Chapter 4

Graph Transformation Applied to Document Image Analysis

Graph transformation has been applied to a variety of problems in document image analysis, including recognition of circuit diagrams, music notation, mathematical notation, and dimension sets in engineering drawings. This chapter provides an overview of these applications, and describes a PROGRES program for recognition of mathematical notation[2].

4.1 Overview of Document Image Analysis

The information from paper documents needs to be available in electronic form, to permit editing, database creation, and other applications. Figure 4.1 illustrates operations on a document image and its information content. Text regions in a document image can be recognized by Optical Character Recognition (OCR). Specialized recognizers are needed for figures and mathematical expressions. The construction of mathematics recognizers has been investigated for several decades [BlGr97], but such recognizers are not yet available on the market.

[2]This chapter includes excerpts from references [GrBl95], [BlSc98] and [Blos98].

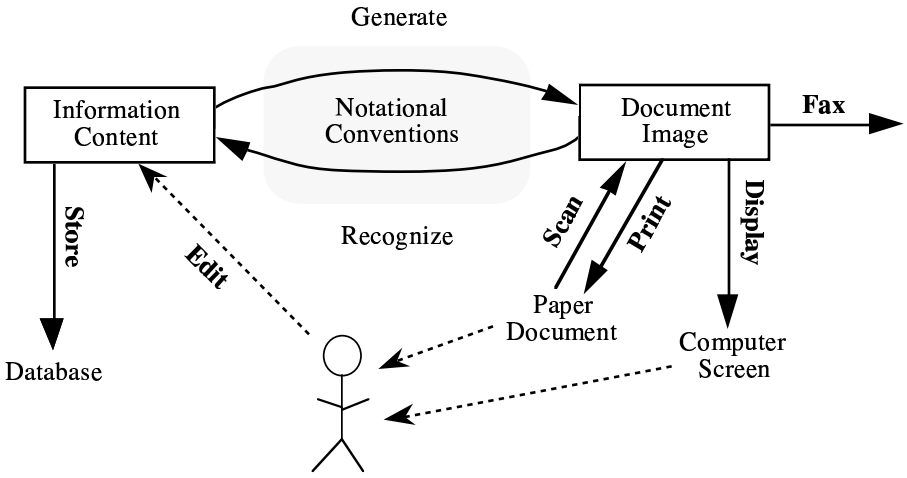


Figure 4.1 A paper document containing mathematical notation is scanned to form a document image. The document image supports operations such as faxing and monitor display. When the document image is processed by a recognizer, mathematical expressions are put into a format such as LaTeX or Mathematica. This allows further operations such as symbolic algebra manipulations, speech synthesis (in a reading machine for visually impaired users), or editing and generation of a new document image.

Pattern-recognition problems such as this are typically solved using three steps [Fu82]:

- Define a set of primitives for the domain, as well as a set of relations among the primitives.
- Implement algorithms to find the primitives and relations in a given input.
- Interpret this set of primitives and relations relative to some model of the domain.

The domain model can be organized in many ways, including graph productions, a string grammar, a semantic net, a blackboard system, or procedurally coded rules. Syntactic and structural approaches are reviewed in [Sanf92], including string grammars, tree grammars, transition network grammars, prototype pattern representation, and accompanying methods of analysis (matching, parsing, heuristic search, decision trees, constraint propagation). Few criteria are available for choosing the best approach to modeling a particular domain [Blos96]. The graph transformation approach is attractive because graphs are a general and flexible mechanism for expressing spatial and logical relations among arbitrary sets of pattern primitives.

4.2 Systems Using Graph Rewriting for Document Image Analysis

The following is a chronological discussion of systems that apply graph transformation to document image analysis. These are research systems, most of which work with a restricted subset of the notation in question. Constructing a commercial system is very difficult because of the image noise and variability that must be accommodated.

4.2.1 Parsing Images of Neural Networks [Pfal72]

This paper introduces the idea of using graph grammars to describe pictures. Input images contain blobs (neurons) with attached tree-like structures (dendrites). A graph grammar is used to parse the input, determining how to separate overlapping dendrites. The first stage of parsing groups endpoints into incomplete lower dendrites, then into y junctions, and finally into N-junctions (junctions between a dendrite and the cell body). The second stage of parsing interprets X junctions as being either crossing axons or touching axons. The system was tested on synthetic images.

4.2.2 Analysis of Circuit Diagrams and Flowcharts [Bun82]

This paper makes a strong case for the use of graph productions to transform an input graph into an output graph. Whereas parsing algorithms for graph grammars can be very costly, the application of ordered graph productions is much more efficient and involves no backtracking. The control specification uses sequential, conditional, and looping constructs to describe the order of production application. Compound symbols in circuit diagrams and flowcharts are recognized using this approach.

A circuit diagram image is preprocessed to extract the set of lines (and circles) that occur in the image. This preprocessing is simulated in the experiments. The set of lines is converted to an initial graph, with one node per line endpoint or line intersection. These nodes are connected by graph edges wherever corresponding edge segments were extracted from the image. Node attributes record the (x, y) image location being represented. The image locations allow graph productions to test constraints such as near-parallelism of two edges. The output graph contains nodes representing compound drawing entities (capacitors, transistors, resistors).

The ordering of productions is useful in defining the recognition algorithm. First, error-correcting productions remove certain irregularities from the input graph. This includes closing gaps in lines and shortening lines that extend past their ideal endpoints (as when a person draws a T intersection with the vertical line extending too far upwards). Once these error-corrections have been exhaustively applied, the recognition of compound symbols begins. Here again, ordering is important. The rectangles that represent resistors are recognized first. Once these have been recognized, remaining right-angle bends are certain to be bends in signal lines. About 60 productions are used to process circuit diagrams, and 30 productions to process flowcharts.

4.2.3 Analysis of Dimension Sets in Engineering Drawings [DoPn88]

In this work, a web grammar is used to describe the syntax of dimension sets in engineering drawings. An engineering drawing describes the structure of a physical object, often using several orthogonal views. Dimension sets state the distance or angle between two places on the object, and can be drawn in many ways:

The graphic components of a dimension set include text, shape, tail, contour, arrowhead, and witness line. These components form the primitives of the web grammar. The grammar is small and non-recursive. About 10 web productions are used to generate (or parse) virtually all of the dimension-sets which can occur in machine drawings. The grammar is being adapted for use in the Machine Drawing Understanding System (MDUS). Engineering drawings are difficult to process, due to their large size and high noise levels (smudges, stains from coffee cups, breaks in lines, text in small fonts). MDUS algorithms for primitive recognition are discussed in [DLDC93].

4.2.4 Analysis of Music Notation [FaBl93]

This work extends the approach proposed by Bunke [Bun82] to recognition of music notation. The following issues had to be addressed. Firstly, music notation is not graph-based, unlike the circuit-diagram and flowchart notations processed by Bunke. Therefore it is much less clear how to construct the initial graph from the set of image primitives. Secondly, music notation contains physically distant symbols which have semantically important relationships. For example, the pitch of a note can be affected by accidentals (sharp ♯ or flat ♭) placed just before a note, placed earlier in the measure, or placed in the key signature at the start of the staff. In contrast, analysis of circuit diagrams and flowcharts involves only recognition of local line configurations.

Recognition phases named *Build*, *Weed*, and *Incorporate* are used in solving this problem. The system begins with a set of music symbols and their image coordinates. It is assumed that there are no errors in symbol recognition. Each symbol gives rise to one attributed node in the initial graph. There are no edges in the initial graph, since there is no way to predict which symbol pairs have important spatial relationships. *Build* productions inspect pairs of nodes and add edges for potentially interesting associations. *Weed* productions remove uninteresting or conflicting associations. The remaining associations are interpreted by *Incorporate* productions, which use the attributes of long-lived nodes to record the image interpretation. For example, *Build* productions create edges from a `note` node to several preceding accidentals (possibly including one just before the note, several earlier in the measure, several belonging to key signatures earlier on the staff). A *Weed* production removes the edges to all but the closest preceding accidental. An *Incorporate* production updates the `pitch` attributes of the `note` node, and deletes the edge to the `accidental` node. After the `accidental` node is completely disconnected, it is removed as well.

The *Build*, *Weed*, *Incorporate* sequence is used within each of the following processing stages: connect noteheads and accidentals to the bar line ending their measure, find the clef governing each notehead and accidental, associate noteheads and stems, associate accidentals and noteheads, use duration dots to update note and rest durations, form notes into chords, use flags and beams to update the duration of chords. Approximately 50 productions are used in the prototype system, which processes only a small subset of music-notation constructs.

4.2.5 Analysis of Music Notation Using Simplified Graph Productions [Pies94] [Baum95]

This work defines a simplified type of graph production, to allow efficient implementation. Graph productions are restricted to having at most three nodes on either LHS (left hand side) or RHS (right hand side). Edges are treated as node attributes, not as separate objects. The applicability predicates are restricted in form. Seven types of productions are defined, to increase the variety of operations that can be expressed using this restricted form of graph production. The production types, which are closely tailored to the application, are CD_START (start a control diagram), LOCALIZE (apply the production to a subpart of the graph, defined relative to a tree structure imposed on the graph), DELOCALIZE (undo the effect of a previous LOCALIZE), REPLACE (replace the instance of LHS by the first node in RHS), GENERATE (keep the LHS instance and create the first node in RHS subject to various syntax conditions), ASSIGN (do not create new nodes, but possibly delete nodes), and REASSIGN (adjust the tree structure that is imposed on the graph).

A complete system was implemented for a subset of music notation, and tested on three pages of music (*Dickie's Rag* by Heger and 2 pages of Bach's *Wohltemperiertes Klavier*). An image is first processed to remove staff lines. Next each connected component (a connected black region left in the image after staff-line removal) is processed by a nearest-neighbor classifier, producing up to three hypotheses for which symbol is represented. This provides the input to the graph transformation. Approximately 110 productions (not the complete set) are shown in [Pies94].

4.2.6 Analysis of Mathematical Notation [GrBl95]

This work uses graph productions to interpret mathematical expressions which have irregular layout (Figure 4.2). Details of the graph transformation approach are described in Section 4.3. Briefly, input to the graph transformation system is a correct set of mathematical symbols, annotated with (x, y) image locations. Initial graph construction is similar to [FaBl93]: a discrete graph with one node per symbol. In the first implementation [GrBl95], recognition is divided into phases *Build*, *Constrain*, *Rank*, and *Incorporate*. During a subsequent translation to PROGRES, undertaken by A. Schürr, the software was reorganized into *Build*, *Constrain*, and *Parse* phases. A copy of the code is available at [URL98]. *Build* productions create edges between any nodes that may share a significant spatial relationship. *Constrain* productions examine a larger context in the graph, to remove unneeded spatial relationships. This is the trickiest part of the computation. Proper identification of the important spatial relationships permits the subsequent *Parse* phase to operate without backtracking. *Parse* productions must ensure that operators are not applied prematurely. For example, in a hand-written expression with layout $\frac{a}{b} + c$ the + operation should not be evaluated first, even though a + c are in approximate horizontal alignment. Were the division bar to extend past the c, then the + should be evaluated first. *Parse* productions use alignment tests to check for these conditions. Approximately 60 productions are used in [GrBl95] and 45 productions in the PROGRES prototype. These prototypes process only a small subset of mathematical notation.

4.2.7 Analysis of Mathematical Notation [LaPo97]

This work uses a context sensitive graph grammar to implement mathematics recognition. The parser operates bottom-up, without backtracking. The absence of backtracking is achieved by carefully adding context conditions to the graph productions, to cover every case where two different productions might apply to the same node. It is assumed that accurate point-size information is available, thus permitting accurate, local identification of subscripts and superscripts. This allows simpler parsing strategies than were used in [GrBl95]. For example, point size information disambiguates the configuration x_i which might occur in the context $x_i y_i$ or $a^x i$. The size of this system (number of productions) is not available; the implementation of the graph builder and graph grammar required approximately 7000 lines of KLONE code.

4.2.8 Analysis of Music Notation with Symbol Uncertainty [FaBl98]

This work addresses the problem of interpreting music notation when the symbol recognizer produces a list of possible interpretations for each symbol. Graph transformation is used to formulate a generalized version of discrete relaxation, in which the constraints are not known *a priori*. This permits additional constraints to be formulated as recognition proceeds. Graph productions interleave the application of constraints with the interpretation of groups of music symbols. Approximately 180 graph productions are used, organized into *Build*, *Constrain*, *Split*, and *Incorporate* phases.

4.3 PROGRES Program for Interpreting Images of Mathematical Notation

This section describes a PROGRES program which addresses some of the problems involved in the interpretation of mathematical notation. Many approaches have been taken in the construction of recognizers for mathematical notation. Years ago Anderson proposed an elegant coordinate grammar approach to the problem [Ande77]. This inspiring work assumes restrictive rules for symbol positioning, for example, that the limits of a summation lie within the x-extent of the 'Σ' symbol. Other approaches include projection profile cutting, stochastic context free grammars, procedurally-coded rules, and statistical labeling for subscript and superscript identification [BlGr97].

It may not be apparent that mathematics recognition is a difficult problem. Certainly, linear mathematical expressions, as found in programming languages, are easily processed using well-understood string grammars and parsers. Images of two-dimensional mathematical expressions are much more difficult to process. First, the mathematical symbols are difficult to segment and recognize; the large character set and large range of point sizes places this beyond the capability of standard OCR systems. Second, parsing or other processing methods must be able to handle incorrectly recognized or missing symbols. Third, even if the mathematical symbols are perfectly recognized, they are difficult to interpret due to the subtle use of space in mathematical notation. All three of these are major difficulties; we focus only on the third point. We assume as input a correct set of mathematical symbols, annotated with (x, y) image locations. The following characteristics of mathematical notation make these symbols difficult to interpret.

The layout of symbols is not fully standardized, and is particularly irregular for handwritten expressions (Figure 4.2). This makes it difficult to identify the logical meaning of spatial relationships. In the following sequence, it is not clear whether the middle elements represent multiplication or exponentiation: $2x \quad 2x \quad 2^x \quad 2^x \quad 2^x \quad 2^x$. A related problem is that the baseline cannot be found without global examination of the expression. For example, the meaning of x_i is not clear locally; it is different in $x_i y_i$ compared to $a^x i$. Point-size information can help, but point sizes are difficult to recognize, and are highly variable in handwritten expressions.

Figure 4.2 Two handwritten mathematical expressions illustrating the irregular layout that may be used.

Mathematical symbols can take on various roles. A dot can be a decimal point, a multiplication symbol, an annotation such as $\dot{x}$, or noise. A horizontal line can be a division symbol, part of a compound symbol such as '=', a negation symbol, or a subtraction symbol. The role of a line is particularly difficult to determine for handwritten expressions, since writers use variable line length and placement.

4.3.1 Build, Constrain, Parse

The graph transformations which process a mathematical expression are organized into phases. In the original version, Build, Constrain, Rank, and Incorporate phases were used [GrBl95]. During a subsequent translation to PROGRES, the software was reorganized into Build, Constrain, and Parse phases, which are described below. The

complete PROGRES source code is available [URL98]. This source code can be executed by the PROGRES interpreter, or it can be automatically translated to C. A graphical user interface allows the intermediate stages of the graph transformation process to be inspected. The input to the program is a list of mathematical symbols and their (x, y) locations. The output of the program is a string describing the expression, in a format similar to that used in programming languages.

The processing steps are illustrated in Figure 4.3. First, a handprinted or typeset mathematical expression is scanned to turn it into an image. The image is processed by a symbol recognizer. In our system, symbol recognition is performed by hand, and is assumed to be done correctly. The resulting symbols are converted into an initial graph which contains one node per symbol, and no edges. Each node has attributes which store the identity of the character as well as the image location of the character. For example, a "y" in the image gives rise to a node with label `DescendingLetter`, a `meaning` attribute "y", and attributes `xmin`, `xmax`, `ymin`, `ymax` providing the coordinates of the bounding box. The productions in the Build phase create edges between any nodes that may share a significant spatial relationship. The Constrain phase examines a larger context in this graph, to remove unneeded spatial relationships. This is the trickiest part of the computation. Proper identification of the important spatial relationships greatly simplifies the job of the subsequent Parse phase.

•••Paste Math Recognition Figure Here•••

Figure 4.3 A document is scanned to produce image a). Parts b) to f) show snapshots of the processing of this data, as seen in a window of the PROGRES interpreter. Nodes in the initial graph b) have attributes `xmin`, `xmax`, `ymin`, `ymax` which are not shown. The Build phase adds edges for all spatial relationships that might be useful, producing a complete graph for this small example. The constrain phase examines the graph context, to remove unneeded edges such as `c sub d`. The Parse phase collapses subgraphs that represent high-precedence operations. The final graph f) contains a single node whose `meaning` attribute is a string describing the original expression.

The central data structure in this computation is the graph that represents the mathematical expression. Its type declaration, called a *graph schema*, is shown in Figure 4.4. The graph schema defines all possible node labels, and their associated attributes. The node labels are organized into a class hierarchy, with multiple inheritance. This allows graph productions to match nodes with greater or lesser selectivity. For example, suppose the expression graph contains a node that represents a plus symbol. A production rule can match this plus node using an LHS node labeled `Plus`, `PLUSMINUS` , `TERMINAL_OPERATOR` , `SYMBOL` , `TERMINAL` , `OPERATOR` , or `OBJECT` . The `OBJECT` label is most general, matching any node in the expression graph. The `Plus` label is most specific, matching only nodes that represent plus symbols.

Node classes are defined for both terminals and nonterminals. Any node belonging to class TERMINAL represents a mathematical symbol and has position attributes `xmin`, `xmax`, `ymin`, `ymax`. These nodes are created as part of the initial graph. Nodes belonging to class NONTERMINAL are created during the recognition process, to represent the interpretation of the mathematical expression. They do not have position attributes.

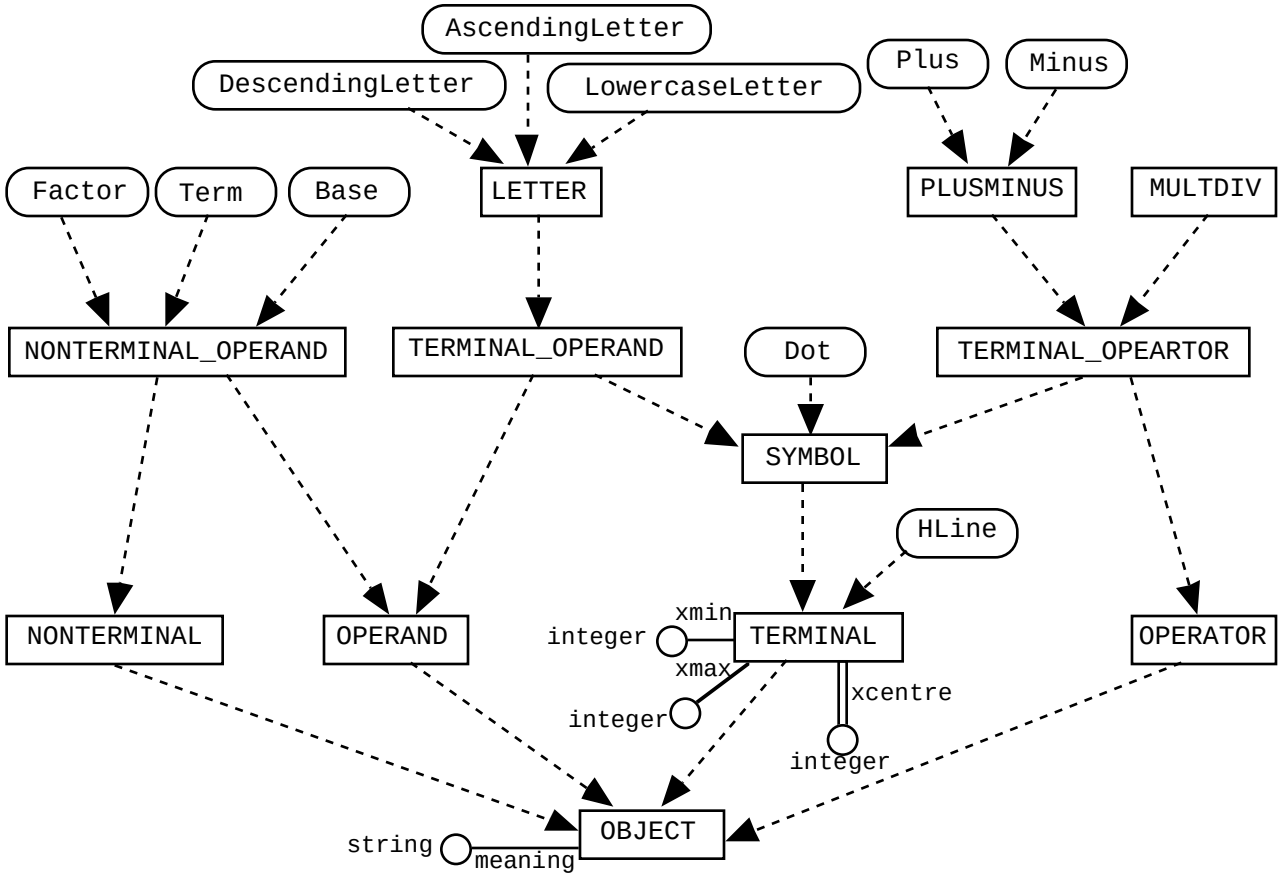


Figure 4.4 An excerpt of the graph schema for the expression graph. The graph schema can be viewed and edited graphically or textually. The class hierarchy has node class `OBJECT` at the root, and node types at the leaves. Attribute definitions are shown for node classes `OBJECT` and `TERMINAL` . An `OBJECT` has an attribute `meaning` of type `string`. This attribute is inherited by all node classes and node types. A `TERMINAL` has two integer attributes, `xmin` and `xmax`, and a derived attribute `xcentre` which is defined as `avg( self.xmin, self.xmax)`. The attributes `ymin`, `ymax`, and `ycentre` are defined analogously.

Three global parameters are used in defining criteria for spatial relationships. This is illustrated in Figure 4.5(b). The maximum horizontal separation between two horizontally-adjacent symbols, called `hmax`, is used in

function `precedes`. The maximum vertical separation between two vertically-adjacent symbols, called `vmax`, is used in function `super`. The maximum vertical deviation between two horizontally-adjacent syntactic units, called `htol`, is used in function `left`. If these parameters are set to small values, then mathematical expressions are not interpretable unless the symbols are close to the ideal alignment. This might be appropriate for typeset expressions, but would lead to rejection of many handwritten expressions. If these parameters are set to large values, then expressions with quite irregular layout can be interpreted. Large parameter values increase the execution time: the `Build` phase adds many more edges to the expression graph, because it accepts more distant symbols as potentially being in horizontal, vertical, subscript or superscript relations. These extra edges, in turn, create more work for the `Constrain` phase. The `Parse` phase is unaffected.

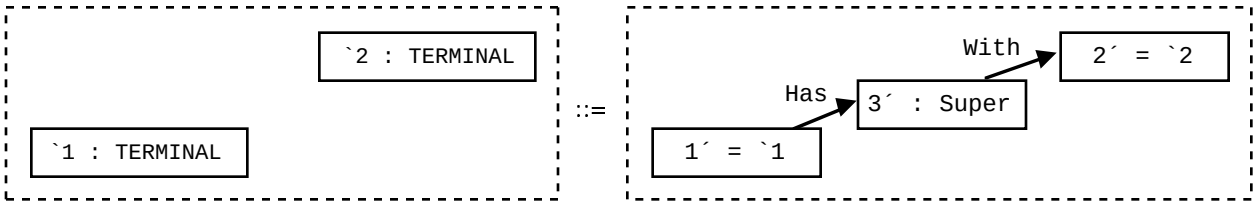
Build productions have a simple structure, shown in Figure 4.5(a). The LHS tests whether two `TERMINAL` node meet the criteria for a particular spatial relationship. If so, then this relationship is inserted into the graph.

Several different types of Constrain productions are used. The first type, shown in Figure 4.6(a), removes a particular kind of unneeded spatial relationship. The second type, shown in Figure 4.6(b), recognizes the syntactic role of a symbol. The first type of production is applied exhaustively before moving on to the latter type of production. This allows the syntactic-role identification to proceed with the knowledge that unneeded spatial relationships have already been removed.

Parse productions find and interpret syntactically-meaningful configurations of symbols. No backtracking is necessary – the Parse productions are written so that any execution path leads to the correct answer. In case of a syntactically incorrect expression, all execution paths lead to a failed parse, but the nature of the failure can vary depending on which path is taken. Multiple execution paths arise in two ways: (1) the control structure specifies that several graph productions are to be applied nondeterministically, and (2) a graph production is applicable to several locations in the host graph.

Parse productions avoid premature processing of operators. For example, in a hand-written expression with symbol placement $\frac{a}{b} + c$, the + operation should not be processed first, even though a + c are in approximate horizontal alignment. Were the division bar to extend past the c, then the + should be processed first. Parse rules use alignment tests to check for these conditions, as in Figure 4.6.

```
(* Build a Super association between any two symbols in a base-superscript configuration.  *)
production Build6 =
```

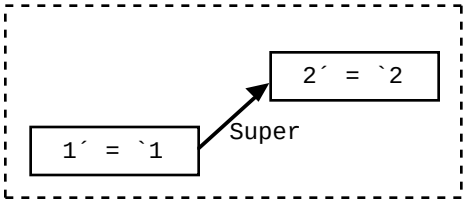



```
  condition super (`1, `2 );
end;
```

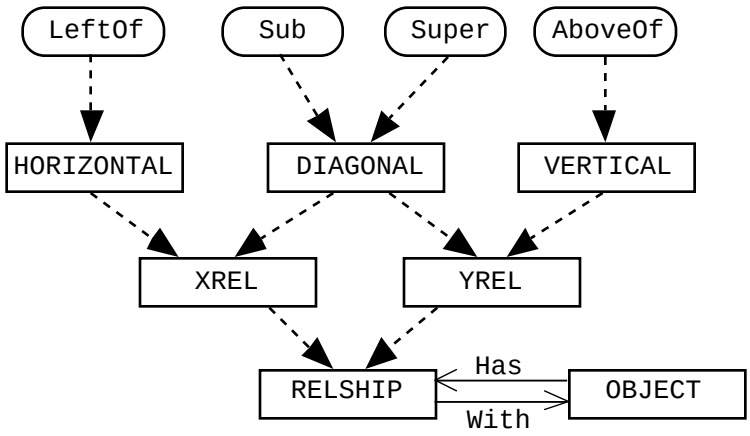
(a)

```
function between :  ( a, b, c : integer) -> boolean = (a < b) and (b < c)  end;
function hBounded : ( a, b : TERMINAL) -> boolean = between ( b.xmin, a.xcentre, b.xmax ) end;
function precedes : ( a, b : TERMINAL) -> boolean =
                     between ( 0, b.xcentre - a.xmax, hmax )  and  (a.xmin < b.xmin)   end;
function left :     ( a, b : TERMINAL) -> boolean =
                     precedes (a, b)   and   (absInt (a.ycentre - b.ycentre ) < htol)  end;
function super :    ( a, b : TERMINAL) -> boolean =
                     (not hBounded (a, b))   and   precedes (a, b)
                      and   between (htol, a.ycentre - b.ycentre, vmax)               end;
```

(b)



(c)

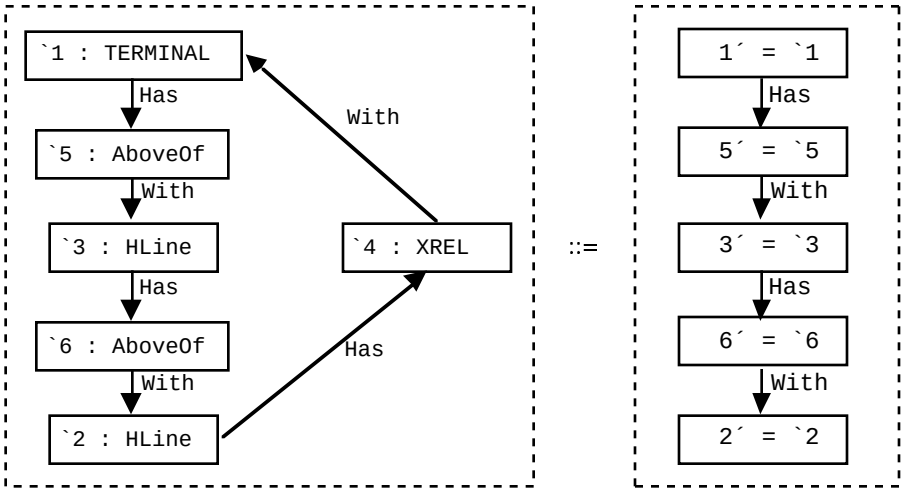


(d)

Figure 4.5 The Build production (a) tests whether two TERMINAL nodes have a spatial relationship satisfying the super condition. This condition is defined using the functions in (b). If the condition is met, we would like to insert a Super edge between the two TERMINAL nodes (c). Instead, the RHS of production (a) inserts a Super node. This is done because PROGRES allows class hierarchies to be defined for node labels, but not for edge labels.

The class hierarchy for relationships is defined in (d), and is part of the graph schema illustrated in Figure 4.4. Lines with open arrow heads declare edge labels. For example, a has edge originates at an OBJECT node and terminates at a RELSHIP node. The relationship hierarchy is extensively used by Constrain and Parse rules, to match spatial relationships with a variable degree of specificity.

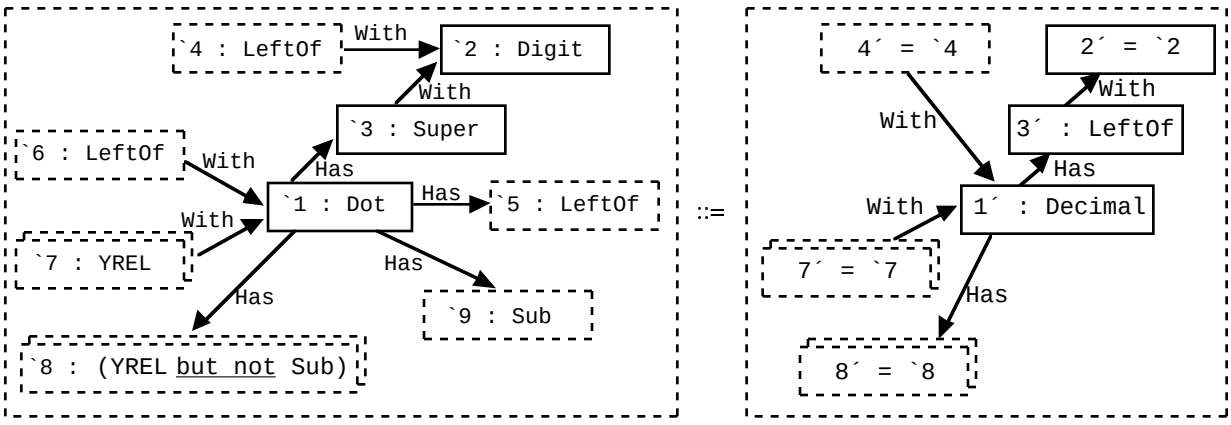
```
(* Remove horizontal or superscript relationships that cross a horizontal line.   *)
(* Class XREL is defined in Figure 4.5(d).                                        *)
production ConstrainA1 =
```




```
end;
```

(a)

```
(* Convert a Dot to a Decimal if the appropriate conditions are met.           *)
(* Other productions convert a Dot to Multiply, and remove Dots that are noise. *)
production ConstrainC1 =
```

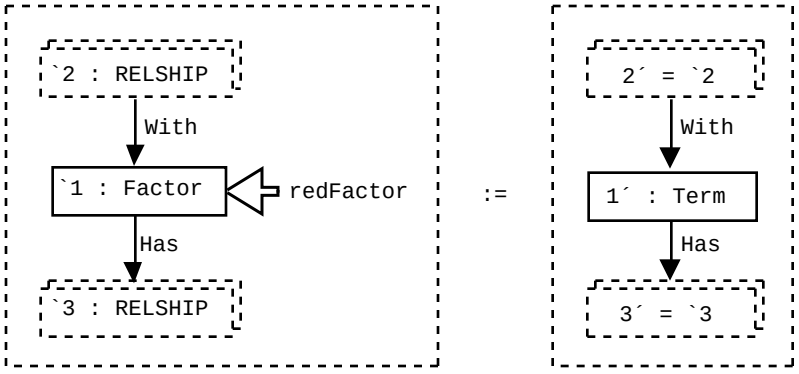



```
end;
```

(b)

Figure 4.6 Two Constrain productions. Rule (a) removes LeftOf, Sub and Super relationships that cross a horizontal line. In (b), nodes with single dashed borders, such as `4, match zero or one nodes in the expression graph. Nodes with double dashed borders, such as `7, match any number of nodes. Node `2 matches the first digit that follows the decimal point. Node `4 transfers an incoming LeftOf relationship (if any) from this digit to the decimal point. Nodes 7 and 8 preserve spatial relationships. Nodes 5, 6, and 9 match spatial relationships that are deleted.

```
(* Reduce the leftmost Factor in a Term *)
production ReduceFactor =
```




```
  transfer 1´.meaning := `1.meaning;
end;
```

Figure 4.6 A Parse production. The restriction `redFactor` tests that the `Factor` node `1 is the leftmost factor and hence should be reduced to a `Term`. If node `1 does not have an incoming `LeftOf` edge, `redFactor` returns `True`, allowing the reduction to go ahead. If there is an incoming `LeftOf` edge, `redFactor` follows this to reach a node *N*. Node *N* may be a multiplication or division involving node `1; in this case `redFactor` fails and node `1 is not reduced from `Factor` to `Term`. A subtle test is needed to determine whether node *N* involves node `1 in a multiplication or division. This test checks that *N* and `1 are aligned, in the sense that they have the same division bars above and below them, and that *N* is a `MULTDIV` or `Operand` node. A MULTDIV node means that the math expression contains an explicit * or / operator, whereas an `Operand` node arises when implied multiplication is used.

As mentioned, imperative control constructs are used to order the application of productions. Syntactically-incorrect expressions cause the graph transformation to stop early, failing to reduce the expression graph to a single node. The following extract illustrates the control constructs.

```
transaction ProcessInput =
    (* Apply four transactions in succession *)
    ReadScannerOutput & BuildPhase & ConstrainPhase & Parse
end;

transaction Parse =
   ReduceSymbols
 & loop  (* Repeat the three transactions until no more productions apply *)
      ReduceFactors & ReduceExpressions & ReduceFractionLines
   end
end;

transaction ReduceExpressions =
  ReduceFactor  (* This production is shown in Figure 4.6 *)
& loop   ReduceMult      end
& loop   ReduceMultDiv   end
& ReduceTerm
& ReduceMinus
& loop   ReducePlusMinus end
end;
```

Graph transformation has been a useful tool in our study of mathematical notation and music notation. In recognition of music notation, graph productions are similarly organized into phases named Build, Weed, and Incorporate [FaBl93]. Execution efficiency can be increased by restricting the types of graph productions that are used [Baum95]. A challenging problem is posed by image noise. This causes the symbol recognizer to misrecognize symbols, or to segment them incorrectly (splitting one symbol into two or merging two symbols into one). The graph transformation described above makes no allowance for noise or uncertainty. In extending this work [FaBl98], the symbol recognizer is allowed to produce several possible interpretations for each symbol. For example, a symbol might be reported as a filled notehead ●, unfilled notehead ○, or noise The initial graph has an `exclusion` edge between every pair of competing interpretations. A `Build` phase adds edges for potentially important spatial relationships. Then `Constrain`, `Split`, and `Incorporate` phases are applied in a loop. `Split` productions enumerate competing interpretations, which arise from the competing symbol identities presented at the start. The competing interpretations are culled by `Constrain` productions. Further work remains to be done, both to address symbol segmentation problems, and to allow application of soft constraints. Soft constraints are constraints that are usually observed in music and mathematical notation; they relate to making the notation readable and aesthetically pleasing. For example, in music notation it is usual to leave more space after longer notes. This provides evidence for resolving a symbol-recognition ambiguity between ● and ○.

Many approaches have been taken in the construction of recognizers for mathematical notation. Our work with graph transformation focuses on accepting expressions with irregular layout. Years ago Anderson proposed an elegant coordinate grammar approach to the problem [Ande77]. This inspiring work assumes a strict symbol layout; for example, the limits of a summation must lie within the x-extent of the 'Σ' symbol. Other approaches include projection profile cutting, stochastic context free grammars, procedurally-coded rules, and statistical labeling for subscript and superscript identification [BlGr97].

We find that diagram recognition problems are very naturally formulated using a graph data structure which must be transformed into a desired output. Graph transformation is a convenient tool for exploring these problems.