<fname>, <list>) writes the list of strings to the file.
--
%function write_tree "EXPTRwrite_tree" 2

%external "extlib.o"
%external "exprlib.o"

%start EXPR_TREE

--
-- PRODUCTIONS SECTION
--
%%
EXPR_TREE => touches_R(CHILD,ASSIGNMENT)
EXPR_TREE.value = CHILD.value;
EXPR_TREE.trans = append(CHILD.trans,ASSIGNMENT.trans);
EXPR_TREE.dummy = write_tree(0,EXPR_TREE.trans);
//

ASSIGNMENT => contains(rectangle, text)
ASSIGNMENT.trans = list(concat("ASSIGN ",text.sval));
//
```

```
NODE => CONSTANT_NODE
NODE.value = CONSTANT_NODE.ival;
NODE.trans = CONSTANT_NODE.trans;
//

NODE => VARIABLE
NODE.value = VARIABLE.value;
NODE.trans = VARIABLE.trans;
//

--
-- The following productions implement binary operators
--

NODE => touches_R(CHILD,NODE1) -- CHILD is the left child
NODE.value = NODE1.value;
NODE1.right_value = CHILD.value;
NODE.trans = append(CHILD.trans,NODE1.trans);
Where:
CHILD.R2 == NODE1.L2;
NODE1.right_pos > CHILD.R1;
//

NODE1 => touches_R(CHILD,NODE0) -- CHILD is the right child
NODE1.right_pos = CHILD.R1;
NODE1.value = NODE0.value;
NODE0.right_value = NODE1.right_value;
NODE0.left_value = CHILD.value;
NODE1.trans = append(CHILD.trans,NODE0.trans);
Where:
CHILD.R2 == NODE0.L2;
//

NODE0 => PLUS_NODE
PLUS_NODE.left_value = NODE0.left_value;
PLUS_NODE.right_value = NODE0.right_value;
NODE0.value = PLUS_NODE.value;
NODE0.trans = list("ADD");
//
NODE0 => MINUS_NODE
MINUS_NODE.left_value = NODE0.left_value;
MINUS_NODE.right_value = NODE0.right_value;
NODE0.value = MINUS_NODE.value;
NODE0.trans = list("SUBTRACT");
//
NODE0 => TIMES_NODE
TIMES_NODE.left_value = NODE0.left_value;
TIMES_NODE.right_value = NODE0.right_value;
NODE0.value = TIMES_NODE.value;
```

```
NODE0.trans = list("MULTIPLY");
//
NODE0 => DIVIDE_NODE
DIVIDE_NODE.left_value = NODE0.left_value;
DIVIDE_NODE.right_value = NODE0.right_value;
NODE0.value = DIVIDE_NODE.value;
NODE0.trans = list("DIVIDE ");
//


CONSTANT_NODE => contains(rectangle,text)
CONSTANT_NODE.ival = atoi(text.sval);
CONSTANT_NODE.trans = list(concat("PUSH ",text.sval));
Where:
is_integer(text.sval) == 1;
//

--
-- Note that we require the variable to be a non-integer.
--
VARIABLE => contains(rectangle,text)
VARIABLE.trans = list(concat("PUSH ",text.sval));
Where:
is_integer(text.sval) != 1;
//


PLUS_NODE => contains(rectangle,text)
PLUS_NODE.value = PLUS_NODE.left_value + PLUS_NODE.right_value;
Where:
strcmp(text.sval,"+") == 0;
//

MINUS_NODE => contains(rectangle,text)
MINUS_NODE.value = MINUS_NODE.left_value - MINUS_NODE.right_value;
Where:
strcmp(text.sval,"-") == 0;
//

TIMES_NODE => contains(rectangle,text)
TIMES_NODE.value = TIMES_NODE.left_value * TIMES_NODE.right_value;
Where:
strcmp(text.sval,"*") == 0;
//

DIVIDE_NODE => contains(rectangle,text)
DIVIDE_NODE.value = DIVIDE_NODE.left_value / DIVIDE_NODE.right_value;
Where:
strcmp(text.sval,"/") == 0;
```

```
//

CHILD => points_from(arrow,NODE)
CHILD.value = NODE.value;
CHILD.trans = NODE.trans;
//

--
-- OBJECTS SECTION
--
%%

-- no objects, since this grammar is for GREEN.
```

C.3 Finite State Automata

This section contains the complete grammar definition file used to define our finite state automata language. This grammar translates a graphical representation of a finite state automaton into a string representation.

```
--
-- This file implements a Finite State Automata language
--

-- DECLARATIONS SECTION
--
-- First we declare the attributes we will use
--
%attribute pos:Integer
%attribute sval:String
%attribute @name:String
%attribute @state:String
%attribute @trans:String
%attribute @written:Integer

--
-- Now we declare the terminal and nonterminal symbols
--

%terminal octagon{state}
%terminal text{sval}[1]
%terminal arrow{}[1]

%nonterminal FSA{trans,written}
%nonterminal STATELIST{trans}
%nonterminal STATE_A{pos,trans}
%nonterminal STATE{name,trans}
%nonterminal ARC{pos,name,trans}
%nonterminal ARROW{pos,trans}
```

```
--
-- Now we declare our external function entry points and object file.
--

--
-- write_fsa(<fname>, <fsa-string>) writes the string to the file.
--
%function write_fsa "FSAwrite_fsa" 2

%external "fsalib.o"

%start FSA


--
-- PRODUCTIONS SECTION
--
%%
FSA => STATELIST
FSA.written = write_fsa("fsa.translate",FSA.trans);
FSA.trans = concat("FSA{",concat(STATELIST.trans,"}"));
//

STATELIST => adjacent_to(STATELIST#1,STATELIST#2)[1]
STATELIST.trans = concat(STATELIST#1.trans,
 concat(",",STATELIST#2.trans));
//
STATELIST => STATE_A
STATELIST.trans = concat(STATE_A.trans,"}");
//

STATE_A => touches_L(ARC,STATE_A#1)
STATE_A.pos = ARC.pos;
STATE_A.trans = concat(STATE_A#1.trans,concat(",",ARC.trans));
Where:
STATE_A#1.pos < ARC.pos;
//
$TATE_A => touches_L(ARC,STATE_A#1)
STATE_A.pos = ARC.pos;
STATE_A.trans = concat(STATE_A#1.trans,ARC.trans);
Where:
STATE_A#1.pos == 0;
//

STATE_A => STATE
STATE_A.trans = STATE.trans;
STATE_A.pos = 0;
//
```

```
STATE => contains(octagon,text)
STATE.name = concat("S(",concat(text.sval,")"));
STATE.trans = concat(STATE.name,"{");
octagon.state = STATE.name;
//

ARC => labels(text,ARROW)
ARC.name = concat("Arc(",concat(text.sval,")"));
ARC.trans = concat(ARC.name,ARROW.trans);
ARC.pos = ARROW.pos;
//

ARROW => points_to(arrow,~octagon)
ARROW.trans = concat("->",octagon.state);
ARROW.pos = arrow.L1 + 1000*arrow.L2;
//

%%



--
-- OBJECTS SECTION
--
%%

-- no objects, since this grammar is for GREEN.
```

Bibliography

[1] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation and Compiling*, volume 1. Prentice-Hall, 1972.

[2] Alfred V. Aho. Indexed grammars - an extension of context-free grammars. *Journal of the ACM*, 15(4):647–671, October 1968.

[3] Robert H. Anderson. Syntax-directed recognition of hand-printed two-dimensional mathematics. In Melvin Klerer and Juris Reinfelds, editors, *Interactive Systems for Experimental Applied Mathematics*, pages 436–459. Academic Press, 1968.

[4] Sreedhar Annamalai. Parsing pictures with object transformation. Master's thesis, Brown University, 1988.

[5] Jacques Bertin. *Semiology of Graphics.* The University of WIsconsin Press, Madison, Wisconsin, 1983. Translated by William J. Berg from *Sémiologie graphique* by Jacques Bertin, 1967.

[6] Eric Allan Bier and Maureen C. Stone. Snap-dragging. *Computer Graphics*, 20(4):233–240, August 1986. Proceedings of SIGGRAPH '86.

[7] Gregor V. Bochmann. Semantic evaluation from left to right. *Communications of the ACM*, 19(2):55–62, February 1976.

[8] Tommaso Bolognesi and Diego Latella. A formal definition of a subset of G-LOTOS. Technical Report C89-23, C.N.R/CNUCE, Pisa, Italy, August 1989.

[9] Tommaso Bolognesi and Diego Latella. Techniques for the formal definition of the G-LOTOS syntax. In *1989 IEEE Workshop on Visual Languages*, pages 43–49, Rome, October 1989.

[10] Jeffrey G. Bonar and Blaise W. Liffick. A visual programming language for novices. In Shi-Kuo Chang, editor, *Visual Languages and Visual Programming.* Plenum Publishing Co., 1988.

[11] Alan H. Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, October 1981.

[12] Brown University. *The Brown Workstation Environment Programmer's Manual*, version 5.0 edition, 1988.

[13] Horst Bunke. Attributed programmed graph grammars and their application to schematic diagram interpretation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-4(6):574–582, November 1982.

[14] S. K. Chang and S. Levialdi, editors. *Journal of Visual Languages and Computing.* Academic Press, 1990.

[15] Shi-Kuo Chang. A method for the structural analysis of two-dimensional mathematical expressions. *Information Sciences*, 2:253–272, 1970.

[16] Shi-Kuo Chang. Picture processing grammar and its applications. *Information Sciences*, 3:121–148, 1971.

[17] Shi-Kuo Chang, Michael J. Tauber, Bing Yu, and Jing-Sheng Yu. A visual language compiler. *IEEE Transactions on Software Engineering*, 15(5):506–525, May 1989.

[18] Carlos Christensen. An example of the manipulation of directed graphs in the AMBIT/G programming language. In Melvin Klerer and Juris Reinfelds, editors, *Interactive Systems for Experimental Applied Mathematics*, pages 423–435. Academic Press, 1968.

[19] P. T. Cox, F. R. Giles, and T. Pietrzykowski. Prograph: a step towards liberating programming from textual conditioning. In *1989 IEEE Workshop on Visual Languages*, pages 150–156, Rome, Italy, October 1989.

[20] Larry S. Davis and Thomas C. Henderson. Hierarchical constraint processes for shape analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-3(3):265–277, May 1981.

[21] Andrea A. diSessa and Harold Abelson. Boxer: A reconstructible computational medium. *Communications of the ACM*, 29(9):859–868, September 1986.

[22] Peter Eades and Roberto Tamassia. Algorithms for automatic graph drawing: An annotated bibliography. Technical Report CS-89-09, Brown University, October 1989.

[23] Peter Eades and Lin Xuemin. How to draw a directed graph. In *1989 IEEE Workshop on Visual Languages*, pages 13–17, Rome, Italy, October 1989.

[24] Jay Earley. *An Efficient Context Free Parsing Algorithm.* PhD thesis, Carnegie-Mellon University, 1977.

[25] Jerome Feder. Plex languages. *Information Sciences*, 3(3):225–241, 1971.

[26] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, second edition edition, 1990.

[27] K. S. Fu. Tree languages and syntactic pattern recognition. In C. H. Chen, editor, *Pattern Recognition and Artificial Intelligence*, pages 257–291. Academic Press, 1976.

[28] K. S. Fu. *Syntactic Pattern Recognition and Applications.* Prentice-Hall, Inc., 1982.

[29] King-Sun Fu and Bharat K. Bhargava. Tree systems for syntactic pattern recognition. *IEEE Computer*, C-22(12):1087–1099, December 1973.

[30] Will D. Gillett and T. D. Kimura. The syntax and parsing of two-dimensional languages. Technical Report WUCS-86-14, Washington University, July 1986.

[31] James Gips. A syntax-directed program that performs a three-dimensional perceptual task. *Pattern Recognition*, 6:189–200, December 1974.

[32] James Gips. *Shape Grammars and their Uses.* Birkhäuser Verlag, Basel, 1975.

[33] Ephraim P. Glinert and Steven L. Tanimoto. Pict: an interactive graphical programming environment. *IEEE Computer*, 17(11):7–25, November 1984.

[34] Eric J. Golin and Steven P. Reiss. Parsing in a visual language environment. Technical Report CS-89-06, Brown University, 1987.

[35] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[36] Thomas C. Henderson and Ashok Samal. Shape grammar compilers. *Pattern Recognition*, 19(4):279–288, 1986.

[37] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[38] Adobe Systems Incorporated. *PostScript Language Reference Manual.* Addison-Wesley Publishing Company, 1985.

[39] American National Standards Institute. Programmer's hierarchical interactive graphics system (PHIGS) functional description, archive file format, clear-text encoding of archive file. Technical Report X3.144-1988, ANSI, 1988.

[40] S. C. Johnson. Yacc: Yet another compiler-compiler. Technical report, Bell Laboratories, 1974.

[41] Gail Kaiser. *Semantics for Structure Editing Environments.* PhD thesis, Carnegie-Mellon University, 1985.

[42] B. W. Kernighan. PIC – a graphics language for typesetting. Technical Report 85, Bell Laboratories, Murry Hill, NJ, 1982.

[43] Takayaki Dan Kimura. Determinancy of hierarchical dataflow model. Technical Report WUCS-86-5, Washington University, March 1986.

[44] Takayaki Dan Kimura, Julie W. Choi, and Jane M. Mack. A visual language for keyboardless programming. Technical Report WUCS-86-6, Washington University, March 1986.

[45] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[46] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, 1969.

[47] David Kurlander and Steven Feiner. Editable graphical histories. In *1988 IEEE Workshop on Visual Languages*, pages 127–134, Pittsburgh, October 1988. IEEE.

[48] Fred Lakin. Spatial parsing for visual languages. In Shi-Kuo Chang, Tadao Ichikawa, and Panos A. Ligomenides, editors, *Visual Languages*, pages 35–85. Plenum Press, 1986.

[49] Fred Lakin. Visual grammars for visual languages. In *AAAI87*, pages 683–688, 1987.

[50] M. E. Lesk and E. Schmidt. Lex - a lexical analyzer generator. In *UNIX Programmer's Manual Supplementary Documents*. University of California, 1984.

[51] Frank Ludolph, Yu-Ying Chow, Dan Ingalls, Scott Wallace, and Ken Doyle. The fabrik programming environment. In *1988 IEEE Workshop on Visual Languages*, pages 222–230, Pittsburgh, PA, October 1988.

[52] Jock D. Mackinlay. *Automatic Design of Grahpical Presentations*. PhD thesis, Stanford University, 1986.

[53] Mark W. Maimone, J. D. Tygar, and Jeannette M. Wing. Miro semantics for security. In *1988 IEEE Workshop on Visual Languages*, pages 45–51, Pittsburgh, PA, October 1988.

[54] Gerald Masini and Roger Mohr. MIRABELLE, a system for structural analysis of drawings. *Pattern Recognition*, 16(4):363–372, 1983.

[55] David L. Milgram and Azriel Rosenfeld. Array automata and array grammars. In *Information Processing 71*, pages 69–74. North-Holland Publishing, 1971.

[56] Roger Mohr. Precompilation of syntactical descriptions and knowledge directed analysis of patterns. *Pattern Recognition*, 19(4):255–266, 1986.

[57] Mark Moriconi and Dwight F. Hare. Visualizing program designs through PegaSys. *IEEE Computer*, 18(8):72–85, August 1985.

[58] Brad A. Myers. Displaying data structures for interactive debugging. Technical Report CSL-80-7, Xerox PARC, June 1980.

[59] I. Nassi and B. Shneiderman. Flowchart techniques for structured programming. *ACM SIGPLAN Notices*, 8(8):12–26, 1973.

[60] Donald A. Norman. Some observations on mental models. In Dedre Gentner and Albert L. Stevens, editors, *Mental Models*, chapter 1. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1983.

[61] P. Stanley Peters and Robert W. Ritchie. Context-sensitive immediate constituent analysis: Context-free languages revisited. *Mathematical Systems Theory*, 6(4):324–333, 1973.

[62] John L. Pfaltz. Web grammars and picture description. *Computer Graphics and Image Processing*, 1(2):193–220, 1927.

[63] Man-Chi Pong. A graphical language for concurrent programming. In *1986 IEEE Workshop on Visual Languages*, pages 26–33, Dallas, TX, June 1986.

[64] Georg Raeder. *Programming in Pictures.* PhD thesis, University of Southern California, 1984.

[65] Steven P. Reiss. Pecan: Program development systems that support multiple views. Technical Report CS-83-29, Brown University, 1983.

[66] Steven P. Reiss. A conceptual programming environment. Technical Report CS-86-20, Brown University, August 1986.

[67] Steven P. Reiss. Working in the Garden environment for conceptual programming. *IEEE Software*, 4(6):16–27, November 1987.

[68] Steven P. Reiss, Eric J. Golin, and Robert V. Rubin. Prototyping visual languages with the garden system. In *1986 IEEE Workshop on Visual Languages*, pages 105–110, October 1986.

[69] Steven P. Reiss and Joseph N. Pato. Displaying program and data structures. Technical Report CS-86-19, Brown University, April 1986.

[70] Steven P. Reiss and John T. Stasko. The Brown Workstation Environment: A user interface design toolkit. In *IFIP Working Conference on Engineering for Human Computer Communication*, Napa Valley, CA, August 1989. North Holland.

[71] Thomas W. Reps. *Generating Language-Based Environments.* The MIT Press, 1984.

[72] Gabriele Rohr. Using visual concepts. In Shi-Kuo Chang, Tadao Ichikawa, and Panos Ligomenides, editors, *Visual Languages*, pages 325–348. Plenum Press, 1986.

[73] Azriel Rosenfeld. *Picture Languages: Formal Models for Picture Recognition.* Academic Press, 1979.

[74] Douglas T. Ross. Applications and extensions of SADT. *IEEE Computer*, 18(4):25–25, April 1985.

[75] P. D. Rovner and D. A. Henderson. On the implementation of AMBIT/G: A graphical programming language. In *IJCAI*, pages 9–20. IJCAI, May 1969.

[76] Robert V. Rubin, Eric J. Golin, and Steven P. Reiss. Thinkpad: A graphical system for programming by demonstration. *IEEE Software*, 2(2):73–79, March 1985.

[77] Arto Salomaa. *Formal Languages.* Academic Press, 1973.

[78] A. Sanfeliu and K. S. Fu. Tree-graph grammars for pattern recognition. In Hartmut Ehrig, Manfred Nagl, and Grzegorz Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science*, volume 153 of *Lecture Notes in Computer Science*, pages 349–368. Springer-Verlag, 1983.

[79] David A. Scanlan. Structured flowcharts outperform pseudocode: An experimental comparison. *IEEE Software*, 6(9):28–36, September 1989.

[80] Robert W. Schiefler and Jim Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[81] Ted Selker and Larry Koved. Elements of visual language. In *1988 IEEE Workshop on Visual Languages*, Pittsburgh, PA, 1988.

[82] Linda G. Shapiro and Robert J. Baron. ESP^3: A language for pattern description and a system for pattern recognition. *IEEE Transactions on Software Engineering*, SE-3(2):169–183, March 1977.

[83] Alan C. Shaw. A formal picture description scheme as a basis for picture processing systems. *Information and Control*, 14:9–52, 1969.

[84] Q.Y. Shi and K. S. Fu. Parsing and translation of (attributed) expansive graph languages for scene analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-5(5):472–485, September 1983.

[85] N. C. Shu. Formal: A forms-oriented and visual-directed application system. *IEEE Computer*, 18(8):38–49, August 1985.

[86] Nan C. Shu. *Visual Programming.* Van Nostrand Reinhold Company, 1988.

[87] Savid N. Smith. Visual programming in the interface construction set. In *1988 IEEE Workshop on Visual Languages*, pages 109–120, Pittsburgh, PA, October 1988.

[88] R. Tamassia, G. Di Battista, and C. Batini. Automatic graph drawing and readability of diagrams. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-18(1):61–79, 1988.

[89] James W. Thatcher. Tree automata: An informal survey. In Alfred V. Aho, editor, *Currents in the Theory of Computing*, pages 143–172. Prentice,-Hall, Inc., 1973.

[90] Genoveffa Tortora and Paolo Leoncini. A model for the specification and interpretation of visual languages. In *1988 IEEE Workshop on Visual Languages*, pages 52–60. IEEE, October 1988.

[91] Pierluigi Della Vigna and Carlo Ghezzi. Context-free graph grammars. *Information and Control*, 37:207–233, 1978.

[92] Anthony I. Wasserman and Peter A. Pircher. A graphical extensible integrated environment for software development. *SIGPLAN Notices*, 22(1):131–142, January 1987. Proceedings of Second Symposium on Pratical Software Development Environments.

[93] C. J. Van Wyk. A graphics typesetting language. In *Proceedings of the ACM SIGPLAN Symposium on Text Manipulation*, pages 99–107, Portland, OR, June 1981.

[94] Daniel Yellin. Generalized attributed parsing. Technical report, IBM T.J. Watson Research Center, July 1987.

[95] Daniel M. Yellin. *Attribute Grammar Inversion and Source-to-source Translation.* PhD thesis, Columbia University, 1987.

[96] R. Yeung. MPL - a graphical programming environment for matrix processing based on logic and constraints. In *1988 IEEE Workshop on Visual Languages*, pages 137–143, Pittsburgh, October 1988.

[97] I. Yoshimoto, N. Monden, M. Hirakawa, M. Tanaka, and T. Ichikawa. Interactive iconic programming facility in HI-VISUAL. In *1986 IEEE Workshop on Visual Languages*, pages 34–41, Dallas, TX, 1986.

[98] Kai Ching You and King-Sun Fu. A syntactic approach to shape recognition using attributed grammars. *IEEE Trans. on Systems, Man and Cybernetics*, SMC-9(6):334–345, June 1979.