1995

Syntax-directed Interpretation Of Visual Languages

Shane Denis Dunne

Syntax-Directed Interpretation of Visual Languages

by

Shane Dunne

Department of Computer Science

Submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

Faculty of Graduate Studies
The University of Western Ontario
London, Ontario
June 1995

ISBN 0-612-03449-6

Canada

Name SHANT DENIS DUNNE

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

COMPUTER SCIENCE

SUBJECT TERM

0984

SUBJECT CODE

U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS
- Architecture 0729
- Art History 0377
- Cinema 0900
- Dance 0378
- Fine Arts 0357
- Information Science 0723
- Journalism 0391
- Library Science 0399
- Mass Communications 0708
- Music 0413
- Speech Communication 0459
- Theater 0465

EDUCATION
- General 0515
- Administration 0514
- Adult and Continuing 0516
- Agricultural 0517
- Art 0273
- Bilingual and Multicultural 0282
- Business 0688
- Community College 0275
- Curriculum and Instruction 0727
- Early Childhood 0518
- Elementary 0524
- Finance 0277
- Guidance and Counseling 0519
- Health 0680
- Higher 0745
- History of 0520
- Home Economics 0278
- Industrial 0521
- Language and Literature 0279
- Mathematics 0280
- Music 0522
- Philosophy of 0998
- Physical 0523
- Psychology 0525
- Reading 0535
- Religious 0527
- Sciences 0714
- Secondary 0533
- Social Sciences 0534
- Sociology of 0340
- Special 0529
- Teacher Training 0530
- Technology 0710
- Tests and Measurements 0288
- Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS
- Language
  - General 0679
  - Ancient 0289
  - Linguistics 0290
  - Modern 0291
- Literature
  - General 0401
  - Classical 0294
  - Comparative 0295
  - Medieval 0297
  - Modern 0298
  - African 0316
  - American 0591
  - Asian 0305
  - Canadian (English) 0352
  - Canadian (French) 0355
  - English 0593
  - Germanic 0311
  - Latin American 0312
  - Middle Eastern 0315
  - Romance 0313
  - Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY
- Philosophy 0422
- Religion
  - General 0318
  - Biblical Studies 0321
  - Clergy 0319
  - History of 0320
  - Philosophy of 0322
- Theology 0469

SOCIAL SCIENCES
- American Studies 0323
- Anthropology
  - Archaeology 0324
  - Cultural 0326
  - Physical 0327
- Business Administration
  - General 0310
  - Accounting 0272
  - Banking 0770
  - Management 0454
  - Marketing 0338
- Canadian Studies 0385
- Economics
  - General 0501
  - Agricultural 0503
  - Commerce-Business 0505
  - Finance 0508
  - History 0509
  - Labor 0510
  - Theory 0511
- Folklore 0358
- Geography 0366
- Gerontology 0351
- History
  - General 0578
  - Ancient 0579
  - Medieval 0581
  - Modern 0582
  - Black 0328
  - African 0331
  - Asia, Australia and Oceania 0332
  - Canadian 0334
  - European 0335
  - Latin American 0336
  - Middle Eastern 0333
  - United States 0337
- History of Science 0585
- Law 0398
- Political Science
  - General 0615
  - International Law and Relations 0616
  - Public Administration 0617
- Recreation 0814
- Social Work 0452
- Sociology
  - General 0626
  - Criminology and Penology 0627
  - Demography 0938
  - Ethnic and Racial Studies 0631
  - Individual and Family Studies 0628
  - Industrial and Labor Relations 0629
  - Public and Social Welfare 0630
  - Social Structure and Development 0700
  - Theory and Methods 0344
- Transportation 0709
- Urban and Regional Planning 0999
- Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES
- Agriculture
  - General 0473
  - Agronomy 0285
  - Animal Culture and Nutrition 0475
  - Animal Pathology 0476
  - Food Science and Technology 0359
  - Forestry and Wildlife 0478
  - Plant Culture 0479
  - Plant Pathology 0480
  - Plant Physiology 0817
  - Range Management 0777
  - Wood Technology 0746
- Biology
  - General 0306
  - Anatomy 0287
  - Biostatistics 0308
  - Botany 0309
  - Cell 0379
  - Ecology 0329
  - Entomology 0353
  - Genetics 0369
  - Limnology 0793
  - Microbiology 0410
  - Molecular 0307
  - Neuroscience 0317
  - Oceanography 0416
  - Physiology 0433
  - Radiation 0821
  - Veterinary Science 0778
  - Zoology 0472
- Biophysics
  - General 0786
  - Medical 0760

EARTH SCIENCES
- Biogeochemistry 0425
- Geochemistry 0996
- Geodesy 0370
- Geology 0372
- Geophysics 0373
- Hydrology 0388
- Mineralogy 0411
- Paleobotany 0345
- Paleoecology 0426
- Paleontology 0418
- Paleozoology 0985
- Palynology 0427
- Physical Geography 0368
- Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES
- Environmental Sciences 0768
- Health Sciences
  - General 0566
  - Audiology 0300
  - Chemotherapy 0992
  - Dentistry 0567
  - Education 0350
  - Hospital Management 0769
  - Human Development 0758
  - Immunology 0982
  - Medicine and Surgery 0564
  - Mental Health 0347
  - Nursing 0569
  - Nutrition 0570
  - Obstetrics and Gynecology 0380
  - Occupational Health and Therapy 0354
  - Ophthalmology 0381
  - Pathology 0571
  - Pharmacology 0419
  - Pharmacy 0572
  - Physical Therapy 0382
  - Public Health 0573
  - Radiology 0574
  - Recreation 0575
  - Speech Pathology 0460
  - Toxicology 0383
- Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences
- Chemistry
  - General 0485
  - Agricultural 0749
  - Analytical 0486
  - Biochemistry 0487
  - Inorganic 0488
  - Nuclear 0738
  - Organic 0490
  - Pharmaceutical 0491
  - Physical 0494
  - Polymer 0495
  - Radiation 0754
- Mathematics 0405
- Physics
  - General 0605
  - Acoustics 0986
  - Astronomy and Astrophysics 0606
  - Atmospheric Science 0608
  - Atomic 0748
  - Electronics and Electricity 0607
  - Elementary Particles and High Energy 0798
  - Fluid and Plasma 0759
  - Molecular 0609
  - Nuclear 0610
  - Optics 0752
  - Radiation 0756
  - Solid State 0611
- Statistics 0463

Applied Sciences
- Applied Mechanics 0346
- Computer Science 0984
- Engineering
  - General 0537
  - Aerospace 0538
  - Agricultural 0539
  - Automotive 0540
  - Biomedical 0541
  - Chemical 0542
  - Civil 0543
  - Electronics and Electrical 0544
  - Heat and Thermodynamics 0348
  - Hydraulic 0545
  - Industrial 0546
  - Marine 0547
  - Materials Science 0794
  - Mechanical 0548
  - Metallurgy 0743
  - Mining 0551
  - Nuclear 0552
  - Packaging 0549
  - Petroleum 0765
  - Sanitary and Municipal 0554
  - System Science 0790
- Geotechnology 0428
- Operations Research 0796
- Plastics Technology 0795
- Textile Technology 0994

PSYCHOLOGY
- General 0621
- Behavioral 0384
- Clinical 0622
- Developmental 0620
- Experimental 0623
- Industrial 0624
- Personality 0625
- Physiological 0989
- Psychobiology 0349
- Psychometrics 0632
- Social 0451

ABSTRACT

Recent trends suggest that it will soon be practical to implement graphical user interfaces wherein a large part of the communication between human and computer will take the form of structured graphics, i.e., diagrams, notations, etc., which have come to be called "visual languages" in the literature. This thesis argues that implementation of such interfaces will be greatly facilitated by the development and use of syntax-directed translation methods for visual languages, and the embodiment of such methods in automated tools for application development, as has been done for compilers. An algebraic formalism to express syntax and semantics for visual languages is developed, and compared with similar formalisms developed by others. Tractability and algorithms for parsing are considered, and test implementations described. Several examples are presented, showing how real notations may be captured and parsed using the formalism. The design of interactive, visual language based application programs is considered in some detail, with emphasis on notations such as directed graph diagrams.

ACKNOWLEDGEMENTS

Special thanks are due to Dr. Richard Helm of IBM Thomas J. Watson Research Laboratory, Dr. Kim Marriott of Monash University, Dr. Kent Wittenburg of Bell Corporate Research, and Dr. Eric Golin of Brown University, for providing me with pre-publication copies of their latest research reports. I should also like to thank Mr. Joxan Jaffar of IBM Thomas J. Watson Research Laboratory for providing an interpreter for the constraint logic programming language CLP(R), which greatly facilitated my implementation experiments.

I am indebted to Dr. Dorothea Blostein of Queen's University, who shared with me her very considerable experience and insight into the problem of visual language parsing. Furthermore, her assistance in locating research materials was invaluable.

Initial material support for this work was provided by the Government of Canada, in the form of a scholarship from the Natural Sciences and Engineering Research Council. Final completion of this dissertation was made possible through the flexibility afforded me by my employers at the John P. Robarts Research Institute, especially my very patient supervisor Dr. Aaron Fenster. I should also like to express my personal thanks to Dr. Brian Rutt, also of the Institute, for his invaluable counsel during a very difficult period.

I should like to thank my advisor, Dr. Helmut Jürgensen, who believed in me and in this project from the very beginning, even at times when I did not.

Finally, I want to thank my wonderful wife and partner, Nancy Mucklow, whose encouragement, cajoling, patient listening, and hard work cleared the way for me to complete this dissertation.

TABLE OF CONTENTS

Chapter 1

Introduction

The computer can be many things, but above all it is a tool for human communication, primarily written communication.

But human writing takes many forms, from the numerous scripts used for recording written speech, which can collectively be called *text*, to the many specialized notations used in mathematics, engineering, science and the arts. The term *visual languages* has recently been coined to refer to these forms of writing, together with non-traditional, computer-specific forms such as visual programming languages (where pictures are used to specify computations).

Most of today's commercial word processors handle only text in European alphabets, though some support limited forms of mathematical notation, and support for Asian, Middle Eastern and other scripts is beginning to appear. Interactive drawing programs and computer aided design (CAD) systems provide limited support for other notational forms, but most cannot enforce subtle rules of layout and are hence clumsy; CAD systems for electronics and music composition and printing systems are important exceptions. Outside the commercial realm, a great deal of effort is being expended in attempts to extend the power of the computer to non-textual writing forms, e.g., Knuth's TEX [82] addresses mathematical notation, and many experimental systems have been developed for music, chemical notation, and other notations. (See [38, 39, 42] for references.)

These developments suggest that people desire the means to process non-textual notations as easily as they now deal with text. Given that this desire exists, it is appropriate to consider how we ought to design "notation processing systems". That is the subject of this dissertation.

1.1 Notations, notational systems, and visual languages

Let us define a *notation* as a collection of *marks* on paper or a display screen, having meaning to a person according to some set of established conventions, collectively called a *notational system*. E.g., an electronic circuit diagram is a notation; electronic circuit diagrams in general constitute a notational system.

The various script systems used for writing human spoken languages are of course notational systems, collectively called *text*. This dissertation deals primarily with notational systems other than text, such as

- standard positional mathematical notation
- chemical structure notation
- music and dance notation
- electrical and logic schematic diagrams
- diagrams of Petri nets, finite automata, and the like.

Notational systems are characterized by well-defined rules governing what arrangements of marks constitute valid notations, and the meaning of valid notations. Hence rough sketches and entirely *ad hoc* abbreviations do not qualify as notations, and neither do paintings or photographs.

The existence of rules governing validity (syntax) and meaning (semantics) makes notational systems akin to *languages*. Indeed, the term "visual languages" has recently come into use in the computer science literature, but its use is not entirely synonymous with the definition of notational systems given above. The differences are discussed in Chapter 2.

1.2 Notational systems and formal languages

It is beyond the scope of this dissertation to discuss whether or not notational systems are "languages" in the linguistic sense. Central to my thesis, however, is the notion that *formal languages* can be used to model those aspects of notational systems which are important for computation. My review of the literature, presented in Chapters 2 and 3, revealed that this notion is becoming widely accepted.

Established formal language theory does not provide adequate algebraic models to capture the syntax of notations, and a major contribution of this dissertation is the development of a new theoretical formalism, given in Chapter 4.

1.3 The structure of notations

We have defined a notation as a collection of marks, e.g. on paper, but we can be a bit less general than that. Most notational systems define certain very specific marks which are usually called *symbols*, and notations are two-dimensional arrangements of symbols. Many notational systems call for symbols to be connected e.g. by line segments; whether or not one chooses also to call the line segments "symbols" is largely a matter of taste.

Meaning in notations is conveyed by the symbols which are present, and by certain visually apparent relationships among them. Helm *et al.* [70] identify the following three classes of relationships:

1. *Network:* Lines, arrows, etc. connect related elements.
2. *Topological:* Relationships are indicated by containment, intersection and touching.
3. *Geometric:* Relationships are indicated by relative proximity, orientation, or size.

Most notations involve more than one class of relationship. For example, electrical schematic diagrams use network relationships to express how components are interconnected and geometric relationships to relate part numbers, voltage readings, etc. used as labels to the components, nodes, etc. they describe. Musical notation uses network relationships to indicate phrasing (beamed note-groups), topological relationships to denote pitch (intersections between note-heads and staff lines), and geometric relationships to associate ancillary matter (phrase marks, stylistic indications, lyrics) with notes. Venn diagrams are an exception, using topological relationships almost exclusively.

1.4 Notation processing systems

This dissertation is concerned with the design of *interactive notation processing systems*, i.e., interactive hardware/software systems which support *input, processing,* and

output of notations. The primary focus is on non-textual notations.

- *Input* in the interactive context means
  1. *editing*, normally visual editing (see below), and
  2. capturing aspects of notation structure and meaning in suitable data structures.
- *Processing* has two aspects:
  1. *Structural* or *cosmetic* processing, aimed at producing superior-looking output, e.g. automatic formatting.
  2. *Semantic* processing, which is concerned primarily with the meaning of the input notations, e.g. analysis, simulations, interpretation of notations as programs or data.
- *Output* can be a product or a process, depending on one's point of view.
  - Output as a product can be many things; for the purposes of this dissertation it means printed output, either on paper or a screen.
  - Output as a process means the process of creating appropriate data structures and/or communicating with operating system services to ensure that printed output will be available, either immediately (e.g. talking directly to a printer) or at some indefinite time in the future (e.g. by creating a data file in a standard format such as PostScript).

My M.Sc. thesis [39] addressed structural processing (automatic formatting) and output of notations. This dissertation concentrates on the input problem.

Of course, there are many notational systems, and one would prefer not to have to redo the entire design and implementation process for every one individually. By modelling notational systems as formal languages, we can abstract to a higher level and devise a generic design for a notation processing system, which can be customized for any notational system. This has not really been done for CAD systems, which tend to be purpose-built, one at a time. It has been done, however, for programming language compilers, on the basis of modelling programming languages as (augmented context-free) formal languages.

The standard design for a compiler involves three phases of processing:

1. The *lexical* phase identifies the low-level structures of the input program, yielding a representation which very much reflects the formal model of the language.

2. The *syntactic* or parsing phase develops the higher-level structure of the program, in terms of a derivation under the grammar which defines the formal model.

3. The *semantic* phase produces some kind of data structure which represents the meaning of the input program.

My thesis is essentially the proposal that a similar three-phase design can be used for notation processing. Since my interest is in *interactive* systems, my design bears more resemblance to an interactive programming environment (see below) than to an ordinary compiler, but the essential concept is the same. That is,

- A formal language model is chosen, which is sufficiently expressive to capture the essential characteristics of a wide variety of languages (in this case notational systems).

- A generic design for language (notation) processors is developed, based around a parsing algorithm for the formal language model.

- Specific systems are implemented by customizing the design. Language-specific (notational system-specific) details are determined according to the formal specification of the formal model.

This notion of generic design is not sufficiently well developed that I can make major claims about it, but it is important in that it represents the conceptual context of my work on formal language representations. It is discussed throughout this introductory chapter, and again in more detail in Chapter 6.

1.5 Visual editors, structure, cognitive overhead

In interactive systems, the issue of input develops into the somewhat more complex issue of *editing*. In the modern setting this involves the use of a *visual editor*. A visual editor is a program[1] which

- maintains some kind of *data structure(s)*
- presents one or more visual representations (called *views*) of the data structure(s) to a user
- interprets a language of *commands* (called its command language), which are user actions mediated by input devices and software
- permits the user to effect, via commands, incremental changes to the structure(s), which are reflected in (usually) visible changes to the view(s).
- is nearly always used in conjunction with (or is part of) other software (operating systems, compilers, document processors, CAD systems).

Using a visual editing system comprised of a visual editor, related software, and the necessary computing and I/O hardware, is vastly more complex than writing with pen and paper. The complexity arises because the user, *while trying to concentrate on what he or she is trying to write,* must remain aware of

1. at least some aspects of the data structure being edited
2. the nature of the relationship between the data structure and its visual representation(s) which appear in the view(s)
3. the incremental manipulations which the editor supports, and the commands required to effect them
4. aspects of the system environment, e.g. the fact that a mouse click in a certain screen region will cause a switch to an entirely different program.

Let us call these the *cognitive overhead* involved in using an editing system.

Cognitive overhead is something of a necessary evil in software design. Systems with low cognitive overhead tend to be lacking in features, while feature-rich systems

[1] or part of a program. Throughout this dissertation the word "program" is used in this generic way.

tend to have high cognitive overhead. The basic problem is that nothing is achieved without communication between human and computer, and that communication takes the form of views and commands, whose form is strongly influenced by aspects of the data structures and the system environment.

My generic design for interactive notation processing systems promotes low cognitive overhead, for the following reasons:

- High-level (syntactic) structure is inferred automatically via parsing techniques. This allows the user to work with a flat mental model of the editor data structure, i.e., the individual marks of which notations are composed.

- The parsing technique is "lazy" in the sense that the user must identify groups of marks to be parsed. Furthermore parsing decisions may be retracted, either automatically when marks are deleted, or manually upon request of the user. These methods put the user in control of the parsing process, allowing him or her to alternate between unencumbered creative work (writing) and more formalized interaction (driving the parser).

- The proposed approach encourages uniformity of data structure, views, and command languages among multiple applications, which reduces cognitive overhead across entire integrated systems.

1.6 Some notes on technology

When I first began considering the issue of notation processing in 1988, I believed that the technology now called "pen-based computing" would be an important component of any notation processing system. I still believe this, but the matter of I/O technology is no longer central to my thesis as it was in my original proposal. Nonetheless, it is worth spending a few moments considering the kinds of technology which might, in the near future, be used to realize the ideas presented in this dissertation.

Historically, the ability of computers to process the many forms of human writing was severely limited by available hardware. The first computers had such small registers and memories that efficient bit-packing was a critical issue; the result was that even the distinction between upper and lower-case letters had to be ignored. When CPU and memory capacity increased, generations of successively higher-resolution raster printers and bit-mapped CRT displays gradually provided for case and even font

distinctions, and eventually proportional character spacing and WYSIWYG ("what you see is what you get") interactive displays.

Today's computer workstation, featuring a high-resolution CRT display, laser or ink-jet printer, keyboard and mouse, is highly unbalanced. Its graphical output capabilities extend to multiple fonts, international script systems, and graphical renderings whose quality approaches the photographic, but its input capabilities are nowhere near as sophisticated. The keyboard, a superb input device for touch-typing of European text is clumsy when used for mathematics or Asian scripts. The mouse, very good for pointing and selecting displayed objects, is clumsy for drawing and useless for writing.

Graphics tablets are devices which continuously sample and report the position of a pen-like stylus on a flat drawing surface. Any tablet which can track stylus movement sufficiently quickly and accurately can in theory be used as an input device for handwriting and sketching. Since the first tablets were introduced in the 1960's, automatic handwriting recognition has been an attractive, but so far elusive goal. Recently, *interactive tablets* have been developed, in which a flat-panel liquid crystal display (LCD) is integrated into the writing surface. These devices are even better suited to handwriting and drawing, because it is possible to leave a trail of "ink" (illuminated pixels) behind the pen on the writing surface.

Interactive tablets (and complete "pen-based computers" incorporating them) are today limited to niche markets because of high cost, low display resolution and computing capacity, and the lack of acceptable handwriting recognition software, but this situation is changing. Display resolution and computing power are improving incrementally for interactive tablets just as for other kinds of computers. Some techniques for automatic handwriting and symbol-recognition have been devised (see [139]), but require significant computing power; these will have to wait until such power is available. In the meantime, more expedient methods which require users to learn and use a simplified input alphabet have become widely accepted.

In light of these developments, it would appear that excellent hardware is now or will soon be available to support interactive notation processing. Our ability to take advantage of such hardware, however, will depend critically on appropriate software designs; hence the motivation for my research.

1.7 The problem: a generic design for notation processing systems

To summarize, my thesis concerns the design of interactive systems for processing notations other than text. The key issues I have considered are:

- Development of a generic design for such systems, which can be customized for any notational system, in order to focus attention on major design issues and to encourage code re-use.
- Elaboration of a formal model capable of capturing the structure of notations, upon which to base the generic design.
- Designing for interactivity.
- Designing for emerging graphical input and display technologies such as interactive tablets.

Having investigated all of these aspects in some detail, I chose to focus primarily on the second issue—the lack of an appropriate formalism. When I began my research, very little had been published on this question, and hence I decided to propose a new formalism, presented in Chapter 4. Recently, however, several formal models for visual languages have been proposed (see Chapter 3), which suggests that I was correct to focus on this aspect of the problem.

I have also considered parsing with the proposed formalism, which is discussed in Chapter 5, and designing complete interactive notation processing systems, which is examined in Chapter 6.

1.8 Proposed solution: guided parsing within an editing paradigm

I suggest that, given an appropriate grammar formalism, notations can indeed be parsed like computer programs, and that it should be possible to

- achieve modularity in design and implementation by separating processing into lexical, syntactic and semantic phases.
- understand (visual, notational) language classes and their complexity, and on the basis of that understanding make informed choices about what kinds of grammars to use and parsers to build.
- develop tools to systematize and partially automate the process of implementing notation processors (customizing the generic design), such as parser generators.

These goals have already been realized in the field of compilers, and a good generic design for compilers has already been developed.[2] To develop a generic design for *interactive* notation-processing systems, however, inspiration comes not from compilers *per se*, but rather from *interactive programming environments* or IPE's. An IPE is an interactive software system which facilitates development of programs in some programming language, providing the functions of editing, compilation, and execution/debugging, integrated to a greater or lesser degree.

To date, three essentially different styles of IPE's have been developed.

1. The *traditional, non-integrated* IPE, ranging from environments such as the UNIX operating system to "Turbo Pascal" (Borland, Inc., Scotts Valley, CA) and its derivatives, combines a conventional visual text editor and batch-mode compiler, with convenient ways to switch between what are really independent editing and compilation phases.
2. A *syntax-directed program editor* such as the Cornell Program Synthesizer [140], uses a program's parse tree, rather than its text, as its basic data structure, and provides commands which effect transformations on the parse tree. Some systems of this type don't need to do any parsing at all; the user in effect specifies the derivation of the program in terms of its grammar, via a sequence of commands which transform an initially empty parse tree into the parse tree for the desired program.

[2] See any text on compiler design theory, such as [1].

3. A *textually-oriented program editor* permits the user to modify the program text directly, like a conventional text editor, but also applies parsing to build up a forest of partial parse trees as an auxiliary data structure, and maintains consistency between the two structures as editing proceeds.

The first approach is by far the easiest to implement, and hence the most common, but also offers the least number of advanced features. The second approach has failed in practice, essentially because of cognitive overhead; programming is difficult enough without having to think about syntactic structure before one's program is even complete. The third approach has been defended by Lee [86] as the most intuitive, largely because it obviates this cognitive overhead, but it is by far the most difficult to implement.

These three styles or approaches to IPE design can be applied to interactive notation processing systems.

1. A batch-mode parser can be constructed, entirely independent of any editing capability. This is the approach I took in some early experiments. described in Chapter 5, and which has also been taken in recent work at IBM [70].

2. A *syntax-directed notation editor* would manipulate some kind of syntax description as its basic data structure while presenting notations in its views. This approach has been tried by Göttler [61].

3. A *graphically-oriented diagram editor* would handle simple geometric primitives such as line segments, etc. in its basic data structure, but attempt to parse these into a parallel syntactic description. This approach has been tried by Wittenburg [154] at MCC, and most recently at MIT [149] (see Chapter 3).

The first approach fails as soon as the first parsing error, or ambiguous situation arises. In an interactive setting, the user could perhaps be alerted to the problem and asked to either modify the input or choose between parsing alternatives, but if such requests come too frequently, or are too repetitive, they will constitute cognitive overhead.

The second approach will have the same drawbacks in the domain of notations as it has in the domain of programming. Forcing users to think about syntax, when what they are most concerned with is meaning, is unacceptable cognitive overhead.

The third approach, which I believe to be ultimately the most useful, can be seen as a middle ground between the first and second approaches. The first approach

tries to do everything with parsing, while the second approach doesn't do any parsing at all. In the third approach, the system parses what it can and waits for further information concerning the rest. In Chapter 5 I consider ways in which the parsing might be managed so as to avoid badgering the user with requests for clarifying information. Chief among these is the notion of *guided parsing*, where the user is able to direct the parsing process so as to avoid most ambiguity entirely. Note also that a system built according to the third approach, because it maintains a (partial) syntactic description of the input, can in theory offer the kinds of syntactic operations (automatic selection and manipulation of syntactic units) which are the main strength of the second approach.

1.9 Thesis

I am concerned with the design of *interactive notation processors*—interactive software systems where the communication between human and computer mainly takes the form of notations, as defined above. My thesis is that notations can be processed in ways reminiscent of the translation and interpretation of programs. The paradigm of textually oriented program editors can be adapted for notations, in which case we speak of "graphically oriented notation editors". I claim the following:

1. Many notational systems can be modelled using a variant of formal languages, which I call *attributed set languages* (ASL's) and define in Chapter 4. These languages are defined by grammars, called *attributed set grammars* (ASG's).

2. Attributed set encodings of notations are natural and intuitive, and well-suited to automatic graphical interpretation.

3. ASGs can be used both to generate and recognize (parse) attributed set encodings of notations.

4. Practical problems in ASG-based parsing, such as cognitive overhead due to too-frequent requests to the user for clarification, and combinatorial explosion of options to be considered in parsing, can be managed by adopting a guided parsing approach, which gives the user control over the parsing process.

5. The proposed notation processing paradigm is implementable, in a manner consistent with good software engineering practice.

Attributed set languages and their grammars are defined in Chapter 4. Directed graphs and trees (and tree diagrams) are used as key examples to illustrate the facility with which complex notational systems can be modelled by attributed set languages defined by ASG's, in contrast to other formalisms reviewed in Chapters 2 and 3. Chapter 4 addresses not only syntactic but also semantic aspects of the attributed set formalism, in order to show how ASG's can be used to generate representations of notations.

Chapter 5 discusses ASG-based parsing. Building on the directed graph and tree examples developed in Chapter 4, I show how the attributed set languages used to model these notations can be parsed according to the grammars for those languages. The parsing algorithm is not particularly efficient; in fact it requires time exponential in the size of the input. However in the interactive setting this inefficiency is not critical; I suggest a technique of interactively guiding the parsing process which effectively "tames" the exponential complexity by reducing the amount of input which the parser must handle at one time.

The ideal evidence for a claim of implementability would be a full, working implementation, but in my case this is impractical. I have, however, implemented a few key aspects of the paradigm, in the form of parsing programs in the logic programming language CLP(R) [67, 68]. These are discussed in Chapters 4 and 5. Software engineering issues such as code re-usability and development tools are discussed in Chapter 6, in which I suggest a generic design for interactive notation processing systems.

1.10 Plan of the dissertation

This dissertation consists of seven chapters. The first is this introduction, which motivates and defines the problem of designing interactive notation processors, outlines my proposed approach to the problem, and presents the statement of my thesis along with a summary of the arguments to follow.

Chapter 2 presents a general review of related work. Five distinct perspectives on the subject are identified: pattern recognition, language theory, interactive programming environments, visual languages, and computer-human interaction. The chapter reviews the literature in each of these areas, and concludes with a summary of relevant contributions from each.

Chapter 3 reviews five specific published visual language formalisms in some de-

tail. These are Graphic Functional Grammars (due to Kojima and Myers), Relational Grammars (due to Wittenburg *et al.*), Attributed Multiset Grammars (due to Golin and Reiss), Constraint Multiset Grammars (due to Marriott *et al.*), and Conditional Set Rewrite Systems (due to Najork and Kaplan). The chapter concludes with an discussion of the strengths and weaknesses of the various approaches.

My own formalism is introduced in Chapter 4. I define *attributed sets*, and develop a formal semantics for their interpretation as e.g. graphics. I then define the notion of an *attributed set grammar* (ASG), which generates sets (languages) of attributed sets. I show how attributed set languages (ASLs) can be used to model algebraic and notational structures, including strings, directed graphs, binary trees and tree diagrams with layout constraints, and in each case give an ASG which g .ierates the appropriate ASL. I identify a class of *monotonic* ASGs, in which rewriting according to productions always adds new symbols or introduces terminal symbols, and show that the membership problem for languages of such ASGs is solvable. I suggest a simple parsing algorithm with exponential time complexity, and present a working implementation in Prolog.

Chapter 5 discusses ASG parsing. It begins with an account of some early implementation experiments, which I performed before having fully developed the ASG formalism. This is followed by a discussion of the parsing algorithm introduced at the end of Chapter 4, which is actually much simpler than the approach used in the first experiements. I show how constraint testing may be directly integrated into the algorithm, and give working examples implemented in CLP(R). The question of *reduce/reduce conflict* (when more than one grammar production might be applied in a given context in a bottom-up parser) is then considered. Finally, I discuss how the parsing technique can be adapted for use in an interactive setting where parsing should be done incrementally, and it is necessary to adjust to changes in the input due to the user's editing actions.

Chapter 6 deals with design of interactive notation processing systems, based on extension of the notion of syntax-directed program editing to the domain of visual rather than textual languages, using the ASG formalism developed in Chapter 4 and the parsing methods discussed in Chapter 5.

The final chapter contains a discussion of the proposed approach, conclusions, and suggestions for future work.

Chapter 2

Six Perspectives on Interactive Notation Processing

In this chapter I present a broad survey of work related to issues arising in interactive notation processing. Due to the large number of works cited, the coverage in this chapter is broad and shallow. The following chapter contains a more in-depth examination of five specific research initiatives. My aim in these chapters is to:

1. present a historical review of the entire field of notation processing.
2. distill some general principles and "lessons" from others' experience.
3. distinguish the focus of my own work.

2.1 Perspectives on Interactive Notation Processing

To the best of my knowledge, there are as yet no books on "notation processing". The notion of notation processing as a distinct focus of study appears to be emerging slowly, as developments in computing hardware (graphical I/O devices, memory capacity and CPU power) begin to make interactive processing of non-textual symbolic data appear potentially practical [70, 92, 154]. A great deal of relevant work has been published in several research disciplines, each of which has its own perspective, i.e., its own historical tradition of approach, goals, and methodology. I have identified six distinct perspectives within the literature:

1. the *pattern-recognition* perspective, which focusses on the problem of engineering machines to .ecognize notations;

2. the *language theory* perspective, which sees notational systems as a class of formal languages, and aims to apply methods and results from the theory of formal languages to the problem of defining and parsing notations;

3. the *artificial intelligence* perspective, which includes natural language processing and methods of automatic logical inference;

4. the *interactive programming environments* perspective, which addresses the problem of processing complex programming language structures within an interactive context;

5. the *visual languages* perspective, which posits that pictures might profitably be used to represent computations and other well-defined processes or structures, and considers the problems involved in doing this practically;

6. the *computer-human interaction* perspective, which considers the design of interactive graphics applicatic ns in general.

2.2 The Pattern-Recognition Perspective

The idea that computers might be used to recognize and process human notations was first explored seriously within the field of pattern recognition, notably by the late K. S. Fu at Purdue University, whose 1974 book "Syntactic Methods in Pattern Recognition" [53] introduced some notions from formal language theory to a discipline then dominated by statistical methods, suggesting that these ideas might provide a good basis for a new kind of pattern recognition, involving hierarchical composition (parsing) of lower-level recognition results to yield a high-level pattern description.

Today, the pattern recognition field recognizes three main approaches:

1. The *statistical* or *decision-theoretic* approach is based on the idea of capturing salient aspects of inputs in the form of a vector of numeric "features" and identifying which of several predefined pattern classes it belongs to, by comparing the input feature vector against so-called template vectors which are exemplary of each class and choosing the class whose template matches best. This is the oldest approach, and the one which is best supported by mathematical theory (predominantly statistics, specifically Bayesian reasoning).

2. The *structural* approach is based on the idea of combining the results of low-level recognition processes, each of which deals with part of an input pattern, to obtain a structured *description* of the input. Syntactic methods are but one kind of structural composition technique.

3. The *connectionist* or neural-network approach has at its heart the concept of a "learning machine" which obviates the need for analysis of specific pattern-classification applications by being able to "learn" the desired classification behaviour based on a training process in which patterns are presented to the machine together with their correct classification.

A good summary of the field is given in [123]. Of course, the various approaches may profitably be combined; hybrid methods are described in [11].

The pattern recognition literature reflects a strong engineering orientation. On the whole, the work in the field has been directed toward building machines (with or without a software component) to function in real-world environments, rather than developing a theory of such machines. Matsushima *et al.* [93], for example, describe a robot system which reads pages of printed music seen through a television camera and plays them on an electronic keyboard with mechanical hands and feet. Most works are more modest, focussing on one or two techniques applied to one notation. In music recognition, techniques applied include syntactic methods for high-level structure [5], and the use of domain-specific knowledge [80]. Other applications studied include handwritten letters and numbers [51, 74] (see also [139]), dimensions in technical drawings [24, 33], and electrical circuit symbols [87].

The pattern recognition field emphasizes the importance of recognizing patterns (whatever that may mean in specific applications) despite the presence of an expected amount of variation, both random ("noise") and systematic (varying image size, orientation, etc.). In decision-theoretic methods this is handled by introducing continuous measures of classification, and reasoning according to statistical theory, or more recently using fuzzy-logic methods; see [155]. In structural and syntactic methods, the usual approach is to try to build some flexibility into the underlying matching mechanism or into the grammar itself. With respect to the former, Wang and Pavlidis [145] discuss optimal inexact matching of strings, and Blostein and Haken [7] give a more applied example concerning inexact matching of rhythm templates in music analysis. With respect to the latter, there has been much interest in stochastic grammars. Fu [53] devotes two chapters to them, and see [19] for a recent example

involving mathematical equation recognition.

Another important approach to dealing with input variability is some kind of preprocessing step, which has three aims:

1. Reduce or eliminate the (usually random) variation, e.g. see [90] for a method to reduce noise in pen-strokes captured by a digitizing tablet.

2. Reduce the sheer bulk of data involved to simplify the ensuing computations, e.g. see [101, 97] for curve encoding methods.

3. Perform a transformation to a different domain of representation, wherein the key computations required for recognition are simpler, and/or where systematic input variations are collapsed. See [105, 97] for methods of encoding curves which are invariant under common geometric transformations.

The references given in this list are ones I believe to be specifically relevant to notation processing, especially processing of hand-drawn notations captured on a tablet, but discussions of techniques of noise reduction, data reduction and transforms abound in the pattern-recognition literature.

Much of the pattern-recognition research which is relevant to notation processing is aimed at developing systems for *document image processing*, i.e., systems which can "read" paper documents and produce some kind of data structure representation of their contents with minimal human intervention. Recent surveys of the document image processing literature are [78], which presents an overview, [146], which focusses on recognizing the structure of text, and [79], which examines techniques for graphics. Two bibliographies focussing on music document recognition are [130] and [6].

As might be expected, this literature is dominated by the problem of *segmentation*, i.e., determining where the relevant structures are within vast pixel arrays. The survey of graphics recognition techiques mentioned above [79] for example devotes twenty-six pages to preprocessing and segmentation issues but only seven pages to "graphics recognition and interpretation".[1] A 1993 paper entitled "MUSER: A prototype musical score recognition system using mathematical morphology" [96] says much about segmentation but very little about recognition of structure.

[1] Furthermore, syntactic methods are not even mentioned in these seven pages. Syntactic techniques do not appear to be popular in the pattern recognition field, probably because they are not yet adequately developed; see [77] for a critique of the limitations of the syntactic approach.

An aspect of the machine-orientation of the pattern-recognition field is dominance of fully automatic "hands-off" mechanisms as opposed to semi-automatic, human-assisted methods. The result is that few ideas relevant to the design of interactive systems are to be found in the pattern-recognition literature.

Another dominant idea in pattern recognition is the idea of *inference*, i.e., that a pattern-recognition system ought to be able to infer the basis for its proper operation from exposure to a *training set* of input patterns paired with the expected output. This is most apparent in the neural-network approach, but Fu [53] devotes a chapter to automatic inference of grammars in the syntactic approach. Some progress appears to have been made on grammar inference; see [123, page 200] for some references.

2.3 The Language Theory Perspective

Algebraic formal language theory, introduced by the linguist Noam Chomsky in a 1956 paper [18], was quickly adopted by computer scientists and expanded to yield a powerful theoretical basis for the design of programming languages and their translators (interpreters and compilers). The idea that similar success might be possible in notation processing has occurred to many people; I call this idea the "language theory perspective".

Traditional formal language theory deals with *strings*—sequences of atomic symbols drawn from fixed, finite alphabets—which are essentially perfect models for computer languages composed of e.g. ASCII-coded characters. However for visual languages, and pictures in general, it is not at all obvious what kind of mathematical model should be used. The history of the language theory perspective until very recently could be summed up as a search for appropriate models.

The earliest efforts to apply formal language methods to pictorial data involved attempts to model visual structures using strings. These fall into two categories:

1. using characters to encode segments of the boundary of a two-dimensional shape
2. using one alphabet to encode primitive shapes, and a second one to encode (geometric, topological) relationships between sub-pictures

The first approach is based on the observation that any shape in two dimensions has an outline which is one-dimensional. A start point on the outline may be chosen arbitrarily, and the outline may be "walked" from there in an arbitrarily-chosen direction until the start point is reached again. (Shapes with holes can either be defined

by multiple outlines or by using a contour-walking algorithm which may cross the interior of the shape in certain cases as described in [136].) In practice the outline is walked in a discrete fashion, thus defining distinct segments which may be encoded as characters in some alphabet.[2]

A popular encoding technique is chain coding [50], where the unit of division is the distance between 8-connected points on a regular Cartesian grid (e.g. adjacent pixels) and the 8 possible line segment orientations are encoded as octal digits. This encoding has been studied in some detail; Tou [141] discusses topological aspects while Siromoney *et al.* [134] examine the application to syntactic pattern recognition. A generalized form of chain coding does not require a grid, but splits object outlines at corner points and encodes the orientation of the resulting line segments using the same 8 standard codes; Stallings [136] uses such a method to encode Chinese character shapes. Yet another encoding technique is to divide outlines into approximately equal-length segments and encode each segment's curvature, rather than orientation; this method has been used [53, Appendix A] to encode chromosome shapes.

One of the earliest examples of the second approach is Shaw's "picture description language" (PDL) [131], which encodes pictures composed of "primitives" possessing two attachment points called "head" and "tail". The primitives themselves may be anything; from a structural point of view they may be treated like arcs in a directed graph. PDL encodes each primitive using a different letter; sequences of such letters encode chain graphs, which as a whole have one head and one tail. Also provided are five "operators", one being a unary prefix operator which reverses the head/tail orientation of its operand, the other five being binary infix operators which encode various ways of attaching its two operands. The logical structure of a PDL-encoded picture is a directed acyclic graph (DAG) which has one distinguished initial node (tail) and one distinguished terminal node (head).

[2]Choosing a different starting point corresponds to a circular permutation of the resulting code word. Changing the walk direction corresponds to reversal of the code word.

A similar approach has been used more recently in computer graphics, to interpret strings generated by *L systems* as branching structures such as plants. L systems were first proposed by the late Aristid Lindenmayer in 1968 [88] as an algebraic formalism to model development of living organisms at the cellular level. They are similar to conventional string grammars, except that:

1. no distinction between terminal and nonterminal symbols is made.
2. the entire *sequence* of words produced during a derivation (which may be infinite) is important.
3. at each derivation step, all symbols in a word are rewritten in parallel.

Alvy Ray Smith of Lucasfilm first proposed the use of L systems to generate realistic image sequences illustrating growing plants and trees [135]. Further work in this area has been done by Prusinkiewicz [111, 112], Hanan [66], and Jürgensen and Chien [16].[3]

Briefly, the encodings used with L systems are similar to those of PDL but their logical structure is trees rather than DAGs.

Recall the 3-way taxonomy of relationships (network, topological, geometric) due to Helm *et al.* mentioned in the previous chapter. In PDL and the various L-system languages, the "operator" characters encode what Helm *et al.* would call *network* relationships. The same approach has also been used to encode geometric relationships, the earliest example being Anderson's 1968 Ph.D. thesis [53, Appendix C] on syntactic recognition of two-dimensional mathematical expressions. In 1989, Chou [19, 20] applied advanced stochastic parsing techniques to the same problem, using essentially the same encoding. Recently (1993) Salter [122] has given examples where topological relationships are encoded.

Since I began my research, many papers have been published which further explore the use of special symbols to represent relations in string encodings of shapes; for lack of a standard term I call this approach *relation coding.* [72, 27, 47, 25] focus on efficient parsing, [155] integrates fuzzy-logic methods into the parser to deal with input variations, and [26] discusses automatic generation of efficient parsers via a parser generator. While these certainly represent important contributions, they are not directly relevant to notation processing because their underlying assumptions

[3] A more complete account of the history of L systems applied to graphics, with additional references, is given by Jürgensen and Chien in [16].

about visual language structure, arising from a determination to use string encodings, are too restricted.

Recognizing the limitations of string encodings, language theorists sought to extend the classical definitions of string-generating grammars (and string-recognizing automata) to more complex structures. One of the first such extensions involved generalizing from strings to (n-dimensional) arrays [119, 118, 117, 102]. As with relation coding, the array paradigm is simply not suitable for the kinds of notations I am interested in. [123] discusses generalization from strings to trees, which is really nothing new since relation coding provides a natural encoding for tree structures based on inorder (preorder, postorder) traversal using infix (prefix, postfix) relation operators.

Of greater interest in the context of notation processing are *graph grammars*, which have been the subject of annual international conferences since 1983.[4] Fahmy and Blostein [45, 44] have investigated applications to recognition of musical score structures, Göttler [61] has investigated syntax-directed diagram editors based on graph grammars (specifically programmed, attributed graph grammars), Pfeiffer [106] considers the use of graph-grammar techniques to replace pointers in programming languages, and Schürr [127, 126] presents very recent work related to visual programming (see later in this chapter).

Devising a graph grammar formalism which is simple, powerful, and useful in applications has proved difficult. The earliest proposals, which defined productions as rewriting subgraphs of a graph, resorted to cumbersome formulations to deal with the *embedding problem*, i.e., specifying exactly how connections to the rest of the graph (the parts not being rewritten) were to be preserved through the rewriting step. A key step toward solving this problem was the NLC (node label controlled) rewriting approach introduced by Rozenberg in 1986 [120].

The NLC formalism, while compact and powerful, had the disadvantage that many basic questions concerning NLC graph grammars turned out to be undecidable. A more fruitful approach was introduced the following year by Habel and Kreowski [64], involving a generalization from graphs to *hypergraphs*. In a hypergraph, edges are generalized to *hyperedges* having several incoming and outgoing "tentacles" (the ordinary edge with one each is then a special case). *Hyperedge Replacement Systems* (HRS) are hypergraph-generating grammars in which hyperedges are replaced by

[4]See the various Proceedings published by Springer-Verlag, in the *Lecture Notes in Computer Science* series, volumes 73 (1979), 153 (1985), 291 (1987), and 532 (1991). Each proceedings volume bears the title *Graph Grammars and their Application to Computer Science*.

hypergraphs. The embedding problem is dealt with by requiring that productions specify the exact correspondence between tentacles of the hyperedge being replaced and nodes of the hypergraph which replaces it; these nodes are not actually included in the rewriting, but stand as placeholders representing the nodes to which the rewritten hyperedge was attached.

Many results on HRS have been published in a 1992 book by Habel [63] and since then, further results concerning parsing algorithms and complexity have been published by Drewes [34, 35, 36]. A rigorous mathematical treatment of graph rewriting systems, including both graph and hypergraph formalisms, is given by Courcelle [29].

A number of interesting formalisms predated the recent unifying developments in (hyper)graph grammars. Two of the most important are *web grammars* introduced by Rosenfeld in 1969 [109] and *plex grammars* introduced by Feder in 1971 [46]; both are summarized in [53]. Web grammars, which generate labelled directed graphs ("webs") are mostly of historical significance; Rozenberg [120] credits them as having originated the area of graph grammars. Plex grammars generate "plex structures" consisting of so-called NAPEs (n attaching-point entities). What is particularly interesting about plex structures is that connections are specified explicitly, rather than implicitly as in graphs. For use in modelling notations, this feature is intuitive; we have a notion of symbols (modelled by NAPEs) and a distinct specification of their interconnections. Feder's original paper gave examples of modelling chemical structures, logic circuit diagrams, and program flowcharts. Interest in plex languages for pattern recognition persisted [12] until the introduction of HRS. [63] proves that HRS and plex grammars are essentially equivalent.

Hypergraphs are quite expressive for modelling notations, but they are inadequate for my purposes because they model only the connectivity structure of notations, not dimensions, orientation, or layout. Hence while hypergraphs might be a good choice for modelling electronic circuit diagrams, they would be poor choices for Venn diagrams, mathematical formulae, or music notation.

In order to model dimensions, orientation and layout of notations, we require representations which can carry metric information. Atomic objects like vertices and hyperedges need to be augmented with *attributes*. Attributed grammars for strings [28, 37] have been studied for some time, and applied with great success to compiler design [1]. Attributed representations and related grammar formalisms are discussed in Chapter 3. chapter 4 introduces the attributed set formalism, which is essentially an attributed version of hypergraphs and HRS.

2.4 The Artificial Intelligence Perspective

Notations are human languages, which happen to be visual rather than spoken. With the exception of new notations designed specifically for computer processing (see next section), notations arose specifically because the human cognitive apparatus is capable of processing them. Hence, the field of artificial intelligence (AI), which seeks to understand and emulate human cognition, also constitutes an important perspective on the problems of notation processing. It is not my intention to survey the vast field of AI here. Rather I will concentrate on two key developments:

1. progress in natural language understanding, especially new grammatical formalisms and associated parsing algorithms.

2. the use of formal logic to describe, reason about, and even implement computations (logic programming).

While in computer science the main goal of research into language formalisms has been *efficient parsing*, in natural language understanding (NLU) the focus has been on *expressive power*, the main problem being to capture the subtleties of natural languages whose design makes no allowances for the limitations of known computer processing techniques. To this end, a variety of very general language formalisms and associated parsing algorithms have been developed for NLU [3].

Two of the most powerful language formalisms developed for NLU are *definite clause grammars* (DCG) [108], also called *logic grammars*, and *unification grammars* [132]. Both are closely related to developments in logic programming.

A DCG is a description of a language expressed in the "definite clause subset" of first-order logic, and can be interpreted as a Prolog program, which is a parser for the language. DCGs extend the usual context-free string grammar formalism in three ways:

1. DCGs provide for context-dependency.

2. A DCG parser can build arbitrary tree structures as it works, which can be used to represent the meaning of the input string.

3. Since a DCG is a Prolog program, it may perform any amount of auxiliary processing in addition to parsing. Most importantly, reduction of input structures can be made conditional, according to the outcome of auxiliary processing.

It should be noted, however, that although a DCG description of a language is an executable parser for that language when interpreted as a Prolog program, it may not be an efficient one:

> Normally a substantial transformation wuld be necessary to turn a DCG conceived as a theoretical description of a language into a practical implementation. It is an interesting problem for future research to see whether such transformations can be performed systematically, possibly by generalising known results on parsing with context-free grammars. [108, page 270]

The DCG formalism has been applied to syntactic pattern recognition of gene structures (expressed as base sequences in one dimension), sailboat structure (expressed as sets of line segments forming outline drawings in two dimensions), and postal bar codes (one-dimensional bit strings, with possible multi-bit errors) by Searls and Leibowitz [129]. The work of Helm and Marriott at IBM [70], discussed further in the next chapter, could also be characterized as a generalization of the DCG formalism. The ASG formalism, described in chapter 4, is most conveniently implemented in a constraint logic programming environment, and hence is also related to the DCG formalism.

Unification grammars are described succinctly by Pulman [113] as follows:

> The prototypical unification grammar consists of a context-free skeleton, enriched with a set of *feature + value* specifications on the grammatical symbols in the rules and associated lexicon. These feature specifications may involve variables, and may be recursive (i.e., the values may be interpreted as referring to a whole category). Whereas parsing and generating

> sentences using grammars with atomic grammatical labels involves a test for equality between symbols in a rule and those in a tree, in unification grammars the test is whether two non-atomic descriptions "unify," i.e., can be made identical by appropriate mutual substitutions of terms. (Emphasis in original.)

Unification grammars have been used by Wittenburg [154] for describing and parsing notations such as simple mathematical expressions and flowcharts. To deal with two-dimensional geometric relationships which are part of the specification of notations, Wittenburg extends the unification principle to incorporate general function evaluation, according to the method proposed by Ait-Kaci and Nasr [2]. The extensions allow the unification of variables with applicative functional expressions, with mechanisms to delay actual evaluation of the expressions until all required data are available. The result is a general mechanism for reasoning with constraints. Wittenburg's work is described further in the next chapter.

The notion that graphical structures can be succinctly described as a set of attributed primitives or "marks" together with a set of inequalities or *constraints* on their attributes has a long history in computer graphics, beginning with Sutherland's highly influential SketchPad system [138]. The central notion of in [39] was that the structure of notations, for typesetting purposes, is best expressed in terms of marks and constraints. The central notion of this work is that the same idea holds for recognition purposes as well.

Solving systems of constraints, and more generally, reasoning with constraints, has been studied extensively in AI, culminating in the development of *constraint logic programming* (CLP) systems [76, 67]. CLP is a general scheme for implementing logic programming languages in which the basic operation of unification of atoms with the head of a rule is replaced by a more general constraint solving procedure. The scheme is "instantiated" to yield a specific CLP language by implementing a constraint solving procedure for a chosen domain; an example is the language CLP(R) [68], which handles constraints in the domain of real numbers. As with the integration of logic and functional programming [2], implementations usually include some kind of lazy evaluation to delay ev[illegible]ion of functional expressions until all argument values are known.

In [69], Helm and Marriott proposed use of constraints in formal specification of visual language syntax. This led to work on automatic parsing of visual languages [70], elaboration of a grammatical formalism [92] which is discussed further in the next

chapter, and most recently to the development of a visual language parser generator [17]. Most of Helm and Marriott's implementations have been done in CLP(R).

Further developments in logic programming will undoubtedly be significant for interactive notation processing. For example, a key problem in notation parsing is to deal with partial information and ambiguity, which may be easier to express in terms of modal logic rather than ordinary first-order logic. Scherl [52] proposes methods for automated modal theorem proving, which might form the basis for future modal logic programming systems.

2.5 The Interactive Programming Environment Perspective

I envisage interactive notation processing systems comprised of

- a *visual editor* which allows the user to manipulate a notation as a 2-dimensional arrangement of symbols
- a *parser* which infers the presence of higher-level structures bearing meaning, and
- one or more systems for *interpreting* that meaning.

This general design has much in common with an interactive programming environment, and hence the approaches taken in the design of such environments constitute another perspective on the problem of designing notation processors.

Traditional programming environments consisting of a text editor, compiler and runtime system (and/or interactive symbolic debugger) lead by their very nature to the familiar "edit-compile-run" development cycle. If any phase of compilation or execution runs into trouble, one cannot simply fix the mistake in the program and continue processing from where it left off; it is always necessary to begin the entire cycle again. Development systems based around an interpreter (e.g. LISP and Prolog systems) avoid the problem by eliminating the compilation phase entirely, but it should be noted that little production code is written for purely interpretive environments; to ensure fast program execution there is usually some kind of preprocessing (compilation) step.

The inefficiency—in terms of CPU time and users' time—inherent in the edit-(compile)-run cycle led in the late 1970's to an interest in more tightly-coupled program development systems. The essential argument was as follows: Most iterations

of the cycle involve small editing changes, whose effects are highly local. Hence by applying suitable caching techniques, it should be possible to avoid re-doing the majority of the work whose outcome would not change. (See [86] for a more detailed discussion of the issues and early developments.)

Much of the work done on "integrated programming environments" was aimed at caching intermediate parsing results. Two basic approaches were tried. In the *syntax directed editing* approach epitomized by the Cornell Program Synthesizer (see [140, 62], and also [94], and [14]), the editor's data structure was made to reflect the parse tree of the program being edited, and editing commands operated on syntactic units. In the less common *textually oriented program editing* approach advocated by Lee and some others (see [86]), the editor operated primarily on text but also maintained a parse tree, and attempted to maintain consistency between the two data structures at all times.

As one might guess, the syntax-directed editing approach was much easier to implement than the textually-oriented approach and hence was more successful. Eventually, syntax directed editor generator tools (similar to compiler generators) were developed [116]. Associated with the syntax-directed editing approach was the notion of *top-down refinement* proposed in 1981 by Denning [31], by which new program text was entered using a template-filling strategy. This obviated the need for a conventional parser, because the sequence of the user's input commands essentially "spelled out" the derivation of the new program material from its highest-level nonterminal. However it also made entry of common program structures such as arithmetic expressions unacceptably complex [147]; this problem was addressed in later systems by allowing small syntactic units to be selected, edited as text, and re-parsed [143].

The apparent success of both approaches led to suggestions that notations might be handled similarly. Göttler [61] implemented some syntax directed notation editors using top-down refinement as the input strategy. Wittenburg *et al.* implemented simple editors for mathematical expressions and flowcharts, using an incremental parsing strategy more closely related to the textually-oriented editing approach [154], and more recently has applied the same strategy in the domain of interactive document design [149].

The notion of integrated programming environments appears to have remained an intellectual exercise; neither of the two main approaches has (to my knowledge) been adopted in any commercial product of significance. The work of Göttler and

Wittenburg suggests, however, that conceptually similar approaches may yet be appropriate for interactive notation processing. Integrated editor/parsers for programming seemed like a good idea in contrast to the tedious batch compilation methods in use at the time; today much of the tedium has been eliminated by faster CPU's, separate compilation of small program units, and integrated development systems where the compiler gives up at the first syntax error and instantly reactivates the editor with the cursor near the error. Results obtained by Marriott [92] (see next chapter), suggest that notation parsing is likely to have much worse computational complexity—probably exponential in the size of the input—than program parsing, meaning that incremental methods will remain preferable even as machine performance improves.

2.6 The Visual Languages Perspective

The notion of graphical, or visual, programming—using pictures instead of text to represent programs, ideally with greater clarity and ease of learning—has a long history. In a review of the field, Myers [98] mentions a visual programming system developed at MIT's Lincoln Laboratory described in a 1963 Ph.D. thesis. For overviews of the many developments since then, see [15, 98, 99, 133]; most of the seminal papers in the field are collected in [54].

Sometime around 1990 the term "visual languages" came into vogue, as a general descriptive label covering graphical programming and a variety of related research areas. There is now a *Journal of Visual Languages and Computing* (JVLC, published by Academic Press since 1990) and annual *Workshops on Visual Languages* organized by the IEEE Computer Society. Despite the fact that there appears to be no precise definition of the term "visual language", there is quite definitely a community of researchers who identify with it, and who read and publish in the JVLC and the IEEE Workshop proceedings. The opinions and efforts of these researchers collectively constitute a very important perspective on notation processing.

The field now called "visual languages" is still defining itself, as evidenced by the large number of "taxonomies" which have been published to date [15, 98, 99, 133]. The primary goals of the field appear to be:

- *Visual programming*, the idea of interpreting pictures as computer programs.
- *Program visualization*, the use of automatically-generated graphics (sometimes

with animation) to enhance human understanding of program (or algorithm, or data) structure and function.

- *Programming by example* (PBE), the notion that programming *per se* might be avoided if computers were capable of abstracting functional, etc. specifications from user-supplied examples, often expressed graphically.

Work in graphical programming is relevant to interactive notation processing, because graphical programming environments *are* interactive notation processing applications. The other two areas are beyond the scope of my present work.

The inherent assumption in visual programming is that programs might usefully be represented pictorially rather than as text. Although outsiders have expressed considerable skepticism (see [114]) about how realistic this notion is, it remains a dominant theme in research. Shu [133] observes that while traditional programming languages will probably remain the norm for "...large software systems on the serial machines where the program efficiency is a primary concern", visual methods might be more appropriate for end-users who are not professional programmers. Myers [99], pointing to the success of spreadsheets, argues that given a less imposing paradigm, non-programmers can and do create nontrivial "programs", and suggests that pictorial paradigms may be similarly successful.

One of the first visual programming environments was Pascal/HSD, which was an attempt to express Pascal control structures such as if-then-else, while, etc. in diagram form. Other systems in the same vein include Pict [55] and Iconic PICL [32]. This approach has met with limited success, and researchers have begun to propose new control structures which are as powerful as the conventional ones but which are more amenable to visual representation. Examples include the languages PROGRAPH [30], which has become commercially available for Macintosh systems, and Hyperflow [81], a research system designed specifically for pen-based computers.

Pictorial representations are especially attractive in applications where there is inherent concurrency or parallelism. Reader [115] describes a system for functional programming, which is naturally expressed by directed acyclic graph-structured diagrams with nodes representing functions and arcs representing data streams, and where there is a natural mapping onto data-flow type parallel processing architectures. In 1990, I designed a graphical programming system for specifying non-trivial arrangements of communicating processes under the UNIX operating system, expressed pictorially using boxes (representing processes) connected by arrows (representing data flow).

The system was implemented in two consecutive senior undergraduate programming projects [9, 137].

Although descriptions of complete research systems are still common in the visual language (VL) literature, since about 1990 there has been a trend toward papers focussing on particular sub-problems, such as:

- Visual editing, and graphical user interfaces in general [22, 56]
- Visual Language theory: language formalisms, and theoretical and algorithmic results obtained using them. Sub-areas include
  - preprocessing [90] and lexical processing [21]
  - grammar formalisms [110, 95]
  - parsing algorithms [110, 70, 151]
  - data structures to support efficient parsing [71, 4]
- design of new visual languages [114]
- automatic layout of computer-generated diagrams [91]

Of greatest relevance to interactive notation processing are works on grammar formalisms and parsing algorithms, which are discussed in detail in the next chapter.

2.7 The Human-Computer Interaction Perspective

Primary concerns in the field of human-computer interaction (HCI) include:

1. ergonomics, i.e., approaches to design of interactive computer systems informed by understanding of human cognitive and motor capabilities.
2. user interface management systems and their standardization.
3. interaction techniques, especially graphical techniques e.g. menus, direct manipulation, etc.

The first two are not relevant to interactive notation processing, except in the general sense that research results in these areas have become identified with good practice in the design and implementation of interactive systems.

Use of syntactic techniques for preprocessing of graphical input was proposed in 1984 by Hamlin [65], but appears to have met with little interest. Vlissides and

Linton [144] describe a general approach to building graphical editors in which the user's manipulations of views are interpreted appropriately as modifications of some underlying "domain-specific" data structure. Their approach, though not syntactic in nature, would be applicable to the editing component of notation processors.

The advent of pen-based computer systems has prompted the development of pen-based operating systems [13], renewed interest in handwriting recognition [139], and "gestural" input techniques where specific pen movements are interpreted as commands [148]. (Weber's approach [148] uses syntactic techniques, based on interpretation of input gestures as strings.)

Within the HCI field much work has been done on the design and implementation of visual editors, including editors for graphics. Van der Vegt [142] proposes a user interface for editing hierarchically structured drawings, wherein one's current position in the hierarchy is always indicated via highlighting of all objects at or below the current node; In Chapter 4 I describe how similar methods might be used in notation processors.

Kurlander and Feiner [84] describe search and replace techniques for graphical editors where both the search specification and some aspects of the replacement are specified graphically, using constraints. Although such techniques might profitably be applied in interactive notation editors, I suspect that methods based on notation semantics would ultimately be more useful. For example, in a music notation editor, rather than searching for a specified arrangement of musical symbols, it would likely be preferable to search for a given harmonic or rhythmic structure, perhaps specified via a user-supplied example notation.

2.8 Discussion and Summary

Notation-oriented projects reported in the pattern-recognition literature have tended to be ambitious but also to fall far short of the ultimate goal. The focus of effort in these projects has usually been preprocessing and segmentation, with little time left over for higher-level structure analysis. I have attempted to avoid this pitfall by concentrating entirely on high-level analysis, and hence I have found the pattern-recognition literature interesting but not directly relevant.

On the other hand, the pattern-recognition approach emphasizes noise-tolerance and general robustness, as well as automatic inference. These aspects are beyond the scope of this dissertation, but would be extremely important for future work in refinement and implementation of the designs proposed here.

Somewhat surprisingly, the language-theory literature contains few proposed formalisms for language structures other than strings. Essentially only two other structures have been studied: arrays and hypergraphs (with trees and ordinary graphs as special cases). These appear to have been chosen because their properties were well understood, rather than because they were particularly well-suited for modelling e.g. notations.

I contend that the hardest part of applying syntactic methods to notation processing is not the development of appropriate grammars, but rather the choice of a truly appropriate *representation* upon which formalisms such as grammars can be based. One of the major contributions of this thesis is the attributed set formalism proposed in Chapter 4, which I believe to be appropriate for modelling notations.

The string encodings discussed above in the section on the language theoretic perspective, are rather clumsy as representations of two-dimensional structures. Attributed encodings, on the other hand, are quite appealing in that attributes provide a means to capture metric parameters such as line segment lengths. Plex structures and hypergraph representations are also interesting, in that they represent symbols and their relationships separately. The attributed set formalism proposed in Chapter 4 also distinguishes symbols and their relationships.

I have suggested that notations are a kind of natural language, and hence that syntactic analysis of notations will bear a greater resemblance to natural language understanding (NLU) than to parsing of, say, programming languages. Kent Wittenburg evidently thought so too, when he chose to apply NLU tools such as unification grammars to notation parsing (see next chapter). The AI approach of constraint-

based programming is also highly applicable to notations; augmenting an attributed representation with constraints provides a natural way to represent geometric relationships. All of the approaches discussed in Chapter 3 make use of attributes and constraints.

Combining definite clause grammars with constraint logic programming essentially yields the approach of Marriott (see next chapter).

New methods for reasoning with partial and/or uncertain information, now being developed in AI, while beyond the scope of this dissertation, will definitely be useful in the refinement and implementation of the designs proposed here.

The generic design I propose for interactive notation processing systems, discussed in Chapter 6, has much in common with interactive programming environments (IPE). The big debate in IPE during the 1980's concerned syntax-directed editing *vs.* textually-oriented editing. The former is easier to implement but harder to use, the latter is much harder to implement but potentially easier to use. I believe that the arguments in favour of the textually-oriented program editing approach, based on the cognitive overhead inherent in the syntax-directed approach, remain as strong as ever, and hence a similar approach should be pursued for notations. A *graphically-oriented notation editor* would operate on a database of graphical primitives, such as geometric primitives or strokes in a pen-based system, and infer the presence of higher-level structures automatically by incremental parsing.

The field formerly called "graphical programming" has undergone a gradual change of name, first to "visual programming" and now to "visual languages". This reflects not only a broadening of the field beyond programming into areas such as VL design, but also the gradual adoption of a view of visual interaction as a language-processing problem. The VL field is still to some degree in the same state as syntactic pattern recognition—numerous grammar formalisms and associated parsing algorithms have been proposed, but the question of appropriate underlying representations has received little attention. Much attention has been devoted to parsing efficiency, which I believe is premature given the lack of good representations.

My own priorities have been different. I have devoted considerable effort to elaborating what I consider to be an appropriate representation for notation structures, and much less to issues of parsing efficiency. In interactive environments, where parsing is performed in arbitrarily small increments, efficiency is unimportant. In fairness, however, I must note that most work in the VL field deals with artificial languages,

e.g. visual programming languages, rather than notations which predate computerization. The proposed representations may indeed be adequate in such cases, and efficient parsing (or computation of any kind) is always a worthy goal.

Computer-human interaction research has produced a steady stream of results which have, sooner or later, become synonymous with good practice in design and implementation of interactive systems. Interactive notation processing systems will be no exception.

Chapter 3

Five Approaches Reviewed in Detail

The previous chapter presented a general discussion of previous work related to the problem of notation processing. In this chapter I focus on five specific research efforts which are similar enough to my own that a more detailed review is called for.

3.1 Context-Free Grammars, Derivation, Parsing

All of the formalisms discussed in this chapter are extensions of the well-known context-free grammar formalism. To establish a vocabulary for the discussion to follow, it is necessary to recall some definitions from formal language theory, and in some cases to choose specific terms among several in common usage.

> *NOTE: Some of the terms used in this chapter, e.g. "sentence" and "string", are used or defined in ways inconsistent with convention. This has been done in order to be able to provide a consistent terminology with which to describe substantially differing formalisms.*

An *alphabet* is a finite, nonempty set, the elements of which are usually called *symbols.* In the simplest formalisms, symbols are atomic entities; in more elaborate ones, they may have associated *attributes.* A *language* is a set of *sentences*; For the purposes of this chapter I define a "sentence" very broadly as a collection of symbols. In traditional formal language theory, sentences are *sequences* of symbols; I call these *strings.* In most of the formalisms discussed in this chapter, sentences are unordered collections of some kind, such as sets or *multisets.* A multiset is an unordered collection of elements whose elements need not be distinct.[1]

[1] A pocketful of coins is a good example of a multiset. Multiset construction never "collapses" the way set construction can, i.e., if A and B are multisets, we can be sure that $|A \cup B| = |A| + |B|$, whereas if they are sets all we can say is $|A \cup B| \leq |A| + |B|$.

A *context-free grammar* (CFG) is a formalism initially devised for describing languages of strings, but which has been extended in ways described below to deal with set and multiset languages as well. A CFG has the form (N, T, P, s) where N and T are alphabets (called the *nonterminal* and *terminal* alphabets respectively), $s \in N$ is called the *start symbol*, and P is a set of *productions* having the form $X_0 \rightarrow X_1 \ldots X_n$ where $X_0 \in N$, $X_1 \ldots X_n \in N \cup T$; X_0 is called the *head* of the production, $X_1 \ldots X_n$ the *body*.[2] A sentence (string, set, multiset) consisting only of terminal symbols is called a *terminal sentence*.

Productions are templates which indicate how parts of sentences may be rewritten. Every formalism has its own definition of how these templates are *matched*, but generally each symbol X_i in the production is matched with one symbol in the sentence under consideration. Rewriting according to a production is called *applying* the production, and may be done in one of two ways. If one symbol is matched with the head of the production and rewritten as a group of symbols which match with those in the body, this is called a *derivation* step. If a group of symbols are matched with the body and rewritten by a single symbol which matches the head, this is called a *reduction* step.

CFG derivations can always be represented as trees, since every derivation (reduction) step expands (reduces to) a single symbol; these are called *parse trees*. Trees, however, are not sufficient to capture the structure of certain objects, directed graphs being the most important example. Hence some of the formalisms discussed in this chapter use representations more complex than trees for parse structures; the most common is a directed acyclic graph.

The *language of a CFG* $G = (N, T, P, s)$, denoted $L(G)$, is that set of terminal strings obtainable by performing a sequence of valid derivation steps starting with the one-letter string consisting only of the start symbol s. The sequence which yields a given string w is called the *derivation of* w *under* G. By generalizing from a one-letter string to a single-symbol sentence in a manner appropriate to each CFG-related formalism—let us call this a *start sentence for* G—we can generalize the notions of the language of a grammar and derivation of a sentence.

Given an element S of some set L, we might in general ask whether or not $S \in L$; this is called the *membership problem* for L. More importantly, we might ask whether

[2]It is common to say "left side" and "right side", but this can be confusing because in some cases production are written the other way around e.g. $X_1 \ldots X_n \rightarrow X_0$.

the membership problem is decidable in general for sets $L = L(G)$ defined by (some type of) grammar G. For CFG's the membership problem is decidable and the result may be computed in polynomial time. Marriott ([92], see section 3.5 below) provides interesting results concerning more general grammar formalisms.

Given a sentence $S \in L(G)$, one might perform a sequence of reduction steps which is exactly the reverse of the derivation of S under G, to obtain a start sentence for G. Based on this observation we can imagine an algorithm which takes a terminal sentence S as input, applies reductions in a systematic manner, and outputs "yes" if $S \in L(G)$ and "no" otherwise. Such an algorithm is called a *recognition algorithm* for $L(G)$. A recognition algorithm which, in the "yes" case, reports the derivation of S is called a *parsing algorithm* or *parser*.

Work on the theoretical basis of compiler design has yielded some useful extensions of the CFG formalism [1]. In an *attribute grammar* [28], each grammar symbol has an associated set of *attributes*, and each production has an associated set of *semantic rules* of the form $b \leftarrow f(c_1, \ldots, c_k)$, indicating how an attribute b of some symbol represented in the production is computed as a function of attributes $c_1, \ldots, c_k$ of symbols represented in the production. If b is an attribute of the head symbol and the c_i are attributes of the body symbols, b is called a *synthesized attribute*. If b is an attribute of one of the body symbols, it is called an *inherited attribute*.[3] In compilers, attributes are useful for semantic processing (e.g. code generation) and to allow for slight context-dependencies (e.g. verifying that variables are declared before being used). In systems for processing notations, attributes are desirable because notations themselves are parametrized in terms of attributes relating to graphical variables such as position, size and orientation.

Even more powerful grammar formalisms have been developed in the course of work in natural language understanding (see section 2.4). When the assignments $b \leftarrow f(c_1, \ldots, c_k)$ in an attribute grammar's semantic rules are replaced by equational constraints $b = f(c_1, \ldots, c_k)$, and the matching process involved in applying productions is changed from simple variable binding to *unification*, the result is a *unification grammar* [132]. A unification grammar expressed in the definite clause subset of first-order logic is called a *definite clause grammar* (DCG) [108]. DCGs are of particular interest because they can be interpreted as logic programs; a DCG

[3] In [1], a related formalism called *syntax-directed definition* is defined. A syntax-directed definition is like an attribute grammar, but the "functions" f in semantic rules need not be pure functions—they may have side effects.

interpreted as a logic program is a recognition algorithm for the language it describes.

CFGs, attribute grammars, and unification grammars are all string-generating grammar formalisms. The formalisms discussed below, with the exception of the first one, are attempts to adapt ideas from these formalisms to the generation of set or multiset languages. All make use of attributes. In considering each proposed formalism, I have attempted to answer the following questions:

1. *Representation:* What is the structure of sentences?
   (a) What form do symbols take?
   (b) What kinds (types) of attributes are used?
   (c) What auxiliary structures (if any) are used, e.g. relations?
   (d) How are sentences formed, e.g. sets or multisets?
   (e) How are sentences used to encode the structures of notations?
2. *Matching:* How are productions matched with parts of sentences?
   (a) How are attributes matched, e.g. simple binding of values to symbolic variables, unification of expressions, etc.?
   (b) When expressions are involved, are there any limits on their power?
3. *Parse structure:* Can derivations be represented as trees or is something more general (e.g. directed acyclic graph) required?
4. *Expressive power:* Are there languages which the formalism cannot describe?
5. *Symmetry:* Can the formalism be used for both derivation and parsing?
6. *Rigour:* When examples are given, there is an implicit claim that a given grammar G defines a given language L. Are both G and L adequately defined, and is it proved that $L(G) = L$?

3.2 Kojima and Myers: Graphic Functional Grammars

The work of Keiji Kojima and Brad Myers [83] serves as strong evidence that sequential representations are not adequate for notations. They begin by observing that in modern computerized document-processing systems, documents containing picture structures such as charts, tables, and diagrams are represented as sequences

of procedure calls in languages such as PostScript. This suggests that it might be possible for a computer to analyze such sequences so as to re-generate a high-level specification of the meaning of pictures. To this end, Kojima and Myers devised a grammar formalism called *graphic functional grammars* (GFG) wherein sentences are sequences of procedure calls.

The GFG formalism is based on the following definitions. A *graphic function* is a procedure whose execution "draws some geometrical figures in a multi-dimensional space" [83, page 111]. A *graphic vocabulary* (GV) is a finite set of graphic functions. A *graphic sentence* is a finite sequence of graphic function calls.[4] A GFG is a tuple $G = (V_N, V_T, S, P)$ where V_N is a GV of nonterminal functions, V_T is a GV of terminal functions, S is a start function, and P is a set of "production rules". Each production rule has the form

$$\begin{aligned} \alpha(x_1, \ldots, x_p) \quad \rightarrow \quad & \beta_1(y_{11}, \ldots, y_{1q}), \ldots, \beta_n(y_{n1}, \ldots, y_{nq}) \\ & \mid \gamma(z_1, \ldots, z_r). \end{aligned}$$

α is the head symbol, the β_i are the body symbols; these are of course procedures. The x_i and y_{ij} denote expressions corresponding to the formal parameters. $\gamma(z_1, \ldots, z_r)$ represents "...constraints which are denoted as a formula of first-order predicate logic" [83, page 112]. The process of matching rules to graphic sentences is not entirely clear; the initial definition suggests a simple binding of formal to actual parameters, but later unification is mentioned. In a GFG derivation step, a procedure call matching the rule head can be rewritten as a sequence of calls matching the rule body, provided the constraint $\gamma(z_1, \ldots, z_r)$ is satisfied under the involved binding (unification). The language $L(G)$ of a GFG G is that set of terminal graphic sentences (graphic sentences consisting only of calls to functions in the GV V_T) obtained by "...repeatedly applying rules to the start function" [83, page 112]; based on examples given this seems to mean applying rules to a *call* of the start function, with associated actual parameters.

The fact that GFG derivations proceed from a call of the start symbol with actual parameters, together with the fact that the formalism appears to permit expressions of entirely arbitrary power in rules, allows a GFG for directed graph diagrams to be

[4] "...a finite sequence of graphic functions...where actual parameters are assigned to their formal parameters." [83, page 112].

written very simply:

$$
\begin{aligned}
V_N &= \{\,\mathrm{graph}(p), \mathrm{node}(p), \mathrm{arc}(p)\,\} \\
V_T &= \{\,\mathrm{circle}(x,y), \mathrm{arrow}(x_s, y_s, x_e, y_e)\} \\
S &= \mathrm{graph}(p) \\
P &= \{\,\mathrm{graph}(p) \rightarrow \mathrm{node}(p)\,\mathrm{arc}(p), \\
&\quad\ \mathrm{node}(p) \rightarrow \mathrm{circle}(x,y)\,\mathrm{node}(p') \mid p = p' \cup \{(x,y)\}, \\
&\quad\ \mathrm{node}(p) \rightarrow \epsilon \mid p = \emptyset, \\
&\quad\ \mathrm{arc}(p) \rightarrow \mathrm{arrow}(x_s, y_s, x_y, y_e)\,\mathrm{arc}(p) \mid p \supseteq \{(x_s, y_s), (x_e, y_e)\}, \\
&\quad\ \mathrm{arc}(p) \rightarrow \epsilon \quad \}
\end{aligned}
$$

Let us call this GFG $G_{\mathbf{graph}}$. The trick which allows $G_{\mathbf{graph}}$ to generate graph structures is, clearly, the parameter p whose value is a set of points in the plane (the centres of circles representing graph nodes) and the constraints involving set union and containment operators. Derivation of a graphic sentence representing a graph diagram proceeds from a start sentence consisting of a single call to the start function *graph*, whose p parameter must already be assigned an appropriate value. (If the graph is to have, say, four vertices, p must be a set of four points in the plane, which furthermore should be chosen so that the generated diagram will have a suitable layout.)

It is easily verified that $G_{\mathbf{graph}}$ can generate any graph, that is, that any graph diagram can be encoded as a sequence of calls to the terminal functions *circle* and *arrow*, which is in the language of that GFG. Kojima and Myers note, however, that $G_{\mathbf{graph}}$ generates only sentences in which all the *circle* calls precede all the *arrow* calls, but it is conceivable that in the output of a computerized publishing system, the calls might appear in any order. They go on to make the following observations:

- Reformulating a GFG such that its language is closed under permutation is perhaps possible but is difficult even for a small example such as $G_{\mathbf{graph}}$.
- One could transform input sentences to acceptable forms before parsing. (For $G_{\mathbf{graph}}$ this would be trivial.)
- Not all graphic languages are such that all permutations of a graphic sentence give rise to the same graphic output. They call languages which do have this property *commutative*. Non-commutativity is due to *side effects* situations where the execution of one function changes a global variable, which affects the execution of functions called subsequently.

- Commutative languages could be parsed using a generate-and-test method, generating all permutations of an input sentence and passing each to the parser, but this would naturally yield exponential time complexity. An alternative would be a parsing method which is order-independent.

They prove that every graphic language can be effectively translated to an equivalent commutative language; the practical usefulness of this proof, however, is limited.[5] They also give a GFG parsing algorithm for commutative languages which, by using constraints in production rules to determine the order in which procedure calls are processed, is capable of accepting input in any order.[6]

Kojima and Myers started with a sequence-based formalism and ended up devising a parsing algorithm which essentially ignores input order. Their experience illustrates convincingly that sequence-based representations are not appropriate for graphical structures.

---

[5]The method used is analogous to the "Z-buffer" technique used in computer graphics, wherein surface primitives defined in 3-dimensional space can be rendered in any order to a 2-dimensional pixel array, with occlusion dealt with on a per-pixel basis by associating a *depth* variable with every pixel, and permitting pixel values to be changed only if the new primitive being rendered has a lesser depth than has so far been recorded for that pixel. Whenever a pixel is actually modified, the corresponding depth variable is also updated. The algorithm given by Kojima and Myers for transforming a sequence S of procedure calls to an equivalent set of procedure calls whose execution—in any order—will generate the same output, involves the addition of one guard variable to every global variable and the encoding, in new actual parameters added to every procedure call, the ordinal position of the corresponding procedure call in the input sequence and *the values of every global variable expected when that call is about to be executed* in the input sequence. The procedure definitions are modified to compare their ordinal position argument against the current value of the guard variable for each global, modifying the global (and updating the guard) only if the current guard value is less than the ordinal value. This approach will clearly only be practical when the number of bits required for all of the global variables involved is small, but in raster graphics systems the entire frame buffer is global. Kojima and Myers give an interesting example, however, where instead of encoding the initial values of all globals as actual parameters, the required values are computed when needed by variants of the basic primitive-drawing functions.

[6]This algorithm, which involves unification, constraint solving, and backtracking, is difficult to analyze. The algorithms given by Wittenburg, Golin and Marriott (see below in this chapter) are more carefully presented.

3.3 Wittenburg, Weitzman and Talley: Relational Languages

The approach to notation processing defined by Kent Wittenburg and his collaborators Louis Weitzman and Jim Talley, has come to be called *Relational Languages* or RL [152, 151]. The RL approach was developed initially for natural language processing, and then applied in the course of research on automatic recognition of hand-sketched graphic input (i.e., interactive notation processing) at the Microelectronics and Computer Technology Corporation in Austin, Texas. (See [153] for a discussion of the history and motivations of the approach.) The focus of these research efforts has been on parsing, and perhaps more specifically on parsing efficiency. In [154], Wittenburg states that the utility of the RL grammar formalism for language generation remains an open question.

An RL is a set of *RL expressions*, each of which is a set S of objects which may have associated attributes plus a set of n-ary relations on the powerset of S.[7] For example, the character string "abba" could be modelled in the RL formalism as the set $S = \{c_1(a), c_2(b), c_3(b), c_4(a)\}$ and the relation left-of $= \{(\{c_1\}, \{c_2\}), (\{c_2\}, \{c_3\}), (\{c_3\}, \{c_4\})\}$. The elements of S in this case represent the individual characters of the string; each carries an attribute indicating the letter which actually appears in the corresponding character position.

By explicitly specifying relations among objects, RL expressions can capture quite complex structures. [154] gives examples including flow charts and positional mathematical notation, which rely on relations such as "left-of", "above", "centered-in-X" etc. RL expressions are hence a generalization of strings, which encode only one kind of relation (left-to-right concatenation), encoded implicitly via juxtaposition of symbols.

The RL approach is based on the PATR-II unification grammar formalism proposed by Shieber [132]. Unification grammars were developed within the field of computational linguistics to describe and characterize natural languages. Briefly, a unification grammar is like a context-free grammar but

1. symbols have associated attributes called features, each of which has acts like a pointer to a value which may be blank (i.e., variable), a constant (e.g. "noun phrase", "singular", etc.) or another set of features (called a *feature structure*).

[7] The papers by Wittenburg *et al.* do not actually give an explicit mathematical definition of the RL formalism. I have synthesized the definitions given here based on what I have read.

2. the feature structures associated with terminal symbols are listed explicitly; the list is called the *lexicon* of the grammar.

3. when a production, called a *rule*, is applied (either in parsing or generation), the matching of feature structures is done via *unification*, a bi-directional substitution process much like term unification in Prolog.

4. productions are augmented with a set of *constraints* which usually declare pairs of features equivalent.

The details of unification grammars are intricate and beyond the scope of this dissertation. See [132] for more information.

Shieber's PATR-II formalism describes languages of strings, and hence the order of symbols in the body of rules is significant, indicating the order of concatenation of the related language elements. The only kind of constraint allowed is equation of two features. Wittenburg has proposed a formalism called F-PATR [150], which extends Shieber's PATR-II formalism in two ways:

1. The symbols in the body of a rule are treated as a set rather than as a sequence; their order is thus irrelevant.

2. The unification method is extended to allow use of functional expressions in constraints, following the approach of Ait-Kaci and Nasr [2]. Functional constraints may involve variables, and may take one of two forms:

   (a) A variable may be set equal to any functional expression.

   (b) Functional expressions which return a Boolean result may be used directly as constraints.

   The former are used to propagate and transform feature values, as in attribute grammars [28]. The latter are used to set conditions on successful unification.

Following is an example of a rule in an F-PATR grammar for mathematical notation, somewhat simplified from [154].

Example 1: Vertical infix division

The rule which defines fraction structures consisting of two formulae, one above the other, separated by a horizontal line, begins with a context-free basis production. The nonterminal (called the rule *head*) appears on the right side rather than the left as is customary for ordinary context-free grammars; the expansion of that nonterminal (called the rule *body*) appears on the left.

HorizLine Numerator Denominator → Fraction

This is followed by several constraints. The first group ensures that the symbols involved have the correct syntactic categories.

⟨HorizLine syntax⟩ = vert-infix-divide

⟨Numerator syntax⟩ = formula

⟨Denominator syntax⟩ = formula

⟨Fraction syntax⟩ = formula

The expressions in angle brackets are called *paths*, and are derived from the PATR-II formalism. In these examples, the first entry identifies a symbol and the second names one of that symbol's features. (In cases where feature values are themselves feature structures, paths may contain more than two entries.)

The next constraint illustrates how semantics may be captured within the F-PATR formalism. Note that this is a type-a functional constraint (as defined above); Lisp syntax is used for functional expressions.

⟨Fraction semantics⟩ =

(divide ⟨Numerator semantics⟩ ⟨Denominator semantics⟩)

The next constraint defines the propagation of *cover* features. The cover of a grammar symbol is the subset of the objects in the RL expression for which that symbol stands. Again a type-a functional constraint is used.

⟨Fraction cover⟩ =

(union ⟨HorizLine cover⟩ ⟨Numerator cover⟩ ⟨Denominator cover⟩)

The next group of constraints are type-b. In parsing, these define conditions under which this production may be applied. The italicized relations *below* and *above* are distinguished as *expander relations*, discussed below.

(*below* ⟨HorizLine cover⟩ ⟨Numerator cover⟩)
(*above* ⟨HorizLine cover⟩ ⟨Denominator cover⟩)
(wider-than ⟨HorizLine cover⟩ ⟨Numerator cover⟩)
(wider-than ⟨HorizLine cover⟩ ⟨Denominator cover⟩)

In [154], Wittenburg *et al.* define a parsing algorithm for RL grammars, which is a variant of chart parsing (see [3]) a bottom-up technique normally used for parsing strings. A chart parser works by repeatedly looking for matches between substrings of the input and production bodies, keeps track of partial matches in a data structure called a chart, and repeatedly attempts to extend partial matches until all elements of the production body are matched, at which point it can infer the presence of the nonterminal in the head of the production. In notation parsing there is no predefined order in which to match the symbols in rule bodies; Wittenburg's approach solves this problem by requiring that certain binary type-b constraints called *expander relations* be present in all rules, which collectively define a total order on the symbols in the rule body. When the chart parser has matched the first symbol in this total order, it can use the expander relations to search for suitable candidates to match the second symbol, and so on.[8]

The chart parsing algorithm works bottom-up, processes input incrementally, and will return the correct result irrespective of the order in which it looks at the input symbols. These are important advantages for interactive parsing: parsing can begin as soon as the user enters the first symbol, there is no need to enter symbols in any particular order, and editing changes (inserting, deleting, and moving symbols) can be handled with ease.[9]

[8]The searches are performed via database operations, and can be highly efficient. The parsing algorithm is flexible enough to deal with cases where the search returns more than one candidate. The requirement on expander relations in rules ensures, however, that there will always be at least one.

[9]Insertion is trivial because the parser works incrementally—there is simply one more symbol in the database among many which the parser has not yet seen. Cover attributes, which are represented in the chart, play a key rôle in handling deletion—following deletion of a symbol, all active arcs containing that symbol in their cover are deleted from the chart. A move operation is implemented as deletion followed by insertion of the appropriately modified symbol.

In a recent paper [151], Wittenburg pursues the matter of parsing efficiency still further, defining a subclass of RL grammars called *fringe RL grammars* and shows how a predictive parsing algorithm based on the method of Earley [43] can be applied to these grammars. The efficiency issue is important because, if the parser has no clues as to how the input may be ordered, the se rch for the "next" input element matching a rule symbol can range over the entire input, and hence the time complexity of the parsing problem becomes exponential in the size of the input.

The key to the RL parsing approaches is the use of expander relations, which define an implicit order on input symbols. This allows algorithms developed for parsing strings to be adapted to non-sequential structures including notations and databases. This approach is a more sophisticated version of that discussed in section 2.3, wherein special operators are used to represent spatial relationships within a string representation of a two-dimensional structure. The difference is that Wittenburg *et al.* have broken entirely out of the string representation paradigm, choosing a set-based representation where objects and their relationships are represented separately and explicitly. Moreover, they have explored the nature of the relationships in detail, finding that certain of them (expander relations) can under the right conditions be used to "drive the parse" through a potentially exponential range of possible paths.

The RL approach also has limitations. Since derivations are represented as trees, directed graphs cannot be described by RL grammars. Since RL grammars have to date been used only for parsing, their generative properties have not been explored. Hence although Wittenburg *et al.* have presented a few examples of notations and corresponding RL grammars, they do not prove that the given grammar actually generates the correct language.

3.4 Golin and Reiss: Attributed Multiset Grammars

In his 1990 Ph.D. thesis, Eric Golin proposed a formalism called *attributed multiset grammar* (AMG), derived from conventional context-free grammars as follows:

- Symbols carry attributes, e.g. graphical coordinates.
- The body of each production is a multiset instead of a sequence.
- Each production is augmented with
  1. A *semantic function* which defines how the attributes of the head symbol are computed from the attributes of the symbols in the body.
  2. A *constraint* which is a Boolean function of the attributes of the body symbols. When actual symbols (bearing actual attribute values) are matched with the production body, a reduction step may be performed only if the constraint is satisfied.

Language elements are multisets of symbols bearing attributes which are normally real numbers or symbolic constants (e.g. "solid" for a line style attribute). Attributes are represented by symbolic identifiers (variables) in productions, and the matching process involves a consistent assignment of these variables to the actual attributes, much like formal/actual parameter assignment in a procedural programming language. The semantic function and constraint are somewhat analogous to functional constraints in the Wittenburg's RL formalism, but in an RL grammar there is no clear distinction made between constraints which serve to propagate attributes and those which set conditions on applicability of rewriting steps.

In [59, 60], Golin and Reiss define *picture layout grammars* (PLG) according to the following definitions:

- a *picture element* is a graphical primitive such as a line segment, text string, or "shape" (any complex graphical object such as an ellipse, characterized by its XY extent or bounding box).
- a *picture* is a set of picture elements arranged on a plane.
- a *visual language* is a set of pictures.
- a PLG is an AMG in which the productions correspond to picture composition operators.

Hence, a PLG describes the syntax of a visual language by defining how its pictures are logically composed from sub-pictures, down to the level of terminal symbols which correspond to primitive picture elements.

Golin discusses parsing with PLG's in his Ph.D. thesis and a subsequent paper [57]. In later work [58], he introduces a closely related formalism called object oriented PLG's (OOPLG) involving notions such as classes, inheritance and methods taken from object-oriented programming, and describes a visual language compiler generator which, given an OOPLG specification, generates a C++ program which parses the corresponding visual language by applying his parsing algorithm. These are very significant achievements, but the essential aspects of Golin's approach are contained within the earlier and simpler PLG formalism.

In a PLG, every grammar symbol has four attributes lx, by, rx, ty. For line segments these denote the endpoints (lx, by) and (rx, ty); for other shapes they denote the X-extent $[lx, rx]$ and Y-extent $[by, ty]$. Writing productions is simplified by a macro facility called *production operators*, which automatically generate the semantic function and constraint according to one of many standard templates. For example, the short form $A \rightarrow \text{over}(B, C)$ describes a situation where structure B is situated atop structure C, and may be reduced to a single structure A; it generates the production $A \rightarrow BC$, a constraint which ensures that the extent of B is centred above and does not overlap the extent of C, and a semantic function which defines the extent of A as encompassing the extents of B and C.

The AMG and PLG formalisms also include one very important feature, added specifically to permit description of directed graph structures. Symbols in the body of any production may be marked as *remote*, meaning that during parsing, they must match with some symbol in the parse tree, but that "remote" symbol is not actually rewritten in the course of a reduction step. The following pair of productions therefore suffice to describe directed graphs:

1. ARC → points-to (LABELLED-ARROW, <u>NODE</u>)

2. NODE → points-from (ARC, NODE)

The underlining in the first production indicates a remote symbol. Labelled arrows are first rewritten according to the first production as LABELLED-ARROW nonterminals. The production specifies that each arrow must point to some NODE, but since this is flagged remote, it is not actually rewritten in the reduction. Once

LABELLED-ARROWs have been reduced to ARCs in this way, they may be further reduced along with the NODE from which they emanate by production 2, which rewrites the NODE as itself; eventually a single NODE remains after all ARCs have been reduced.

Reductions involving remote symbols add non-tree arcs to the parse representation; the AMG/PLG parse structure is a directed acyclic graph. I used essentially the same technique in my early experiments with implementation of visual language parsers, described in [41] and also in chapter 5 of this dissertation. The remote symbol technique adds significant expressive power to the formalism, but makes it less amenable to analysis.

Golin's papers do not formally define the language of an AMG/PLG, and hence there are no proofs that a given grammar indeed generates or recognizes a given language. Unlike Wittenburg, Golin does not discuss the issues of parsing incrementally; his parsing algorithm operates as a batch procedure only.

3.5 Helm and Marriott: Constraint Multiset Grammars

The work of Richard Helm and Kim Marriott of IBM T. J. Watson Research Center and Monash University constitutes probably the strongest single influence on my own research. Their approach, now called *constraint multiset grammars* (CMG) was first described in a 1991 paper [70], which gave examples of the CMG formalism but unfortunately no details of the elegant mathematical development underlying it. The details given below are taken from two 1994 papers [92, 17] sent to me by Dr. Marriott.

The CMG formalism is in many ways a refinement of Golin's AMG. That is, a CMG is a context-free grammar where symbols have attributes, production bodies are multisets, constraints specify the relationships among attributes of symbols in productions, and productions may refer to symbols which they do not actually rewrite. The major difference between the CMG and AMG approaches is that the former rests on a firmer mathematical foundation, based on the semantics of constraint logic.

In the CMG formalism, grammar symbols are typed. Each type defines the number, names and types of attributes for symbols of that type. The term *symbol* is actually reserved to refer to symbols in productions, which as in AMG have variable attributes. Instances of a given symbol type, with specific values assigned to attributes, are called *tokens*. The language of a CMG is a set of *sentences*, each of which is a multiset of tokens.

CMG productions have the form

$$\begin{aligned} T(\vec{x}) \;\leftarrow\; & T_1(\vec{x}_1)\ldots T_n(\vec{x}_n) \\ & \textbf{where exist } T'_1(\vec{x}'_1)\ldots T'_m(\vec{x}'_m) \\ & \textbf{where } C \textbf{ and } \vec{x} = F. \end{aligned}$$

The ellipses (...) represent multisets. The first line is the underlying context-free production; the T_i are symbol types, the $\vec{x}_i$ are symbolic references to vectors of attributes. The T'_i are called *existentially quantified symbols;* these must be matched against tokens, but they are not actually rewritten when the production is applied. These are analogous to remote symbols in Golin's formalism. C denotes a conjunction of constraints over $\vec{x}_1, \ldots, \vec{x}_n, \vec{x}'_1, \ldots, \vec{x}'_m$ and F denotes a function of $\vec{x}_1, \ldots, \vec{x}_n$ and $\vec{x}'_1, \ldots, \vec{x}'_m$ Note that the expression $\vec{x} = F$, which is analogous to the Golin's semantic function, is actually just one more constraint. (The semantics of the equal sign is assertion of equality as in algebra, not one-way assignment as in a procedural programming language such as C.)

Productions are used to rewrite sub-multisets of multisets of tokens in derivation or recognition. Matching of symbols to tokens is done as in AMG, by constructing an appropriate (partial) assignment from the variables $\vec{x}, \vec{x}_1, \ldots, \vec{x}_n, \vec{x}'_1, \ldots, \vec{x}'_m$ to the actual attribute values in the tokens. It is implicit in the formalism that distinct symbols in a production are always matched with distinct tokens. Hence a production like

$$\begin{aligned} \mathrm{arc}(p,q) \;\leftarrow\; & \mathrm{arrow}(p_1,p_2) \\ & \text{where exist } \mathrm{state}(p'_1), \mathrm{state}(p'_2) \\ & \text{where } p_1 = p'_1 \wedge p_2 = p'_2 \text{ and } (p,q) = (p_1,p_2) \end{aligned}$$

in a CMG for finite-state automaton diagrams, could be used to describe the situation of two distinct states (at locations p'_1 and p'_2) linked by an arrow, but not the situation of an arrow going from a state to itself; a different production would be required for the latter.

Marriott [92] gives a rigorous formal semantics of CMGs (building on ideas published earlier in [69]), the details of which are beyond the scope of this review. Briefly, the full definition of a CMG includes a specification of a *computational domain* D with an associated *domain theory* T_D, being the first-order theory which axiomatizes D and functions over D (including constraints, which are Boolean functions). Notions such as derivation and the language of a CMG are then defined by treating the

sets of constraints in productions as constraint logic programs, whose semantics are well-defined.

Marriott also gives some important decidability and complexity results concerning the *membership problem* for CMGs, i.e. deciding whether or not a given multiset of tokens is in the language of a given CMG. For entirely arbitrary CMGs the problem is undecidable, even if no attributes are involved. The basic problem turns out to be productions which rewrite a nonterminal as another nonterminal, potentially introducing cycles into derivations. Marriott defines a class of *cycle-free* CMGs, proves that the membership problem for these is NP-complete, and presents an incremental parsing algorithm for cycle-free CMGs. In an even more recent paper [17], Chok and Marriott define a slightly more restrictive class of *stratifiable* CMGs which allows the use of *negative constraints* (of the form "**not exist** $T_1'' \ldots T_m''$"), and describe an automatic parser generator.

As mentioned above, CMG is essentially a refinement of Golin's AMG, and hence the two formalisms can describe essentially the same class of visual languages, which includes directed graphs. The major difference is that, because the semantics of CMGs are rigorously defined, the CMG formalism can be used for generation as well as recognition. Helm and Marriott have reported practical success with the former as early as 1990 [69]. A major consequence is that it is possible to define the language of a CMG, and hence to prove that a given CMG generates (recognizes) a given language.

3.6 Najork and Kaplan: Conditional Set Rewrite Systems

The formalisms discussed so far have the common property that they are based on "context-free" rewriting rules having a single symbol at the head. To express languages involving some context dependency (such as directed graph structures), both Golin *et al.* and Marriott *et al.* provided a mechanism of "remote" or "existentially quantified" symbols which, while not actually rewritten during application of a production, are nonetheless required to be present. Marc Najork and Simon Kaplan [100] on the other hand, chose to allow arbitrary numbers of symbols in the head of productions, in their formalism called *conditional set rewrite systems* (CSRS). Here is their very concise description of the formalism:

Informally, a CSRS consists of an ordered sequence of rewrite rules, which are *guarded* by a condition. Conditions are predicate applications, closed over conjunction and disjunction. Predicates are defined through Horn clauses. The syntax o' CSRS is as follows:

$$
\begin{array}{lcll}
t & ::= & x|k(t_1,\cdots,t_n) & \text{(term)} \\
\phi & ::= & P(t_1,\cdots,t_n)|\phi_1 \wedge \phi_2|\phi_1 \vee \phi_2 & \text{(formula)} \\
\delta & ::= & P(t_1,\cdots,t_n) \Leftarrow \phi & \text{(pred. def.)} \\
\rho & ::= & t_1,\cdots,t_n \rightarrow t'_1,\cdots,t'_n \textbf{ if } \phi & \text{(rule)} \\
\Sigma & ::= & (\rho_1 \cdots \rho_m, \delta_1 \cdots \delta_n) & \text{(system)}
\end{array}
$$

A *term* is either a *variable* or a *constructor* applied to some terms. A *formula* is either a *predicate symbol* applied to some terms, or the *conjunction* or *disjunction* of two formulas. A *predicate definition* defines $P(t_1,\cdots,t_n)$ to hold if the formula ϕ holds. A *rewrite rule* replaces the terms $t_1,\cdots,t_n$ in a set σ by terms $t'_1,\cdots,t'_n$ if ϕ holds. A *conditional set rewrite system* consists of an ordered sequence of rewrite rules, and a set of predicate definitions.

Given a set of terms σ and a rewrite rule $t_1,\cdots,t_m \rightarrow t'_1,\cdots,t'_m$ **if** ϕ, the rule is *applicable* if σ contains terms matching $t_1,\cdots,t_m$, and ϕ holds. *Applying* an applicable rewrite rule means replacing $t_1,\cdots,t_m$ in σ by $t'_1,\cdots,t'_n$. A rewrite step $\sigma \rightarrow \sigma'$ results from applying the *first* applicable rewrite rule to σ. We say that σ_0 rewrites to σ_n ($\sigma_0 \rightarrow_* \sigma_n$) if there is a sequence of rewrite steps $\sigma_0 \rightarrow \sigma_1 \rightarrow \cdots \rightarrow \sigma_n$. We say that σ is in *normal form* if there is not σ' s.t. $\sigma \rightarrow \sigma'$.

We allow for two notational simplifications: First, instead of $P(t_1,\cdots,t_n) \Leftarrow$ true, we simply write $P(t_1,\cdots,t_n)$ (and similar for $t_1,\cdots,t_m \rightarrow t'_1,\cdots,t'_n$ if true). Second, we allow an argument to be a simple function application instead of a term. For instance, we allow $t(n) \rightarrow t(n+1)$, which could be expanded to $t(n) \rightarrow t(n')$ if $\text{plus}(n,1,n')$.

A few clarifications need to be added:

1. The definition of a *term* given above describes occurrences of terms in rewrite rules. A CSRS defines permissible rewritings of (subsets of) sets of terms where the variables are replaced by actual values. For lack of other terminology, let us call the former *formal terms* and the latter *actual terms*.

2. The matching of actual terms in a set σ to formal terms in a rule head involves an assignment or binding from variables to actual values. The same assignment is applied to variables in the condition part of the rule, and the semantics of predicates applied to determine whether or not the condition is satisfied. If it is

satisfied, the same assignment is again applied to the formal terms in the rule body, yielding the actual terms which are substituted for the originally matched actual terms in σ.

3. A CSRS defines only valid rewriting steps. To define a language of sets of actual terms, one specifies a set ω of formal terms and states that a set σ is in the language if it can be rewritten to a set σ' matching ω. An elegant approach is to include a rule of the form $\omega \rightarrow \emptyset$ and state that a set σ is in the language if it can be rewritten to the empty set.

Najork and Kaplan give examples to illustrate that, like other formalisms, CSRSs can be ambiguous: a given input may be rewritten in several ways, which may lead to the same or different normal forms. They note that although algorithms are known which can determine whether or not a given string CFG is ambiguous, the same cannot be said of CSRS or the formalisms of Golin or Helm and Marriott. They outline proofs that CSRSs are at least as powerful as string CFGs, Golin's PLGs and Helm and Marriott's CMGs,[10] showing how each of these formalisms can be expressed in CSRS form. They give an example showing how a CSRS may be used to translate a set representation of a visual language into a sequential text language, and claim that this capability is unique to the CSRS formalism.[11]

The CSRS formalism is elegant and powerful. The associated definition of the language of a CSRS, involving rewriting of sentences to a specified normal form, seems oriented more toward recognition than generation. However, the definition is rigorous enough that even though Najork and Kaplan did not provide proofs that the CSRSs in their examples indeed generated the languages involved, it is straightforward to envisage the form such proofs would take. Najork and Kaplan do not provide a parsing algorithm; this is in keeping with their stated objective of proposing a mechanism for "describing the syntax of multidimensional languages".

[10]Actually they show that CSRSs are more powerful than Helm and Marriott's formalism, but they cite an earlier (1991) paper which did not include existential quantification of symbols.

[11]CMGs as defined most recently by Marriott [92] should be capable of such translations. CMGs are essentially programs in the constraint logic language CLP(R), and as such have the computing power of Turing machines. Furthermore, CLP(R) programs operate on terms whose definition exactly parallels Najork and Kaplan's.

3.7 Observations

Let us return to the points of comparison which I gave in the first section of this chapter.

1. *Representation:*

   - Clearly, choice of representation is important in defining grammar formalisms for pictures. All of the researchers cited above believed, as do I, that attributes are essential to the formulation of natural grammar formalisms for visual languages (VLs), basically because the pictures one would call "VL sentences" are composed of parametrized marks.
   - Early efforts to define VL grammars attempted to build upon results established with strings, and hence tried to force VLs into sequential representations. It is instructive that Kojima and Myers, who did not even try to define a new sequential representation for pictures but instead chose a well-established one (procedure call sequences), became entangled in issues related to order.
   - When order is eliminated, we have either sets or multisets. Both Golin and Reiss and Helm and Marriott chose multisets, but neither gave reasons. When one is accustomed to string grammar formalisms such as CFGs, multisets are attractive because they preserve cardinality. For example, when the CFG production $A \rightarrow BC$ rewrites the string BAC to $BBCC$, the string's length increases by one. The multiset analogue is straightforward: the number of symbols involved increases by one. In the set analogue, however, set construction causes the result to collapse to $\{B, C\}$; the cardinality actually decreases. However, the extra complexity of multiset representations buys nothing. In attributed set representations, it is the attribute values which distinguish symbols. Consider a multiset of attributed symbols representing parametrized marks, which could be interpreted as procedure calls to draw the marks. What does the multiplicity of symbols express? Drawing one or a hundred identical copies of a mark at the identical position, orientation, etc. yields the same result.

2. *Matching:*

   - All of the formalisms discussed in this chapter allow powerful expressions to be used in attribute computations. Even Kojima and Myers's GFG formalism, which is context-free in form and does not include existential quantification of symbols, can in a sense generate directed graph structures because its rewriting steps are capable of building and dismantling sets.[12]
   - Most of the formalisms make the applicability of rewriting steps contingent on the outcome of constraints, and some (most notably Helm and Marriott's CMG) assume a powerful simultaneous constraint-solving capability within the rewriting formalism itself.
   - The solving of constraints is in general not even limited in scope to individual rewriting steps—the constraint-based formalisms also make use of unification, augmented to support expression evaluation, as a means of propagating constraints into expanded contexts until they become evaluable.
   - These techniques essentially give the grammar formalisms computing power equivalent to a Turing machine. (If there are no limits on expressions, they might as well be Turing machine computations.) The result is that tremendous generative power is bought at the expense of analyzability and, as Marriott [92] has shown, decidability of the membership problem.

3. *Parse structure:*

   - Several of the authors appear to favour tree-like parse structures. This is intimately related to the nearly universal practice of defining rewriting rules based on context-free basis productions having a single symbol at the head.
   - Tree structures are familiar (from established work in string parsing with CFGs and attribute grammars). Aside from that I see little advantage in them.

[12] I say "in a sense" because the rewriting process itself does not generate the set of graph nodes, but rather "transcribes" a set (of (x, y) coordinates of node positions) supplied as an attribute of the start symbol. Hence the set-generating capability of the GFG formalism is inherent in the instantiation of the start symbol, not in the rewriting process.

4. *Expressive power:*

   - I consider the ability to generate arbitrary directed graph structures to be a minimum requirement for a visual grammar formalism. With the exception of Najork and Kaplan's CSRS, most formalisms resort to "tricks" to achieve this.
   - There is also the question of context dependency in derivations. I conjecture that "context sensitive" grammar formalisms which permit an arbitrary number of symbols in the head of a production are inherently more powerful than "context free" formalisms which permit only one head symbol. So far, however, I have been unable to prove this.

5. *Symmetry, Rigour:*

   - Not all of the formalisms discussed above can be used in both generation and recognition (parsing) of languages.
   - Although parsing of visual languages is the practical goal (in order to realize interactive notation-processing systems), if a formalism cannot be used for generation it is difficult to specify precisely what language a given instance of that formalism actually defines.
   - Without a definition of the language of a grammar G, we cannot prove that a parser based on G will actually recognize the language we expect.

A final observation links the last of these points to the first. Any grammar formalisms, grammars, parsing algorithms, etc. we devise will operate not on notations, but on their *representations* in a computer's memory. Before we proceed to write grammars, prove them correct with respect to the language of representations, and so on, we must be satisfied that our chosen representation scheme captures the salient features of notations. This is outside the realm of mathematical proof.

Chapter 4

Attributed Sets, Languages, and Grammars

In this chapter I propose that a specific algebraic structure—the *attributed set*—can be used to model multidimensional objects including notations. I consider the semantics of interpreting attributed set models, especially as graphics. I then develop a grammar formalism—*attributed set grammar* or ASG—in order to have finite descriptions of potentially infinite sets of such models, which I call *attributed languages.* I present several examples to illustrate the use of attributed sets and ASGs in modelling algebraic and notation structures. Finally, I consider the use of ASGs in syntactic analysis (parsing) of attributed sets. I show that the membership problem is decidable for a certain class of attributed set languages, and present a working attributed set acceptor program in Prolog.

4.1 Algebraic preliminaries and notation

In this section I clarify my interpretations of some conventional mathematical notions, and define "classes" and "instance," which are generalizations of the conventional notions of Cartesian product and tuple, respectively.

4.1.1 Sets, relations and functions

Let us accept as needing no definition the standard mathematical notions of *set* and *relation*, including all related operations such as powerset (denoted e.g. $\mathcal{P}(A)$), Cartesian product of sets (denoted e.g. $A \times B$), domain and range of a relation (denoted e.g. $\mathcal{D}(\mathrm{R})$ and $\mathcal{R}(\mathrm{R})$ respectively), and composition of relations (denoted e.g. $\mathrm{R} \circ \mathrm{S}$). General subset (superset) relationship is denoted e.g. $A \subseteq B$ ($A \supseteq B$); moreover when it is certain that $A \neq B$ we write $A \subset B$ ($A \supset B$) instead. Set difference is denoted e.g. $A \setminus B$ (interpretation: $(A \setminus B) \cup (A \cap B) = A$).

Sets are herein denoted primarily by uppercase italic letters e.g. A, B, etc., with the following exceptions:

- standard designators are used where appropriate, e.g.
  - $\emptyset$ denotes the empty set.
  - $\mathbb{N}$ denotes the set of natural numbers $\{1, 2, 3, \ldots\}$.
  - $\mathbb{N}_0$ denotes $\mathbb{N} \cup \{0\}$.
  - $\mathbb{Z}$ denotes the set of integers.
  - $\mathbb{R}$ denotes the set of real numbers.
  - $\mathbb{B}$ denotes the set $\{0, 1\}$.
- uppercase bold letters e.g. $\mathbf{A}$, $\mathbf{B}$, etc. denote sets of sets.
- uppercase roman and Greek letters e.g. R, S, Σ denote relations of all kinds. Underscoring is added to denote a class (defined below, e.g. $\underline{\mathrm{C}}$).

Let us denote the (possibly infinite) union of the members of a set of sets $\mathbf{X} = \{X_1, X_2, \ldots\}$ as $\cup\mathbf{X}$, i.e., $\cup\mathbf{X} = X_1 \cup X_2 \cup \cdots$.

A relation $\mathrm{F} \subseteq A \times B$ is called a *partial function from A to B* iff $\forall a \in A$, $b, c \in B$, $(a, b), (a, c) \in \mathrm{F}$ implies $b = c$. Furthermore,

- If $\mathcal{D}(\mathrm{F}) = A$, F is called a *total function*. Every total function is also a partial function.
- If $\mathcal{R}(\mathrm{F}) = B$, F is called *surjective*.
- If $\forall a, b \in A$, $c \in B$, $(a, c), (b, c) \in \mathrm{F}$ implies $a = b$, F is called *injective*, and the inverse relation $\mathrm{F}^{-1} : B \rightarrow A$ is also a partial function.
- If F is both injective and surjective, it is called *bijective* or "a bijection," and its inverse is also a bijection.
- If F is a bijective total function, then $|A| = |B|$ and $\mathcal{R}(\mathrm{F}^{-1}) = A$. If F is bijective but not total, $|A| > |B|$ and $\mathcal{R}(\mathrm{F}^{-1}) \subset A$.

Let us extend the usual functional notation to permit subsets of a (partial) function's domain as arguments, i.e. let $\mathrm{F} : X \rightarrow Y$ and $Z \subseteq \mathcal{D}(\mathrm{F})$. Then $\mathrm{F}(Z)$ denotes the set $\{\mathrm{F}(z) \mid z \in Z\}$. Note that for any function F, $\mathrm{F}(\mathcal{D}(\mathrm{F})) = \mathcal{R}(\mathrm{F})$.

Let us also allow the notation of functions to be used with general relations. Let A and B be sets and $\mathrm{R} \subseteq A \times B$. The notation $\mathrm{R}(x)$ where $x \in A$ denotes the set $\{y \mid (x,y) \in \mathrm{R}\}$ if $x \in \mathcal{D}(\mathrm{R})$ or $\emptyset$ otherwise. This is extended to subsets of the relation's domain as defined above for functions, i.e., the notation $\mathrm{R}(X)$ where $X \subseteq A$ denotes the set $\cup_{x \in X} \mathrm{R}(x)$.

4.1.2 Atomic alphabets, strings, string languages

In this chapter I define algebraic structures which I call "attributed alphabets", "attributed words", and "attributed languages". For the sake of brevity, the prefix "attributed" is often omitted. On the few occasions when I need to refer to the conventional meanings of the terms "alphabet", "word", and "language", I use the terms "atomic alphabet", "string", and "string language" (respectively) instead. For completeness, I now define these terms and some associated notation.

An *atomic alphabet* is a finite, non-empty set. The elements of the set may be called *letters*, and are generally treated as atomic objects without internal structure. Let A be an atomic alphabet. A *string over* A is a finite sequence of letters which are elements of A. E.g., let $A = \{a, b, c\}$. Strings over A include a, $abbc$, and the *empty string*, which contains no letters. The empty string (over any atomic alphabet) is denoted by ε. The set of all strings over an antomic alphabet A is denoted A^*. (Note $\varepsilon \in A^*$ for any atomic alphabet A.) Most discussions of formal language theory do not distinguish between the letter $a \in A$ and the one-letter string $a \in A^*$. A *string language over* A is any subset of A^*.

4.1.3 Boolean values and operations, predicates

Let us denote the the *Boolean set* $\{0,1\}$ by $\mathbf{B}$. The Boolean operations "AND" and "OR" are denoted by the two-place infix operators $\wedge$ and $\vee$, respectively, defined over $\mathbf{B}$ as follows:

$$x \wedge y = \begin{cases} 1, & \text{if } x = y = 1; \\ 0, & \text{otherwise.} \end{cases}$$

$$x \vee y = \begin{cases} 0, & \text{if } x = y = 0; \\ 1, & \text{otherwise.} \end{cases}$$

Any function whose range is $\mathbf{B}$ is called a *predicate.*

4.1.4 Families

Let S be a set. A *family of elements from* S is a total function from a set I, called the *index set of* F, to S, e.g. $\mathrm{F} : I \to S$. The elements of $\mathcal{R}(\mathrm{F})$ are called the *members* of the family. Given $i \in I$, member $\mathrm{F}(i)$ is called the *ith member*, or the *member indexed by* i.

The notion of family is a generalization of the notion of tuple or sequence; indeed a sequence can be considered as a family indexed by an initial subset of $\mathbf{N}$. This motivates an alternative notation for families, wherein the elements are subscripted by their corresponding indices and written between parentheses, e.g. $(\alpha_1, \beta_2, \alpha_3)$. Ellipses may be used, e.g. $(s_1, s_2, \ldots, s_k)$ in the finite case or $(s_1, s_2, \ldots)$ in the infinite case. The shorter form $(s_i)_{i \in I}$ may also be used, with s_i referring to the ith element in a generic sense.

The notations (a_x, b_y) and (b_y, a_x) specify the same family, much as the notations $\{x, y\}$ and $\{y, x\}$ specify the same set. However, if the index set has been written explicitly in some order, e.g. $I = \{x, y\}$, it is most natural to write families indexed by I in the same order, e.g. (a_x, b_y).

4.1.5 Classes and instances

The Cartesian product provides a simple way of constructing or specifying sets of tuples, i.e. the notation $A \times B \times C$ where A, B, and C are sets, specifies the set $\{(a, b, c) \mid a \in A, b \in B, c \in C\}$. In this section I define an analogous notion for families, which I call the *class*.[1]

A *class* is a family of sets, i.e., a family whose members are sets. Classes will usually be denoted by underscored roman capital letters. A *instance* of a class $\underline{\mathrm{C}} : I \to \mathbf{S}$ is a family $\mathrm{F} : I \to \cup\mathbf{S}$ such that $\forall i \in I$, $\mathrm{F}(i) \in \underline{\mathrm{C}}(i)$. The set of all instances of a class $\underline{\mathrm{C}}$ is denoted $\underline{\text{instances}}\,(\underline{\mathrm{C}})$.

The members of a class are called *sorts*. The range of a class $\underline{\mathrm{C}}$ is denoted $\underline{\text{sorts}}\,(\underline{\mathrm{C}})$.

When a class has been written explicitly in parenthesized form, e.g. $(\mathbf{R}_x, \mathbf{R}_y)$, it is most natural to write its instances with their members listed in the same order, e.g. $(1.2_x, -3.7_y)$. The subscripts appearing in the instances are then redundant, and may be omitted without loss of clarity.

---

[1] My usage of the terms class and instance is analogous to the similarly-named notions in object-oriented systems. It is unrelated to the set-theoretic concept of "class".

Example 2: Modelling some geometric objects

Let us define classes to model some geometric objects defined in the plane, using Cartesian coordinates.

A point has two real-valued coordinates x and y; this can be captured by a class such as

$$\underline{\mathrm{C}} = (\mathbf{R}_x, \mathbf{R}_y).$$

An instance of $\underline{\mathrm{C}}$ could be written e.g. $(-2.1_x, 4.5_y)$. Since $\underline{\mathrm{C}}$ has been explicitly written above with the xth member first and the yth member second, we might as well write simply $(-2.1, 4.5)$.

A line segment is defined by its two endpoints $P = (x_1, y_1)$ and $Q = (x_2, y_2)$; this can be captured by a class

$$\underline{\mathrm{D}} = (\mathbf{R}^2_P, \mathbf{R}^2_Q).$$

or alternatively,

$$\underline{\mathrm{E}} = (\mathbf{R}_{x_1}, \mathbf{R}_{y_1}, \mathbf{R}_{x_2}, \mathbf{R}_{y_2})$$

The line segment from $P = (1, 2)$ to $Q = (10, 20)$ could be represented either as the $\underline{\mathrm{D}}$-instance $((1, 2), (10, 20))$. or as the $\underline{\mathrm{E}}$-instance $(1, 2, 10, 20)$.

Note that either $\underline{\mathrm{D}}$-instances or $\underline{\mathrm{E}}$-instances could also be used to model rectangles aligned with the coordinate axes, if we interpret point P as the upper-left corner and Q as the lower-right corner.

4.2 Attributed words and languages

In conventional formal language theory, symbols (also called letters) are atomic and words are sequences of symbols. In this section I develop an analogous formalism in which symbols have associated attributes, and words are sets, rather than sequences, of such attributed symbols. The motivation for doing so is that the notion of attributed word captures the structure of notations in a way which is both natural and amenable to computation.

4.2.1 Attributed alphabets and symbols

An *attributed alphabet* is a family $(\underline{\mathrm{C}}_k)_{k \in K}$ of classes. The indices $k \in K$ are called *class names.*

Let $A = (\underline{C}_k)_{k \in K}$ be an attributed alphabet. An *attributed symbol of class* k, *drawn from* A is a pair (k, ι) where $k \in K$ and ι is an instance of class $\underline{C}_k$. The set of all attributed symbols of class k drawn from A, i.e., $\{(k, \iota) \mid \iota \in \underline{\text{instances}}(\underline{C}_k)\}$ is denoted $\underline{\text{symbols}}_k(A)$. The set $\bigcup_{k \in K} \underline{\text{symbols}}_k(A)$ is denoted $\underline{\text{symbols}}(A)$.

The parenthesized notation for families is extended to attributed symbols (k, ι) by writing the class name to the left of the parenthesized notation for ι. This is illustrated in the example below.

We extend the definition of the operator $\underline{\text{sorts}}$ to attributed alphabets thus: for $A = (\underline{C}_k)_{k \in K}$, $\bigcup_{k \in K} \underline{\text{sorts}}(\underline{C}_k)$ is denoted $\underline{\text{sorts}}(A)$.

Example 3: Geometric objects, continued

Recall from Example 2 the objective of modelling points, line segments, and rectangles aligned with the coordinate axes, and the classes $\underline{C}$ and $\underline{D}$ defined for the purpose.

Let $K = \{\text{point}, \text{lineseg}, \text{rectbox}\}$ be a set of class names, and define an attributed alphabet $A = (\underline{C}_k)_{k \in K} = \left(\underline{C}_{\text{point}}, \underline{D}_{\text{lineseg}}, \underline{D}_{\text{rectbox}}\right)$. Rather than modelling the point $(1, 2)$, say, as just the $\underline{C}$-instance $(1, 2)$, we model it as an attributed symbol of class "point", i.e., point$(1, 2)$. We have in effect added a label (class name) to the representation which clarifies its interpretation.

Both the representations of line segments and those of rectangular boxes involve instances of the class $\underline{D}$. However we can now clearly distinguish between representations of line segments and rectangles, by modelling the former as attributed symbols of class *lineseg*, e.g. lineseg$((1, 3), (-2.4, 3))$, and the latter as attributed symbols of class *rectbox*, e.g. rectbox$((1, 3), (4, 5))$.

4.2.2 Attributed words and languages

Let A be an attributed alphabet. An *attributed word over* A is any finite subset of $\underline{\text{symbols}}(A)$. The set of all attributed words over A is denoted $\underline{\text{words}}(A)$. An *attributed language over* A is any subset of $\underline{\text{words}}(A)$. The adjective "attributed" may be omitted where no confusion would result.

Example 4: Geometric interpretation of words and languages

Recall the attributed alphabet A defined in the previous example. The attributed word

$$T = \{\, \text{lineseg}((-1,0),(0,2)), \text{lineseg}((0,2),(1,0)), \text{lineseg}((1,0),(-1,0)) \,\}$$

could be used to represent a triangle with vertices $(-1,0)$, $(0,2)$, and $(1,0)$. The infinite set

$$L = \left\{ \{\text{lineseg}(p_1,p_2), \text{lineseg}(p_2,p_3), \text{lineseg}(p_3,p_1)\} \;\middle|\; p_1,p_2,p_3 \in \mathbf{R}^2 \right\}$$

could be called a language of representations of triangles. L contains a representation for every triangle in the plane.[2]

4.2.3 Comparison with ordinary formal language theory

When drawing comparisons with ordinary formal language theory, I use the term *string* in place of the term "word".

Note that an attributed word is a *set*, while a string is a *sequence*; this is why I call this formalism the "attributed set" formalism. Strings may contain repeated symbols; attributed words can not. An attributed word may, however, contain multiple symbols of the same class, but with different attributes.

By defining and interpreting attributes appropriately, we can use them to model many kinds of relationships among symbols. In this dissertation I am most concerned with geometric relationships in the plane, but as the following example shows we can also express left-to-right sequence (concatenation) using attributes.

Example 5: Modelling strings with attributed words

Let $X = \{a,b\}$ be an (ordinary, non-attributed) alphabet. One way to encode strings over X as attributed words is to use the letters a,b as class names and let each symbol have one attribute identifying the position of the corresponding letter in the string. That is, define $\underline{\mathrm{C}} = (\mathbf{N}_{\mathrm{pos}})$ and $\mathrm{A} = (\underline{\mathrm{C}}_a, \underline{\mathrm{C}}_b)$. The string *abba* would be encoded as the word

[2] It does not, however, contain every representation. The triangle with vertices p_1, p_2, and p_3 could also be represented by a word of the form $\{\text{lineseg}(p_1,p_2), \text{lineseg}(p_1,p_3), \text{lineseg}(p_2,p_3)\}$. That is why I call *$L$ a language of representations of triangles*, rather than *the language of all triangle representations*.

$\{a(1), b(2), b(3), a(4)\}$. The empty string would be encoded as the empty word $\emptyset$.

Of course, many words over A are not valid encodings of strings over X. We can define a language $L \subset \underline{\text{words}}(\text{A})$ which consists entirely of valid encodings of strings over X, as the set of all $W \in \underline{\text{words}}(\text{A})$ such that

1. $\forall (k_1, \iota_1), (k_2, \iota_2) \in W,\ \iota_1 = \iota_2$ implies $k_1 = k_2$.
2. $\forall (k_1, \iota_1) \in W$, either $\iota_1(\text{pos}) = 1$ or $\exists (k_2, \iota_2) \in W$ such that $\iota_2(\text{pos}) = \iota_1(\text{pos}) - 1$.

Condition 1 guarantees that there are no two letters occupy the same position in the string. Condition 2 ensures that the positions are always numbered consecutively starting with 1.

The issue of valid *vs.* invalid encodings is considered later in section 4.5.1.

4.2.4 Term systems

A many-sorted *term system* is a bijection $\Sigma : \mathbf{V} \rightarrow \mathbf{S}$ where

1. $\mathbf{V} = \{V_1, V_2, \ldots, V_n\}$ is a finite set of sets which are pairwise disjoint.
2. $\mathbf{S} = \{S_1, S_2, \ldots, S_n\}$ is a finite set of sets.
3. $(\cup\mathbf{V}) \cap (\cup\mathbf{S}) = \emptyset$.

Condition 2 implies that a term system is a class; in keeping with class terminology the elements of the range $\mathbf{S}$ are called *sorts* and we write $\underline{\text{sorts}}(\Sigma) = \mathbf{S}$. The elements of each sort S_i are called *constants of sort* S_i *under* Σ. (The "under Σ" can be omitted when no confusion would result.) The elements V_i of $\mathbf{V}$ are called *variable domains*, and the elements $v \in V_i$ of each $V_i \in \mathbf{V}$ are called *variables of sort* $\Sigma(V_i)$ *under* Σ. Since Σ is a bijection, and since as stated in condition 1, the V_i are pairwise disjoint, each variable is of exactly one sort under Σ. (Note that no such restriction holds for constants.) Constants and variables under Σ are collectively called *terms under* Σ, or Σ-*terms*.

Let $\Sigma : \mathbf{V} \rightarrow \mathbf{S}$ be a term system. We define:

$$\underline{\text{constants}}(\Sigma) = \cup\mathbf{S}$$

$$\underline{\text{variables}}(\Sigma) = \cup\mathbf{V}$$

$$\underline{\text{sort}}_\Sigma : \underline{\text{variables}}(\Sigma) \rightarrow \underline{\text{sorts}}(\Sigma) \text{ where}$$

$$\forall V_i \in \mathbf{V},\ \forall v \in V_i,\ \underline{\text{sort}}_\Sigma(v) = \Sigma(V_i)$$

$$\underline{\text{terms}}_\Sigma : \mathbf{S} \rightarrow \mathcal{P}(\cup\mathbf{S} \bigcup \cup\mathbf{V}) \text{ where}$$

$$\forall S_i \in \mathbf{S},\ \underline{\text{terms}}_\Sigma(S_i) = S_i \cup \Sigma^{-1}(S_i)$$

Given any variable under Σ, operator $\underline{\text{sort}}_\Sigma$ returns its sort. Given any sort of Σ, operator $\underline{\text{terms}}_\Sigma$ returns the union of that sort and the variable domain of that sort, i.e., for any $S_i \in \mathbf{S}$, elements of $\underline{\text{terms}}_\Sigma(S_i)$ are all terms of sort S_i.

Example 6: Term systems

Let $\mathbf{V} = \{U, W\}$, $U = \{u_1, u_2, \ldots\}$, $W = \{w_1, w_2, \ldots\}$, $\mathbf{S} = \{\mathbf{Z}, \mathbf{R}\}$, and $\Sigma = (\mathbf{Z}_U, \mathbf{R}_W)$. We have $\underline{\text{sorts}}(\Sigma) = \{\mathbf{Z}, \mathbf{R}\}$, $\underline{\text{constants}}(\Sigma) = (\mathbf{Z} \cup \mathbf{R}) = \mathbf{R}$, and $\underline{\text{variables}}(\Sigma) = U \cup W$.

1, 2, 3, etc. are constants of sort $\mathbf{Z}$ and sort $\mathbf{R}$. 1.2 is a constant of sort $\mathbf{R}$ only. u_1, u_2, etc. are variables of sort $\mathbf{Z}$, e.g. $\underline{\text{sort}}_\Sigma(u_1) = \underline{\text{sort}}_\Sigma(u_{99}) = \mathbf{Z}$. w_1, w_2, etc. are variables of sort $\mathbf{R}$, e.g. $\underline{\text{sort}}_\Sigma(w_{50}) = \underline{\text{sort}}_\Sigma(w_{1000}) = \mathbf{R}$.

1, 1.5, 3, u_1, and w_{100} are examples of Σ-terms. $\underline{\text{terms}}_\Sigma(\mathbf{Z}) = \mathbf{Z} \cup U$. $\underline{\text{terms}}_\Sigma(\mathbf{R}) = \mathbf{R} \cup W$.

4.2.5 Assignments, extended classes and alphabets

Let $\Sigma : \mathbf{V} \rightarrow \mathbf{S}$ be a term system. A *Σ-assignment* is a partial function α : $\underline{\text{variables}}(\Sigma) \rightarrow \underline{\text{constants}}(\Sigma)$ which preserves sortedness, i.e., $\forall v \in \mathcal{D}(\alpha)$, $\alpha(v) \in \underline{\text{sort}}_\Sigma(v)$.

Let $\underline{C} : I \rightarrow \mathbf{S}$ be a class with $\underline{\text{sorts}}(\underline{C}) = \mathbf{S} = \underline{\text{sorts}}(\Sigma)$. Define the *extended class* $\underline{C}^\Sigma : I \rightarrow \underline{\text{terms}}_\Sigma(\mathbf{S})$ where $\forall i \in I$, $\underline{C}^\Sigma(i) = \underline{\text{terms}}_\Sigma(\underline{C}(i))$. This simply means that instances of the extended class may have either constant or variable attributes, with sortedness preserved.

Lemma 1 $\forall i \in I,\ \underline{\mathbf{C}}^{\Sigma}(i) \cap \underline{constants}(\Sigma) = \underline{\mathbf{C}}(i)$.

Proof: In the following, $i \in \mathcal{D}(\underline{\mathbf{C}})$. $\underline{\mathbf{C}}(i)$ is a sort of $\underline{\mathbf{C}}$; it contains only constants. By the definition above, $\underline{\mathbf{C}}^{\Sigma}(i) = \underline{\text{terms}}_{\Sigma}(\underline{\mathbf{C}}(i))$, which contains all constants and all variables of sort $\underline{\mathbf{C}}(i)$. $\underline{\text{constants}}(\Sigma)$ consists of all constants under Σ, and no variables under Σ, which are not also constants.[3] Hence the intersection removes all variables from $\underline{\mathbf{C}}^{\Sigma}(i)$, leaving $\underline{\mathbf{C}}(i)$. □

Lemma 2 $\underline{instances}(\underline{\mathbf{C}}) \subseteq \underline{instances}\left(\underline{\mathbf{C}}^{\Sigma}\right)$. *The inclusion is proper unless* $\cup\mathbf{V} = \emptyset$.

Proof: The inclusion is a consequence of Lemma 1. If $\cup\mathbf{V} = \emptyset$, i.e. there are no variables, then $\forall i \in \mathcal{D}(\underline{\mathbf{C}})$, $\underline{\text{terms}}_{\Sigma}\,\underline{\mathbf{C}}(i) = \underline{\mathbf{C}}(i)$ and $\underline{\text{instances}}(\underline{\mathbf{C}}) = \underline{\text{instances}}\left(\underline{\mathbf{C}}^{\Sigma}\right)$. Suppose $\cup\mathbf{V} \neq \emptyset$ and let $v \in \cup\mathbf{V}$. Let ι be any instance of $\underline{\mathbf{C}}$ containing an attribute a of the same sort as v. Substituting v in place of a in ι yields an instance of $\underline{\mathbf{C}}^{\Sigma}$ which is not an instance of $\underline{\mathbf{C}}$. □

Given $\iota \in \underline{\text{instances}}\left(\underline{\mathbf{C}}^{\Sigma}\right)$ and a Σ-assignment α, define the operation of *substitution* in ι according to α as replacing every variable $v \in \mathcal{D}(\alpha)$ which appears in ι by its image $\alpha(v)$. The result is denoted $\iota|_{\alpha}$, and in the general case, $\iota|_{\alpha} \in \underline{\text{instances}}\left(\underline{\mathbf{C}}^{\Sigma}\right)$.

Lemma 3 *Let* $\underline{variables}(\iota)$ *denote the set of variables appearing in* ι.

1. $\underline{variables}(\iota|_{\alpha}) \subseteq \underline{variables}(\iota)$ *and the inclusion is proper iff* $\underline{variables}(\iota) \cap \mathcal{D}(\alpha) \neq \emptyset$.
2. $\iota|_{\alpha} \in \underline{instances}(\underline{\mathbf{C}})$ *iff* $\underline{variables}(\iota) \subseteq \mathcal{D}(\alpha)$.

Proof 1: The construction of $\iota|_{\alpha}$ replaces variables by constants; hence the inclusion. As the replacement takes place for every occurrence in ι of a variable in $\mathcal{D}(\alpha)$, the inclusion is proper if and only if at least one variable in $\mathcal{D}(\alpha)$ occurs in ι. □

Proof 2: If all variables occurring in ι are in $\mathcal{D}(\alpha)$, then $\underline{\text{variables}}(\iota|_{\alpha}) = \emptyset$ and hence, $\iota|_{\alpha} \in \underline{\text{instances}}(\underline{\mathbf{C}})$. Conversely, if $\iota|_{\alpha} \in \underline{\text{instances}}(\underline{\mathbf{C}})$, then $\underline{\text{variables}}(\iota|_{\alpha}) = \emptyset$ and, consequently, every variable occurring in ι is replaced by a constant; that is, $\underline{\text{variables}}(\iota) \subseteq \mathcal{D}(\alpha)$. □

Let $\mathrm{A} = (\underline{\mathbf{C}}_k)_{k \in K}$ be an attributed alphabet. Define the *extended alphabet* $\mathrm{A}^{\Sigma} = \left(\underline{\mathbf{C}}_k^{\Sigma}\right)_{k \in K}$ wherein each class is extended as above. The notion of substitution can now

[3]Normally, we will want constants and variables to be distinct, but this is not required by the definition of a term system.

be extended to attributed words in the obvious way. First note that an attributed symbol $\sigma = (k, \iota)$ drawn from A^{Σ} is such that $\iota \in \underline{\text{instances}}\left(\underline{\mathrm{C}}_k^{\Sigma}\right)$, and hence we can define $\sigma|_\alpha = (k, \iota|_\alpha)$. In general $\sigma|_\alpha \in \underline{\text{symbols}}_k\left(\mathrm{A}^{\Sigma}\right)$.

Lemma 4 *Let* $\underline{\text{variables}}(\sigma)$ *denote the set of variables appearing in* σ.

1. $\underline{\text{variables}}(\sigma|_\alpha) \subseteq \underline{\text{variables}}(\sigma)$ *and the inclusion is proper iff* $\underline{\text{variables}}(\sigma) \cap \mathcal{D}(\alpha) \neq \emptyset$.
2. $\sigma|_\alpha \in \underline{\text{symbols}}_k(\mathrm{A})$ *iff* $\underline{\text{variables}}(\sigma) \subseteq \mathcal{D}(\alpha)$.

Proof: Analogous to the proof of Lemma 3. □

Now let $W \in \underline{\text{words}}\left(\mathrm{A}^{\Sigma}\right)$, and define $W|_\alpha = \{\sigma|_\alpha \mid \sigma \in W\}$. In general $W|_\alpha \in \underline{\text{words}}\left(\mathrm{A}^{\Sigma}\right)$.

Lemma 5 *Let* $\underline{\text{variables}}(W)$ *denote the set of variables appearing in* W.

1. $\underline{\text{variables}}(W|_\alpha) \subseteq \underline{\text{variables}}(W)$ *and the inclusion is proper iff* $\underline{\text{variables}}(W) \cap \mathcal{D}(\alpha) \neq \emptyset$.
2. $W|_\alpha \in \underline{\text{words}}(\mathrm{A})$ *iff* $\underline{\text{variables}}(W) \subseteq \mathcal{D}(\alpha)$.

Proof: Analogous to the proof of Lemma 3. □

Substitution in an instance ι (symbol σ, word W) normally reduces the number of variables present (item 1 in the three preceding lemmas). When no variables remain after substitution (item 2 in the lemmas), $\iota|_\alpha$ ($\sigma|_\alpha$, $W|_\alpha$) is called *ground* and α is called a *grounding assignment* for ι (σ, W). Alternatively we say α *grounds* ι (σ, W).

Example 7: Geometric interpretation of symbols with variables

Let $\underline{\mathrm{C}} = (\mathbf{R}_x, \mathbf{R}_y)$ and $\mathrm{A} = \left(\underline{\mathrm{C}}_{\text{point}}\right)$. Words over A can be interpreted as sets of points in the plane.

Now let $V = \{v_1, v_2, \ldots\}$ and $\Sigma = (\mathbf{R}_V)$. The following are all attributed symbols drawn from A^{Σ}:

- point$(-5.1, 4.2)$ is ground; it represents the specific single point whose coordinates are $(-5.1, 4.2)$.
- point$(-5.1, v_{99})$ potentially represents any single point on the line $x = -5.1$.
- point(v_1, v_2) is constant-free; it potentially represents any single point in the plane.

Example 8: Geometric interpretation of attributed words with variables

Recall the language of representations of triangles given in Example 4, and the corresponding definitions relating to the alphabet A. Let $P = \{p_1, p_2, \ldots\}$ and $\Sigma = (\mathbf{R}_P^2)$. $\{\text{lineseg}(p_1, p_2), \text{lineseg}(p_2, p_3), \text{lineseg}(p_3, p_1)\}$ is a word over A^Σ, which potentially represents any triangle with vertices in the plane.

Let α be any Σ-assignment which assigns $\alpha(p_1) = (-1, 0)$, $\alpha(p_2) = (0, 2)$, and $\alpha(p_3) = (1, 0)$. Note that $W|_\alpha$ is precisely the (ground) word T given in Example 4.

4.3 Semantics

Several examples above have suggested how attributed words can be interpreted graphically. In this section I define two levels of formal semantics. The first level applies to ground words only, and involves well-defined mappings from attributed symbols to elements of some chosen set. The second level introduces *constraints* mappings from attributed symbols to predicates—and the mechanism of *constraint satisfaction* which, under the right circumstances, determines a grounding assignment automatically.

4.3.1 Procedural interpretation of ground words

The following example suggests a possible mechanism for graphical interpretation of ground words, which gives an intuitive motivation for the formal semantics to be introduced.

Example 9: Procedural interpretation

Consider the following Pascal procedure declarations.

procedure dot (**real** x, y);
begin
  ...draw a circular dot at location (x, y) on output device...
end;

procedure lineseg (**real** x_1, y_1, x_2, y_2);
begin
  ...draw a line segment with endpoints (x_1, y_1) and (x_2, y_2)...
end;

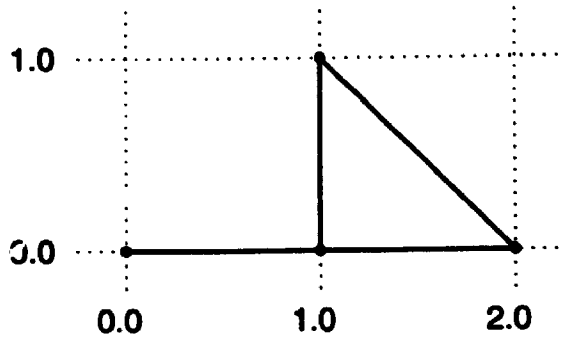


Figure 4.1: Simple unlabelled graph

Let $\underline{C} = (\mathbf{R}_x, \mathbf{R}_y)$, $\underline{E} = (\mathbf{R}_{x_1}, \mathbf{R}_{y_1}, \mathbf{R}_{x_2}, \mathbf{R}_{y_2})$, and $A = \left(\underline{C}_{\text{dot}}, \underline{E}_{\text{lineseg}}\right)$. Symbols drawn from A, such as dot(0, 1) or lineseg(1, 1, 1, 0), look a lot like Pascal procedure calls. Indeed, the word $W = \{\text{dot}(0,1), \text{dot}(1,1), \text{dot}(2,1), \text{dot}(1,0), \text{lineseg}(0,1,1,1), \text{lineseg}(1,1,2,1), \text{lineseg}(1,1,1,0), \text{lineseg}(1,0,2,1)\}$, interpreted as a set of calls to the two procedures above, would give rise to a picture like that shown in Figure 4.1.

The above example is not quite complete. We have to ensure that:

1. The initial conditions are defined, i.e. before any procedures are executed, the output device should be blank.

2. The order of calls doesn't matter. Since an attributed word is a set rather than a sequence, we could conceivably issue the procedure calls in any order (or even in parallel); the resulting picture should be the same in any case.

The functional interpretation defined in the next section eliminates these difficulties.

4.3.2 Functional interpretation: A formal semantics of ground words

Let $A = (\underline{C}_k)_{k \in K}$ be an attributed alphabet. An *interpretation scheme for* A is a quadruple $(S, \oplus, 1_S, F)$ where

- $(S, \oplus, 1_S)$ is a commutative monoid, i.e.,
  - S is a set.
  - $\oplus$ is an associative, commutative binary operator on S, under which S is closed.
  - 1_S is an identity element, i.e., $\forall s \in S \;\; 1_S \oplus s = s \oplus 1_S = s$.

- $F = (f_k)_{k \in K}$ is a family of partial functions such that $\forall k \in K$, $\mathcal{D}(f_k) =$ instances$(\mathbb{C}_k)$ and $\mathcal{R}(f_k) = S$.

Let A be an attributed alphabet, and let $\varphi = (S, \oplus, 1_S, F)$ be an interpretation scheme for A. Define the partial function $\|_\varphi$: words$(A) \rightarrow S$ as follows:

$$\|_\varphi(W) = \begin{cases} 1_S, & \text{if } W = \emptyset; \\ \bigoplus\limits_{(k,\iota) \in W} f_k(\iota), & \text{otherwise.} \end{cases}$$

This function defines at most one *interpretation* of a word over A as an element of the set S. By analogy to substitution, we write $W\|_\varphi$ instead of $\|_\varphi(W)$ from here on.

Lemma 6 *$\|_\varphi$ is total iff all the f_k are total.*

Proof: Let $W \in$ words(A). W consists of symbols (k, ι) where $\iota \in$ instances$(\mathbb{C}_k)$. If all f_k are total then $f_k(\iota)$ is defined for all k and ι; hence $\|_\varphi$ is total. Conversely, if $\|_\varphi$ is total, then $f_k(\iota)$ must be defined for all k and ι; hence all the f_k are total. □

Lemma 7 $\forall W_1, W_2 \in$ words(A), $W_1\|_\varphi \oplus W_2\|_\varphi = (W_1 \cup W_2)\|_\varphi$.

Proof: The result follows by substitution into the definition of $\|_\varphi$ above. □

Suppose a word W can be partitioned into two subwords (subsets) W_1 and W_2, and suppose $W_2\|_\varphi = 1_S$. Substituting into the equation of Lemma 7, we get $W_1\|_\varphi = (W_1 \cup W_2)\|_\varphi$, meaning that the interpretation of $W = W_1 \cup W_2$ is the same as that of just W_1. A practical application of this notion is that if we want attributed symbols of a certain class k to have no effect on the interpretation of words, we can simply define the associated interpretation function f_k as total with range $\{1_S\}$. An interpretation function of this kind is called a *null interpretation for symbols of class k under φ*. Use of null interpretations is discussed in Section 4.3.5 below.

The purpose of defining the semantics of attributed words in this way is that it allows us to deal with unordered collections of attributed symbols, while remaining consistent with our intuition concerning sequences of procedure calls. Hence we choose to define the interpretations of attributed words as elements of a commutative monoid S whose operator $\oplus$ combines the results of functions independently of the order in which those results might be computed. The identity element 1_S serves two purposes: it defines the interpretation of the empty word, and also us to define interpretation schemes in which certain symbols have no effect on the interpretation of words.

Elements of S can be thought of as images on overhead transparencies. The operator $\oplus$ is like placing a stack of transparencies on the projector: the image projected will not change if the transparencies are stacked in a different order (commutativity), or if any part of the stack is replaced by a single transparency containing the appropriate composite image (associativity). The projector still produces an image (albeit a simple one) even when all the transparencies are removed; this image is analogous to 1_S, the interpretation of the empty word.

The next example suggests a more practical scenario, illustrating how functional interpretations of attributed words might be evaluated in a computer system and peripherals.

Example 10: Functional interpretation

Let $p, q \in \mathbf{N}$ be constants, and let S be the set of all $p \times q$ binary matrices. Let $\oplus$ be a binary operator over S such that, given any two $p \times q$ binary matrices X and Y, $X \oplus Y$ is defined to be the $p \times q$ binary matrix whose (i,j)th element is the logical OR of the (i,j)th elements of X and Y. Let 1_S be the $p \times q$ binary matrix all of whose elements are zero. It is easily verified that $(S, \oplus, 1_S)$ is a commutative monoid.

Elements of S, represented in the form of $p \times q$ binary arrays in a computer, could be interpreted as black-and-white pictures by a device such as a laser printer or CRT controller.

Recall the definitions of $\underline{\mathrm{C}}$, $\underline{\mathrm{E}}$ and A from Example 9. Define the family $F = (d_{\mathrm{dot}}, l_{\mathrm{lineseg}})$ such that d is a $\underline{\mathrm{C}}$-function into S and l is a $\underline{\mathrm{E}}$-function into S, elements of $\mathcal{R}(d)$ would be rendered by some output device as circular dots, and elements of $\mathcal{R}(l)$ would be rendered as line segments. $\varphi = (S, \oplus, 1_s, F)$ is an interpretation scheme for A.

If d and l are defined appropriately, any word $W \in \underline{\mathrm{words}}(\mathrm{A})$ can be interpreted as a picture composed of dots and line segments by sending the representation of $W\|_\varphi$ to the output device. If W is the specific word given in Example 9, $W\|_\varphi$ should render as shown in Figure 4.1.

The approach to interpretation described in the above example is basically the one used by most modern computer graphics systems including TEX, PostScript, and screen drawing libraries such as Xlib (X window system) and QuickDraw (Apple Macintosh computers).

It is impossible to be entirely formal in defining graphical semantics. All we have just defined is a systematic method to interpret a given word as an element of some set S, i.e., to interpret one kind of mathematical object as another kind. This is as far as any system of formal semantics can go.

Informally, we could say "S is the set of all graphics", but since "graphics" are subjective phenomena, there can be no rigorous mathematical definition of them. The example above appeals to the widely understood notion of a graphical output device which can transform a mathematically defined object into a subjectively defined graphic. This is far from rigorous, but for the purposes of the present discussion, it is sufficient.

4.3.3 Non-ground words

The notion of grounding assignment provides the basis for graphical interpretation of non-ground words, i.e., words having some variable attributes. Let A be an attributed alphabet, φ be an interpretation scheme for A, and Σ be a term system with $\underline{\text{sorts}}(\mathrm{A}) = \underline{\text{sorts}}(\Sigma)$. Let W be a non-ground word over A^{Σ}, and let α be a Σ-assignment which grounds W, i.e., $W|_\alpha \in \underline{\text{words}}(\mathrm{A})$. Then, provided $W|_\alpha$ is in the domain of $\|_\varphi$, it has an interpretation $(W|_\alpha)\|_\varphi \in S$ as described above.

> **Example 11: Graphical interpretation via a grounding assignment**
>
> Recall the definitions used in Example 10. Let $V = \{v_1, v_2, \ldots\}$ and $\Sigma = (\mathbf{R}_V)$.
>
> Let $W' = \{\text{vertex}(v_1, v_2), \text{dot}(v_2, v_2), \text{dot}(v_3, v_2), \text{dot}(v_2, v_1), \text{lineseg}(v_1, v_2, v_2, v_2), \text{lineseg}(v_2, v_2, v_3, v_2), \text{lineseg}(v_2, v_2, v_2, v_1), \text{lineseg}(v_2, v_1, v_3, v_2)\}$.
>
> Let $\alpha = \{(v_1, 0), (v_2, 1), (v_3, 2)\}$. Note that $W'|_\alpha$ is ground.
>
> $W'|_\alpha$ is exactly the word W given in Example 10; the chosen output device should thus render $(W'|_\alpha)\|_\varphi$ as shown in Figure 4.1.

In this example, the grounding Σ-assignment can be interpreted as defining the "environment parameters" for the diagram. In typesetting, for example, most aspects of symbol placement are made according to environment parameters such as page width and margins.

In later examples, rather than explicitly giving a grounding assignment and procedure definitions, I shall simply give a picture labelled using the variables from a

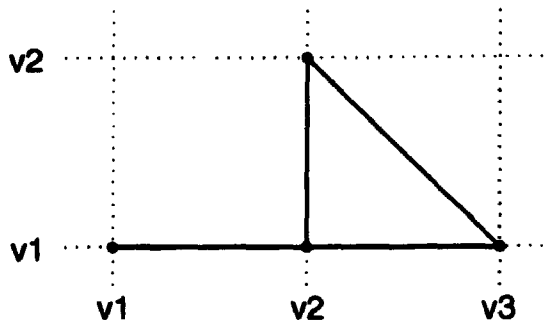


Figure 4.2: How W' might be realized

non-ground word, to suggest how such an assignment and procedures might be defined. E.g., for W' in the above example the picture might be as shown in Figure 4.2.

4.3.4 Constraints

The algebra $(\mathbf{B}, \wedge, 1)$ is a commutative monoid. Let $\mathrm{A} = (\underline{\mathrm{C}}_k)_{k \in K}$ be an attributed alphabet and $\Pi = (\pi_k)_{k \in K}$ be a family of predicates. Then $\beta = (\mathbf{B}, \wedge, 1, \Pi)$ is an interpretation scheme for A, which I call a *Boolean* interpretation scheme. Given any (ground) word $W \in \underline{\text{words}}(\mathrm{A})$, $W\|_\beta$ is a Boolean value (0 or 1).

Now let Σ be a term system with $\underline{\text{sorts}}(\Sigma) = \underline{\text{sorts}}(\mathrm{A})$. Let $W \in \underline{\text{words}}\left(\mathrm{A}^\Sigma\right)$, and α be a Σ-assignment which grounds W. Then $(W|_\alpha)\|_\beta \in \mathbf{B}$. (If α does not ground W, $(W|_\alpha)\|_\beta$ is undefined.)

If we want to have $(W|_\alpha)\|_\beta = 1$, the choice of assignment α is *constrained* by W. Adding additional symbols to W would, in general, constrain the choice still further. Hence we call the symbols *constraints* and W a *system of constraints.* If $(W|_\alpha)\|_\beta = 1$ holds, we call α a *satisfying assignment for W under β*, or say that α *satisfies W under β.* (The "under β" can be omitted if no confusion would result.)

Given $W \in \mathrm{A}^\Sigma$, we might wish to have a satisfying assignment computed automatically, if one exists. This process is called *constraint satisfaction.* Constraint satisfaction algorithms are known for many predicate types.

Example 12: Constraints

Let $\underline{\mathrm{E}} = (\mathbf{R}_a, \mathbf{R}_b, \mathbf{R}_c, \mathbf{R}_x, \mathbf{R}_y)$, $\mathrm{A} = (\underline{\mathrm{E}}_{\text{lineq}})$, $V = \{x, y\}$, $\Sigma = (\mathbf{R}_V)$, and $\beta = (\mathbf{B}, \wedge, 1, \Pi)$ with $\Pi = (\pi_{\text{lineq}})$ and

$$\pi((a, b, c, x, y)) = \begin{cases} 1, & \text{if } ax + by = c; \\ 0, & \text{otherwise.} \end{cases}$$

Under the Boolean interpretation scheme β, ground atributed symbols of class *lineq* represent assertions about linear equations. For example, the attributed symbol lineq$(1,-1,1,2,1)$ represents the (true) assertion that $(2,1)$ is a solution to the equation $x - y = 1$. By making the x and y attributes of such symbols be variables, we can represent linear equations themselves. For example, the word

$$\{\text{lineq}(0,1,1,x,y), \text{lineq}(1,-1,1,x,y)\}$$

represents the system of linear equations

$$\begin{aligned} y &= 1 \\ x - y &= 1. \end{aligned}$$

Any system of this general kind (where there are exactly as many equations as there are variables, and the equations are linearly independent) can be solved using methods from linear algebra. In this case the system is satisfied by the Σ-assignment $\alpha = \{(x,2),(y,1)\}$.

4.3.5 Interpretation of non-ground words via constraints

Section 4.3.2 introduced the notion of functional interpretation, by which we may interpret ground words as elements of some set S, typically corresponding to graphics. Section 4.3.4 introduced Boolean interpretation, by which we may interpret non-ground words as systems of constraints and, given an appropriate constraint satisfaction algorithm, compute a grounding assignment automatically. Putting these two ideas together, we have a systematic means to interpret non-ground words as, e.g., graphics.

Let $\mathrm{A} = (\underline{\mathrm{C}}_k)_{k\in K}$ be an attributed alphabet, Σ be a term system with $\underline{\text{sorts}}(\Sigma) = \underline{\text{sorts}}(\mathrm{A})$, $\varphi = (S, \oplus, 1_S, F)$ with $F = (f_k)_{k\in K}$ be an interpretation scheme for A, and $\beta = (\mathbb{B}, \wedge, 1, \Pi)$ with $\Pi = (\pi_k)_{k\in K}$ be a Boolean interpretation scheme for A. Assume there exists a constraint satisfaction algorithm for the predicates π_k.

Given $W \in \underline{\text{words}}\left(\mathrm{A}^{\Sigma}\right)$, we define the set $S_W \subseteq S$ of interpretations of W as

$$\begin{aligned} S_W = \{s \in S \mid &\text{there exists a } \Sigma\text{-assignment } \alpha \\ &\text{which grounds } W \text{ such that } (W|_\alpha)\|_\beta = 1 \text{ and } s = (W|_\alpha)\|_\varphi\} \end{aligned}$$

The ideal situation (for practical implementation) is that S_W contains exactly one element, i.e., that the interpretation of the word W is unique. It could also arise that $S_W = \emptyset$ or $|S_W| > 1$; these possibilities are discussed in Section 4.3.6 below.

Example 13: Graphical interpretation via constraints

Refer to Figure 4.3. Suppose we want to represent a triangle not by giving the coordinates of its vertices, but rather by giving three line equations which define the sides of the triangle, and stating the relationship these bear to the vertices. We shall do this by defining an extended alphabet with classes *lineseg* and *lineq*. As in Example 4, attributed symbols of class *lineseg* will represent line segments defined by their endpoint coordinates. As in Example 12, symbols of class *lineq* will represent linear constraints.

The particular problem illustrated in Figure 4.3 is captured by six linear equations:

$$\begin{aligned} x_1 + y_1 &= 1 \quad \text{from } L_3 \\ y_1 &= 1 \quad \text{from } L_2 \\ x_2 - y_2 &= 1 \quad \text{from } L_1 \\ x_2 + y_2 &= 1 \quad \text{from } L_3 \\ y_3 &= 1 \quad \text{from } L_2 \\ x_3 - y_3 &= 1 \quad \text{from } L_1 \end{aligned}$$

The solution is $(x_1, y_1) = (0,1)$, $(x_2, y_2) = (1,0)$, $(x_3, y_3) = (2,1)$ as shown in the figure.

Let $\underline{\mathrm{E}} = (\mathbf{R}_{x_1}, \mathbf{R}_{y_1}, \mathbf{R}_{x_2}, \mathbf{R}_{y_2})$, $\underline{\mathrm{F}} = (\mathbf{R}_a, \mathbf{R}_b, \mathbf{R}_c, \mathbf{R}_x, \mathbf{R}_y)$, $K = \{\text{lineseg}, \text{lineq}\}$, $\mathrm{A} = (\underline{\mathrm{E}}_{\text{lineseg}}, \underline{\mathrm{F}}_{\text{lineq}})$, $V = \{x_1, x_2, \ldots, y_1. y_2, \ldots\}$, $\Sigma = (\mathbf{R}_V)$.

Let $\varphi = (S, \oplus, 1_S, F)$, where $F = (f_k)_{k \in K}$, be an interpretation scheme for A. As in Example 10, elements of the set S are intended to be realized as graphics by some device. The function f_{lineseg} is equivalent to the function l in Example 10; it draws (creates the representations of) line segments. The function f_{lineq} is total and maps its entire domain to 1_S; it is a null interpretation for symbols of class *lineq* under φ.

Let $\beta = (\mathbf{B}, \wedge, 1, \Pi)$ where $\Pi = (\pi_k)_{k \in K}$ be a Boolean interpretation scheme for A. The predicate π_{lineseg} is total and maps its entire domain to 1; it is a null interpretation for symbols of class *lineseg* under β. The predicate π_{lineq} is equivalent to the predicate π in Example 12; under β symbols of class *lineq* represent linear constraints.

The six linear equations above can be encoded as

$$W_{\text{lineq}} = \{\text{lineq}(1,1,1,x_1,y_1), \text{lineq}(0,1,1,x_1,y_1), \text{lineq}(1,-1,1,x_2,y_2), \text{lineq}(1,1,1,x_2,y_2), \text{lineq}(0,1,1,x_3,y_3), \text{lineq}(1,-1,1,x_3,y_3)\}.$$

The three line segments forming the triangle can be encoded as

$$W_{\text{lineseg}} = \{\text{lineseg}(x_1,y_1,x_3,y_3), \text{lineseg}(x_3,y_3,x_2,y_2), \text{lineseg}(x_2,y_2,x_1,y_1)\}.$$

Consider the word $W = W_{\text{lineq}} \cup W_{\text{lineseg}}$. A constraint satisfaction algorithm based on Gaussian elimination would suffice to solve the equation $(W|_\alpha)\|_\beta = 1$, yielding the Σ-assignment $\alpha = \{(x_1,0),(y_1,1),(x_2,1),(y_2,0),(x_3,2),(y_3,1)\}$. Note that because the interpretation of symbols of class *lineseg* under β is null, $(W|_\alpha)\|_\beta = (W_{\text{lineq}}|_\alpha)\|_\beta$; the *lineseg* symbols are effectively ignored by the constraint satisfaction algorithm. Substituting according to α, we obtain

$$W|_\alpha = \{\text{lineq}(1,1,1,0,1), \text{lineq}(0,1,1,0,1), \text{lineq}(1,-1,1,1,0), \text{lineq}(1,1,1,1,0), \text{lineq}(0,1,1,2,1), \text{lineq}(1,-1,1,2,1), \text{lineseg}(0,1,2,1), \text{lineseg}(2,1,1,0), \text{lineseg}(1,0,0,1)\}.$$

Because the interpretation of *lineq* symbols under φ is null, the interpretation $(W|_\alpha)\|_\varphi$ (an element of S, which we expect to be realized as a picture similar to Figure 4.3) is the same as $(W_{\text{lineseg}}|_\alpha)\|_\varphi$, i.e., the *lineq* symbols are effectively ignored by the functional interpretation.

Realization of the final interpretation by an output device would result in a graphic consisting of the three heavy lines in Figure 4.3, forming a triangle.

This approach formalizes the mark-setting techniques suggested in my M.Sc. thesis [39]. Symbols with non-null interpretation under φ give rise to visible output and are thus called *marks* in that thesis; those with non-null interpretation under β, which are called *constraints*, determine the attributes of the visible marks, i.e. their position, size, orientation, etc. These two sets of symbols may overlap, reflecting the fact that some classes of mark may have intrinsic constraints, e.g. a "horizontal line" mark class would by definition constrain its two endpoints to have the same y-coordinate.

The appearance of marks is fixed by the functions in the family F. The semantics of constraints is fixed by the predicates in the family Π. The syntax of both marks

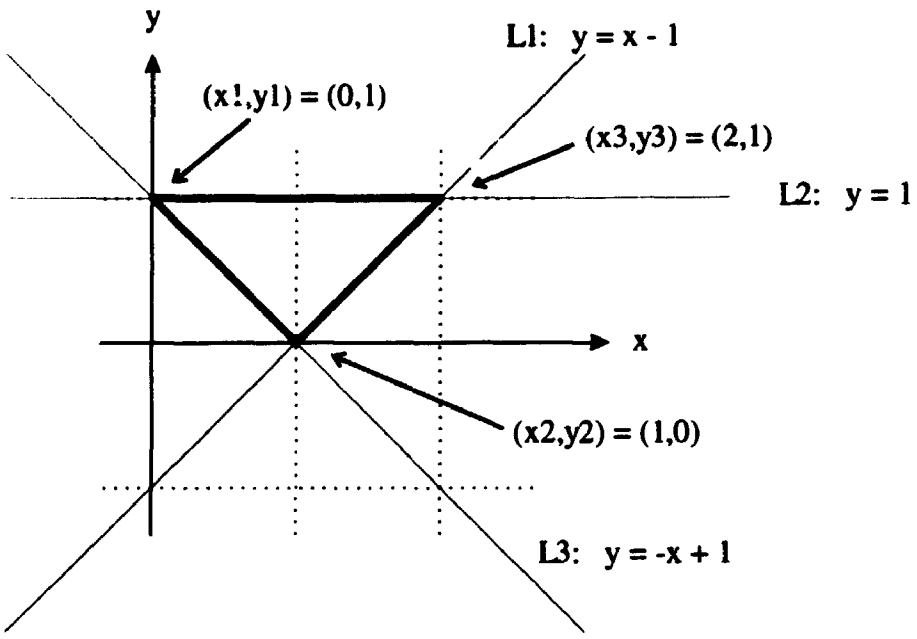


Figure 4.3: A triangle defined by three line equations

and constraints is fixed by the attributed alphabet A; the term system Σ extends this basic syntax to permit the use of variables.

4.3.6 Solvable, over- and under-constrained systems

Recall the definitions of A, Σ, φ and β in Section 4.3.5 above and let $W \in \underline{\text{words}}(A)$. If W has *at least one* satisfying assignment, it is called *satisfiable.*

A Σ-assignment α is *minimal* for W if $\mathcal{D}(\alpha) = \underline{\text{variables}}(W)$.

Lemma 8 *If W has at least one satisfying assignment, then it has at least one minimal satisfying assignment.*

Proof: Let α be a Σ-assignment which satisfies W. Let α' be the restriction of α to $\underline{\text{variables}}(W)$. Then α' also satisfies W and is minimal. □

If W has *exactly one* minimal satisfying assignment, it is called *solvable*. Constraint-satisfaction algorithms (called constraint *solvers* in my M.Sc. thesis) are easiest to apply to solvable systems, because there is no need to choose among several (perhaps infinitely many) candidate satisfying assignments.

The ideal situation is that we have a constraint solver capable of reasoning about the predicates Π, and that W is solvable with minimal satisfying assignment α. The constraint solver generates α automatically, and $W|_\alpha$ follows.

Assuming the existence of an appropriate constraint solver, W may not be solvable for one of two reasons:

1. W may be unsatisfiable.

2. W may be satisfiable but not solvable, i.e. it may admit multiple minimal satisfying assignments.

In case 1 we say W is *overconstrained*; in case 2 we say it is *underconstrained*. If W is overconstrained, it might possibly be made solvable by removing or modifying one or more constraints; this possibility will not be considered further.[4]

If W is underconstrained, it may possibly be made solvable by adding constraints. Without precise knowledge about the predicates Π and their properties, we cannot give an algorithm to determine which additional constraints should be added. We can at least note, however, that symbols added to W should be pure constraints, i.e., they should have null interpretations under φ.

Practical graphics-generating software would typically produce underconstrained representations, providing the minimal number of constraints required to express the shape of the eventual output picture, but leaving certain aspects such as overall size or orientation as variables. The additional constraints needed to produce a solvable system would normally be determined by the *environment*, i.e. details such as page size and margins, amount of space required by other graphics, etc. This is analogous to the TEX typesetting system, in which subsets of the input define graphical units with internal constraints ("boxes"), whose final position and size on the output page are determined by constraints added in the final phase of processing ("output routine").

To summarize, graphical interpretation of the word W involves four phases:

1. Choose a set W' of additional constraints, having null interpretation under φ, such that the system $W \cup W'$ is solvable. (If W is already solvable, $W' = \emptyset$.) If there can be no such set W', W is overconstrained —stop.

2. Solve $((W \cup W')|_\alpha)\|_\beta = 1$ to obtain α.

3. Substitute, letting $W'' = (W \cup W')|_\alpha$.

4. Realize $W''\|_\varphi$ as a graphic by passing it to some output device.

[4]Overconstrained systems often arise with the TEX typesetting system; that is when it issues warnings concerning "overfull (or underfull) boxes".

Note that because the interpretation of W' under φ is null, we could say W in place of $(W \cup W')$ in phase 3.

Example 14: Interpretation with added constraints

Refer to Figure 4.4. Suppose we want to represent the fact that a rectangular box aligned with the coordinate axes consists of four line segments joining the vertices, which are aligned horizontally and vertically as in the figure. We shall do this by defining an extended alphabet with classes *lineseg* and *equal*. As in Examples 4 and 13, attributed symbols of class *lineseg* will represent line segments defined by their endpoint coordinates. Symbols of class *equal* will represent equality constraints, e.g. equal(x_1, x_2) would mean $x_1 = x_2$.

The representation of the box will look like this (details below):

$$\begin{aligned} W \;=\; \{\, &\text{lineseg}(x_1, y_1, x_2, y_2), \text{lineseg}(x_2, y_2, x_3, y_3), \\ &\text{lineseg}(x_3, y_3, x_4, y_4), \text{lineseg}(x_4, y_4, x_1, y_1), \\ &\text{equal}(x_1, x_4), \text{equal}(x_2, x_3), \text{equal}(y_1, y_2), \text{equal}(y_3, y_4) \,\} \end{aligned}$$

This captures the rectangular shape of the box and its orientation with respect to the coordinate axes, but there aren't enough constraints to determine the actual point coordinates. For this example we shall fix the coordinates of the upper-left and lower-right vertices, by adding the additional constraints

$$W' = \{\, \text{equal}(x_1, 10), \text{equal}(y_1, 30), \text{equal}(x_3, 50), \text{equal}(y_3, 10) \,\}$$

Adding these constraints yields a solvable system (eight independent equations in eight unknowns). The solution is illustrated in Figure 4.4.

Let $\underline{\mathrm{C}} = (\mathbf{R}_x, \mathbf{R}_y)$, $\underline{\mathrm{E}} = (\mathbf{R}_{x_1}, \mathbf{R}_{y_1}, \mathbf{R}_{x_2}, \mathbf{R}_{y_2})$, $K = \{\text{lineseg}, \text{equal}\}$, $\mathrm{A} = (\underline{\mathrm{E}}_{\text{lineseg}}, \underline{\mathrm{C}}_{\text{equal}})$, $V = \{x_1, x_2, \ldots, y_1, y_2, \ldots\}$, and $\Sigma = (\mathbf{R}_V)$. Let $\beta = (\mathbf{B}, \wedge, 1, \Pi)$ with $\Pi = (\pi_k)_{k \in K}$ be a Boolean interpretation scheme for A. Let the π_k be total with

$$\begin{aligned} \pi_{\text{lineseg}}(x_p, y_p, x_q, y_q) &= 1 \\ \pi_{\text{equal}}(x, y) &= \begin{cases} 1, & \text{if } x = y; \\ 0, & \text{otherwise} \end{cases} \end{aligned}$$

Let $\varphi = (S, \oplus, 1_S, F)$ with $F = (f_k)_{k \in K}$ be a functional interpretation scheme for A, where elements of the set S can be realized as graphics by some output device as in Example 10. Let the f_k be total with f_{lineseg} drawing line segments as in Example 10 and $f_{\text{equal}}((x, y)) = 1_S$ for all $(x, y) \in \underline{\text{instances}}(\underline{\text{C}})$.

Let W be as above. The process of interpreting W occurs in four phases, as follows.

Phase 1: We observe that W is satisfiable but not solvable. Let W' be as above, expressing additional constraints to fix the coordinates of the upper left and lower right vertices of the box. We have

$$\begin{aligned} W \cup W' \;=\; \{ &\text{lineseg}(x_1, y_1, x_2, y_2), \text{lineseg}(x_2, y_2, x_3, y_3), \\ &\text{lineseg}(x_3, y_3, x_4, y_4), \text{lineseg}(x_4, y_4, x_1, y_1), \\ &\text{equal}(x_1, x_4), \text{equal}(x_2, x_3), \text{equal}(y_1, y_2), \text{equal}(y_3, y_4), \\ &\text{equal}(x_1, 10), \text{equal}(y_1, 30), \text{equal}(x_3, 50), \text{equal}(y_3, 10) \} \end{aligned}$$

Phase 2: Solve $((W \cup W')|_\alpha)\|_\beta = 1$ yielding assignment α:

$$\alpha = \{ (x_1, 10), (y_1, 30), (x_2, 50), (y_2, 30), (x_3, 50), (y_3, 10), (x_4, 10), (y_4, 10) \}$$

Phase 3: Substitute to obtain

$$\begin{aligned} W'' = (W \cup W')|_\alpha \;=\; \{ &\text{lineseg}(10, 30, 50, 30), \text{lineseg}(50, 30, 50, 10), \\ &\text{lineseg}(50, 10, 10, 10), \text{lineseg}(10, 10, 10, 30), \\ &\text{equal}(10, 10), \text{equal}(50, 50), \text{equal}(30, 30) \} \end{aligned}$$

Note that the set has become smaller. This is because the substitution makes several distinct symbols of class *equal* identical.

Phase 4: Send $W''\|_\varphi$ to the output device, yielding a picture as illustrated in Figure 4.4.

4.4 Attributed set grammars

Having developed attributed analogues of the conventional language-theoretic notions of symbol, word, and language, I now develop a grammar formalism through

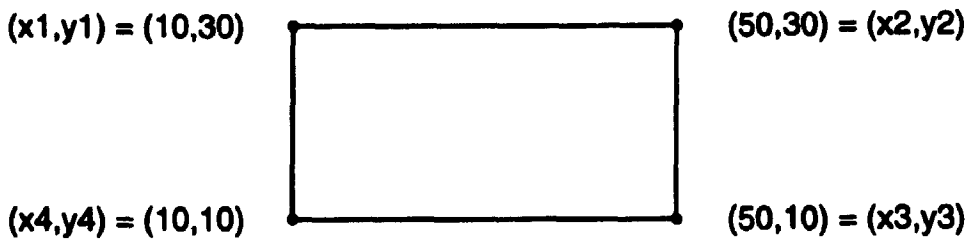


Figure 4.4: Rectangular box

which attributed languages may be generated and recognized. Since the elements of attributed languages (attributed words) are themselves sets, I call the formalism *attributed set grammars.*

The purpose of this section is to create a theoretical basis for the design of software to generate and recognize notations. Hence I do not attempt to be rigorous in my analysis of the new formalism, though I do investigate a few of its properties in order to show its applicability to problems involving notations. Detailed theoretical investigation of the formalism is beyond the scope of this dissertation, and is hence left for the future.

4.4.1 Attributed set rewriting

Let A and B be attributed alphabets, and let Σ be a term system with $\underline{\text{sorts}}(\Sigma) = \underline{\text{sorts}}(\mathrm{A} \cup \mathrm{B})$. A $(\mathrm{A}, \mathrm{B}, \Sigma)$ *rewrite rule* is a pair $\rho = (X, Y)$ where $X \in \underline{\text{words}}\left(\mathrm{A}^{\Sigma}\right)$ and $Y \in \underline{\text{words}}\left(\mathrm{B}^{\Sigma}\right)$. The word X is called the *head* of the rule, Y the *body.* A rewrite rule $\rho = (\{x_1, \ldots, x_m\}, \{y_1, \ldots, y_n\})$ (where the x_i and y_i are attributed symbols) is usually denoted $\rho: \ x_1, \ldots, x_m \rightarrow y_1, \ldots, y_n$.

Let $W \in \underline{\text{words}}(\mathrm{A} \cup \mathrm{B})$, and $\rho = (X, Y)$ be a $(\mathrm{A}, \mathrm{B}, \Sigma)$ rewrite rule. A Σ-assignment α is said to be *admissible for ρ in context of W* if the following conditions hold:

1. $\underline{\text{variables}}(X \cup Y) \subseteq \mathcal{D}(\alpha)$
2. $|X|_\alpha| = |X|$
3. $|Y|_\alpha| = |Y|$
4. $Y|_\alpha \cap (W \setminus X|_\alpha) = \emptyset$

If such an α exists, and $X|_\alpha \subseteq W$, ρ is said to be *applicable to W*, and the process of *applying ρ to W under α* rewrites the subset $X|_\alpha$ of W as the set $Y|_\alpha$, yielding a

new word $W' = W \setminus X|_\alpha \cup Y|_\alpha$. We say W *rewrites to* W' *via* ρ *under* α, denoted $W \underset{\alpha}{\overset{\rho}{\Longrightarrow}} W'$.

Condition 1 above simply requires that all variables occurring in a rule be bound to values via the assignment. Hence the attributed set rewriting process cannot introduce variables where none previously existed. As the following example illustrates, conditions 2, 3, and 4 above ensure that $|W'| = |W| - |X| + |Y|$.

Example 15: attributed set rewriting

Let $\underline{\mathrm{C}} = (\mathbb{N}_{\mathrm{id}})$, $\mathrm{A} = (\underline{\mathrm{C}}_a)$, $\mathrm{B} = (\underline{\mathrm{C}}_a, \underline{\mathrm{C}}_b)$, $V = \{v_1, v_2, \ldots\}$ and $\Sigma = (\mathbb{N}_V)$. Consider the $(\mathrm{A}, \mathrm{B}, \Sigma)$ rewrite rules

$$\begin{aligned} \rho_1 : \quad a(v_1), a(v_2) &\rightarrow a(v_1) \\ \rho_2 : \quad a(v_1) &\rightarrow a(v_1), a(v_2) \\ \rho_3 : \quad a(v_1) &\rightarrow a(v_1), b(v_2) \end{aligned}$$

and the Σ-assignments

$$\begin{aligned} \alpha_1 &= \{(v_1, 1), (v_2, 1)\} \\ \alpha_2 &= \{(v_1, 1), (v_2, 2)\} \\ \alpha_3 &= \{(v_1, 1), (v_2, 3)\}. \end{aligned}$$

Observe the following:

- α_2 is admissible for ρ_1 in context of $\{a(1), a(2)\}$, and $\{a(1), a(2)\} \underset{\alpha_2}{\overset{\rho_1}{\Longrightarrow}} \{a(1)\}$. If condition 2 (of the four conditions of admissibility stated above) were not required, α_1 would also be admissible, the matching of the head of ρ_1 would collapse, and we would have $\{a(1), a(2)\} \underset{\alpha_1}{\overset{\rho_1}{\Longrightarrow}} \{a(1), a(2)\}$. The intent of ρ_1—to turn two distinct symbols into one—would thus be violated.
- α_2 is admissible for ρ_2 in context of $\{a(1)\}$, and $\{a(1)\} \underset{\alpha_2}{\overset{\rho_2}{\Longrightarrow}} \{a(1), a(2)\}$. If condition 3 were not required, α_1 would also be admissible, the matching of the tail of ρ_2 would collapse, and we would have $\{a(1)\} \underset{\alpha_1}{\overset{\rho_2}{\Longrightarrow}} \{a(1)\}$. The intent of ρ_2—to turn one symbol into two distinct symbols—would thus be violated.
- α_3 is admissible for ρ_2 in context of $\{a(1), a(2)\}$, and $\{a(1), a(2)\} \underset{\alpha_3}{\overset{\rho_2}{\Longrightarrow}} \{a(1), a(2), a(3)\}$. If condition 4 were not required, α_2 would also

be admissible, the substitution would collapse, and we would have $\{a(1), a(2)\} \underset{\alpha_2}{\overset{\rho_2}{\Longrightarrow}} \{a(1), a(2)\}$. Again the intent of ρ_2 would be violated.

- α_1, α_2 and α_3 are all admissible for ρ_3 in context of $\{a(1)\}$, but in context of $\{a(1), b(1)\}$ only α_2 and α_3 are admissible.

From the first three observations in the above example, we see that the purpose of admissibility conditions 2, 3, and 4 is to ensure that cardinality is affected by rewriting in a manner consistent with intuition. In string rewriting we speak of "length-increasing" (preserving, decreasing) rules; in attributed set rewriting we have the corresponding notion "cardinality-increasing" (preserving, decreasing).

The fourth observation in the example illustrates that, when an attribute variable appears in the body of a rule, but not in the head, the value assigned to that variable is chosen freely, subject only to the conditions of admissibility. The distinct variables v_1 and v_2 appearing in rule ρ_3 need not be assigned distinct values in every case (witness the case $\{a(1)\} \underset{\alpha_1}{\overset{\rho_3}{\Longrightarrow}} \{a(1), b(1)\}$), i.e., there is no inherent requirement that admissible assignments be injective. The only requirement is that distinct *symbols* in the rule correspond to distinct symbols in the words W and W'.

4.4.2 Attributed set grammars

The preceding section defined rewrite rules for attributed sets in a very general way. From now on we are concerned exclusively with rules of a certain form, called *productions*. Let N and T be disjoint attributed alphabets, and Σ be a term system with $\underline{\text{sorts}}(\Sigma) = \underline{\text{sorts}}(\mathrm{N} \cup \mathrm{T})$. A $(\mathrm{N}, \mathrm{T}, \Sigma)$-*production* is a $(\mathrm{N} \cup \mathrm{T}, \mathrm{N} \cup \mathrm{T}, \Sigma)$ rewrite rule $X \rightarrow Y$ whose head X contains at least one symbol drawn from N^Σ, i.e., $X \in \underline{\text{words}}\left((\mathrm{N} \cup \mathrm{T})^\Sigma \setminus \mathrm{T}^\Sigma\right)$.

An *attributed set grammar* or ASG is a tuple $(\mathrm{T}, \mathrm{N}, \Sigma, W_0, P)$ where

1. T and N are disjoint attributed alphabets, called the *terminal* and *nonterminal* alphabets respectively.

2. Σ is a term system with

   (a) $\underline{\text{sorts}}(\Sigma) = \underline{\text{sorts}}(\mathrm{T} \cup \mathrm{N})$, i.e., the extended alphabets T^Σ and N^Σ are defined.

(b) $\cup \underline{\text{sorts}}(\Sigma) \cap \cup \underline{\text{variables}}(\Sigma) = \emptyset$, i.e., we can always tell constants and variables apart.

3. $W_0 \in \underline{\text{words}}(\mathrm{N})$ is the *start word*, which must consist of a single symbol.

4. P is a finite set of $(\mathrm{N}, \mathrm{T}, \Sigma)$-productions.

Attributed symbols drawn from T^Σ are called "symbols of terminal class", "terminal symbols", or "terminals". Attributed symbols drawn from N^Σ are called "symbols of nonterminal class", "nonterminal symbols", or "nonterminals". An ASG is a mechanism for generating (ground) words over T. The nonterminals are simply any other symbols which may be needed in the generation process.

A sequence

$$W_0 \underset{\lambda 1}{\overset{p_1}{\Longrightarrow}} W_1 \underset{\lambda 2}{\overset{p_2}{\Longrightarrow}} W_2 \underset{\lambda 3}{\overset{p_3}{\Longrightarrow}} \cdots \underset{\lambda k}{\overset{p_k}{\Longrightarrow}} W_n$$

is an (n-step) *derivation (of the word W_n) in* Γ. For any W_i, W_j with $i \leq j$ in such a sequence, we say W_i *derives* W_j *in* $j - i$ *steps*, denoted $W_i \overset{j-i}{\Longrightarrow} W_j$. When the precise number of steps is unimportant we write $W_i \overset{*}{\Longrightarrow} W_j$ meaning "W_i derives W_j in zero or more steps." If we know $W_i \neq W_j$ we may write $W_i \overset{+}{\Longrightarrow} W_j$ meaning "W_i derives W_j in one or more steps."

The *language of the ASG* Γ, denoted $L(\Gamma)$, is defined to be

$$L(\Gamma) = \{W \in \underline{\text{words}}(\mathrm{T}) \mid W_0 \overset{*}{\Longrightarrow} W \text{ under } \Gamma\}.$$

Example 16: ASG for non-empty sets

Let $\underline{\mathrm{C}} = (\mathbf{N}_{\mathrm{id}})$ be a class, and $\mathrm{T} = (\underline{\mathrm{C}}_t)$ and $\mathrm{N} = (\underline{\mathrm{C}}_n)$ be (resp. terminal and nonterminal) attributed alphabets. Words over T (terminal words) are comprised of 0 or more symbols of class t, distinguished only by their "id" attribute. Words over N (nonterminal words) are comprised of 0 or more symbols of class t, distinguished only by their "id" attribute.

Let $V = \{v_1, v_2, \ldots\}$ be a variable domain and $\Sigma = (\mathbf{N}_V)$ be a term system. Any Σ-assignment binds values in $\mathbf{N}$ to variables in V.

Consider the ASG $\Gamma_{\mathrm{nes}} = (\mathrm{T}, \mathrm{N}, \Sigma, W_0, P)$ where $W_0 = \{n(1)\}$ and P consists of the productions

$$\begin{aligned} p_1 : \quad & n(v_1) \rightarrow n(v_1), n(v_2) \\ p_2 : \quad & n(v_1) \rightarrow t(v_1). \end{aligned}$$

Consider the Σ-assignments $\alpha_1 = \{(v_1, 1), (v_2, 6)\}$ and $\alpha_2 = \{(v_1, 6)\}$, and observe that

$$\{n(1)\} \underset{\alpha_1}{\overset{p_1}{\Longrightarrow}} \{n(1), n(6)\} \underset{\alpha_1}{\overset{p_2}{\Longrightarrow}} \{t(1), n(6)\} \underset{\alpha_2}{\overset{p_2}{\Longrightarrow}} \{t(1), t(6)\}$$

is a valid 3-step derivation of the word $\{t(1), t(6)\}$ in Γ_{nes}. It is easy to verify that

$$L(\Gamma_{\text{nes}}) = \{W \in \underline{\text{words}}(\text{T}) \mid W \neq \emptyset \text{ and } n(1) \in W\}.$$

In later examples, derivations are written vertically, set braces are omitted, and assignments are written beside the derivation arrows as lists of equations of the form *variable = value*. For greater clarity, the symbol(s) in each word which are rewritten at the next derivation step are underlined. For instance, the derivation in example 16 above would be written as

$$\begin{array}{l} \underline{n(1)} \\ \quad p_1 \Downarrow v_1 = 1, v_2 = 6 \\ \underline{n(1)}, n(6) \\ \quad p_2 \Downarrow v_1 = 1 \\ t(1), \underline{n(6)} \\ \quad p_2 \Downarrow v_1 = 6 \\ t(1), t(6) \end{array}$$

In example 16, all words in $L(\Gamma)$ were bound to contain one symbol with the attribute value 1, since that value appeared explicitly in the start word and neither of the productions was capable of eliminating an attribute value. In contrast, in the derivation above, the value 6 was chosen freely at the first step; any other value (except 1, which was already fixed) could have been chosen instead. The next example shows how we can ensure that all attribute values are chosen freely, and also illustrates how an ASG can generate the empty word $\emptyset$.

Example 17: free generation of sets

Recall the definitions of $\underline{\text{C}}$, T, V and Σ from example 16. Observe that $\emptyset$ can be considered a class (whose index set is $\emptyset$), and hence can be a member of an attributed alphabet just like any other class. Let us define a new nonterminal alphabet $N' = (\emptyset_n)$. The only symbol which can be drawn from N' has no attributes and is denoted $n()$.

Consider the ASG $\Gamma_{\text{set}} = (\mathrm{T}, \mathrm{N}', \Sigma, W_0', P')$ where $W_0' = \{n()\}$ and P' contains the productions

$$\begin{aligned} p_1' : \quad & n() \rightarrow \emptyset \\ p_2' : \quad & n() \rightarrow n(), t(v_1). \end{aligned}$$

It is easy to verify that $L(\Gamma_{\text{set}}) = \underline{\text{words}}(\mathrm{T})$. Γ_{set} can generate the empty word $\emptyset$ via the 1-step derivation

$$\begin{array}{c} \underline{n()} \\ p_1' \Downarrow \\ \emptyset \end{array}$$

and words not containing $t(1)$, e.g.

$$\begin{array}{c} \underline{n()} \\ p_2' \Downarrow v_1 = 7 \\ \underline{n()}, t(7) \\ p_2' \Downarrow v_1 = 8 \\ \underline{n()}, t(7), t(8) \\ p_1' \Downarrow \\ t(7), t(8) \end{array}$$

4.4.3 ASG vs. other formalisms

Compare the ASG formalism with some of the formalisms described in chapter 3.

- Like all of the formalisms described in chapter 3, ASGs use attributes.
- Like all but Kojima and Myers's GFG, the structures generated by ASGs (attributed sets) are inherently unordered.
- Unlike Golin's AMG and Helm and Marriott's CMG, ASGs generate sets rather than multisets.[5]

[5]When elements are atomic, multisets are fundamentally different from sets. When symbols are attributed, however, the extra information (element multiplicity) in multisets can always be encoded in attributes. For example, we can add a distinguishing attribute to symbols as in example 16. Another possibility is to add a $\mathbb{N}$-valued attribute to each class to represent element multiplicity directly.

- Like Najork and Kaplan's CSRS, ASGs permit any number of symbols in the heads and bodies of productions. As will be shown in a later example, this makes it very easy to define an ASG to generate representations of directed graphs.

- Unlike all of the formalisms described in chapter 3, ASGs do not use constraints, and cannot compute new attribute values via expressions.

I do not claim that the ASG formalism is superior to more powerful formalisms such as AMG, CMG, and CSRS. I do believe, however, that the inherent simplicity of the ASG formalism will facilitate analysis. Furthermore, I would hope that a detailed analysis of the properties of ASGs might yield insight into the more complex (multi)set-generating formalisms. The latter might be expected to inherit properties of ASGs which inhere in the generation of sets rather than sequences, independent of the computing power added by attribute expressions and constraints.

4.5 Modelling with attributed sets and ASGs

I have already given a few very simple examples showing how attributed sets (i.e., attributed words) may be used to model structures such as character strings and geometrical figures. In this section I present larger examples to show that attributed sets may be used to represent or "model" useful algebraic and notational structures, and that furthermore, ASGs may be defined to generate such representations. The algebraic structure examples include strings, directed graphs, and binary trees. This is followed by a notation example: binary tree diagrams layout constraints.

4.5.1 Semantics of modelling

The term *model* is often used to refer to an algebraic structure chosen to represent the salient characteristics of an object, situation, process, etc. in human experience. For the purposes of this discussion, however, I adopt a much more restricted definition. A *model* is a algebraic structure, which serves as a convenient and (ideally) unambiguous representation for some other algebraic structure, which I call an *object*. I restrict my attention to finite models which are attributed words.

A completely rigorous definition of the notions of objects and models is beyond the scope of this dissertation; I instead use examples to make my ideas clear. However

let me suggest that a rigorous definition of the notions would involve (at least) the following:

1. a well-defined class[6] O of objects, and an equivalence relation $\approx$ on O.

2. an attributed alphabet T and a language $L_M \subseteq \underline{\text{words}}(\text{T})$ whose elements (which we shall call *models*) are to be interpreted as representations of objects $o \in O$.

3. a well-defined *encoding relation* $\mathcal{M} \subseteq O \times L_M$, and a well-defined *decoding relation* $\mathcal{R} \subseteq L_M \times O$, such that

$$\forall o \in O,\ \forall (o, W'), (o, W'') \in \mathcal{M},\ \forall (W', o'), (W'', o'') \in \mathcal{R},\ o \approx o' \approx o''.$$

The key point is that the encoding and decoding processes (captured mathematically by the relations $\mathcal{M}$ and $\mathcal{R}$) are unlikely to be one-to-one, and some information is sure to be lost in encoding and/or decoding. Item 3 above formalizes the notion that it should be possible to encode an object and subsequently decode it, such that no *significant* information is lost. The notion of "significant information" is formalized by the equivalence relation in item 1. Item 2 establishes that I am interested only in finite attributed set encodings.

4.5.2 Character strings

Example 5 suggested a simple way to model strings using attributed sets, by augmenting each character with a single $\mathbb{N}$-valued attribute indicating its ordinal position in the string, starting with 1. We cannot define an ASG to generate such representations and nothing else, because in attributed set rewriting, attribute values are always freely chosen (see section 4.4.1).

A different representation for strings, which is more amenable to generation by an ASG, is based on the familiar linked list data structure. To simulate pointers in attributed sets, we can give each symbol two integer attributes *self* and *next*. The value of *self* will be a unique identifier for the symbol and that of *next* will be a copy of the unique identifier for the next symbol in the sequence. We also require one special value—analogous to a nil pointer—which must be different from all valid

[6] In this context, the word "class" is not used in the special sense introduced in this chapter, but in the usual mathematical sense.

Figure 4.5: A "linked list" model of the string *abba*

identifiers. Let us use the symbol ♯ to denote this special value, and define the set $\mathbf{N}^{\sharp} = \mathbf{N} \cup \{\sharp\}$ as the domain for *self* and *next* attributes.

Let V be an atomic alphabet, and define the class $\underline{C} = (V_{\text{letter}}, \mathbf{N}^{\sharp}_{\text{self}}, \mathbf{N}^{\sharp}_{\text{next}})$, and the attributed alphabet $\mathrm{T} = (\underline{C}_t)$. We shall model strings over V as attributed words over T. Suppose $V = \{a, b\}$. The string *abba* could be represented by the word

$$\{t(a, 20, 21), t(b, 21, 9), t(b, 9, 100), t(a, 100, \sharp)\}.$$

or indeed, any other word having the same structure. The absolute values of the "pointer" attributes mean nothing; all that matters is that they encode the sequential structure of the string being modelled. The pointer structure for this example is easier to understand when shown in diagram form, as in figure 4.5.

A shortcoming of this model representation is that it is not easy to tell which symbol represents the first letter in the string.

Now suppose we want to model substring rewriting rules of the form $p_1 \cdots p_n \rightarrow q_1 \cdots q_m$, directly on the model representation using attributed set rewriting rules. We define two sets of variables $L = \{p_0, p_1, \ldots, q_0, q_1, \ldots\}$ and $U = \{l, r, u_0, u_1, \ldots, v_0, v_1, \ldots\}$, and a term system $\Sigma = (V_L, \mathbf{N}^{\sharp}_U)$. We can model the substring rewriting rule $p_1 \cdots p_n \rightarrow q_1 \cdots q_m$, as the (T, T, Σ) rewriting rule

$$t(p_1, l, u_1), t(p_2, u_1, u_2), \ldots, t(p_n, u_{n-1}, r) \rightarrow t(q_1, l, v_1), t(q_2, v_1, v_2), \ldots, t(q_n, v_{m-1}, v).$$

The use of the distinguished variables l and r takes care of pointer manipulations. The use of the "u" variables in the head ensures that what is being replaced is indeed the representation of a substring. The use of the "v" variables in the body ensures that the replacement is also the representation of a substring. The use of completely different variables in head and body ensure that new "pointer attribute" values are always freely generated, and are guaranteed to be different from others already in use.[7]

A more serious shortcoming of the model representation becomes apparent when we consider modelling "erasing rules" of the form $p_1 \cdots p_n \rightarrow \varepsilon$, because the pointer

[7]This is analogous to what happens in a Lisp system with automatic garbage collection.

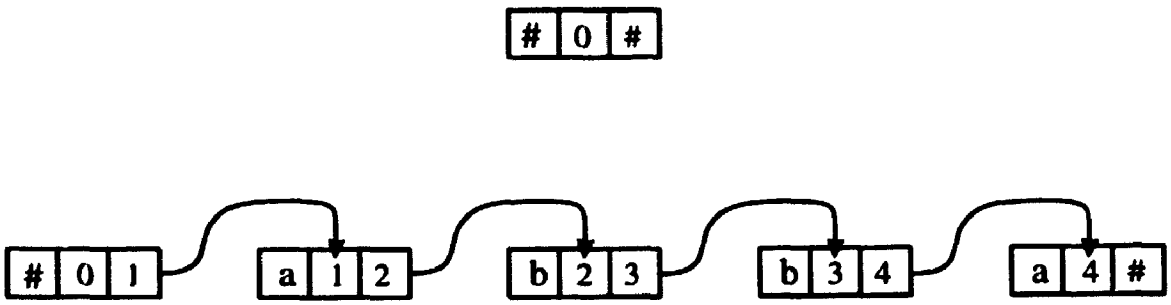


Figure 4.6: Revised linked list models of strings ϵ and *abba*

manipulation required would involve modifying the *next* attribute of the symbol representing the letter "to the left of p_1", i.e. the symbol whose *next* attribute value equals the *self* attribute value for the symbol which represents p_1.

We can solve this problem by modifying our "linked list" representation to use a "dummy link" at the beginning of the list. We define $V^\sharp = V \cup \{\sharp\}$, re-define $\underline{\mathbf{C}}$ as $(V^\sharp_{\text{letter}}, \mathbf{N}^\sharp_{\text{self}}, \mathbf{N}^\sharp_{\text{next}})$ (and re-define T and Σ accordingly), and take steps to ensure that, in the model representation of a string, there is always exactly one symbol whose *letter* attribute is the special value $\sharp$. Coincidentally, this also solves the problem of finding the beginning of the string in the attributed set representation—we simply look for this symbol and follow its *next* pointer.

Figure 4.6 illustrates the structure of the attributed sets $\{t(\sharp, 0, \sharp)\}$, which is a model of the empty string ϵ, and $\{t(\sharp, 0, 1), t(a, 1, 2), t(b, 2, 3), t(b, 3, 4), t(a, 4, \sharp)\}$, which is a model of the string *abba*. I say "a model" rather than "the model" because the model representations are not unique. The exact values of the *self* and *next* attributes are unimportant; only their relationships (which model the linked-list structure) matter.

With the revised representation, the substring-erasing rule $p_1 \cdots p_n \rightarrow \epsilon$ can be modelled by the (T, T, Σ) rewriting rule

$$t(p_0, l, u_1), t(p_1, u_1, u_2), \ldots, t(p_n, u_n, r) \rightarrow t(p_0, l, r).$$

Note that there are $n + 1$ symbols in the head of this rule, while there are only n letters in the head of the corresponding substring rule. The extra symbol (the first one listed above) matches the attributed symbol which represents the letter immediately preceding p_1 in the string, or the "dummy link" symbol if there is no such letter). This symbol gets rewritten so as to maintain the proper pointer structure.

Suppose we are given a set L of strings to model, and we know that $L \subseteq V^*$ for

some atomic alphabet V. In terms of the semantics defined at the beginning of this section, we can say the following about the encoding scheme just defined:

1. The object class O is the set L, and the equivalence relation $\approx$ is ordinary equality.

2. The model alphabet is T and the language $L_M \subseteq \underline{\text{words}}(\mathrm{T})$ of valid models is

$$\Big\{W \quad \in \underline{\text{words}}(\mathrm{T}) \;\Big|\; \exists n \in \mathbf{N}^\sharp,\ t(\sharp, 0, n) \in W \text{ and } \forall l_1 \in V^\sharp, \forall n_1 \in \mathbf{N}, \forall n_2 \in \mathbf{N}^\sharp,$$
$$\text{if } t(l_1, n_1, n_2) \in W$$
$$\text{then } n_1 \neq n_2 \text{ and } \neg(\exists l_2 \in V^\sharp \exists n_3 \in \mathbf{N}^\sharp,\ t(l_2, n_1, n_3) \in W)$$
$$\text{and either } n_2 = \sharp \text{ or } \exists l_2 \in V, \exists n_3 \in \mathbf{N}^\sharp,\ t(l_2, n_2, n_3) \in W\Big\}.$$

3. I shall not attempt to give a rigorous mathematical definition of the encoding relation $\mathcal{M} \subseteq O \times L_M$ and the decoding relation $\mathcal{R} \subseteq L_M \times O$. However the foregoing discussion of the encoding technique should suffice to suggest that both relations are well-defined, and moreover, that encoding followed by decoding results in no information loss whatever.[8]

Now suppose that the set L of strings to be modelled is defined by a string-generating grammar G ($L = L(G)$). I show that given G we can, by a simple procedure, define an ASG Γ_G which is equivalent to G in the sense that

$$\forall s \in L(G),\ \mathcal{M}(s) \subseteq L(\Gamma_G)$$

and

$$\forall W \in L(\Gamma_G).\ \mathcal{R}(W) \in L(G).$$

Before continuing let us briefly review the definition of a string-generating grammar; I adapt the definition given by Salomaa [121, page 9].

A string-generating grammar (SGG) is a tuple $G = (V_N, V_T, X_0, F)$ where

- V_N is an atomic alphabet of *nonterminal letters*
- V_T is an atomic alphabet of *terminal letters*

[8] $\mathcal{M}$ is one-to-many; this is because there is infinite choice concerning the precise values of the *self* and *next* attributes. However $\mathcal{R}$ is many-to-one, and in fact $\mathcal{R} \circ \mathcal{M}$ is an identity. This is only possible because the atomic alphabet V, which defines the entire range of possible objects, is used in the definition of the model language L_M.

- $X_0 \in V_N$ is the *initial letter*
- F is a finite set of substring rewriting rules of the form $p_1 \cdots p_n \rightarrow q_1 \cdots q_m$ where $p_i, q_j \in V_N \cup V_T$, $1 \leq i \leq n$, $1 \leq j \leq m$, and at least one of the p_i is an element of V_N.

Let $V = V_N \cup V_T$. Given two strings $s_1, s_2 \in V^*$, we write $s_1 \Longrightarrow_G s_2$ if $s_1 = l_1 p_1 \cdots p_n r_1$ and $s_2 = l_2 q_1 \cdots q_m r_2$ where $l_1, r_1, l_2, r_2 \in V^*$ and $p_1 \cdots p_n \rightarrow q_1 \cdots q_m \in F$. $\Longrightarrow_G$ is a relation on V^*; its reflexive transitive closure is denoted $\overset{*}{\Longrightarrow}_G$. The *language* $L(G)$ *of* G is defined as $L(G) = \{s \in V_T^* \mid X_0 \overset{*}{\Longrightarrow}_G s\}$.

Given an SGG $G = (V_N, V_T, X_0, F)$, the main part of the work in defining the equivalent ASG $\Gamma_G = (\mathrm{T}, \mathrm{N}, \Sigma, W_0, P)$ is converting the substring rewriting rules to $(\mathrm{N}, \mathrm{T}, \Sigma)$-productions, which is done essentially according to the procedure just described. To ensure exact compliance with the definition of an ASG, however, two small modifications to the procedure are required.

The first change involves the start word W_0. We cannot define W_0 as a model of the one-letter string X_0 ($W_0 \in \mathcal{M}(X_0)$), because in our linked-list representation such a model would consist of two symbols (one representing X_0 plus the dummy link), and by definition $|W_0| = 1$. To get around this we introduce a new, nullary nonterminal class s, let $W_0 = \{s()\}$, and introduce one production of the form $W_0 \rightarrow W$ where $W \in \mathcal{M}(X_0)$.

The second change involves maintaining, in Γ_G, the distinction between terminal and nonterminal symbols as defined by G. The first step is to define the alphabets $\mathrm{T} = (\underline{\mathrm{C}}_t)$ and $\mathrm{N} = (\underline{\mathrm{C}}_n)$. This complicates the translation of substring rewriting rules to productions in two ways:

1. We must now ensure that in productions, terminal letters are translated as symbols of class t and nonterminal letters are translated as symbols of class n.
2. Every substring rewriting rule of G must now be translated into two productions in Γ_G, which are identical except that the "left-context" symbols are class t in one and class n in the other.

These ideas are illustrated in the following example.

Example 18: Modelling an SGG with an ASG

Let $G = (\{X\}, \{a, b\}, X, \{X \to \varepsilon, X \to aXb)$. Observe that $L(G) = \{a^i b^i \mid i \in \mathbb{N}_0\}$.[9]

Let $\mathbb{N}^\sharp = \mathbb{N} \cup \{\sharp\}$, $V^\sharp = \{X, a, b, \sharp\}$, $L = \{p_0, p_1, \ldots, q_1, q_2 \ldots\}$, $U = \{l, r, u_0, u_1, \ldots, v_0, v_1, \ldots\}$, $\Sigma = (V_L^\sharp, \mathbb{N}_U^\sharp)$, $\underline{\mathrm{C}} = (v^\sharp_{\mathrm{letter}}, \mathbb{N}^\sharp_{\mathrm{self}}, \mathbb{N}^\sharp_{\mathrm{next}})$, $\mathrm{T} = (\underline{\mathrm{C}}_t)$, $\mathrm{N} = (\emptyset_s, \underline{\mathrm{C}}_t)$, and $W_0 = \{s()\}$. Consider the ASG $\Gamma_G = (\mathrm{T}, \mathrm{N}, \Sigma, W_0, P)$ where P consists of the productions

$$
\begin{aligned}
p_0 &: \quad s() \to t(\sharp, v_1, v_2), n(X, v_2, \sharp) \\
p_{1a} &: \quad n(p_0, l, u_1), n(X, u_1, r) \to n(p_0, l, r) \\
p_{1b} &: \quad t(p_0, l, u_1), n(X, u_1, r) \to t(p_0, l, r) \\
p_{2a} &: \quad n(p_0, l, u_1), n(X, u_1, r) \to n(p_0, l, v_1), t(a, v_1, v_2), n(X, v_2, v_3), t(b, v_3, r) \\
p_{2b} &: \quad t(p_0, l, u_1), n(X, u_1, r) \to t(p_0, l, v_1), t(a, v_1, v_2), n(X, v_2, v_3), t(b, v_3, r)
\end{aligned}
$$

Observe that the productions $p \in P$ maintain the pointer structure of our model scheme, regardless of the actual values which are (freely) assigned to *self* and *next* attribute values at each rewriting step. I claim that

1. $\forall s \in L(G)$, $\mathcal{M}(s) \subseteq L(\Gamma_G)$.
2. $\forall W \in L(\Gamma_G)$, $\mathcal{R}(W) \in L(G)$.

The proof of the first claim follows from observation that for every valid derivation $X \overset{*}{\Longrightarrow}_G s$ in G, there is a valid derivation $W_0 \overset{*}{\Longrightarrow} W$ in Γ_G with corresponding rewriting steps, such that $W \in \mathcal{M}(s)$. The only room for variation is in the free assignment of *self* and *next* attribute values, which, as noted above, is unimportant. Vice versa for the second claim.

4.5.3 Directed graphs

In the conclusion of chapter 3 (section 3.7, page 57), I suggested that the ability to generate arbitrary directed graph structures should be considered a minimum requirement for a visual grammar formalism. In this section I define an attributed set modelling scheme for directed graphs, and present a simple ASG which can generate models of every directed graph in this scheme.

A *directed graph* or digraph is a pair (V, E) where V is a set of *nodes* and $E \subseteq V^2$ is a binary relation on V. Each pair $(v_1, v_2) \in E$ is called an *arc* and is interpreted

[9] This example is based on an SGG described by Salomaa; see [121, page 5,10] for details.

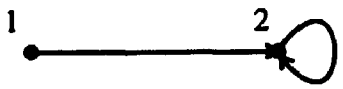


Figure 4.7: A simple directed graph diagram with node labels

as a one-way connection from node ν_1 to node ν_2. If $\nu_1 = \nu_2$ the arc is called a *loop*. Digraphs are typically notated as diagrams using dots or circles to represent nodes (often with accompanying textual or numeric labels) and arrows to represent arcs as shown in figure 4.7.

We choose to encode digraphs as attributed word models as follows:

- nodes are numbered, with arbitrary distinct elements of $\mathbb{N}$.
- each node is represented by an attributed symbol of class *node* with one $\mathbb{N}$-valued attribute called *self* whose value is that node's identifying number.
- arcs (ν_1, ν_2) are represented by attributed symbols of class *arc* having two $\mathbb{N}$-valued attributes *from* and *to*, whose values are the identifying numbers of nodes ν_1 and ν_2, respectively.

For example, the digraph illustrated in figure 4.7 could be encoded as the model

$$\{\,\mathrm{node}(1), \mathrm{arc}(1,2), \mathrm{node}(2), \mathrm{arc}(2,2)\,\}.$$

Note that, as in the preceding strings example, the exact attribute values are unimportant—only their relationships matter. Our models are thus words over an alphabet $\mathrm{T} = (\underline{\mathrm{C}}_{\mathbf{node}}, \underline{\mathrm{D}}_{\mathbf{arc}})$ where $\underline{\mathrm{C}} = (\mathbb{N}_{\mathbf{self}})$ and $\underline{\mathrm{D}} = (\mathbb{N}_{\mathbf{from}}, \mathbb{N}_{to})$. Specifically, the language of valid digraph models is

$$L_{\mathbf{digraph}} = \Big\{ W \in \underline{\mathrm{words}}(\mathrm{T}) \;\Big|\; \forall x, y \in \mathbb{N},\ \mathrm{arc}(x,y) \in W \text{ implies } \mathrm{node}(x), \mathrm{node}(y) \in W \Big\}.$$

That is, there may be zero or more *node* symbols, and zero or more *arc* symbols, subject only to the requirement that the *from* and *to* attributes of every *arc* must refer to some *node*.

Now let us formalize the encoding relation $\mathcal{M}$ and the decoding relation $\mathcal{R}$ for this modelling scheme. Let **DG** be the class of all digraphs and, for every $(V, E) \in \mathbf{DG}$, let

$$\mathcal{M}((V,E)) = \Big\{ W \in \underline{\text{words}}(\mathrm{T}) \;\Big|\; \forall \mu : V \to \mathsf{N}, \text{ such that } \mu \text{ is injective and } \forall \nu \in V,\ \text{node}(\mu(\nu)) \in W \text{ and } \forall (\nu_1, \nu_2) \in E,\ \text{arc}(\mu(\nu_1), \mu(\nu_2)) \in W \Big\}.$$

For every model $W \in L_{\text{digraph}}$, let Z denote the set of *self* attribute values of *node* symbols in W, and let

$$\mathcal{R}(W) = \Big\{ (V,E) \;\Big|\; V \text{ is a set, } |V| = |Z|,\ E \subseteq V^2 \text{ and } \exists \beta : Z \to V, \text{ such that } \beta \text{ is a bijection and } \forall n_1, n_2 \in \mathsf{N},\ \text{arc}(n_1, n_2) \in W \text{ implies } (\beta(n_1), \beta(n_2)) \in E \Big\}.$$

Let $\approx$ denote equivalence up to renaming of nodes; $\approx$ is an equivalence relation on **DG**. Given the above definitions of $\mathcal{M}$ and $\mathcal{R}$, it should be apparent that

$$\forall (V,E) \in \mathbf{DG},\ \forall ((V,E),W'), ((V,E),W'') \in \mathcal{M},\ \forall (W',(V',E')), (W'',(V'',E'')) \in \mathcal{R},\ (V,E) \approx (V',E') \approx (V'',E'').$$

That is, encoding a digraph (V, E) and subsequently decoding it will always yield a digraph equivalent to the first up to renaming of the vertices. The set V is lost in the encoding,[10] but upon decoding it will be replaced by some other set with the same number of elements as V.

Having defined the language L_M of models of digraphs, we now define an ASG Γ_{digraph} with $L(\Gamma_{\text{digraph}}) = L_{\text{digraph}}$. Let $U = \{u_1, u_2, \ldots\}$ be a variable domain and $\Sigma = (\mathsf{N}_U)$ be a term system. Consider the ASG $\Gamma_{\text{digraph}} = (\mathrm{T}, \mathrm{N}, \Sigma, W_0, P)$ where $\mathrm{T} = (\underline{\mathrm{C}}_{\text{node}}, \underline{\mathrm{D}}_{\text{arc}})$ as above and $\mathrm{N} = (\emptyset_{\text{digraph}}, \underline{\mathrm{C}}_{\mathrm{n}})$, $W_0 = \{\text{digraph}()\}$ and P consists of the productions

$$\begin{aligned}
p_0 &: \quad \text{digraph}() \to \emptyset \\
p_1 &: \quad \text{digraph}() \to \mathrm{n}(u_1) \\
p_2 &: \quad \mathrm{n}(u_1) \to \text{node}(u_1) \\
p_3 &: \quad \mathrm{n}(u_1) \to \mathrm{n}(u_1), \mathrm{n}(u_2) \\
p_4 &: \quad \mathrm{n}(u_1) \to \mathrm{n}(u_1), \text{arc}(u_1, u_1) \\
p_5 &: \quad \mathrm{n}(u_1), \mathrm{n}(u_2) \to \mathrm{n}(u_1), \mathrm{n}(u_2), \text{arc}(u_1, u_2)
\end{aligned}$$

[10] It must be, because we have no way of using V in the definition of the model language L_{digraph}.

Before proving that $L(\Gamma_{\text{digraph}}) = L_{\text{digraph}}$, let us consider two examples of digraph model derivation under Γ_{digraph}.

Example 19: Null digraph

The null digraph $(\emptyset, \emptyset)$ is modelled by the null word $\emptyset$, which has the derivation

$$\begin{array}{c} \underline{\text{digraph()}} \\ p_1 \Downarrow \\ \emptyset \end{array}$$

In a practical implementation of this modelling scheme for digraphs, we might not want to assign meaning to the empty word. An alternative is to modify the encoding scheme to use an explicit non-empty model for the null digraph. That is, we could define $\mathrm{T}' = (\emptyset_{\text{empty}}, \underline{\mathrm{C}}_{\text{node}}, \underline{\mathrm{D}}_{\text{arc}})$ and define the ASG $\Gamma'_{\text{digraph}} = (\mathrm{T}', \mathrm{N}, \Sigma, W_0, P')$ where P' is identical to P except that production p_0 is replaced by

$$p'_0: \quad \text{digraph()} \rightarrow \text{empty()}.$$

In this encoding scheme, the null digraph is modelled by the word {empty()}, which has the derivation

$$\begin{array}{c} \underline{\text{digraph()}} \\ p'_1 \Downarrow \\ \text{empty()} \end{array}$$

Example 20: Digraph as in figure 4.7

The digraph illustrated in figure 4.7 has a model representation

$$\{\text{node}(1), \text{node}(2), \text{arc}(1,2), \text{arc}(2,2)\}$$

which can be derived under Γ_{digraph} as follows:

$$\underline{\text{digraph}()}$$
$$p_1 \Downarrow v_1 = 1$$
$$\underline{\text{n}(1)}$$
$$p_3 \Downarrow v_1 = 1, v_2 = 2$$
$$\underline{\text{n}(1)}, \underline{\text{n}(2)}$$
$$p_5 \Downarrow v_1 = 1, v_2 = 2$$
$$\text{n}(1), \underline{\text{n}(2)}, \text{arc}(1,2)$$
$$p_4 \Downarrow v_1 = 2$$
$$\underline{\text{n}(1)}, \text{n}(2), \text{arc}(1,2), \text{arc}(2,2)$$
$$p_2 \Downarrow v_1 = 1$$
$$\text{node}(1), \underline{\text{n}(2)}, \text{arc}(1,2), \text{arc}(2,2)$$
$$p_2 \Downarrow v_1 = 2$$
$$\text{node}(1), \text{node}(2), \text{arc}(1,2), \text{arc}(2,2)$$

Lemma 9 $L(\Gamma_{\text{digraph}}) \subseteq L_{\text{digraph}}$.

Proof: Let $W \in L(\Gamma_{\text{digraph}})$. Then W has a derivation $W_0 \stackrel{*}{\Longrightarrow} W$ under Γ_{digraph}. If W contains no symbols of class *arc*, then $W \in L_{\text{digraph}}$ by definition.

Suppose the symbol $\text{arc}(x, y) \in W$ for some $x, y \in \mathbb{N}$, and consider the attributed word sequence W_0, W_1, etc. within the derivation $W_0 \Longrightarrow W_1 \Longrightarrow \cdots \Longrightarrow W$.

If $x = y$, the symbol must have been introduced in a derivation step $W_i \stackrel{p_4}{\Longrightarrow} W_{i+1}$, implying that the symbol $\text{n}(x)$ must have been an element of both W_i and W_{i+1}. Only production p_2 can eliminate this symbol; it rewrites it as $\text{node}(x)$. No production can eliminate symbols of class *node*. We know that p_2 must be applied at some point later in the derivation, because we know $W \in L(\Gamma_{\text{digraph}})$ and hence W must contain only terminal symbols. Hence the condition

$$\forall x, y \in \mathbb{N},\ \text{arc}(x, y) \in W \text{ implies } \text{node}(x), \text{node}(y) \in W$$

is met and thus $W \in L_{\text{digraph}}$.

If $x \neq y$, the symbol $\text{arc}(x, y)$ must have been introduced in a derivation step $W_i \stackrel{p_5}{\Longrightarrow} W_{i+1}$, implying that the symbols $\text{n}(x)$ and $\text{n}(y)$ must have been elements of both W_i and W_{i+1}. By an argument similar to the one above, we have $\text{node}(x) \in W$ and $\text{node}(y) \in W$ and again $W \in L_{\text{digraph}}$. □

Lemma 10 $L_{\text{digraph}} \subseteq L(\Gamma_{\text{digraph}})$.

Proof: Let $W \in L_{\text{digraph}}$. If $W = \emptyset$ the result follows immediately. Otherwise W consists of n symbols $\text{node}(x_1), \text{node}(x_2), \ldots, \text{node}(x_n)$ and m symbols $\text{arc}(f_1, t_1), \ldots, \text{arc}(f_m, t_m)$ subject to the conditions given in the definition of L_{digraph}. A derivation for W in Γ_{digraph} can be constructed as follows:

1. Start with W_0.
2. Apply p_1 once (with a Σ-assignment α such that $\alpha(u_1) = x_1$) yielding $\{\text{n}(x_1)\}$.
3. Apply p_3 $n-1$ times (with appropriate assignments) yielding $\{\text{n}(x_1), \ldots, \text{n}(x_n)\}$.
4. Apply p_4 and/or p_5 as appropriate, for a total of m steps to generate the $\text{arc}(f_1, t_1), \ldots, \text{arc}(f_m, t_m)$.
5. Apply p_2 n times to convert all the symbols of class n to symbols of class *node*.

□

Theorem 1 $L(\Gamma_{\text{digraph}}) = L_{\text{digraph}}$.

Proof: By lemmas 9 and 10 above. □

4.5.4 Binary trees

A *binary tree* is a particular kind of digraph (V, E) which is either the null digraph or for which all of the following conditions hold:

1. There is a distinguished node called the *root* $r \in V$, such that $\neg(\exists\, v \in V,\ (v, r) \in E)$. (The root has in-degree 0.)
2. For every $v \in V \setminus \{r\}$, there exists exactly one $v' \in V$ such that $(v', v) \in E$. (Every node except the root has in-degree 1.)
3. For every $v \in V$, there exist at most two other nodes $v' \in V$ such that $(v, v') \in E$. (Every node has out-degree 0, 1, or 2.)[11]

[11] This is a simplified definition. More common definitions designate each subtree of a node as either "left" or "right".

A node with out-degree 0 in a binary tree is called a *leaf*. Nodes other than the root and leaves are called *interior nodes*.

Since binary trees are digraphs, they can be modelled as described in section 4.5.3 above. That is to say, the language L_{BT} of models of binary trees is a subset of $L_{\mathbf{digraph}}$, defined as follows:

$$L_{\mathrm{BT}} = \{W \in \mathrm{L}_{\mathbf{digraph}} \mid \text{ either } W = \emptyset \text{ or conditions 1-4 below hold } \}$$

1. $\mathrm{node}(1) \in W$.
2. $\neg\exists\, i \in \mathsf{N},\ \mathrm{arc}(i,1) \in W$.
3. $\forall i \in \mathsf{N} \setminus \{1\},\ \mathrm{node}(i) \in W$ implies that there exists exactly one $j \in \mathsf{N}$ such that $\mathrm{node}(j), \mathrm{arc}(j,i) \in W$.
4. $\forall i \in \mathsf{N} \setminus \{1\},\ \mathrm{node}(i) \in W$ implies that there exist at most two distinct $j \in \mathsf{N}$ such that $\mathrm{arc}(i,j) \in W$.

Note how in this encoding scheme, the assignment of attribute values is not completely free—the value 1 is reserved to identify the root node. The four conditions above are of course obtained directly from the three conditions given above in the definition of a binary tree.

Recall the following definitions from the digraph example above. $U = \{u_1, u_2, \ldots\}$ (a variable domain). $\Sigma = (\mathsf{N}_U)$ (a term system). $\mathrm{T} = (\underline{\mathrm{C}}_{\mathbf{node}}, \underline{\mathrm{D}}_{\mathbf{arc}})$ where $\underline{\mathrm{C}} = (\mathsf{N}_{\mathbf{self}})$ and $\underline{\mathrm{D}} = (\mathsf{N}_{\mathbf{from}}, \mathsf{N}_{to})$.

Let $\mathrm{N}' = (\underline{\mathrm{C}}_{\mathrm{n}})$ and consider the ASG $\Gamma_{\mathrm{BT}} = (\mathrm{T}, \mathrm{N}', \Sigma, W_0', P')$ where $W_0' = \{\mathrm{subtree}(1)\}$ and P' consists of the productions

$$\begin{aligned}
p_1' &: \quad \mathrm{subtree}(u_1) \rightarrow \mathrm{node}(u_1) \\
p_2' &: \quad \mathrm{subtree}(u_1) \rightarrow \mathrm{node}(u_1), \mathrm{subtree}(u_2), \mathrm{arc}(u_1, u_2) \\
p_3' &: \quad \mathrm{subtree}(u_1) \rightarrow \mathrm{node}(u_1), \mathrm{subtree}(u_2), \mathrm{subtree}(u_3), \mathrm{arc}(u_1, u_2), \mathrm{arc}(u_1, u_3)
\end{aligned}$$

It should be obvious that $L(\Gamma_{\mathrm{BT}}) = L_{\mathrm{BT}} \setminus \{\emptyset\}$.[12] Note that

1. production p_1' creates leaf nodes.
2. production p_2' creates an interior node with one subtree.
3. production p_3' creates an interior node with two subtrees.

[12]For simplicity, I have omitted the extra mechanism (an extra nullary nonterminal symbol class and two extra productions) required to generate the empty word in this example. They can easily be added following the digraph example.

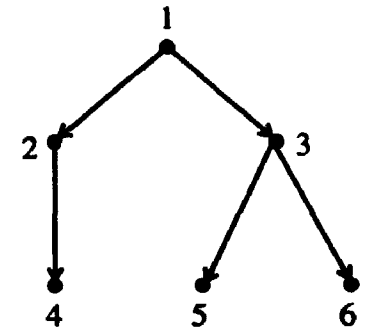


Figure 4.8: Binary tree

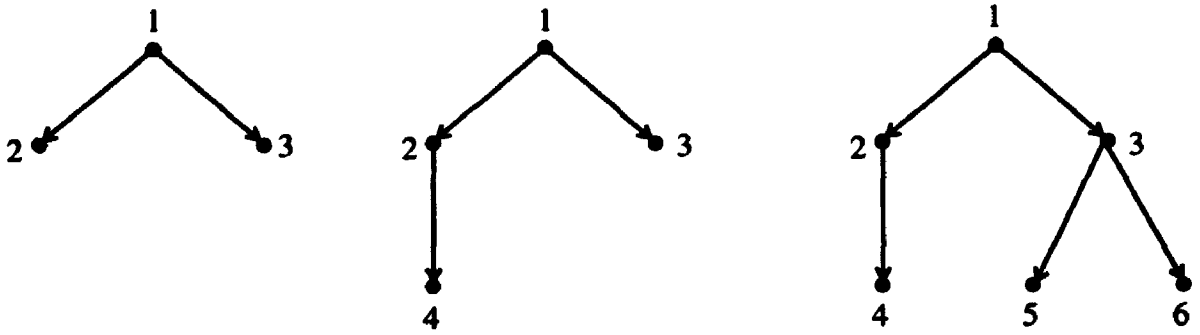


Figure 4.9: Binary tree development: first three rewriting steps

Example 21: Binary tree generation

Consider the binary tree illustrated in figure 4.8. A model for this tree (where the *self* attribute values correspond to the node labels in figure 4.8) is

$$\{\, \mathrm{node}(1), \mathrm{node}(2), \mathrm{node}(3), \mathrm{node}(4), \mathrm{node}(5), \mathrm{node}(6),$$
$$\mathrm{arc}(1,2), \mathrm{arc}(1,3), \mathrm{arc}(2,4), \mathrm{arc}(3,5), \mathrm{arc}(3,6) \,\}$$

This word has the following derivation in Γ_{BT}:

$$\underline{\text{subtree}(1)}$$
$$p_3' \Downarrow u_1 = 1, u_2 = 2, u_3 = 3$$
$$\text{node}(1), \underline{\text{subtree}(2)}, \text{subtree}(3), \text{arc}(1,2), \text{arc}(1,3)$$
$$p_2' \Downarrow u_1 = 2, u_2 = 4$$
$$\text{node}(1), \text{node}(2), \text{subtree}(4), \text{arc}(2,4),$$
$$\underline{\text{subtree}(3)}, \text{arc}(1,2), \text{arc}(1,3)$$
$$p_3' \Downarrow u_1 = 3, u_2 = 5, u_3 = 6$$
$$\text{node}(1), \text{node}(2), \underline{\text{subtree}(4)}, \text{arc}(2,4),$$
$$\text{node}(3), \text{subtree}(5), \text{subtree}(6), \text{arc}(3,5), \text{arc}(3,6), \text{arc}(1,2), \text{arc}(1,3)$$
$$p_1' \Downarrow u_1 = 4$$
$$\text{node}(1), \text{node}(2), \text{node}(4), \text{arc}(2,4),$$
$$\text{node}(3), \underline{\text{subtree}(5)}, \text{subtree}(6), \text{arc}(3,5), \text{arc}(3,6), \text{arc}(1,2), \text{arc}(1,3)$$
$$p_1' \Downarrow u_1 = 5$$
$$\text{node}(1), \text{node}(2), \text{node}(4), \text{arc}(2,4),$$
$$\text{node}(3), \text{node}(5), \underline{\text{subtree}(6)}, \text{arc}(3,5), \text{arc}(3,6), \text{arc}(1,2), \text{arc}(1,3)$$
$$p_1' \Downarrow u_1 = 6$$
$$\text{node}(1), \text{node}(2), \text{node}(4), \text{arc}(2,4),$$
$$\text{node}(3), \text{node}(5), \text{node}(6), \text{arc}(3,5), \text{arc}(3,6), \text{arc}(1,2), \text{arc}(1,3)$$

If we interpret attributed symbols of class *subtree* graphically as dots, just like those of class *node*, we can interpret all of the intermediate words generated in the derivation graphically. Figure 4.9 illustrates the result after each of the first three rewriting steps, which are the ones which actually build the tree structure.

The graphical interpretation of intermediate derivation results suggested in example 21 suggests the intriguing possibility that ASGs might be used to model organic development, in the same manner as Lindenmayer systems [88] (see section 2.3). Looking at Γ_{BT} as a developmental system for modelling tree growth we have the following "botanical" interpretations:

- Symbols of class *subtree* are analogous to buds.
- Production p'_1 causes a bud to mature into a leaf.
- Production p'_2 causes a bud to sprout into a stalk with a new bud at the end.
- Production p'_3 causes a bud to branch into two stalks with a new bud at the end of each.

4.5.5 Binary tree diagrams with layout constraints

The preceding examples have illustrated the use of attributed sets to model purely algebraic structures. I now turn to graphical structures with geometric layout constraints, choosing to model binary tree diagrams such as figure 4.8, with constraints to determine layout. I build on the earlier binary tree example, adding only what is needed to model and generate appropriate layout constraints.

Formatting technique

Automatic formatting of tree diagrams has been considered by Brüggemann-Klein and Wood [10], Luo [89], and others.[13] Luo presents a number of aesthetic rules for tree diagram layout due to other authors, and one of her own.[14] The method discussed below adheres to some of these rules, but my intent here is to illustrate the ability of ASGs to generate constraint-based models, not to propose a realistic formatting technique.

My formatting technique works for diagrams representing binary trees as defined in section 4.5.4. The diagrams are constructed of dots representing nodes and line segments representing arcs, as in figures 4.8 and 4.9, but without the node labels. Associated with each subtree is its root node position (x, y coordinates) and

[13] See the references in these two works for additional citations.

[14] See [89, pages 5,6,10].

its horizontal extent (minimum and maximum node x-coordinates). The layout is constrained according to two rules:

1. When a node has one subtree, the root of the subtree is positioned directly beneath it, a distance dy below.

2. When a node has two subtrees, they are arranged so that their roots are a distance dy below and equidistant in x from the node. The horizontal separation of the two subtrees is such that the minimum x-coordinate of the right subtree[15] is an amount dx greater than the maximum x-coordinate of the left subtree.

Lexical representation: G, Σ

Let $\underline{\mathrm{C}} = (\mathbf{R}_x, \mathbf{R}_y)$, $\underline{\mathrm{D}} = (\mathbf{R}_{x_1}, \mathbf{R}_{y_1}, \mathbf{R}_{x_2}, \mathbf{R}_{y_3})$, and $\underline{\mathrm{E}} = (\mathbf{R}_a, \mathbf{R}_b, \mathbf{R}_c)$ be classes, and $\mathrm{G} = (\underline{\mathrm{C}}_{\mathbf{node}}, \underline{\mathrm{D}}_{\mathbf{arc}}, \underline{\mathrm{C}}_{\mathbf{equal}}, \underline{\mathrm{E}}_{\mathbf{sum}}, \underline{\mathrm{E}}_{\mathbf{mid}})$ be an attributed alphabet. Let $R = \{dx, dy, px, py, lx, rx, x, x_1, x_2, \ldots, y, y_1, y_2, \ldots\}$ be a variable domain and $\Sigma = (\mathbf{R}_R)$ be a term system.

We shall model the structure and layout of binary tree diagrams as words over G^{Σ}. Symbols of class *node* and *arc* will be interpreted as marks with no inherent constraints, and symbols of other classes will be interpreted as pure constraints, as follows:

- node(x, y) represents a dot centred at point (x, y).
- arc(x_1, y_1, x_2, y_2) represents a line segment from point (x_1, y_1) to point (x_2, y_2).
- equal(a, b) represents the constraint $a = b$.
- sum(a, b, c) represents the constraint $a = b + c$.
- mid(a, b, c) represents the constraint $a = \frac{1}{2}(b + c)$.

Words $W \in \underline{\text{words}}\left(\mathrm{G}^{\Sigma}\right)$ can be interpreted as a diagram via a grounding Σ-assignment, as described section 4.3.5.

[15]Recall that the definition of binary tree given in section 4.5.4 does not distinguish "left" and "right" subtrees. By "right subtree" I mean the subtree which, in the diagram layout, happens to have the greater x coordinate.

Syntactic representation: T, N, $\hat{\Sigma}$

Let $\hat{R} = \{\hat{lx}, \hat{lx}', \hat{rx}, \hat{rx}', \hat{cx}, \hat{cx}', \hat{cx}'', \hat{x}, \hat{x}_1, \hat{x}_2, \ldots, \hat{y}, \hat{y}_1, \hat{y}_2, \ldots\}$ be a variable domain and $\hat{\Sigma} = (R_{\hat{R}})$ be a term system. We can think of a $\hat{\Sigma}$-assignment as a conversion from the variable domain $\hat{R}$ to the variable domain R. Alternatively, we can say that variables under Σ are constants under $\hat{\Sigma}$, but this is potentially confusing. Let us call them *literals*.

Define the classes $\underline{\hat{C}} = (R_x, R_y)$, $\underline{\hat{D}} = (R_{x_1}, R_{y_1}, R_{x_2}, R_{y_3})$, $\underline{E} = (R_a, R_b, R_c)$, and the alphabet $T = (\underline{\hat{C}}_{\text{node}}, \underline{\hat{D}}_{\text{arc}}, \underline{\hat{C}}_{\text{equal}}, \underline{\hat{E}}_{\text{sum}}, \underline{\hat{E}}_{\text{mid}})$. Note that $\underline{\text{words}}(T) \subseteq \underline{\text{words}}\left(G^{\Sigma}\right)$. That is, any word over T is also a word over G^{Σ}, which happens not to contain any attributes which are constants under Σ. We shall define an ASG which generates words over T, treating attributes $r \in R$ (which we call *literals* in this context) as constants for syntactic purposes. Once generated, such words can be given a semantic interpretation as defined in section 4.5.5 above, where the attribute values are treated as variables.

Let $\underline{F} = (R_x, R_y, R_l, R_r)$ be a class, and $N = (\underline{F}_{\text{subtree}})$ be an attributed alphabet. We can think of an attributed symbol subtree$(x, y, x_{\min}, x_{\max})$ (which will be a nonterminal in the ASG defined below) as representing a subtree of a binary tree diagram, whose root node is located at point (x, y) and whose extent in the x-direction is given by $x_{\min}$ and $x_{\max}$.

An ASG for binary tree diagram models

We now define an ASG Γ_{BTD} which generates words over T, which model the logical structure and layout of binary tree diagrams as described in sections 4.5.5 and 4.5.5.

Let $\Gamma_{\text{BTD}} = (T, N, \hat{\Sigma}, W_0, P)$ where $W_0 = \{\text{subtree}(px, py, lx, rx)\}$ and P consists of the productions

$$
\begin{aligned}
p_1: \quad & \text{subtree}(\hat{x}, \hat{y}, \hat{lx}, \hat{rx}) \rightarrow \text{node}(\hat{x}, \hat{y}), \text{equal}(\hat{x}, \hat{lx}), \text{equal}(\hat{lx}, \hat{rx}) \\
p_2: \quad & \text{subtree}(\hat{x}, \hat{y}, \hat{lx}, \hat{rx}) \rightarrow \text{node}(\hat{x}, \hat{y}), \text{subtree}(\hat{x}, \hat{cy}, \hat{lx}, \hat{rx}), \\
& \qquad \text{arc}(\hat{x}, \hat{y}, \hat{x}, \hat{cy}), \text{sum}(\hat{cy}, \hat{y}, dy) \\
p_3: \quad & \text{subtree}(\hat{x}, \hat{y}, \hat{lx}, \hat{rx}) \rightarrow \text{node}(\hat{x}, \hat{y}), \text{subtree}(\hat{cx}', \hat{cy}, \hat{lx}, \hat{rx}'), \\
& \qquad \text{subtree}(\hat{cx}'', \hat{cy}, \hat{lx}', \hat{rx}), \\
& \qquad \text{arc}(\hat{x}, \hat{y}, \hat{cx}', \hat{cy}), \text{arc}(\hat{x}, \hat{y}, \hat{cx}'', \hat{cy}), \\
& \qquad \text{sum}(\hat{cy}, \hat{y}, dy), \text{sum}(\hat{lx}', \hat{rx}', dx), \text{mid}(\hat{x}, \hat{cx}', \hat{cx}'')
\end{aligned}
$$

Note the use of the *literals* px, py, lx and rx in W_0, and dx, dy in the productions. These literals will appear in every word generated under Γ_{BTD}.

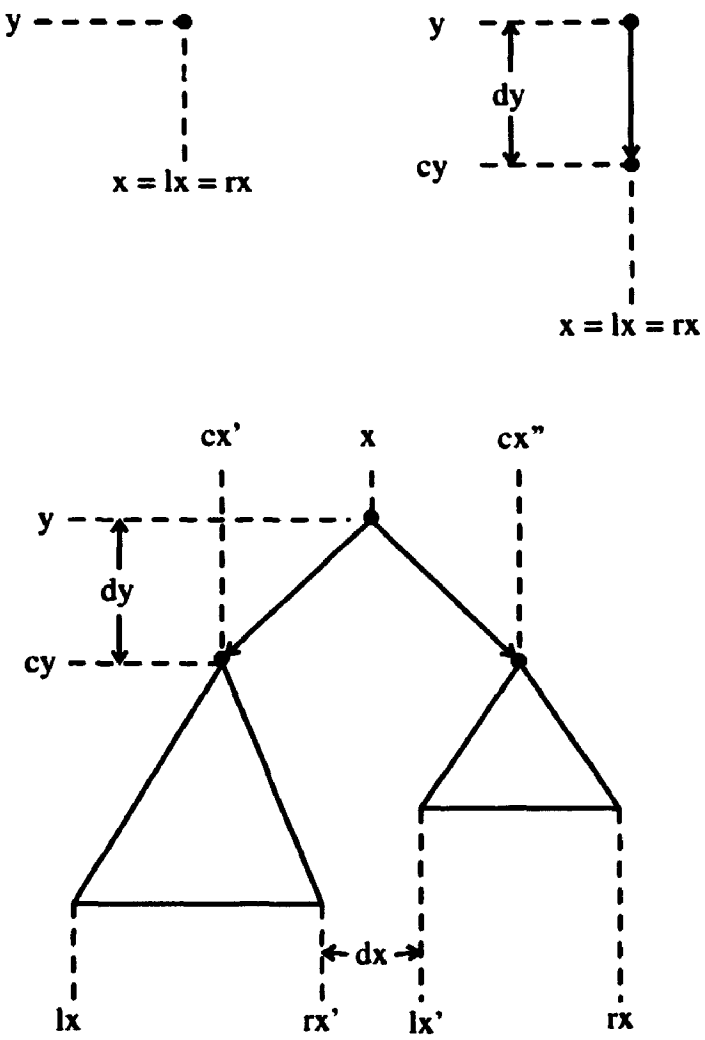


Figure 4.10: Geometric relationships in productions of Γ_{BTD}

Compare the definition of Γ_{BTD} with that of Γ_{BT} in section 4.5.4. The productions are exactly comparable. The difference is that instead of node and arc symbols being distinguished by integer identifiers, they are now distinguished by (x, y) coordinates, and we simultaneously generate constraints to enforce the rules of layout defined above. Figure 4.10 illustrates the geometric relationships captured by each of the three productions in Γ_{BTD}.

Example 22: The binary tree diagram of figure 4.8

A model for the binary tree diagram shown in figure 4.8 (not including the node labels), including all of its layout constraints as described above, is in $L(\Gamma_{BTD})$. Its derivation directly parallels the one given in example

21. Here is its derivation, in abbreviated form:

$$
\begin{array}{c}
\underline{\text{subtree}(px, py, lx, rx)} \\
p_3 \Downarrow \hat{x} = px, \hat{y} = py, \hat{lx} = lx, \hat{rx} = rx, \\
\quad \hat{cy} = y_1, \hat{cx'} = x_1, \hat{rx'} = x_2, \\
\quad \hat{cx''} = x_3, \hat{lx'} = x_4 \\
\quad \cdots \\
p_2 \Downarrow \hat{x} = x_1, \hat{y} = y_1, \hat{lx} = lx, \hat{rx} = x_2 \\
\quad \cdots \\
p_3 \Downarrow \hat{x} = x_3, \hat{y} = y_1, \hat{lx} = x_4, \hat{rx} = rx, \\
\quad \hat{cy} = y_3, \hat{cx'} = x_5, \hat{rx'} = x_6, \\
\quad \hat{cx''} = x_7, \hat{lx'} = x_8 \\
\quad \cdots \\
p_1 \Downarrow \hat{x} = x_1, \hat{y} = y_1, \hat{lx} = lx, \hat{rx} = x_2 \\
\quad \cdots \\
p_1 \Downarrow \hat{x} = x_5, \hat{y} = y_3, \hat{lx} = x_4, \hat{rx} = x_6 \\
\quad \cdots \\
p_1 \Downarrow \hat{x} = x_7, \hat{y} = y_3, \hat{lx} = x_8, \hat{rx} = rx \\
\text{node}(px, py), \text{arc}(px, py, x_1, y_1), \text{arc}(px, py, x_3, y_1), \\
\text{node}(x_1, y_1), \text{node}(x_1, y_2), \text{equal}(x_1, lx), \text{equal}(lx, x_2), \text{sum}(y_2, y_1, dy), \\
\text{node}(x_3, y_1), \text{arc}(x_1, y_1, x_1, y_2), \text{arc}(x_3, y_1, x_5, y_3), \text{arc}(x_3, y_1, x_7, y_3), \\
\text{node}(x_5, y_3), \text{equal}(x_4, x_6), \text{equal}(x_5, x_4), \\
\text{node}(x_7, y_3), \text{equal}(x_8, rx), \text{equal}(x_7, x_8), \\
\text{sum}(y_3, y_1, dy), \text{mid}(x_3, x_5, x_7), \text{sum}(x_8, x_6, dx), \\
\text{sum}(y_1, py, dy), \text{mid}(px, x_1, x_3), \text{sum}(x_4, x_2, dx)
\end{array}
$$

In the semantic interpretation of this model as a word over G^{Σ}, there are a total of seventeen variables ($px, py, dx, dy, lx, rx, x_1, \ldots, x_8, y_1, y_2, y_3$) and thirteen constraints, the latter representing thirteen (linearly independent) linear equations. Hence the system is under-constrained with four remaining degrees of freedom. Graphically, these four degrees of freedom correspond to the horizontal and vertical spacing constants dx and dy, and the position of the entire diagram in the XY coordinate space.

To obtain a solution we can fix the "environment parameters" dx and dy, to determine the spacing, and px and py, to locate the diagram by setting the coordinates of its root node. Alternatively, we could leave px and dx

free and set lx and rx instead, e.g. to fit the resulting diagram into a column of a given width.

Note that by putting the names of the variables which we consider to be "environment variables" into the start word W_0 and (in the case of dx and dy) the productions as *literals*, we ensure that these variable names will be used consistently in every word generated by the ASG.

4.5.6 Context dependency

The ASG for digraphs introduced in Section 4.5.3 involves some *context dependency*, expressed in the form of a production (p_5) having more than one symbol in its head. Each of the two symbols involved must be matched in the context of the other. By contrast, the ASG for binary trees given in Section 4.5.4 has no context dependency.

I conjecture that only a context-dependent ASG can generate the language L_{digraph} defined in Section 4.5.3. Unfortunately, I have been unable to prove this conclusively.

Analysis of the generative power of ASGs is a promising area for future investigation.

4.6 Parsing with ASGs

Having developed a grammar formalism, I now show that it can be used in syntactic analysis (parsing) of one class of attributed set languages.

4.6.1 The membership problem, acceptance and parsing of ASLs

Let T be an attributed alphabet, and $L \subseteq \underline{\text{words}}(\text{T})$ be an attributed set language over T. Given an attributed word $W \in \underline{\text{words}}(\text{T})$, determination of whether or not $W \in L$ is called the *membership problem for L*.

In practice we would like to have an algorithm which, given the word W, would at least answer "yes" if $W \in L$ and "no" otherwise, and which would halt on all inputs, i.e. not go into an infinite loop. Such an algorithm is called an *acceptor for L*. Even more useful would be an algorithm which, instead of just answering "yes" for inputs $W \in L$, would report a derivation for W in an ASG Γ_L, where $L(\Gamma_L) = L$. Such an algorithm is called a *parser*.

4.6.? Monotonic ASGs, decidability of the membership problem

I have not been able to prove whether or not the membership problem is decidable for all ASLs. In this section I define a class of ASLs for which the membership problem is decidable.

Let $\Gamma = (\mathrm{T}, \mathrm{N}, \Sigma, W_0, P)$ be an ASG. Γ is called *monotonic* if, for every production $X \rightarrow Y \in P$, three conditions hold.

1. $|X| \leq |Y|$.
2. if $|X| = |Y|$, Y must consist entirely of terminal symbols.
3. if X contains terminal symbols, Y must contain a greater number of terminal symbols than X.

Lemma 11 *The length of any derivation of a word W in a monotonic ASG can be at most $2|W| - 1$.*

Proof: Let $\Gamma = (\mathrm{T}, \mathrm{N}, \Sigma, W_0, P)$ be a monotonic ASG, and let $W \in L(\Gamma)$. Consider a derivation of W in Γ,

$$W_0 \underset{\alpha_1}{\overset{p_1}{\Longrightarrow}} W_1 \underset{\alpha_2}{\overset{p_2}{\Longrightarrow}} W_2 \Longrightarrow \cdots \underset{\alpha_{n-1}}{\overset{p_{n-1}}{\Longrightarrow}} W_{n-1} \underset{\alpha_n}{\overset{p_n}{\Longrightarrow}} W.$$

By definition of an attributed word, $|W|$ is finite. By definition of an ASG, $|W_0| = 1$. By definition of a monotonic ASG, $|W_0| \leq |W_1| \leq \cdots \leq |W_{n-1}| \leq |W|$ (condition 1 in the definition above). Furthermore since the only derivation steps which do not add new symbols must by definition introduce terminal symbols (condition 2 above), which cannot subsequently be rewritten (condition 3 above). Hence the number of steps n in the derivation is bounded. In the worst case there will be at most $|W| - 1$ steps which introduce no terminals (which by definition must add at least one new symbol each) followed by $|W|$ steps, each of which rewrite one nonterminal symbol by one terminal symbol. □

Theorem 2 *The membership problem is decidable for any ASL described by a monotonic ASG.*

Proof: Let L be an ASL described by an ASG $\Gamma_L = (\mathrm{T}, \mathrm{N}, \Sigma, W_0, P)$. We can construct an acceptor algorithm for L using the productions P as follows. Given a word $W \in \underline{\text{words}}(\mathrm{T})$, the algorithm tries to build a derivation for W in Γ in reverse,

by matching the bodies of productions $X \rightarrow Y$ to subsets of W via Σ-assignments α and rewriting these by the sets $X|_\alpha$. If it succeeds in constructing a derivation for W (by rewriting W to W_0) it outputs "yes". By Lemma 11, the algorithm can give up once $2|W| - 1$ steps have been tried, and there are can be no more than $|P|$ choices to try at each step. Hence the algorithm will halt for all finite inputs. If it exhausts all possibilities without success, it outputs "no". □

The acceptor algorithm suggested in the proof of Theorem 2 can easily be implemented using back-tracking, and, since in "yes" cases it actually constructs a derivation of the input word W, it can easily be turned into a parser. This is discussed in Chapter 5.

Theorem 3 *The worst-case time complexity of the algorithm described in the proof of Theorem 2 is $O(2^n)$, where n is the number of symbols in the input word W.*

Proof: In effect, the algorithm searches a tree of possibilities to a maximum depth of $2|W| - 1$ levels, which branches as much as $|P|$ ways at every node. The total number of paths which must be searched is thus $|P|^{2|W|-1}$. Each reverse application of a production (which can be called a *reduction step*) requires matching all of the variables in the body of the production to constants. This can be considered a constant-time operation since it the number of variables in each production is fixed and finite. Hence the worst-case time complexity of the algorithm is $O(2^{|W|})$. □

4.6.3 A simple bottom-up parser in Prolog

The algorithm suggested in the proof of Theorem 2 is easily implemented using back-tracking. If a programming language such as Prolog or CLP(R) is used, there is no need to program the back-tracking, and the matching/reduction mechanism is almost trivial. Attributed symbols can be represented as compound terms, where the functor is the class name and the arguments (which may be variables or constants)[16] Attributed sets can be represented as lists,[17] provided we are careful to ensure that

[16] In both Prolog and CLP(R), constants may be numbers or character strings beginning with a lower-case letter, while variables must begin with an upper-case letter or an underscore. A single underscore, used as a variable, is called an *anonymous variable* and is occasionally used for stylistic reasons. For details of Prolog syntax and semantics, see [23]; for CLP(R), see [68].

[17] The Prolog/CLP(R) syntax for lists uses square brackets e.g. `[x(1),abc,a(b,c)]`. The notation `[H|T]`, used in rules, instantiates variable `H` to the first element (head) of the list and `T` to the rest (tail).

elements are not duplicated (or that duplicate elements are harmless). Furthermore, the productions of the ASG under consideration can be expressed as rules as with Definite Clause Grammars (see Chapter 2), obviating the need for a specialized data structure to represent them.

Figure 4.11 gives the entire Prolog source code for an acceptor based on the ASG Γ_{digraph} defined in Section 4.5.3. Note that production p_0, which generates the empty set, has not been included, so the language accepted by the program is actually (the Prolog list representations of all words in) $L_{\text{digraph}} \setminus \{\emptyset\}$.[18]

The program contains cuts, primarily to ensure that it does not go into infinite recursion with uninstantiated variables. However these also allow for early termination in some cases; for example the main routine for accept cuts as soon as a complete derivation is found.

The productions p_1 through p_5 of Γ_{digraph} have been placed into the program in reverse order. This is because, if there are two symbols of class n in the list at some point, a reduction could be performed according to p_3 or p_5. (This situtation is called a *reduce/reduce conflict* and is discussed further in chapter 5.) If a reduction is performed according to p_3 when p_5 should have been used, an n symbol will be incorrectly erased, and the result is that the program will go into infinite recursion.[19] Ordering the rules as shown will cause p_5 to be favoured over p_3, because Prolog always chooses rules in the order in which they appear.

[18]This can be fixed by adding a rule of the form `accept([]).`

[19]This is not really serious. The problem is that this implementation does not prune the search tree by giving up when a test derivation gets too long. This feature could of course be added.

```
% productions coded as prod(head,body)
prod([n(V1),n(V2)],[n(V1),n(V2),arc(V1,V2)]).  % p5
prod([n(V1)],[n(V1),arc(V1,V1)]).              % p4
prod([n(V1)],[n(V1),n(_)]).                    % p3
prod([n(V1)],[node(V1)]).                      % p2
prod([digraph],[n(_)]).                        % p1

% acceptor
accept([digraph]).     % accept the start symbol
accept(W) :-           % accept W if
   prod(X,Y),          % there is some production X->Y where Y matches W
   match(Y,W,W1),      % (W1 is unmatched part of W, this step grounds X)
   union(X,W1,W2),     % W2 is result after performing reduction
   accept(W2),         % keep reducing
   !.

% match(A,B,C)
% A is a subset of B, and C is its complement w.r.t. B
match([],B,B) :- !.
match([E|A],B,C) :-
   element(E,B,B1),
   match(A,B1,C).

% element(+E,+S,-T)
% succeeds if E is an element of set S; T is S minus E
element(E,[F|R],R).
element(E,[X|S],[X|T]) :-
   element(E,S,T).

% union(+X,+Y,?Z)
% Z is union of sets X and Y
union([],X,X).
union([X|R],Y,Z) :-
   element(X,Y,_), !,
   union(R,Y,Z).
union([X|R],Y,[X|Z]) :-
   union(R,Y,Z).
```

Figure 4.11: Prolog acceptor for L_{digraph}

Chapter 5

Parsing Attributed Set Languages

Chapter 4 ended by proving that, in theory at least, it is possible to parse attributed sets. In this chapter I consider the matter of attributed set parsing in greater detail, with particular emphasis on parsing attributed set encodings of notations. I describe two experimental implementations, address the problems of reduce-reduce conflict and ambiguity, discuss incremental parsing, and propose a *guided parsing* approach useful in interactive applications.

5.1 Implementation 1: 1991–1992

In 1991 I made a first attempt at implementing a notation-processing system, which I presented in a poster at the 1992 *IEEE Workshop on Visual Languages* in Seattle, and also described in a departmental Technical Report [41]. My development of the theory of attributed sets and grammars was strongly influenced by experience gained with this early prototype.

5.1.1 Environment and goals

My goal in 1991 was to create a prototype of a complete notation processing system, including a graphical editing capability, some kind of notation parser, and verifiable semantic interpretation of the parser's output. To save time and reduce the complexity of the problem, I chose to sacrifice full integration and interactivity in the first prototype. I developed a separate editor and parser, and chose as the interpreter a graphical programming system developed by two senior undergraduate students at the University of Western Ontario.

The graphical programming system, an "Iconic Shell Tool" (IST) for the UNIX operating system [8], was written in C for the X Window System [124, 125, 104]. The first version of IST was written in 1990 by Christin Boyd as her senior undergraduate

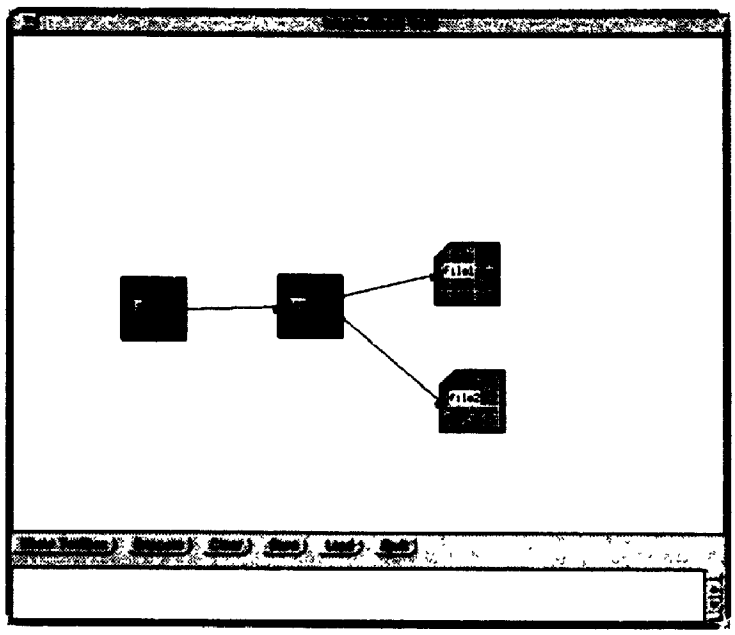


Figure 5.1: Appearance of the IST graphical programming system

project in Computer Science [9]. Corrections and enhancements were added in 1991 by Rodney Steensma, for his senior undergraduate project [137]. Both projects were done under my supervision.[1]

IST is an interactive program with a highly graphical user interface. An example of its on-screen appearance is shown in Figure 5.1. It allows interactive construction and editing of a diagram called a *cgraph* (computation graph), consisting of labelled icons linked by arrows. The icons may represent either files or programs; the arrows represent data flow. At any time, the user may request that the cgraph be executed, in which case it is first analyzed to determine that certain validity conditions are met. If the cgraph is valid, IST opens the appropriate files, runs the appropriate programs as separate processes, and establishes the necessary data flow connections using I/O redirection and/or pipes. It is also possible to save cgraphs in disk files and restore them at a later session.

[1]Lest I be accused of manipulating two hapless undergraduates for my own ends, I should like to mention that I conceived the IST concept, and supervised the first project, before I began my doctoral studies. While working in an image processing laboratory, I had written a suite of programs for basic image processing operations, which could be used in various combinations. Many of the researchers in the laboratory had great difficulty using these programs because they found the standard UNIX command-line interface too complex and unforgiving. I designed IST as an alternative.

Cgraphs can be thought of as directed graphs where each node (represented by an icon) is labelled and assigned one of eight types based on its label and its in- and out-degree, according to the following scheme:

1. In-degree 0, out-degree 1, label is a command string: type is *source*.
2. In-degree 1, out-degree 0, label is a command string: type is *sink*.
3. In-degree 1, out-degree 1, label is a command string: type is *filter*.
4. In-degree 2, out-degree 1, label is a command string: type is *merger*.
5. In-degree 1, out-degree 2, label is a command string: type is *splitter*.
6. In-degree 0, out-degree 1, label is a file name: type is *inputFile*.
7. In-degree 1, out-degree 0, label is a file name: type is *outputFile*.

The two arcs coming into merger nodes, and the two arcs leaving splitter nodes, are ordered; they are not interchangeable. In the IST display window, nodes are represented by icons in the IST display, arcs as arrows. IST uses distinct icons (selected by the user from an on-screen menu) to differentiate the different node types. These icons have "handles" at which arrow endpoints may be attached. By double-clicking with the mouse on a displayed icon, a window may be called up which allows editing the associated label (file name or command string).

In IST, there is no need for any kind of parsing, because the cgraph itself is not the input to the program. The input is the sequence of mouse gestures made by the user, which are interpreted in temporal and spatial context, yielding the cgraph (manifested in IST's internal data structures and as a picture on the screen) as a by-product. This is a common graphical user interface (GUI) programming approach, which is also used in computer-aided design (CAD) programs. An inherent problem with this approach is that such GUIs must be purpose-built, and few automated tools are available to make the programming easier. This is precisely the motivation for this thesis, as was mentioned in Chapter 1 and as will be discussed further in Chapter 6.

My objective for the prototype notation processing system was to create a new "front end" for IST, in which the input was purely graphical and syntactic distinctions (e.g. between node types and concerning the ordering of incoming and outgoing arcs)

was determined via a parsing technique. Because I was greatly interested in pen-based computer systems at the time, I decided to implement a graphical editor whose input was in the form of a stream of pen-strokes from a graphics tablet.

The pen-stroke editor was implemented in C++ on a personal computer (PC) running the MS-DOS operating system. A Wacom SD-421E digitizing tablet was used, connected to the PC via a high-speed serial link. The user could draw freehand on the tablet; the resulting strokes appeared immediately on the PC screen. Facilities for interactive selection of strokes were provided, with selected strokes displayed in a distinctive colour. All currently-selected strokes could be deleted by pressing a button. Pictures could be saved as disk files, and restored for further modification at a later editing session.

The tablet, which was capable of sampling the pen position 205 times per second with a resolution of about one one-thousandth of an inch, generated a prodigious amount of data. Some of the preprocessing techniques I implemented to reduce it are described in [41]. For the present discussion it suffices to say that the editor program could produce, on demand, an output file consisting of a list of line segment descriptions, each containing endpoint coordinates (x_1, y_1), (x_2, y_2). (This was distinct from the files used to save and restore pictures, in which additional detail was retained.)

5.1.2 Design of the input notation

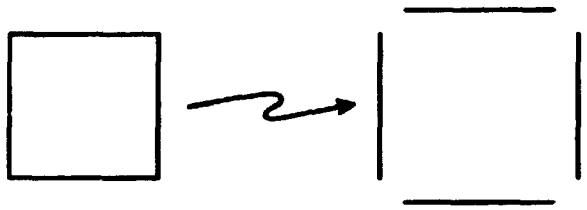
The job of the parser was to take the list of line segments produced by the editor, and "compile" it into a cgraph, which it would output as a file in the format used by IST to save and restore cgraphs. I did not attempt to deal with node labels in cgraphs—the parser only had to recognize the graph structure as described above. The notational system I chose for freehand drawing of cgraphs is illustrated in Figure 5.2, and characterized as follows:

1. Process nodes were represented by four line segments—two horizontal and two vertical—forming a rectangular box.

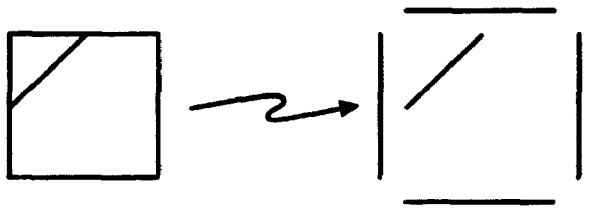
2. File nodes were also represented as rectangular boxes, with the addition of a fifth line segment cutting diagonally across the top-left corner.

3. Arcs were represented by three line segments—one long one for the shaft and two short ones forming a vee for the arrowhead.

Process icon: 4 line segments which (nearly) meet at corners



File icon: box plus a diagonal slash



Arc: 3 line segments forming shaft + arrowhead

Figure 5.2: Input notation for cgraphs

4. The distinction between node types (the eight types listed above) was determined contextually, according to the number of incoming and outgoing arrows.

5. For splitter and merger node types, the distinction between the two input or output arcs was made by comparing the appropriate arrow endpoint with the endpoints of the appropriate vertical box side, and testing whether the point of attachment was in the upper or the lower one-third of the box side.[2]

Important aspects of this notation are:

1. There is only one graphical primitive required—the line segment.

2. Icons and arrows can be recognized by finding groups of line segments which meet simple alignment criteria.

3. Determination of icon type is more complex, requiring analysis of the context (number of incoming and outgoing arrows) of each icon.

5.1.3 Technique: parse forest and stepwise parsing

The cgraph parser is written in the constraint logic programming language CLP(R) [67, 76, 68]. The complete source code for the program is included in Appendix A, along with a description of the CLP(R) programming language.

The parser works by manipulating a data structure called a *parse forest*, which is a collection of parse trees. Each parse tree corresponds to part of the input picture, i.e., a subset of the primitives composing it. The leaves of the trees correspond to input primitives, which are line segments in the cgraph case. Trees (and subtrees) correspond to identified configurations of input primitives, which for cgraphs include arrowheads, arrows, boxes, file icons, and the various types of process icons. All nodes carry attributes, as in a conventional attribute grammar [28, 1].

Initially, the parse forest consists of a set of isolated leaf nodes representing the line segments forming a picture. The parser works in steps. At each step, which is called a *reduction*, two or more trees in the parse forest are combined to form a single tree, as descendents of a new common root node. This is done according to one of

[2] In retrospect, this is too restrictive. It would have been better simply to distinguish which arc was uppermost and which was below.

several *reduction rules* which have the general form

> **if** there are root nodes $R_1, \ldots, R_m$ and $S_1, \ldots, S_n$ in the
> parse forest whose attributes satisfy a set C_A of constraints
> **then** create a new root node R (of the appropriate type) whose
> immediate descendents are $R_1, \ldots, R_m$ and whose attributes
> are computed from those of $R_1, \ldots, R_m$ and $S_1, \ldots, S_n$
> by solving the set C_P of constraints.

Constraints C_A are called the ***applicability constraints***, because they determine the applicability of the reduction. Constraints C_P are called the ***propagation constraints***, because they control the propagation of attributes up the parse trees. (In terms of conventional attribute grammars, the cgraph parser uses synthesized attributes only.)

After such a reduction, the nodes $R_1, \ldots, R_m$ are no longer classified as root nodes in the parse forest, but nodes $S_1, \ldots, S_n$ still are. The nodes $S_1, \ldots, S_n$ represent the *context* in which the nodes $R_1, \ldots, R_m$ are reduced to R. The S_i in this parsing method are essentially the same as "remote symbols" in Golin and Reiss's AMG, and "existentially quantified symbols" in Helm and Marriott's CMG (see Chapter 3). This element of context dependency is needed to enable determination of the type of each icon in a cgraph, as described above.

The reduction rules were easily encoded as logic procedures in CLP(R). For example, the procedure encoding how a tree representing an arrowhead and a (one-node) tree representing a line segment (shaft of the arrow) are combined to yield a new tree representing an arrow is shown in Figure 5.3. The first two lines (beginning with **extract**) find candidate pairs of arrowhead and line-segment objects among the roots of the current parse forest. The next two lines check that the length of the line segment exceeds an empirically chosen threshold. The next line calls a function **arrow_match** which succeeds (returns true) if the apex of the arrowhead meets one end of the line segment to within an empirically chosen tolerance. The CLP(R) interpreter repeatedly executes the procedure until an arrowhead and line segment are found which meet all these conditions, or until all possibilities have been exhausted. If a match is found, the last line of the procedure combines the two matched subtrees under a new root node of type "arrow".

The parser works by applying reductions repeatedly, until no more are applicable. Hence its output is not necessarily a single parse tree; it is just another forest. Moreover, note that the input forest need not consist entirely of isolated nodes. In the most general sense, the parser takes one forest as input and, provided at least

```
#define SHAFT_LENGTH_MIN 200
#define HEAD_SHAFT_MEET_TOLERANCE 50
reduce(arrow,Li,Ni,Ri,Lo,No,Ro) :-
    extract(arrowhead(Ahead,_,Ah),Ri,R1),
    extract(lineseg(Shaft,[],S1,S2),R1,R2),
    length(S1,S2,LS),
    LS >= SHAFT_LENGTH_MIN,
    arrow_match(S1,S2,Ah,HEAD_SHAFT_MEET_TOLERANCE,T,H),
    add_root(arrow(Li,[Ahead,Shaft],T,H),Li,Ni,R2,Lo,No,Ro).
```

Figure 5.3: Reduction rule for arrow

some reductions are applicable, produces as its output a new forest containing fewer and higher trees than the input forest. (If no productions are applicable, the output equals the input.)

5.1.4 Handling reduce-reduce conflict

The simplest kind of parser built according to this "tree-building" method would simply apply reductions irrespective of order, and stop when none of the available reduction rules are found to be applicable. Such an "unrestricted" parser would work only if, at every step except the terminating one, exactly one of the reduction rules was applicable. This was not the case for the cgraph notation as I defined it. The situation where a bottom-up parser must choose among two or more applicable reductions is called *reduce-reduce conflict.*

In some cases, reduce-reduce conflict can be eliminated by modifying the reduction rules themselves. For example, if the rule for process icons looks for four line segments forming a box, and the rule for file icons looks for five line segments, four of which form a box while the fifth cuts diagonally across its top-left corner, obviously both rules will be applicable when the input is a file icon. However, if the rules are changed so that there is one which combines four line segments into a "box" and another which looks for a box and another line segment to combine into a file icon, the reduce-reduce conflict is eliminated.

However, there were other cases with the cgraph notation where I could not find a way of formulating the reduction rules so as to eliminate all reduce-reduce conflicts. For example, the rule for a filter node looked for a box in the context of one incoming and one outgoing arrow, while the rule for a sink node looked only for an incoming

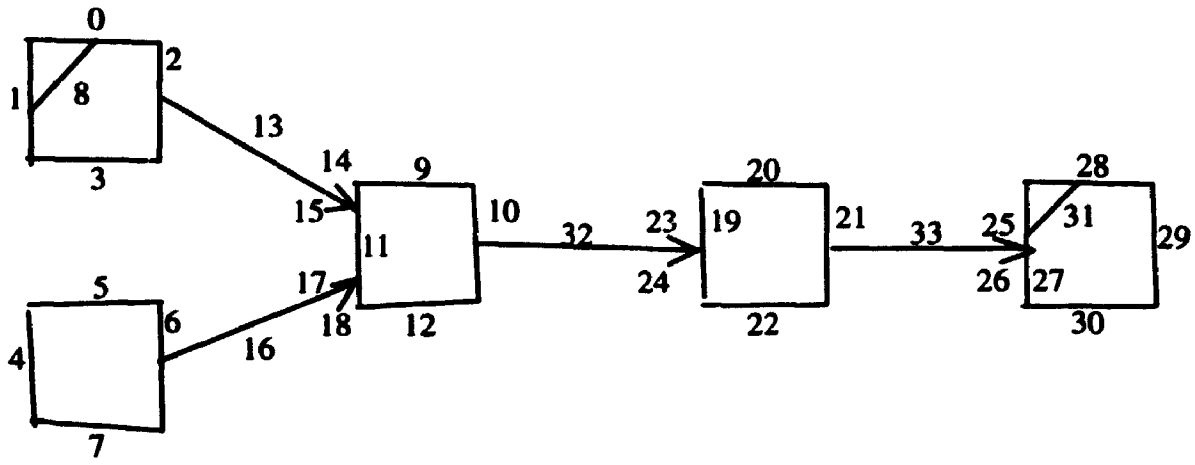


Figure 5.4: Cgraph as seen by parser

arrow. In both rules (and indeed all process node rules), the arrows had to be treated as context symbols and were not rewritten; this was absolutely necessary because each arrow has two ends, and therefore would have to be found a second time when the box at its other end was being reduced. I dealt with this problem by changing the parsing algorithm so as to impose an order on the reductions. For cgraphs it would first try to find all boxes, then all file icons, then all arrowheads, and so on.

5.1.5 Example: parsing a cgraph

Figure 5.4 shows a cgraph as seen by the parser. The figure has been reconstructed from an actual input file which was produced by the pen-stroke editor by reducing (to a set of 34 line segments) a hand-drawn cgraph similar to the figure.[3] A listing of the input file is included in Appendix A. In the list of line segments produced by the pen-stroke editor for input to the parser, the segments are numbered in the order in which they were input. In figure 5.4, these numbers are used to label the segments. Note that segments which belong together were not necessarily input consecutively, e.g. in two cases the shaft of an arrow was drawn much later than the arrowhead.

Figure 5.5 shows the parse forest produced by the parser from the input illustrated in figure 5.4. Again, this figure has been reconstructed from the actual parser output file (a listing of which is included in Appendix A). The parser actually produces two output files. The second one is in the format expected by IST, and can be loaded

[3] I was unable to find a hard-copy of the hand-drawn picture. It would have been similar, except that the lines would not be as straight, and the numeric labels would of course not be included.

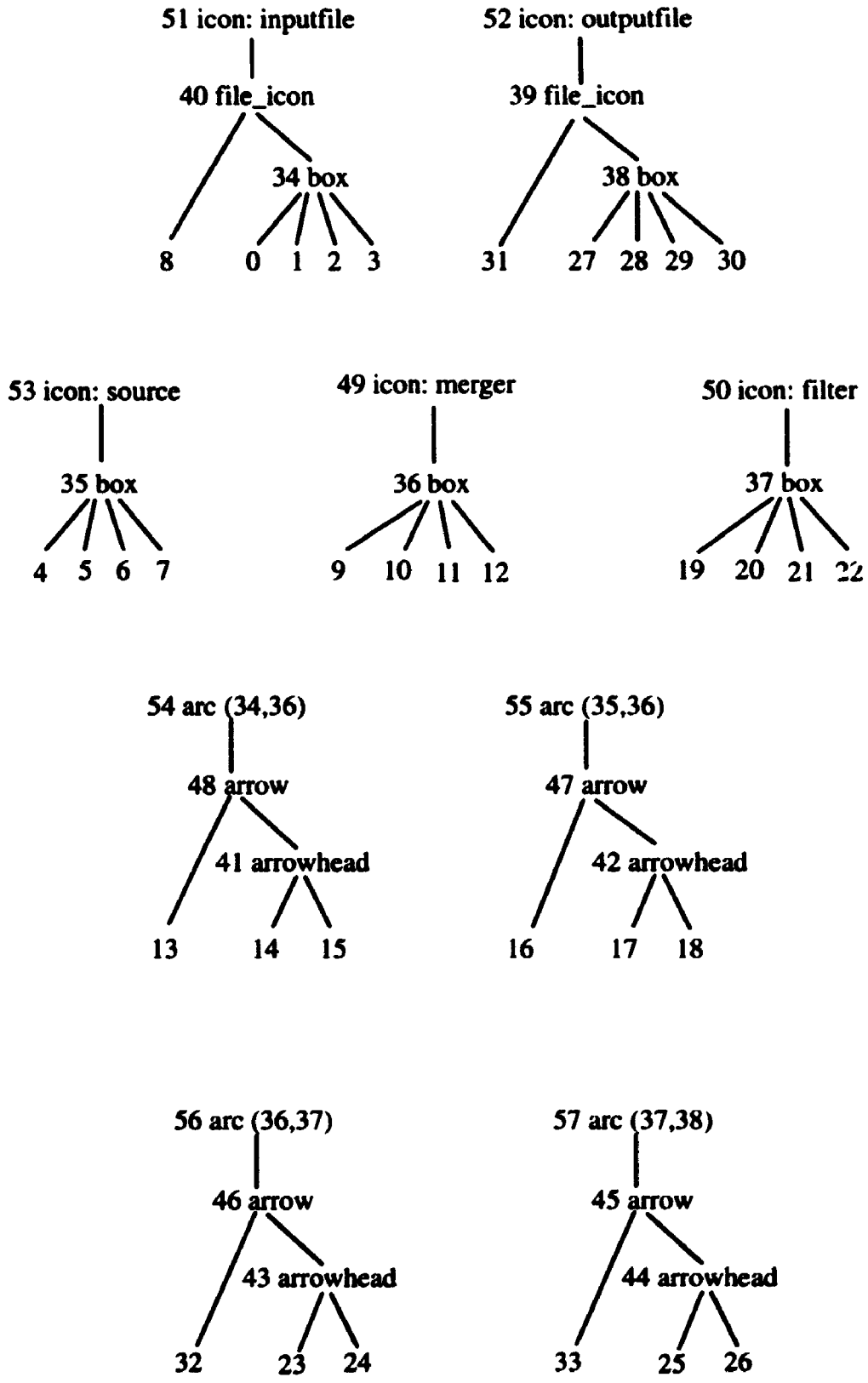


Figure 5.5: Parse forest produced for the cgraph

just like a saved IST cgraph file. It will appear with the proper icons and the same layout as the hand-drawn cgraph, but the icon labels (file names, command strings) will be blank. The latter can be added using IST, at which point the cgraph can be executed normally.

5.1.6 Comments on implementation 1

The implementation described above was intended to be simple enough to be completed in a couple of months, yet realistic enough to engage some of the problems which would be expected in a more elaborate implementation. In that respect, it was a success. In particular it allowed me to get some experience with the problem of reduce-reduce conflicts, which I discuss further in Section 5.4 below, and the problem of context dependency.

In the first prototype I was able to consider nearly all aspects of notation processing, including preprocessing (e.g. filtering of pen-stroke data), lexical processing (forming the list of line segments), syntax analysis (parsing), and semantic interpretation (importing the result into IST). The major limitation of the prototype was that I made no attempt to integrate the various phases of processing, and to deal with interactive operation. To do so would have involved a lot of programming, and so, rather than pursuing this aspect in a second implementation, I chose to devote all of my efforts to developing a formalism on which to base improved implementations.

I should also note that the "parser" in the first implementation was not really a parser, because it was not based on a grammar formalism. Rather, it was a kind of rewriting system, and in this respect was similar to Najork and Kaplan's CSRS (see Chapter 3).

5.2 Implementation 2: 1994–1995

At the end of Chapter 4 I presented a short Prolog program which implements the ASG-based acceptor algorithm suggested in the proof of Theorem 2, using the ASG from the digraph example discussed in Section 4.5.3. In this section I discuss converting that acceptor program into a parser, and adding support for constraints.

My implementation experiments were all done with CLP(R), which is in many ways a superset of Prolog. When I say that a program is "in Prolog", I mean that it is written in the "Prolog subset" of CLP(R), i.e., that it does not use any of CLP(R)'s

constraint features. Such programs should execute in a true Prolog environment, but I have not verified this.

5.2.1 From acceptor to parser

The entire Prolog source code for digraph parsing is given in Figure 5.6. Compare this with Figure 4.11.[4] An extra argument has been added to each of the **prod** procedures, specifying which production it encodes. The **accept** procedure has become **accept_print**, which, after each reduction, prints the new word (bound to **W2**, encoded as a Prolog term list) and the production just applied, numbered and with its variables replaced by the matched attribute values.

Note that **accept_print** prints its results during the unwinding of the recursion; the result is that a derivation is printed in forward order, beginning with the start word and progressing to the word which was input. The new procedure **parse** is intended to be used as a goal. It calls **accept_print** and then completes the derivation by printing the input word again.

Example 23: directed graph parsing

Figure 5.7 is a transcript of a run of the parser given in Figure 5.6. The program itself is loaded when the CLP(R) system is started up. The goal given following the prompt "`1 ?-`" is an attributed set model of the digraph given in Example 20 on page 97. The program's output is nearly identical to the derivation given in Example 20; there is a small difference in the order in which the productions are applied, which does not affect the final result.

5.2.2 Integrating constraint testing

Section 4.3.5 introduced a formal semantics of attributed sets (words) involving constraints. Section 4.5.5 gave an example (tree diagrams with layout constraints) showing how an ASG could generate such words, which could be "realized" through a subsequent constraint-solving process.

In the *generation* of attributed sets involving constraints, the constraint symbols could be treated just like any other symbols; their interpretation as constraints was

[4]For ease of comparison, all of the program listings included as figures are reproduced, together, in Appendix A.

```
% productions coded as prod(number,head,body)
prod(5,[n(V1),n(V2)],[n(V1),n(V2),arc(V1,V2)]).
prod(4,[n(V1)],[n(V1),arc(V1,V1)]).
prod(3,[n(V1)],[n(V1),n(_)]).
prod(2,[n(V1)],[node(V1)]).
prod(1,[n0],[n(_)]).

% parser as seen by user: prints derivation of W if one exists
parse(W) :-
   nl,accept_print(W),writeln(W).

% internal parser: acceptor augmented with printing functions
accept_print([n0]).
accept_print(W) :-
   prod(P,X,Y),
   match(Y,W,W1),
   union(X,W1,W2),
   accept_print(W2),
   writeln(W2),
   printf("p% : % -> %\n",[P,X,Y]),
   !.

% match(A,B,C)
% A is a subset of B, and C is its complement w.r.t. B
match([],B,B) :- !.
match([E|A],B,C) :-
   element(E,B,B1),
   match(A,B1,C).

% element(+E,+S,-T)
% succeeds if E is an element of set S; T is S minus E
element(E,[E|R],R).
element(E,[X|S],[X|T]) :-
   element(E,S,T).

% union(+X,+Y,?Z)
% Z is union of sets X and Y
union([],X,X).
union([X|R],Y,Z) :-
   element(X,Y,_), !,
   union(R,Y,Z).
union([X|R],Y,[X|Z]) :-
   union(R,Y,Z).
```

Figure 5.6: Digraph parser in Prolog

```
CLP(R) Version 1.2
(c) Copyright International Business Machines Corporation
1989 (1991, 1992) All Rights Reserved

1 ?- parse([node(1),arc(1,2),node(2),arc(2,2)]).

[n0]
p1 : [n0] -> [n(2)]
[n(2)]
p3 : [n(2)] -> [n(2), n(1)]
[n(2), n(1)]
p4 : [n(2)] -> [n(2), arc(2, 2)]
[n(1), n(2), arc(2, 2)]
p5 : [n(1), n(2)] -> [n(1), n(2), arc(1, 2)]
[n(2), n(1), arc(1, 2), arc(2, 2)]
p2 : [n(2)] -> [node(2)]
[n(1), arc(1, 2), node(2), arc(2, 2)]
p2 : [n(1)] -> [node(1)]
[node(1), arc(1, 2), node(2), arc(2, 2)]

*** Yes
```

Figure 5.7: Sample run of the Digraph parser

defined as a separate issue of semantics. In theory, the same is true for parsing. In practice, however, a parser will have to work with a database produced by some kind of graphical editor, which would normally contain only mark symbols with constant attributes.[5] That is, the parser's input will be comparable to a word which has been generated by the ASG *and* subsequently realized as a ground word. Furthermore, however, all the constraint symbols will have been stripped out of this ground word.

This does not greatly complicate the parsing process, however. The parser need only test, before reducing according to a production $X \rightarrow Y$, that all constraints in Y are satisfied by the constant attributes of the input symbols matched with Y. This was the approach used in my first implementation, and it works equally well for the second one. In CLP(R), constraints can be used as goals, so we can encode productions as rules where the rule head encodes the non-constraint aspects and the rule body encodes the constraint aspect. In effect, this integrates constraint testing into the parsing algorithm by making each reduction step conditional upon successful satisfaction of the constraints. This is essentially the same technique used by Wittenburg *et al.* in their Relational Language parsers, Golin and Reiss in their AMG/PLG parser, and Helm and Marriott in their CMG parsers (see Chapter 3).

Figure 5.8 gives the CLP(R) code for a parser for the binary tree diagram language given in Section 4.5.4, which integrates testing of constraints. Note that this listing is incomplete; the procedures **match**, **element**, and **union** are unchanged from Figure 5.6 and have been omitted.

New rules have been added to represent the constraints *equal, sum,* and *mid.* The encoding of productions via **prod** rules has been changed again, adding an argument **params** by which the symbolic environment variables **DX**, **DY**, **PX**, and **PY** could be bound to constants once and thereafter treated as global constants.[6] Corresponding changes have been made to *parse* and *accept_print.* Note how the all of the constraint symbols from each production body have been placed into the body of the corresponding **prod** rule, thereby becoming subgoals.

---

[5]Some kinds of graphical editors allow the user to specify alignment constraints for a picture. These kinds of editors could conceivably supply a "higher-level" picture description to a parser.

[6]For parsing, these are not really environment variables any more but true constants, analogous to the empirically chosen constants in the first parser implementation. The constant values could have been placed directly into the **prod** rules, obviating the need for the new **param** argument. This coding technique, however, is cleaner and facilitates experimental adjustment of the constants.

```
% constraints
equal(A,B) :- A=B.
sum(A,B,C) :- A=B+C.
mid(A,B,C) :- A=(B+C)/2.

% productions coded as prod(number,params,head,body)
prod(3,[DX,DY],
   [subtree(X,Y,LX,RX)],
   [node(X,Y),subtree(CXP,CY,LX,RXP),subtree(CXDP,CY,LXP,RX),
    arc(X,Y,CXP,CY),arc(X,Y,CXDP,CY)]
   ) :- sum(CY,Y,DY), sum(LXP,RXP,DX), mid(X,CXP,CXDP).
prod(2,[_,DY],
   [subtree(X,Y,LX,RX)],
   [node(X,Y),subtree(X,CY,LX,RX),arc(X,Y,X,CY)]
   ) :- sum(CY,Y,DY).
prod(1,[_,_],
   [subtree(X,Y,LX,RX)],
   [node(X,Y)]
   ) :- equal(X,LX), equal(LX,RX).

% parser as seen by user: prints derivation of W if one exists
parse(PARMS,W) :-
   nl,accept_print(PARMS,W),writeln(W).

% internal parser: acceptor augmented with printing functions
accept_print(_,[subtree(_,_,_,_)]).
accept_print(PARMS,W) :-
   prod(P,PARMS,X,Y),
   match(Y,W,W1),
   union(X,W1,W2),
   accept_print(PARMS,W2),
   writeln(W2),
   printf("p% : % -> %\n",[P,X,Y]),
   !.
```

Figure 5.8: Integrating constraint testing into the parser

```
test :- PX=0, PY=0, DX=4, DY=4,
    equal(X1,LX),equal(LX,X2),sum(Y2,Y1,DY),equal(X4,X6),
    equal(X5,X4), equal(X8,RX),equal(X7,X8),sum(Y3,Y1,DY),
    mid(X3,X5,X7),sum(X8,X6,DX), sum(Y1,PY,DY),mid(PX,X1,X3),
    sum(X4,X2,DX),
    writeln([node(PX,PY),arc(PX,PY,X1,Y1),arc(PX,PY,X3,Y1),
    node(X1,Y1),node(X1,Y2),node(X3,Y1),arc(X1,Y1,X1,Y2),
    arc(X3,Y1,X5,Y3),arc(X3,Y1,X7,Y3),node(X5,Y3),node(X7,Y3)]).

go :- parse([4,4],[
    node(0, 0), arc(0, 0, -3, 4), arc(0, 0, 3, 4), node(-3, 4),
    node(-3, 8), node(3, 4), arc(-3, 4, -3, 8), arc(3, 4, 1, 8),
    arc(3, 4, 5, 8), node(1, 8), node(5, 8)
    ]).
```

Figure 5.9: Constructing test input for the binary tree diagram parser

Example 24: Parsing a binary tree diagram

Figure 5.9 shows some CLP(R) code used to set up a test of the binary tree diagram parser listed in Figure 5.8 above.

My objective was to test the parsing of the binary tree diagram illustrated in Figure 4.8 on page 101, represented by a *ground version* of the attributed set model given in Example 22. Execution of the goal **test** produces such a ground word (with no constraint symbols), by solving the set of constraints taken from Example 22. Evaluation of **test** as a goal yields the ground word (encoded in Prolog/CLP(R) list form) from which the rule go was constructed. Evaluation of the goal **go** yields a derivation very similar to that given in Example 22.

Figure 5.10 is a transcript of a CLP(R) session (edited by adding spaces and line breaks to make it easier to read) showing execution of the goal **go** in context of the program listed in Figure 5.8. The derivation produced is similar to that given in Example 22; the only differences are in the order of production applications, which do not affect the final result. (A more complete transcript appears in Appendix A.)

The example above is contrived in the sense that I had to apply some programming (the goal **test**) to construct an input word which the tree diagram parser would

```
2 ?- go.

[subtree(0, 0, -3, 5)]

p3 : [subtree(0, 0, -3, 5)] -> [node(0, 0), subtree(-3, 4, -3, -3),
      subtree(3, 4, 1, 5), arc(0, 0, -3, 4), arc(0, 0, 3, 4)]

[subtree(3, 4, 1, 5), subtree(-3, 4, -3, -3), node(0, 0),
 arc(0, 0, -3, 4), arc(0, 0, 3, 4)]

p3 : [subtree(3, 4, 1, 5)] -> [node(3, 4), subtree(1, 8, 1, 1),
      subtree(5, 8, 5, 5), arc(3, 4, 1, 8), arc(3, 4, 5, 8)]

[subtree(5, 8, 5, 5), subtree(1, 8, 1, 1), subtree(-3, 4, -3, -3),
 node(0, 0), arc(0, 0, -3, 4), arc(0, 0, 3, 4), node(3, 4),
 arc(3, 4, 1, 8), arc(3, 4, 5, 8)]

p1 : [subtree(5, 8, 5, 5)] -> [node(5, 8)]

[subtree(1, 8, 1, 1), subtree(-3, 4, -3, -3), node(0, 0),
 arc(0, 0, -3, 4), arc(0, 0, 3, 4), node(3, 4), arc(3, 4, 1, 8),
 arc(3, 4, 5, 8), node(5, 8)]

p1 : [subtree(1, 8, 1, 1)] -> [node(1, 8)]

[subtree(-3, 4, -3, -3), node(0, 0), arc(0, 0, -3, 4),
 arc(0, 0, 3, 4), node(3, 4), arc(3, 4, 1, 8), arc(3, 4, 5, 8),
 node(1, 8), node(5, 8)]

p2 : [subtree(-3, 4, -3, -3)] -> [node(-3, 4), subtree(-3, 8, -3, -3),
      arc(-3, 4, -3, 8)]

[subtree(-3, 8, -3, -3), node(0, 0), arc(0, 0, -3, 4),
 arc(0, 0, 3, 4), node(-3, 4), [illegible](3, 4), arc(-3, 4, -3, 8),
 arc(3, 4, 1, 8), arc(3, 4, 5, [illegible]), node(1, 8), node(5, 8)]

p1 : [subtree(-3, 8, -3, -3)] -> [node(-3, 8)]

[node(0, 0), arc(0, 0, -3, 4), arc(0, 0, 3, 4), node(-3, 4),
 node(-3, 8), node(3, 4), arc(-3, 4, -3, 8), arc(3, 4, 1, 8),
 arc(3, 4, 5, 8), node(1, 8), node(5, 8)]

*** Yes
```

Figure 5.10: Sample run of the tree diagram parser

accept. This is because the constraints in the tree diagram grammar of Section 4.5.5 were designed for *generation*, not for parsing, and hence are too unforgiving. I chose to use this example as it is, however, because it is directly comparable to the ASG and example given in Section 4.5.5.

For parsing in practice, we would use a slightly different set of productions in which the constraints were relaxed, e.g. using inequality tests against empirically chosen constants (as in Implementation 1 above) in place of exact equality testing. Essentially this would involve changing the rules for goals `equal`, `sum`, and `mid` in the program.[7]

This suggests an interesting application. Suppose we were to construct a pair of ASG's $\Gamma_{\mathbf{gen}}$ and $\Gamma_{\mathbf{parse}}$ for some notational system, identical except that the former used tight constraints suitable for picture generation, while the latter used looser constraints suitable for parsing. Given a rough sketch of a notation, we could parse it according to $\Gamma_{\mathbf{parse}}$ and then *re-generate* the notation by deriving it in $\Gamma_{\mathbf{gen}}$, by applying the derivation sequence (of production numbers and assignments) found in the parse. This would be a form of *beautification* [107] for notations.

5.3 Data structures for ASG-based parsing

In this section I consider some of the requirements of a data structure for parsing attributed set languages defined by ASGs. A detailed treatment of this issue is beyond the scope of this dissertation and is left for future study.

The principal data structure used in my first prototype notation parser was the *parse forest*, described in Section 5.1.3 above. The parse forest is a set of parse trees, which the parser merges by creating new root nodes. To apply the parse forest concept to ASG-based parsing, we should first specify that the forest consists of "partial" parse trees, whose root node need not represent a symbol of the start class, i.e., the class of the single symbol in the start word of the ASG. It is worth noting at this point that in notation parsing applications, the start class need not represent an entire notation; the ASG could be defined so that the start class represents a symbol or common sub-structure used in notations. This is discussed further in Section 5.5 below.

[7] This is not quite sufficient, because there are places in the grammar where duplication of a variable in a production in effect *implies* an equality constraint, which would have to be reformulated as a test for approximate equality for parsing.

The tree structure arises naturally for "context-free" reductions, where one or more attributed symbols are rewritten as a single symbol. For "context-sensitive" reductions, where one or more attributed symbols are rewritten as a single symbol *within a specified context*, we can still use a tree structure (as I have done in the first implementation), by allowing the newly-created root node to contain special pointers to the nodes representing its context, but not making the context nodes children of the new root node in the usual sense. (The distinction is that after the reduction is performed, the context nodes are still roots, while the children of the newly-created root node are not.) As mentioned earlier this is analogous to Golin's notion of remote symbols and to existentially quantified symbols in the work of Helm and Marriott. Allowing the use of context pointers, in effect, turns a tree structure into a directed acyclic graph.

For general reductions, e.g. a reduction which rewrites three symbols as two new symbols, tree structures are inadequate. One approach is to generalize from a tree to an *acyclic hypergraph*. A hypergraph is a generalization of a directed graph, where the concept of an arc or "edge" is generalized to a "hyperedge" having arbitrarily many incoming and outgoing "tentacles", each of which can attach to a node. An ordinary digraph arc is a special case of hyperedge, having exactly one incoming and one outgoing tentacle. (For more details concerning hypergraphs, see [63] and [29].)

In the usual kind of parse tree, each arc represents a reduction. In a "parse hypergraph", each hyperedge would represent a reduction.[8] For incremental parsing, we can generalize from a single parse hypergraph to a *multi-component hypergraph*), i.e., a set of hypergraphs which the parser combines in a manner analogous to the combining of trees in a parse forest. As in the forest case, certain nodes are considered "roots" and are thus visible to the parser as candidates for combination via reductions. Reductions always introduce one new hyperedge and cause the nodes pointed to by its outgoing tentacles to no longer be considered roots. Unlike the forest case, a single reduction may create more than one new root.

Either the forest or the multi-component acyclic hypergraph data structures can be extended to keep track of multiple possibilities. This is discussed briefly in the next section.

---

[8]It is worth noting that the dual of a parse hypergraph is an ordinary digraph whose nodes represent reductions and whose arcs represent attributed symbols. This may be easier to represent as a data structure.

5.4 Handling reduce-reduce conflict and ambiguity in ASG-based parsing

The parsing algorithm introduced in acceptor form at the end of Chapter 4, and developed further in this chapter, uses backtracking to explore a tree of possible reduction sequences. Each path in the tree represents a different sequence of reductions; those sequences which end in a word matching the start word of the ASG in question qualify as derivations of the input word.

There would be no need for backtracking if, for any input, at most one reduction were applicable at every stage of the rewriting process. The need for backtracking arises because the ASG used by the parser may, in general, have one or both of the following two properties:

1. When it is possible that the parser may find more than one reduction applicable at some point in its rewriting process, we say the ASG has inherent reduce-reduce conflict.

2. When more than one sequence of reductions terminates with a word matching the start word of the ASG, we say the ASG is *ambiguous.*

Reduce-reduce conflict causes branching in the search tree. Ambiguity means that some words in the language of the ASG have multiple derivations.

5.4.1 Reduce-reduce conflict

Reduce-reduce conflict primarily affects parsing efficiency. All it does is create more paths in the search tree, which is explored exhaustively in the full backtracking algorithm.

In the second prototype parser, I found it useful to order the **prod** rules representing productions in such a way that reductions which rewrote several symbols would be favoured over those rewriting fewer symbols. This allowed quicker convergence to a solution for input words in the language but had no effect on efficiency for words outside the language.

In the first prototype, reduce-reduce conflict was more of a problem, as was described above. This was mainly because the first prototype did not make very effective use of backtracking. As already mentioned, the first prototype was not a true parser, but simply a rewriting system. Because it was not based on a grammar formalism, it had reduction rules but no start word. As a result the "parser" had no goal, and could not backtrack because there was no criterion of success or failure.

This kind of situation is bound to occur in interactive applications, if we would like to perform some reductions even when the input is incomplete, i.e., cannot yet be reduced to the start word of the ASG. This is discussed further in Section 5.5.

5.4.2 Ambiguity

When a word has more than one derivation in an ASG, a full backtracking parser will find them all. The second prototype is not a full backtracking parser, however, because of the cut at the end of the `accept_print` procedure.[9] Hence it will report at most one derivation for any input.

Most of the ASGs used as examples in this dissertation are ambiguous. In the examples in this chapter, the parser in every case reported a derivation which was slightly different from the derivation given in the corresponding example in Chapter 4. The differences, however, were slight, nothing more than permutations of the derivation steps.

In practical applications, ambiguity need not be a problem; it depends on how we choose to interpret derivations. In the next chapter I suggest that interpretation be done, as in compilers, via *semantic actions* associated with (certain) reductions. If the ambiguity is such that all of the derivations of an input word are just permutations of a common set of reductions, we may be able to ignore it if we can define the semantic actions such that they are order-independent. Furthermore, if semantic actions are associated only with certain key reductions, differences other than permutations might also be ignored, assuming this were appropriate.

In an interactive context, multiple derivations of a given input might give rise to different feedback to the user, who could then choose the most appropriate one. This is discussed briefly below, and further in Chapter 6.

---

[9] The cut is used primarily to avoid infinite recursion with uninstantiated variables, which I now realize could be eliminated by using the built-in **ground** predicate to fail in such situations. It does greatly simplify the recovery of derivations, however.

5.4.3 Dealing with reduce-reduce conflict and ambiguity

I can suggest six ways to approach the problems of reduce-reduce conflict and ambiguity in practice:

1. *Modify the grammar:* In some cases ambiguity can be eliminated by changing the grammar, e.g. by introducing a new nonterminal symbol.

2. *Program the grammar:* One can ensure that only certain programmed sequences of reductions will be applied. This is a powerful technique but corresponds in theory to programming the grammar, the effects of which may be difficult to analyze (and which are certainly beyond the theory of ASGs as developed in this dissertation).

3. *Ask the user:* In an interactive context, the user might be informed of the ambiguity and asked to choose among the various possibilities, either by cycling through them or by choosing from an auxiliary "menu".

4. *Search breadth-first:* In an interactive application, if we want to perform some reductions even when the input is not yet complete, we might use the user-query method just mentioned. Replacing the usual depth-first search backtracking strategy with a breadth-first one might present choices to the user in a more appropriate order.

5. *Track multiple possibilities:* The user-query method will be unacceptable if too many queries are generated. The parser would ideally keep track of a number of possibilies, automatically rejecting those which, following editing changes made by the user, are found to be untenable.

Tracking of multiple possibilies will require extensions to the parser's primary data structure, which might be a forest or a multi-component acyclic hypergraph as described in the previous section. One possibility is to permit several arcs or hyperedges representing possible reductions to be added,[10] perhaps linked into a list to indicate their relationship as a set of mutually-exclusive possibilities. I cannot at this point suggest how these extra possibilities would be automatically deleted from the structure; this will require experience and is an area for future investigation. In

[10] In the forest case the forest becomes a multi-component directed acyclic graph. In the hypergraph case there is no change to the logical structure.

the next section I make some suggestions concerning how multiple possibilities might be presented in the user interface.

5.5 Parsing in an interactive context

The two prototype parsers discussed above are *batch* parsers—they receive all of their input at once. In interactive applications, where the user edits a notation over a period of time, it may in some instances be useful to be able to begin parsing before the notation is complete, perhaps even updating the result of the parse as editing changes are made. This is called *incremental parsing.*

I envisage using ASG-based parsing, together with a simple graphics editing capability, as an *input technique* for notations. For example, in a computer-aided design (CAD) system for electronics, the user might use a graphics tablet to sketch circuit diagrams, rather than having to select component symbols from a large on-screen palette as is common in CAD today. With incremental parsing, the user's sketched symbols might be recognized as soon as they were completed, and immediately redrawn to inform the user of their successful recognition.

Incremental parsing introduces three problems which are not present in batch parsing:

1. The need to parse smaller units, e.g. individual component symbols instead of complete circuits in the electronics CAD example, and later combine the results of parsing in some way.

2. The need to handle deletions. When symbols are deleted from the input, the parser must respond by retracting any parsing decisions which depend on those symbols.

3. The need to back-track automatically when newly-added data are incompatible with previous parsing decisions. E.g., in the cgraph example, four line segments forming a box might be interpreted first as a process icon, but addition of a diagonal slash would clarify the user's intent to input a file icon.

The parse forest used in the first prototype was designed primarily to address the problem of combining smaller parsed structures into larger ones. In designing this structure I was strongly influenced by Lee's work on incremental parsing of programs

[86].[11] It is conceivable that the parse forest approach could be used interactively. Newly-added input data would create new single-node trees in the forest (one per graphical primitive). Re-application of the basic parsing algorithm (applying as many reductions as possible) would then build up additional trees having these new nodes as leaves.

Wittenburg [153] has proposed a straightforward approach to handling deletions. His basic data structure is a table (called a *chart*) containing entries for input elements plus *constituents*—entries indicating partially or completely matched reduction rules. Each constituent entry has an attribute called its *cover*, which identifies the subset of input primitives upon which it depends. When a primitive is deleted, all constituents whose cover contains that primitive are deleted also. A similar approach could be applied with a parse forest—deleting a leaf node would cause any nodes which were ancestors of that leaf, or which pointed to it as context, to be deleted also.

The third problem is the most subtle. In the parse forest technique, the parser looks only at the current set of roots in the parse forest. Once part of the input has been parsed the corresponding nodes in the forest are no longer roots. For instance, imagine that in the cgraph example suggested above, four line segments have been reduced to a *box* node, and this has been further reduced to some kind of process-icon node. A diagonal slash is then added. The parser is unable to reduce this new object because the *box* node with which it might be reduced is no longer a root. The solution is to delete the process-icon node, but how is the parser to know this? Faced with new input which it cannot reduce, the parser could

1. discard all but the leaf nodes and re-build the parse forest, or

2. discard leaf nodes corresponding to parts of the input picture in the neighborhood of the newly-added object, or

3. ask the user to identify which parts of the picture need to be re-parsed to incorporate the added material.

The problems of incremental parsing are interesting, but it is not clear that they will be worth solving for practical notation processing systems. Incremental parsing techniques have been tried for interactive programming environments (see Section

[11] Lee's approach, described in his 1986 Master's thesis, was actually much more ambitious. Whereas my first prototype allowed trees to be combined only by their roots, Lee suggested mechanisms which allowed more general merging of sub-trees.

2.5), but have not come into widespread use. Today, a typical programming environment permits the user to edit program units as text, and at any time activate a compiler, linker, etc. which operates as a batch process. Incremental techniques offer little advantage in such systems for two reasons:

1. With the exception of very large programming projects, the turnaround time for batch compilation is often acceptable.

2. The user has considerable control over turnaround time by choosing how large programs are to be divided into separately-compilable units.

Similar conditions may prevail in notation processing The ASG-based batch parsing algorithm discussed above has exponential time complexity, and hence is inherently less efficient than parsing algorithms used in compilers. However, notations will typically be small compared with programs—perhaps a few hundred graphical primitives as opposed to many thousands of lines of code. The guided parsing technique suggested in the next section would afford users the choice to divide the parsing of a large notation into parts, which is analogous to separate compilation.

The applicability and value of incremental parsing may also depend to some extent on the notational system used in an application. In some systems, such as symbolic mathematics, notations (e.g. equations) have such a regular, recursively defined structure that it makes sense to parse whole notations. In other systems, such as music, the meaning of notations is determined much more by semantic aspects than by syntax (structure). In such cases, it might be more appropriate to use parsing to identify meaningful subunits of the input (e.g. clefs, notes, accidentals, etc. in music), but leave the task of combining these subunits into a meaningful whole to the semantic level of processing. This is discussed further in Chapter 6. For both kinds of notational systems, incremental processing of some kind is required in an interactive system. However, whether it is the parsing or the semantic interpretation (or both) which is incremental might depend on the application.

5.6 Guided parsing

Parsing algorithms are often characterized as "top-down" or "bottom-up". A top-down parser attempts to construct a derivation for its input in forward order, by

repeatedly applying productions, beginning with the start word[12] and continuing until the entire input is reconstructed. A bottom-up parser attempts to construct a derivation in reverse, by repeatedly applying reductions, beginning with the entire input and continuing until only the start word remains. Both prototypes discussed in this chapter are bottom-up parsers, and hereafter I restrict my attention to bottom-up approaches.

These descriptions are adequate for batch parsing, but in an interactive context other possibilities become apparent:

1. The user might be given control over how long a sequence of reduction steps is applied, i.e., we clarify the meaning of the word "repeatedly".

2. The user might choose a subset of the input for parsing, i.e., we relax the absoluteness implied by the words "entire input".

I use the term *guided parsing* to describe the use of both of these variations.

Graphical editor programs commonly provide a means for the user to *select* or highlight a subset of the graphical primitives in a picture, in order to perform some operation on that subset. In a notation processing application of the kind I envisage, parsing would simply be one more operation which could be applied to the current selection. Each reduction step might have associated user feedback, e.g. the user's roughly-sketched notational symbol might be replaced in the display by a neater-looking one drawn by the computer. Changes of colour, line style, etc. could also be employed.

Consider the cgraph notation, and imagine a fully integrated, interactive cgraph processor application with a pen-stroke editor for input. Having sketched some amount of new input, the user might use the editor's selection mechanism to highlight four line segments and activate the parser to perform one reduction step, yielding a *box* node in the parse forest and replacing the user's four line segments with a neatly drawn rectangle. A second activation of the parser might reduce the *box* to a *source*, *filter*, etc., and replace the rectangle with the appropriate icon. The question of which applicable reduction to perform might at this point be put to the user; perhaps the user might be able to cycle through the choices, seeing the various icons appear in

[12] I use the ASG term. For string-generating grammars one would say "start symbol" or "initial letter".

turn, or perhaps a menu containing all of the icons might be popped up, from which the user would choose in one step.

The user might also be permitted to select an icon in the display and activate an "un-parse" function to undo the appropriate reduction(s) and restore the original four line segments. The parse/unparse operations are similar in concept to the group/ungroup operations available in most graphics editors, and would probably be as easy to use.

The two preceding paragraphs suggest that a guided parser would perform exactly one reduction at a time, but this is not the only possibility. In the egraph example, there is no reason why, having selected four line segments forming a rectangle, the user could not be presented immediately with a choice of icons. The fact that the grammar requires such a rewriting to be performed as two distinct reductions is immaterial. Multi-reduction sequences might be defined in either of two ways:

1. Certain nonterminals in the ASG could be designated as "interesting". The parser would apply reductions until one or more "interesting" symbols were obtained.

2. Certain sequences of reductions might be programmed into the parser. These sequences could either be chosen by the system designer, or, potentially, learned automatically by the system from observations of user behaviour.

The guided parsing approach offers four advantages:

1. The exponential time complexity of the backtracking ASG-based parsing algorithm is "tamed", because the user is allowed to choose how much of the input is parsed at one time.

2. The user may add any amount of annotation, decoration, notes, etc. to notations, and simply never select their component input primitives for parsing.

3. Ambiguity and reduce-reduce conflict are less serious, because the guided parsing technique allows the user to specify reduction choices in a natural way.

4. Cognitive overhead is reduced, because the user can defer parsing for as long as is desired. The user can sketch and edit freely while concentrating only on getting the input picture correct, and then spend a short time concentrating only on getting the parsing right before continuing the creative work.

Some of the issues raised in this section are discussed further in Chapter 6.

5.7 Summary and Conclusions

Attributed sets representing notations can indeed be parsed. I have demonstrated two prototype attributed set parsers, one of which predated the ASG paradigm and a second which is based upon it. Fully integrated, interactive implementations of attributed set parsers would have to deal with reduce-reduce conflict and ambiguity and, to some extent, would have to operate incrementally. The notion of guided parsing provides a useful approach for dealing with all of these issues within an integrated, interactive notation processing system. Data structures such as the multi-component acyclic hypergraph may be useful in such implementations, but further experience is needed to design suitable management techniques, especially to deal with multiple candidate reductions.

Chapter 6

A Generic Design for Interactive Notation Processors

In this chapter I propose a generic design for interactive notation processing applications, which can be customized for particular notations and applications. The design is based on concepts used and proven in interactive programming environments.

> *NOTE: The ideas presented in this chapter are far-reaching and will require much further development. I include them in this dissertation because they have guided my research and will provide directions for future investigation.*

6.1 INPs and concept of a generic design

I am interested in designing interactive computer programs, where a substantial part of the information communicated between user and computer takes the form of notations, as defined in Chapter 1. I call such programs "interactive notation processors" (INPs). Examples of INPs (and their corresponding notational systems) include the following:

- Computer-aided design (CAD) systems for electronic circuit design (standard electrical circuit diagrams, logic diagrams).
- Symbolic or numerical mathematics systems (positional mathematical notation, equations and/or inequalities).
- Interactive systems for music editing, composition, and/or analysis (common musical notation, percussion notations, other specialized music notations).
- Graphical programming systems (a graphical programming language).

- Technical document preparation systems (any notation which is to be published).

In the past, systems of this kind have been constructed in one of two ways:

- A text encoding of the notation is devised, and the application accepts and parses text files in this format.
- A special-purpose graphical user interface (GUI) is developed, permitting direct manipulation of a notation on a display screen.

The first approach has many advantages, not least of which is the fact that until recently good quality, inexpensive, graphical I/O devices were rare. Perhaps the biggest advantage is that a program accepting text input does not have to provide an editing interface of its own; any standard text editor can be used. This simplifies the task of creating new applications. It also frees users from the necessity to learn "yet another editing command language", but only by forcing them to learn "yet another special-purpose input language".

The second approach, used primarily for CAD systems, has the advantage that it allows users to work with the notations they already know, but the cost in development time is very high. Command languages for such systems also tend to be very complex, primarily because the input to the system is in fact the sequence of editing commands, interpreted in (various kinds of) context; the notation which appears on the screen is produced as a by-product of the command processing.

In this chapter I propose a third approach, which combines the advantages of the other two while avoiding their drawbacks. The ideas presented here are not specific to any particular notational system or application. By considering features which will be common to most INPs, I attempt to develop a generic INP design, which can be customized or tailored to specific applications and notational systems.

A well thought-out generic design can simplify the process of developing new applications, and encourages the development of new *tools* and *reusable program components* to streamline the instantiation process. Probably the most striking example of a successful generic design is that of a compiler, which today is customized for new programming languages and target systems using tools such as lexical-analyzer generators, parser generators, and even code-generator generators. Other examples of successful generic designs are *filters* under the UNIX operating system, and standardized GUIs which are being incorporated into all modern operating systems.

6.2 Development of a generic design

My proposal for a generic INP design is based on four observations. The first three relate to program development systems, the fourth to standardized GUI management systems.

1. A compiler translates one complex language to another using the method of *syntax-directed translation*, which breaks down into *lexical*, *syntactic*, and *semantic* processing phases. Most INPs will involve translation of an input notation into an internal data structure representing its meaning, and the concept of syntax-directed translation can be applied to this kind of translation also.

2. Interactive programming environments (IPEs) make better use of programmers' time, and are easier to use than batch-type systems. Hence an IPE is a better model than a compiler, upon which to base a generic INP design.

3. IPEs based on *textually-oriented program editing* are easier to understand and use than those based on *syntax-directed program editing*. Hence the former, suitably modified to *graphically-oriented notation editing*, is the better model upon which to base a generic INP design.

4. In modern operating systems for workstation and personal computers, interactive programs requiring text input can make use of system-supplied visual text editing facilities, thereby acquiring a powerful and widely-understood input editing capability and advanced features such as *cut and paste*, without the need for any special code.

The notion that translating a notation into a data structure which captures its meaning is similar to compiling a program is at the heart of all of the research initiatives described in Chapter 3, as well as my own research. The three distinguished phases of processing in a compiler have obvious analogues in notation translation, as follows:

The lexical phase for notations is the conversion of graphical primitives into attributed terminal symbols suitable for parsing.

The syntactic phase is parsing according to an ASG for the chosen attributed set representation of the notation.

The semantic phase is construction of the data structure which captures the meaning of the input notation.

Although the lexical phase is beyond the scope of this dissertation, I have considered some lexical aspects for notations in the course of my work—see [41].[1] This dissertation concentrates primarily on the syntactic phase.

In a typical modern compiler, the grammar for the input language is augmented with attributes which carry semantic information. Semantic processing takes place under the control of *semantic actions* associated with each grammar production, which are executed whenever a reduction is made according to that production (see [1]). The ASG formalism already provides for attributes, and, as described in the next section, semantic actions triggered by reductions can also be used for semantic processing of notations.

If compilers represent a good paradigm upon which to base a generic design for INPs, IPEs represent a better one. This is primarily because the compiler-like aspects of INPs are less important (to the development of useful systems) than the *interaction* aspects. As mentioned in Chapter 5, notations are likely to be small compared with computer programs, and hence the complexity of parsing is not such a pressing issue. But consider the INP applications suggested at the beginning of this chapter. Each one is the kind of system which people would use extensively for creative work. Hence interaction issues—primarily user productivity and cognitive overhead—will be very important to the success of INPs. Implementing notation editing, parsing and interpretation in anything but a tightly-integrated, seamless interactive environment is simply out of the question.

The notion of extending the IPE paradigm to structures other than text and computer programs is not new. Fraser [48, 49] proposed a "generalized text editor" for arbitrary data structures, Scofield [128] extended this concept to multiple editors, each associated with different classes in an object-oriented system, and Notkin [103] described how an entire operating system could be built around the concept. All three authors, however, were primarily concerned with textual encodings of data structures, and syntax-directed editing.

In all truly integrated IPEs the primary GUI looks and behaves like a visual editor. In almost all IPEs, this is a syntax-directed program editor, which can only

[1]The interested reader should also consult [21] for a perceptive overview of lexical aspects in visual language processing.

manipulate programs (not arbitrary texts) and in which editing commands effect syntactic transformations. This paradigm has been heavily criticized (see e.g. [147]), mainly on the grounds of cognitive overhead. Briefly, the argument is that syntax-oriented commands make common editing operations much harder to perform than they would be under an ordinary text editor. The logical alternative is to create an IPE whose program editor works like an ordinary text editor, and uses incremental parsing techniques to update an internal representation of a program's syntax as it is edited. This approach is described in detail by Lee [86].

The argument against the syntax-directed approach is just as strong for notations as it is for programs. Göttler [61] has described implementations of syntax-directed diagram editors; to me they sound difficult to use. The alternative is to implement INPs as *graphically oriented notation editors*, whose primary GUI is a fairly conventional graphics editor, linked with a parsing capability which updates an internal representation of a notation's syntax as it is edited.

Syntax-directed editors do offer some useful features. For example, the user can ask the system to insert a "template" for a given syntactic unit (e.g. program, procedure, expression) and then edit its particulars. Or, having selected some part of the text, the user can request that the selection be expanded to encompass the next-larger complete syntactic unit. Both of these features can easily be provided in a textually-oriented program editor, however. Similarly, syntax-oriented editing features for notations can be provided within a graphically oriented notation editor.

For pen-based computer systems, there is another compelling argument in favour of the graphically-oriented approach: it is far easier to draw symbols and notation structures than it is to specify them via syntax-oriented editing commands. At issue is the question of how best to utilize the pen input device, which is coupled to the inherent dexterity of the hand and fingers. Allowing the user to simply draw is the obvious approach. In the early 1990s several papers were published concerning "gestural" input techniques (see [73, 85]), involving detection and discrimination of coordinated pen movements; such techniques have not come into widespread use.

Today's operating systems offer reusable, standardized GUI components ranging from simple buttons buttons, pop-up menus, etc. to text editing subwindows. The latter greatly simplify the implementation of text-based applications. The generic INP design described in the next section includes a standard graphics editor component, which provides similar benefits for INP applications.

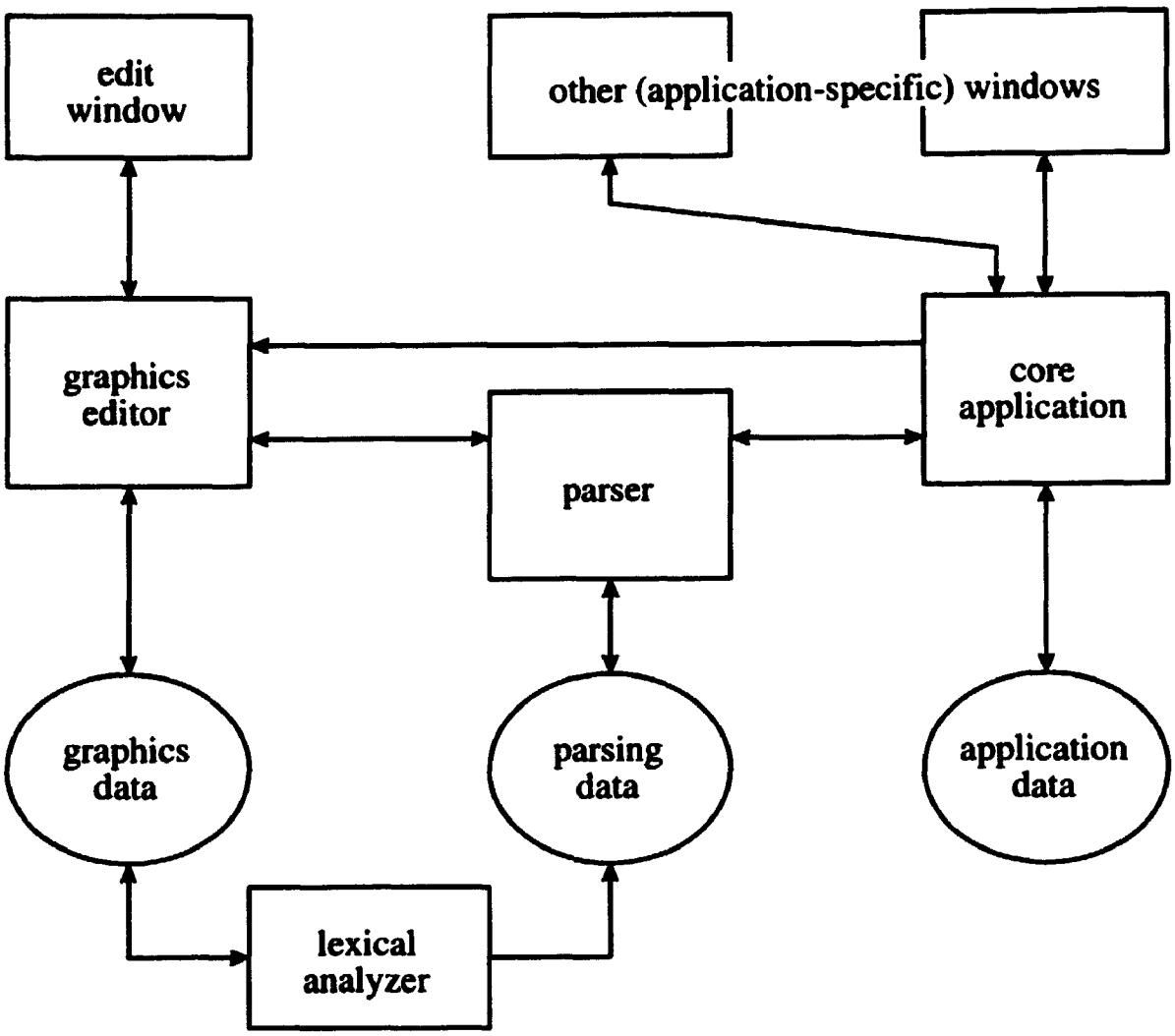


Figure 6.1: Schematic of the generic INP design

6.3 The proposed generic INP design

Figure 6.1 is a schematic diagram of my generic INP design, illustrating the major software components and how they communicate. The INP presents one or more windows on the screen. One window (or subwindow) looks and behaves as a simple graphical editor; the others contain GUI components specific to the application. The user may at any time select parts of the picture in the editor window and activate parsing on them, using the guided parsing method as described in Chapter 5. When certain reductions are performed, associated semantic actions cause messages to be sent to the core application, which responds by updating its internal data structures and/or sending return messages to the editor and/or the parser.

User interface issues for INPs are discussed in Section 6.5 below. I have addressed detailed design of the graphics editor module in a 1992 report [40]; hence I mention only a few key features here. The remainder of this section focusses on communication among the various components of the generic design illustrated in Figure 6.1.

The *graphics data* component in Figure 6.1 is a database of graphical primitives which together form one or more pictures. It should be possible to save the graphics data for a picture into a disk file and later restore it, and also to import and export such data (from the edit window to another owned by the same or a different application), under user control, via a *cut and paste* mechanism.

The *parsing data* component is the parser's representation of the current picture and its syntactic structure. Suggestions for appropriate data structures were made in the previous chapter (Section 5.3). Maintaining the lowest-level parts of the parsing data structure as editing changes are made, including appropriate format conversions, is the job of the *lexical analyzer* component, which is automatically activated every time new objects are added to the graphics data base.

The lexical analyzer assigns unique identifiers, which I call *handles*, for nodes it creates in the parsing data structure, and passes these back to the graphics data structure as attributes of graphics objects. This mechanism allows the graphics editor to identify its current selection to the parser, using the appropriate handles, and to respond to commands from the parser to delete or modify graphics objects identified by their corresponding handles.[2]

The *application data* component represents any data structures used by the *core application*. These are maintained entirely by the core application. Like the graphics data structure, the application data structures may contain references to objects in the parsing data component, in the form of handles as just discussed. (Handles for the lowest-level nodes in the parsing data structure are assigned by the lexical analyzer; the parser assigns handles for the rest.) This allows the application to identify objects in commands to both the parser and editor.

The *edit window* (which might be a sub-region of a larger window) presents, via the *graphics editor* module, a graphical view of the picture(s) stored in the graphics data module. User input associated with the edit window (and system input arising

[2]My first prototype used handles. The pen-stroke editor, when generating input for the parser as a list of line segment objects, assigned each a unique numeric identifier which was simply a non-negative integer. These integer handles actually reflected the order in which the segments were input by the user.

from e.g. cut and paste commands) is interpreted by the editor, as commands to navigate within the view, select objects, insert new objects and delete existing ones. User commands relating to parsing, as discussed in Section 5.6, would be passed on to the parser, along with appropriate parameters such as the current selection.

When the parser performs a reduction, a semantic action procedure associated with that reduction may send a message to the core application, indicating that a particular type of notational symbol has been recognized in the input. The message contains the handle to the newly-created node in the parser data structure. The core application responds to this "add object" message by updating its internal data structures, and replies with a message to the graphics editor, parser, or both, to arrange for appropriate user feedback, such as a change of colour or display style, or substitution of the user's sketched notational symbol with a neatly-drawn one from a special font or library.

When the parser undoes a reduction (e.g. in response to an explicit user request or because some input that reduction depends upon has been deleted), a second associated semantic action procedure is performed, sending a "delete object" message to the core application. The core application responds by updating its own data structures, and sending messages to the editor to provide appropriate user feedback.

The core application performs verification tests on its internal data structures after receiving each "add object" and "delete object" message. Verification following addition can be used for semantic checking, as is done in compilers, allowing the ASG used to construct the parser to define a somewhat larger language than the core application will actually accept. Verification failure might present the user with an error message, or send messages to the parser and/or editor to recover from the error in some way, or both. Verification following deletion could be used to trigger automatic unparsing of objects associated semantically with the deleted object.

As mentioned in the previous chapter, semantic validation also provides for a large degree of flexibility in determining which aspects of notational structure will be recognized syntactically (primarily by the parser), and which semantically (primarily by the core application). E.g. for applications involving symbolic mathematics, entire equations can be parsed, but systems of simultaneous equations are probably best dealt with at the semantic level. By contrast, in music applications, individual notes, chords, and other small notational structures could be parsed, but just about all other structural aspects would probably be easier to analyze semantically.

6.4 Customizing the generic design

Customizing the generic design just proposed, i.e., developing a design for a particular INP application, involves the following:

1. Precise definition of the notational system.
2. Design of the GUI and user-computer dialogues.
3. Decisions concerning the divisions between lexical, syntactic, and semantic aspects of notations.
4. Specification of the lexical analyzer component.
5. Writing an ASG for the parser component.
6. Design of the communication protocol for the system, including the message formats, senders/recipients, and conditions under which each kind of message will be sent.
7. Definition of semantic actions associated with (reduction according to) each production in the ASG (one for doing the reduction and one for undoing it) in terms of message sending.
8. Definition of how the editor, parser, and core application should respond to received messages.
9. Definition of the remaining aspects of the core application, e.g. functionality, data structures, algorithms.

The effort involved in implementing new INP applications can be greatly reduced if appropriate *tools* and *reusable components* are developed. "Tools" are programs such as automatic code generators, which can reduce development time by supporting programming at a very high and appropriate level of abstraction. "Reusable components" are libraries of source code and data structure declarations which can be linked (possibly dynamically) into new INP applications, or provided within the target operating environment as *server* processes.

A standard graphics editor could be provided in several ways, such as:

- Source code file(s) with documentation explaining how to add message sending and receiving functionality.
- A standard "widget" within a standard GUI manager such as X Windows, with a corresponding small library of interface routines and a documented application programming interface (API).

The standard graphics editor should include appropriate data structure definitions and routines to operate on graphics data, including hooks to interface with a user-supplied lexical analyzer. It should also have features analogous to those of standard text editing components, such as the ability to save graphics data to disk, read saved disk files, and export and import graphics data under user control via the operating system's cut and paste facility.

The complexity of lexical analysis depends very much on the types of objects manipulated by the graphical editor. If the editor provides only well-defined graphic object types such as line segments, rectangles, text strings, etc., lexical analysis is largely a matter of format conversions, and a standard lexical analyzer could be provided as part of the standard editor package. If the editor supports freehand sketching, e.g. with a pen-based interface, and the application designer anticipates the need for subtle interpretation of curve shapes, the lexical analyzer will necessarily be much more complex. Even in such cases, however, standard analyzer components could be provided, such as routines to:

- smooth and reduce the data requirements of strokes.
- reformat strokes as e.g. collections of line segments, using techniques such as curvature analysis.
- recognize widely-used notational symbols such as letters and digits.

The application programmer would add his or her own routines to the system-supplied ones, and would probably want to modify the lexical analyzer's top-level control algorithm.

The parser could be provided as a reusable component, taking specifications of the ASG, semantic actions and response to received messages, as run-time parameters. Language-specific parsers, however, would be more efficient, and could be generated

automatically via an ASG-based parser generator tool. Chok and Marriott [17] describe a parser generator based on the CMG formalism discussed in Chapter 3, which generates parsers in C++ source code. Development of such a parser generator for the ASG formalism is a logical next step for this research. Parsers generated by such a tool would of course include all necessary routines to establish and maintain the parser's data structures.

All that is left is the implementation of the core application. Even this could be simplified to some degree by provision of a standard API and library routines for message-passing, which the standard graphics editor and generated parser would also use. It would be best if the message-passing system offered debugging facilities to allow message-passing activity to be monitored during program development.

6.5 Some user interface issues

When I first began thinking about INPs in about 1989, I envisaged systems somewhat similar to today's pen-based "personal digital assistants" where notational symbols would be recognized automatically as soon as they were entered by the user. In time I realized that such an "eager" recognition strategy would not only be difficult to implement (primarily due to parsing ambiguity), but would also probably confuse the user. This led to the notion of guided parsing introduced in Chapter 5.

Another way in which my thinking about INP user interfaces has changed is that a few years ago I envisaged that special auxiliary views might be provided, e.g. a view displaying something like a parse forest, while today I think INPs should present very simple GUIs which focus attention on notations. The conscious decision *not* to present syntactic information directly in the user interface has strongly affected my formulation of the notion of guided parsing and of the generic INP design developed in this chapter.

Syntactic information can be presented to the user indirectly using highlighting. Having selected one or more graphical primitive objects in a notation, the user could ask the system to expand the current selection to include all objects covered by the next higher nonterminal in the parse structure. The four standard cursor-control keys on a typical terminal could be used to control syntactic selection and navigation as follows:

UP The up-arrow key expands the selection by moving to a higher syntactic level as just described.

DOWN The down-arrow key shrinks the selection by moving to a lower syntactic level. Most nonterminals would probably have more than one direct descendent in the parse structure; the system would choose one arbitrarily.

LEFT and RIGHT The left- and right-arrow keys select the next and previous sibling nonterminal at the same syntactic level.

This interface is similar to that proposed by van der Vegt [142] for editing drawings having a hierarchical group structure. Indeed, as noted in Chapter 5, parse/unparse operations are conceptually similar to group/ungroup, so it is not surprising that similar interface strategies can be used with both.

A similar interface can be used to guide parsing. The user selects a group of graphic objects and activates the parser, which attempts to apply a (sequence of) reductions. The result is indicated by a change in the current selection (and probably also by changes of colour and graphical style). At this point the cursor keys may be used as follows:

UP The parser applies a further reduction (sequence), thus moving up a syntactic level.

DOWN The most recently performed reduction is undone, thus moving down a syntactic level.

LEFT and RIGHT When there is reduce-reduce conflict, the left- and right-arrow keys select the next or previous candidate reduction, thus moving among choices at the same syntactic level.

These are ideas I have considered for INP user interfaces. How useful they are can only be determined through experience. In time, user interfaces for INPs might become standardized to the extent that support for them could be incorporated into the tools and reusable components for customizing the generic design.

Chapter 7

Conclusions and Future Work

In this dissertation I have described the goal of developing interactive notation processing applications (INPs), according to a "graphically oriented notation editor" paradigm adapted from the "textually oriented program editor" paradigm used in interactive programming environments. I have developed specific techniques for this purpose, including the attributed set representation, attributed set grammars (ASG), an ASG-based parsing algorithm, and a generic design for INPs.

7.1 Conclusions

I set out to prove that formal languages could be used to model structural aspects of notations, in such a way as to allow them to be recognized by a computer using a parsing technique, and I have succeeded in doing so. This required a reformulation the standard notions of formal language, generalizing from sequences of atomic symbols to sets of attributed symbols.

My hope was that notation recognition could be structured into three phases lexical, syntactic, and semantic—like compilation of programming languages, but I expected that interactive notation processing (INP) programs would be more like complete interactive programming environments than just compilers. The generic INP design presented in Chapter 6 explains this expectation, which frames the specific results of this thesis.

In Chapter 1 I made five specific claims concerning my techniques, which I now briefly address:

1. *ASGs can be used to model many notational systems.* I have dealt with two examples: directed graphs and tree diagrams. The former are similar to other network-structured notational systems such as electrical circuit diagrams, logic

diagrams, and flow charts.[1] The latter are similar to recursively defined notational systems having an underlying tree structure, such as mathematical expressions. Detailed consideration of specific notations is an area for future investigation.

2. *Attributed set encodings of notations are natural, intuitive, and well-suited for computation.* Attributed set encodings, especially those involving constraints, are much more natural and intuitive than encodings which have been attempted in the past, such as string encodings. Their usefulness for computation is amply illustrated by the ease with which ASG-based parsers are constructed.

3. *ASGs can be used to generate and parse encodings of notations.* Generation has been demonstrated in Chapter 4, parsing in Chapter 5.

4. *Practical problems with ASG-based parsing can be mitigated using guided parsing.* Chapter 5 explains the problems of reduce/reduce conflict, ambiguity, and incremental parsing of incomplete input, and shows how the guided parsing technique mitigates all three.

5. *The graphically-oriented notation editor paradigm for INPs is implementable in a manner consistent with good software engineering practice.* I have demonstrated ASG-based parsing via two batch-mode prototypes, discussed in Chapter 5. In Chapter 6 I suggested concrete methods for implementation of other aspects of the graphically-oriented notation editor paradigm, and suggested how specialized development tools and reusable software components can reduce the programming effort required to implement INPs.

7.2 Future Work

In the future, I would like to continue the development of the ASG formalism. One of my hopes for this dissertation was to present a formal proof that context-dependency increases the generative power of ASGs, using the digraph encoding language L_{digraph} discussed in Section 4.5.3 as the principal example. I came very close to completing a proof that no context-free ASG (one having only single symbols in production heads) can generate L_{digraph}, but ultimately gave up to concentrate on more pressing aspects

[1] The "cgraph" notation discussed in Chapter 5 is defined by rewriting rules which can easily be encoded in an ASG.

of the dissertation. I conjecture that a strict hierarchy of generative power, very similar to the Chomsky hierarchy of string-generating grammars, can be developed and proven for ASGs. I would like very much to obtain results concerning the decidability and (in decidable cases) computational complexity of the membership problem for specific classes of attributed set languages.

The backtracking ASG-based parsing algorithm described in Chapter 5 has exponential time complexity, which is at least manageable in practice, but I would also like to explore alternatives. Wittenburg's Relational Language approach, and Helm and Marriott's CMG approach, gain efficiency by making use of properties of relations to guide the selection of symbols for parsing. I suspect that, for attributed set languages involving constraints (which, in practice, will be most of them), similar techniques can be applied in ASG-based parsing.

The question of which notational systems can adequately be modelled using attributed sets and ASGs needs further exploration. I would like to try to develop attributed set encodings and ASGs for a variety of notational systems such as electrical schematic diagrams, logic diagrams, and perhaps music notation.

I should also like to explore applications of the ASG formalism outside notation processing. Possibilities include developmental modelling systems as suggested in Chapter 4, and notation beautification as suggested in Chapter 5. The proposed approach for notation beautification bears some resemblance to interactive graphical search and replace techniques proposed by Kurlander and Feiner [84]. Perhaps ASG-based methods might be applicable to the graphical search problem as well. Other potentially interesting applications of ASGs include modelling of three-dimensional structures, and non-graphical structures such as music.

In Section 4.5.1, I have begun the process of developing a formal semantics of attributed set modelling, but this needs further development in order to be mathematically rigorous. This is another area which I would like to explore.

Of course, the most obvious next step is implementation of some INP applications. One of the most fascinating aspects of this work, in my opinion, is the issue of how users can be shielded from the complexity inherent in the ASG-based, graphically oriented notation editor paradigm, and be given truly useful and usable INP applications. Related to this is development of tools and reusable software components to streamline the process of customizing the generic INP design, as discussed in Chapter 6.

The one component in the generic design which this dissertation leaves rather poorly specified is the lexical analysis module. It is in this phase that techniques of noise suppression, adapted from those developed in the field of pattern recognition, can be applied. Real understanding of the lexical analysis phase will come only in light of deeper understanding of the generative power of ASGs, and which kinds of lexical processing can usefully extend the generative power.

Finally, I chose not to address parsing uncertain input. My justification for this decision was that one must first understand the problems of notation processing without uncertainty, before one can properly deal with uncertainty (both in formalisms and in implementations) in effective ways. However, I cannot deny that working with forms of human writing, especially with technologies such as pen-based input systems which capture many subtleties, will involve uncertainty. For this reason, exploration of stochastic and fuzzy lexical and syntactic analysis techniques, and new developments in AI concerning reasoning with partial and conditional information, remains one of my long-term goals.

Appendix A

Listings of programs, input and output files

This Appendix collects some materials related to the prototype parser implementations and demonstrations of same presented in Chapter 5 of this dissertation.

A.1 Notes on the CLP(R) programming language

The code listings given here are written in the constraint logic programming language CLP(R) [67, 76, 68]. Constraint logic programming (CLP) languages are a variant of automatic theorem provers such as Prolog [23, 75], where unification is replaced by a more general constraint-solving mechanism. The symbol in parentheses identifies the domain of computation in which constraints may be specified, e.g. in CLP(R) the "R" refers to the real domain `R`.

In CLP(R) it is possible to match the terms `f(Y,10)` and `f(X+7,X)`, for example, giving rise to the pair of constraints $Y = X + 7$ and $X = 10$, which are immediately solved so as to bind variable X to the value 10 and Y to the value 17. One may also use constraints as goals, e.g., the goal

```
2X+Y=17, 8Y-3X=41.
```

would succeed with X bound to 5 and Y bound to 7.

The CLP(R) system is capable of dealing with quite complex systems of constraints, including inequalities and nonlinear expressions such as exponentiation, transcendental functions, and so on. Systems of linear constraints are solved using the Simplex algorithm. Evaluation of nonlinear constraints is automatically delayed until, via substitutions of known values, they can be reduced to linear form.

The syntax of CLP(R) is nearly identical to that of Prolog. I have described some programs included here as "in Prolog", meaning that I implemented them in CLP(R), but without using any of the constraint mechanisms. Hence these programs should be executable by a standard Prolog interpreter with identical results, but I have not tested this.

A.2 The first prototype parser

The CLP(R) source files here were processed by the UNIX C preprocessor, and hence contain directives such as **#define** and **#include** familiar to C programmers.

A.2.1 File parse.clpr

This is the main file for the prototype parser.

```
#include "list.clpr"
#include "reduce.clpr"
#include "output.clpr"

/* restricted bottom-up parser: The first argument to restricted_parse
   is a list of nonterminal classes to be reduced (by reduce_all, whose
   first argument is a single class) in order.
*/
reduce_all(Class,Li,Ni,Ri,Lo,No,Ro) :-
    reduce(Class,Li,Ni,Ri,Lt,Nt,Rt),
    reduce_all(Class,Lt,Nt,Rt,Lo,No,Ro).
reduce_all(_,Li,Ni,Ri,Li,Ni,Ri).

restricted_parse([],Li,Ni,Ri,Li,Ni,Ri).
restricted_parse([H|T],Li,Ni,Ri,Lo,No,Ro) :-
    reduce_all(H,Li,Ni,Ri,Lt,Nt,Rt),
    restricted_parse(T,Lt,Nt,Rt,Lo,No,Ro).
restricted_parse([],Li,Ni,Ri,Li,Ni,Ri).


/* general bottom-up parser */
parse(Li,Ni,Ri,Lo,No,Ro) :-
    reduce(_,Li,Ni,Ri,Lt,Nt,Rt),
    parse(Lt,Nt,Rt,Lo,No,Ro).
parse(Li,Ni,Ri,Li,Ni,Ri).
```

```
/* main program: batch parser */
:- dynamic(nodelist,2).
::- consult(seg_db).          % establish nodelist/2
::- nodelist(Ni,Li),
    restricted_parse(
        [dot,box,file_icon,arrowhead,arrow,merger,splitter,filter,
         inputfile,outputfile,source,sink,arc],
        Li,Ni,Ni,Lo,No,Ro),
    tell(parse_result),
    convert_result(No,Ro),
    told,
    tell(parse_forest),
    print_pf(Lo,No,Ro),
    told.
::- halt.
```

A.2.2 File list.clpr

This file contains code to manipulate lists, and the routine `print_pf` used in the main program to produce the output file `parse_forest` (see later) which outputs a text representation of the parse forest.

```
#ifndef LIST
#define LIST

/* member(X,List)
   - true if X is a member of List
   - modes: any
   e.g. member(+X,+List) true if X in List
        member(-X,+List) instantiates X to all members of List
*/
member(M, [M | _]).
member(M, [_ | T]) :-
   member(M, T).

/* extract(X,List1,List2)
   - true if X is a member of List1, List2 is result of deleting that
     member from the list
   - modes: any
   e.g. extract(+X,+List,-Out) instantiates Out or fails
   if X not in List
*/
extract(H, [H | T], T).
extract(M, [H | T], [H | T2]) :- extract(M, T, T2).
```

```
/* printlist(List)
   - always true (unless arg not a list)
   - prints elements of List one per line
   - modes: (+) is the only sensible one
*/
printlist([]).
printlist([H | T]) :- print(H), nl, printlist(T).

/* print_pf(L,N,R)
   - L,N,R form a parse forest
   - prints forest as an indented list
   - modes: (+,+,+) is the only sensible one
*/
print_pf(L,N,R) :-
    printf("Parse forest (% nodes):\n",[L]),
    convert_rootlist(R,[],R_ids),
    print_trees(N,R_ids,0).

/* convert a rootlist (1st arg) to a list of node id's (3rd arg) */
convert_rootlist([H|T],IDLi,IDLo) :-
    arg(1,H,Id),
    convert_rootlist(T,[Id|IDLi],IDLo).
convert_rootlist([],IDLi,IDLi).

/* print all trees with nodes in nodelist N given list of
   root node id's */
print_trees(N,[H|T],Indentlevel) :-
    print_tree(N,H,Indentlevel),
    print_trees(N,T,Indentlevel).
print_trees(_,[],_).

print_tree(N,Root_id,Indentlevel) :-
    member(Root_node,N),
    arg(1,Root_node,Root_id),
    indent(Indentlevel),
    print(Root_node),nl,
    arg(2,Root_node,Children),
    print_trees(N,Children,Indentlevel+1).

indent(Level) :-
    Level > 0,
    printf("   ",[]),
    indent(Level-1).
indent(0).
```

```
/* print all nodes in a list, such that the principal functor of the
   node descriptor matches a given one
*/
print_matching(Functor,[H|T]) :-
    print_if_match(Functor,H),!,
    print_matching(Functor,T).
print_matching(_,[]).

print_if_match(Functor,Term) :-
    functor(Term,Functor,_),
    print(Term),nl.
print_if_match(_,_).
#endif
```

A.2.3 File reduce.clpr

This file contains the various reduction (rewriting) rules of the parser, coded as CLP(R) rules of type **reduce**.

```
#include "geometry.clpr"

/* add_root(New,Li,Ni,Ri,Lo,No,Ro)
   - New is a node (structured term)
   - <Li,Ni,Ri> is a parse forest
   - <Lo,No,Ro> is the parse forest when the node New is added to both
     the nodelist Ni and rootlist Ri of <Li,Ni,Ri>
   - e.g. mode (+,+,+,+,-,-,-) add a new root
   - This procedure is called at the end of all reduce/5 procedures
*/
add_root(New,Li,Ni,Ri,Lo,No,Ro) :-
    Lo = Li + 1,
    No = [New | Ni],
    Ro = [New | Ri].


/* reduce/5 : (Class,Li,Ni,Ri,Lo,No,Ro)
   - <Li,Ni,Ri> is a parse forest
   - <Lo,No,Ro> is the parse forest which results after performing a
     reduction on <Li,Ni,Ri>, yielding a new root of Class
   - intended modes:
        (+,+,+,-,-) apply a given production once
        (-,+,+,-,-) apply any applicable production once
   - The textual order of reduce rules in this file imposes an order
     of precedence among them: rules listed first have precedence.
   - Each of the following reduce declarations has a header comment
```

```
     showing the form of the node term which is added to the parse
     forest.
*/

/* dot(_,_,P) : a very short lineseg, average location P
*/
#define DOT_TOLERANCE 10
reduce(dot,Li,Ni,Ri,Lo,No,Ro) :-
    extract(lineseg(Id, [], P1, P2), Ri, Rt),
    close(P1,P2,DOT_TOLERANCE),
    P = avg(P1,P2),
    add_root(dot(Li,[Id],P),Li,Ni,Rt,Lo,No,Ro).

/* box(_,_,NW,NE,SW,SE) : a box and its four corners
*/
#define CORNER_TOLERANCE 50
#define MIN_SIDE_LENGTH 200
#define MAX_SIDE_DEVIATION 50
reduce(box,Li,Ni,Ri,Lo,No,Ro) :-
    extract(lineseg(N, [], P1, P2), Ri, R1),
    horiz(P1,P2,MIN_SIDE_LENGTH,MAX_SIDE_DEV,NW2,NE1),
    extract(lineseg(E, [], P3, P4), R1, R2),
    vert(P3,P4,MIN_SIDE_LENGTH,MAX_SIDE_DEV,NE2,SE1),
    close(NE1,NE2,CORNER_TOLERANCE),
    extract(lineseg(S, [], P5, P6), R2, R3),
    horiz(P5,P6,MIN_SIDE_LENGTH,MAX_SIDE_DEV,SW1,SE2),
    close(SE1,SE2,CORNER_TOLERANCE),
    extract(lineseg(W, [], P7, P8), R3, R4),
    vert(P7,P8,MIN_SIDE_LENGTH,MAX_SIDE_DEV,NW1,SW2),
    close(SW1,SW2,CORNER_TOLERANCE),
    close(NW1,NW2,CORNER_TOLERANCE),
    avg(NW1,NW2,NW),
    avg(NE1,NE1,NE),
    avg(SW1,SW2,SW),
    avg(SE1,SE2,SE),
    add_root(box(Li, [N,E,S,W], NW, NE, SW, SE),Li,Ni,R4,Lo,No,Ro).
```

```
/* file_icon(_,_,NW,NE,SW,SE) : as box but with a diagonal slash
*/
#define MEET_TOLERANCE 50
reduce(file_icon,Li,Ni,Ri,Lo,No,Ro) :-
    extract(box(Box, _, NW, NE, SW, SE), Ri, R1),
    extract(lineseg(Slash, [], P1, P2), R1, R2),
    sort_lr(P1,P2,PL,PR),
    inbbox(PL,NW,SW,MEET_TOLERANCE),
    inbbox(PR,NW,NE,MEET_TOLERANCE),
    add_root(file_icon(Li,[Box,Slash],NW,NE,SW,SE),Li,Ni,R2,Lo,No,Ro).


/* arrowhead(_,_,P) : an arrowhead with vertex P
*/
#define AHEAD_VERTEX_TOLERANCE 30
#define AHEAD_LENGTH_MAX 100
reduce(arrowhead,Li,Ni,Ri,Lo,No,Ro) :-
    extract(lineseg(Side1,[],P11,P12),Ri,R1),
    length(P11,P12,L1),
    L1 <= AHEAD_LENGTH_MAX,
    extract(lineseg(Side2,[],P21,P22),R1,R2),
    length(P21,P22,L2),
    L2 <= AHEAD_LENGTH_MAX,
    pick_vertex(P11,P12,P21,P22,ARROW_VERTEX_TOLERANCE,P),
    add_root(arrowhead(Li,[Side1,Side2],P),Li,Ni,R2,Lo,No,Ro).

/* arrow(_,_,T,H) : an arrow with tail at H, head at P
*/
#define SHAFT_LENGTH_MIN 200
#define HEAD_SHAFT_MEET_TOLERANCE 50
reduce(arrow,Li,Ni,Ri,Lo,No,Ro) :-
    extract(arrowhead(Ahead,_,Ah),Ri,R1),
    extract(lineseg(Shaft,[],S1,S2),R1,R2),
    length(S1,S2,LS),
    LS >= SHAFT_LENGTH_MIN,
    arrow_match(S1,S2,Ah,HEAD_SHAFT_MEET_TOLERANCE,T,H),
    add_root(arrow(Li,[Ahead,Shaft],T,H),Li,Ni,R2,Lo,No,Ro).


/* The following productions are context-sensitive.  Their purpose
   is to reclassify box, file_icon and arrow nodes as cgraph elements
   based on context.  In each case, more than one extract operation is
   performed, but only the result of the first one is used to form the
   post-reduction rootlist.
```

```
   I have omitted negative constraints from the rules for filters,
   mergers and splitters, to keep the rules simple.  This means mergers
   and splitters must be reduced before all other process icons.

   All the icon nodes are represented as structured terms of the form

   icon(<id>,[<child id>],<icon class name>,<child box id>,NW,NE,SW,SE)

   where <id> is the usual node id, there is but one child in the child
   list, <icon class name> is one of {splitter, merger, etc.}, <box id>
   is the node id of the underlying box.  This is important for later
   conversion to IST format, because arcs (see below) also refer to box
   id's.  The corner points NW,NE,SW,SE are as for the underlying box.
*/
#define ARROW_BOX_MEET_TOLERANCE 50

/* icon(_,_,merger,boxid,NW,NE,SW,SE) : a merger icon
*/
reduce(merger,Li,Ni,Ri,Lo,No,Ro) :-
    extract(box(Box,_,NW,NE,SW,SE),Ri,R1),
    extract(arrow(_,_,_,Ahead1),R1,R2),
    inbbox(Ahead1,NW,SW,ARROW_BOX_MEET_TOLERANCE),
    extract(arrow(_,_,_,Ahead2),R2,R3),
    inbbox(Ahead2,NW,SW,ARROW_BOX_MEET_TOLERANCE),
    extract(arrow(_,_,Atail,_),R3,_),
    inbbox(Atail,NE,SE,ARROW_BOX_MEET_TOLERANCE),
    add_root(icon(Li,[Box],merger,Box,NW,NE,SW,SE),Li,Ni,R1,Lo,No,Ro).

/* icon(_,_,splitter,boxid,NW,NE,SW,SE) : a splitter icon
*/
reduce(splitter,Li,Ni,Ri,Lo,No,Ro) :-
    extract(box(Box,_,NW,NE,SW,SE),Ri,R1),
    extract(arrow(_,_,_,Ahead),R1,R2),
    inbbox(Ahead1,NW,SW,ARROW_BOX_MEET_TOLERANCE),
    extract(arrow(_,_,Atail1,_),R2,R3),
    inbbox(Atail1,NE,SE,ARROW_BOX_MEET_TOLERANCE),
    extract(arrow(_,_,Atail2,_),R3,_),
    inbbox(Atail2,NE,SE,ARROW_BOX_MEET_TOLERANCE),
    add_root(icon(Li,[Box],splitter,Box,NW,NE,SW,SE),Li,Ni,R1,Lo,No,Ro).
```

```
/* icon(_,_,filter,boxid,NW,NE,SW,SE) : a filter icon
*/
reduce(filter,Li,Ni,Ri,Lo,No,Ro) :-
    extract(box(Box,_,NW,NE,SW,SE),Ri,R1),
    extract(arrow(_,_,Atail,_),R1,R2),
    inbbox(Atail,NE,SE,ARROW_BOX_MEET_TOLERANCE),
    extract(arrow(_,_,_,Ahead),R2,R3),
    inbbox(Ahead,NW,SW,ARROW_BOX_MEET_TOLERANCE),
    add_root(icon(Li,[Box],filter,Box,NW,NE,SW,SE),Li,Ni,R1,Lo,No,Ro).

/* icon(_,_,inputfile,boxid,NW,NE,SW,SE) : an input-file icon
*/
reduce(inputfile,Li,Ni,Ri,Lo,No,Ro) :-
    extract(file_icon(Ficon,[Box,_],NW,NE,SW,SE),Ri,R1),
    extract(arrow(_,_,Atail,_),R1,R2),
    inbbox(Atail,NE,SE,ARROW_BOX_MEET_TOLERANCE),
    not(
        extract(arrow(_,_,_,Ahead),R2,_),
        inbbox(Ahead,NW,SW,ARROW_BOX_MEET_TOLERANCE)
        ),
    add_root(icon(Li,[Ficon],inputfile,Box,NW,NE,SW,SE),
     Li,Ni,R1,Lo,No,Ro).

/* icon(_,_,outputfile,boxid,NW,NE,SW,SE) : an output-file icon
*/
reduce(outputfile,Li,Ni,Ri,Lo,No,Ro) :-
    extract(file_icon(Ficon,[Box,_],NW,NE,SW,SE),Ri,R1),
    extract(arrow(_,_,_,Ahead),R1,R2),
    inbbox(Ahead,NW,SW,ARROW_BOX_MEET_TOLERANCE),
    not(
        extract(arrow(_,_,Atail,_),R2,_),
        inbbox(Atail,NE,SE,ARROW_BOX_MEET_TOLERANCE)
        ),
    add_root(icon(Li,[Ficon],outputfile,Box,NW,NE,SW,SE),
     Li,Ni,R1,Lo,No,Ro).
```

```
/* icon(_,_,source,boxid,NW,NE,SW,SE) : a source icon
*/
reduce(source,Li,Ni,Ri,Lo,No,Ro) :-
    extract(box(Box,_,NW,NE,SW,SE),Ri,R1),
    extract(arrow(_,_,Atail,_),R1,R2),
    inbbox(Atail,NE,SE,ARROW_BOX_MEET_TOLERANCE),
    not(
        extract(arrow(_,_,_,Ahead),R2,_),
        inbbox(Ahead,NW,SW,ARROW_BOX_MEET_TOLERANCE)
        ),
    add_root(icon(Li,[Box],source,Box,NW,NE,SW,SE),
     Li,Ni,R1,Lo,No,Ro).

/* icon(_,_,sink,boxid,NW,NE,SW,SE) : a sink icon
*/
reduce(sink,Li,Ni,Ri,Lo,No,Ro) :-
    extract(box(Box,_,NW,NE,SW,SE),Ri,R1),
    extract(arrow(_,_,_,Ahead),R1,R2),
    inbbox(Ahead,NW,SW,ARROW_BOX_MEET_TOLERANCE),
    not(
        extract(arrow(_,_,Atail,_),R2,_),
        inbbox(Atail,NE,SE,ARROW_BOX_MEET_TOLERANCE)
        ),
    add_root(icon(Li,[Box],sink,Box,NW,NE,SW,SE),
     Li,Ni,R1,Lo,No,Ro).

/* arc(_,_,src-id,src-num,src-class,dest-id,dest-num,dest-class)
   An arc src->dest.
   Both -id values are the id's of the boxes underlying icons.
   Both -num values are taken from the set {first,only,second}.
   The  -class values are the icon class (inputfile, sink, etc.)
   at each end.
*/
reduce(arc,Li,Ni,Ri,Lo,No,Ro) :-
    extract(arrow(Arrow,_,Atail,Ahead),Ri,R1),
    extract(icon(_,_,Sclass,Sid,_,TNE,_,TSE),R1,R2),
    inbbox(Atail,TNE,TSE,ARROW_BOX_MEET_TOLERANCE),
    con_num(Atail,TNE,TSE,Snum),
    extract(icon(_,_,Dclass,Did,HNW,_,HSW,_),R2,_),
    inbbox(Ahead,HNW,HSW,ARROW_BOX_MEET_TOLERANCE),
    con_num(Ahead,HNW,HSW,Dnum),
    add_root(arc(Li,[Arrow],Sid,Snum,Sclass,Did,Dnum,Dclass),
     Li,Ni,R1,Lo,No,Ro).
```

A.2.4 File geometry.clpr

This file contains logic procedures for various kinds of geometric constraint tests.

```
#ifndef GEOMETRY
#define GEOMETRY

/* Each of the following predicate definitions is preceded by a comment
   giving the principal functor name and symbolic names for the formal
   parameters.  Parameters beginning with a capital P are points, which
   are 2-element lists [X,Y], where both X and Y are real.  All other
   parameters are real.
*/

/* close(P1,P2,Tolerance)
   - true if P1,P2 differ in neither X nor Y by an amount greater than
     Tolerance
   - e.g. mode (+,+,+) test for closeness
*/
close([X1,Y1],[X2,Y2],Tolerance) :-
    abs(X2-X1) < Tolerance,
    abs(Y2-Y1) < Tolerance.


/* length(P1,P2,L)
   - L is length of line segment from P1 to P2
*/
length([X1,Y1],[X2,Y2],L) :-
    L= pow(pow(X2-X1,2)+pow(Y2-Y1,2),0.5).


/* avg(P1,P2,P3)
   - P3 is spatial average of P1,P2
   - e.g. mode (+,+,-) compute average
*/
avg([X1,Y1],[X2,Y2],[X,Y]) :-
    X = (X1+X2)/2, Y = (Y1+Y2)/2.
```

```
/* inbbox(P1,P2,P3,Tolerance)
   - true if P1 falls with.n the bounding box with diagonally opposite
     corners P2,P3, expanded by Tolerance in all 4 directions.
*/
inbbox([X,Y],[X1,Y1],[X2,Y2],Tolerance) :-
    sort(X1,X2,Xmin,Xmax),
    sort(Y1,Y2,Ymin,Ymax),
    X <= Xmax + Tolerance,
    X >= Xmin - Tolerance,
    Y <= Ymax + Tolerance,
    Y >= Ymin - Tolerance.


/* sort(X1,X2,Xmin,Xmax)
   - Xmin,Xmax are X1,X2 sorted in increasing order
*/
sort(X1,X2,Xmin,Xmax) :-
    X2 < X1,
    Xmin = X2,
    Xmax = X1,
    !.
sort(X1,X2,X1,X2).


/* sort_lr(P1,P2,Pl,Pr)
   - Pl,Pr are P1,P2 sorted in left-to-right order
   - e.g. mode (+,+,-,-) sort two points
*/
sort_lr([X1,Y1],[X2,Y2],[Xl,Yl],[Xr,Yr]) :-
    X2 < X1,
    Xl = X2, Yl = Y2,
    Xr = X1, Yr = Y1,
    !.
sort_lr(P1,P2,P1,P2).


/* sort_tb(P1,P2,Pl,Pr)
   - Pl,Pr are P1,P2 sorted in top-to-bottom order
   - e.g. mode (+,+,-,-) sort two points
*/
sort_tb([X1,Y1],[X2,Y2],[Xt,Yt],[Xb,Yb]) :-
    Y2 < Y1,
    Xt = X2, Yt = Y2,
    Xb = X1, Yb = Y1,
    !.
sort_tb(P1,P2,P1,P2).
```

```
/* horiz(P1,P2,Wmin,Hmax,Pl,Pr)
   - if the bounding box given by opposite corners P1,P2 has
     width >= Wmin and height <= Hmax, Pl,Pr are P1,P2 sorted
     left-to-right
   - e.g. mode (+,+,+,+,-,-) test for horiz. line and sort endpoints
*/
horiz([X1,Y1],[X2,Y2],Wmin,Hmax,[Xl,Yl],[Xr,Yr]) :-
    abs(X2-X1) >= Wmin,
    abs(Y2-Y1) <= Hmax,
    sort_lr([X1,Y1],[X2,Y2],[Xl,Yl],[Xr,Yr]).


/* vert(P1,P2,Hmin,Wmax,Pl,Pr)
   - if the bounding box given by opposite corners P1,P2 has
     width <= Wmax and height >= Hmin, Pl,Pr are P1,P2 sorted
     top-to-bottom
   - e.g. mode (+,+,+,+,-,-) test for vert. line and sort endpoints
*/
vert([X1,Y1],[X2,Y2],Hmin,Wmax,[Xt,Yt],[Xb,Yb]) :-
    abs(X2-X1) <= Wmax,
    abs(Y2-Y1) >= Hmin,
    sort_tb([X1,Y1],[X2,Y2],[Xt,Yt],[Xb,Yb]).


/* pick_vertex(P11,P12,P21,P22,Tolerance,P)
   - if any pair of endpoints of the two line segments P11-P12 and
     P21-P22 are close enough to meet the tolerance, set P to the
     average of the two points.
*/
pick_vertex(P11,P12,P21,P22,Tolerance,P) :-
    close(P11,P21,Tolerance),
    avg(P11,P21,P).
pick_vertex(P11,P12,P21,P22,Tolerance,P) :-
    close(P11,P22,Tolerance),
    avg(P11,P22,P).
pick_vertex(P11,P12,P21,P22,Tolerance,P) :-
    close(P12,P2.,Tolerance),
    avg(P12,P21,P).
pick_vertex(P11,P12,P21,P22,Tolerance,P) :-
    close(P12,P22,Tolerance),
    avg(P12,P22,P).
```

```
/* arrow_match(PS1,PS2,PH,Tolerance,PST,PSH)
   - PS1,PS2 are endpoints of a lineseg which may be the shaft of an
     arrow.  PH is the vertex of an arrowhead.  If PH is close to
     either PS1 or PS2 (within the Tolerance), set PSH to the average
     of PH and the matching PSx point, PST to the other endpoint.
*/
arrow_match(PS1,PS2,PH,Tolerance,PST,PSH) :-
    close(PS1,PH,Tolerance),
    avg(PS1,PH,PSH),
    PST = PS2.
arrow_match(PS1,PS2,PH,Tolerance,PST,PSH) :-
    close(PS2,PH,Tolerance),
    avg(PS2,PH,PSH),
    PST = PS1.


/* con_num(PA,PT,PB,Num)
   - PA is an arrow endpoint, PT,PB are the top and bottom vertices of
     one of a box's vertical sides.  Set Num to
        only, if PA falls in the middle third of the segment PT-PB
        first, if PA is in the upper third
        second, if PA is in the lower third.
   - mode (+,+,+,-) is the only one I have considered
   - the implementation assumes the segment PT-PB really is vertical,
     and only considers Y coordinates.
*/
con_num([_,YA],[_,Ymin],[_,YMax],only) :-
    YA <= (YMax-Ymin)*0.66 + Ymin,
    YA >= (YMax-Ymin)*0.33 + Ymin.
con_num([_,YA],[_,Ymin],[_,YMax],first) :-
    YA < (YMax-Ymin)*0.33 + Ymin,
    YA >= Ymin.
con_num([_,YA],[_,Ymin],[_,YMax],second) :-
    YA > (YMax-Ymin)*0.66 + Ymin,
    YA <= YMax.


#endif
```

A.2.5 File output.clpr

This file contains routines used by the main program to produce the output file **parse_result**, which is in the format required by the IST iconic shell tool application. NOTE: the first **printf** line has been reformatted to fit the page margins for this thesis.

```
#ifndef OUTPUT_CONV
#define OUTPUT_CONV

convert_result(N,R) :-
    printf("%%%%%%%%GSTFILE%%%%%%%%\n",[]),
    convert_nodes(R),
    printf("*---------------------------------
                        ---------------------------------*\n",[]),
    convert_arcs(N,R).

convert_nodes([]).
convert_nodes([H|T]) :-
    do_node(H),
    convert_nodes(T).

do_node(icon(_,_,Class,Num,[X,Y],_,_,_)) :-
    icon_type(Class,Itype),
    printf("% % %.0f %.0f \n",[Itype,Num,X/10,Y/10]),
    print_slots(Class),
    print_expecteds(Class).
do_node(_).

icon_type(source,0).
icon_type(sink,1).
icon_type(filter,2).
icon_type(merger,3).
icon_type(splitter,4).
icon_type(inputfile,5).
icon_type(outputfile,7).

print_slots(source) :- printf("0 0 0 0 1 0 \n",[]).
print_slots(sink) :- printf("0 1 0 0 0 0 \n",[]).
print_slots(filter) :- printf("0 1 0 0 1 0 \n",[]).
print_slots(merger) :- printf("1 0 1 0 1 0 \n",[]).
print_slots(splitter) :- printf("0 1 0 1 0 1 \n",[]).
print_slots(inputfile) :- printf("0 0 0 0 1 0 \n",[]).
print_slots(outputfile) :- printf("0 1 0 0 0 0 \n",[]).
```

```
print_expecteds(source) :- printf("1 ! 1\n",[]).
print_expecteds(sink) :- printf("1 ! 1\n",[]).
print_expecteds(filter) :- printf("1 ! 1\n",[]).
print_expecteds(merger) :- printf("1 1 ! 1\n",[]).
print_expecteds(splitter) :- printf("1 ! 1 1\n",[]).
print_expecteds(inputfile) :- printf("1 ! 1\n",[]).
print_expecteds(outputfile) :- printf("1 ! 1\n",[]).

convert_arcs(N,[H|T]) :-
    do_arc(N,H),
    convert_arcs(N,T).
convert_arcs(_,[]).

do_arc(N,arc(_,_,Sid,Snum,Sclass,Did,Dnum,Dclass)) :-
    conn_num(Snum,Outnum),
    conn_num(Dnum,Innum),
    out_type(Sclass,Outtype),
    in_type(Dclass,Intype),
    out_coords(N,Sid,Snum,X1,Y1),
    in_coords(N,Did,Dnum,X2,Y2),
    printf("% % % % % % %.0f %.0f %.0f %.0f\n",
        [Sid,Outnum,Outtype,Did,Innum,Intype,X1,Y1,X2,Y2]).
do_arc(_,_).

conn_num(first,0).
conn_num(only,1).
conn_num(second,2).

out_type(source,3).
out_type(filter,3).
out_type(merger,3).
out_type(splitter,3).
out_type(inputfile,0).

in_type(sink,7).
in_type(filter,7).
in_type(merger,7).
in_type(splitter,7).
in_type(outputfile,4).

out_coords(N,Boxid,Cnum,X,Y) :-
    member(box(Boxid,_,[NWX,NWY],_,_,_),N),
    locate_output(NWX,NWY,Cnum,X,Y).
```

```
#define INPUT_OFFSET_X 0
#define OUTPUT_OFFSET_X 64
#define ONLY_OFFSET_Y 32
#define FIRST_OFFSET_Y 21
#define SECOND_OFFSET_Y 43
locate_output(NWX,NWY,only,NWX/10+OUTPUT_OFFSET_X,
      NWY/10+ONLY_OFFSET_Y).
locate_output(NWX,NWY,first,NWX/10+OUTPUT_OFFSET_X,
      NWY/10+FIRST_OFFSET_Y).
locate_output(NWX,NWY,second,NWX/10+OUTPUT_OFFSET_X,
      NWY/10+SECOND_OFFSET_Y).

in_coords(N,Boxid,Cnum,X,Y) :-
    member(box(Boxid,_,[NWX,NWY],_,_,_),N),
    locate_input(NWX,NWY,Cnum,X,Y).

locate_input(NWX,NWY,only,NWX/10+INPUT_OFFSET_X,
     NWY/10+ONLY_OFFSET_Y).
locate_input(NWX,NWY,first,NWX/10+INPUT_OFFSET_X,
     NWY/10+FIRST_OFFSET_Y).
locate_input(NWX,NWY,second,NWX/10+INPUT_OFFSET_X,
     NWY/10+SECOND_OFFSET_Y).

#endif
```

A.3 First prototype demonstration

The demonstration of the first prototype parser given in Chapter 5 makes use of graphical representations of the input and output files used in the actual demonstration. The latter are listed here.

A.3.1 Input file seg_db

This file, from which figure 5.4 was produced, was output by the pen-stroke editor. It is a condensed and simplified representation of a picture drawn by hand on the PC screen, produced by reducing the user's pen-strokes to a small number of line segments. The techniques involved, which include cluster analysis, low-pass filtering, and curvature analysis, are described in [41].

```
nodelist([
   lineseg(0,[],[903,846],[930,1130]),
   lineseg(1,[],[898,837],[1243,829]),
   lineseg(2,[],[1241,836],[1247,1111]),
   lineseg(3,[],[1247,1111],[944,1123]),
   lineseg(4,[],[935,1506],[964,1769]),
   lineseg(5,[],[935,1486],[1262,1466]),
   lineseg(6,[],[1262,1466],[1279,1765]),
   lineseg(7,[],[1279,1765],[958,1797]),
   lineseg(8,[],[1041,842],[918,1021]),
   lineseg(9,[],[2005,1133],[2036,1445]),
   lineseg(10,[],[1997,1116],[2358,1124]),
   lineseg(11,[],[2358,1124],[2371,1456]),
   lineseg(12,[],[2371,1456],[2038,1471]),
   lineseg(13,[],[1250,967],[1999,1210]),
   lineseg(14,[],[1957,1170],[1995,1216]),
   lineseg(15,[],[1995,1216],[1943,1223]),
   lineseg(16,[],[1272,1600],[2019,1391]),
   lineseg(17,[],[1955,1377],[2002,1386]),
   lineseg(18,[],[2002,138 ,[1972,1427]),
   lineseg(19,[],[2920,1152],[2933,1464]),
   lineseg(20,[],[2925,1142],[3324,1146]),
   lineseg(21,[],[3324,1146],[3334,1488]),
   lineseg(22,[],[3334,1488],[2928,1477]),
   lineseg(23,[],[2860,1277],[2908,1302]),
   lineseg(24,[],[2908,1302],[2851,1330]),
   lineseg(25,[],[3681,1307],[3743,1332]),
   lineseg(26,[],[3743,1332],[3693,1344]),
   lineseg(27,[],[3738,1176],[3727,1522]),
   lineseg(28,[],[3729,1169],[4094,1190]),
   lineseg(29,[],[4094,1190],[4096,1517]),
   lineseg(30,[],[4096,1517],[3736,1520]),
   lineseg(31,[],[3878,1176],[3727,1381]),
   lineseg(32,[],[2373,1283],[2893,1300]),
   lineseg(33,[],[3350,1316],[3726,1328])
  ], 34).
```

A.3.2 The output file `parse_result`

This is the first output file produced by the main routine, which is an interpretation of the roots of the final parse forest in the format of a saved IST cgraph file.

```
%%%%GSTFILE%%%%
0 35 94 150
0 0 0 0 1 0
1 ! 1
7 38 373 117
0 1 0 0 0 0
1 ! 1
5 34 90 84
0 0 0 0 1 0
1 ! 1
2 37 292 115
0 1 0 0 1 0
1 ! 1
3 36 200 112
1 0 1 0 1 0
1 1 ! 1
*-----------------------------------------------------------------*
37 1 3 38 1 4 356 147 373 149
36 1 3 37 1 7 264 144 292 147
35 1 3 36 2 7 158 182 200 155
34 1 0 36 0 7 154 116 200 133
```

A.3.3 The output file parse_forest

This is the second output file produced by the main program. It is a symbolic listing of the final parse forest. Figure 5.5 was produced by careful analysis of this file. The format is a nested list of lists. Elements of each list are indented to the same level. Each sublist is further indented than the list which contains it. NOTE: Several lines in this file are too long to fit the page margins of this thesis, and have been split so that each continuation line is aligned at the left with the first line.

```
Parse forest (58 nodes):
icon(49, [36], merger, 36. [2001, 1124.5], [2358, 1124], [2037, 1458],
[2371, 1456])
   box(36, [10, 11, 12, 9], [2001, 1124.5], [2358, 1124], [2037, 1458],
   [2371, 1456])
      lineseg(10, [], [1997, 1116], [2358, 1124])
      lineseg(11, [], [2358, 1124], [2371, 1456])
      lineseg(12, [], [2371, 1456], [2038, 1471])
      lineseg(9, [], [2005, 1133], [2036, 1445])
```

```
icon(50, [37], filter, 37, [2922.5, 1147], [3324, 1146], [2930.5,
1470.5], [3334, 1488])
   box(37, [20, 21, 22, 19], [2922.5, 1147], [3324, 1146], [2930.5,
   1470.5], [3334, 1488])
      lineseg(20, [], [2925, 1142], [3324, 1146])
      lineseg(21, [], [3324, 1146], [3334, 1488])
      lineseg(22, [], [3334, 1488], [2928, 1477])
      lineseg(19, [], [2920, 1152], [2933, 1464])
icon(51, [40], inputfile, 34, [900.5, 841.5], [1243, 829], [937,
1126.5], [1247, 1111])
   file_icon(40, [34, 8], [900.5, 841.5], [1243, 829], [937, 1126.5],
   [1247, 1111])
      box(34, [1, 2, 3, 0], [900.5, 841.5], [1243, 829], [937,
      1126.5], [1247, 1111])
         lineseg(1, [], [898, 837], [1243, 829])
         lineseg(2, [], [1241, 836], [1247, 1111])
         lineseg(3, [], [1247, 1111], [944, 1123])
         lineseg(0, [], [903, 846], [930, 1130])
      lineseg(8, [], [1041, 842], [918, 1021])
icon(52, [39], outputfile, 38, [3733.5, 1172.5], [4094, 1190],
[3731.5, 1521], [4096, 1517])
   file_icon(39, [38, 31], [3733.5, 1172.5], [4094, 1190], [3731.5,
   1521], [4096, 1517])
      box(38, [28, 29, 30, 27], [3733.5, 1172.5], [4094, 1190],
      [3731.5, 1521], [4096, 1517])
         lineseg(28, [], [3729, 1169], [4094, 1190])
         lineseg(29, [], [4094, 1190], [4096, 1517])
         lineseg(30, [], [4096, 1517], [3736, 1520])
         lineseg(27, [], [3738, 1176], [3727, 1522])
      lineseg(31, [], [3878, 1176], [3727, 1381])
icon(53, [35], source, 35, [935, 1496], [1262, 1466], [961, 1783],
[1279, 1765])
   box(35, [5, 6, 7, 4], [935, 1496], [1262, 1466], [961, 1783],
   [1279, 1765])
      lineseg(5, [], [935, 1486], [1262, 1466])
      lineseg(6, [], [1262, 1466], [1279, 1765])
      lineseg(7, [], [1279, 1765], [958, 1797])
      lineseg(4, [], [935, 1506], [964, 1769])
arc(54, [48], 34, only, inputfile, 36, first, merger)
   arrow(48, [41, 13], [1250, 967], [1987.5, 1201.5])
      arrowhead(41, [14, 15], [1976, 1193])
         lineseg(14, [], [1957, 1170], [1995, 1216])
         lineseg(15, [], [1995, 1216], [1943, 1223])
      lineseg(13, [], [1250, 967], [1999, 1210])
```

```
arc(55, [47], 35, only, source, 36, second, merger)
   arrow(47, [42, 16], [1272, 1600], [1998.75, 1386.25])
      arrowhead(42, [17, 18], [1978.5, 1381.5])
         lineseg(17, [], [1955, 1377], [2002, 1386])
         lineseg(18, [], [2002, 1386], [1972, 1427])
      lineseg(16, [], [1272, 1600], [2019, 1391])
arc(56, [46], 36, only, merger, 37, only, filter)
   arrow(46, [43, 32], [2373, 1283], [2888.5, 1294.75])
      arrowhead(43, [23, 24], [2884, 1289.5])
         lineseg(23, [], [2860, 1277], [2908, 1302])
         lineseg(24, [], [2908, 1302], [2851, 1330])
      lineseg(32, [], [2373, 1283], [2893, 1300])
arc(57, [45], 37, only, filter, 38, only, outputfile)
   arrow(45, [44, 33], [3350, 1316], [3719, 1323.75])
      arrowhead(44, [25, 26], [3712, 1319.5])
         lineseg(25, [], [3681, 1307], [3743, 1332])
         lineseg(26, [], [3743, 1332], [3693, 1344])
      lineseg(33, [], [3350, 1316], [3726, 1328])
```

A.4 The Prolog acceptor for digraphs

This is a reproduction of the ASG-based acceptor for the digraph language, originally given in Figure 4.11 at the end of Chapter 4. It is presented here to facilitate comparison with the parsers listed later.

```
% productions coded as prod(number,head,body)
prod(5,[n(V1),n(V2)],[n(V1),n(V2),arc(V1,V2)]).
prod(4,[n(V1)],[n(V1),arc(V1,V1)]).
prod(3,[n(V1)],[n(V1),n(_)]).
prod(2,[n(V1)],[node(V1)]).
prod(1,[n0],[n(_)]).

% acceptor
accept([n0]).
accept(W) :-
   prod(_,X,Y),
   match(Y,W,W1),
   union(X,W1,W2),
   accept(W2),
   !.

% match(A,B,C)
% A is a subset of B, and C is its complement w.r.t. B
match([],B,B) :- !.
match([E|A],B,C) :-
   element(E,B,B1),
   match(A,B1,C).

% element(+E,+S,-T)
% succeeds if E is an element of set S; T is S minus E
element(E,[E|R],R).
element(E,[X|S],[X|T]) :-
   element(E,S,T).

% union(+X,+Y,?Z)
% Z is union of sets X and Y
union([],X,X).
union([X|R],Y,Z) :-
   element(X,Y,_), !,
   union(R,Y,Z).
union([X|R],Y,[X|Z]) :-
   union(R,Y,Z).
```

A.5 The second prototype (ASG-based) parsers

A.5.1 The digraph parser

Here is the digraph parser presented as Figure 5.6 in Chapter 5. Note its similarity to the acceptor listed above.

```
% productions coded as prod(number,head,body)
prod(5,[n(V1),n(V2)],[n(V1),n(V2),arc(V1,V2)]).
prod(4,[n(V1)],[n(V1),arc(V1,V1)]).
prod(3,[n(V1)],[n(V1),n(_)]).
prod(2,[n(V1)],[node(V1)]).
prod(1,[n0],[n(_)]).

% parser as seen by user: prints derivation of W if one exists
parse(W) :-
   nl,accept_print(W),writeln(W).

% internal parser: acceptor augmented with printing functions
accept_print([n0]).
accept_print(W) :-
   prod(P,X,Y),
   match(Y,W,W1),
   union(X,W1,W2),
   accept_print(W2),
   writeln(W2),
   printf("p% : % -> %\n",[P,X,Y]),
   !.

% match(A,B,C)
% A is a subset of b, and C is its complement w.r.t. B
match([],B,B) :- !.
match([E|A],B,C) :-
   element(E,B,B1),
   match(A,B1,C).

% element(+E,+S,-T)
% succeeds if E is an element of set S; T is S minus E
element(E,[E|R],R).
element(E,[X|S],[X|T]) :-
   element(E,S,T).
```

```
% union(+X,+Y,?Z)
% Z is union of sets X and Y
union([],X,X).
union([X|R],Y,Z) :-
   element(X,Y,_), !,
   union(R,Y,Z).
union([X|R],Y,[X|Z]) :-
   union(R,Y,Z).
```

A.5.2 Demonstration: digraph parsing

Here is a copy of the transcript of the test run of the digraph parser, originally given as Figure 5.7 in Chapter 5.

```
CLP(R) Version 1.2
(c) Copyright International Business Machines Corporation
1989 (1991, 1992) All Rights Reserved

1 ?- parse([node(1),arc(1,2),node(2),arc(2,2)]).

[n0]
p1 : [n0] -> [n(2)]
[n(2)]
p3 : [n(2)] -> [n(2), n(1)]
[n(2), n(1)]
p4 : [n(2)] -> [n(2), arc(2, 2)]
[n(1), n(2), arc(2, 2)]
p5 : [n(1), n(2)] -> [n(1), n(2), arc(1, 2)]
[n(2), n(1), arc(1, 2), arc(2, 2)]
p2 : [n(2)] -> [node(2)]
[n(1), arc(1, 2), node(2), arc(2, 2)]
p2 : [n(1)] -> [node(1)]
[node(1), arc(1, 2), node(2), arc(2, 2)]

*** Yes
```

A.5.3 Binary tree parser with integrated constraint testing

Here is the complete listing of the parser for the binary tree diagram language, for which a partial listing was given as Figure 5.8 in Chapter 5. This is the only program listed in this Appendix which is a true CLP(R) program, i.e., which makes use of more than the Prolog subset of the CLP(R) language.

Note that the rule **test** was written first, and the result of its execution (which can be seen in the transcript listed below) was used in the formulation of the rule **go**,

which actually performs the demonstration.

```
% constraints
equal(A,B) :- A=B.
sum(A,B,C) :- A=B+C.
mid(A,B,C) :- A=(B+C)/2.

% productions coded as prod(number,params,head,body)
prod(3,[DX,DY],
   [subtree(X,Y,LX,RX)],
   [node(X,Y),subtree(CXP,CY,LX,RXP),subtree(CXDP,CY,LXP,RX),
    arc(X,Y,CXP,CY),arc(X,Y,CXDP,CY)]
   ) :- sum(CY,Y,DY), sum(LXP,RXP,DX), mid(X,CXP,CXDP).
prod(2,[_,DY],
   [subtree(X,Y,LX,RX)],
   [node(X,Y),subtree(X,CY,LX,RX),arc(X,Y,X,CY)]
   ) :- sum(CY,Y,DY).
prod(1,[_,_],
   [subtree(X,Y,LX,RX)],
   [node(X,Y)]
   ) :- equal(X,LX), equal(LX,RX).

% parser as seen by user: prints derivation of W if one exists
parse(PARMS,W) :-
   nl,accept_print(PARMS,W),writeln(W).

% internal parser: acceptor augmented with printing functions
accept_print(_,[subtree(_,_,_,_)]).
accept_print(PARMS,W) :-
   prod(P,PARMS,X,Y),
   match(Y,W,W1),
   union(X,W1,W2),
   accept_print(PARMS,W2),
   writeln(W2),
   printf("p% : % -> %\n",[P,X,Y]),
   !.

% match(A,B,C)
% A is a subset of B, and C is its complement w.r.t. B
match([],B,B) :- !.
match([E|A],B,C) :-
   element(E,B,B1),
   match(A,B1,C).
```

```
% element(+E,+S,-T)
% succeeds if E is an element of set S; T is S minus E
element(E,[E|R],R).
element(E,[X|S],[X|T]) :-
   element(E,S,T).

% union(+X,+Y,?Z)
% Z is union of sets X and Y
union([],X,X).
union([X|R],Y,Z) :-
   element(X,Y,_), !,
   union(R,Y,Z).
union([X|R],Y,[X|Z]) :-
   union(R,Y,Z).

test :- PX=0, PY=0, DX=4, DY=4,
    equal(X1,LX),equal(LX,X2),sum(Y2,Y1,DY),equal(X4,X6),equal(X5,X4),
    equal(X8,RX),equal(X7,X8),sum(Y3,Y1,DY),mid(X3,X5,X7),sum(X8,X6,DX),
    sum(Y1,PY,DY),mid(PX,X1,X3),sum(X4,X2,DX),
    writeln([node(PX,PY),arc(PX,PY,X1,Y1),arc(PX,PY,X3,Y1),
    node(X1,Y1),node(X1,Y2),node(X3,Y1),arc(X1,Y1,X1,Y2),
    arc(X3,Y1,X5,Y3),arc(X3,Y1,X7,Y3),node(X5,Y3),node(X7,Y3)]).

go :- parse([4,4],[
    node(0, 0), arc(0, 0, -3, 4), arc(0, 0, 3, 4), node(-3, 4),
    node(-3, 8), node(3, 4), arc(-3, 4, -3, 8), arc(3, 4, 1, 8),
    arc(3, 4, 5, 8), node(1, 8), node(5, 8)
    ]).
```

A.5.4 Demonstration: binary tree diagram parsing

Here is a copy of the transcript of the test run of the binary tree diagram parser, originally given as Figure 5.10 in Chapter 5. NOTE: Some of the lines in the original transcript were too long to fit the page margins of this thesis, and have been split.

```
1 ?- test.
[node(0, 0), arc(0, 0, -3, 4), arc(0, 0, 3, 4), node(-3, 4),
node(-3, 8), node(3, 4), arc(-3, 4, -3, 8), arc(3, 4, 1, 8),
arc(3, 4, 5, 8), node(1, 8), node(5, 8)]

*** Yes
```

```
2 ?- go.

[subtree(0, 0, -3, 5)]

p3 : [subtree(0, 0, -3, 5)] -> [node(0, 0), subtree(-3, 4, -3, -3),
      subtree(3, 4, 1, 5), arc(0, 0, -3, 4), arc(0, 0, 3, 4)]

[subtree(3, 4, 1, 5), subtree(-3, 4, -3, -3), node(0, 0),
 arc(0, 0, -3, 4), arc(0, 0, 3, 4)]

p3 : [subtree(3, 4, 1, 5)] -> [node(3, 4), subtree(1, 8, 1, 1),
      subtree(5, 8, 5, 5), arc(3, 4, 1, 8), arc(3, 4, 5, 8)]

[subtree(5, 8, 5, 5), subtree(1, 8, 1, 1), subtree(-3, 4, -3, -3),
 node(0, 0), arc(0, 0, -3, 4), arc(0, 0, 3, 4), node(3, 4),
 arc(3, 4, 1, 8), arc(3, 4, 5, 8)]

p1 : [subtree(5, 8, 5, 5)] -> [node(5, 8)]

[subtree(1, 8, 1, 1), subtree(-3, 4, -3, -3), node(0, 0),
 arc(0, 0, -3, 4), arc(0, 0, 3, 4), node(3, 4), arc(3, 4, 1, 8),
 arc(3, 4, 5, 8), node(5, 8)]

p1 : [subtree(1, 8, 1, 1)] -> [node(1, 8)]

[subtree(-3, 4, -3, -3), node(0, 0), arc(0, 0, -3, 4),
 arc(0, 0, 3, 4), node(3, 4), arc(3, 4, 1, 8), arc(3, 4, 5, 8),
 node(1, 8), node(5, 8)]

p2 : [subtree(-3, 4, -3, -3)] -> [node(-3, 4), subtree(-3, 8, -3, -3),
      arc(-3, 4, -3, 8)]

[subtree(-3, 8, -3, -3), node(0, 0), arc(0, 0, -3, 4), arc(0, 0, 3, 4),
 node(-3, 4), node(3, 4), arc(-3, 4, -3, 8), arc(3, 4, 1, 8),
 arc(3, 4, 5, 8), node(1, 8), node(5, 8)]

p1 : [subtree(-3, 8, -3, -3)] -> [node(-3, 8)]

[node(0, 0), arc(0, 0, -3, 4), arc(0, 0, 3, 4), node(-3, 4),
 node(-3, 8), node(3, 4), arc(-3, 4, -3, 8), arc(3, 4, 1, 8),
 arc(3, 4, 5, 8), node(1, 8), node(5, 8)]

*** Yes
```

REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[2] H. Ait-Kaci and R. Nasr. Integrating logic and functional programming. *Lisp and Symbolic Computation*, 2:51–89, 1989.

[3] J. Allen. *Natural Language Understanding.* Benjamin/Cummings Pub. Co., Menlo Park, CA, 1987.

[4] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–516, September 1975.

[5] D. Blostein. Structural analysis of music notation. In *Proceedings of SSPR90: IAPR Workshop on Syntactic and Structural Pattern Recognition, Murray Hill, NJ*, page 481. International Association for Pattern Recognition, 1990.

[6] D. Blostein and H. Baird. A critical survey of music image analysis. In H. Baird, H. Bunke, and K. Yamamoto, editors, *Structured Document Image Analysis*, pages 405–434. Springer-Verlag, 1992.

[7] D. Blostein and L. Haken. Template matching for rhythmic analysis of music keyboard input. In *Proceedings of the Tenth International Conference on Pattern Recognition, Atlantic City, NJ*, pages 767–770, 1990.

[8] S. R. Bourne. *The UNIX System.* Addison-Wesley, Reading, MA, 1983.

[9] C. Boyd. A graphical shell tool. CS490y thesis final report, Computer Science Dept., University of Western Ontario, 1990.

[10] A. Brüggemann-Klein and D. Wood. Drawing trees nicely with TEX. Research Report CS-87-05, University of Waterloo, February 1987.

[11] H. Bunke. Hybrid pattern recognition methods. In H. Bunke and A. Sanfeliu, editors, *Syntactic and Structural Pattern Recognition: Theory and Applications*, pages 307–347. World Scientific Publ. Co., 1990.

[12] H. Bunke and B. Haller. Syntactic analysis of context free plex languages for pattern recognition. In *Proceedings of SSPR90: IAPR Workshop on Syntactic and Structural Pattern Recognition, Murray Hill, NJ*, pages 57–77. International Association for Pattern Recognition, 1990.

[13] R. M. Carr. The point of the pen. *Byte*, pages 211–221, February 1991.

[14] R. Chandhok, D. Garlan, D. Goldenson, P. Miller, and M. Tucker. Programming environments based on structure editing: the GNOME approach. In A. S. Wojcik, editor, *AFIPS Conference Proceedings: 1985 National Computer Conference*, pages 359–369. AFIPS Press, Reston, VA, 1985.

[15] S.-K. Chang. Visual languages: A tutorial and survey. *IEEE Software*, pages 29–39, January 1987.

[16] T. W. Chien and H. Jürgensen. Parameterized L systems for modelling: Potential and limitations. In G. Rozenberg and A. Salomaa, editors, *Lindenmayer Systems: Impacts on Theoretical Computer Science, Computer Graphics, and Developmental Biology*, pages 213–229. Springer-Verlag, 1992.

[17] S. S. Chok and K. Marriott. Parsing visual languages. Technical Report 94/200, Computer Science Dept., Monash University, 1994.

[18] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, IT-2:113–124, 1956.

[19] P. A. Chou. Recognition of equations using a two-dimensional stochastic context-free grammar. In *Visual Communications and Image Processing IV*, pages 852–863. Society of Photo Optical Instrumentation Engineers, 1989.

[20] P. A. Chou. A Cocke-Younger-Kasami parsing algorithm for high-dimensional context-free grammars. In *Proceedings of SSPR90: IAPR Workshop on Syntactic and Structural Pattern Recognition, Murray Hill, NJ*, page 483. International Association for Pattern Recognition, 1990.

[21] A. L. Chow and R. V. Rubin. Topological composition systems: Specifications for lexical elements of visual languages. In *Proceedings of the 1991 IEEE Workshop on Visual Languages*, pages 118–124. IEEE Computer Society Press, Los Alamitos, CA, 1991.

[22] W. V. Citrin. Requirements for graphical front ends for visual languages. In *Proceedings of the 1993 IEEE Workshop on Visual Languages*, pages 142–150. IEEE Computer Society Press, Los Alamitos, CA, 1993.

[23] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, fourth edition, 1994.

[24] S. Collin and D. Colnet. Syntactical analysis of technical drawing dimensions. In *Advances in Syntactic and Structural Pattern Recognition*, pages 280–289. World Scientific Publ. Co., 1992.

[25] G. Costagliola and S.-K. Chang. DR parsers: a generalization of LR parsers. In *Proceedings of the 1990 IEEE Workshop on Visual Languages*, pages 174–180. IEEE Computer Society Press, Los Alamitos, CA, 1990.

[26] G. Costagliola, S. Orefice, G. Polese, G. Tortora, and M. Tucci. Automatic parser generation for pictorial languages. In *Proceedings of the 1993 IEEE Workshop on Visual Languages*, pages 306–313. IEEE Computer Society Press, Los Alamitos, CA, 1993.

[27] G. Costagliola, M. Tomita, and S.-K. Chang. A generalized parser for 2-D languages. In *Proceedings of the 1991 IEEE Workshop on Visual Languages*, pages 98–104. IEEE Computer Society Press, Los Alamitos, CA, 1991.

[28] B. Courcelle. Attribute grammars: Definitions, analysis of dependencies, proof methods. In B. Lorho, editor, *Methods and Tools for Compiler Construction*, pages 81–102. Cambridge University Press, 1984.

[29] B. Courcelle. Graph rewriting: An algebraic and logic approach. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 5. Elsevier Science Publishers B.V., Amsterdam and MIT Press, Cambridge, MA, 1990.

[30] P. T. Cox and T. Pietrzykowski. Using a pictorial representation to combine dataflow and object-orientation in a language independent programming mechanism. In *Proceedings International Computer Science Conference 1988*. IEEE, 1988.

[31] P. J. Denning. Smart editors. *Communications of the ACM*, 24(8):491–493, 1981.

[32] V. Di Gesù and D. Tegolo. The iconic interface for the PIctorial C Language. In *Proceedings of the 1992 IEEE Workshop on Visual Languages*, pages 119–124. IEEE Computer Society Press, Los Alamitos, CA, 1992.

[33] D. Dori. Self structural syntax directed pattern recognition of dimensioning components in engineering drawings. In *Proceedings of SSPR90: IAPR Workshop on Syntactic and Structural Pattern Recognition, Murray Hill, NJ*, pages 88–112. International Association for Pattern Recognition, 1990.

[34] F. Drewes. Parsing ordered graphs generated by hyperedge replacement. Technical Report 10/90, Fachbereich Mathematik und Informatik, Universität Bremen, 1990.

[35] F. Drewes. NP-completeness of k-connected hyperedge-replacement languages of order k. *Information Processing Letters*, 45:89–94, 1993.

[36] F. Drewes. Recognising k-connected hypergraphs in cubic time. *Theoretical Computer Science*, 109:83–122, 1993.

[37] G. D. P. Dueck and G. V. Cormack. Modular attribute grammars. *The Computer Journal*, 33(2):164–172, 1990.

[38] S. D. Dunne. Computer setting of music scores: Survey and problem analysis. Technical Report 174, Computer Science Dept., University of Western Ontario, 1989.

[39] S. D. Dunne. A new paradigm for computer formatting of specialized notations. Technical Report 227, Computer Science Dept., University of Western Ontario, 1989.

[40] S. D. Dunne. Interactive pen-based editors as components of integrated writing and visual language processing systems. Term paper for course CS624b, September 1992.

[41] S. D. Dunne. Towards interactive pen input of visual languages. Technical Report 325, Computer Science Dept., University of Western Ontario, 1992.

[42] S. D. Dunne and H. Jürgensen. Foundations for a general mark-setting system with applications to musical score formatting. Technical Report 171, Computer Science Dept., University of Western Ontario, 1987.

[43] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13:94–102, 1970.

[44] H. Fahmy. A graph-grammar approach to high-level music recognition. Technical Report 91-319, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, September 1991.

[45] H. Fahmy and D. Blostein. A graph grammar programming style for recognition of music notation. *Machine Vision and Applications*, 6(2-3):83–99, 1993.

[46] J. Feder. Plex languages. *Information Science*, 3:225-241, 1971.

[47] F. Ferrucci, G. Pacini, G. Tortora, and G. Vitiello. Efficient parsing of multidimensional structures. In *Proceedings of the 1991 IEEE Workshop on Visual Languages*, pages 105–110. IEEE Computer Society Press, Los Alamitos, CA, 1991.

[48] C. W. Fraser. A generalized text editor. *Communications of the ACM*, 23(3):154–162, 1980.

[49] C. W. Fraser and A. A. Lopez. Editing data structures. *ACM Transactions on Programming Languages and Systems*, 3(2):115-125, 1981.

[50] H. Freeman. On the encoding of arbitrary geometric configurations. *IRE Transactions on Elecironic Computers*, June 1961.

[51] R. Freund. Syntactical analysis of handwritten characters by using quasi-regular programmed array grammars with attribute vectors. In *Advances in Syntactic and Structural Pattern Recognition*, pages 310–319. World Scientific Publ. Co., 1992.

[52] A. M. Frisch and R. B. Scherl. A general framework for modal deduction. In J. A. Allen, R. Fikes, and E. Sandewall, editors, *Principles of Knowledge Representation and Reasoning: Proceedigns of the Second International Conference.* Morgan Kaufman, 1991.

[53] K. S. Fu. *Syntactic Methods in Pattern Recognition*, volume 112 of *Mathematics in Science and Engineering.* Academic Press, New York, 1974.

[54] E. P. Glinert, editor. *Visual Programming Environments.* IEEE Computer Society Press, Los Alamitos, CA, 1990. (in two volumes).

[55] E. P. Glinert and S. Tanimoto. Pict: an interactive graphical programming environment. *IEEE Computer*, 17:7–25, 1984.

[56] E. J. Golin. Interaction diagrams: A visual language for controlling a visual program editor. In *Proceedings of the 1991 IEEE Workshop on Visual Languages*, pages 152–158. IEEE Computer Society Press, Los Alamitos, CA, 1991.

[57] E. J. Golin. Parsing visual languages with picture layout grammars. *Journal of Visual Languages and Computing*, 2:371–393, 1991.

[58] E. J. Golin and T. Magliery. A compiler generator for visual languages. In *Proceedings of the 1993 IEEE Workshop on Visual Languages*, pages 314–321. IEEE Computer Society Press, Los Alamitos, CA, 1993.

[59] E. J. Golin and S. P. Reiss. The specification of visual language syntax. In *Proceedings of the 1989 IEEE Workshop on Visual Languages*, pages 105–110. IEEE Computer Society Press, Los Alamitos, CA, 1989.

[60] E. J. Golin and S. P. Reiss. The specification of visual language syntax. *Journal of Visual Languages and Computing*, 1:141–157, 1990.

[61] H. Göttler. Graph grammars and diagram editing. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors. *Graph Grammars and Their Application to Computer Science, 3rd International Workshop, Warrenton, Virginia, December 1986*, pages 216–231, Berlin, 1987. Springer-Verlag.

[62] Susan H. and T. Teitelbaum. Generating editing environments based on relations and attributes. *ACM Transactions on Programming Languages and Systems*, 8(4):577–608, October 1986.

[63] A. Habel. *Hyperedge Replacement: Grammars and Languages*, volume 643 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[64] A. Habel and H.-J. Kreowski. May we introduce to you: Hyperedge replacement. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Graph-Grammars and their Application to Computer Science: 3rd International Workshop*, pages 15–26. Springer-Verlag, 1986.

[65] G. Hamlin. Software for device-independent graphical input. In *Graphics Interface '82*, pages 23–27. Association for Computing Machinery, 1982.

[66] J. S. Hanan. *Plantworks: A Software System for Realistic Plant Modelling*. PhD thesis, University of Regina, Canada, 1988.

[67] N. C. Heintze, J. Jaffar, C. Lassez, J.-L. Lassez, K. McAloon, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. Constraint logic programming: A reader. Notes for Tutorial Session 3, Fourth IEEE Symposium on Logic Programming, San Francisco, August 31 – September 4, 1987.

[68] N. C. Heintze, J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. *The CLP(R) Programmer's Manual*. IBM Thomas J. Watson Research Center, Yorktown Heights, NY, version 1.2 edition, September 1992.

[69] R. Helm and K. Marriott. Declarative specification of visual languages. In *Proceedings of the 1990 IEEE Workshop on Visual Languages*, pages 98 103. IEEE Computer Society Press, Los Alamitos, CA, 1990.

[70] R. Helm, K. Marriott, and M. Odersky. Building visual language parsers. In *Procedings of CHI 1991 (New Orleans, Louisiana)*. Association for Computing Machinery, 1991.

[71] R. Helm, K. Marriott, and M. Odersky. Spatial query optimization: From Boolean constraints to range queries. Research Report RC 17231 (#76325) 9/30/91, IBM Research Division, T. J. Watson Research Center, Yorktown Heights, N.Y. 10598, 1991.

[72] T. C. Henderson and A. Samal. Table driven parsing for shape analysis. *Pattern Recognition*, 19(4):279–288, 1986.

[73] T. R. Henry, S. E. Hudson, and G. L. Newell. Integrating gesture and snapping into a user interface toolkit. In *UIST Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology, Snowbird, Utah*, pages 112–122. Association for Computing Machinery, 1990.

[74] G. E. Hinton, C. K. Williams, and M. D. Revow. Adaptive elastic models for hand-printed character recognition. In J. E. Moody, S. J. Hanson, and R. P. Lippmann, editors, *Advances in Neural Information Processing Systems 4*. Morgan Kauffmann, San Mateo, CA, 1992.

[75] C. J. Hogger. *Introduction to Logic Programming*. Academic Press, 1984.

[76] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pages 111–119. Association for Computing Machinery, 1987.

[77] T. Kakuma and E. Tanaka. A pessimistic view of syntactic pattern recognition for graphics recognition. In *Proceedings of SSPR90: IAPR Workshop on Syntactic and Structural Pattern Recognition, Murray Hill, NJ*, pages 165–181. International Association for Pattern Recognition, 1990.

[78] R. Kasturi and L. O'Gorman. Document image analysis: A bibliography. *Machine Vision and Applications*, 5:231–243, 1992.

[79] R. Kasturi, R. Raman, C. Chennubhotla, and L. O'Gorman. Document image analysis: An overview of techniques for graphics recognition. In *Proceedings of SSPR90: IAPR Workshop on Syntactic and Structural Pattern Recognition, Murray Hill, NJ*, pages 192–229. International Association for Pattern Recognition, 1990.

[80] H. Kato and S. Inokuchi. The recognition system for printed piano music using musical knowledge and constraints. In *Proceedings of SSPR90: IAPR Workshop on Syntactic and Structural Pattern Recognition, Murray Hill, NJ*, pages 231–248. International Association for Pattern Recognition, 1990.

[81] T. D. Kimura. Hyperflow: A visual programming language for pen computers. In *Proceedings of the 1992 IEEE Workshop on Visual Languages*, pages 125–132. IEEE Computer Society Press, Los Alamitos, CA, 1992.

[82] D. E. Knuth. *The TEXbook*. Addison-Wesley, 1986.

[83] K. Kojima and B. A. Myers. Parsing graphic function sequences. In *Proceedings of the 1991 IEEE Workshop on Visual Languages*, pages 111 117. IEEE Computer Society Press, Los Alamitos, CA, 1991.

[84] D. Kurlander and S. Feiner. Interactive constraint-based search and replace. In *Procedings of CHI 1992*. Association for Computing Machinery, 1992.

[85] G. Kurtenbach and W. Buxton. GEdit: A test bed for editing by contiguous gestures. *SIGCHI Bulletin*, 23(2):22-26, 1991.

[86] J. J. Lee. A model for textually-oriented program editing. Technical Report 151, Computer Science Dept., University of Western Ontario, 1986.

[87] S.-W. Lee, J. H. Kim, and F. C. A. Groen. Translation-, rotation-, and scale-invariant recognition of hand-drawn electrical circuit symbols with attributed graph matching. In *Proceedings of SSPR90: IAPR Workshop on Syntactic and Structural Pattern Recognition, Murray Hill, NJ*, pages 273 292. International Association for Pattern Recognition, 1990.

[88] A. Lindenmayer. Mathematical models for cellular interaction in development, parts I and II. *Journal of Theoretical Biology*, 18:290-315, 1968.

[89] T. Luo. TreeDraw: A tree-drawing system. Master's thesis, University of Waterloo, Ontario, Canada, 1993. (Available as University of Western Ontario Dept. of Computer Science technical report no. 396).

[90] E. Mandler. Advanced preprocessing technique for on-line recognition of hand-printed symbols. In R. Plamondon, C. Y. Suen, and M. L. Simner, editors, *Computer Recognition and Human Production of Handwriting*, pages 19 36. World Scientific Publ. Co., 1989.

[91] J. Marks. A syntax and semantics for network diagrams. In *Proceedings of the 1990 IEEE Workshop on Visual Languages*, pages 104 110. IEEE Computer Society Press, Los Alamitos, CA, 1990.

[92] K. Marriott. Constraint multiset grammars. In *Proceedings of the 1994 IEEE Workshop on Visual Languages*. IEEE Computer Society Press, Los Alamitos, CA, 1994.

[93] T. Matsushima, I. Sunomoto, T. Harada, K. Kanamori, and S. Ohteru. Automated high speed recognition of printed music ('VABOT-2 vision system). In *'85 ICAR: International Conference on Advanced Robotics, September 9-10, 1985, Tokyo, Japan*, pages 477–482, 1985.

[94] R. Medina-Mora and P. H. Feiler. An incremental programming environment. *IEEE Transactions on Software Engineering*, SE-7(5):472–482, September 1981.

[95] B. Meyer. Pictures depicting pictures: On the specification of visual languages by visual grammars. In *Proceedings of the 1992 IEEE Workshop on Visual Languages*, pages 41–47. IEEE Computer Society Press, Los Alamitos, CA, 1992.

[96] B. R. Modayer, V. Ramesh, R. M. Haralick, and L. G. Shapiro. MUSER: A prototype musical score recognition system using mathematical morphology. *Machine Vision and Applications*, 6(2-3):140–150, 1993.

[97] F. Mokhtarian and A. Mackworth. Scale-based description and recognition of planar curves and two-dimensional shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(1), January 1986.

[98] B. A. Myers. Visual programming, programming by example and program visualization: A taxonomy. In *Conference Proceedings, CHI'86: Human Factors in Computing Systems*, pages 59–66, New York, 1986. ACM Press.

[99] B. A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1:97–123, 1990.

[100] M. A. Najork and S. M. Kaplan. Specifying visual languages with conditional set rewrite systems. In *Proceedings of the 1993 IEEE Workshop on Visual Languages*, pages 12–18. IEEE Computer Society Press, Los Alamitos, CA, 1993.

[101] H. Nishida and S. Mori. A structural description of curves by quasi-topological features and singular points. In *Proceedings of SSPR90: IAPR Workshop on Syntactic and Structural Pattern Recognition, Murray Hill, NJ*, pages 310–334. International Association for Pattern Recognition, 1990.

[102] M. Nivat, A. Saoudi, and V. R. Dare. Parallel generation of finite images. *International Journal of Pattern Recognition and Artificial Intelligence*, 3(3/4):279–294, 1989.

[103] D. Notkin. *Interactice Structure-Oriented Computing*. PhD thesis, Carnegie-Mellon University, 1984.

[104] O'Reilly and Associates, Inc. *The X Window System Series*. O'Reilly and Associates, Inc., Sebastopol, CA, 1990. (in seven volumes).

[105] J. O'Rourke and R. Washington. Curve similarity via signatures. In G. T. Toussaint, editor, *Computational Geometry*, pages 295–317. Elsevier Science Publishers B.V. (North-Holland), 1985.

[106] Joseph J. P. Jr. Using graph grammars for data structure manipulation. In *Proceedings of the 1990 IEEE Workshop on Visual Languages*, pages 42–47. IEEE Computer Society Press, Los Alamitos, CA, 1990.

[107] T. Pavlidis and C. J. van Wyck. An automatic beautifier for drawings and illustrations. *Computer Graphics*, 19(3):225–234, July 1985.

[108] F. C. N. Pereira and D. H. D. Warren. Definite clause grammars for language analysis: A survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13:231–278, 1980.

[109] J. L. Pfaltz and A. Rozenfeld. Web grammars. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence, Washington, DC*, pages 609–616, May 1969.

[110] J. J. Pfeiffer Jr. Parsing graphs representing two dimensional figures. In *Proceedings of the 1992 IEEE Workshop on Visual Languages*, pages 200–206. IEEE Computer Society Press, Los Alamitos, CA, 1992.

[111] P. Prusinkiewicz and J. Hanan. *Lindenmayer Systems, Fractals, and Plants*, volume 29 of *Lecture Notes in Biomathematics*. Springer-Verlag, New York, 1989.

[112] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, New York, 1990.

[113] S. Pulman. Unification and the new grammatism. In Yorick Wilks, editor, *Theoretical Issues in Natural Language Processing*, chapter 2.2, pages 32–35. Lawrence Erlbaum Associates, Hillsdale, NJ, 1989.

[114] D. R. Raymond. Characterizing visual languages. In *Proceedings of the 1991 IEEE Workshop on Visual Languages (Kobe, Japan)*. IEEE Computer Society Press, Los Alamitos, CA, 1991.

[115] G. Reader. *Programming in Picture*. PhD thesis, University of Southern California, 1984.

[116] T. W. Reps and T. Teitelbaum. *The Synthesizer Generator: a system for constructing language-based editors*. Springer-Verlag, New York, 1989.

[117] A. Rosenfeld. Isotonic grammars, parallel grammars, and picture grammars. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence 6*. University of Edinburgh, 1971.

[118] A. Rosenfeld. Array grammar normal forms. *Information and Control*, 23:173–182, 1973.

[119] A. Rosenfeld. Array grammars. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Graph Grammars and Their Application to Computer Science, 3rd International Workshop, Warrenton, Virginia, December 1986*, pages 67–70, Berlin, 1987. Springer-Verlag.

[120] G. Rozenberg. An introduction to the NLC way of rewriting graphs. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Graph-Grammars and their Application to Computer Science: 3rd International Workshop*, pages 55–66. Springer-Verlag, 1986.

[121] A. Salomaa. *Formal Languages*. Academic Press, New York, 1973.

[122] I. Salter. A framework for formally defining the syntax of visual languages. In *Proceedings of the 1993 IEEE Workshop on Visual Languages*, pages 244–248. IEEE Computer Society Press, Los Alamitos, CA, 1993.

[123] R. J. Schalkoff. *Pattern Recognition: Statistical, Structural, and Neural Approaches*. John Wiley and Sons, Inc., New York, 1992.

[124] R. W. Scheifler. *X Window System Protocol, Version 11.* MIT Press, 1986.

[125] R. W. Scheifler. *X Window System Protocol Encoding, Version 11.* MIT Press, 1987.

[126] A. Schürr. PROGRES, a visual language and environment for PROgramming with Graph REwriting Systems. Aachener Informatik-Berichte AIB 94-11, RWTH Aachen (Germany), 1994.

[127] Andy Schürr. Specification of graph translators with triple graph grammars. Aachener Informatik-Berichte AIB 94-12, RWTH Aachen (Germany), 1994.

[128] J. A. Scofield. *Editing as a Paradigm for User Interaction.* PhD thesis, University of Washington, 1985.

[129] D. B. Searls and S. A. Leibowitz. Logic grammars as a vehicle for syntactic pattern recognition. In *Proceedings of SSPR90: IAPR Workshop on Syntactic and Structural Pattern Recognition, Murray Hill, NJ*, pages 402-422. nternational Association for Pattern Recognition, 1990.

[130] E. Selfridge-Field. Optical recognition of music notation: A survey of current work. In W. Hewlett and E. Selfridge-Field, editors, *Computing in Musicology 9.* Center for Computer-Assisted Research in the Humanities, Menlo Park, California, 1993-4.

[131] A. C. Shaw. A formal picture description scheme as a basis for picture processing systems. *Information and Control*, 14:9-51, 1969.

[132] S. M. Shieber. *An Introduction to Unification-Based Approaches to Grammar.* CSLI Lecture Note Series no. 4. Center for the Study of Language and Information, Stanford, CA, 1986.

[133] N. C. Shu. Visual programming languages: A perspective and a dimensional analysis. In S.-K. Chang, T. Ichikawa, and P. A. Ligomenides, editors, *Visual Languages*, pages 11–34. Plenum Press, New York, 1986.

[134] R. Siromoney, A. Huq, M. Chandrasakaran, and K. G. Subramanian. Pattern classification with equal matrix grammars. In *Proceedings of SSPR90: IAPR Workshop on Syntactic and Structural Pattern Recognition, Murray Hill, NJ*, pages 440–449. International Association for Pattern Recognition, 1990.

[135] A. R. Smith. Plants, fractals, and formal languages. *Computer Graphics*, 18(3):1–10, 1984.

[136] W. W. Stallings. Recognition of printed chinese characters by automatic pattern analysis. *Computer Graphics and Image Processing*, 1:47–65, 1972.

[137] R. Steensma. Enhancements to an iconic shell tool for UNIX. CS490y thesis final report, Computer Science Dept., University of Western Ontario, 1992.

[138] I. E. Sutherland. Sketchpad: A man-machine graphical communication system. In *AFIPS Conference Proceedings, Spring Joint Computer Conference*, 1963. Reprinted in [54], pp 198–215.

[139] C. C. Tappert, C. Y. Suen, and T. Wakahara. The state of the art in on-line handwriting recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(8):787–808, 1990.

[140] T. Teitelbaum and T. Reps. The Cornell Program Synthesizer: A syntax directed programming environment. *Communications of the ACM*, 24(9):563–573, 1981.

[141] J. T. Tou. An approach to understanding geometrical configurations by computer. *International Journal of Computer and Information Sciences*, 9(1), February 1980.

[142] J. van der Vegt. Editing objects of a hierarchical structured drawing. Technical Report CS-R9074, Centrum voor Wiskunde en Informatica, Amsterdam, December 1990.

[143] M. H. H. van Dijk and J. W. C. Koorn. GSE, a generic syntax-directed editor. Technical Report CS-R9045, Centrum voor Wiskunde en Informatica, Amsterdam, September 1990.

[144] J. M. Vlissides and M. A. Linton. Unidraw: A framework for building domain-specific graphical editors. In *UIST Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology, Williamsburg, VA*, pages 159–167. Association for Computing Machinery, November 1989.

[145] Y. P. Wang and T. Pavlidis. Optimal correspondence of string subsequences. In *Proceedings of SSPR90: IAPR Workshop on Syntactic and Structural Pattern Recognition, Murray Hill, NJ*, pages 460–479. International Association for Pattern Recognition, 1990.

[146] T. Watanabe, Q. Luo, and N. Sugie. Structure recognition methods for various types of documents. *Machine Vision and Applications*, 6(2-3):163–176, 1993.

[147] R. C. Waters. Program editors should not abandon text oriented commands. *ACM SIGPLAN Notices*, 17(7):39–46, July 1982.

[148] G. Weber. FINGER: A language for gesture recognition. In D. Diaper et. al., editor, *Human-Computer Interaction—INTERACT '90.* Elsevier Science Publishers B.V. (North-Holland), 1990.

[149] L. Weitzman and K. Wittenburg. Relational grammars for interactive design. In *Proceedings of the 1993 IEEE Workshop on Visual Languages*, pages 4–11. IEEE Computer Society Press, Los Alamitos, CA, 1993.

[150] K. Wittenburg. F-PATR: Functional constraints for unification grammars. Unpublished manuscript, Bellcore.

[151] K. Wittenburg. Earley-style parsing for relational grammars. In *Proceedings of the 1992 IEEE Workshop on Visual Languages*, pages 192–199. IEEE Computer Society Press, Los Alamitos, CA, 1992.

[152] K. Wittenburg. The relational language system. Technical Memorandum TM-ARH-022353, Bellcore, Morristown, NJ, December 1992.

[153] K. Wittenburg and L. Weitzman. Visual grammars and incremental parsing for interface languages. In *Proceedings of the 1990 IEEE Workshop on Visual Languages*, pages 111–118. IEEE Computer Society Press, Los Alamitos, CA, 1990.

[154] K. Wittenburg, L. Weitzman, and J. Talley. Unification-based grammars and tabular parsing for graphical languages. *Journal of Visual Languages and Computing*, 2(4):347–370, 1991.

[155] B. Yu and S.-K. Chang. A fuzzy visual language compiler. In *Proceedings of the 1990 IEEE Workshop on Visual Languages*, pages 162–167. IEEE Computer Society Press, Los Alamitos, CA, 1990.