Applying Compiler Techniques to Diagram Recognition

D. Blostein, J. Cordy, R. Zanibbi
Department of Computing and Information Science
Queen's University, Kingston, Ontario, Canada
{ blostein, cordy, zanibbi }@cs.queensu.ca

Abstract

Compiler techniques are effective and efficient in processing textual programming languages. These techniques can be adapted to recognition and processing of two-dimensional languages (diagrams). Already, grammars and parsers have been used in a variety of diagram-recognition and diagram-processing tasks. Here we explore the use of two other compiler techniques in pattern recognition systems. The first is compiler-style use of trees and tree transformation. The second is a multi-pass control structure, with a clear separation between layout, lexical, syntactic, and semantic analysis. Our proposal is illustrated on a case study involving recognition of hand-drawn mathematics notation.

1. Introduction

The technology for compiler writing is mature and highly successful [1]. The adaptation of compiler techniques has had strong influence on syntactic pattern recognition [13] and visual language research [25]. Techniques which were imported include the use of grammars (array grammars [26], tree grammars [2] [13], set grammars [3] [15], graph grammars [29]), and parsing technologies (for example, CYK and Early algorithms for context free grammars [13], and linear-time LR and LL parsing algorithms for more restricted languages [11]). Attributes and attribute-computation rules are used with all types of grammars [13]. For example, Anderson uses numeric attributes for bounding box coordinates, and string attributes to record the meaning of the interpreted math notation [3].

There are further possibilities for exploiting compiler technology in diagram recognition. We begin by briefly reviewing the software organizations used in diagram recognition and compiler writing. These differ markedly, in part due to differences in the problem domain, and in part due to tradition and historical accident. Two major differences in the problem domain are that compilers process one-dimensional input whereas diagram recognition systems process two-dimensional input, and that compilers have noise-free input, whereas diagram recognition systems must handle noise and variability.

In this paper, we make the following recommendations regarding the structuring of a diagram recognition system. We illustrate this through a case study on recognition of mathematics notation. The extent to which these ideas can be applied to other pattern recognition problems remains to be determined.

- Find linear structures in the input. Use these as a basis for finding secondary linear structures. In our case study, the linear structures are baselines in mathematics notation. Intelligent search functions and symbol-specific definitions of image sub-regions are used to locate symbols within a baseline. Early extraction of linear structures makes subsequent processing efficient and easy to reason about.
- Organize the linear structures into a tree. This arises naturally when linear structures are recursively extracted from the input. The tree forms the basis for subsequent, compiler-style processing. In our case study, a single tree (of baselines) suffices. In more complex domains, such as music notation, multiple trees could be used to capture various ways of organizing the input.
- Divide processing into passes for layout, for lexical analysis within and across linear structures, for syntax, and for semantics. This provides for robust processing of input – early passes process all input, even if there are syntax errors or unknown constructs. Also, dialects of the diagram notation can be supported more easily: early passes are the same for all dialects.
- Use a simple, fixed control structure. In our case study, a sequence of passes suffices.
- Use attributed trees and tree transformation technology. Trees provide a natural and convenient method for representing grammar-based processes. For this reason, compiler writers have developed a range of highly efficient techniques for manipulating trees. Tree transformation systems express these manipulations in a concise and easy to understand high-level form.

2. Software architectures

A variety of software organizations have been used in diagram recognition systems, as is briefly reviewed here. Efforts are underway to provide reusable code for recognition systems [28] [32] [34].

The blackboard architecture is a general and flexible framework for combining diverse knowledge sources; applications include recognition of engineering drawings [35], mail pieces [36], and text [31], as well as construction of a drawing-interpretation kernel [28]. Knowledge sources communicate via a blackboard data structure. The blackboard represents multiple, conflicting recognition hypotheses, divided into levels of abstraction (e.g., raw image, thresholded image, labeled image, text-line, and block [36]). The blackboard contents trigger invocation of knowledge sources, thus allowing evidence to be accumulated from diverse knowledge sources in an adaptive manner.

Schema-based systems use schema classes to define prototypical drawing constructs, using class and instance hierarchies for specialization and composition. Schemata have been used with constraint satisfaction for interpretation of sketch maps [27], and with a control grammar for interpretation of engineering drawings [19].

A grammar defines a language via a *start symbol* and a set of *productions* (*rewrite rules*). A parser determines whether a given input is a member of this language [13]. In contrast, a *transformational* grammar does not define a language; instead productions and a control structure are used to rewrite the input to produce a desired output [6] [13]. In most pattern-recognition applications involving grammars, symbol recognition is performed separately, with the grammar used to process the resulting symbols (e.g. [3]). In contrast, Chou's stochastic grammar [8] describes the image down to the pixel level.

Some diagram recognition systems contain explicit models of document generation. Kopec and Chou use a Hidden Markov Model of the document generator in order to find the maximum likelihood interpretation of the document image [22] [23]. A descriptive (rather than generative) model of the domain is used in the recognition framework proposed in [33].

3. Compiler technology

Techniques used in compiler construction include the following [1]. Firstly, language analysis is separated into passes (Figure 1). The primary division is into lexical, syntax, and semantic passes. Pattern recognition systems generally do not use this division. Instead, layout, lexical, and syntactic processing are closely integrated, in order to use higher level constraints to help disambiguate symbol recognition (e.g. [22]).

A second compiler construction technique is the intensive use of grammars, both for parsing and for syntax-directed translation. Virtually all processes in a compiler are driven by grammatically-based techniques – lexical analysis is specified and implemented using regular grammars [24], creation of the parse tree is driven by a context-free grammar [18], semantic analysis is achieved using attribute grammars [21], and code generation is often specified using a transformational grammar [14].

A third compiler-construction technique is tree transformation. Highly efficient tree manipulation is provided by ordered attribute grammars [20], by the TXL system [9] and the Gentle system [30]. Tree manipulations provide a convenient and efficient way to implement a wide range of language-based tasks such as translation, design recovery from source code, and automatic programming [10].

4. Case Study: Math Recognition

As a case study, we compare two approaches to mathematics recognition; others are surveyed in [4] [7]. The input to both systems consists of a set of symbols, annotated by bounding box coordinates.

Handwritten mathematics notation poses many challenges. Symbol recognition must cope with a large character set, a range of font sizes, and small symbols such as commas and accents, which are easily confused with noise. Spatial relations are difficult to define precisely; e.g., the gradual transition from multiplication to exponentiation in $2x$ 2^x 2^x 2^x 2^x. This problem is exacerbated by inexact symbol placement and irregular symbol sizes, which are common in handwritten notation (Figure 2). The meaning of spatial relations depends on context; consider, for example, the pattern x_i in contexts $x_i y_j$ and $a^x i$. Mathematics notation is compact, with little redundancy that might aid recognition.

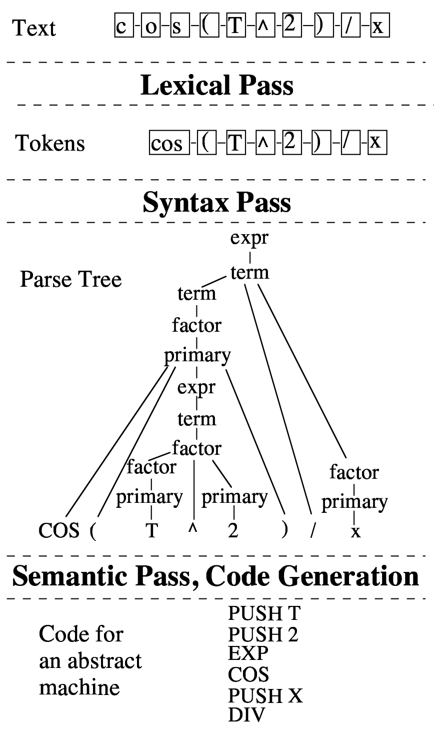

Figure 1. Compiler passes, for the input "cos(T^2)/x".

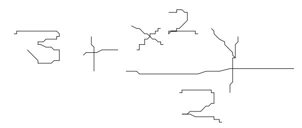

Figure 2. Example of inexact symbol placement: the "3+" is placed higher than it would be in typeset notation.

The first system in our case study uses graph transformation rules organized into four phases [16]. The initial graph contains one node per symbol, attributed with bounding box coordinates. The *Build* phase adds edges to represent Above, Below, Left, Sub, and Super relations. The *Constrain* phase removes contradictory edges, and determines the role of dots and horizontal lines. The *Rank* phase assigns operator precedences, and the *Incorporate* phase replaces subexpressions by single nodes. In a second implementation, using the PROGRES language, *Build*, *Constrain*, and *Parse* phases are used [5]. Positive aspects of this system include the following. On the positive side, (1) these small prototype systems were the first recognition systems capable of handling irregular symbol placement in handwritten math notation, and (2) we found that graph transformation was a natural and convenient style of computation. On the negative side, scaling up of the system is difficult. It is true that a division into phases helps structure the recognition (see also [12]). Nevertheless, it is difficult to extend the system to recognize an additional math-notation construct: it is hard to find the right places where code must be added. In retrospect, one source of difficulty may be that the phases are not divided along layout, syntax and semantics lines. Also, the generality of the graph structure may be a source of difficulty. Graph edges easily and naturally represent any number of relations, but this generality can make it difficult to reason about the state of the graph. If the input can be represented as a tree, as is done below, it is easier to reason about the state of the computation.

The second system in our case study, DRACULAE [37] [38], adapts compiler techniques to mathematics recognition. The input to a compiler is a string; this is tokenized by the Lexical Pass, and converted to a tree by the Syntax Pass (Figure 1). The input to diagram recognition is two-dimensional. Processing options include:

- Translate the diagram into a string, and then apply compiler methods directly. This approach is impractical because it is unwieldy to represent and manipulate spatial relations encoded in a string.
- Translate the diagram into a graph, and then apply graph transformation rules, as discussed above. This is a very general approach. Tree-based compiler technology is not easily applicable.
- Translate the diagram into a tree, and then apply compiler-style tree transformation. Translation to a tree proceeds by repeatedly finding linear sub-structures in the diagram, as discussed next.

The passes used in DRACULAE are illustrated in Figure 3. Most of the system design effort was directed at the Layout Pass, which converts the set of input symbols into a tree. The remaining passes were relatively easy to construct due to the ease of adapting compiler tools and technology. As described in [38], the Layout Pass uses search functions: Start() locates the first symbol in a baseline, and Hor() finds the next symbol in a baseline. Image subregions are defined around baseline symbols, and the search functions are recursively applied in these subregions. The initial inspiration for this approach came from Positional Grammars [11]; significant extensions were needed to handle the inexact symbol placement common in handwritten math expressions.

We now compare the compiler passes (Figure 1) to the passes DRACULAE uses for math notation (Figure 3). DRACULAE's Layout Pass converts the set of symbols to a tree. A compiler has no counterpart to the Layout Pass. The output of the Layout Pass is a Baseline Structure Tree, which captures the spatial structure of the input. This tree is produced for any input, even if it contains syntax errors (e.g., mismatched parentheses) or novel math constructs. DRACULAE operates without backtracking. The linear sub-structures captured in the Baseline Structure Tree are exploited to achieve this efficiency.

A compiler's Lexical Pass operates on linear input and produces linear output. DRACULAE's two Lexical Passes operate on tree input and produce tree output. The Lexical Pass for Tokens performs lexing within one baseline (combine “c" “o” “s” into “cos”). The Lexical Pass for Relations performs lexing between baselines. It handles explicit operators (e.g. division bars) and implicit operators (e.g. exponentiation). If the input expression conforms to correct math syntax, all spatial relations (ABOVE, SUPER, etc.) are replaced by operation labels.

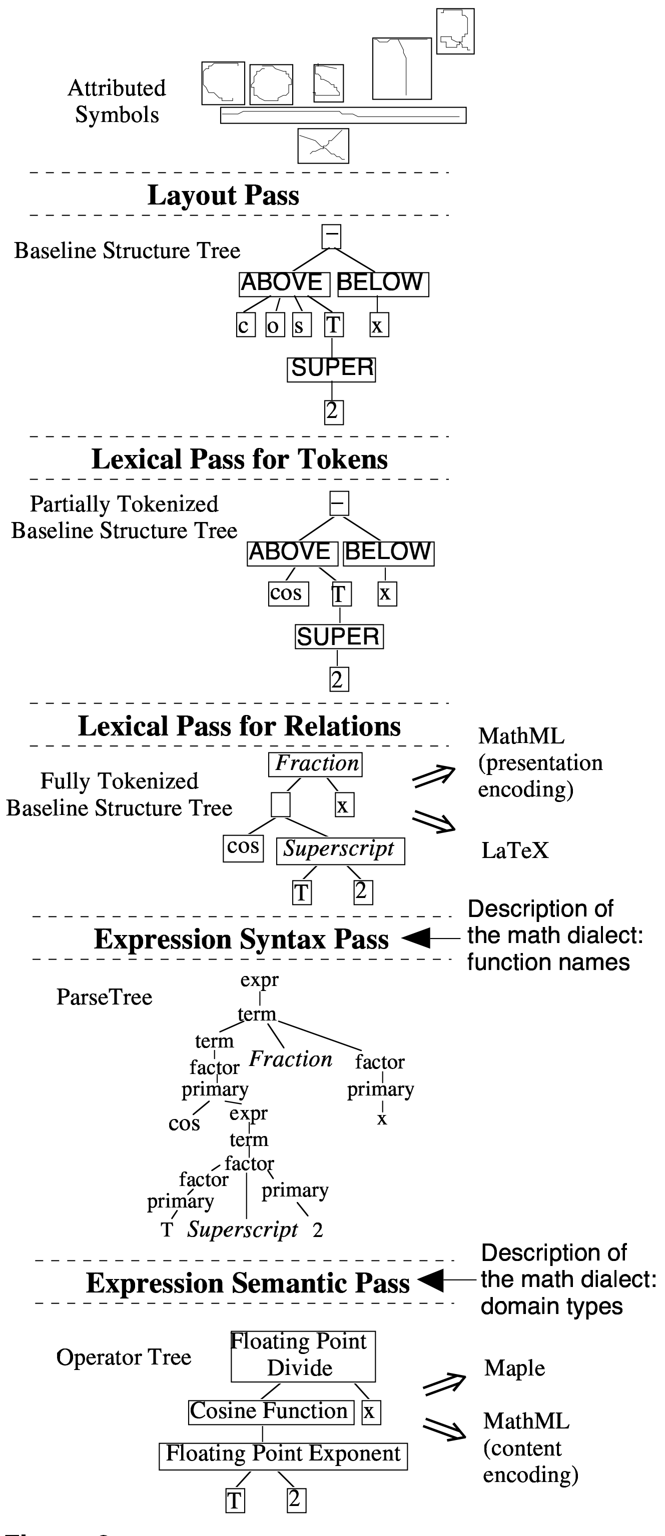

Figure 3. Overview of DRACULAE, a math recognition system which adapts compiler techniques [37] [38].

A compiler's Syntax Pass (with linear input and tree output), corresponds to DRACULAE's Expression Syntax Pass (with tree input and tree output). The Expression Syntax Pass replaces operation labels by explicit math operators. This requires external information which is not present in the expression itself. For example, function names are needed, to decide whether "$a(b)$" means "apply function a to operand b" or "multiply a by b". Similarly, type information is needed by the Expression Semantic Pass, to determine if the math operations refer to real numbers, vectors, or matrices. A compiler obtains corresponding information from declarations located in other parts of the source code.

5. Conclusion

The application of compiler technology to diagram recognition has been illustrated on a case study: recognition of math notation. We believe that these ideas can be adapted to recognizers for other notations, particularly those which have a reading order, such as music notation. The input symbols are converted to a tree, by repeatedly finding linear substructures. Layout information is processed completely, before moving on to lexical and syntactic processing. This results in an efficient, extensible system.

Acknowledgement

This research is supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] A. Aho, R. Sethi, J. Ullman, *Compilers: Principles, Techniques and Tools*, Addison Wesley, 1986.

[2] O. Akindele, A. Belaïd, "Construction of Generic Models of Document Structure using Inference of Tree Grammars," *Proc. ICDAR '95*, 206–209.

[3] R. Anderson, "Two Dimensional Mathematical Notation," in *Syntactic Pattern Recognition, Applications*, Ed. K. S. Fu, Springer 1977, 147–177.

[4] D. Blostein, A. Grbavec, "Recognition of Mathematical Notation," *Handbook of Character Recognition and DIA*, Eds. Bunke, Wang, World Scientific, 1997, 557–582.

[5] D. Blostein, A. Schürr, "Computing with Graphs and Graph Transformation," *Software – Practice and Experience*, **29**(3), 1999, 197-217.

[6] H. Bunke, "Attributed Programmed Graph Grammars and Their Application to Schematic Diagram Interpretation," *IEEE PAMI* **4**(6), Nov. 1982, 574–582.

[7] K. Chan, D. Yeung, "Mathematics Expression Recognition: a Survey," *Intl. J. Document Analysis and Recognition*, **3**(1), Aug. 2000, 3-15.

[8] P. Chou, "Recognition of Equations Using a Two-Dimensional Stochastic Context-Free Grammar," *SPIE Proceedings Series* Vol. 1199, 1989, 852–863.

[9] J. Cordy, C. Halpern, E. Promislow, "TXL: A Rapid Prototyping System For Programming Language Dialects," *Computer Languages*, **16**(1), 1991, 97-107.

[10] J. Cordy, T. Dean, A. Malton, K. Schneider, "Software Engineering by Source Transformation - Experience with TXL," *Proc. SCAM'01*, Nov. 2001, 168-178.

[11] G. Costagliola et al., "Positional Grammars: A Formalism for LR-Like Parsing of Visual Languages," in *Visual Language Theory*, Springer, 1998, 171-191.

[12] H. Fahmy, D. Blostein, "A Graph Grammar Programming Style for Recognition of Music Notation," *Machine Vision and Applications*, **6**(2), 1993, 83–99.

[13] K. S. Fu, *Syntactic Pattern Recognition and Applications*, Prentice Hall 1982.

[14] M. Ganapathi, C. Fischer, "Affix Grammar Driven Code Generation," *ACM Trans. Programming Languages and Systems,* **7**(4), 1985, 560-599.

[15] E. Golin, S. Reiss, "The Specification of Visual Language Syntax," *J. Visual Langs.&Comput.*, **1**(2), 1990, 141–157.

[16] A. Grbavec, D. Blostein, "Mathematics Rec. Using Graph Rewriting," *Proc. ICDAR '95*, 417–421.

[17] O. Hitz, L. Robadey, R. Ingold, "Analysis of Synthetic Document Images," *Proc. ICDAR '99*, 374-377.

[18] S. Johnson, "YACC - Yet Another CompilerCompiler," Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ 07974, 1975.

[19] S. Joseph, T. Pridmore, "Knowledge-Directed Interpretation of Mechanical Engineering Drawings," *IEEE PAMI*, **14**(9), Sept. 1992, 928–940.

[20] U. Kastens, "Ordered Attribute Grammars," *Acta Informatica*, **13**, 1980, 229-256.

[21] U. Kastens, B. Hutt, E. Zimmermann, *GAG: A Practical Compiler Generator*, LNCS **141**, Springer, Berlin, 1982.

[22] G. Kopec, P. Chou, "Doc. Image Decoding Using Markov Source Models," *IEEE PAMI*, **16**(6), 1994, 602–617.

[23] G. Kopec, P. Chou, D. Maltz, "Markov Source Model for Printed Music Decoding," *J. Electronic Imaging*, **5**(1), Jan. 1996, 7–14.

[24] M. Lesk, E. Schmidt, "Lex-A Lexical Analyzer Generator," Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, NJ 07974, 1975.

[25] K. Marriott, B. Meyer, K. Wittenburg, "A Survey of Visual Language Specification and Recognition," in *Visual Langu-age Theory*, Springer, 1998, 5-85.

[26] A. Mercer, A. Rosenfeld, "An Array Grammar Programming System," *CACM*, **16**(5), 1973, 299-305.

[27] I. Mulder, A. Mackworth, W. Havens, "Knowledge Structuring and Constraint Satisfaction: The Mapsee Approach," *IEEE PAMI*, **10**(6), Nov. 1988, 866–879.

[28] B. Pasternak, "Processing Imprecise and Structural Distorted Line Drawings by an Adaptable Drawing Interpretation Kernel," *Proc. DAS '94*, 349–363.

[29] *Proc. Intl. Workshops Theory and Application of Graph Transformations* (was *Graph Grammars and Their Applic. to CS*). LNCS Vols. 73, 153, 291, 532, 1073, 1764, Springer.

[30] F. Schroer, *The GENTLE Compiler Construction System*, Oldenbourg, 1997.

[31] R. Sennhauser, "Integration of Contextual Knowledge Sources Into a Blackboard-based Text Recognition System," *Proc. DAS '94*, 211–228.

[32] A. Smeulders, T. ten Kate, "Software System Design for Paper Map Conversion," LNCS **1072**, Springer, 1996, 204–211.

[33] M. Stückelberg, D. Doermann, "Model-Based Graphics Recognition," LNCS **1941**, Springer, 2000, 121-132.

[34] K. Tombre, C. Ah-Soon, P. Dosch, A. Habed, G. Masini, "Stable, Robust and Off-the-Shelf Methods for Graphics Recognition," *Proc. 14th ICPR*, 1998, **1**, 406-408.

[35] P. Vaxivière, K. Tombre, "Knowledge Organization and Interpretation Process in Engineering Drawing Interpretation," *Proc. DAS '94*, 313–321.

[36] C. Wang, S. Srihari, "A Framework for Object Recognition in a Visually Complex Environment...," *Intl. J. Computer Vision*, **2**, 1989, 125–151.

[37] R. Zanibbi, D. Blostein, J. Cordy, "Baseline Structure Analysis of Handwritten Mathematics Notation," *Proc. ICDAR 2001*, 768-773.

[38] R. Zanibbi, D. Blostein, J. Cordy, "Recognizing Handwritten Mathematical Expressions Using Tree Transformation," *IEEE PAMI*, to appear.