A Language for Specifying and Comparing Table Recognition Strategies

by

Richard Zanibbi

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen's University
Kingston, Ontario, Canada
December 2004

Abstract

Table recognition algorithms may be described by models of table location and structure, and decisions made relative to these models. These algorithms are usually defined informally as a sequence of decisions with supporting data observations and transformations. In this investigation, we formalize these algorithms as strategies in an imitation game, where the goal of the game is to match table interpretations from a chosen procedure as closely as possible. The chosen procedure may be a person or persons producing 'ground truth,' or an algorithm.

To describe table recognition strategies we have defined the Recognition Strategy Language (RSL). RSL is a simple functional language for describing strategies as sequences of abstract decision types whose results are determined by any suitable decision method. RSL defines and maintains *interpretation trees*, a simple data structure for describing recognition results. For each interpretation in an interpretation tree, we annotate *hypothesis histories* which capture the creation, revision, and rejection of individual hypotheses, such as the logical type and structure of regions.

We present a proof-of-concept using two strategies from the literature. We demonstrate how RSL allows strategies to be specified at the level of decisions rather than

algorithms, and we compare results of our strategy implementations using new techniques. In particular, we introduce *historical* recall and precision metrics. Conventional recall and precision characterize hypotheses accepted after a strategy has finished. Historical recall and precision provide additional information by describing all generated hypotheses, including any rejected in the final result.

Co-Authorship

Chapter 2 was written in collaboration with my supervisors Dr. Dorothea Blostein and Dr. James R. Cordy. Chapter 2 previously appeared as a paper in the International Journal of Document Analysis and Recognition (Volume 7, Number 1, September 2004).

Acknowledgements

This dissertation is the culmination of years of work, much of it being comprised of tangential research projects, personal interests, and plain wastes of time. I wish to acknowledge here those who helped insure that this document was actually completed, rather than forgotten in a dark corner.

First I wish to thank my supervisors, Dr. Dorothea Blostein and Dr. James R. Cordy, for putting a lot of faith in me and allowing me to make my own mistakes (of which there were many). The central ideas and approach in this thesis were significantly refined through their comments and oversight of my work.

I have been lucky enough to get help from friends when I needed it. In no particular order, these include: Jeremy Bradbury, Chris McAloney, Burton Ma, Harinder Aujla, James Wasserman, Michael Lantz, Dan Ghica, Dean Jin, Edward Lank, Ken Whelan, Amber Simpson, Jiro Inoue, and Michelle Crane. Each of these people endured rambling explanations and complaints about my work, and offered helpful advice and information anyway. Thanks so much, guys.

Dr. John Handley and Dr. George Nagy have provided me with helpful insights into pattern recognition and the research profession. In the summer of 2003, Dr. Nagy reminded me that if I didn't finish my thesis in a couple years' time, my daughter would start school before I stopped. Avoiding that situation has been a great

motivator.

I want to thank the members of Queen's School of Computing support staff that I've dealt with over the last six years: Debby Robertson, Irene Lafleche, Sandra Pryal, Nancy Barker, Tom Bradshaw, Gary Powley, Richard Linley, and Dave Dove. They made my time in the School of Computing enjoyable and productive, and saved me on a few occasions.

I am dedicating this dissertation to Katey, my wife, and Alexandra, our daughter. Katey has supported me through this whole process, and having her to bounce research ideas off of has been a great help. Alix is a healthy dose of reality, fun, and happiness. I love them both very much.

In closing, I'm very happy to report that I managed to finish school before Alix started.

Statement of Originality

This thesis and the research to which it refers are the product of my own work. Any ideas or quotations from the work of other people, published or otherwise, are fully acknowledged in accordance with the standard referencing practices in Computer Science. I gratefully acknowledge the helpful guidance and support of my supervisors, Dr. Dorothea Blostein and Dr. James R. Cordy. The second chapter of this thesis was written in collaboration with Dr. Blostein and Dr. Cordy, and appeared previously in the International Journal of Document Analysis and Recognition (Volume 7, Number 1, September 2004).

Contents

List of Figures

Chapter 1

Introduction

Document recognition[42, 44, 79] was one of the earliest application areas of pattern recognition research. It is motivated by the desire to translate documents into content-based encodings which can then be used in reading machines for the blind, for compressing and editing existing documents, and for information retrieval tasks (e.g. indexing, searching, and clustering). Document recognition may be broken down into a number of subtasks, the best known of which is optical character recognition[92] (OCR). OCR research is concerned with recovering the content of textual regions. However, the non-textual (or 'graphic') regions of a document often include important information. These graphic regions include mathematical notation, machine drawings, and images. This dissertation is concerned with systems for recognizing tables, another common type of graphic region.

In this dissertation we address three methodological problems with current systems for recognizing tables: informal system specifications, the confounding of decision effects, and the effort required for constructing table recognition systems. We motivate table recognition and describe some of its associated challenges in Section

1.1, define the problems to be addressed in Section 1.2, and finally summarize the organization of this document in Section 1.3.

1.1 Table Structure Recognition

The two main tasks of table recognition are locating and decomposing table regions in encoded documents. These tasks have been termed *table detection* and *table structure recognition*, respectively[55]. In the literature, table recognition techniques have been applied to digitized images of document pages, text files, and documents written in a markup language such as HTML.

The information retrieval applications of table recognition systems are intriguing. For example, Rus and Subramanian[100] and Yoshida et al.[121] describe techniques for recognizing tables in documents returned by internet search queries, and then clustering these tables or their contents based on content type. Table regions and their contents may be incorporated into sophisticated search queries[93]. One domain of application is scientific research, where automated or semi-automated tools could be used to collect experimental data from tables in research papers. The collected data could then be examined directly by the researcher, or passed on to automated systems that find patterns such as correlations in data. Another important domain of application is converting table information to audible text for persons who are blind[98]. All of these applications rely heavily on the quality of table detection and structure recognition algorithms, both of which are still maturing.

The specific table recognition problem we will use for illustration in this dissertation is the recognition of table structure from the spatial arrangement of words and lines in a table region. In a real system these regions would need to be produced by

earlier processing. For example, words and lines could be defined by first applying optical character recognition (OCR), and then locating words and lines in the image (i.e. *segmenting* word and line regions within the image).

Even when provided with word and line locations, recognizing table structure is often non-trivial. At the highest level of abstraction, the three main tasks in recognizing table structure are as listed below.

High-level Tasks of Table Structure Recognition:

1. Determining cell locations and their constituent regions (e.g. words)
2. Determining table rows and columns (the *cell topology*)
3. Determining the *indexing structure* of the table, defined as the set of paths through *header* (labelling) cells to *data* cells, which do not label other cells.

As examples of difficulties in performing these three tasks, consider Figures 1.1, 1.2, and 1.3, which are taken from scanned images compiled in the University of Washington English/Technical Document Database[89], a standard data set used by the document recognition community.

For the table in Figure 1.1, it is difficult to determine which columns the header cells at the top of the table are associated with. The 'table' itself appears to be at least two tables (the largest visible area, and the 'Target' and 'Grand Means' labelled columns at the bottom left). This 'table' also contains additional annotations, such as the 'Number of arcs.' How should such annotations be incorporated into a model used to detect and decompose tables?

Arc Target	*Face Left*			*Diff*	*Face Right*				*Mean*		*Reduced Mean*			*v'*	*v*	*v²*
Omega	0	00	06	+3	180	00	09	0	00	07.5	0	00	00	0.0	2.0	3.9
T.4	21	46	29	+4	201	46	33	21	46	31	21	46	23.5	-1.8	0.2	0.1
1 Astro	63	17	21	+5	243	17	26	63	17	23.5	63	17	16	-1.9	0.1	0.0
Wild	100	24	01	+4	280	24	05	100	24	03	100	23	55.5	-3.4	-1.4	2.0
RA	142	10	53	-5	322	10	48	322	10	50.5	142	10	43	-2.9	-0.9	0.8
													Σ	-9.9	0.0	
Omega	45	02	08	+5	225	02	13	45	02	10.5	0	00	00	0.0	-1.8	3.3
T.4	66	48	26	+9	246	48	35	66	48	30.5	21	46	20	1.8	-0.1	0.0
2 Astro	108	19	17	+6	288	19	23	108	19	20	63	17	09.5	4.6	2.8	7.8
Wild	145	26	00	0	325	26	00	145	26	00	100	23	49.5	2.6	0.8	0.6
RA	187	12	55	-9	7	12	46	187	12	50.5	142	10	40	0.1	-1.7	2.9
													Σ	+9.1	0.0	
Omega	90	05	07	+3	270	05	10	90	05	08.5	0	00	00	0.0	-0.2	0.1
T.4	111	51	33	-2	291	51	31	111	51	32	21	46	23.5	-1.8	-2.0	3.9
3 Astro	153	22	24	-2	333	22	22	153	22	23	63	17	14.5	-0.4	-0.6	0.4
Wild	190	28	55	+4	10	28	59	190	28	57	100	23	48.5	3.6	3.4	11.6
RA	232	15	48	+2	52	15	50	232	15	49	142	10	40.5	-0.4	-0.6	0.4
													Σ	+1.1	0.0	
Omega	135	07	10	12	315	07	22	135	07	16	0	00	00	0.0	0.1	0.0
T.4	156	53	38	-4	336	53	34	156	53	36	21	46	20	1.8	1.8	3.3
4 Astro	198	24	29	+7	18	24	36	198	24	32.5	63	17	16.5	-2.4	-2.3	5.3
Wild	235	31	11	0	55	31	11	235	31	11	100	23	55	-2.9	-2.8	7.8
RA	277	17	51	+4	97	17	55	277	17	53	142	10	37	3.1	3.2	10.2
													Σ	-0.4	0.0	
																$\Sigma v^2 = 64.4$

Target	**Grand Means**
Omega	0 00 00.0
T.4	21 46 21.8
Astro	63 17 14.1
Wild	100 23 52.1
RA	142 10 40.1

Number of arcs a = 4;

Number of targets t = 5

Standard Deviation of Grand Mean Direction: $s_T = \sqrt{64.4/4\ (4\text{-}1)(5\text{-}1)} = \pm 1.16''$

Table 1. Example of traditional station adjustment procedure and error analysis for horizontal direction measurements. [Originally prepared by K.I.Groenhout, January 1980 and subsequently modified by the authors.]

Figure 1.1: Table from UW-I database, image h01c. Among other challenging features, this table has ambiguous cell topology and even ambiguous scope. Which columns are spanned by 'Face Right' at the top of the table? Are the Grand Mean summaries at the bottom left part of the larger table, or a separate table? How does one characterize the additional annotations (e.g. 'Number of arcs')?

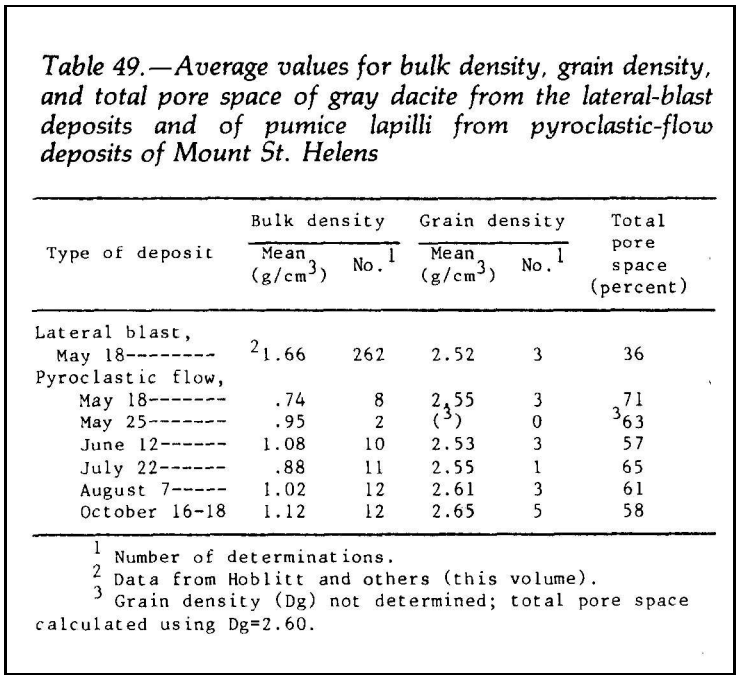


Table 49.—Average values for bulk density, grain density, and total pore space of gray dacite from the lateral-blast deposits and of pumice lapilli from pyroclastic-flow deposits of Mount St. Helens

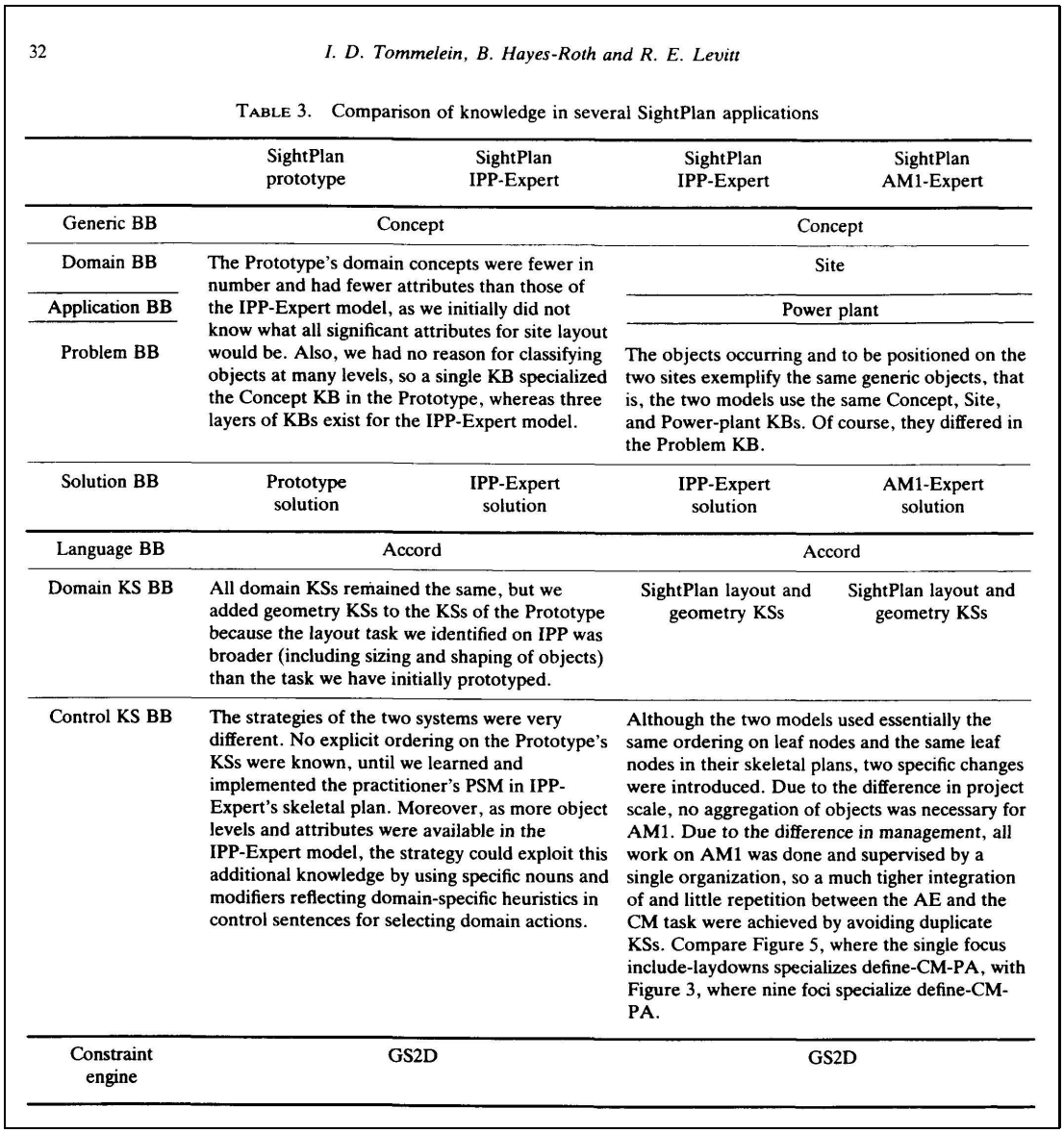
Type of deposit	Bulk density		Grain density		Total pore space (percent)
	Mean (g/cm^3)	No.[1]	Mean (g/cm^3)	No.[1]	
Lateral blast,					
May 18--------	[2]1.66	262	2.52	3	36
Pyroclastic flow,					
May 18-------	.74	8	2.55	3	[3]71
May 25-------	.95	2	([3])	0	[3]63
June 12------	1.08	10	2.53	3	57
July 22------	.88	11	2.55	1	65
August 7-----	1.02	12	2.61	3	61
October 16-18	1.12	12	2.65	5	58

[1] Number of determinations.
[2] Data from Hoblitt and others (this volume).
[3] Grain density (Dg) not determined; total pore space calculated using Dg=2.60.

Figure 1.2: Table from UW-I database, image a038. This table contains nested column header cells at the top of the table, which can be difficult to detect when lines do not separate them and/or spacing is irregular. This table makes use of footnotes for both header *and* data cells, which confuses the row structure (see the row associated with 'Pryoclastic Flow → May 25')

The table in Figure 1.2 contains footnotes for both header and data cells. In addition to being difficult to interpret, in this example the footnotes confuse the row structure of the table. It requires some effort to determine whether some of the footnote numbers are associated with the cell above or below themselves. This table also contains column headers that are nested (e.g. 'Bulk density' → 'Mean (g/cm^3)'). When lines do not separate nested headers as in Figure 1.2, it can be difficult to determine the indexing structure of these cells, and it may even confuse cell locations.

There are a number of more fundamental challenges for table recognition. Defining



32 I. D. Tommelein, B. Hayes-Roth and R. E. Levitt

TABLE 3. Comparison of knowledge in several SightPlan applications

	SightPlan prototype	SightPlan IPP-Expert	SightPlan IPP-Expert	SightPlan AM1-Expert
Generic BB	Concept		Concept	
Domain BB	The Prototype's domain concepts were fewer in number and had fewer attributes than those of the IPP-Expert model, as we initially did not know what all significant attributes for site layout would be. Also, we had no reason for classifying objects at many levels, so a single KB specialized the Concept KB in the Prototype, whereas three layers of KBs exist for the IPP-Expert model.		Site	
Application BB			Power plant	
Problem BB			The objects occurring and to be positioned on the two sites exemplify the same generic objects, that is, the two models use the same Concept, Site, and Power-plant KBs. Of course, they differed in the Problem KB.	
Solution BB	Prototype solution	IPP-Expert solution	IPP-Expert solution	AM1-Expert solution
Language BB	Accord		Accord	
Domain KS BB	All domain KSs remained the same, but we added geometry KSs to the KSs of the Prototype because the layout task we identified on IPP was broader (including sizing and shaping of objects) than the task we have initially prototyped.		SightPlan layout and geometry KSs	SightPlan layout and geometry KSs
Control KS BB	The strategies of the two systems were very different. No explicit ordering on the Prototype's KSs were known, until we learned and implemented the practitioner's PSM in IPP-Expert's skeletal plan. Moreover, as more object levels and attributes were available in the IPP-Expert model, the strategy could exploit this additional knowledge by using specific nouns and modifiers reflecting domain-specific heuristics in control sentences for selecting domain actions.		Although the two models used essentially the same ordering on leaf nodes and the same leaf nodes in their skeletal plans, two specific changes were introduced. Due to the difference in project scale, no aggregation of objects was necessary for AM1. Due to the difference in management, all work on AM1 was done and supervised by a single organization, so a much tigher integration of and little repetition between the AE and the CM task were achieved by avoiding duplicate KSs. Compare Figure 5, where the single focus include-laydowns specializes define-CM-PA, with Figure 3, where nine foci specialize define-CM-PA.	
Constraint engine	GS2D		GS2D	

Figure 1.3: Table from UW-I database, image v00c. This table has varying column and row structure. It is difficult without knowing the subject domain to determine the indexing structure of the table. Are the column headers 'wrapped'?

'tables' is difficult because tables are frequently adapted to suit individual needs, such as the additional annotations in Figure 1.1. This often makes it difficult to define what constitutes a 'correct' recognition result[51]. Adding to the complexity of the problem, frequently information about the language or subject matter of a table is essential for proper interpretation[57]. As an example, consider the columns and indexing structure of the table shown in Figure 1.3.

1.2 Problem Statement

This dissertation addresses three methodological problems in the current table recognition literature. These are the use of informal system specifications, the confounding of decision effects, and the difficulty of constructing table recognition systems. We describe each of these in the following.

Informal System Specifications. Table recognition systems are described informally in the literature. Models used in decision making are often only partially described, and must be inferred from the sequence of described operations, which may also be partial. This makes the replication and comparison of techniques difficult, and sometimes impossible. These informal descriptions also make it hard to discern the decisions made by a table recognition system.

Confounded Decision Effects. Intermediate interpretations are not preserved in the table recognition literature. No record of how hypothesized table properties are altered or rejected is maintained. Recognition results are then analyzed only from their final interpretation,

making it difficult to determine the effect of individual decisions: the aggregate effect of a set of decisions cannot be separated into effects of individual decisions. This makes it difficult to determine which decision(s) caused a recognition 'error,' for example.

Ease of Implementation. Most table recognition systems are implemented as a program in a general-purpose programming language, though some systems use syntactic methods[2, 26, 36, 96, 107] to specify recognition behaviour using models of recognition targets (a model of table structure, for example). The systems built in general-purpose languages tend to use models that are simpler and more flexible than those described for syntactic methods: particularly in situations such as early design and experimentation, this is an advantage. Currently there is no middle ground: methods either require complete and often complex model definitions, or are loosely defined as a sequence of operations in a general-purpose programming language, in which case a great deal of additional infrastructure must be constructed. The effort required to build infrastructure for these systems would be better spent on designing and refining recognition methods.

The last two problems are related to the first. Formalizing and recording decisions made by a system would reduce the confounding of decision effects. Difficulties arise for systems built in general-purpose programming languages because common subsystems are not formalized and then constructed in a way that allows reuse.

Reuse is one of the largest benefits of syntactic techniques: these systems are 'programmed' by specifying only the model. However, one must be in a position to properly define a useful model, which for complicated tasks may involve detailed knowledge of how a model is interpreted and applied (e.g. an understanding of search algorithms used, including any heuristics). This is the benefit of informality: we can flexibly alter our problem descriptions and solutions as needed *independently* of other descriptions and solutions. Problems arise when we then need to unify a set of such informal problem descriptions and solutions.

In this dissertation we address these three methodological problems by proposing a middle ground for formalization: a language that specifies table recognition methods using a fixed set of decision types and a syntax that captures the types of model components used in decision making. The is the Recognition Strategy Language (RSL), a simple functional 'glue' language for combining arbitrary decision techniques. The approach taken in RSL was inspired in part by the Tcl language[85], which is used to combine systems constructed in different languages and architectures. As we shall see, various parts of the RSL language were designed specifically to address the problems above.

1.3 Overview of Chapters

In this Section we summarize the organization and content of the chapters in this dissertation.

In Chapter 2 table recognizers are characterized as algorithms for recovering table model instances in data, and recognition operations are divided into three broad classes: *observations* that describe data, *transformations* that manipulate data, and

inferences that generate and test hypotheses. A brief summary of evaluation techniques in the literature is also provided.

Chapter 3 characterizes table recognition problems as a simple class of imitation games, where table recognizers act as strategies in the game. We formalize these strategies using the Recognition Strategy Language (RSL), a simple functional 'glue' language that combines inferencing functions. Among other benefits discussed, RSL automatically maintains data structures for recognition results.

Chapter 4 describes the implementation of RSL, along with additional analysis and visualization tools. Much of Chapter 4 is not necessary for understanding the subsequent chapters in the dissertation, but provides details of interest that may be consulted as needed.

We demonstrate the usefulness of RSL for specifying and implementing table recognition systems in Chapter 5, where we describe RSL implementations for two of the table structure recognition algorithms in the literature, those of Handley[43] and Hu et al.[53]. We also describe a simple game in which these algorithms imitate cell locations in tables defined by the author. We analyze the outcome of this game using two new metrics, *historical recall* and *historical precision*, which take rejected and revised hypotheses into account.

Finally, in Chapter 6 the contributions of the dissertation, open problems, and directions for extending and generalizing the work are presented.

Chapter 2

A Survey of Table Recognition: Models, Observations, Transformations, and Inferences

2.1 Introduction

Many documents contain tables that could be recovered for reuse, compression, editing, and information retrieval purposes. In table recognition, a definition of table location and composition (a table model) is used to recover tables from encoded documents. Hu et al.[55] have termed the two main sub-tasks of table recognition table detection and table structure recognition. In table detection, instances of a table model are segmented. In table structure recognition, detected tables are analyzed and decomposed using the table model. In this chapter we break down table detection and structure recognition further, describing both as sequences of three basic operations: observations, transformations, and inferences. Observations include

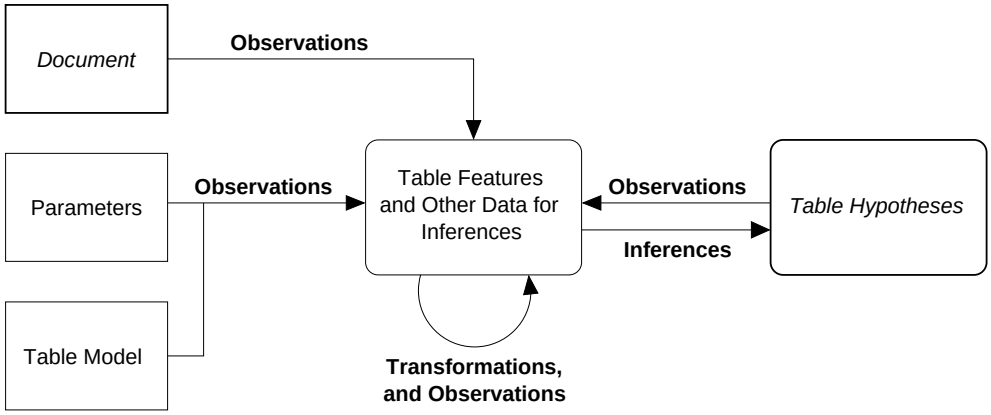


Figure 2.1: The Table Recognition Process. A table model defines the structures that a table recognizer searches for. Table recognizers detect and decompose tables using observations, transformations, and inferences. Inferences generate and test table location and structure hypotheses. Observations provide the data used by inferences; these are feature measurements and data lookups performed on the input document, table model, input parameters, and existing features and hypotheses. Transformations of features permit additional observations. Input parameters define or constrain the table model, observations, transformations, and inferences of a table recognizer.

---

feature measurements and data lookup, transformations are operations that alter or restructure data, and inferences generate and test hypotheses (e.g. table locations).

In this survey we present the table recognition literature in terms of the interaction of table models, observations, transformations, and inferences, as presented in Figure 2.1. Surveys that take other views of the literature are also available[42, 74, 75]. We use the view presented in Figure 2.1 to assist us in answering important questions about the decisions made by table recognizers. What decisions are made? What is assumed when decisions are made? What inferencing techniques make decisions?

On what data are decisions based? Table models play a crucial role in the decision making process, as they define which structures are sought after and define or imply a set of assumptions about table locations and structure. Input parameters define decision thresholds and tolerances, provide values for table model parameters, and may include additional information used in decision making.

In the remainder of this chapter we define tables and describe table models used in recognition (Section 2.2), outline the observations, transformations and inferences used by different systems for table detection and structure recognition (Sections 2.3, 2.4, and 2.5), address performance evaluation methods used to determine the sufficiency of a table recognizer for a specific recognition task (Section 2.6), and finally identify open problems and conclude in Section 2.7.

2.2 Table Models

Tables are one of the visualizations people use to search and compare data[71]. More specifically, they visualize indexing schemes for relations, which may be understood as a set of $n-$tuples where n is the number of sets in the relation[31]. The sets of a relation underlying a table are called domains or dimensions. A relation may be presented many different ways in a table. Dimensions may be laid out in different row and column arrangements, repeated, or ordered in various ways. The arrangement of dimensions in a table affects which data are most easily accessed and compared[20, 41, 71].

The parts of a table are described in Figure 2.2. Dimensions of a relation whose elements are to be searched and compared in a table have their elements located in the body; the names and elements of remaining dimensions are placed in the boxhead

and stub as headers, which are then used to index elements located in the body. The stubhead may contain a header naming or describing the dimension(s) located in the stub. Often headers are nested to visually relate dimension names and elements, and to factor one dimension by another. For example, the dimension 'school term' is factored by 'year' in the stub of the table in Figure 2.2; this is indicated by indenting (nesting) the 'school term' elements (e.g. 'Winter') below the 'year' elements (e.g. '1992'). As another example, the header for element 'Ass3' is nested below the name of its associated dimension ('Assignment'). Regions where individual dimension names and elements are located are called cells. A group of contiguous cells in the body is referred to as a block. Cells are separated visually using ruling lines (e.g. the boxhead and stub separators in Figure 2.2) and whitespace, or tabulation[31]. This results in the familiar arrangement of cells in rows and columns. If the boxhead, stub, and stubhead are absent, we are left with a list or matrix of values; these do not have headers used to index data, which is a defining feature of tables.

In practice individuals often alter or adapt the parts of a table as presented in Figure 2.2. For example, headers or explanatory text might appear in the body of a table. In some cases, tables even contain tables within cells, or are compositions of tables, producing complicated indexing structures[41, 114]. However, the majority of tables studied in the table recognition literature are described well by Figure 2.2.

Tables also often have associated text, including titles (e.g. Figure 2.6), captions, data sources (e.g. Figure 2.3), and footnotes or additional text that elaborate on cells (e.g. Figure 2.6). The text in a document that cites a table sometimes provides additional information regarding a table's contents[30]. The focus in table recognition so far has been on recovering tables themselves; only a small number of papers have

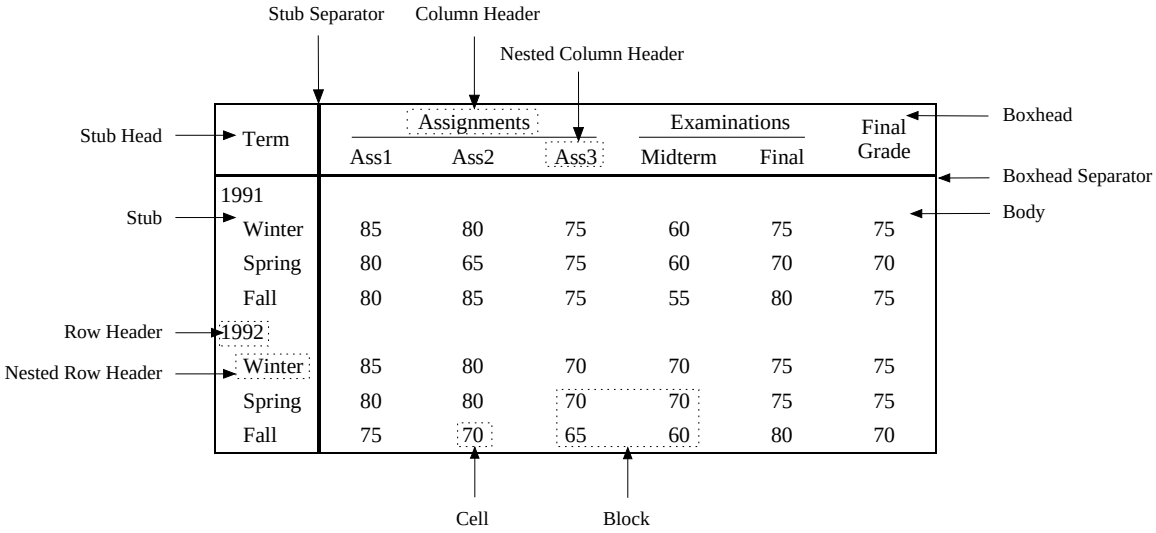


Figure 2.2: Table Anatomy. The example and terms shown here are taken from Wang[114], where terminology from the Chicago Manual of Style[38] is used. Though not shown in this figure, tables often have associated text regions, such as a title, footnotes, or the source for table data.

---

addressed text associated with a table[30, 87, 93, 111].

In this chapter we will restrict our discussion to tables that present textual data, as this is the class of tables that have been studied for recognition. This includes tables encoded in text files (e.g. ASCII), such as the one rendered for presentation in Figure 2.6. Sources of plain text tables include email messages and automated reporting systems (e.g. the EDGAR financial reporting system[67]). The remainder of Section 2.2 addresses the physical and logical structure of tables, HTML encodings of table structure, and models used in generating and recognizing tables.

2.2.1 Physical and Logical Structure of Tables

As with all entities sought after in document recognition, tables have physical and logical structure[42, 44, 79]. For tables, physical structure describes where regions containing parts of tables are located in an image or plain text file (e.g. Figure 2.6), while logical structure defines the types of these regions and how they form a table. All regions of interest in a table have both physical and logical structure. For example, the location of a line in a table image is part of the physical structure of a table, while the type of a region (in this case, 'line') is part of the logical structure of a table. Similarly, the intersection of two lines is defined in logical structure, while the location of the intersection is defined using geometry (part of physical structure).

We define the most abstract level of logical structure for tables to be the indexing scheme from headers to cells located in the body of a table. This defines a relation describing a table, but it may not be minimal (e.g. the table may repeat dimensions of a relation). Defining the minimal relation underlying a table requires knowledge about the subject matter domain of the table, such as how dimensions are related, fixed ranges for particular dimensions, or synonyms for dimension names. This is information that tables themselves do not provide: their function is to visualize an indexing scheme for data in a relation, not to interpret the relation in the data domain. As a result, we consider anything more abstract than the table's indexing structure to be part of a table's subject matter domain rather than part of the logical structure of the table itself.

For a table encoding, the least abstract level of logical structure describes the type of the smallest regions of interest. In an image this might be a connected component. For plain text files, this might be be regions of adjacent non-whitespace character

(connected components of characters in the text file). For HTML files, this might be a tagged cell. At this level there is no relational structure, only primitive regions with types.

Intermediate levels of logical structure describe the composition of smaller regions into larger ones, and relate regions to one another. For example, a series of connected components may be joined into a line (relating the connected components), which is found to intersect with another line (relating the two lines). An important intermediate level of logical structure describes cell adjacencies, or topology. Cell topology is often described using a table grid. Table grids are formed by extending all line and whitespace cell separators to the edges of a table region (see Figure 2.4). The grid allows indexing cell locations using a scheme similar to those in spreadsheets, in which columns and rows are enumerated. Depending on the structure of a table, grid locations may be empty, or cells may span multiple grid locations as in Figure 2.4.

The physical structure of a table can be encoded in a text or image file, while logical structure may be encoded using a markup language such as HTML (see Figure 2.5). The tags in the markup language describe data types and relations (i.e. they define a graph on labelled data). In HTML, tags can be used to define the table grid (contents and relative positions of cells), types of separator (lines vs. whitespace), and the location of the body, header (boxhead), and footer areas, and indexing structure. HTML does not encode the stub location or underlying relation of a table.

In practice, tables encoded in HTML often do not use the indexing, header, or footer tags, and use tags that are not part of the table tag set (for an example, see Figure 2.5). Also, the table environment is often used to layout lists and matrices of data in a grid, with no indexing by headers[58, 116].

Journal	Full Name	Details	
		Appears	Publisher
TPAMI	IEEE Transactions on Pattern Analysis and Machine Intelligence	monthly	IEEE
IJDAR	International Journal on Document Analysis and Recognition	quarterly	Springer-Verlag
PR	Pattern Recognition	monthly	Elsevier
IJPRAI	International Journal on Pattern Recognition and Artificial Intelligence	eight times/year	World Scientific

Source: from a listing of pattern recognition journals provided online at http://www.ph.tn.tudelft.nl/PRInfo/

Figure 2.3: A Table Describing Document Recognition Journals. This fully ruled table was rendered by an HTML viewer using the source file shown in Figure 2.5. Note the text below the table indicating the source of the data presented in the table.

2.2.2 A Table Model for Generation: Wang's Model

The most complete table model in the literature was designed to support generating table images from logical structure descriptions by Wang[114]. Wang's model separates table structure into three parts: an abstract indexing relation, a topology defining the placement and ordering of dimensions within the boxhead, stub, or both regions, and formatting attributes which include fonts, separator types, and layout constraints. Formatting attributes are associated with logical structures: dimensions and their elements, table regions (e.g. body or stub), and blocks of cells. In Wang's scheme, the table grid and cell topology (cell adjacencies) are defined by a combination of the topology on dimensions and formatting rules.

Wang's model is appealing because it is reasonably simple and separates concerns cleanly, with editing driven by logical structure rather than blocks of cells, as in many

A1–A2	B1–B2	C1–D1	
		C2	D2
A3	B3	C3	D3
A4	B4	C4	D4
A5	B5	C5	D5
A6	B6	C6	D6

Figure 2.4: Grid Describing the Location of Cells for the Table in Figure 2.3. Table grids are formed by extending all separators to the border of a table. Here rows are represented by numbers, and columns by letters. The table in Figure 2.3 has a 'Manhattan' layout, in which all separators meet at right angles. For non-Manhattan layouts cells may not be rectangle-shaped (and the table grid more complex as a result). Occasionally cells occupy more than one location in the table grid: these are called spanning cells. The topmost cells in Figure 2.3 are all spanning cells, located at grid locations A1-A2, B1-B2 and C1-D1.

conventional table editing tools (e.g. spreadsheets). In Wang's scheme what we have called logical structure is separated into layout, presentation, and logical classes (see Section 4.1 of Wang's thesis[114]). Wang's model does not describe footnotes or other text associated with a table (e.g. titles or captions). Stub heads are assumed to be empty, and headers are assumed to be located only in the boxhead and stub of the table. Wang's model is also not designed to handle nested tables, in which cell contents are themselves tables; however, this type of table is fairly unusual.

2.2.3 Table Models for Recognition

In the literature, table models for recognition must support two tasks: the detection of tables, and the decomposition of table regions into logical structure descriptions. They tend to be more complex than generative models, because they must define

```
<TABLE RULES=ALL BORDER=1 CELLPADDING=5 ALIGN=CENTER>
  <THEAD>
    <TR>
      <TD ROWSPAN=2>Journal</TD>
      <TD ROWSPAN=2 ALIGN=CENTER>Full Name</TD>
      <TD COLSPAN=2 ALIGN=CENTER>Details</TD>
    </TR>
    <TR>
      <TD>Appears</TD>
      <TD>Publisher</TD>
    </TR>
  </THEAD>
  <TBODY>
    <TR>
      <TD>TPAMI</TD>
      <TD>IEEE Transactions on Pattern Analysis
          and Machine Intelligence</TD>
      <TD>monthly</TD>
      <TD>IEEE</TD>
    </TR>
    ...
  </TBODY>
</TABLE>

<P ALIGN=CENTER>
  Source: from a listing of pattern recognition
  journals provided online at
  http://www.ph.tn.tudelft.nl/PRInfo/
</P>
```

Figure 2.5: Partial HTML Source Code for the Table in Figure 2.3. Note how ROWSPAN and COLSPAN attributes are used to define cells that span rows and columns respectively, and that the table boxhead (THEAD) and body (TBODY) regions are explicitly labelled. The data source for the table is labelled as a paragraph (P). This type of formatting rather than logical structure description is common in practice, making automated retrieval and clustering tasks for HTML tables difficult[39, 58, 116, 121].

and relate additional structures for recovering the components of generative models. Figure 2.7 presents a number of these additional structures, such as connected components and line intersections. The usefulness of a table model for recognition in a set of documents is determined by the proportion of tables described by the model, the degree to which the model excludes other types of data (e.g machine drawings), and how reliably the objects and relations of the model can be inferred. The efficacy of a model is difficult to assess in advance of doing a performance evaluation (see Section 2.6). Usually table models are designed informally by trying to describe the tables in a set of documents (e.g. as was done for Wang's model[114]).

Only a small number of models for recognition have been described explicitly in the literature[31, 56, 87]. These models are less complete than Wang's. More commonly the structures, relations, and assumptions of a table model for recognition are

Table I. Available Document Recognition Software

Source	Packages	Web Site	Note
AABBYY	FineReader	http://www.abbyy.com	Commercial
ScanSoft	OmniPage OmniForm TextBridge	http://www.scansoft.com	Commercial
ExperVision	TypeReader WebOCR	http://www.expervision.com	Commercial: WebOCR on-line service is free
CharacTell	Simple OCR	http://www.simpleocr.com	Commercial
Musitek	SmartScore	http://www.musitek.com	Commercial Music Recognition, Scoring
Michael D. Garris (NIST)	Form-Based Handprint Rec. Sys.	http://www.itl.nist.gov/ iaui/894.03/databases/ defs/nist_ocr.html	Free, with source code Unrestricted Use. Large training sets
Donato Malerba et. al.	Wisdom++	http://www.di.uniba.it/ ~malerba/wisdom++/	Free for research and teaching purposes
R. Karpischek et. al.	Clara OCR	http://www.claraocr.org	GPL*
J. Schulenburg et. al.	JOCR	http://jocr.sourceforge.net	GPL*
Klaas Freitag	Kooka	http://www.kde.org/apps/kooka	GPL*, Scanning and OCR interface

*GPL: Freely available under GNU General Public License

Figure 2.6: A Table Describing Available Document Recognition Software. This table is from an ASCII text file, and is unruled. Note the title and the footnote below the table; the footnote is referenced using asterisks in the rightmost column of the table. Together, the title and footnote span all the gaps between columns. This type of arrangement complicates the detection of separators in projection profiles (horizontal and vertical histograms of foreground pixels/characters[6, 18, 43, 47, 61, 66, 70, 100, 111, 124]).

Primitive Structures

- Run lengths[18, 19, 66, 70, 120]
- Connected components[1, 48, 50, 63, 66, 70, 103, 120, 124]
- Separators
  - Lines[18, 19, 62, 66, 91]
  - Whitespace [37, 43, 52, 57, 65, 82, 111, 117, 124]
- Intersections
  - Of separators[4, 5, 18, 66, 100, 109, 112, 119]
  - Of lines and text[6, 50, 122]
- Characters
  - Provided in text or markup files[31, 39, 52, 63, 82, 93, 110]
  - From Optical Character Recognition[7, 65, 73, 87, 103, 111]
- Text lines[62, 84, 102, 103]
- Other Symbols
  - Arrow heads (to repeat cell values[6])
  - X's (to cancel cells[4])

Table-Specific Structures

- Table grid[5, 37, 43, 48, 61, 70, 111, 124]
- Cells
  - Multi-line cells[43, 54, 60, 82]
  - Spanning cells[39, 87, 111]
  - Cell Topology, usually as rows and columns of cells[31, 43, 48, 54, 61, 82, 100, 110, 124]
- Table regions: boxhead, stub, and body[54, 59, 93, 121]
- Captions, titles, sources, footnotes, and other text associated with tables[30, 87, 93, 111]
- Whole Tables (for table detection[17, 39, 52, 62, 64, 65, 70, 82, 100, 109, 116, 118])
- Indexing structure
  - Indexing relation for tables[31, 39, 110, 121]
  - Entry structure in tables of contents[7, 10, 106, 107]

Figure 2.7: Types of Structures in Table Models for Recognition. For table structure recognition, the most common outputs are the table grid, cell topology, and table regions (body, stub, and boxhead). Less commonly, some papers go further and encode the indexing relation[31, 39, 110, 121] or entry structure in tables of contents[7, 10, 106, 107]. Multi-line cells contain mutliple text lines.

---

determined by the sequence of observations, transformations, and inferences used by a table recognizer. As an example, from operations that locate column separators at gaps of a vertical projection, we learn the recognition model has a notion of horizontally adjacent columns, where columns are separated by uninterrupted whitespace gaps; this implicit model cannot describe the table shown in Figure 2.6. In many papers the description of operations, and thus the reader's view of the table model, is partial.

A table model may be static, where all parameters are fixed before run-time, or

adaptive, where parameters are altered at run-time based on input data. Figure 2.8 lists a number of static and adaptive parameters of table models. Some adaptive parameters used in table recognition are fairly sophisticated, including line grammars for tables of contents[7], and regular-expressions describing the types of cells in a table[87, 100]. In the literature, model parameters have been set both manually and using machine learning methods[17, 88]. Parameters have included encodings of domain knowledge such as bigrams[57] and ontologies[110] (graph-based knowledge encodings); these encodings are used in the analysis of cell content. Thresholds, tolerances, domain knowledge encodings, and other parameters constrain a table model, and consequently affect what may be inferred by a table recognizer. In this way, they specify a set of assumptions about table location, structure, and content.

As for any other pattern recognition model, there are a number of issues in designing a table model including the 'curse of dimensionality' (the required training sample size growing exponentially as the number of parameters increases) and the complexity of rule-based systems. Perlovsky has provided a brief and informative history of these two problems[86].

2.3 Observations

Observations measure and collect the data used for decision making in a table recognizer. As shown in Figure 2.1, observations may be made on any available data; the input document, table model, input parameters, existing hypotheses, or the current set of observations. Figure 2.8 lists a number of observations made by table recognition systems. Observations may be categorized by the type of data on which an observation is made: images and text files (physical structure), descriptions of table

structure (logical structure), sets of existing observations (descriptive statistics), or parameters.

For physical structure, observations include geometry, histograms, and textures. Geometric observations include perimeter, representative points (e.g. centroid), area, height, width, aspect ratio and angle. They also include distances, angles, and areas of overlap between two regions. Histograms are often observed when locating text lines and to define the table grid. Textural features include cross-counts (a transition count for a line of pixels in a binary image or characters in a text file) and density (proportion of 'on' to 'off' pixels in a binary image, or character to blank cells in a text file). Texture metrics have been used to classify regions[66, 111].

For logical structure, observations made include table structures, edit distance, cell cohesion measures, graphs, and table syntax. The edit distance[13] from logical structure description A to logical structure description B is a weighted linear combination of the number of insertions, deletions, and substitutions required to transform A to B. In the table recognition literature, edit distance has been used to derive regular expression 'types' for columns[87, 100] and in performance evaluation (see Section 2.6). Cell cohesion measures[59, 110, 116] are used to measure whether cells exhibit dimension name and element relationships (e.g. in a column of cells[59, 110]), and to determine the consistency of cell properties in a block of cells[116].

For descriptive statistics, observations include cardinality (counting), probabilities, weighted linear combinations of observations, comparisons, and summary statistics. Variance and standard deviation have been employed to define tolerances and thresholds[62]. In addition, periodicity (spatial regularities or intervals) has been used to classify primitive text regions[102] and for detecting regularities in row and

Physical Structure

- Geometry
  - Height, Width, and Area
  - Aspect ratio (height:width)
  - Angle (e.g. of a line)
  - Skew estimation
    - From skew of detected lines[69]
    - From bounding boxes[62]
  - Docstrum: angle, distance between connected components[84]
  - Overlap of regions (e.g. table regions[17])
  - Region perimeter[50, 100]
  - Representative point
    - Centroid[84]
    - Top-left corner[119]
    - Text baseline y-position[48]
  - Distance between points (e.g. between centroids[84])
- Histograms (Projection Profiles)
  - Projected image pixels[19, 80, 102]
  - Projected bounding boxes[40, 47, 61]
  - Boxes projected as symmetric triangles [124]
  - Boxes projected as 'M' shapes of constant area[65]
  - Weighted projections (e.g. by vertical position[43, 124])
- Texture
  - Value transition count (cross-counts)[66, 111]
  - Pixel density[16]
  - Character density[31]

Parameters

- Static or Adaptive
  - Probability (e.g. for table detection[118])
  - Thresholds (adaptive examples: [47, 62, 124])
  - Tolerances (e.g. for use in X-Y cutting[16])
  - Weights (e.g. for linear combinations[64])
- Adaptive
  - Line grammar (e.g. for table of contents[7])
  - Regular expressions for cell contents[87, 100]
- Encoded Domain Knowledge (Static)
  - Word bigrams[57]
  - Ontologies[110]

Logical Structure

- Table structures (see Figure 2.7)
- Edit distance[13]
  - Deriving reg. expressions for strings[87, 100]
- Cell block cohesion measures[59, 110, 116]
- Graphs
  - Line intersections[112]
  - Form structure[15]
  - Table indexing structure[37, 55]
- Table Syntax (as grammars; see Figure 2.12)

Descriptive Statistics

- Cardinality (counting)
- Probability (e.g. computed from a sample)
- Weighted Linear Combinations of Observations
  - 'Columness' of a region[64]
  - 'Tableness' of a region[118]
- Comparisons
  - Difference (e.g. between heights[110])
  - Derivative (e.g. of histograms[19, 102])
  - Inner ('dot') product and cosine of vectors [103, 116, 121]
  - Correlation (e.g. of text line spacings[65])
  - Word uniqueness[116, 121]
- Summary Statistics
  - Range, Mean, Median
  - Variance/standard deviation[62]
  - Periodicity
    - In histograms[102]
    - In column, row structure[116]
    - Line/string periodicity[93]

Figure 2.8: Observations in the Table Recognition Literature. Observations are classified based on whether they are taken from an image or text file containing a table (physical structure), from a description of table structure and/or content (logical structure), from a set of existing observations (descriptive statistics), or from system parameters. Static parameters are set before execution; adaptive parameters are set at run-time.

column structure (e.g. in HTML[116] and plain text[93]).

Parameters of table recognizers were discussed in Section 2.2. Here we will elaborate further on the use of domain knowledge observations in table recognition, as this is a promising approach in table recognition. Hurst has provided a number examples in which layout information alone is insufficient for defining table grids, cell scopes, and cell topology[57]; the analysis of table content relative to a domain model is required in these examples. To address this, he and Nasukawa[60] proposed improving cell segmentation by constraining detected text continuations using word bigrams. The constraints afforded by these bigrams appear to improve cell segmentation and topology analysis[57]. In Section 2.5.2 we describe a system by Tubbs and Embley using the correspondence of cell contents to relationships and concepts in an ontology for detecting header nesting and factoring, and in computing cell cohesion measures[110].

2.4 Transformations

Transformations restructure existing observations to emphasize features of a data set, to make subsequent observations easier or more reliable. Figure 2.9 lists transformations used in the table recognition literature. As we did for observations, we classify transformations by the type of data to which they are applied: physical structure, logical structure, or to descriptive statistics.

Physical structure transformations include the Hough Transform[45], which is used to approximate parameters of geometric shapes. The Hough Transform is commonly applied to table images in order to support the detection of table lines. Affine transformations[99] are also commonly used, in particular rotations and shearing

Physical Structure

- Image Binarization (e.g.[43, 102])
- Image compression
  - Run-length encoding[120]
  - Block adjacency graph[122]
- Image resampling
  - Subsampling[18, 50]
  - Supersampling[84]
  - Quadtree[102]
- Hough transform[45] (e.g. for locating lines)
- Affine transformations: rotation, shearing, translation and scaling[99], (e.g. used for de-skewing an image[62])
- Interpolation to recover parts of characters intersected by lines[122]
- Mathematical Morphology[45]
  - RLSA (Run-length smoothing algorithm[120])
  - Dilations and closings
    - In images[83, 6]
    - In text files[63]
    - For joining lines[122]
  - Thinning[61]
  - Edge detection[112, 119]

Logical Structure

- Merging/splitting of regions
  - Cells[59]
  - Tables[100, 121]
  - Splitting region at detected separators[70]
- Graph/tree transformations
  - To correct structural errors[7]
  - Join regions into a table region[96]
- Filtering
  - Small regions for noise reduction[68, 83, 103]
  - Textures, images and half-tones[103]
- Insertion of table lines[48, 70]
- Produce boxes from line intersections [3, 112]
- Sorting and Indexing
  - Sorting (e.g. boxes by geometric attributes[8])
  - Indexing (e.g. of cells[37, 61])
- Translation
  - HTML to character matrix[39, 116]
  - Map strings to regular expressions[87]
  - Transform tokens of a single class to a uniform representation ([82, 87])
  - Encoding recognized form data[15, 122]
  - Indexing relation of a table[37]

Descriptive Statistics

- Histogram smoothing[102]
- Histogram thresholding

Figure 2.9: Transformations in the Table Recognition Literature. Transformations are classified based on whether they are applied to an image or text file containing a table (physical structure), to a description of table structure and/or content (logical structure), or to descriptive statistics.

transforms are applied to correct rotation and shearing in scanned images[1, 83, 84].

Other physical structure transformations include the image transformations that are often referred to as 'preprocessing': compression, resampling, binarization, and mathematical morphology. Resampling is used to provide low resolution versions of an input image, as researchers have found that this provides a useful alternate view of the document[50, 102]. Mathematical morphology[45] is concerned with set-theoretic operations defining how indexed sets (structuring elements) alter elements of other indexed sets. In the table recognition literature, morphological operations have been applied to both binary images and to text files[63]. Structuring elements used in table recognition are usually horizontal and vertical bars, used to close gaps. These types of structuring elements are used in the run-length smoothing algorithm (RLSA[120]), to thin objects[61], and to detect corners[112].

Logical structure transformations include tree and graph transformations, which have been used to merge and split regions (e.g. into tables[96]) and correct errors in table of contents entries[7]. Other logical structure transformations include filtering small objects assumed to be noise[68, 83, 103], producing shapes from point lists[3, 112], ordering and indexing objects, and translation to alternate representations (e.g. from HTML to plain text[39, 116]). Green and Krishnamoorthy[37] have provided an elegant translation from recognized table structure to a table's indexing relation using templates.

We quickly note a pair of transformations modifying descriptive statistics. Histogram smoothing[102] and thresholding have been used to reduce variance when trying to locate text lines and separators in projections.

Some of the transformations described in this section produce implicit inferences.

As an example, Handley[42] has pointed out that the morphological operations of the Run-Length Smoothing Algorithm[120] concurrently classify regions as foreground or background. Image binarization and noise filtering have the same side-effects. Many systems quietly assume that the regions output by these algorithms are valid foreground regions; a set of hypotheses about foreground regions are immediately accepted as valid.

2.5 Inferences

2.5.1 Classifiers, Segmenters, and Parsers

Inferences decide whether or how a table model can be fit to a document, through the generation and testing of hypotheses. More specifically, inferences decide whether physical and logical structures of the table model exist in a document using data observed from the input document, input parameters, table model, transformed observations, and table hypotheses (as shown in Figure 2.1). As seen in Figures 2.10, 2.11, and 2.12, a large variety of inferencing techniques have be used in table recognition. Comparing inferences, even for the same target structure, is often difficult because different observations or decision techniques are used.

In studying inferences in table recognition, we found the following categorization of techniques to be useful.

Classifiers: assign structure and relation types in the table model to data.

Segmenters: determine the existence and scope of a type of table model structure in data.

Decision Tree

- Single Dimension
  - Thresholding (e.g. threshold a 'columness' feature to locate columns[64])
  - Priority of separators (e.g. table lines by thickness[37])
  - Using area to classify noise vs. signal[18, 68, 83, 91]
  - Character class (e.g. alphabetic, non-alphabetic, other[82])
- Multiple Dimensions
  - Connected components
    - Defining[45]
    - Classifying[48, 61, 62, 66]
  - Document region classification[62, 66, 115]
  - C4.5 decision tree induction[94] (for table detection[82])
  - Word token sets[107]
  - Table/non-table classification[116]
  - Text orientation (vertical vs. horizontal[66])
  - Chain code line segment type[69]

Neural Network

- Optical character recognition[103]
- Logo recognition[15]

Nearest Neighbour

- k-nn (e.g. for defining clusters[84])
- Weighted k-nn (e.g. for table detection[116])

Syntactic

- String matching (e.g. HTML cell types[116])
- Regular expressions (e.g. assigning types to text lines[52, 82, 111])
- Part of speech tagging (e.g. to classify roles of words in tables of contents[7])

Statistical

- Bayesian Classifier ('Naive Bayes')
  - Table detection[116]
  - Functional class of text block (e.g. author, title for table of contents [106])
- Bayesian network (e.g. assigning labels to regions in tables of contents[106])
- Probabilistic relaxation[99] (assigning labels to words in tables of contents[10])

Figure 2.10: Classifiers: Inferences Used to Assign Structure and Relation Types. Classification techniques used in table recognition include decision tree, nearest neighbour, neural network, syntactic, and statistical methods. Decision trees are by far the most common technique.

Parsers: produce graphs on structures according to table syntax, defined in the table model.

This categorization separates the concepts of typing, locating, and relating structures (determined by classifiers, segmenters, and parsers, respectively). These inferencing classes are interdependent, however. The cyclic dependence between classifiers and segmenters has been well documented[14]. Parsers use classification and segmentation to define inputs for analysis, and in producing a parse (see Figure 2.12). In the other direction, parse results can be used to provide context in segmentation and

classification.

Space does not permit a detailed discussion of the inferencing methods in the literature. Instead we will briefly outline classifiers, segmenters, and parsers used in table recognition as summarized in Figures 2.10, 2.11, and 2.12.

Figure 2.10 separates classifiers into decision tree, nearest neighbour, neural network, syntactic, and statistical methods. We take a very general view of classification in which assigning any type to data is taken to be classification; this includes identifying an image region as being a connected component, for example. Decision trees are by far the most common classification method used in table recognition. An alternate organization for these classification methods is provided by the types they assign (which correspond to table model structures and relations). This type of organization can be seen in Figure 2.7, where classes of structures used in table models are listed.

Figure 2.11 summarizes segmenters, which search data for table model components using a binary classifier. The binary classifier tests the presence or absence of a table model component in a data region, while the objective function of the search controls the scope of segmented regions. As a simple example, consider a connected component segmenter; a simple classification defines connected components, while the objective function of the search ensures that only the largest connected components are actually segmented. Figure 2.11 categorizes segmentation in table recognition by whether methods cluster or partition data. Clustering has been used to set parameters adaptively using K-means clustering[124] and to cluster regions hierarchically based on distance[53]. We include closure on relations as a type of clustering, for example of a proximity relation[63]. Yoshida et al.[121] have presented a technique for clustering HTML tables in the world wide web, producing meta-tables.

Clustering

Connected components
- Creation (e.g. for adjacent pixels[45], for adjacent word boxes[63])
- Clustering connected components[62]

Tables by content[121]
K-means clustering (of projection histogram groups[124])
Hierarchical clustering of regions by distance[52]
Transitive closure (e.g. of a proximity relation [63])

Partitioning

Using breadth first search (e.g. to segment columns[53])
Using best-first search (e.g. to recover body, stub, and boxhead[59])
Table detection
- Using dynamic programming[52]
- Using best-first search[118]
- Using simplex[81] algorithm[17]
- Using iterative decreasing step[17]

Recursive Partitioning
- X-Y cut[80]: alternating horizontal and vertical partitions at projection histogram minima[37]
- Modified X-Y cuts, using histogram minima and lines[16]
- Recursive line partitioning (e.g. by 'best' separating line[66], by line separator priority[37])

Exact Matching
- Splitting text columns into rows at blank lines[63, 82]

Figure 2.11: Segmenters: Inferences Used to Locate Structures. Segmenters employ a binary classifier and a search function to locate table model components. Target regions matched by the classifier that also satisfy the objective function of the search are clustered or partitioned within the data set.

Some partitioning segmentations are very simple, such as when rows are segmented at blank lines in text files[63, 82]. Others are more sophisticated, such as segmenters used in table detection. An important differentiating feature in table detection methods is the type of search used. These have included best first search[118], dynamic programming[52], the simplex algorithm[17], and Cesarini et al.'s iterative decreasing step[17]. The most commonly used recursive partitioning methods in table recognition are variants of the X-Y cut algorithm of Nagy and Seth[80]. Recursive

partitioning methods actually produce a parse as well as segment regions, as the result describes a hierarchy of regions which may be used to determine the table grid and cell topology, for example[37].

Parsers used in the literature are summarized in Figure 2.12. Parsers produce graphs describing logical structure according to table syntax defined in a grammar. The grammars used in parsing are part of the table model. Parsers have been used to apply probabilistic constraints on region types[115], for table detection[27, 96], to define page grammars for X-Y cut-based analysis[68, 113], and to parse entry structure in tables of contents[107]. The parsing technique proposed by Takasu et al.[107] is interesting, because input tokens are initially provided with sets of possible types assigned by a decision tree which the parser then constrains to produce a valid interpretation. In the final interpretation, tokens are assigned a single type.

A small number of techniques for automatically inducing grammars from data have been described in the literature. Takasu et al.[108] have described a method for inducing a grammar for tables of contents from labelled data. Adaptive methods have been used to define regular expressions describing the data types of cell contents[87, 100], and a context-free grammar for table of contents entry structure[7].

2.5.2 Inference Sequencing

Table recognizers are frequently required to make inferences based on hypotheses produced by other inferences. For example, many systems will generate hypotheses of line locations. Inferring the location of line intersections needs to assume temporarily that these line hypotheses are valid. At a later point these line hypotheses may be found to be invalid by another inference. In many cases however, once hypotheses are

Hidden Markov Models

Maximizing region adjacency probabilities[115]

Attributed Context-Free Grammars

Tables in images (with parse control-flow directives: [26])
Table form box structure[3, 8, 119]
Form structure[27]
Using input tokens with multiple types to parse tables of contents[107]
For tables in HTML files[116]
For page segmentation[68, 113]

Graph Grammars

Table form cell composition[2]
Table structure from word boxes[96]

Figure 2.12: Parsers: Inferences Used to Relate Structures. Parsers produce graphs describing the logical structure of table model components. Here parsing techniques are categorized by the type of grammar encoding logical structure syntax: Hidden Markov models, attributed context-free grammars, and graph grammars. In the process of defining relational structure, parsers both segment and classify data. Consider the simple production rule $A \rightarrow BC$. Applying this rule in a parse clusters type 'B' and 'C' elements in the specified order to produce a type 'A' object.

---

accepted, they are not reconsidered. Rus and Subramanian have proposed an elegant graph-based notation for describing this type of architecture without feedback[100].

Tubbs and Embley[110] have proposed an alternate architecture for recognizing genealogical tables, making use of an ontology describing inheritance. The role of the ontology may be understood as a parameter of their table model (see Section 2.3). Input to the system describes cell locations and their text contents. Potentially valid hypotheses regarding cell topology, entry structure, and properties of cell contents defined using the ontology (e.g. dimension name and element relationships) are placed in matrices. An iterative control structure is then used to alter confidence weights

associated with each hypothesis using an ordered rule set. Iteration stops when a fixed point is reached for hypothesis weights, or after a fixed number of iterations. Decision rules then determine which hypotheses to accept, and how these accepted hypotheses are integrated into an interpretation of entry structure. In this scheme hypotheses dynamically affect one another through feedback.

Commonly in the literature an ordered sequence of classifiers and segmenters is used to infer instances of table model structures. As an example, consider a method in which table cells are segmented, and then columns of cells are segmented. This process can be considered a simple form of parsing, in which a hierarchy defining the composition of objects is defined. For the previous example, the composition of the column is described by the set of segmented cells. X-Y cutting[80] is another example, where the recursive segmentation of regions produces a tree.

One advantage of explicit table models is that their syntax may be encoded in a grammar. How the grammar is applied for table detection or structure recognition can then be controlled using different parsing algorithms, which result in different operation sequences. This ability to specify search strategy as a table recognizer parameter is useful both for characterizing and comparing methods.

2.6 Performance Evaluation

After the design of a table recognizer is complete, the designer will have some questions. How fast is the system? How well does the table model fit tables in the intended domain (e.g. images of technical articles, or HTML pages on the world wide web)? How reliable are the inferences made by the system, or put differently, what are the type and frequency of errors made by the system? In this section we describe the

methods and metrics used in performance evaluation of table recognition, which are then used to address the last two questions.

In order to train or evaluate a table recognizer the logical and/or physical structure of tables in documents must be encoded; this encoding is referred to as ground truth. It is produced using another table model, which must be able to describe the outputs of the table recognizer[51]. Ground truth may be produced manually with interactive tools[55, 95] or by automatically generating document sets with ground truth[72, 90, 117]. Automatic generation permits a degree of experimental control not possible with a random document sample, but then correspondence to real document sets must be addressed. Whether real or generated automatically, many tables have ambiguities, permitting multiple interpretations[52, 55]. An open problem is how, or whether, to encode ground truth for these documents[51, 73].

In the table recognition literature three methods have been used to separate available documents with ground truth into training and testing sets: resubstitution, in which all documents are used in training and testing; the 'circuit-training' or leave-one-out' method, in which each document is tested exactly once, with remaining documents used for training each time; or by randomly assigning documents to the training and testing sets. Resubstitution is seldom used as it produces positively biased results[45].

Comparing systems in the literature is difficult, because table models are usually not explicitly defined and differ significantly between systems, there are no available benchmark data sets with ground truth, and systems are seldom described in enough detail to permit replication. However, one trend appears to be that table recognizers

using detailed, narrowly-defined models to recover tables in well-characterized document sets appear to perform their intended tasks best (for a good example of this, see Shamillian et al.[103]). This is explained in part by substantial *a priori* knowledge permitting strong, well-grounded assumptions to be incorporated into the table model. This reduces the number of table model parameters necessary, making both training and understanding the behaviour of the table recognizer simpler.

In the remainder of this section we describe the performance metrics used in table detection and structure recognition, and mention some experimental design issues.

2.6.1 Recall and Precision

For classification, given a class X and a set of input patterns with associated ground-truth, recall is the percentage of type X input patterns classified correctly, and precision is the percentage of input patterns assigned type X correctly. Over the set of all possible classes, recall and precision are equivalent. For individual classes, the number of correct class X classifications need not match the number of items correctly assigned class X. To see this, consider the case where a binary classifier (returning X or Y) always assigns class X.

For evaluating the segmentation of regions (e.g. for table detection), the splitting and merging of ground truth regions needs to be taken into account[55, 72]. To accommodate this, modified recall and precision metrics that make use of the area of overlap between detected and ground truth regions have been devised[17, 64]. The area of overlap between regions in recognizer output and regions in ground truth are then used as weights in modified recall and precision metrics that take merging and splitting into account.

Precision and recall are sometimes combined into a single 'F-measure.' In the table recognition literature this is non-standard, as both the arithmetic and harmonic mean of precision and recall have been called an 'F-measure'[39, 110]. The harmonic mean of precision (P) and recall (R) is defined as:

$$H(R, P) = \frac{2RP}{R + P}$$

The two metrics have different biases. For example, given the sum of recall and precision $T = R + P$, the maximum harmonic mean value is obtained when $R = P = T/2$. In contrast, for the arithmetic mean the resulting value depends only on the sum of recall and precision, and not on their relative sizes.

2.6.2 Edit Distance and Graph Probing

Edit distance (see Section 2.3) has been used to compare detected tables to ground-truth (taking false positives, false negatives, merges, and splits into account[52]). It has also been used in table structure recognition to compare graphs describing the logical structure of tables[76]. Edit distance has some associated problems. The minimal editing sequence is not necessarily unique. Edit distance can be computationally intensive to determine. Finally, appropriately setting the weights of editing operations is difficult: in the literature operation weights are always set to one.

To address these problems a new metric called table agreement has been proposed[55]. Agreement is computed by automatically generating queries about the number, content, topology, and indexing structure of cells in logical table structure encodings for recognizer output and ground truth. Queries are verified or contradicted by

searching the other table encoding; agreement is defined as the percentage of verified queries. This process of generating and verifying queries from graphs is called graph probing[76].

2.6.3 Experimental Design

Evaluation in table recognition is still maturing. Performance evaluations are usually made from final outputs, ignoring individual inferences. As a result, the effects of individual decisions are often confounded (inseparable) in the evaluation. Wang and Hu have made a step forward in this regard. They performed an experiment making use of a fixed classification method (an induced decision tree) while varying the set of observations used for training the classifier[116]. In their design, the inference method and observations are clearly separated into independent factors.

A number of other experimental design issues remain in the area. These include a need for better sampling[73] and comparisons of experimental conditions[77]. A test worth considering for the robust comparison of conditions is the Analysis of Variance (ANOVA[46]).

2.7 Conclusion

We have presented the table recognition literature from the viewpoint that table recognizers may be understood as sequences of decisions (inferences) supported by observations and transformations of available data. These decisions are made relative to a table model describing the location and composition of tables in a set of documents. Figure 2.7 summarizes the structures used in table models for recognition.

Observations, transformations, and inferences made in the table recognition literature are summarized in Figures 2.8, 2.9, 2.10, 2.11, and 2.12. In Section 2.6 we describe performance evaluation methods for determining the sufficiency of a table recognizer for recovering and/or analyzing tables in a set of documents. We have tried in our discussion to point out various assumptions inherent or implied in different operations and sequences of operations in the literature.

As pointed out in Section 2.6, it appears at present that simple, domain-specific table models have more promise than complex, general models. It may be worth studying whether combinations of table recognizers with simple models yield improvements in performance over the current state-of-the-art for large, heterogeneous document sets such as technical articles.

Other avenues for future work include extending the proposed table models of Wang[114] and others, further exploring content-based methods for recognition (including cell cohesion metrics and the use of domain knowledge), improving experimental design and evaluation techniques at both the level of individual decisions and whole systems, defining corpora of ground-truth tables for use in the community, and exploring new observations, transformations, and inferences for use in table recognition.

Chapter 3

A Functional Language for Recognition Strategies

In this chapter we introduce the Recognition Strategy Language (RSL), a functional 'glue' language for combining inference functions to create table recognition systems. We characterize table recognition problems as simple imitation games in Section 3.2. We then introduce *interpretation trees*, which are used to formalize the inference sequences of RSL specifications (called *strategies*) in Section 3.3. Various aspects of RSL are summarized in terms of the table models, observations, transformations, and inferences expressible within RSL. The operations of RSL are defined in Section 3.9, and then the chapter closes with a brief summary in Section 3.11.

3.1 Motivation

In Section 2.1 we posed a number of questions related to understanding the decisions made by a table recognition system. Reordered for the current discussion, these were:

- What decisions are made?
- What inferencing techniques make decisions?
- On what data are decisions based?
- What is assumed when decisions are made?

In the last chapter we addressed these questions generally, to provide a sense of the current state-of-the-art in table recognition research. We described how decisions about table location and structure in data (inferences) are supported by observations and transformations. We listed and summarized a number of the inferencing techniques that have been used, categorized into classifiers, segmenters, and parsers. The observations and transformations that produce data from which decisions are made were also summarized. Finally, some of the implicit and explicit assumptions used by table recognition systems were indicated in the discussion.

In this chapter we turn our attention from table recognition in general to the design, construction, and comparison of individual table recognition systems. Currently, there is a substantial amount of effort involved in building a table recognition system; in general, these systems are built from scratch in a general-purpose programming language. Significant subsystems must be constructed before analytical functions (observations, transformations, and inferences) may be built. These subsystems include the data structures for recognized objects and intermediate observations, the control flow architecture, and geometric libraries (e.g. to support bounding box and line intersection analysis). Our experience in building previous document recognition prototypes is that the time spent on these preliminaries tends to dwarf the time spent implementing analytical functions when building recognition systems in this manner.

Table recognition system designers would benefit from even small amounts of tool support, to provide them with more time to design and revise the analytical parts of their systems. Ideally such tools would also assist them in addressing the questions above, as these are primary concerns when a designer is creating, analyzing, debugging, revising, or comparing recognition algorithms.

To address this need, we have designed the Recognition Strategy Language (RSL). RSL is a simple functional 'glue' language for quickly combining and sequencing inferencing functions. This approach is partly inspired by John Ousterhout's Tcl language[85] for rapidly combining components implemented in various languages and frameworks. Among other differences, Tcl is a procedural, typeless language (all data is text-typed), while RSL is a functional, typed language with a small set of types: parameters can be number or string literal-valued, and a small number of parameterizable structured text record types are used to describe decision results. Our first implementation of RSL was produced by translating RSL programs into the functional language TXL[22, 25]; we describe this translation in the next chapter.

RSL implementations automatically maintain sets of hypotheses for the designer, including histories of hypothesis creation and revision, and of changes caused by individual inferencing functions. In addition to saving the designer effort, this solves the problem of confounded inferences cited in Section 2.6.3: one can always determine the effects of individual inferences. Statistical confidence values and multiple inferencing results are also accommodated and managed by RSL.

The syntax of RSL captures the composition of and dependencies between table model component types. This helps address the problem of implicit table models

raised in the last chapter. The resulting model descriptions are type and parameter-based. For example, the RSL syntax can capture that columns are made of cells, and depend on vertical lines and certain parameters when constructed, but not that only the leftmost column can be labelled as a table stub. An advantage of having such loosely defined models is that they can be easily modified without serious concerns about model consistency. This flexibility is particularly helpful in the early stages of designing a system. There are of course substantial advantages to having more explicit models, as discussed in the previous chapter.

RSL is intended for describing the sequential feed-forward architectures common in the literature, as well as feed-forward architectures that use inferences producing multiple results. In the remainder of this chapter we describe the game-based motivation of RSL, the RSL syntax and informal semantics, and the table models, observations, transformations, and inferences used in RSL programs.

3.2 Table Recognition as Imitation Games

In creating a table recognizer, at some point a designer needs to assess how well their system interprets tables, in other words, to perform an evaluation. A number of important questions are then raised. Which set of documents, or parts of documents, is the system intended for? How should members of this set be sampled to produce test sets for evaluation? How does one determine the 'correct set' of interpretations? It has been pointed out previously that this last question, which we will call the 'ground truth' question, is a hard one; it is difficult, perhaps even impossible for universally 'correct' sets to even be defined[51].

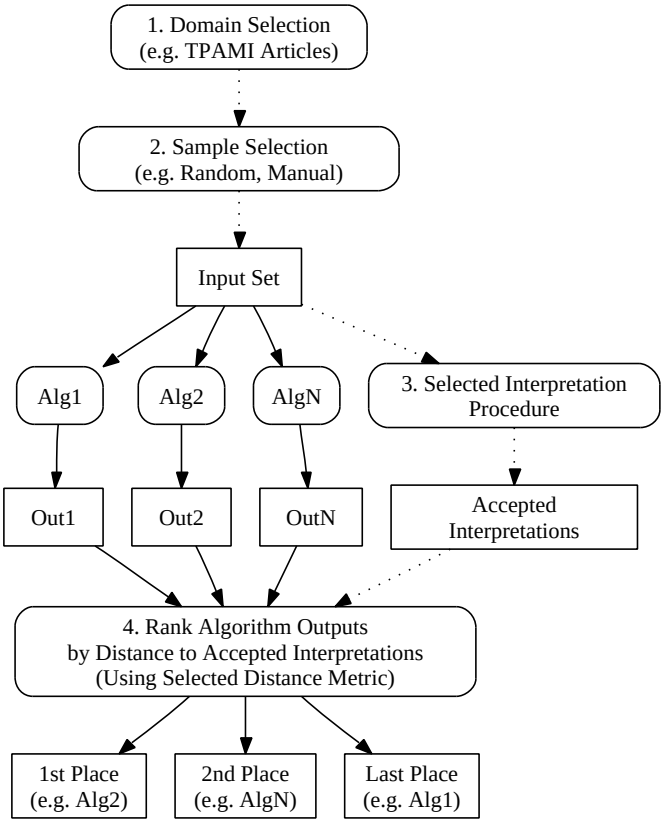


Figure 3.1: Table Recognition as Imitation Games. A game is defined by four elements: a domain, a sampling method, an interpretation procedure used to generate the set of accepted interpretations, and a distance metric used for ranking recognition results. The goal of the game is to produce the algorithm whose output most closely matches the set of accepted interpretations, according to the distance metric.

This state of affairs is not that surprising when one considers that as with natural language, tables are adapted to the individual needs of the persons using them, and so there is a wide variation in the types, styles, and methods of interpretation for tables. We end up in a situation where to know whether a table recognition system interprets an input correctly depends upon *who we are asking*. We propose that the best empirical measure of a table recognizer's performance is one where 'who we are asking' is known, and where results of such evaluations are parameterized by this person, persons, or algorithm(s). Taking things a step further, if we view evaluation as part of and not subsequent to recognizer design, then table recognition algorithms can be seen as strategies for a class of imitation games.

In a table recognition imitation game, designers try to maximize the 'placing' of their algorithm against a set of other algorithms by best imitating the output of a selected interpretation procedure (e.g. that used by a person defining 'ground truth', or a chosen algorithm). In a 'solitaire' game examining only a single algorithm, designers simply try to match the outputs of the interpretation process they are trying to imitate as closely as possible. This class of games is illustrated for the multiple algorithm case in Figure 3.1.

A 'table recognition imitation game' is specified by four things: a domain defining the set from which inputs used in the game are taken, a sampling method for drawing the input set from the domain, an interpretation procedure used to define the set of accepted interpretations in the game, and a distance metric for comparing the outputs of algorithms to the accepted interpretation set. Algorithms are ranked by the similarity of their outputs to the accepted interpretation set as determined by a distance metric (e.g. some combination of recall and precision).

Table recognition strategies in the game may use very different table models reflecting their different approaches, but for the game to be fair the algorithms must agree on the types and relations on regions to be compared by the distance metric. As an example, for table structure recognition these common subsets of the models might include cells containing words, rows and columns containing cells, and a binary indexing relation defined on cells. Clearly, it is in the best interest of a participant in a formal game (such as at a conference) to make their table models include or accommodate this 'evaluation' model, in order for their outputs to be properly assessed by the distance metric.

In designing table recognition algorithms informally, for example without a particular document set or distance metric in mind, one might say that a designer plays a series of games of 'imitation solitaire,' trying to best imitate their own model(s) of table location and structure for some set of examples. Some intuitive sense of the distance between the algorithm outputs and what they think the output 'should be' is used to evaluate results. In this situation, the designer both designs the algorithm and produces the accepted interpretations, and so consciously or not, is trying to imitate his or her self.

It is also worth noting in this game view that heuristics become acceptable or even preferable if they produce better results than more general techniques. From this vantage point, heuristics are tactical assumptions, simplifications intended to reduce the space of possible model instances while maximizing a score within a particular game. Whether a heuristic generalizes is only of interest if one wants or needs to apply the algorithm to another game of a different construction (e.g. a different domain, interpretation process, or evaluation metric). This may partly explain why

procedural methods employing heuristics are so common in the literature; designers appear generally to have a particular imitation game, or class of imitation games in mind. As pointed out earlier, the 'general case' game is hard to even properly define, so simplifying assumptions of some sort or other seem unavoidable.

The Recognition Strategy Language makes the executable descriptions of these imitation game strategies more abstract than implementations in a conventional programming language. In RSL, strategies are defined in terms of a set of legal 'moves,' which are types of decisions. For a given decision type the process used to determine the decision outcome can be arbitrary; for example, the result of classifying cells as data or headers could be returned from a neural network, nearest neighbour classifier, grammar-based classifier, or an interface could be used to collect classification results from a human operator (e.g. a 'ground truther'). By fixing a set of decision types in RSL, we can more uniformly and abstractly describe the types and results of inferences in a strategy.

Decisions in RSL strategies may be compared between strategies and the results of these decisions recorded for later analyzing and comparing strategies, as we shall see. The set of decision types in RSL include region classification, region segmentation, defining binary relations on regions, and rejecting hypotheses. Additional 'moves' accept and reject sets of hypotheses (*interpretations*), and control the application of other 'moves' (a simple conditional construct). RSL operations are described in detail in Section 3.9, and summarized in Appendix A.

3.3 Interpretation Trees

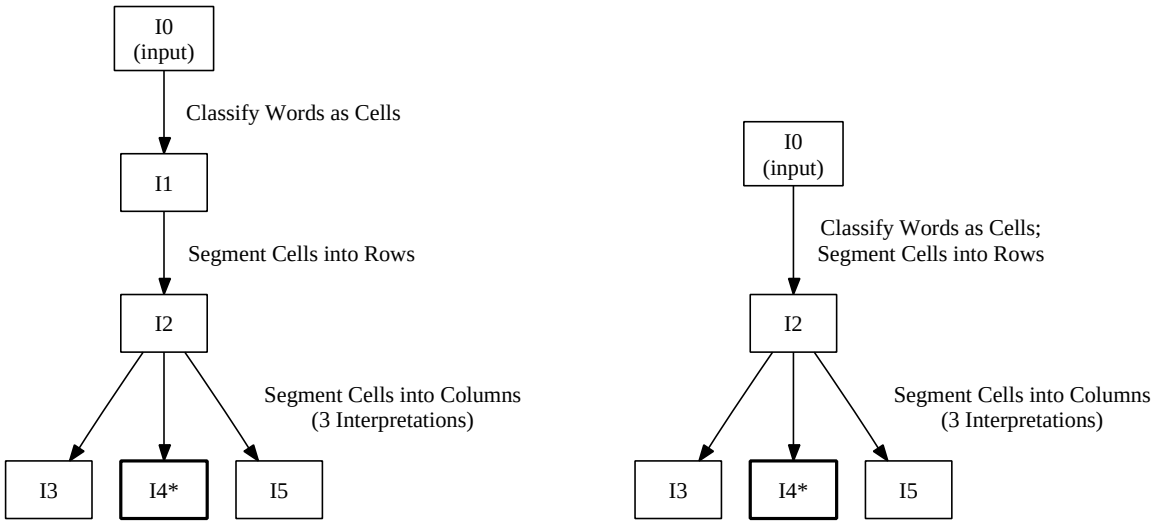
If as in RSL we formalize the possible decisions about table location or structure, we can characterize the operations carried out by a table recognition strategy using a tree. This *interpretation tree* specifies the sequence of inferences made during recognition, branching whenever alternate interpretations exist, such as when alternate classifications of a region (e.g. as header or data cell) are considered plausible.

Each node in an interpretation tree represents an *interpretation*: hypothesized regions and relations on regions such as cell topology and indexing structure that have been inferred from the input. Some nodes in an interpretation tree may be distinguished as accepted nodes, whose associated interpretations are returned as output. All remaining interpretations are considered to be rejected by the strategy. If no nodes are marked as accepted when a strategy completes, the strategy is considered to have rejected the input data.

Interpretation trees may be described in extensive (complete) and normal (reduced) form (see Figure 3.2). An extensive hypothesis tree represents all intermediate interpretations. A normalized hypothesis tree reduces all sequences of nodes in which no branching occurs to a single node. The normalized tree nodes compactly represent interpretation states before branching occurs.

Interpretation trees represent a possible-worlds view of recognition outcomes[34], where at any given time a set of interpretations may be considered plausible, with each interpretation or 'world' comprised of a different set of accepted hypotheses. We will often refer to the set of plausible interpretations at any one time as the *candidate interpretations*.

RSL currently describes recognition strategies that construct interpretation trees



a. Extensive Form b. Normal Form

Figure 3.2: Interpretation Trees. Nodes represent *interpretations*, which describe hypothesized regions and relations on regions such as cell topology or indexing structure. Edges in the trees edges represent inferencing operations that decide table model locations and structure. Extensive trees (a) provide separate nodes for all intermediate interpretations, while trees in normal form (b) represent non-branching paths in the tree using the final node in the path, collapsing the inference sequence into a single edge. The set of nodes at any depth in the tree are called the *candidate interpretations* at that point in the analysis. The asterisked nodes (I4*) are *accepted interpretations* that will be returned as output.

breadth-first. Starting with the input data as the first candidate interpretation, inference functions are applied separately to each candidate interpretation, generating one or more new candidate interpretations from each (and removing the input interpretations from the set of candidate regions). Note that an inference may have a result in which there is no effect, leaving the hypotheses of the candidate interpretation unaltered. At each step in the strategy, an operation is applied to all candidate interpretations, thereby building the tree breadth-first.

In RSL, accepting an interpretation makes it a leaf in the interpretation tree, removing that interpretation from further analysis. This reflects a common practice in the table recognition literature, where once an acceptable interpretation is produced, analysis stops; most commonly the strategy terminates when a single acceptable interpretation is produced, as indicated in the previous chapter.

There is of course no reason why a recognition strategy could not build an interpretation tree depth-first, or in some other order, possibly even revisiting nodes and changing their acceptance state. It is also conceivable to have a strategy that accepts an intermediate interpretation and then continues to alter the interpretation, possibly to produce another accepted interpretation. For example, a coarse-to-fine recognition strategy might propose a set of rows, which is later split producing more rows, returning both as accepted interpretations. As these are uncommon in the table recognition literature, for the time being these possibilities are not expressible in RSL.

3.4 RSL Strategies

RSL programs, called *strategies*, describe the sequential application of inference and supporting functions, transforming a single input graph to one or more output graphs representing the accepted interpretations produced by a recognition strategy. As mentioned in the previous section, the strategies expressible in RSL produce interpretation trees breadth-first (i.e. in a feed-forward fashion). We felt it was natural to create RSL as a functional language, representing decision sequences using function composition. For a concise motivation for using functional programming in computer vision tasks in general, see Breuel[12].

Within RSL itself, only two types of data are defined and manipulated: the values of adaptive parameters, and the interpretation tree. For convenience, we will often distinguish the sets of candidate and accepted interpretations within the interpretation tree, as these are the data observed and transformed by inferencing functions in RSL. At present inferences are permitted to observe only accepted hypotheses when making decisions. We will discuss this further in Section 3.7.

There are no local variables or constants in RSL. This approach was taken in order to make the definition and adaptation of recognition parameters easy to locate. By defining all recognition parameters in one place, designers can determine quickly what the number, names, and types of strategy parameters are. At present, RSL parameters may be number or string literal-valued.

All functions within RSL, which we term *strategy functions*, transform the interpretation tree. Strategy functions describe a sequence of inferencing, parameter adaption, conditional, strategy function call, and output ('write' and 'print') statements. To prevent side-effects, all parameter adaptations have only local scope, and have an effect only within the associated strategy function and any sub-functions. The signature of all strategy functions is:

$$(I, P) \rightarrow I'$$

where I and I' are interpretation trees, and P is the set of parameters passed from the original declaration (for the 'main' strategy function) or from a calling strategy function (which may have altered adaptive parameters before the call). To reduce the verbosity of RSL, parameters of strategy functions are always implicit. An example of the strategy syntax can be seen in Figure 3.3 (from 'strategy' to 'end strategy').

As is common in many programming languages, execution begins with the strategy function named 'main' in an RSL specification.

There is only one conditional construct in RSL strategies, the *for interpretations* command. If present, this is the first operation in an RSL strategy function. It is a conditional test applied to each of the candidate interpretations, to prevent applying the function to some or all of the candidate interpretations (i.e. a guard). The set of candidate interpretations is determined using an external function, defined outside of an RSL strategy. We will see examples of this operation in Chapter 5, where it is used for a single test and to define the termination point of a recursive strategy function.

As mentioned earlier, there are no local variables or constants in RSL; further, there are no commands to directly manipulate the interpretations themselves. Results of inferencing and other operations are expressed using text records, which the RSL core library then uses to update the interpretation tree and interpretations appropriately. This substantially reduces book-keeping overhead and simplifies expressing inference functions for a strategy designer.

External functions used in RSL regularly need to observe the contents of current interpretations, such as to find out how many cells exist, and so a library of graph observing functions must be available for use within the external functions of an RSL strategy. Our experience has been that this library is a bottleneck in the system: a graph-observing library with only low-level operations makes expressing and implementing algorithms unnecessarily difficult.

```
model regions
    Image Word Cell Row Column
end regions

model relations
    % 'contains' relation type defined by default.
    adjacent_right adjacent_below
end relations

recognition parameters
    sMaxRowSeparation          2 % millimetres
    sMaxColumnSeparation       2 % millimetres
    aResolution              300 % dpi; default.
    sMaxIterations          1000
end parameters

strategy main
    adapt aResolution using       % get scan resolution from
        getScanResolution()       % the input image attributes.
        observing
            {Image} regions

    classify {Word} regions as {Cell}       % classify all words as cells.

    relate {Cell} regions with {adjacent_right} using       % define right adjacency
        defineRightAdjacency(sMaxRowSeparation, aResolution) % between cells

    segment {Cell} regions into {Row} regions using  % segment cells into rows,
        mergeRowsFromCells()                         % observing cells and adjacent_right
        observing                                    % edges in interpretation graphs
            {adjacent_right} relations

    relate {Cell} regions with {adjacent_below} using     % define lower adjacency
        defineLowerAdjacency(sMaxColSeparation, aResolution) % between cells

    segment {Cell} regions into {Column} regions using % segment cells into columns,
        mergeColumnsFromCells(sMaxIterations)      % observing cells and adjacent_below
        observing                                  % edges in interpretation graphs
            {adjacent_below} relations

    accept interpretations              % accept all candidate interpretations
end strategy
```

Figure 3.3: Simple RSL Strategy for Table Structure Recognition. Comments are indicated using the '%' symbol.

3.4.1 A Simple Example

We now demonstrate how RSL is used to 'glue' inferencing functions using the simple table structure recognition algorithm shown in Figure 3.3. This example contains the four main components of an RSL strategy: the model region type list (lines 1-3), the model relation type list (lines 5-8), the list of recognition parameters (lines 10-15), and the main strategy function (lines 17 to the end). Calls to external functions for inferences and other operations follow the keyword 'using' in RSL (found on lines 19, 26, 29, 34, and 37).

In RSL, hypotheses are represented using directed graphs with attributes; the model regions list defines the legal set of types that may be associated with a physical region, represented as nodes of a graph. The model relation type list defines the edge types (equivalently, edge labels) that may be used to relate nodes. One relation is pre-defined: the containment, or 'contains' relation, which describes how regions are composed of other regions. For example, a column that contains three cells would be represented as a single node with associated type attribute 'column', the cells as three nodes with associated type attribute 'cell,' and the 'contains' relation would include edges from the column node to each of the cell nodes.

Recognition parameters are static (constant) or adaptive (variable) in RSL. We enforce a very simple Hungarian naming convention[105] for parameters in RSL, with static variables beginning with lower case 's' and adaptive variables beginning with lower case 'a.' All recognition parameters must be defined in this section, as there are no local variables or constants in RSL. This helps insure that parameters of a table recognition system may be easily located and altered.

For this example, the input is a graph containing a node of type 'image,' and a

number of 'word' nodes representing the bounding box location of words in the image. The main function contains seven operations. The first operation (lines 18-21) alters the adaptive parameter *aResolution* to obtain the scanning resolution from attributes of an image represented as a node in the input. Note that this operation makes use of the external function *getScanResolution()*.

Lines 23-39 use three of the basic inferencing operations in RSL: classifying, segmenting, and relating regions. First, all words are classified as cells (line 23). Then cell regions are related using a relation named *adjacent_right*, to represent whether a cell has another cell adjacent to the right of itself. The relation is defined using the external function *defineRightAdjacency()*, which is passed two parameters (lines 25-26). The first parameter represents a distance in millimeters, and the second is our adapted parameter representing the scanning resolution in the input image. The adapted parameter will be used to convert this distance appropriately to a number of pixels. Cell regions are then segmented into rows, using the previously computed *adjacent_right* relation (lines 28-31).

We next perform a similar analysis for columns, this time defining a relation describing the closest cell adjacent below a cell (lines 33-34), followed by another segmentation, this time segmenting cells into column regions referring to the just-defined *adjacent_below* relation (lines 36-39). The *sMaxIterations* parameter specifies the maximum number of iterations for optimizing the column segmentation.

The final operation in the strategy on line 41 indicates RSL should accept all interpretations produced by the previous operation, in this case the column segmentation. As we will describe in more detail later, RSL inferencing operations may produce multiple results. For example, assume that before applying the column segmentation

operation (lines 36-39) we have one interpretation under consideration, and after applying the segmentation operation we now have two possible interpretations. *accept interpretations* as given on line 41 would accept both of these interpretations.

3.5 Table Models in RSL

RSL table models are unique in that they are used to record the history of hypothesis creation and revision directly within interpretations (model instances). Both accepted and rejected interpretations created by an RSL strategy contain their entire construction history, along with an indication of which hypotheses are accepted at the end of analysis. Models are families of directed graphs with attributes that describe the physical location and logical types of regions, the relational containment structure of regions (the 'contains' relation), and additional logical binary relations on regions. The 'model regions' and 'model relations' sections of an RSL specification define the sets of legal region and relation logical types for interpretation graphs.

We note here that the graph-based table models used in RSL resemble that used by Hu et al.[53, 55]. RSL's models are more general, encoding relations in addition to region containment, and using additional node and edge attributes to encode hypothesis histories, but the region structure scheme is identical. We agree with those authors' observation that not strictly enforcing the logical labelling of regions relative to one another is useful: particularly in the design stage when models are first being developed.

Figures 3.5 and 3.6 demonstrate how physical and logical structure are represented by interpretation graphs (table model instances) in RSL. In the rest of this document

```
strategy main
    relate {Word} regions with {below} using
        findLowerAdjacentWords()

    segment {Word} regions into {Column} regions using
        segmentVerticalWordGroupsAsColumns()
        observing
            {below} relations

    reject {Column} classifications using
        rejectSingleCellColumns()

    accept interpretations
end strategy
```

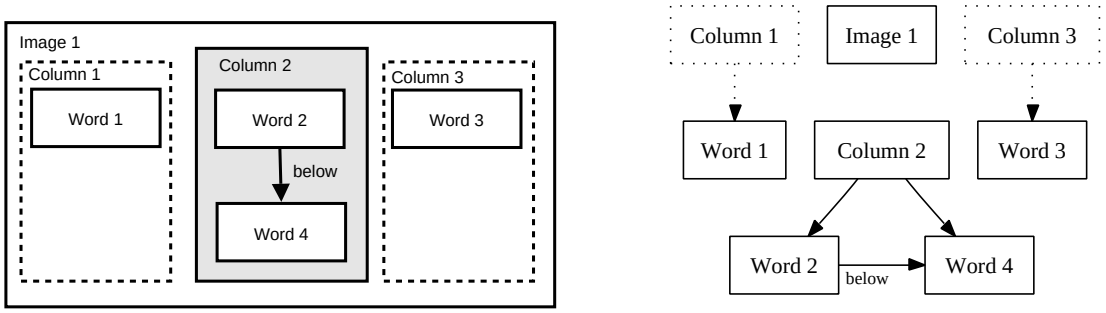
Figure 3.4: RSL Strategy for Segmenting Words into Columns. For brevity, the model regions, model relations, and recognition parameter sections are not shown. A table model instance (*interpretation*) produced by this strategy is shown as a graph in Figure 3.5 and as a text file graph encoding in Figure 3.6.

---

we will often refer to *interpretations* and *interpretation graphs* interchangeably. In the remainder of this Section we summarize the representation of physical and logical structure in RSL table models, and hypothesis histories.

3.5.1 Physical Structure: Region Geometry

The locations of regions in RSL may have two geometric shapes: polylines (defined by a set of two or more points), and bounding boxes. Within an RSL interpretation, nodes of an interpretation graph (model instance) represent physical regions that may be associated with one or more logical types. Where possible, RSL maintains the physical locations of regions automatically. For example, if we segment words into a cell, RSL automatically computes the bounding box of the words contained by the cell and assigns this as an attribute of the new cell region node in the interpretation



a. Region Structure and Below Relation

b. Interpretation Graph

Figure 3.5: An Interpretation Graph Constructed by the Strategy in Figure 3.4. Regions and relations on regions are represented in the interpretation graph on the right. *contains* edges are represented as unlabeled, solid arrows on the right. Rejected *contains* edges and *column* regions are shown using dotted boxes and lines. A textual representation of the graph in (b) is shown in Figure 3.6. The *Image* and *Word* regions are provided as input.

graph.

3.5.2 Logical Structure: Region Types and Relations

Logical structure is defined in RSL by assigning logical types to regions, and through binary relations defined on regions. Logical types are assigned to regions by inferences that create or classify regions (the *create, replace, and classify* operations described in Section 3.9.2). An existing region which is no longer assigned any of these types (e.g. after a rejection of region type) is assigned the default 'REGION' type by RSL, indicating that the region has a physical location, but no logical type. Legal region types are defined in the 'model regions' section of an RSL strategy.

As mentioned earlier, the region containment, or 'contains' relation is always defined, and is intended to be manipulated only indirectly using region segmentation

```
N [Image]
    Image1 {BB 0 0 2500 1000, File "tableRegion.tif", Resolution "300",
        Input, Active "yes"}
N [Word]
    Word1  {BB 200 300 400 400, Input, Active "yes"}
    Word2  {BB 500 300 700 400, Input, Active "yes"}
    Word3  {BB 800 300 1000 400, Input, Active "yes"}
    Word4  {BB 500 700 700 800, Input, Active "yes"}
N [Column]
    Column1 {BB 200 300 400 400, Segment "Inf 2: 0.4", Reject "Inf 3: none",
        Active "no"}
    Column2 {BB 500 300 700 800, Segment "Inf 2: 0.9", Active "yes"}
    Column3 {BB 800 300 1000 400, Segment "Inf 2: 0.6", Reject "Inf 3: none",
        Active "no"}
N [REGION]
    Column1 {BB 200 300 400 400, Reject "Inf 3: none", Active "yes"}
    Column3 {BB 800 300 1000 400, Reject "Inf 3: none", Active "yes"}
E [contains]
    (Column1, Word1) {Relate "Inf 2: 0.4", Reject "Inf 3: none", Active "no"}
    (Column2, Word2) {Relate "Inf 2: 0.9", Active "yes"}
    (Column2, Word4) {Relate "Inf 2: 0.9", Active "yes"}
    (Column3, Word3) {Relate "Inf 2: 0.6", Reject "Inf 3: none", Active "no"}
E [below]
    (Word2, Word4)   {Relate "Inf 1: none", Active "yes"}
```

Figure 3.6: RSL Text Encoding of Interpretation Graph in Figure 3.5b. Logical region types have the format *'N [region type]'*, and logical relation types have the format *'E [relation type]'*. Attributes represent physical locations as bounding boxes (*BB*), whether a hypothesis is accepted (*Active*) or was given as input (*Input*), and hypothesis histories. Histories follow the attribute labels *Segment, Relate,* and *Reject*, with 'inference time' and confidence values given. The inference times in this interpretation graph (e.g. 'Inf 2' represents 'time 2') correspond to the sequence of inferences shown in Figure 3.4.

operations (see Section 3.9.3). The *relate* operation may be used to define relations of other types, for example to represent indexing structure, region adjacency, the physical proximity of regions (e.g. 'close' rows), or any other relationships felt to be useful by the designer for analysis. Each relation has a type and an associated set of edges. The set of legal relation types are defined in the 'model relations' section of an RSL strategy.

3.5.3 Hypothesis History

There are a small number of ways to construct hypotheses in RSL. They include creating a region with a logical type, associating a logical type with a physical region, adding edges to a logical relation, or rejecting any of the previous hypothesis types. To solve the problem of confounded inferences in recognition outputs cited in Section 2.6.3, RSL automatically records when individual hypotheses are created, rejected, and re-instantiated in an interpretation. As an example of re-instantiation, consider a case where a region is labelled as cell, this hypothesis is then rejected, and then later the same region is again labelled as a cell.

A *hypothesis history* is created for each hypothesis in an interpretation graph using annotations (see Figure 3.6). RSL records time stamps for hypotheses using the number of strategy inferences previously applied. Currently all inputs to RSL are treated as givens, a hypothesis set accepted from time '0' (in input), until a strategy completes. The first inference operation in a strategy is applied at time '1,' and so on. Note that this 'time stamp' is equivalent to the depth of an interpretation within an interpretation tree. To ease understanding of interpretation graphs, time stamps are associated with a text label representing the type of inference that created or revised

a hypothesis.

The annotated hypothesis history in an interpretation graph provides a compact representation of a single path in an interpretation tree. From an interpretation graph with an annotated hypothesis history, we can easily determine which hypotheses existed and were active along a path in the interpretation tree at any 'inference time' from the input to the present. In addition, for each hypothesis, associated inferencing operations and results may be looked up in the interpretation tree output of RSL to find the operation performed, number of inference results, and the details of each result. This assists greatly with debugging strategies and analyzing their output. As an example, it is now possible to determine whether a cell region not present in the final output existed at an earlier point in an interpretation's construction.

An interpretation graph represents the set of accepted and rejected hypotheses at the final 'inference time' using a simple binary-valued attribute named 'Active' (see Figure 3.6).

3.6 Inferences in RSL

In RSL, inferences are defined by the type of hypothesis they produce, and the book-keeping operations required to update the interpretation tree appropriately. For example, region classification is distinct from 'general' region *segmentation*, which is distinct from region *merging*; merging and segmentation involve different book-keeping transformations (see Section 3.9.3). The *procedure* used for decision making may be arbitrary. To support this, external inferencing functions are a central part of RSL. Inferencing functions only need to return inference results of the appropriate type; it is the language library itself, or *RSL Core*, which manages the updating of the

interpretation tree and annotating of interpretations (see Figure 3.7).

RSL inferences are always applied to each candidate interpretation individually. For example, if there are three candidate interpretations available when a classification operation is called, the operation is applied individually to each of the candidate interpretations, producing one or more interpretations per candidate interpretation as output. Proceeding in this way, inferences in an RSL strategy produce an interpretation tree breadth-first.

In the remainder of this Section, the roles of external functions and confidences in RSL Inferences are described in more detail. Descriptions of the inferencing commands of RSL may be found in Sections 3.9.2 to 3.9.6.

3.6.1 External Functions

One of the observations made in Chapter 2 was that a large variety of inferencing methods have been used in table recognition, and more might be applied to the problem. As a result, we wished in designing RSL to allow as many decision techniques as possible. With this goal in mind, we made external functions that return structured text results a central part of the RSL language. Each inferencing operation in RSL has an associated structured text format used to describe the results of such an operation. This approach was inspired partly by the Tcl language[85], where all functions return a single type (text) to permit results from various applications to be combined. The structured result formats used for RSL inferences allow decisions from significantly different techniques to be described concisely and uniformly.

Each candidate interpretation graph and the text of the associated RSL operation itself are implicit arguments of all external functions in RSL. To illustrate this,

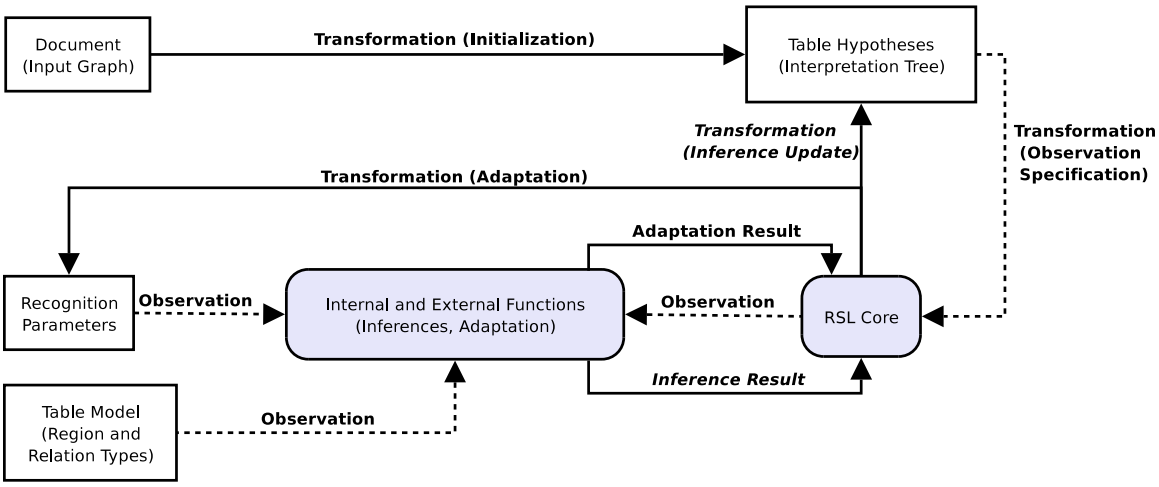


Figure 3.7: RSL Recognition Process. This graph is adapted from the table recognition process illustrated in Figure 2.1. In RSL an input document is used only once, to define the initial interpretation of the interpretation tree. The *RSL Core* library transforms the interpretation tree based on inference results, and transforms observations of current interpretations to enforce observation specifications (see Section 3.7). The RSL Core also updates recognition parameters based on adaptation results. Observations and transformations used within external functions are not shown, because they are not visible within RSL.

consider the following RSL excerpt:

```
classify { Block } regions as { Column } regions using
    classifyBlocksAsColumns(sThreshold)
```

In this example, we can see that the only explicit parameter passed to the external function *classifyBlocksAsColumns()* is sThreshold, a static parameter defined in the recognition parameters section of the strategy (see Figure 3.3 for an example of a recognition parameters section).

In the current implementation, the TXL function signature for the external function in the above example might look like:

```
function classifyBlocksAsColumns Threshold[number]
    RSL_Op[strategy_op] Graph[graph]
```

where the arguments are a numeric threshold (given by sThreshold in the previous example), the RSL command (the text of the previous example), and an interpretation graph.

The purpose of passing the RSL operation text to external functions is to allow libraries to be written for operating on multiple region and/or relation types. For example, suppose we define a single external function to segment regions bounded by lines, called *mergeCellsWithSameAdjacentLines()*. By passing the RSL operation text, we can then use this function multiple places, as in the following example:

```
segment { Cell } regions into { Column } regions using
    mergeCellsWithSameAdjacentLines()
    observing
        { Vertical_line } regions
        { adjacent_left, adjacent_right } relations

...

segment { Cell } regions into { Row } regions using
    mergeCellsWithSameAdjacentLines()
    observing
        { Horizontal_line } regions
        { adjacent_top, adjacent_bottom } relations
```

In the first RSL operation, the external function is used to segment cells bounded by vertical lines on the left and right into columns, and in the second RSL operation, the same external function is used to segment cells bounded by horizontal lines above and below into rows. From the RSL operation text, the external function can determine the output type and the region and relation types to use in the analysis. When used this way, the observed types act as additional parameters of the external function.

The same result could be achieved by passing appropriate parameters specifying lists of model types to external functions, but this is awkward because it substantially increases the size of the recognition parameter list, and is redundant given the presence of the relevant types in the command itself. Passing the operation text does not raise the risk of functions performing unrecorded observations of model types due to the observation mechanism in RSL, described in the next Section.

3.6.2 Confidences

All inferencing operations may have confidence values such as a statistical frequency associated with their results. The value 'none' is used to indicate when no confidence was computed. Confidences are stored as attributes in the interpretation graph, so that they may be used later. An example is shown in Figure 3.6; examining this interpretation, only the segmenting operation from Figure 3.4 at 'inference time' 2 made use of numerical confidences. The remaining inferencing operations did not produce numerical confidences, and so most results in the annotated hypothesis history have a confidence value of 'none.' In RSL, all inferencing operations that do not make use of external functions return results with a confidence value of 'none.'

All hypotheses not provided as input (equivalently, givens) are assigned confidences as a strategy progresses in RSL. An example of where confidences are useful for table recognition include Hurst and Nasukawa's algorithm which uses N-grams to determine the likelihood of word continuations[60]. Within RSL, the continuation confidences could be represented as edges with statistical confidence values in a relation.

3.7 Observations in RSL

The syntax of RSL captures observations of recognition parameters and the current interpretations (see Figure 3.7). To keep RSL simple and flexible, observations within external functions are not captured by RSL. The approach taken in RSL allows more information to be captured in the implementation of feed-forward table recognition strategies; as we will see in the coming chapters, the observation framework in RSL allows us to determine dependencies between different model types, and dependencies of model types on recognition parameters. Determining these dependencies is a laborious and error-prone procedure if performed manually from source listings.

In the remainder of this section we describe how observations of hypotheses and recognition parameters are captured within RSL. Observations made by RSL strategies are summarized in Figure 3.7.

3.7.1 Hypothesis Observations

RSL uses *observation specifications* to indicate which hypothesis types support inferences. These optionally follow external function calls, and partly determine which region and relation types are present in interpretation graphs when passed to an external function. Observation specifications allow us to capture dependencies between hypothesis types in the RSL syntax itself.

In RSL, only *accepted* hypotheses are ever observable by external functions. For example, if a cell region is rejected, it will not be visible to external functions. Currently in RSL, rejected hypotheses are always filtered before an interpretation graph is passed to an external function. This is consistent with common practice in the table recognition literature, where decisions tend to be made only from what is currently

true and histories of rejected hypotheses are not maintained.

To reduce the verbosity of RSL, the built-in region structure or containment ('contains') relation is always visible; this is reasonable because the types of regions in the containment relation cannot be determined without explicitly observing region types.

The syntax of the observation specification is very simple; it lists region and relation types to be made visible to the external function (remembering that the containment relation is visible by default). The syntax for an observation specification is:

observing
{ *regions* } **regions**
{ *relations* } **relations**

where only regions or relations may be specified if needed. Region and relation types are specified as a comma-separated list.

Here we present again an example RSL operation from the previous section, where the external function *mergeCellsWithSameAdjacentLines()* is used to segment cells bounded by vertical lines into columns:

```
segment { Cell } regions into { Column } regions using
    mergeCellsWithSameAdjacentLines()
    observing
        { Vertical_line } regions
        { adjacent_left, adjacent_right } relations
```

The observation specification consists of the tail of this example, starting with the word 'observing.' This observation specification requests that vertical line regions and left and right adjacency relations be visible when interpretation graphs are passed to *mergeCellsWithSameAdjacentVlines*. The set of all visible hypothesis types for the external function are { Cell, Vertical_line } regions, and { adjacent_left, adjacent_right,

contains } relations. Note that existing column regions are not visible to the external function in this example.

Cells are visible in the above example because *scope types* of RSL functions are always visible to external functions. Scope types are the set of regions or relations manipulated by an RSL operation. In the previous example, *Cell* regions are observed by default because these are to be segmented. More generally for classifications and segmentations, scope types are the set of regions to be classified or segmented into new regions. For relation definitions, the region types to be related are visible. For 'create' and interpretation acceptance and rejection operations, only the 'contains' relation is visible by default. For all remaining operations using an external function, the specified region or relation types are visible (e.g. for reject operations).

Result type (output type) regions and relations are not observable by default. This allows us to determine whether hypotheses of the output type are observed when making a decision. For instance, in the previous example cells are segmented to form columns, but we know existing column regions are not used in the analysis, because they are not visible to the external function *mergeCellsWithSameAdjacentVlines*.

For *merge* and *reject* operations where the scope and result types are always the same, the specified region or relation type is observed by default. These operations only make sense if the external function is able to observe the specified type. For example, in order to reject region or relation hypotheses, an external function needs to be able to observe the region classes or relation hypothesis sets to be pruned.

In order to maximize the transparency of hypothesis observations, we enforce this observation scheme even when accepting and rejecting hypotheses; in fact, for these operations only the 'contains' relation is visible by default. The designer must state

explicitly all other model types used to accept or reject an interpretation.

3.7.2 Parameter Observations

Within the RSL specification we can observe which recognition parameters defined within the RSL specification are used, and for making which inferences. These are captured directly within the syntax of external function calls.

Pragmatic issues arise when considering what to parameterize within an RSL specification. For example, it is easy to overlook when a function can be parameterized further, such as when denominators of simple ratios are left as constants in an external function (e.g. for X/2). For some recognition techniques, to parameterize all variables in the algorithms may be difficult, lead to awkward implementations, or may obscure which parameters are most important.

The set of parameters to make available within an RSL specification depends on the task for which RSL is being applied; for an RSL strategy to be used in an experiment where two variables are of interest, it may make sense to parameterize only enough to control the parameters representing those variables and any other parameters on which they depend.

The selection of external parameters is of course a common problem in system design, and is not at all specific to RSL. Selecting appropriate external parameters for systems seems to be as much an art as a science.

3.8 Transformations (Book-Keeping) in RSL

The RSL Core architecture shown in Figure 3.7 was created to reduce the amount of book-keeping required from a table recognition strategy designer. It is the RSL Core and not the designer who maintains the central data structures, the interpretation tree and adaptive parameters. When an inference result is produced, the RSL Core constructs the resulting interpretations from the set of candidate interpretations as appropriate. In the process, the RSL Core annotates the appropriate history information onto the region and relation hypotheses of the new candidate interpretations. If an interpretation is accepted or rejected, it is the RSL Core which annotates the interpretation tree and marks accepted interpretations for output.

The *adapt* operation (see Section 3.9.8) returns a description of adapted parameter values, which the RSL Core then updates. The RSL Core also performs the necessary filtering of candidate interpretations before they are passed to internal and external functions to reflect observation specifications (see the previous Section).

In implementing the strategies described in Chapter 5, we found that the design and implementations were clarified and simplified by the RSL syntax and Core, allowing more of our attention to be devoted to the analytical parts of the system.

3.9 RSL Operations in Detail

In this section we summarize the complete set of RSL operations, grouped by function. For simplicity, inferencing operations that act on individual candidate interpretations are described in the context of being applied to a single interpretation, producing a single result (e.g. *create*, *classify*, and *relate*). Note also that external functions

used in the *accept interpretations*, *reject interpretations*, *for interpretations*, and *adapt* operations must take a set of candidate interpretations as an argument, rather than a single interpretation. For a description of how multiple candidate interpretations and inference results are handled in RSL, see Section 3.10.

For each operation, we provide a brief summary, the syntax of the operation, format of the returned text result, and a description of how the core data structures (interpretation tree and adaptive parameters) are affected.

Note that when inference results are redundant, the RSL Core annotates the time and confidence of the result onto the existing hypothesis (region type or relation edge). For example, classifying an existing cell as a cell will result in that region's logical type hypothesis of type 'cell' being annotated with the time and confidence of this result, leaving the hypothesis otherwise unaffected.

For quick reference, a more concise summary of the RSL operations is provided in Appendix A.

3.9.1 Terminology

Below we summarize terminology used for the RSL operation definitions in the remainder of this section.

region type(s)	identifier(s) representing a region type (or comma-separated list of region types) that have been specified in the **model regions** section of the RSL strategy (see Section 3.4.1)
relation(s)	identifier(s) representing a relation type (or comma-separated list of relation types) that has been specified in the **model relations** section of the RSL strategy (see Section 3.4.1)
node	a node (region) name
nodes	a comma separated list of node names (e.g. 'node1, node2')
pair	an ordered pair of node names
pairs	a comma separated list of ordered node name pairs (e.g. '(node1,node2), (node1,node3)')
interpretation	name of a candidate interpretation (e.g. I0, I1)
confidence	a numerical confidence value or statistic, or the identifier 'none,' used to indicate that no confidence value was produced
point list	a list of (x,y) coordinates in $\mathcal{Z}^{+2}$ (non-negative Cartesian plane)
bounding box	a pair of (x,y) coordinates specifying the top left and bottom right points of a bounding box in $\mathcal{Z}^{+2}$, where the origin (0,0) is at the 'top left' corner of the plane, so that y coordinates increase downwards
external_function	an external inferencing function, as described in Section 3.6.1
parameter list	a comma-separated list of zero or more recognition parameter names defined in the ***r*ecognition parameters** section
observation specification	an observation specification section, as described in Section 3.7
parameter	name of a recognition parameter
value	number or string literal parameter value

3.9.2 Region Creation and Classification

create

create is used to add new regions (lines or bounding boxes) of a specified type to an interpretation graph. An example use of *create* would be for creating lines in whitespace gaps of a table, to make physical separators.

Syntax:

create { *region type* } **regions using**

external_function(*parameter list*)

observation specification

Returns:

A list of regions with a physical location, region type, and confidence value. The physical regions are either lines (A) or bounding boxes (B). The text result is a list of one of the two types of region specification (A or B).

A. [*region type*] : ((POINTS , *point list*) , *confidence*)

B. [*region type*] : ((BB , *bounding box*) , *confidence*)

Effect on Interpretation:

The described regions are added as new nodes of the specified region type to the interpretation graph. The created regions neither contain nor are contained by other regions; only a region node is created.

replace

replace adds new region nodes to an interpretation graph just as in the create command, but also rejects the classification of existing regions of the same type in the process. Replacing pairs of horizontal lines with a single line at their y-midpoint is an example of when this operation is useful.

Syntax:

replace { *region type* } **regions using**

external_function(*parameter list*)

observation specification

Returns:

A list of elements of either form A or B below. Region nodes to the left of the region type are the ones to be replaced.

A. replacing *nodes* [*region type*] : ((POINTS , *point list*) , *confidence*)

B. replacing *nodes* [*region type*] : ((BB, *bounding box*) , *confidence*)

Effect on Interpretation:

The region type hypotheses of the specified type are rejected for the regions to be replaced, and the new regions described are added as nodes of the given type to the interpretation graph.

classify

classify is used to assign types to regions, possibly with associated confidence values. If multiple classes are returned (e.g. in an 'n-best' list), then the first class listed with the maximum confidence value is associated with the region.

Syntax:

A. classify { *set of region type* } **regions as** { *region type* }

B. classify { *set of region type* } **regions as** { *set of region type* } **using**
external_function(*parameter list*)
observation specification

Returns:

A list of node names, with their original types used in analysis for classification, and a list of region types (classes) and confidences representing the classification result, in the format shown below.

node [*region type*] : (*region type 1* , *confidence 1*),
(*region type 2* , *confidence 2*), ...

Effect on Interpretation:

For *classify* statements of form A above ('internal'), all existing regions associated with the specified set of region types are assigned the second region type in the interpretation graph (with confidence 'none').

For *classify* statements of form B above ('external'), if a classification result has a single class, the region is associated with that class. If multiple possibilities are returned, the leftmost 'maximum confidence' class is associated with the class in the interpretation graph.

3.9.3 Region Segmentation

segment

segment defines new regions from sets of existing regions of the given type. An example application of *segment* would be for segmenting cells into new rows or columns.

Syntax:

A. **segment** {*set of region type*} **regions into** {*region type*}

B. **segment** {*set of region type*} **regions into** {*region type*} **using**
external_function(*parameter list*)
observation specification

Returns:

A list of segments described by the segment region type, region set, and confidence value in the format shown below.

[*region type*] : ((*nodes*) , *confidence*)

Effect on Interpretation:

For *segment* statements of form A above ('internal'), all regions associated with the set of region types become children of a new region of the specified type. The containment relation is updated to include edges from the new region to each of the child regions.

For *segment* statements of form B above ('external'), new region nodes of the specified type are defined for segments, and the containment relation ('contains') is updated to include edges from each new region to its set of child regions.

For either form of the *segment* operation, the location of the segmented region is defined as the bounding box of its child regions.

resegment

resegment is used to replace the child regions of a region by altering the containment relation (revising region structure in the process). As an example use of this operation, one might use *resegment* to change the child cells of a column, where cells are added or removed from the column as determined by an external function.

Syntax:

resegment {*set of region type*} **regions into** {*region type*} **using**
external_function(*parameter list*)
observation specification

Returns:

A list of segments described by the node name of the resegmented region, the type of this region, the new child region set, and a confidence value in the format shown below.

node [*region type*] : ((*nodes*) , *confidence*)

Effect on Interpretation:

The children of a region in the containment relation 'contains' are updated: edges are added for new child regions, and edges to child regions that have been removed are rejected. The location of a resegmented region is updated to be the bounding box of the set of new child regions in the associated segment.

merge

merge combines two or more regions of a specified type into a new region of the same type. Once merged, regions are no longer associated with the given type (their region classification hypotheses of that type are rejected), and the regions become associated with the new region as children in the containment relation. *merge* can be used to describe the cell-merging operations common in bottom-up table recognition methods.

Syntax:

merge { *region type* } **regions using**

external_function(*parameter list*)

observation specification

Returns:

A list of new segments, defined by the given region type, the list of regions of that type to be merge, and a confidence value in the format shown below.

[*region type*] : ((*nodes*) , *confidence*)

Effect on Interpretation:

New regions of the given type are created for each new segment, and the given type hypothesis is rejected for the set of child regions listed for each new segment.

3.9.4 Relations on Regions

relate

relate is used to define relations other than the containment relation (for which the segmentation operations in the previous section may be used). All relations in RSL are binary, defined on the set of region nodes in an interpretation graph. Example uses of *relate* are defining spatial relations between regions (e.g. to define cell topology), defining proximity relations (e.g. defining the set of rows that are vertically 'close' to one another), and defining the indexing structure of cells in a table.

Syntax:

relate { *region type [, region type]* } **regions with** { *relation* } **using**

external_function(*parameter list*)

observation specification

Returns:

The given relation type along with a list of ordered pairs on regions and confidences, as shown below.

[*relation*] : (*pair* , *confidence*) , (*pair* , *confidence*) ...

Effect on Interpretation:

The given relation type is updated to include the specified set of region pairs.

3.9.5 Rejecting Region Type and Relation Hypotheses

reject (region types)

This operation rejects region type hypotheses produced by region creation, classification, or segmentation operations. All regions are assigned a base type 'REGION' if they are not associated with any other types in the graph after rejection. Note that this operation does not affect relation structure in any way. For example, containment structure is left in tact for regions demoted to having only type 'REGION'.

Syntax:

A. reject {*set of region type*} **classifications**

B. reject {*set of region type*} **classifications using**

external_function(*parameter list*)

observation specification

Returns:

The keywords 'reject types' are followed by a list of rejection specifications, in the form below. For each rejection specification, the type of regions to be rejected, the associated set of regions, and a confidence value for rejection hypothesis are listed.

reject types:

[*region type*] : (*node* , *confidence*) , (*node* , *confidence*) , ...

[*region type*] : (*node* , *confidence*) , (*node* , *confidence*) , ...

...

Effect on Interpretation:

For *reject* statements of form A above ('internal'), all region type hypotheses associated with the specified set of types are rejected with confidence value 'none.'

For *reject* statements of form B above ('external'), the specified region types in the returned result are rejected, meaning they are set as 'inactive' within the interpretation graph (for an example, see Figure 3.6).

reject (relation edges)

This operation rejects edges of relations produced by the *relate* operation. Note that this operation is not intended for revising the containment relation ('contains'); the segmentation operations described above are intended for that purpose. A simple example of when this might be used is when cell adjacency changes because cells have been merged. We may wish at that point to reject some or all of the existing edges in our cell adjacency relation, as part of updating the relation to reflect the current situation (we might afterward use a *relate* operation to create new edges as appropriate).

Syntax:

A. reject {*set of relations*} **relations**

B. reject {*set of relations*} **relations using**

external_function(*parameter list*)

observation specification

Returns:

The keywords 'reject relations' are followed by a list of rejection specifications, in the form below. For each rejection specification, the type to be rejected along with the associated set of region pairs and confidence value for rejection of each are listed.

reject relations:

[*relation*] : (*pair* , *confidence*) , (*pair* , *confidence*) ...

[*relation*] : (*pair* , *confidence*) , (*pair* , *confidence*) ...

...

Effect on Interpretation:

For *reject* statements of form A above ('internal'), all edges in the specified relation types are rejected with confidence value 'none.' Rejected edges are set as 'inactive' within the interpretation graph (for an example of rejected relation edges, see Figure 3.6).

For *reject* statements of form B above ('external'), all edges of the specified types in the returned result are rejected.

3.9.6 Accepting and Rejecting Interpretations

accept interpretations

accept interpretations marks interpretations from the set of candidates as accepted, adding these to the set of output interpretations. Once accepted, an interpretation is analyzed no further, becoming a leaf of the interpretation tree. Note that an external

function for this operation must accept the set of candidate interpretations and not a single interpretation as an argument.

Syntax:

A. accept interpretations

B. accept interpretations using

external_function(*parameter list*)

observation specification

Returns:

After the keywords 'accept interps,' a list of accepted interpretations, along with a confidence value for each.

accept interps:

(*interpretation* , *confidence*) , (*interpretation* , *confidence*) , . . .

Effect on Candidate Interpretations:

Marks the interpretation as accepted in the interpretation tree, and adds it to the set of output interpretations. The interpretation becomes a leaf of the interpretation tree, and is analyzed no further (i.e. it is no longer a candidate interpretation).

reject interpretations

reject interpretations removes interpretations from the set of candidate interpretations, removing them from further consideration. This turns the interpretation into a leaf of the interpretation tree. Note that an external function for this operation

must accept the set of candidate interpretations and not a single interpretation as an argument.

Syntax:

A. reject interpretations

B. reject interpretations using

external_function(*parameter list*)

observation specification

Returns:

After the keywords 'reject interps,' a list of rejected interpretations, along with a confidence value for each, in the format shown below.

reject interps:

(*interpretation* , *confidence*) , (*interpretation* , *confidence*) , ...

Effect on Candidate Interpretations:

Removes the interpretation from the set of candidate interpretations, turning it into a leaf of the interpretation tree.

3.9.7 Conditional Application of Strategies to Interpretations

for interpretations

for interpretations is the only form of conditional statement in RSL. If present, it appears as the first statement in a strategy function. *for interpretations* uses an external test function to determine which of the candidate interpretations to apply a strategy to. Candidate interpretations for which the external test fails are left as-is.

Examples of this statement are provided in the next chapter, and in the Appendices. Note that an external function for this operation must accept the set of candidate interpretations and not a single interpretation as an argument.

Syntax:

for interpretations using

external_function(*parameter list*)

observation specification

Returns:

After the keywords 'skip interps,' a list of interpretations with confidence values, in the format shown below.

skip interps:

(*interpretation* , *confidence*) , (*interpretation* , *confidence*) , . . .

Effect on Candidate Interpretations:

Candidate interpretations in the returned 'skip list' do not have the associated strategy applied to them. Note that 'skipped' interpretations remain in the set of candidate interpretations unaltered.

3.9.8 Parameter Adaptation

adapt

adapt updates the values of adaptive parameters using an external function (static parameters may not be altered; see Section 3.4.1). The new value of an adapted parameter exists only in the current and nested strategy function scopes (to avoid

side-effects). *adapt* might be used to set a decision threshold based on runtime data, e.g. to define histogram cutting values from observed bounding box projections as done in [124]. Note that an external function for this operation must accept the set of candidate interpretations and not a single interpretation as an argument.

Syntax:

adapt { *adaptive parameter list* } **using**

external_function(*parameter list*)

observation specification

Returns:

A list of parameter names and values, in the format shown below.

parameter value parameter value ...

Effect on Adaptive Parameters:

If a valid result (i.e. parameters exist, and types are correct), adaptive parameters are updated in the current scope. Static parameters are constant, and may not be altered by this operation.

3.9.9 File and Terminal Output

write

write is used to create text file output for runtime data. It takes two arguments, one describing which data to write, and a file name provided as a string literal. The types of data that may be written are:

- *aparams*: adaptive parameter names and values

- *tree*: the normalized interpretation tree structure
- *current*: the set of candidate interpretations
- *accepted*: the set of accepted interpretations

Syntax:

A. write aparams "*file name*"

B. write tree "*file name*"

C. write current "*file name*"

D. write accepted "*file name*"

print

print is identical to the *write* operation except that output is sent to the terminal (standard error stream).

Syntax:

A. print aparams

B. print tree

C. print current

D. print accepted

3.10 Representing Multiple Inference Results

As mentioned previously, all RSL inferences and parameter adaptations are applied to each candidate interpretation. Sometimes an external inferencing function might

indicate that there is more than one possible result worth considering due to some ambiguity. Consider an example where an external function for segmenting words into text lines returns three results for a single candidate interpretation as in the following:

[**Interp 1**]

[Textline] : ((Word1, Word2, ...) , 0.8)

[Textline] : ((Word57, Word58, ...) , 0.9)

[**Interp 2**]

[Textline] : ((Word1, Word2, ... , Word57, Word58, ...) , 0.7)

[**Interp 3**]

The different results are numbered ('Interp 1,' 'Interp 2,' 'Interp 3'). The first result contains two text lines, the second result contains one text line, while the third result contains no text lines (adding nothing to the candidate interpretation under this result). The number at the end of each word list is a confidence value for the result (see Sections 3.6.2 and 3.9).

Multiple results from an inference result in branching within the interpretation tree generated by an RSL strategy. When inference results transforming candidate interpretations are applied by the RSL Core (see Figure 3.7), each candidate interpretation becomes the parent of one or more new candidate interpretations. As in the previous example, it is possible for one of the returned results to have no effect, producing a child identical to the input candidate interpretation.

This mechanism for supporting multiple interpretations can obviously lead to a combinatorial explosion if not used judiciously, though one can reject unlikely interpretations to avoid this. As pointed out in Chapter 2, most table recognition systems

in the literature produce only single results for each inference, resulting in an interpretation 'tree' which is actually a single path: a sequential string of interpretations.

3.11 Summary

In this chapter we described an imitation game-based view of table recognition which motivated the construction of a small, typed functional 'glue' language to describe the inferencing 'moves' of strategies in this game. In comparison with 'imitation game' strategy implementations in general-purpose programming languages, the Recognition Strategy Language (RSL) reduces the amount of book-keeping required from a table recognition strategy designer, automatically records histories of hypothesis creation and revision, and has a syntax that captures the composition and dependencies between model types, and between model types and recognition parameters. RSL is well suited to describing the common feed-forward algorithms in the table recognition literature.

In the next chapter we describe our implementation of RSL, and a number of other tools for visualization and analysis. We will also make use of hypothesis histories to define two new metrics: *historical recall* and *historical precision*.

Chapter 4

Implementation

In the previous chapter we introduced and defined the Recognition Strategy Language (RSL) for implementing table recognition strategies. In this chapter we describe our initial implementation of the Recognition Strategy Language (RSL) using the TXL programming language (Sections 4.1-4.6). We also present a number of utilities used in this project, including tools for visualizing interpretation graphs (Section 4.9) and table models (Section 4.10), manipulating the final 'time' in an interpretation graph (Section 4.7), collecting new metrics from hypothesis histories (Section 4.8), and for manually creating interpretation graphs (Section 4.9.2).

4.1 The RSL Compiler

As shown in Figure 4.1, an RSL strategy is made executable by translating the strategy to a TXL[22, 25] program using the *RSL Compiler*, and then either compiling or interpreting the TXL program. This style of implementation is a form of *source transformation*, and is precisely the type of task for which TXL was originally

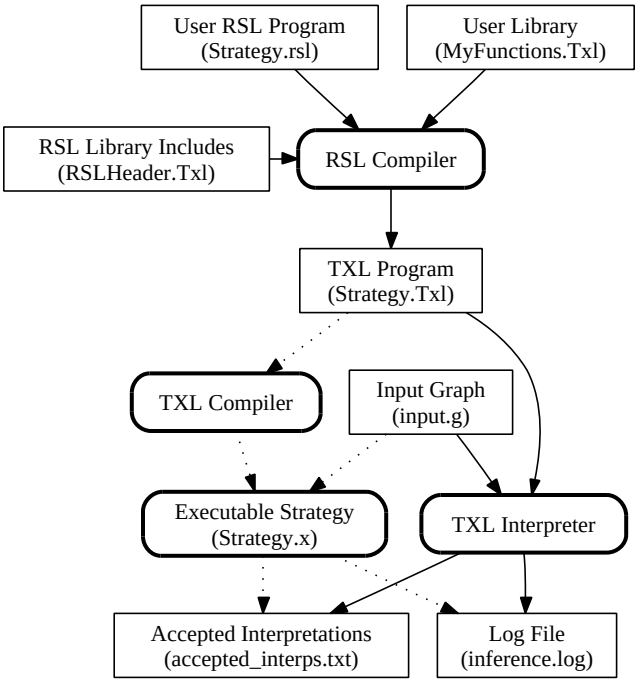


Figure 4.1: Compiling and Running RSL Programs. The output log file contains the results of all internal and external operations, and the interpretation tree constructed by a strategy may be recovered from this file.

designed[25, 24]. The translation from RSL to TXL is performed by another TXL program; we use TXL as both the target and transformation language.

In Figure 4.1, a user provides an RSL program (*Strategy.rsl*) and a library containing the external functions called from the RSL program (*MyFunctions.Txl*). The RSL Compiler translates the strategy into a TXL program (*Strategy.Txl*) that includes the user library and a header file for the RSL Library (*RSLHeader.Txl*). Once translated to TXL, the strategy may be run with an input graph in one of two ways: using the TXL Interpreter, or by compiling the TXL Program to produce an executable strategy *Strategy.x*, and then passing the input to *Strategy.x*. The output

of running a strategy on an input is two files: one containing accepted interpretations (*accepted_interps.txt*), and another containing a record of all operation results (*inference.log*).

Our preliminary implementation of the RSL Compiler is a shell script that translates an RSL strategy into a TXL program, in two steps:

1. The passed strategy file (*Strategy.rsl*) is translated into a partial TXL program (*Partial.Txl*) using another TXL program, *ConvertRSL.Txl*
2. 'include' statements for the user (*MyFunctions.Txl*) and RSL libraries (*RSLHeader.Txl*) are placed at the top of *Partial.Txl*. This produces the final TXL program (*Strategy.Txl*).

The RSL Compiler is a small program written in the GNU Bourne Again Shell scripting language (bash[97]). *ConvertRSL.Txl* is a TXL program of approximately 1300 lines in length, including blank lines and comments. *ConvertRSL.Txl* uses grammars for RSL and TXL that together are approximately 1000 lines (again, including blank lines and comments). Note that only subsets of these grammars are used in the translation.

In the next section we introduce TXL. The RSL to TXL translation produced by *ConvertRSL.Txl* is described in detail in Section 4.3, and the implementation of user libraries in TXL is covered in Section 4.4.

4.2 The TXL Programming Language

TXL is a special-purpose first-order functional programming language originally designed for rapid prototyping of programming language dialects[25]. TXL has been

applied to a wide variety of tasks, including software engineering, source code transformation, VLSI layout, natural language understanding, database migration, network protocol security, and recognizing mathematical expression structure[9, 123]. Perhaps most notably, TXL formed the core of a 'year 2000' conversion process that was applied to billions of lines of source code in an industrial setting[29].

TXL programs first parse input text using a context-free grammar, restructure the parse tree using transformation functions, and then linearize the result and return the transformed text. TXL grammars are ordered, which makes it possible to describe the syntax of multiple formats (e.g. RSL and TXL) within a single grammar, using 'pivot' non-terminals that contain statements from both languages. Within such a non-terminal, rules of the input language (e.g. RSL) are placed first, so that they are applied when TXL parses the input text. An example of a 'pivot' non-terminal is the *region_definition* non-terminal in Figure 4.2.

TXL transformation functions use a pattern-and-replacement syntax (*this-means-that*[22]): the pattern follows the keyword **replace**, and the replacement follows the keyword **by**. In Figure 4.2, the function *translateRegionDefinition* matches an RSL **model regions** section, replacing it with a TXL export statement for the global variable *Regions*. In TXL the **construct** statement defines variables, while **deconstruct** statements decompose the tree rooted at a variable into component parts (terminals and non-terminals). [**repeat** X] is TXL syntax for a sequence of 0 or more X. The complete syntax of TXL is defined in the TXL Programming Language manual[23]. A brief summary of the TXL grammar syntax is provided in Appendix B. In this chapter we will define additional TXL syntax as needed.

We chose to implement RSL in TXL for a number of reasons. First, it is a

RSL Region Type Definition	TXL Region Type Definition
model regions Image Word Line Cell Column Row **end regions**	**export Regions**[**repeat** id] 'Image 'Word 'Line 'Cell 'Column 'Row

```
% a. 'define' indicates non-terminal definition; literals
% are quoted with a single quote (')
% b. [repeat X]: 0 or more [X] nonterminals
% c. [id]: an identifier
define model_regions
    'model 'regions
        [repeat id]
    'end 'regions
end define

define quoted_id
    % '' represents one single quote
    [id] | '' [id]
end define

define TXL_region_export
    'export 'Regions '[ 'repeat 'id ']
        [repeat quoted_id]
end define

define region_definition
    [model_regions] | [TXL_region_export]
end define

function translateRegionDefinition
    replace [region_definition]
        ModelRegions[model_regions]

    deconstruct ModelRegions
        'model 'regions
            RegionTypes[repeat id]
        'end 'regions

    construct TXLCommand[TXL_region_export]
        'export 'Regions '[ 'repeat 'id ']
            RegionTypes[quoteIdentifiers]

    by
        TXLCommand
end function
```

Figure 4.2: TXL Program for Translating Region Type Definitions. *translateRegionDefinitions* converts an RSL Region Type Definition to its TXL equivalent, as shown in the above example. The only function application in this example is at line 36, where **[quoteIdentifiers]** is applied to the list of region name identifiers *RegionTypes* to prevent region names from matching TXL keywords. Elsewhere square brackets contain types, except where quoted as a literal (lines 17 and 35).

functional language, matching RSL's functional paradigm (see Section 3.4). Further, the core data structures of RSL are stored as structured text, functions in RSL return structured text results, and the RSL Core library must frequently manipulate these text encodings. TXL is well suited to performing all of these tasks.

TXL grammars and functions allow many programming tasks to be described at a very high level of abstraction; there are built-in operators to retrieve all subtrees of a non-terminal type, and parsing and searching are *primitive* operations of the language. TXL has a pure functional value semantics, passing all data by value, and so there are performance trade-offs in using TXL versus a language employing data pointers such as C/C++. We were primarily concerned with creating a complete RSL implementation as quickly and simply as possible, and so we chose to trade execution time for implementation time.

4.3 Translating RSL Programs to TXL

In this Section we summarize the process for translating an RSL strategy to a TXL program using the TXL program *ConvertRSL.Txl*. The process of converting an RSL strategy to a TXL program is reasonably straightforward due to the small size and simplicity of RSL. The overall process can be summarized as in the following.

1. The RSL strategy is parsed using the grammar provided in Appendix B
2. The RSL strategy header is translated to create the main function of the TXL program, from which execution begins (see Figure 4.3)
3. The RSL strategy functions are translated to TXL functions. All strategy functions have the prefix 'Str_' added to their name in the TXL program (see

Figure 4.4)

4. All external function calls are converted into TXL functions. This is a template-based conversion, where the user's external functions are 'wrapped' in a TXL function that calls the user's function as well as functions from the RSL Core library. The 'wrapping' functions have the same name as the external function, with the additional prefix 'inf_' (see Figure 4.5)

5. The transformed parse tree is linearized and returned as a partial TXL program. The program is partial because headers must be added to include user and RSL functions called in the program (this is done by the RSL Compiler, described in Section 4.1)

We describe and provide examples for each of these steps in the remainder of this Section.

Though RSL is a functional language, we use the global variable operations of TXL in our implementation. We do this for two reasons. First, TXL limits the number of **construct**, **deconstruct**, and conditional tests (**where**) in each function to a fixed number. Keeping some of the data in global variables reduces the number of these statements required in each function, in particular by keeping static parameters in a global symbol table. Second, without the use of global variables some book-keeping tasks would require data to be passed throughout the entire program, to be altered in only a few places. Examples include counts for the number of inferences performed, and the number of nodes (regions) created by RSL.

Perhaps most importantly, the set of accepted interpretations and the log file containing function results were stored in global variables for the reasons indicated

above. A pure functional implementation of RSL is of course still possible, but would be much more difficult within TXL.

A number of the examples in this Section will make use of source listings with numbered lines. To concisely describe corresponding lines in translations we use the following reference notation: **RSL(1,10-12)** represents lines 1 and 10-12 of the RSL strategy in a Figure (similarly for a TXL listing in the same Figure), and **RSL(5-7):TXL(6-8)** indicates that lines 5-7 of the RSL listing correspond to lines 6-8 of the translated TXL source.

4.3.1 RSL Header to TXL Main Function Translation

The RSL to TXL translations begins by translating the RSL header to the TXL 'main' function as demonstrated in Figure 4.3. Please see the preceding Section for a description of the line referencing conventions used here (e.g. RSL(1-4)). We organize our description by the RSL program sections and supporting TXL code.

model regions: RSL(1-4):TXL(6-7)

Figure 4.2 demonstrates this translation.

model relations: RSL(6-8):TXL(9-10)

This translation is nearly identical to that for **model regions**.

recognition parameters: RSL(10-13):TXL(13-19)

adaptive and static parameter lists are defined at TXL(13-17). In this example there is just one adaptive parameter *aScanResolution* and one static parameter *sNumber*, which is exported as a global variable at TXL(19).

RSL Header

```
model regions
    Image Word Line
    Cell Column Row
end regions

model relations
    above below indexes
end relations

recognition parameters
    sNumber              5
    aScanResolution    300
end parameters

strategy main
    ...
end strategy

strategy subStrategy
    ...
end strategy
```

TXL Main Function

```
function main
    replace [program]
        G[graph]

    % Export region and relation types
    export Regions[repeat id]
        Image Word Line Cell Column Row

    export Relations[repeat id]
        above below indexes contains

    % Create adaptive, static params
    construct AParamList[parameter_list]
        aScanResolution 300

    construct SParamList[parameter_list]
        sNumber 5

    export SParamList[parameter_list]

    % Intialize adaptive params, candidates
    construct InterpGraphs[repeat interp_graph]
        I0 A: G[labelInputs][activateAll]

    construct InitialCore[core_data]
        AParamList InterpGraphs

    % Export empty lists for accepted
    % interps and log file
    export Accepted[repeat interp_graph]
    export LogFile[repeat log_entry]

    by
        InitialCore[rsl_defineCounters]
            [Str_main]  % main strategy fn
            % write output to files
            [rsl_writeAcceptedInterpretations]
            [rsl_writeLog]
end function
```

Figure 4.3: Creating the Main TXL Function for an RSL Strategy

RSL main strategy: RSL(15-17):TXL(35)

the main strategy is later translated as the TXL function *Str_main*, which we will describe in the next Section. At TXL(35), the main strategy is applied to the initial adaptive parameter list and input graph (stored in the variable *InitialCore*).

supporting TXL code: TXL(22-31,34,37-38)

In lines TXL(22-26) the core data structures are initialized; these are represented by the variable *InitialCore*, and consists of the adaptive parameters and initial interpretation tree (the passed graph). In lines TXL(30-31), the set of accepted interpretations and log file are initialized as empty lists, and then exported as global variables.

At the end of the translated TXL main function two RSL library functions are called which write out accepted interpretations and log entries as text files (to *accepted_interps.txt* and *inferences.log*, respectively).

4.3.2 RSL Strategy to TXL Function Translation

In Figure 4.4, we present the translation of two RSL strategy functions to TXL functions. As in the last section, we will organize our summary of the translation by RSL sections and supporting TXL code sections.

strategy main ... end strategy: RSL(1,25,27-29):TXL(1,48,50-52)

strategy function names are altered in their translation to TXL, appending the prefix 'Str_' (RSL(1,27):TXL(1,50)). The delimiters for

RSL Main Strategy

```
strategy main
    for interpretations using
        applyOnlyToInterps(sNumber)
        observing
            {Word} regions

    adapt aScanResolution using
        getScanResolution()
        observing
            {Image} regions

    classify {Word} regions as {Cell}

    subStrategy

    relate {Cell} regions with {indexes} using
        determineIndexStructure(aLanguage)
        observing
            {Word, Cell, Row, Column} regions
            {left, right, above, below} relations

    accept interpretations

    print "Strategy complete."
end strategy

strategy subStrategy
    ...
end strategy
```

TXL Translation of Main Strategy

```
function Str_main
    % P contains adaptive parameters
    replace [core_data]
        P[parameter_list]
        Candidates[repeat interp_graph]

    import sNumber[number]  % static import

    % Define the RSL commands here as variables
    construct Op_0[strategy_op]
        for interpretations using
            applyOnlyToInterps(sNumber)
    ...
    construct Op_6[strategy_op]
        print "Strategy complete."

    % Filter interpretations ('for interp.' op)
    construct SelectedInterps[repeat interp_graph]
        Candidates[for_applyOnlyToInterps sNumber Op_0]

    construct LeaveInterps[repeat interp_graph]
        Candidates[filterInterps SelectedInterps]

    where not
        SelectedInterps[isEmpty]

    construct FilteredCore[core_data]
        P SelectedInterps

    % Apply remaining operations (adapt...print)
    construct Result[core_data]
        FilteredCore[ap_getScanResolution Op_1]
            [rsl_labelRegions 'word 'cell]
            [Str_substrategy]
            [inf_determineIndexStructure "english"
                Op_4]
            [rsl_acceptInterpretations]
            [message "Strategy complete."]

    deconstruct Result
        PNew[parameter_list]
        NewInterps[repeat interp_graph]

    by
        % Return *original* adapt params,
        % new candidate interpretations
        P LeaveInterps[. NewInterps]
end function

function Str_subStrategy
    ...
end function
```

Figure 4.4: RSL Strategy Function Translation

strategies (**strategy...end strategy**) are simply exchanged for TXL function delimiters (**function...end function**).

for interpretations: RSL(2-5):TXL(7,18-28)

interpretations to apply the main strategy to are determined by the external function *applyOnlyToInterps* at RSL(3). This function has been wrapped into a TXL function *for_applyOnlyToInterps* at TXL(19). TXL(18-22) constructs the sets of candidate interpretations to transform or leave intact. The TXL function will not be applied, leaving data matched by the pattern unaltered if the set of candidates to transform is empty, as tested by the **where not** condition at TXL(24-25). If candidates remain, a variable containing the adaptive parameters and set of candidates to transform is created (TXL(27-28)).

At TXL(7), the static parameter *sNumber* passed to *applyOnlyToInterps* is imported; as mentioned earlier, static parameters are stored as global variables in this implementation.

RSL Operation Variables: RSL(2-24):TXL(10-15)

RSL operations are encoded directly within the TXL function using variables. **for interpretations** operations, if present, are named *Op_0*, and any additional operations in the strategy are numbered from 1 (e.g. *Op_0*, *Op_1*, etc.). These are used by external functions and the RSL library. We abbreviate this list using an ellipsis at TXL(13).

Main Strategy Operations: RSL(7-24):TXL(31-38)

the operations after the **for operation** in the main RSL strategy are applied as a sequence of function applications applied to the adaptive parameters and candidate interpretations (represented by *Filtered-Core*). Note the translation of the call to *subStrategy* (RSL(14)) at TXL(34). As described in the next Section, different RSL operation types have different translated names in TXL. *rsl_* prefixes indicate built-in RSL library functions, *ap_* prefixes indicate adaptive parameter operations, *Str_* indicate function calls to other translated strategies, and *inf_* indicates a call to an inferencing function. Some operations are translated directly to TXL operations, such as the **print** statement at RSL(24) being translated directly to a TXL **message** statement at TXL(38).

Parameters are translated depending on their type. For parameters of built-in (*rsl_*) functions and adaptive parameters, the parameter values are passed directly (for adaptive parameters this acts merely as a placeholder; see the next subsection). Static parameters passed to external functions are translated by name, with values being imported from global variables at run time as described above. Translated RSL operations with external functions are always passed the text of the RSL operation itself (e.g. TXL(19,32,35-36)).

TXL Pattern and Replacement: TXL(3-5,40-47)

additional code is made to specify the TXL function pattern (the adaptive parameter list and candidate interpretations at TXL(3-5)), and to deconstruct the result of executing the strategy to make it

possible to return the original parameter list, to avoid side-effects (TXL(40-47))

4.3.3 RSL External Function Call to TXL Function Translation

Figure 4.5 presents a TXL function *inf_determineIndexStructure* which wraps a user's external inference function *determineIndexStructure*. This function is called in Figure 4.4 at TXL(35). While the function shown in Figure 4.5 wraps a specific function for an RSL *relate* operation, the bulk of the code is in fact a template used to wrap all external function calls in the translated TXL program; only very minor variations are required for operations that use the whole candidate set in their analysis (*accept interpretations*, *reject interpretations*, *for interpretations*, *adapt*).

Making reference to sections of Figure 4.5, the main components of a 'wrapping' function in a TXL-translated strategy are:

signature: line 1

a wrapped function always has at least one parameter: the text of the original RSL strategy operation (*RSL_Op*). Additional parameters are automatically named *P1,P2,...* and assigned a type based on the type of the *passed* parameter in the original RSL specification. Type inconsistencies for parameters are caught at compile or interpretation time.

parameter initialization: lines 9-15

```
function inf_determineIndexStructure P1[stringlit] RSL_Op[strategy_op]
    replace [core_data]
        C[core_data]

    deconstruct C
        P[parameter_list] G[repeat interp_graph]

    % Get parameter names from RSL operation
    deconstruct * [list param_name] RSL_Op
        PassedParams[list param_name]

    % Lookup adaptive parameter values in parameter list
    % ** '_' indicates an empty string literal ("")
    construct P1_p[stringlit]
        _[rsl_observeAdaptiveStringlitValue PassedParams 1 P]

    % Apply scope types and observation specification
    % to candidate interpretations
    construct FilteredInterps[repeat interp_graph]
        G[rsl_observeGraphs RSL_Op]

    % Apply external function to each filtered candidate
    % ** '_' indicates an empty list of interp_result
    % ** 'each' indicates to apply determineIndexStructure to
    %     each candidate interpretation in FilteredInterps
    construct NewInterps[repeat interp_result]
        _[determineIndexStructure P1_p RSL_Op each FilteredInterps]

    % Create log entry, and append to the log
    construct Result[repeat inference_result]
        _[rsl_incrementInferenceCount]
         [rsl_createResult RSL_Op FilteredInputs NewInterps]
         [rsl_appendInferenceToLog]

    % Type-check results using the RSL operation
    assert
        Result[rsl_isValid RSL_Op]

    by
        % Have RSL Core library apply the result
        C[rsl_updateCoreData Result]
end function
```

Figure 4.5: Example 'Wrapped' External Inference Function. This function takes two parameters, the first a string literal placeholder for an adaptive parameter, the second being the text of an RSL operation making use of this function. After results have been collected and appended to the log file, the result is checked (line 36) and then applied to the current set of adaptive parameters and candidate interpretations (line 41). **[list X]** is a comma-separated list of 0 or more X objects.

as mentioned in the previous section, static parameters have their values passed within a TXL translation of an RSL strategy function. For adaptive parameters, the original values of the parameters in the original RSL strategy are passed but ignored; as shown here in lines 9-15, the adaptive parameter list P is consulted to obtain current values of adaptive parameters; it is this value that will be passed to the external function (e.g. line 27, where *P1_p* is passed for parameter *P1*).

observing candidate interpretations: lines 19-20

the visibility of model types is controlled based on scope types and the observation specification in the RSL operation text (*RSL_Op*. See Section 3.7 for more on observation specifications). Types which are not visible are removed from all candidate interpretations using the RSL Core library function *rsl_observeGraphs*.

applying user's external function: lines 26-27

in this example the user's external function *determineIndexStructure* is applied to each visible-type-filtered candidate interpretation individually, producing a list of results. Note that for external functions defining operations such as *accept* that make use of the set of all candidate interpretations, only a single result is returned.

creating result entry in log: lines 31-33

RSL Core library functions are called to create an inference log from the list of results produced by the external function. *rsl_appendInferenceToLog*

appends the log entry to the log, defined as a global variable (see Section 4.3)

type-checking result lines 36-37

here the result is checked by the RSL Core library, using the passed RSL operation text. In essence, the result entries are checked for grammatical structure, and to insure types correspond to types in the RSL operation. At present this function is not implemented; the grammatical form of results is enforced, but individual types are not. Currently we check results manually.

update adaptive parameters and candidates: line 37

the *rsl_updateCoreData* function and its related subfunctions are perhaps the largest component of the RSL Library; this function takes any structured text decision result, and updates the adaptive parameter list and candidate interpretation set appropriately.

4.4 Implementing External Functions in TXL

RSL was designed to allow arbitrary decision techniques to produce the results for inferences in an RSL strategy. We illustrate external functions using just one example in this Section, the user library function *determineIndexStructure* referred to previously in Figure 4.4 RSL(17) and Figure 4.5 line 27. Much of the code in Figure 4.6 is repeated in various inference functions, and in practice we defined a macro to quickly produce a template containing much of what is shown in this example.

```
function determineIndexStructure  Language[stringlit]
    RSL_Op[strategy_op] C[interp_graph]
  replace [repeat interp_result]
    R[repeat interp_result]

  deconstruct C
    SInfo[interp_node_info] G[graph]

  deconstruct RSL_Op
    relate { ScopeRegion[id] } regions with { Relation[id] }
    Ext[external_call]
    observing
      { ObservedRegions[list id] } regions
      { ObservedRelations[list id] } relations

  .... % code for decision process omitted...

  % Contruct list of result in format described
  % in Chapter 3 for relate operations; note
  % multiple interps ([Interp 1]...[Interp N])
  % are permitted here.
  construct InterpResults[repeat relate_result]
    ...

  % Pass result back as original candidate name
  % and list of results.
  construct Result[interp_result]
    SInfo
    InterpResults

  by
    % Append results for this candidate
    % to the list of all results
    R[. Result]
end function
```

Figure 4.6: Example TXL External Inference Function from a User Library

User functions in this RSL implementation are all TXL functions taking a list of user-defined parameters, the RSL operation text itself (*RSL_Op*), and an interpretation graph C; the last two arguments must be present for all external functions. Note that the set of all candidate interpretations must be the final argument for user functions to define operations working on the set of candidate interpretations (e.g. *accept*). The result of an external inferencing function is the appropriate structured text format described in Section 3.9. For this example, this will be a *relate*-format result.

As mentioned, the decision procedure may be arbitrary, perhaps using some of the model type names bound in the **deconstruct** of the RSL operation at lines 10-14. We indicate this with an ellipsis at line 16. For a *relate* operation multiple results may be returned; this is reflected in the decision result being a list (*InterpResults*). At lines 27-29, we prepend the identifier for the passed candidate interpretation to the inference result, to preserve this information in the log file.

A function such as *determineIndexStructure* is applied iteratively over the list of candidate interpretations (see Figure 4.5 line 27). As a result, the final *Result* record is appended to a list of results for other candidate interpretations. The result is then returned to the calling 'wrap' function, which updates RSL data as described in the previous Section.

4.5 Running Translated Strategies: the RSL Library

Once an RSL strategy has been translated to a TXL program and the external functions called from the strategy have been defined in a user library, the program may be compiled or interpreted using the RSL library, which implements all of the functionality of RSL. In this Section we briefly summarize the main components of the library and their sizes in lines of code (again including blank lines and comments). In the current implementation, this entire RSL library consists of approximately 12,000 lines of TXL code. This may be broken down into three main components:

Geometry library: (1700 lines) geometric functions for maintaining region structure, constructing bounding boxes, and many convenience

functions for external inferences (e.g. to get the left side of a bounding box)

Graph library (5400 lines): defines operations for defining and manipulating interpretation graphs.

RSL Core library: (see Figure 3.7, 5000 lines) most of the *rsl_* prefixed functions earlier in this chapter are part of the RSL Core, which interprets results from internal and external functions and then updates the interpretation tree and adaptive parameters appropriately. The largest subsystem is the *updateCoreData* function and its subfunctions, which takes any valid text result and then updates the interpretation tree or adaptive parameters appropriately. Definitely the most complex part of the library implementation.

Note that the code would be significantly shorter if TXL supported higher-order functions; significant parts of the code are near duplicates. For example, a roughly 100 line binary search algorithm had to be defined uniquely for every type that would use it (e.g. to use it with 5 types, 500 lines are required). Currently the possibility of higher order functions in TXL is being explored[22].

A designer programming external functions for RSL strategies makes frequent use of the Geometry and Graph libraries; our experience has been that their expressive power is one of the largest bottlenecks in implementing external functions, and much of the time spent on the implementation was spent on extending these libraries, which started out containing only very primitive operations.

In the next Section we describe data manipulated by the RSL Core library.

4.6 RSL Data

Reflecting TXL's text transformation paradigm, the central data structures of RSL are stored in ASCII text files using a structured text format. For the initial implementation we decided against using the Extensible Markup Language (XML[11]) in order for the data be human-readable. TXL has built-in support for XML, and in the future an XML encoding of the central data structures may be worth considering in order to take advantage of the many publicly available tools for XML data.

In the remainder of this Section we describe the encoding formats for interpretation trees, parameter adaptation results, and interpretation graphs used in RSL.

4.6.1 Interpretation Trees and Log Files

RSL strategies build interpretation trees breadth-first; once a candidate interpretation is altered by an inference, it is never observed again. To simplify the implementation, a single set of interpretation graphs are updated and annotated with hypothesis histories (see Sections 3.5.3 and 4.6.3) by the RSL Core library (see Figure 3.7). The results of all decisions passed to the RSL Core library, including all inferences, are recorded in a log file. The interpretation tree generated by an RSL strategy may be reconstructed from this log file after execution of the strategy. An example of a log file containing inferences is shown in Figure 4.7.

Candidate interpretations associated with a single result from an inferencing operation are altered in the set of candidate interpretations in-place. When multiple results are returned from an inferencing operation for a candidate interpretation, the interpretation is copied to produce the appropriate number of result interpretations

```
Inference 1 : (I0)
    relate {Word} regions with {below} using
        findLowerAdjacentWords()
[Interp 1]
    [below] : ((Word2, Word4), none)


Inference 2 : (I0)
    segment {Word} regions into {Column} regions using
        segmentVerticalWordGroupsAsColumns()
        observing
            {below} relations
[Interp 1]
    [Column] : ((Word1), 0.4)
    [Column] : ((Word2, Word4), 0.9)
    [Column] : ((Word3), 0.6)
[Interp 2]
    [Column] : ((Word1, Word2, Word3, Word4))


Inference 3 : (I1)
    reject {Column} classifications using
        rejectSingleCellColumns()
[Interp 1]
    [Column] : (Column1, none), (Column3, none)


Inference 4 : (I2)
    reject {Column} classifications using
        rejectSingleCellColumns()
[Interp 1]


Inference 5 : (ALL)
    accept interpretations
[Interp 1]
    accept interps : (I1, none), (I2, none)
```

Figure 4.7: Example Log File for Strategy in Figure 3.4. The input for this example is the set of Word regions shown in Figure 3.5a. Inference 3 produces the interpretation shown in Figures 3.5b and 3.6. The segment operation (Inference 2) produces a second result in which all words in the input are assigned to one column. Note that the interpretation identifiers (I0, I1, I2) correspond to branches in an extensive interpretation tree, or node names of a normalized interpretation tree (see Section 3.3). The interpretation tree described by this file is shown in Figure 4.8.

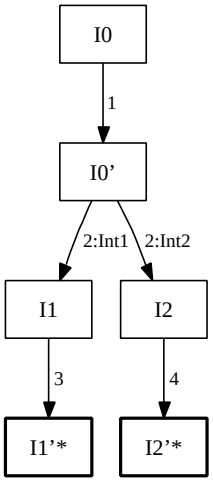


Figure 4.8: Extensive Interpretation Tree for Log File in Figure 4.7. Interpretation identifiers have been primed here (I1', I2') to distinguish interpretations after inference results have been applied. Numbers on edges correspond to the sequence of inference results in the log file. The result of the fifth operation accepting all candidate interpretations is indicated by asterisks on the leaf node interpretation identifiers. Note that for the second inference there are two distinct results; here we have indicated the first result with edge label '2:Int1,' and the second with edge label '2:Int2.'

and then removed, with each copy assigned an identifier indicating that a new branch has been added to the interpretation tree. Each result for the original candidate interpretation is then applied to one of these copies. Interpretations named in results from *accept interpretations* and *reject interpretations* operations are removed from the set of candidate interpretations.

As seen in Figure 4.7, inference results are recorded in the log with entries in the following format:

Inference *inference time* : (*candidate interpretation*)

RSL inference operation

[**Interp 1**]

inference result

...

[**Interp N**]

inference result

inference time is the simply the number of inferences that have been made in the course of a strategy; the first inference of a strategy is applied at time '1.' The *candidate interpretation* which the result is associated with is indicated by I*number* (e.g. I0, I1, I2), representing the branch of an interpretation tree that the candidate interpretation is from. Each *inference result* is presented under a numbered heading (e.g. **Interp 1**). Note that an *inference result* may be empty as for Inference 4 in Figure 4.7. An empty inference result indicates that no hypotheses were generated for that outcome. In this case, the input candidate interpretation is left unmodified in the set of candidate interpretations.

Figure 4.7 shows a possible log file generated by the strategy shown in Figure 3.4. In this strategy, a *below* relation is defined, a segmentation of words into columns based on this relation is performed, columns containing a single word are rejected, and then all candidate interpretations are accepted and returned as output. In Figure 4.7, the segmentation operation produces two possibilities: one in which three columns are detected, and a second in which one column is detected. The log file contains the full text of all inference operations, the candidate interpretations to which they were applied, and all inference results. The interpretation tree represented in Figure 4.7 is shown in Figure 4.8.

Operations that use or manipulate all candidate interpretations (*accept interpretations*, *reject interpretations*, *adapt*, *for interpretations*) return only a single result. An example for the *accept* operation is shown in Figure 4.7.

4.6.2 Adapted Parameters

In addition to inferences, the result of all parameter adaptations are recorded in the log file. Consider the following log entry describing an *adapt* operation that observes *Word* regions in order to alter the number-valued *aNumericThreshold*, and the string literal-valued *aLanguage*. Suppose that *aNumericThreshold* is used to define a vertical proximity threshold for cells, while *aLanguage* is used to capture the detected language of the input table.

Parameter Transform [*adapt number*] : (**Last Inf** : [*inference time*])
adapt aNumericThreshold, aLanguage **using**
updateThresholdAndLanguage()
observing
{Word} **regions**
aNumericThreshold 2.5
aLanguage “french”

adapt number is simply a count of the number of *adapt* operations executed by a strategy. *inference time* is the number of previously executed inferences at the time the *adapt* operation is applied. As with the inference entries, the full text of the RSL command is included in the log entry. Here the external function requested the RSL Core to update *aNumericThreshold* to 2.5 (e.g. a distance in millimetres), and the *aLanguage* parameter to “french”.

4.6.3 Interpretation Graphs

An example of the text encoding for interpretation graphs is shown in Figure 3.6. The table models represented in RSL interpretation graphs are summarized in Section 3.5. RSL Core library operations for filtering logical types to enforce observation specifications is covered in Section 3.7, and previously in this Section we summarized how candidate interpretations are updated. In this section we provide additional details of the interpretation graph encoding.

At the outermost level of the encoding, region (node) and relation (edge) types are represented in the formats *N [RegionType]* and *E [RelationType]*, respectively. Below each region type header is the list of regions that have been associated with the type, including any rejected classifications. Similarly, pairs of physical regions appear below appropriate relation type headings, again including any edges that may have been rejected during analysis.

Region type entries and edges in the graph have a list of comma-separated attributes, each of which consist of an identifier followed by a list of numbers or string literals. All region type entries contain an attribute to describe the physical location (geometry) of a region; this is described either by a bounding box (*BB*), or list of points (*POINTS*). The RSL Core library insures that a region's location attribute is fixed for all associated region type entries when updating an interpretation graph. For example, if a region with associated types *block* and *column* was resegmented, changing the bounding box of the region, then the location attribute would be updated for the region type entry under *block*, and the region type entry under *column*.

Both region types entries and edges have an *Active* attribute with value "yes" or

"no" to indicate whether a hypothesis is accepted in the final state of the interpretation, and attributes that describe the *hypothesis history* of an interpretation (see Section 3.5.3). Attribute identifiers used to represent the hypothesis history are the following:

Input	provided in the input graph
Create	for *create* and *replace* operations
Class	for *classify* operations
Segment	for *segment* and *merge* operations
Resegment	for *resegment* operations
Relate	for all operations that create relation edges (*relate*, *segment*, *resegment*, *merge*)
Reject	for operations that reject region types and relation edges (*reject*, *replace*, *merge*, and *resegment* operations)

Some operations are represented by multiple labels in the graph. For example, a *merge* operation applied to cells will be represented by a *Segment* label for the new region, *Relate* labels for edges from the new region to the merged cells in the containment relation ('contains'), and *Reject* labels for the *cell* region type entries of the merged cells.

All input region types and relation edges are labelled with the attribute *Input*, while region types and relation edges generated by a strategy's execution are annotated using the appropriate history attributes above. To reduce the size of the representation, a hypothesis history identifier appears at most once for each region type entry or edge, followed by a list of string literals describing the inference time and confidence associated with each inference. The format of these string literals is

simply:

"Inf *inference time*: *confidence***"**

As mentioned earlier, *inference time* indicates when an inference is made, using the count of inferences made, and *confidence* is a numerical confidence value associated with the inference, or 'none,' if no confidence value was produced.

The hypothesis history attributes summarize the construction of an interpretation, corresponding to a path in the interpretation tree. This summary is partial. To obtain the complete details of the construction history requires examining the interpretation tree, whose representation in a log file was described previously in this Section. Though partial, the annotated hypothesis history provides enough information to quickly determine when an individual hypothesis (region type or relation edge) was created, rejected, or revised, and by what type of operation. From the history attributes we can also determine which hypotheses were active at each inference time in an interpretation, as we will see in the next Section.

External functions may annotate interpretation graphs with additional attributes, but these annotations will have only local scope. The RSL Core library updates candidate interpretations based only on inference result descriptions (see Figure 3.7), which do not describe these additional attributes. As an example, in an external function that must determine which lines bound cells, a strategy designer might add an attribute to *cell* region type entries that list the names of line regions adjacent to each cell. While useful within the external function, this information would not be added to the interpretation tree when the result of the external function was applied by the RSL Core library.

4.7 Recovering Previous Interpretations

Hypothesis histories permit us to revert an interpretation graph to a previous state quite simply. To revert an interpretation graph G to a time N (where time 0 is the input graph), we can use the function *reverseClockToInference()*, defined below.

reverseClockToInference(G,N)

1. Remove all hypothesis history values (of the form **"Inf** *inference time*: *confidence***"**) in G that have an *inference time* greater than N.
2. Remove all region type entries and relation edges in G with only empty-valued hypothesis history attributes, except for those with the attribute *Input*. These were generated after time N.
3. Update the *Active* attribute for each region type entry and relation edge in G to indicate whether it was accepted at time N. For each region type and edge, if the the last recorded operation performed is of type *Reject*, then set *Active* to "no." Otherwise, set *Active* to "yes."
4. Remove all empty relations and region type sections from G
5. Return G

With some additional filtering, *reverseClockToInference()* can be used to return the state of the graph at each time that a particular region or relation type was altered. For example, we can scan the hypothesis histories of all *cell* regions to obtain the set of inference times at which cells were defined or altered. We can then return the graphs at each of these times. We could also filter the graphs again so that they contained only *cell* regions.

4.8 Metrics Based on Hypothesis Histories

Using the hypothesis histories and the *reverseClockToInference()* function described in the previous Section, we are now able to observe some new and useful metrics for the table recognition literature. In the following discussion, we consider the case where we wish to compare an interpretation S generated by a strategy to an accepted ('valid') interpretation A for a single input file F (e.g. a single table image). These metrics could be used for defining the distance metric in a table recognition imitation game (see Section 3.2).

For a given hypothesis type H (e.g. the set of *word* regions contained by a *cell* region), we can now use the hypothesis history in S to observe the following:

Hypothesis Sets:

AF: set of accepted H hypotheses at the final inference time in A

SF: set of accepted H hypotheses at the final inference time in S

SH: set of H hypotheses accepted at *any* inference time in S
(*history* of all H generated in S)

Metrics:

$$\textit{Recall:}\ \frac{|AF \cap SF|}{|AF|} \qquad \textit{Historical Recall:}\ \frac{|AF \cap SH|}{|AF|}$$

$$\textit{Precision:}\ \frac{|AF \cap SF|}{|SF|} \qquad \textit{Historical Precision:}\ \frac{|AF \cap SH|}{|SH|}$$

Here $|AF \cap SF|$ represents the number of 'correct' H hypotheses at the final inference time in S, and $|AF \cap SH|$ represents number of 'correct' H hypotheses generated at any inference time in the history described in S. We have included conventional recall and precision metrics here for comparison.

With the ability to record rejected hypotheses in our interpretation graphs, we can now observe *historical* recall and precision metrics. These are simply recall and precision metrics for the set of all unique hypotheses that existed at some time in an interpretation graph, as opposed to the recall and precision of just the final set of accepted hypotheses.

The metrics of course depend upon how we define H. In this investigation we will compare only regions. In order to compare regions meaningfully, a common frame of reference is required. There are two possible frames of reference: physical structure, and logical structure. For physical structure, we can compare based on geometry in the input space (e.g. using bounding boxes). For logical structure, we can compare the sets of input regions known to be contained by regions of a logical type. In Chapter 5 we will use the metrics above to compare regions based on the words that they contain; words will be part of the input to strategies, providing a common frame of reference.

Historical recall and precision could be adapted to use weights for partial matches as has been done for conventional recall and precision (see Section 2.6.1). In this investigation we will be using only the basic forms of historical and 'conventional' recall and precision as defined above, without weights.

Historical recall, and *historical precision* provide new information for 'error analysis,' when a strategy designer determines the type, number, and cause of mismatches between accepted interpretations and those interpretations generated by a strategy. These metrics were not considered previously in the literature, perhaps because systems were not constructed in a way allowing them to be easily observed.

4.9 Visualizing and Creating Interpretations

Figure 4.9 presents utilities for translating interpretation graphs to GraphViz[35] (*graph2dot*) and Xfig[101] (*graph2fig*) formats for visualization, and for translating Xfig files to interpretation graphs (*fig2graph*). We have used TXL for each translation, as these are all structured text encodings. We can illustrate logical structure using GraphViz, which produces graphs with nodes and edges as in Figure 4.14, or logical and physical structure using Xfig, where regions are placed overtop of a table image as seen in Figure 4.10. We describe the procedure for creating interpretations using Xfig in Section 4.9.2.

Each translation makes use of a simple text specification file, describing attributes for regions and relations. For the GraphViz (dot) translation, these include node shape and colour, and colour and line style (e.g. solid, dotted) for edges. In the specification files for the Xfig translations, additional *layer* attributes are defined. These attributes determine the Xfig layers within which a region or relation type is embedded. When an Xfig file is viewed, layers may be easily removed from view or added by manipulating a list of checkboxes in the Xfig interface (see Figure 4.10). When translating Xfig files to interpretation graphs, the layer attribute is used to assign types to objects provided in the input. For a given table model we currently use a single specification file for all Xfig translations (*graph2fig, fig2graph,* and *frameCompounds*).

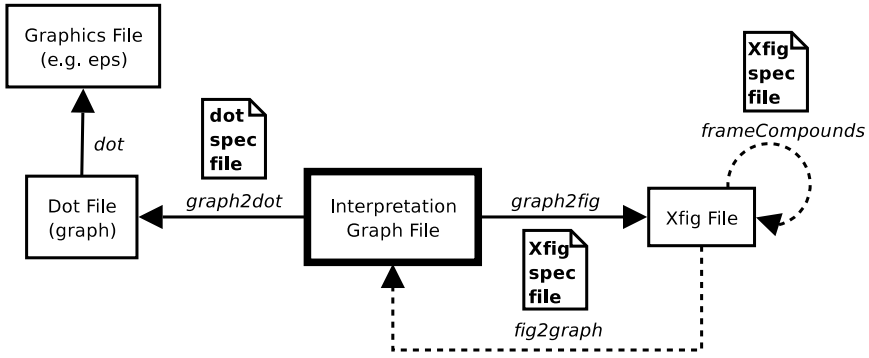


Figure 4.9: Interpretation Graph Translation Utilities. *graph2fig* and *graph2dot* translate interpretation graphs to Xfig and dot (GraphViz) formats respectively. *frameCompounds* and *fig2graph* are used for manually constructing interpretations. All translations use an attribute specification file.

4.9.1 Visualization

GraphViz provides a number of graph layout algorithms. In this document we use the *dot* program, which lays out nodes hierarchically based on edge structure. A specification file as described above is used by the *graph2dot* utility which translates an interpretation graph to a 'dot' graph file. The 'dot' graph file can then be translated to an image format by *dot* (e.g. to an encapsulated postscript file). The translation of the interpretation graph is straightforward: region type entries are mapped to nodes of the graph, and relation edges are mapped to edges of the graph, each with the attributes specified in the dot attribute file.

A similar translation program (*graph2fig*) is used to translate interpretation graphs to Xfig files. The translation procedure is similar to that for dot, with region type entries mapped to boxes or lines, and relation edges mapped to arrows between box and line centers. There are additional attributes in the Xfig specification file that indicate layers for regions and relations (see Figure 4.10). Xfig is a vector drawing

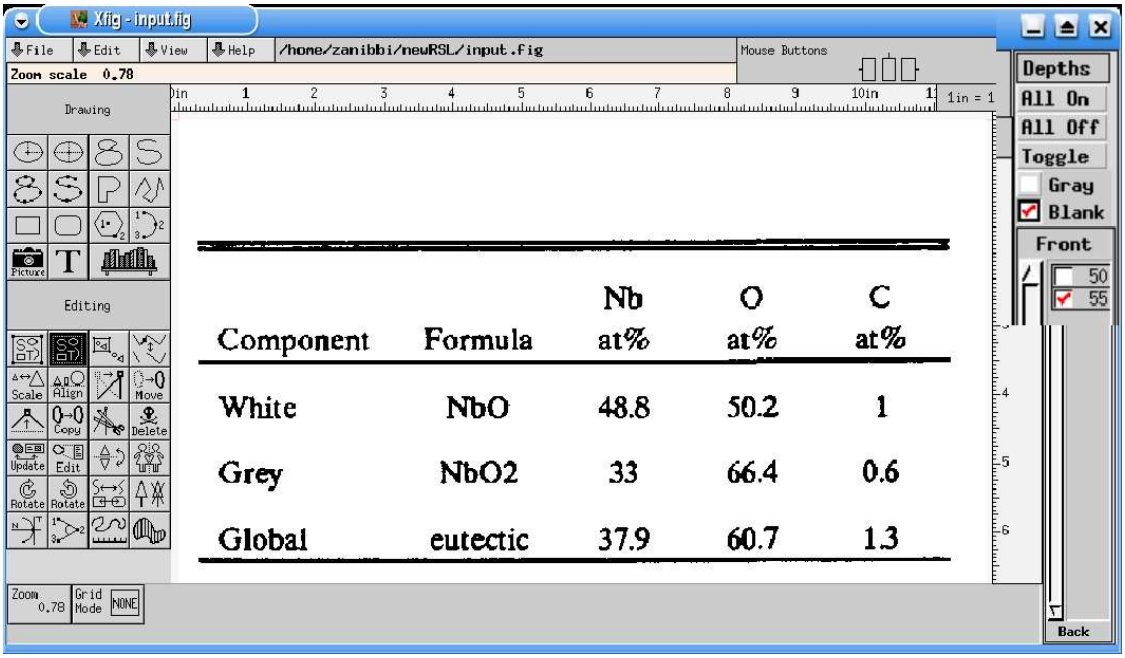
environment in which the region and relation objects can be moved, deleted, and edited as needed. We have found being able to manipulate the interpretation visually and view different image layers to be very helpful in interpreting strategy results, particularly when debugging.

4.9.2 Manually Creating Interpretations

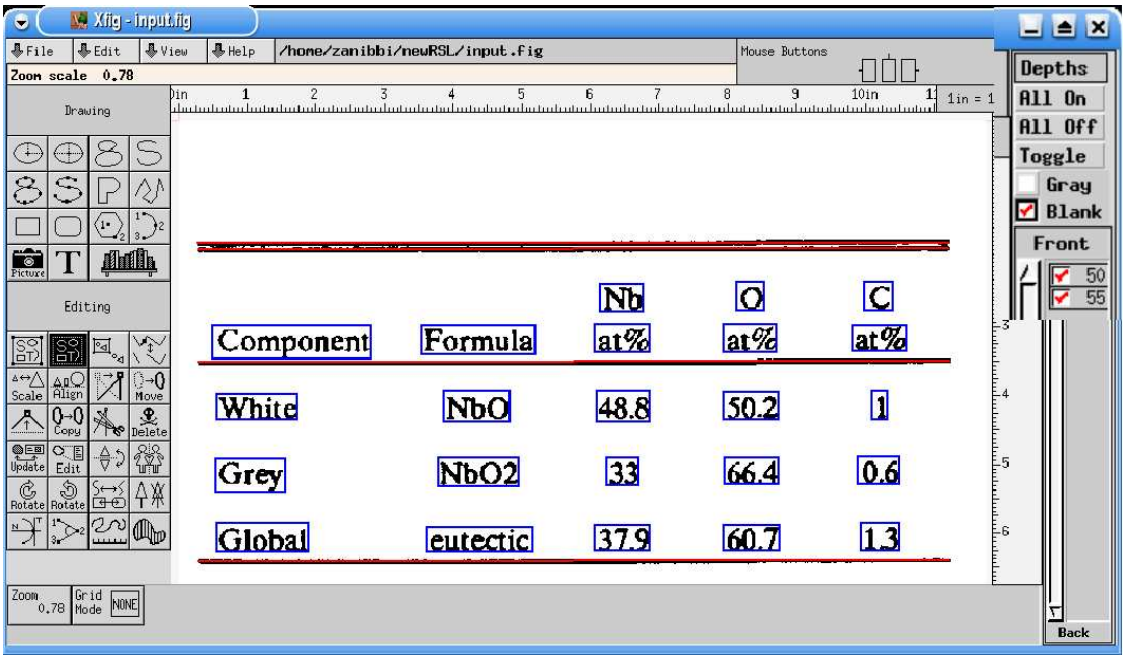
In this section we briefly describe the process for manually creating interpretations using Xfig, *frameCompounds* and *fig2graph*. The approach taken here was informed by time spent using the Illuminator[95] tool for creating document interpretations. We chose to use Xfig because it is more widely used and has more editing tools than the Illuminator environment.

The primitive regions in the table model are drawn as boxes or lines, as shown for words and lines in Figure 4.10b. Words may be combined hierarchically into other region types using the Xfig grouping operation (producing Xfig 'compounds'). All objects in Xfig have an associated comment that may be edited using a pop-up menu. We use these comments to define the characters in word regions and the types of region groupings (see Figure 4.11). Xfig visualization of groupings is quite subtle, as only control points are shown. The program *frameCompounds* creates bounding boxes for compound regions to make them easier to see (see Figure 4.12). *frameCompounds* may be used multiple times if necessary.

Because Xfig's region grouping operation is hierarchical, regions may belong to only one parent compound (i.e. region containment is defined by a tree). This makes



(a) Image layer



(b) Image and primitive regions (words and lines)

Figure 4.10: Object Layers in Xfig. In (a) only the image layer is visible, while in (b) both the image layer and a second layer containing manually placed word boxes and lines are visible. In both screenshots the checkbox list controlling visible layers in the upper-right hand corner has been magnified.

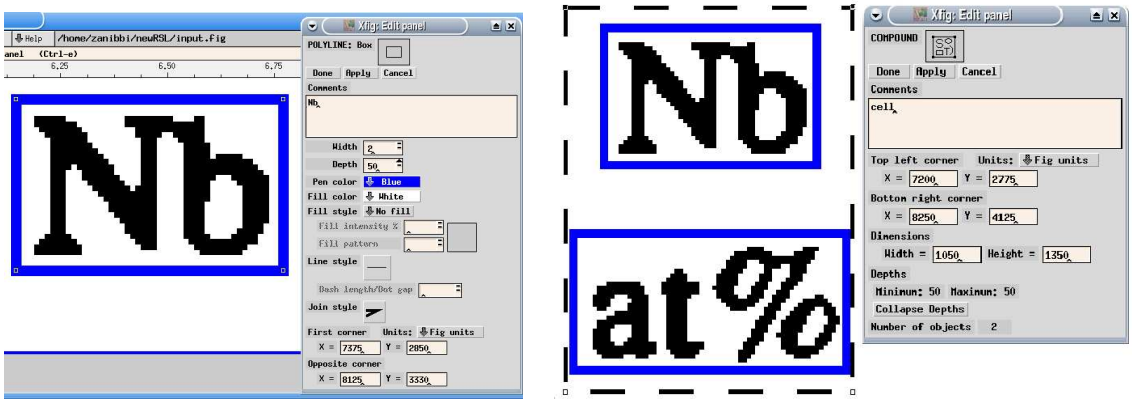


Figure 4.11: Creating Word Characters and Region Types in Xfig. On the left we show a *word* region drawn as a box having its characters entered manually. On the right, we show a cell region containing two words being labelled with its type. The cell region is shown with a dotted line. Within Xfig, only the small control points at the corners of the compound are visible.

it impossible to group cells into rows and then group cells into columns, for example. To overcome this, we created a method for defining groups by drawing polylines and then determining which regions are intersected by the corner and end points (the 'mouse-clicked points') in the line. If multiple regions are intersected by a point, we assume that the smallest area intersected by the point is the selection target. As an example, consider again Figure 4.12 where we have used the grouping operation to define rows of cells. To define a column of cells, we draw a polyline in the appropriate layer, insuring that we intersect cells in the column with points in the line, as shown in Figure 4.13a. We can then use the *fig2graph* utility, which will create the columns within an interpretation graph. The result of running *fig2graph* on 4.13a is shown in Figure 4.13b (produced by running *graph2fig* on the *fig2graph* output).

Relations are defined in a similar manner: regions intersected by the corner and end points of a polyline are used to represent edges of a relation type. In Figure 4.13a

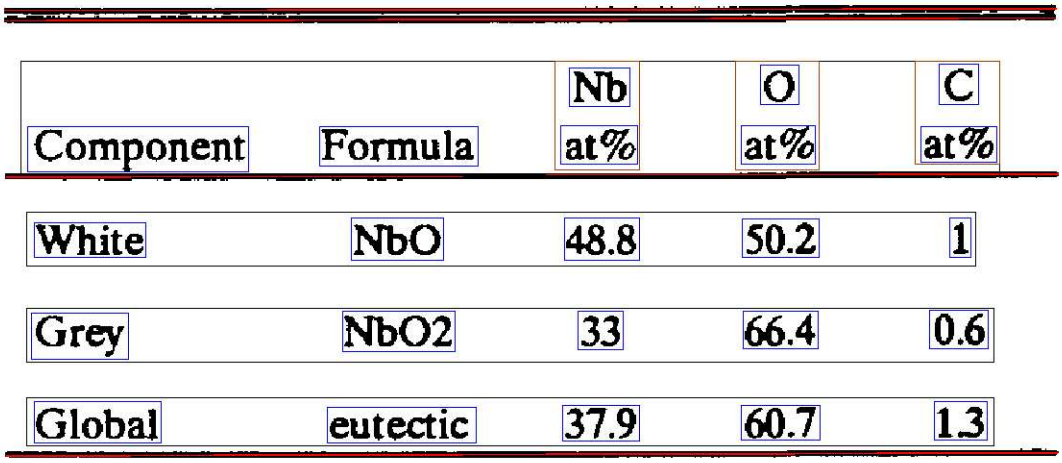
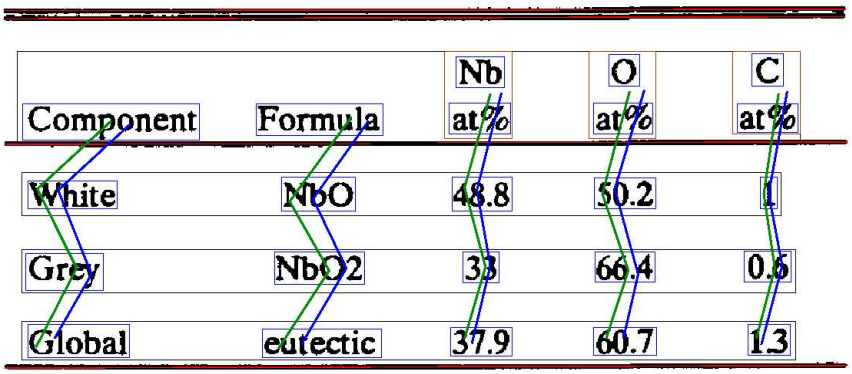


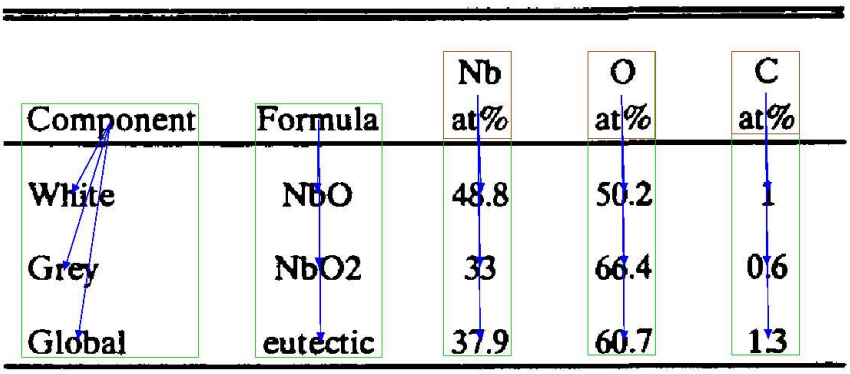
Figure 4.12: Framing Cells and Rows. Suppose that cells and rows are defined from the word regions shown in Figure 4.10b by hierarchically grouping regions and then labelling them in the manner shown in Figure 4.11b. The resulting file is passed to *frameCompounds*, which visualizes the bounding boxes of these regions as shown above. In this example three cells are defined in the boxhead, and four rows. For convenience, we assume that words not in cell compounds are also cells.

we draw lines to define the indexing relation on cells of the table from headers to data. In Figure 4.13b we can see arrows representing the defined indexing structure. The Xfig specification files have an additional attribute available for relations, to allow hierarchical relationships to be defined using a single line. Lines with this attribute define a set of edges, with a single parent (the region intersected by the first point) and a set of children (all regions intersected by remaining points). The cell indexing relation has this attribute in the specification file used to generate Figure 4.13b, producing the hierarchical relations from headers to data cells within columns.

fig2graph automatically defines the region containment relation ('contains') from the defined compounds and line intersections defining region types. This is illustrated

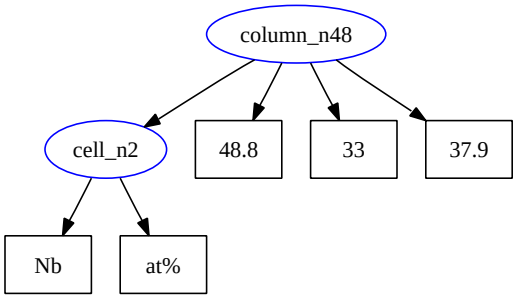


(a) Lines for creating columns (light) and indexing relation edges (dark)

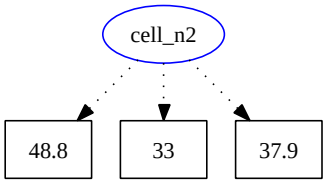


(b) *fig2graph* Column and index edge output. Here cells containing multiple words, columns, and indexing relation edges are shown; all other region and relation types are embedded in hidden layers.

Figure 4.13: Using Polylines to Define Regions and Relations. *fig2graph* allows regions and relations be defined from regions intersected by corners of a polyline.



(a) Region structure (contains relation)



(b) Indexing structure (indexing relation)

Figure 4.14: Interpretation Graph to 'dot' Conversion (*graph2dot*). Here the region (a) and indexing (b) structure for the middle column in Figure 4.13 are shown.

for the middle column of Figure 4.13b in Figure 4.14a. Also shown in Figure 4.14b is the indexing structure for the same column. Both of the graphs in Figure 4.14 were produced by GraphViz (dot). The dot encoding was created by applying *graph2dot* to the interpretation graph generated by *fig2graph*.

4.10 Visualizing Table Models

The RSL syntax is designed to capture table models used in strategies. In this section we describe a pair of simple methods for summarizing table models using graphs which may be visualized. The first is for summarizing region and relation structure in a table model, and the second is for describing additional dependencies between regions, relations, and recognition parameters in a table model.

We can produce graphs summarizing region and relation structure from the scope and result types of RSL inferencing operations (see Section 3.5.3). Currently we

produce a summary using the function *getModelStructure()*, defined below. *getModelStructure()* takes an RSL program R as input and returns a labelled graph S:

getModelStructure(R)

1. Let S be an empty graph
2. Add all region types in R as nodes to S
3. For each inference operation I in R

   Update S based on the inference type of I:

   (a) *segment, resegment*: add edges to S from each scope type (child regions) to each result type (type of segment)

   (b) *relate* : if there are two scope types, then define an edge from the first type to the second in S, labelled by the relation type. If there is one scope type, then define a self-edge labelled by the relation type.

   (c) *classify*: define an edge from each scope type (regions to be classified) to each result type (set of classes) in S with label 'class'

   (d) *merge*: define a self edge for the scope type in S

   (e) Other Types: do nothing
4. Return S

Figure 4.15 visualizes the output of *getModelStructure()* for the RSL program shown in Figure 3.3.

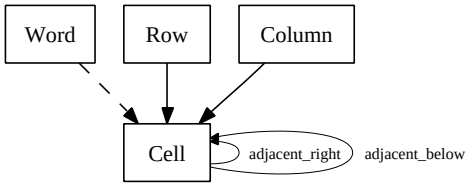


Figure 4.15: Region and Relation Structure Summary for Table Model in Figure 3.3. In this table model, *Row* and *Column* regions contain *Cell* regions. The dashed line indicates that *Word* regions may also be classified as *Cell* regions. The thinner edges indicate the spatial relations defined on *Cell* regions, *adjacent_right*, and *adjacent_below*.

Note that the summary produced by *getModelStructure()* is missing some information; it describes only the types of regions contained by each region type, the relation types and the region types on which relations are defined, and classification relationships between region types. A complete analysis would involve additional steps. For example, it may be that in an RSL program *word* regions are segmented into *row* regions, then all *row* regions are rejected, and later *cell* regions are segmented into *row* regions. In the region syntax summary produced by *getModelStructure()*, there would be edges from *row* to *cell* and *row* to *word*, but *word* and *row* regions never actually co-exist in *row* regions for this example strategy.

The RSL observation specifications and recognition parameters used in external functions provide additional information which can also be described using graphs. We use *getModelDependencies()* to produce this summary, D:

getModelDependencies(R)

1. Let D be an empty graph
2. Add all region types, relation types, and recognition parameters in R as nodes to D

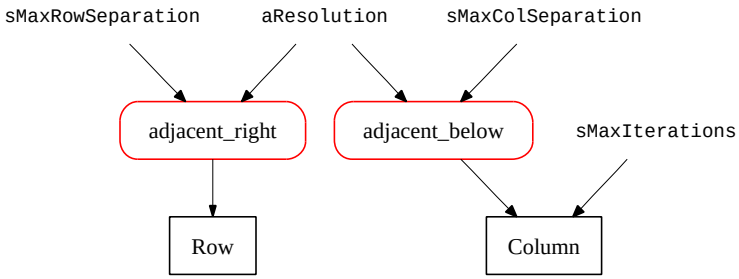


Figure 4.16: Dependency Summary for Strategy in Figure 3.3. Parameters such as *aResolution* appear without surrounding boxes, relation types appear in rounded boxes, and region types appear in rectangles.

---

3. For each inference operation I in R

   Update D based on the inference type of I:

   (a) *classify, segment, resegment, relate, create, replace, merge, reject*: add edges to D from each region and relation type in the observation specification and any parameters passed to an external function to the result type(s) of the function

   (b) Other Types: do nothing

4. For each adaptation operation A in R

   Add edges to D from each observed region type, observed relation type, and parameter passed to an external function, to each adapted parameter altered by the operation

5. Return D

Figure 5.3 visualizes the graph constructed by *getModelDependencies()* for the strategy in Figure 3.3.

4.11 Summary

This chapter outlined an implementation of the Recognition Strategy Language (RSL) in which RSL source code is translated to programs in the TXL programming language. We described the RSL compiler, the RSL to TXL translation procedure, and the data and libraries that implement the functionality of RSL. Tools for visualizing interpretations and table models, manipulating the last 'inference time' of an interpretation, and for manually creating interpretations were also described. Finally, using the available histories in interpretation graphs, we defined two new recognition metrics in Section 4.8: *historical recall*, and *historical precision*.

In the next chapter we describe implementing and comparing two table structure recognition strategies from the literature. The metrics and tools described in this chapter will be used in the next chapter.

Chapter 5

Specifying and Comparing Strategies

In this chapter we describe the implementation of two table structure recognition algorithms using RSL specifications, and then demonstrate comparisons of RSL output through a simple table cell recognition 'game'. The recognition algorithms implemented are those of Handley[43] and Hu et al.[54, 55]. We chose these algorithms because they are feed-forward recognition strategies (and are thus expressible in RSL) and are described in sufficient detail in the literature to be reproduced from their written descriptions. Both analyze table structure from a list of word regions (and in the case of Handley's algorithm, lines as well).

These structure recognition strategies both produce a single interpretation at termination, and while it would not be difficult to alter either strategy to produce multiple interpretations within RSL, we leave this for the future. As given, these systems are already fairly complex, as can be seen in the dependency graphs for our RSL specifications shown in Figures 5.3 and 5.4.

In Sections 5.1 and 5.2 we describe the RSL implementations of the Handley and Hu et al. algorithms, respectively. In Section 5.3 we describe graph-based summaries of table model structure and observation dependencies in RSL strategies. Next, we elaborate on using games to characterize table recognition in Section 5.4, and then describe a simple cell recognition game used to compare the outputs of the two algorithms in Section 5.5.1. We take advantage of the hypothesis histories maintained by the RSL language to observe some new and useful metrics, *historical recall* and *historical precision*, and to directly trace hypotheses back to the inferences that created them.

In our discussion we assume that tables passed to the table structure recognition algorithms have been properly segmented; that is, that the region containing the table has been properly identified. We also assume that all lines passed to the algorithms are part of the segmented table region.

5.1 Handley's Structure Recognition Algorithm

Handley's algorithm[43] may be understood as a geometry-based cell topology recognition technique that combines top-down and bottom-up methods. The central operations of the strategy are defining regions using minima in bounding box histograms (a 'top-down' technique), and merging operations that combine cells, rows, and columns 'bottom-up'. Lines are an important aspect of Handley's algorithm: regions are never merged across lines, underlines are detected and filtered, and fully-ruled tables in which all cells are bounded by lines are handled by special processing. Our RSL specification for Handley's algorithm is provided in Appendix C. The external functions called from the specification have been implemented using approximately 5000

lines of TXL source code (including comments and newlines).

Our implementation of Handley's algorithm differs from the original in some minor ways, which we will describe here. One simplification is that we assume lines provided in the input are part of the table region as either an underline or ruling line; the original algorithm does not make this assumption. Also, the expansion of regions to bounding lines is a common action in the Handley algorithm. We have replaced this with an equivalent representation using relations to represent the line adjacencies of regions (using the relations *adjacent_left*, *adjacent_right*, etc.). These relations are easier to express in RSL; currently there is no operation to directly revise the bounding box of an existing region.

Handley's algorithm is defined as a sequence of rules organized into sixteen steps. These steps appear in the main strategy function, below the **recognition parameters** section of the strategy. The sixteen steps are applied, and then generated interpretations are accepted. As indicated earlier, Handley's algorithm produces single interpretations.

In the **recognition parameters** section of the RSL strategy in Appendix C, we have tried to name parameters based on their function, and present them in the order that they are first used. In some cases such as for *sColumnMinGap*, we have taken a recurring literal value used in similar analysis contexts and represented it as a single parameter.

Handley's algorithm may be broken down into four main stages:

1. Steps 1-2: Initialization and line analysis (including underline detection)

2. Step 3: Analysis for fully ruled tables, where all cells are surrounded by lines. If a fully ruled table is detected, the body of *fullyRuledAnalysis* is applied, the

table region is classified as a fully ruled table, cells sharing line boundaries are merged, and then the resulting cells and separators are returned.

3. Steps 4-9: Cell, column, and row structure are refined.

4. Steps 10-12: Cells in the boxhead of the table are merged according to a number of heuristics intended to reflect popular table styles.

5. Steps 13-16: Row structure is refined, and the positions of column and row separators are make explicit through the use of 'invisible' separators. A 'fully-ruled' table is returned, in which all rows and columns are separated by input lines or 'invisible lines.'

One of the most frequently used strategy functions in the Handley RSL strategy is *analyzeLineAdjacency*. This is used to define line-cell adjacencies. Each time *analyzeLineAdjacency* is called, the current set of line-cell spatial relations are rejected, and then redefined based on current line and cell positions.

In the function *fullyRuledAnalysis* used to detect fully-ruled tables, we see an application of the *for interpretations* operation. In this case it acts as a guard, to insure that only tables that are fully ruled have *fullyRuledAnalysis* applied to them.

5.2 Hu et al.'s Structure Recognition Algorithm

Hu, Kashi, Lopresti, and Wilfong[54, 55] proposed an algorithm for recognizing the cell topology and indexing structure of tables in ASCII text files. An example of recognized indexing structure is shown in Figure 5.1. The algorithm is unique in the literature in that it applies a hierarchical clustering algorithm[32] to the horizontal

Table 49.—Average values for bulk density, grain density, and total pore space of gray dacite from the lateral-blast deposits and of pumice lapilli from pyroclastic-flow deposits of Mount St. Helens

Type of deposit	Bulk density		Grain density		Total pore space (percent)
	Mean (g/cm^3)	No.[1]	Mean (g/cm^3)	No.[1]	
Lateral blast, May 18--------[2]	1.66	26	2.52	3	36
Pyroclastic flow, May 18-------	.74	8	2.55	3	71
May 25-------	.95	2	([3])	0	63
June 12------	1.08	10	2.53	3	57
July 22------	.88	11	2.55	1	65
August 7-----	1.02	12	2.61	3	61
October 16-18	1.12	12	2.65	5	58

[1] Number of determinations.
[2] Data from Hoblitt and others (this volume).
[3] Grain density (Dg) not determined; total pore space calculated using Dg=2.60.

Figure 5.1: Recognized Indexing Structure for Table from UW-I file a038. Here the column indexing structure is indicated by arrows, while the cells in the leftmost column are assumed to be row headers (for legibility we have omitted the boxes for rows here). Note that the column headers in the bottom of the boxhead label all the cells in the column intersected by the indexing arrow, and not any individual cell that may happen to be intersected. The separated header cell in the rightmost column is an artifact of the projection-based textline location method used in our implementation. Originally the Hu et al. algorithm was intended to operate in text files.

spans of words to locate columns in a table region. The algorithm also makes use of the *content* of words, i.e. their contained text characters. Our RSL specification for Hu et al.'s algorithm is provided in Appendix D. The external functions called from the specification were implemented in 3000 lines of TXL code (including comments and blank lines).

As this algorithm was originally designed to work with ASCII text, we needed to make some alterations to allow it to work with lists of words described by bounding boxes. The most essential addition is the strategy function *createTextlines* in the RSL specification for this algorithm. Textline regions are first approximated using a horizontal projection of word bounding boxes onto the Y-axis. The parameter *sThreshold* is used to filter projection intervals with fewer than *sThreshold* boxes associated with themselves. Pairs of overlapping projected intervals are merged if both have a given percentage of their vertical span overlapped by the other region (specified by *sOverlap*). Words are assigned to one of the remaining intervals which intersects their Y-center point, defining the textline with which a word is associated. Naturally, in an ASCII file, correct textlines come with the data.

Other changes included redefining parameters to represent distances in space where formerly they were numbers of rows or columns in an ASCII file. For example, *sMaxBoxheadHeight* (the maximum number of text lines in a boxhead) was formerly '5 lines,' and is now a distance in millimetres.

The main steps in this algorithm are represented by the names of strategies called from within in the main strategy function. These steps are:

1. Create text lines (as described above)

2. Determine column structure, using hierarchical clustering of word regions followed by cuts of the cluster tree, which defines the columns of the table. We have implemented the clustering using a simple recursive strategy function *buildClusterTree*: the *for interpretations* operation is used in this function to define the base case, when the cluster tree is complete.

3. The boxhead is located by assigning lexical types (alphabetic/ non-alphabetic) to words and textlines, and then checking consistencies between lexical types.

4. The indexing structure from column headers to columns is defined.

5. Row structure is determined by merging textlines based on the dominant lexical type (alphabetic/non-alphabetic) of words and columns, the column location of words within textlines, and the vertical distance to adjacent textlines.

6. Cells are then defined by words shared between rows and columns, and are each assigned to the appropriate row and column.

5.3 Summary Graphs for RSL Strategies

The RSL syntax was designed to capture table model structure and observations between logical types directly within the operation syntax. The basic operation scope and result types capture region and relation structure, while dependencies arising from observations made for inferences are described by observation specifications (Section 3.7 describes observation specifications).

The region and relation structure of the table models for the Handley and Hu et al. strategies are shown in Figure 5.2, as defined by the inferencing operations in the

RSL specifications. The algorithm *getModelStructure* for generating the structure graph is defined in Section 4.10. Presented in the graphs are all the region types of the models represented as boxes, and relations on regions are represented by labelled edges (e.g. *adjacent_left* for Handley's algorithm, and *indexes* for Hu et al.'s). Region classifications are indicated using dashed lines: for example, an *hline* (horizontal line) may be classified as an *underline* in Handley's strategy, but not the reverse.

The graphs in Figure 5.2 describe the types of regions that may be classified, segmented, and related to one another. However, these graphs do not capture the *dependencies* that exist between a region or relation type and the region and relation types observed when inferences concerning the type are made. Additionally, we wish to know how parameters passed to external functions affect logical types directly or indirectly.

This type of *observation dependency* graph is shown for Handley's algorithm in Figure 5.3, and Hu et al.'s algorithm in Figure 5.4. The algorithm for constructing observation dependency graphs, *getModelDependencies*, is defined in Section 4.10. Note that dependencies are defined by *incoming* edges: a logical type (region type or relation) depends on parameters, relations and region types from which incoming edges originate. By tracing dependency edges, we can determine what parameters and model types may have indirect effects on a type.

These graphs provide useful but informal summaries. They are essentially primitive static analyses[33] of an RSL strategy. For example, the current region and relation structure graphs can describe the types a region may contain, but not whether a region may contain both types simultaneously. More sophisticated summaries and

analyses would be desirable. As an initial starting point, these informal graphs provide a useful high-level view of RSL strategies.

5.4 Ground Truth and Imitation Games

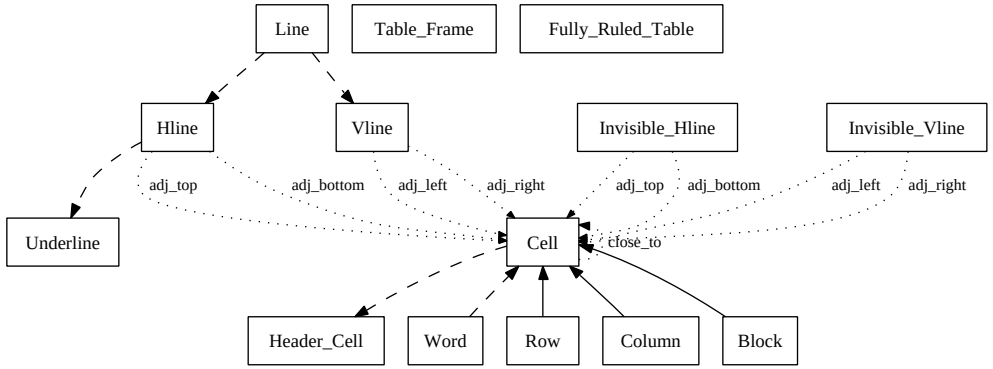
In Section 3.2 we characterized table recognition problems as a simple class of imitation games where algorithms try to best imitate a set of accepted table interpretations generated using some known procedure. In this Section we further discuss a potential role for games in defining table recognition.

Consider the following two problem definitions for table structure recognition. The first is intended to represent a view employing the notion of *ground truth*, while the second expresses the game-based view proposed in this dissertation.

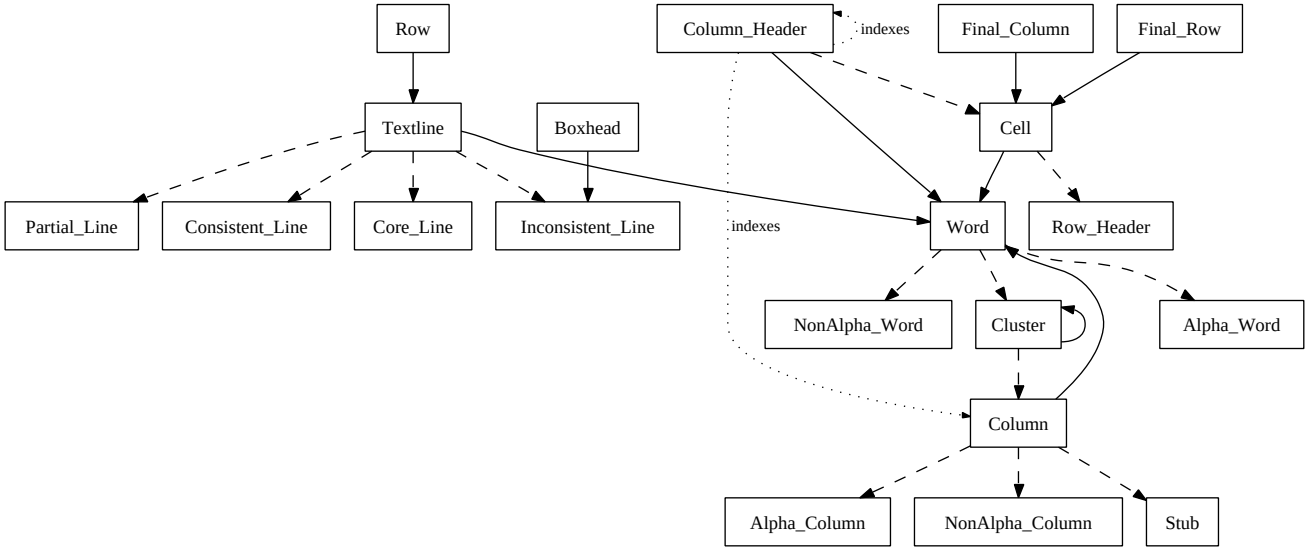
1. **Problem(G):** assume the existence of a (possibly infinite) set of pairs $G = \{(t1, \{i1_1, ..., i1_n\}), (t2, \{i2_1, ..., i2_n\}), ...\}$, where each pair (t, i) describes a table and the corresponding set of correct interpretations for that table. The problem of table structure recognition is to define a recognition function r that maps each $t \in \{t1, t2, ...tn\}$ to the corresponding set of interpretations i in G. We refer to G as *ground truth*.

and

2. **Problem(P,S,D,I):** let P be a population of tables $P = \{t1, t2, ...\}$ and P_s the set of all subsets of P generated by a sampling method S. Further, let $A = \{(t1, \{a1_1, ..., a1_n\}, (t2, \{a2_1, ..., a2_n\}), ...\}$ be the set of *accepted interpretations* produced by a known procedure I. The



(a) Handley's Algorithm



(b) Hu et al.'s Algorithm

Figure 5.2: Region and Relation Structure for Implemented RSL Strategies. Solid lines indicate region membership, dashed lines indicate classification relationships, and dotted lines represent relations on regions.

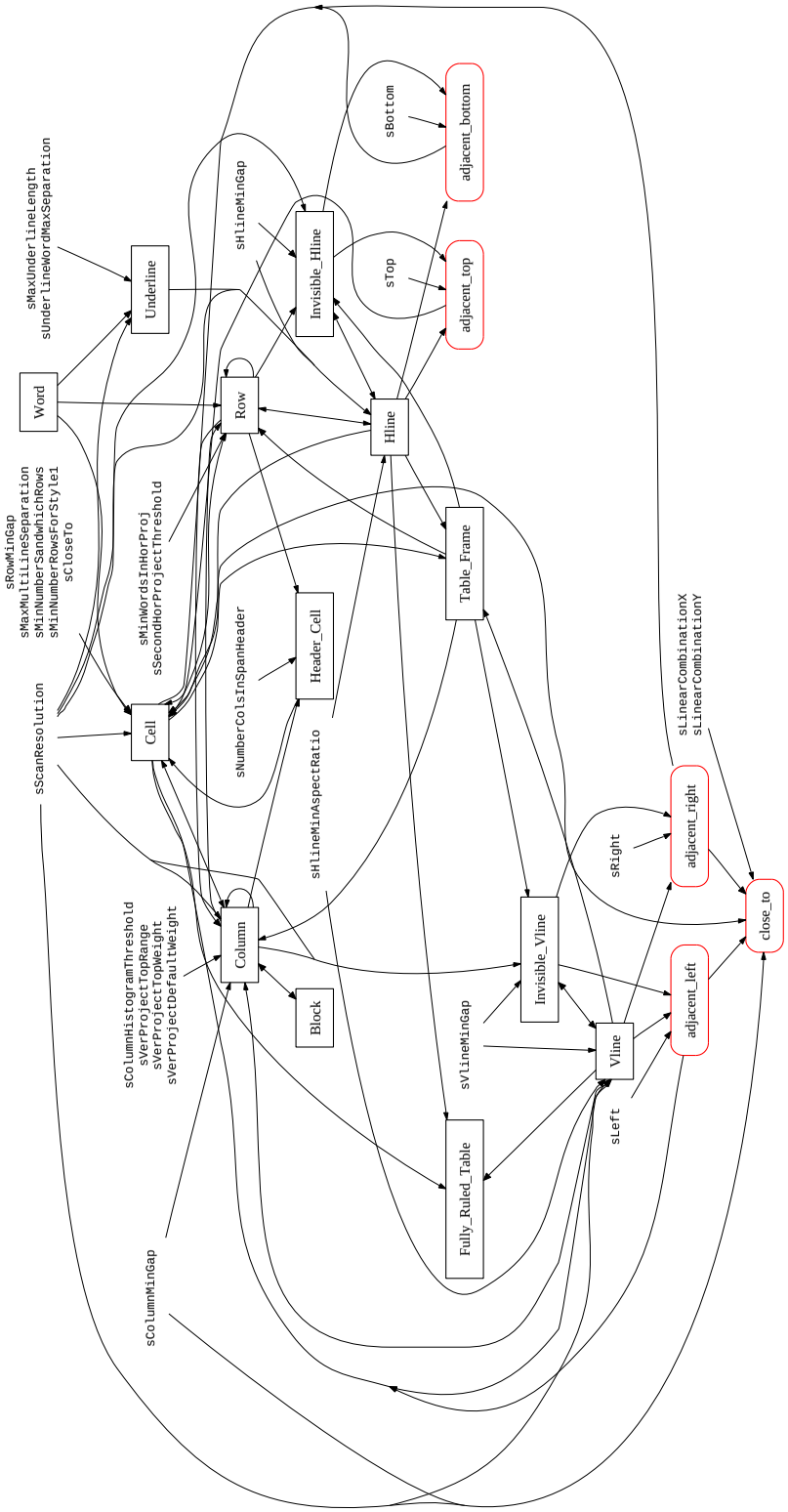


Figure 5.3: Observation Dependencies for Handley RSL Strategy. Dependencies are indicated by incoming edges. Relations are represented by boxes with rounded corners, and parameters appear without boxes.

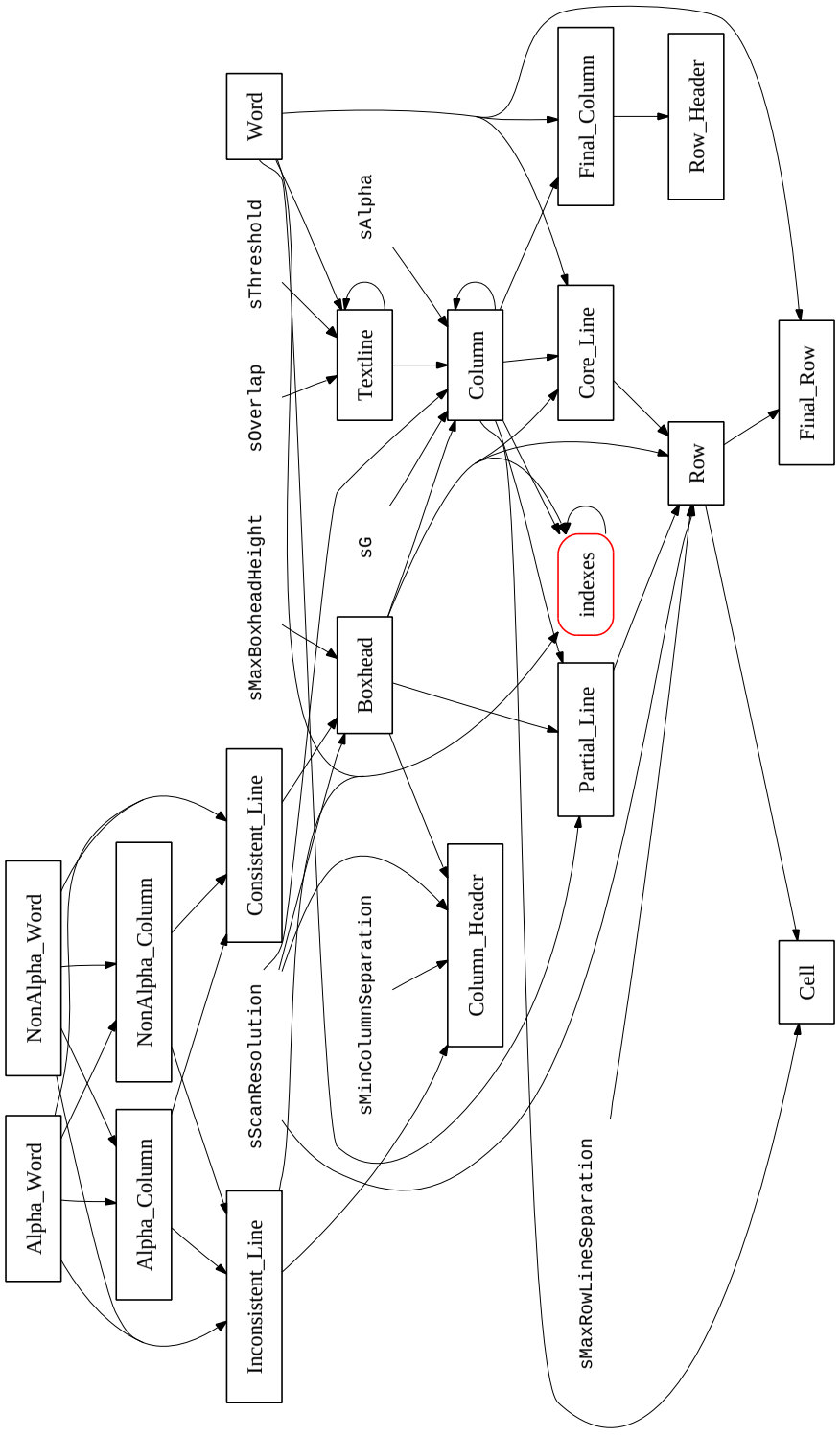


Figure 5.4: Observation Dependencies for Hu et al. RSL Strategy. Dependencies are indicated by incoming edges. Relations are represented by boxes with rounded corners, and parameters appear without boxes.

> problem of table structure recognition is to define the recognition function r that minimizes a distance metric D measured between $R = \{(t1, \{r1_1, ..., r1_n\}), (t2, \{r2_1, ..., r2_n\}), ...\}$, the recognized table structures produced by r, and the corresponding interpretations for each $t1...tn$ in A.

where in practice I is nearly always one or more persons.

The first problem definition supposes that correct interpretations for table structure actually exist[1], and that the goal for table structure recognition research is to eventually reveal G, thereby solving the problem. However, recent developments in the literature[51, 73] seem to indicate that for table recognition at least, G may not exist. Tables are a part of language; they are used like language, with dialects, personalizations, shorthands, and evolution into new forms and functions, guided by the needs of individuals. With that being the case, the probability of finding G seems unlikely indeed.

Further, the view expressed in the first problem definition makes evaluation and *comparing* recognition strategies extremely difficult. If evaluation is defined by the correspondence of a recognition function r's output to G, and G is unknown, then any evaluation or comparison of methods is made relative to a personalized and partial view of the assumed entity G. All evaluations and comparisons become approximate, relative to an unknown ideal. This is an endemic problem within document recognition[55, 72, 90], and more broadly within computer vision[49].

We propose that the second problem definition which describes the construction

[1]This analysis is inspired by Stewart Shapiro's history of the philosophy of mathematics[104]. Among other issues, Shapiro describes a long-standing argument about whether numbers actually *exist* or not.

of parameterized 'games' is more amenable to experimental investigation and theoretical analysis, and does not suffer from the ontological ambiguities of the first problem statement. This second definition also sheds some light on why the comparison of results within the literature is so difficult: many, perhaps even most systems utilize different assumptions (e.g. about the nature of G), effectively leading them to 'play different games' (address different problems). The second problem statement trades a general view of 'table correctness' for a less general problem definition dependent upon parameters of a game (specifically the population P, sampling method S, interpretation procedure I, and distance metric D). The ability to control the game parameters and explicitly fix the source of 'truth' (in our view, imitation targets) is invaluable for the empirical and theoretical study of *specific* table recognition problems.

It appears difficult to define a single process for the 'correct' automated interpretation of table structure. However, we feel that the discovery of interesting and useful properties of the table recognition problem or recognition techniques would be better supported by the fixed context of evaluation and interpretation provided by the game-view of table recognition, as expressed in the second problem statement above.

5.5 Illustrative Example: A Cell Imitation Game

To demonstrate the comparison of interpretations generated by RSL strategies, we pose a simple imitation game where strategies must imitate cell locations in tables as defined by the author. We define the game in the next Section, and then summarize the outcome of the game in Section 5.5.2. Finally, we analyze results using hypothesis histories, interpretation trees, and historical recall and precision in Section 5.5.3.

5.5.1 Game Definition

Figure 5.5 presents the simple game that we will use to illustrate comparing interpretations from RSL output. The four components (P, S, I, D) of this imitation game are:

1. **Domain Selection (P):** the domain is images of technical articles provided in the University of Washington English/Technical Document (UW-I)[89].

2. **Sample Selection (S):** the author chooses one 'simple' table and then four challenging cases (as pointed out by Hu et al.[51]) from the UW-I Database.

3. **Selected Interpretation Procedure (I):** the author first manually defines the locations of lines and words in each table image, which will be the input passed to the two recognition algorithms. The author then constructs a single 'preferred' cell interpretation for each of the five input tables using the tools described in Section 4.9.2, defining cells as sets of the defined word regions. These cell interpretations may be found in Appendix E.

4. **Distance Metric (D):** algorithms are compared by the number of times they match the author most closely (i.e. the winning algorithm must be closer at least three times).

   Cells were compared by the set of word regions that they contained (words regions are provided as input to the algorithms; see above) and we used exact matching for our metrics. For each input table, the harmonic mean of cell recall and precision was used as the distance metric D (see Section 2.6.1).

Please note that this game is only for illustration, and is not really a comparative analysis of the two algorithms in any general sense.

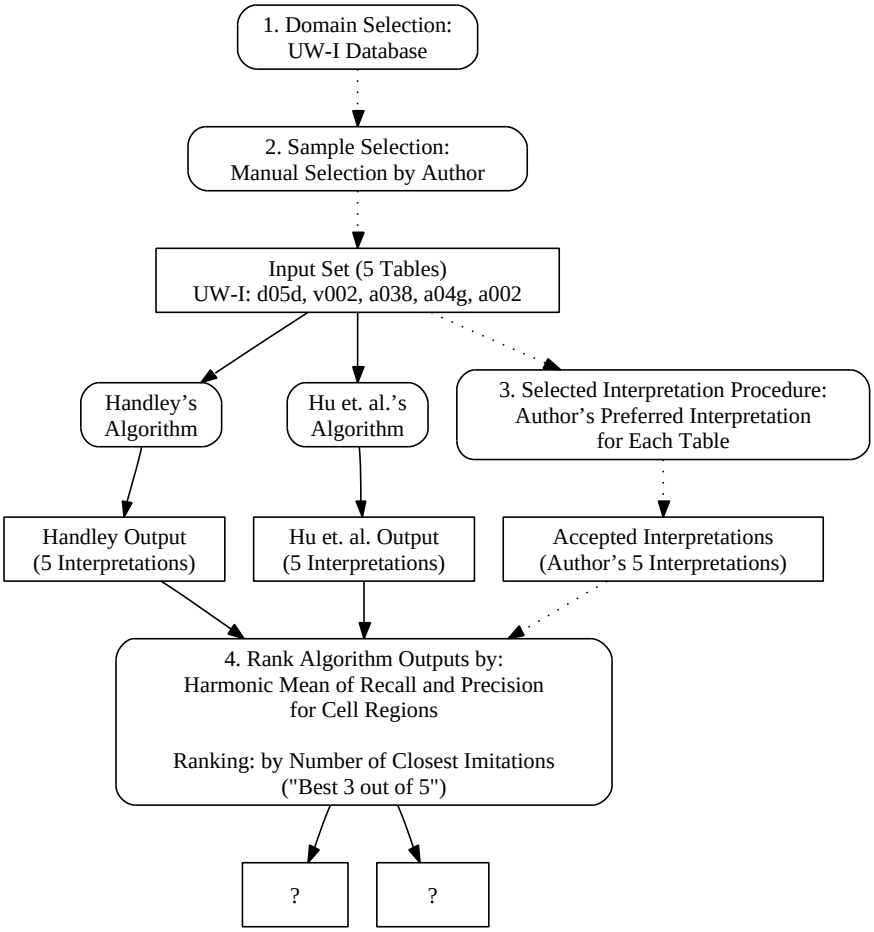


Figure 5.5: Cell Imitation Game. This modification of Figure 3.1 illustrates the cell imitation game described in Section 5.5.1. In this particular game, the author has selected five tables from the UW-I[89] database, and then defined a single 'preferred' interpretation for each.

5.5.2 Game Outcome

The cells produced by the author and both of the recognition algorithms may be found in Appendix E. Summaries of relevant metrics, including the harmonic mean of recall and precision used to decide the outcome for each table are provided in Figures 5.6 and 5.7. The winner in this instance was Handley's algorithm, as it matched tables d05d, v002, and a002 more closely. Hu et al.'s algorithm was the better imitator for the remaining two tables (a038 and a04g).

For table a002, the Hu algorithm matched none of the cells defined by the author, and thus had '0' values in all metrics for that table. Looking at the author and Hu et al. results for table a002 in the Appendix, we see that the Hu et al. split all of the author's defined cells, and so matches none of the author's cells. Many of the cells returned by the algorithm are entirely reasonable, corresponding to a 'finer-grained' cell definition. Using another person to define the accepted interpretations for cells (i.e. changing the procedure I to imitate) may have altered the game outcome.

Here are additional descriptive statistics for harmonic mean of recall and precision values observed in the game:

	Handley (%)	Hu et al. (%)
Harmonic Means		
Range	5.8 - 100	0 - 86.5
Mean	50.8	63.8
Median	25.0	79.1
Standard Deviation	45.5	36.1

Had the game been defined differently, so that the highest mean or median value for

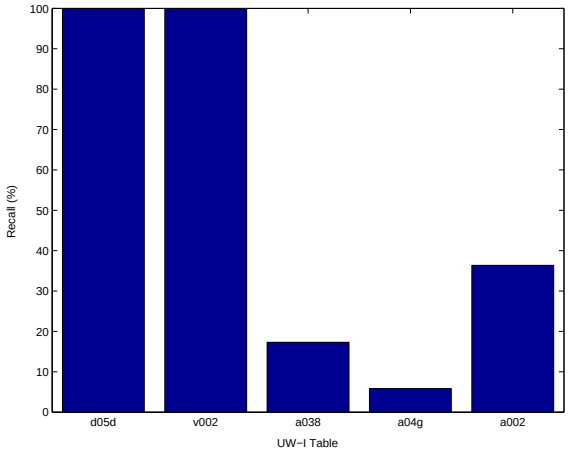
harmonic means was used as the distance metric, the algorithm of Hu et al. would have 'won'. For this sample, the performance of the Handley algorithm varied far more, as can be seen by simply visually comparing the bar charts in Figures 5.6 and 5.7.

5.5.3 Analysis Using Hypothesis Histories

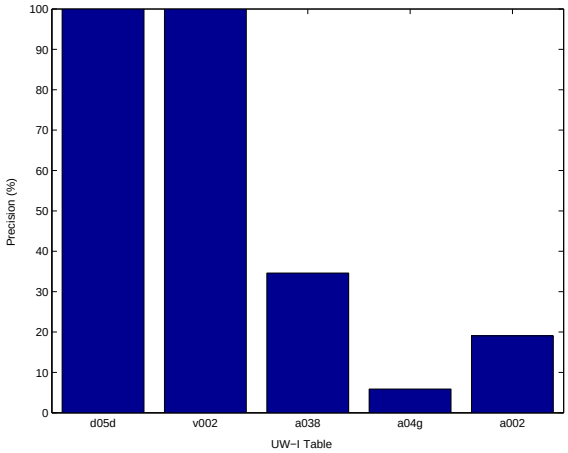
In this section we will analyze the results of the Handley algorithm, making use of historical recall and precision (defined in Section 4.8), the hypothesis histories annotated in the interpretations output by RSL strategies, and the interpretation trees produced by RSL.

In the context of our cell imitation game, historical precision describes the percentage of generated cell hypotheses that match a cell in the author's interpretation. It differs from 'conventional' (or *final*) precision because a strategy may reject one or more of these generated cells.

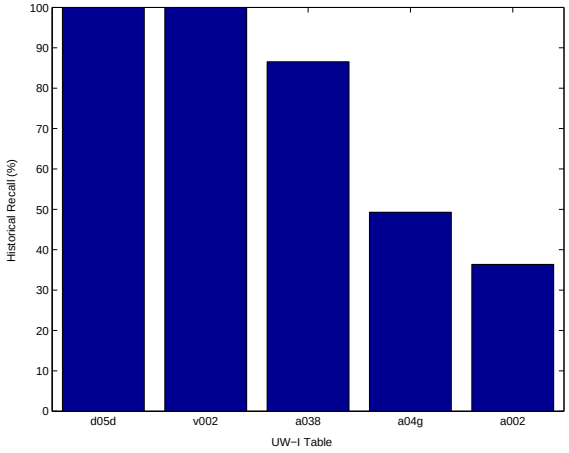
Note that for the Hu et al. algorithm cell hypotheses are never rejected, as cells are only ever created in that strategy, within strategy functions *indexAnalysis* and *bodyCellCreation*. Header cells are defined first, by cutting words within text lines in the detected boxhead of the table (see the strategy function *indexAnalysis* in Appendix D). Later body cells are defined from the intersections of words within detected columns and rows in the strategy function *bodyCellCreation*. Because cell hypotheses are never revised or rejected, the final and 'historical' cell hypothesis sets are the same. As a result, the historical and final recall and precision values are identical for the Hu et al. algorithm (see Figure 5.7).



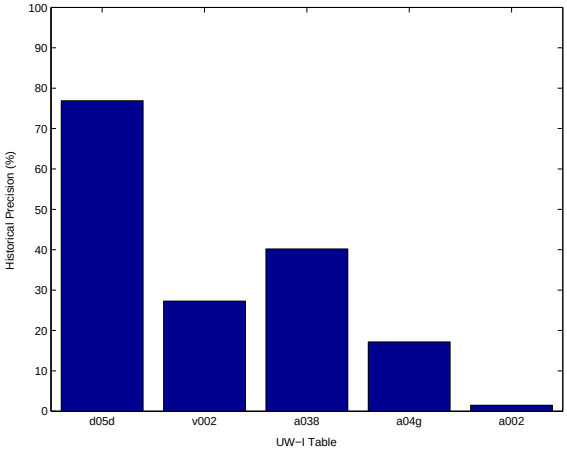
Recall (%)



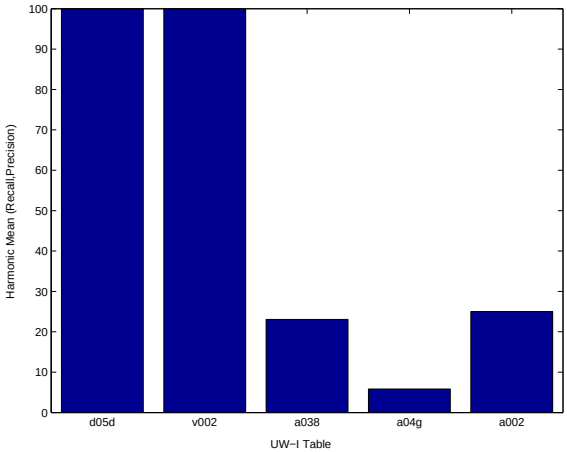
Precision (%)



Historical Recall (%)

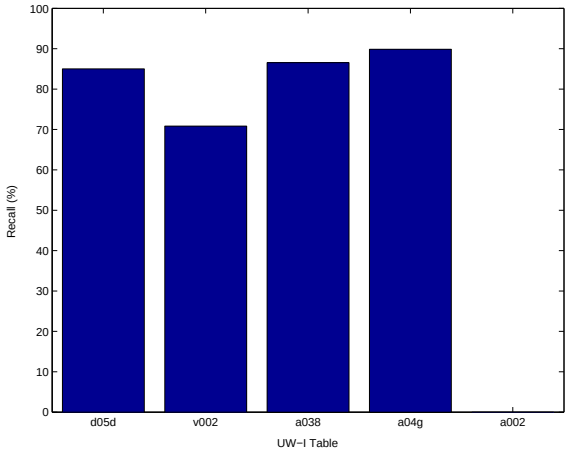


Historical Precision (%)

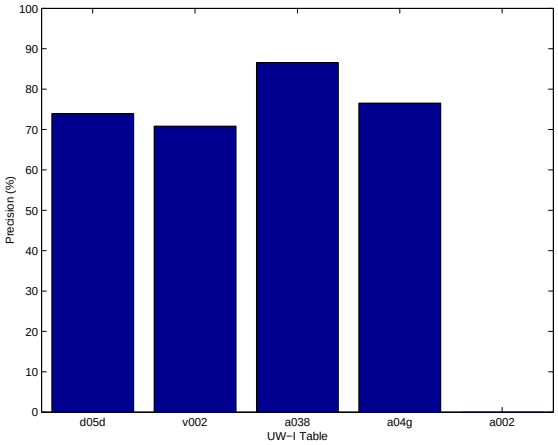


Harmonic Mean of Recall and Precision (%)

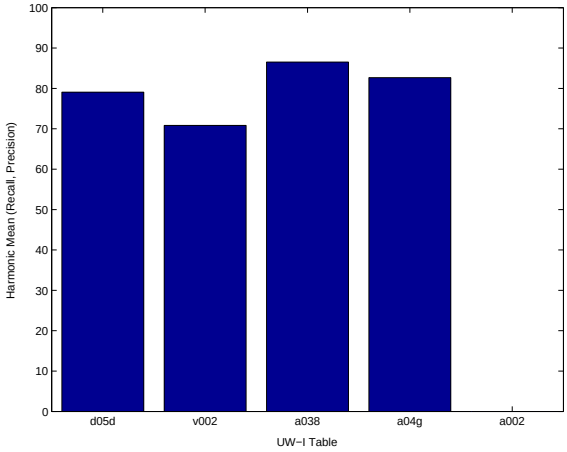
Figure 5.6: Cell Imitation Results for Handley's Algorithm.



Recall (%)



Precision (%)



Harmonic Mean of Recall and Precision (%)

Figure 5.7: Cell Imitation Results for Hu et al.'s Algorithm. Hu et al.'s algorithm only *generates* cells; none are ever rejected. As a result, the 'historical' and final cell hypothesis sets are the same, leading to identical 'historical' and final recall and precision values.

In contrast, the Handley algorithm is based around creating and revising cell hypotheses frequently, so historical recall and precision differ from final recall and precision, as can be seen in Figure 5.6. For the first two tables in the bar charts (d05d and v002) historical and final recall are identical, because all of the author's cells were located during the strategy's progress, and all were returned. For these same two tables, we see that the precision of the final hypothesis set was perfect (all returned cells matched one of the author's), but that the historical precision was lower; in fact, for the second table (v002), only 27.2% of the cell hypotheses generated were 'valid' cells. The algorithm successfully pruned all of the 'invalid' cells before returning the final result.

Historical precision is defined relative to the set of all generated cells. Depending on how hypotheses are rejected in the final result, the final precision may be higher or lower than the historical value. Unless all generated hypotheses match imitation targets, ideally an algorithm has a higher final than historical precision.

Historical recall on the other hand, is different: it is *never* smaller than final recall. Ideally, as for the first two tables, all cells matching the author's cells are returned. Otherwise, the difference between historical recall and final recall is the percentage of imitation targets that have been rejected falsely. We will now consider one such case, for the third table (a038), where almost 70% of the author's cells were proposed and then rejected by the algorithm.

Historical recall and precision are of course only descriptive statistics: they can tell us how many cells were incorrectly rejected, but not why or where. This is where the annotated hypothesis histories of interpretations produced by RSL strategies become useful (see Section 3.5.3). We can use the time stamps associated with each cell

region in the output to determine which of the RSL inferencing operations created, revised, and rejected cell regions. If we additionally make use of the interpretation 'time-reversal' function described in Section 4.7, we can recover the state of the interpretation graph at each of the relevant times, and then compute final and historical recall and precision metrics at each of these times. The result of performing these operations for the interpretation graph produced by the Handley algorithm for table a038 is shown in Figure 5.9. The list of operations from the RSL strategy corresponding to the inference times in Figure 5.9 are shown in Figure 5.8.

In Figure 5.9 we can see that the historical and 'conventional' recall for cells agree until inference time 35, when cells with the same assigned row and column positions are merged, and the recall value then drops substantially (by 50%). At inference time 47, things improve when cells that span rows at the top of the table are merged (by 1.9%: from the interpretation tree in the RSL log we can see that this represents exactly one new cell, the 'Total pore space (percent)' cell at the top of the rightmost column). The recall then only decreases from this operation. At inference time 51, the row header 'Pryoclastic flow,' is merged with cells below itself. Finally at inference time 83, cell locations are revised based on row and column separators defined in *step15* and at the beginning of *step16*, reducing the recall further.

In contrast, historical and final precision are fairly constant through the relevant inference times in Figure 5.9. The final precision is slightly lower than the historical precision. If fewer incorrect cells had been returned (i.e. more incorrect cell hypotheses had been rejected), the final precision would have been higher. For a designer the lesson here is that for this table and possibly ones like it for this game, a great number of incorrect cells are being generated over the course of the strategy (i.e. low

Time	Function	RSL Operation
0	(Input)	
1	*steps1and2*	**classify** {word} **regions as** {cell}
16	*step5*	**merge** {cell} **regions using** mergeClosure(sCloseTo) ...
35	*step9*	**merge** {cell} **regions using** mergeCellsAtSameGridPosition() ...
47	*step11*	**merge** {cell} **regions using** mergeSpanningCells(.....) ...
51	*step11*	**merge** {cell} **regions using** mergeSandwhichedCells(...) ...
83	*step16*	**merge** {cell} **regions using** mergeRegionsSharingLineBounds() ...

Figure 5.8: Handley Algorithm RSL Operations Corresponding to Inference Times in Figure 5.9

historical precision) and returned in the final result (i.e. low final precision).

This provides a simple demonstration of how historical recall, historical precision, hypothesis histories and the interpretation tree may be used to diagnose and characterize *intermediate* decision making by a strategy. This is very useful both for analyzing recognition results, as we have demonstrated above, and when debugging a strategy. To observe the effect of an inference, one only ever need consult the RSL log containing the interpretation tree (which includes all decision results), and the inference times provided in the hypothesis histories recorded for interpretation graphs provide an index into the log.

5.6 Summary

In this chapter we described RSL implementations of two table structure recognition systems. The RSL specifications for these systems are included in Appendices C and D. We demonstrated how table model structure and dependencies between logical types are captured directly by the RSL syntax, and may be used to generate graphs for analysis.

We then provided a comparison of the implemented RSL strategies using a simple cell recognition game, in which the two implemented algorithms' cell interpretations were compared to the author's interpretations. We made use of the hypothesis histories in interpretations generated by RSL in order to observe *historical recall* and *historical precision* metrics. Paired with conventional recall and precision metrics, historical recall and precision determine the quantity of valid hypotheses that a strategy has 'thrown away,' and the overall accuracy of hypotheses generated. We also described how hypothesis histories can be used along with interpretation trees to locate inference results. This is particularly helpful when debugging RSL strategies.

We conclude in the next chapter, where we summarize the contributions of the dissertation and indicate future avenues of research both for the RSL language and other related areas of investigation, including human decision making.

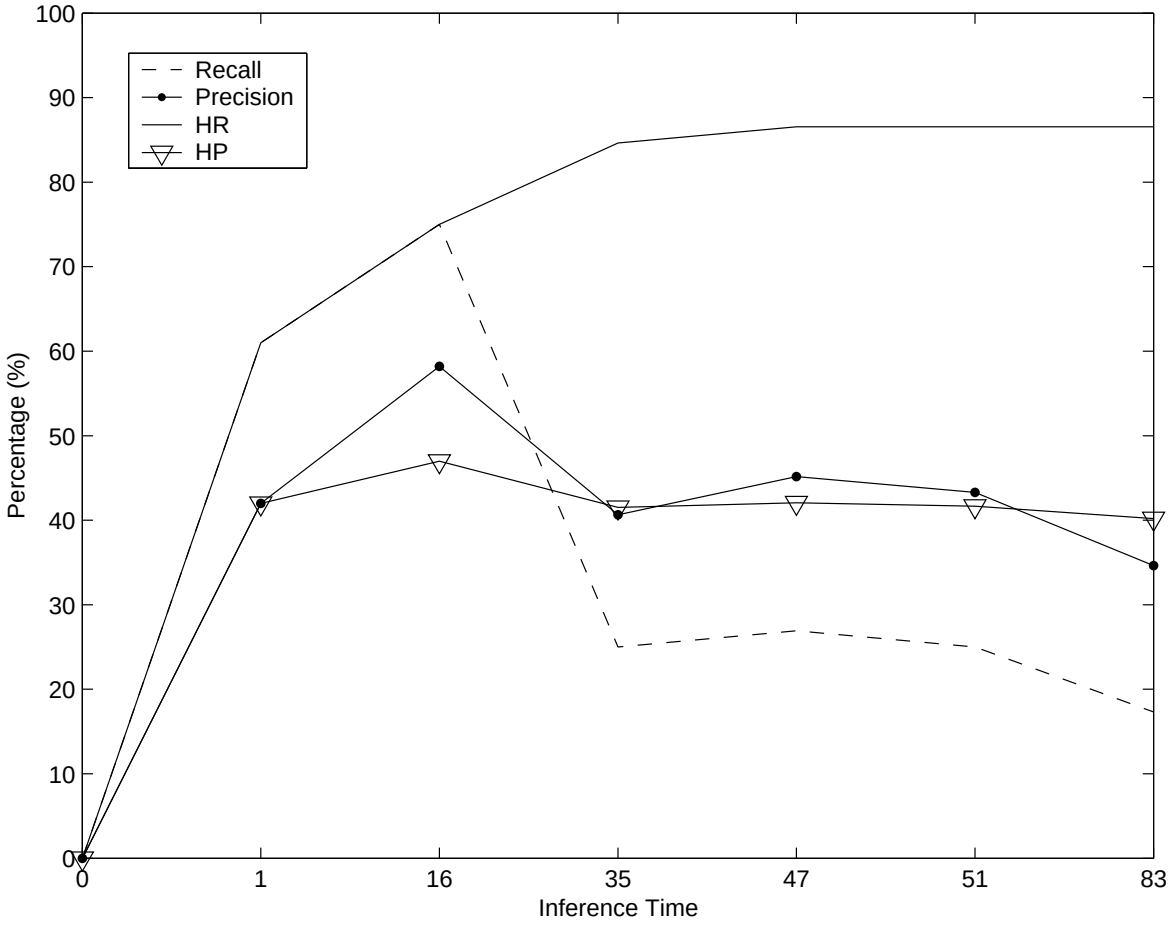


Figure 5.9: Intermediate Results of Handley's Algorithm for table in UW-I a038. 'HR' represents historical recall, and 'HP' historical precision. The lines are drawn for illustration; the inference times represented are only those with explicit numbers labelled on the X-axis. The operation corresponding to each inference time may be determined by consulting the RSL log, which contains an interpretation tree annotated with inference times. Once the text of the operation is found in the log file, it can be located within the RSL strategy. The operations corresponding to the inference times shown here are listed in Figure 5.8.

Chapter 6

Conclusion

Let us briefly review the preceding chapters. In Chapter 2 we surveyed table recognition in terms of decision making in recognition systems (see Figure 2.1). We then characterized table recognition problems in Chapter 3 as games where table recognizers imitate interpretations produced by a selected procedure (see Figure 3.1). In the same chapter we introduced the Recognition Strategy Language (RSL), which was inspired by this game-view where different strategies employ a fixed set of decision types in a well-defined context of evaluation. The language allows arbitrary decision functions to return structured text results that the language automatically records and then applies to data structures describing recognition results (see Figure 3.7).

In Chapter 4 we demonstrated an implementation of RSL in which RSL strategies are translated to TXL[22] programs. Tools for visualization and analysis were also described in Chapter 4, along with *historical* recall and precision metrics for characterizing the set of all generated hypotheses of a given type, including any that have been revised or rejected. In Chapter 5, the RSL-based implementations of two table structure recognition algorithms were described, and a simple game comparing

cell recognition in these algorithms was discussed, making use of historical recall and precision metrics.

In the remainder of this chapter we present the contributions of this dissertation (Section 6.1) and directions for future work (Section 6.2), and then close with a brief summary of the dissertation (Section 6.3).

6.1 Contributions

We have examined three methodological problems in this dissertation: the informality of most table recognition system specifications, the confounding of decision effects, and the necessary effort for constructing informally specified systems (see Section 1.2 for detailed descriptions of these problems). Contributions arising from our efforts are summarized below, organized by the problem that each contribution was intended to address.

Informal System Specifications

The RSL language was created to address the problem of informal strategy specifications in the table recognition literature (see Chapter 3). As demonstrated in Chapter 5, the informal specifications of 'feed-forward' systems can be re-specified in RSL, which formalizes the following:

- **Decision Types and Sequencing.** A fixed set of decision types are defined in RSL (see Section 3.9). Decision types provide a common language for describing strategies independently of the techniques used to make decisions. 'Feed-forward' decision processes are represented uniformly using the underlying function composition model of RSL strategies.

- **Table Models.** Region types and relations between region types in a table model are captured by the syntax of RSL inferencing functions (see Section 3.5).
- **Observation Specifications.** Observation specifications control the visibility of hypothesis types for external decision functions in RSL. External decision functions can only observe the scope types of an operation and those explicitly requested in an observation specification. Observation specifications are a new concept in the literature.

Confounded Decision Effects

- **Saving Intermediate Results.** We introduced *interpretation trees* (see Section 3.3) and *hypothesis histories* (see Section 3.5.3) to capture the effects of all decisions, including rejections and revisions of hypotheses. Previously in the literature interpretations were transformed in-place, losing the intermediate states recorded by interpretation trees and hypothesis histories. By recording all decision results and organizing them by the order in which they occur, the problem of confounded decision effects is resolved.
- **Historical Recall and Precision.** We introduced new metrics to take advantage of the additional information stored in hypothesis histories: *historical recall*, and *historical precision* (see Section 4.8). For a given hypothesis type, these characterize the ratio of imitation targets generated to imitation targets at any point in the course of a strategy's progress (*historical*

recall), and the ratio of imitation targets generated to generated hypotheses (*historical precision*). The use of these metrics is demonstrated in Chapter 5.

Ease of Implementation

- **RSL Core library.** The underlying model of RSL (see Figure 3.7) allows strategy designers to construct recognition systems using an RSL specification and a set of decision functions returning text results which can be easily re-sequenced and reused within the RSL specification itself. Overhead is substantially reduced by the RSL Core library, which takes results returned by decision functions, records them, and then automatically updates recognition result data structures appropriately. This more abstract approach saves considerable effort for a strategy designer who would otherwise be constructing the strategy in a general-purpose programming language.

6.2 Directions for Future Work

The RSL language as described in this dissertation represents only a very specific class of recognition strategies: those that construct interpretation trees breadth first, using only the current set of interpretations and the *accepted* hypotheses within each to support decisions. Also, while RSL was developed to formalize table recognition techniques, the language might be extended to work in other problem domains.

In the list below we describe these and future directions for RSL, along with

additional avenues of future research involving the study of 'recognition games,' collecting empirical data from human decision making, and using RSL to specify manual interpretation processes (for 'ground truth' creation).

- **Other Problem Domains:** RSL might be applied to problems involving one and three-dimensional data, such as speech recognition and segmenting objects in volumes (e.g. bones in CT scans). In the shorter term, RSL might be applied to other problems involving two-dimensional data (e.g. computer vision tasks for images).

- **Additional Control Flow Models:** RSL strategies construct interpretation trees breadth-first. It is probably of interest to extend RSL for describing other (depth-first, best-first) interpretation tree construction methods. One challenge here is preserving the brevity of the language. TXL[22] offers a possible avenue in this regard. TXL uses keywords and special characters to specify different search behaviours for matching the patterns of functions, and for other language operations involving search.

- **Observation Specifications:** inferencing operations are only permitted to observe *accepted* hypotheses in RSL, and only for candidate interpretations (the current leaves of an interpretation tree). It may be interesting to compare strategies in this observation scheme to others that use less or an entirely unconstrained views of hypotheses recorded in an interpretation tree. RSL might be extended to describe and capture details for each of these cases.

- **Accepting/Rejecting Hypotheses:** currently RSL cannot refine accepted interpretations (these are removed from the set to be further transformed), or

revise the acceptance state of interpretations within an interpretation tree. It is probably worth altering RSL to permit these actions. As all decisions are recorded in RSL, these operations would still be transparent. One can imagine scenarios where more information may be available after an interpretation has already been accepted or rejected by a strategy, making it preferable to revise the acceptance state of the interpretation.

- **Geometric Models:** more sophisticated, or multiple geometric models might be used in RSL. As a simple example, regions might be represented by polygons and lines, rather than bounding boxes and lines.

- **Evaluation as Basic RSL Operation:** evaluation operations should be added to the RSL language, as a user-customizable built-in operation. As mentioned in Chapter 3, we view evaluation as part of the problem definition for table recognition (which lead to our game characterization). RSL might provide one or more built-in functions that take a user evaluation function, and apply it to specified interpretations in an interpretation tree. Results could be summarized as text and/or returned in visualization formats (e.g. dot, gnuplot[28]).

- **Language Implementation:** RSL could be implemented in a variety of other languages to allow external functions to be written in those languages. Alternatively, perhaps an implementation could be designed to capture output from functions written in multiple languages. A general architecture for such an implementation would mean that individual decision functions could be implemented in whichever language a strategy designer would prefer (similar to Tcl[85]).

- **'Recognition Games' and Game Theory:** the class of imitation games described in Chapter 3 are extremely simple, and all games are 'scored' using the final interpretations of the selected interpretation procedure to imitate. What would a game where imitation is measured by similarity of strategies to the selected interpretation *process* itself look like? What if strategies were permitted at certain points to make binary queries of the structure of other players' interpretations, or of the interpretations to imitate themselves (e.g. query: 'Do you have a cell containing these input words?', response: 'No.')? Are such games interesting or useful? From these more interesting games can game theory[21, 78] provide helpful insights into strategy design and the table recognition problem?

- **Studying Human Decision Procedures:** could we learn more about table recognition by capturing human decision processes in interpretation trees or some similar formalization? For example, given a user interface for defining interpretations, could things be learned from recording decisions (operations performed, including any alterations or 'undos')[1] and then formalizing these using a language such as RSL? The psychology and expert systems literatures probably have information to offer in this regard (the author is mostly unfamiliar with both).

- **Constructing Interpretations Manually:** RSL could be used for flexibly defining 'ground-truth' protocols, in which a person or persons define a set of accepted interpretations used in table recognition imitation games. External

[1] Previously in a personal communication to the author, Dr. George Nagy proposed the idea of searching user interaction data for useful information.

functions of the RSL strategy would return text summaries for interface operations. Observation specifications might be used to control which parts of an interpretation are visible to persons constructing interpretations at each step.

6.3 Summary

Current systems for locating and analyzing tables in encoded documents are usually described informally, making the understanding, replication, and comparison of methods difficult. Informality leads to the additional problems of inseparability of decision effects, and increased effort in implementation. In this dissertation we proposed the Recognition Strategy Language as a means to address these problems. RSL provides an intermediate level of formalization, relative to strictly formalized syntax-based methods and the more flexible but informally defined operation sequences common in the literature.

Bibliography

[1] A.A. Abu-Tarif. Table processing and understanding. Master's thesis, Rensselaer Polytechnic Institute, 1998.

[2] A. Amano and N. Asada. Complex table form analysis using graph grammar. In *Lecture Notes in Computer Science*, volume 2423, pages 283–386, Berlin, 2002. Springer-Verlag.

[3] A. Amano, N. Asada, T. Motoyama, T. Sumiyoshi, and K. Suzuki. Table form document synthesis by grammar-based structure analysis. In *Proc. Sixth Int'l Conf. Document Analysis and Recognition*, pages 533–537, Seattle, WA, 2001.

[4] J.F. Arias, A. Chhabra, and V. Misra. Efficient interpretation of tabular documents. In *Proc. Thirteenth Int'l Conf. Pattern Recognition*, pages 681–685, Vienna, Austria, 1996.

[5] J.F. Arias, A. Chhabra, and V. Misra. Interpreting and representing tabular documents. In *Proc. Conf. Computer Vision and Pattern Recognition*, pages 600–605, San Francisco, CA, 1996.

[6] S. Balasubramanian, S. Chandran, J. Arias, and R. Kasturi. Information extraction from tabular drawings. In *Proc. Document Recognition I (IS&T/SPIE Electronic Imaging)*, volume 2181, pages 152–163, San Jose, CA, 1994.

[7] A. Belaïd. Recognition of table of contents for electronic library consulting. *Int'l J. Document Analysis and Recognition*, 4(1):35–45, 2001.

[8] L. Bing, J. Zao, and X. Hong. New method for logical structure extraction of form document image. In *Proc. Document Recognition and Retrieval VI (IS&T/SPIE Electronic Imaging)*, volume 3651, pages 183–193, San Jose, CA, 1999.

[9] D. Blostein, J.R. Cordy, and R. Zanibbi. Applying compiler techniques to diagram recognition. In *Proc. Sixteenth Int'l Conf. Pattern Recognition*, pages 123–126, Québec City, Canada, 2002.

[10] F.L. Bourgeois, H. Emptoz, and S.S. Bensafi. Document understanding using probabilistic relaxation: Application on tables of contents of periodicals. In *Proc. Sixth Int'l Conf. Document Analysis and Recognition*, pages 508–512, Seattle, WA, 2001.

[11] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, F. Yergeau, and J. Cowan. Extensible Markup Language (XML) 1.1. Available online at http://www.w3.org/TR/2004/REC-xml11-20040204, February 2004.

[12] T.M. Breuel. Functional programming for computer vision. In *Proceedings of the SPIE - The International Society for Optical Engineering*, volume 1659, pages 216–27, 1992.

[13] H. Bunke. Structural and syntactic pattern recognition. In C. H. Chen, L. F. Pau, and P.S.P. Wang, editors, *Handbook of Pattern Recognition and Computer Vision*, pages 163–209. World Scientific, Singapore, 1993.

[14] R.G. Casey and E. Lecolinet. A survey of methods and strategies in character segmentation. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 18(7):690–706, 1996.

[15] F. Cesarini, M. Gori, S. Marinai, and G. Soda. INFORMys: A flexible invoice-like form-reader system. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 20(7):730–745, 1998.

[16] F. Cesarini, M. Gori, S. Marinai, and G. Soda. Structured document segmentation and representation by the modified X-Y tree. In *Proc. Fifth Int'l Conf. Document Analysis and Recognition*, pages 563–566, Bangalore, India, 1999.

[17] F. Cesarini, S. Marinai, L. Sarti, and G. Soga. Trainable table location in document images. In *Proc. Sixteenth Int'l Conf. Pattern Recognition*, volume 3, pages 236–240, Québec City, Canada, 2002.

[18] S. Chandran and R. Kasturi. Structural recognition of tabulated data. In *Proc. Second Int'l Conf. Document Analysis and Recognition*, pages 516–519, Tsukuba Science City, Japan, 1993.

[19] A.K. Chhabra, V. Misra, and J. Arias. Detection of horizontal lines in noisy run length encoded images: the FAST method. In *Lecture Notes in Computer Science*, volume 1072, pages 35–48. Springer-Verlag, Berlin, 1996.

[20] R.A. Coll, J.H. Coll, and G. Thakur. Graphs and tables: a four-factor experiment. *Comm. ACM*, 37(4):76–86, 1994.

[21] A.M. Colman. *Game theory and experimental games: The study of strategic interaction*. Pergamon Press, Oxford, England, 1982.

[22] J.R. Cordy. TXL - a language for programming language tools and applications. In *Proc. LDTA 2004, ACM 4th International Workshop on Language Descriptions, Tools, and Applications*, pages 1–27, Barcelona, Spain, April 2004.

[23] J.R. Cordy, I. Charmichael, and R. Halliday. *The TXL Programming Language - Version 10.3*. Kingston, Ontario, Canada, 2003.

[24] J.R. Cordy, T.R. Dean, A.J. Malton, and K.A. Schneider. Source transformation in software engineering using the TXL transformation system. *Journal of Information and Software Technology*, 44(13):827–837, October 2002.

[25] J.R. Cordy, C.D. Halpern, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, Jan 1991.

[26] B. Coüasnon. DMOS: A generic document recognition method, application to an automatic generator of musical scores, mathematical formulae and table recognition systems. In *Proc. Sixth Int'l Conf. Document Analysis and Recognition*, pages 215–220, Seattle, WA, 2001.

[27] B. Coüasnon and L. Pasquer. A real-world evaluation of a generic document recognition method applied to a military form of the 19th century. In *Proc. Sixth*

Int'l Conf. Document Analysis and Recognition, pages 779–783, Seattle,WA, 2001.

[28] D. Crawford. GNUPLOT: an interactive plotting program. Available online at: http://www.gnuplot.info/docs/gnuplot.html, 1998.

[29] T.R. Dean, J.R. Cordy, K.A. Schneider, and A.J. Malton. Experience using design recovery techniques to transform legacy systems. In *Proc. ICSM 2001 - IEEE International Conference on Software Maintenance*, pages 622–631, Florence, Italy, November 2001.

[30] S. Douglas and M. Hurst. Layout and language: Lists and tables in technical documents. In *Proc. ACL SIGPARSE Workshop on Punctuation in Computational Linguistics*, pages 19–24, Santa Cruz, CA, 1996.

[31] S. Douglas, M. Hurst, and D. Quinn. Using natural language processing for identifying and interpreting tables in plain text. In *Proc. Fourth Ann. Symp. Document Analysis and Information Retrieval*, pages 535–546, Las Vegas, NV, 1995.

[32] R.O. Duda, P.E. Hart, and D.G. Stork. *Pattern Classification*. Wiley, New York, 2nd edition, 2001.

[33] M.D. Ernst. Static and dynamic analysis: synergy and duality. In *Proc. ICSE Workshop on Dynamic Analysis (WODA)*, Portland, Oregon, USA, May 2003.

[34] R. Fagin, J.Y. Halpern, Y. Moses, and M.Y. Vardi. *Reasoning about Knowledge*. MIT Press, Cambridge, MA, 1995.

[35] E.R. Gansner and S.C. North. An open graph visualization system and its applications to software engineering. *Software - Practice and Experience*, 30(11):1203–1233, 2000.

[36] E. Green and M. Krishnamoorthy. Model-based analysis of printed tables. In *Proc. Third Int'l Conf. Document Analysis and Recognition*, Montreal, Canada, 1995.

[37] E. Green and M. Krishnamoorthy. Model-based analysis of printed tables. In *Lecture Notes in Computer Science*, volume 1072, pages 80–91. Springer-Verlag, Berlin, 1996.

[38] J. Grossman, editor. *Chicago Manual of Style*, chapter 12 (Tables). University of Chicago Press, 14th edition, 1993.

[39] S. Tsai H. Chen and J. Tsai. Mining tables from large scale HTML texts. In *Proc. Eighteenth Int'l Conf. Computational Linguistics*, Saarbrucken, Germany, 2000.

[40] J. Ha, R.M. Haralick, and I.T. Phillips. Recursive X-Y cut using bounding boxes of connected components. In *Proc. Third Int'l Conf. Document Analysis and Recognition*, pages 952–955, Montreal, Canada, 1995.

[41] R. Hall. *Handbook of Tabular Presentation.* The Ronald Press Company, New York, 1943.

[42] J.C. Handley. *Electronic Imaging Technology*, chapter 8 (Document Recognition). IS&T/SPIE Optical Engineering Press, Bellingham, WA, 1999.

[43] J.C. Handley. Table analysis for multi-line cell identification. In *Proc. Document Recognition and Retrieval VIII (IS&T/SPIE Electronic Imaging)*, volume 4307, pages 34–43, San Jose, CA, 2001.

[44] R.M. Haralick. Document image understanding: Geometric and logical layout. In *Proc. Conf. Computer Vision and Pattern Recognition*, pages 385–390, Seattle, WA, 1994.

[45] R.M. Haralick and L.G. Shapiro. *Computer and Robot Vision (2 vols)*. Addison-Wesley, Reading, MA, 1992.

[46] K. Hinkelmann and O. Kempthorne. *Design and Analysis of Experiments: Introduction to Experimental Design*, volume 1. John Wiley and Sons Inc., New York, 1994.

[47] Y. Hirayama. A block segmentation method for document images with complicated column structures. In *Proc. Second Int'l Conf. Document Analysis and Recognition*, pages 91–94, Tsukuba Science City, Japan, 1993.

[48] Y. Hirayama. A method for table structure analysis using DP matching. In *Proc. Third Int'l Conf. Document Analysis and Recognition*, pages 583–586, Montreal, Canada, 1995.

[49] A. Hoover, G. Jean-Baptiste, X. Jiang, P.J. Flynn, H. Bunke, D.B. Goldgof, K. Bowyer, D. Eggert, A. Fitzgibbon, and Robert B. Fisher. An experimental comparison of range image segmentation algorithms. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 18(7):673–689, 1996.

[50] O. Hori and D.S. Doermann. Robust table-form structure analysis based on box-driven reasoning. In *Proc. Third Int'l Conf. Document Analysis and Recognition*, pages 218–221, Montreal, Canada, 1995.

[51] J. Hu, R. Kashi, D. Lopresti, G. Nagy, and G. Wilfong. Why table ground-truthing is hard. In *Proc. Sixth Int'l Conf. Document Analysis and Recognition*, pages 129–133, Seattle, WA, 2001.

[52] J. Hu, R. Kashi, D. Lopresti, and G. Wilfong. Medium-independent table detection. In *Proc. Document Recognition and Retrieval VII (IS&T/SPIE Electronic Imaging)*, volume 3967, pages 291–302, San Jose, CA, 2000.

[53] J. Hu, R. Kashi, D. Lopresti, and G. Wilfong. Experiments in table recognition. In *Proc. Workshop on Document Layout Interpretation and Applications*, Seattle, WA, 2001.

[54] J. Hu, R. Kashi, D. Lopresti, and G. Wilfong. Table structure recognition and its evaluation. In *Proc. Document Recognition and Retrieval VIII (IS&T/SPIE Electronic Imaging)*, volume 4307, pages 44–55, San Jose, CA, 2001.

[55] J. Hu, R.S. Kashi, D. Lopresti, and G.T. Wilfong. Evaluating the performance of table processing algorithms. *Int'l J. Document Analysis and Recognition*, 4(3):140–153, 2002.

[56] M. Hurst. Layout and language: Beyond simple text for information interaction - modelling the table. In *Proc. Second Int'l Conf. Multimodal Interfaces*, Hong Kong, 1999.

[57] M. Hurst. Layout and language: An efficient algorithm for detecting text blocks based on spatial and linguistic evidence. In *Proc. Document Recognition and Retrieval VIII (IS&T/SPIE Electronic Imaging)*, volume 4307, pages 56–67, San Jose, CA, 2001.

[58] M. Hurst. Layout and language: Challenges for table understanding on the web. In *Proc. First Int'l Workshop on Web Document Analysis*, pages 27–30, Seattle, WA, 2001.

[59] M. Hurst and S. Douglas. Layout and language: Preliminary investigations in recognizing the structure of tables. In *Proc. Fourth Int'l Conf. Document Analysis and Recognition*, pages 1043–1047, Ulm, Germany, 1997.

[60] M. Hurst and T. Nasukawa. Layout and language: Integrating spatial and linguistic knowledge for layout understanding tasks. In *Proc. Eighteenth Int'l Conf. Computational Linguistics*, Saarbrucken, Germany, 2000.

[61] K. Itonori. Table structure recognition based on textblock arrangement and ruled line position. In *Proc. Second Int'l Conf. Document Analysis and Recognition*, pages 765–768, Tsukuba Science City, Japan, 1993.

[62] A.K. Jain and B. Yu. Document representation and its application to page decomposition. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 20(3):294–308, 1998.

[63] T.G. Kieninger. Table structure recognition based on robust block segmentation. In *Proc. Document Recognition V (IS&T/SPIE Electronic Imaging)*, volume 3305, pages 22–32, San Jose, CA, 1998.

[64] T.G. Kieninger and A. Dengel. Applying the T-RECS table recognition system to the business letter domain. In *Proc. Sixth Int'l Conf. Document Analysis and Recognition*, pages 518–522, Seattle, WA, 2001.

[65] B. Klein, S. Gökkus, T. Kieninger, and A. Dengel. Three approaches to "industrial" table spotting. In *Proc. Sixth Int'l Conf. Document Analysis and Recognition*, pages 513–517, Seattle, WA, 2001.

[66] H. Kojima and T. Akiyama. Table recognition for automated document entry system. In *High-Speed Inspection Architectures, Barcoding, and Character Recognition (Proc. SPIE)*, volume 1384, pages 285–292, Boston, MA, 1990.

[67] W. Kornfeld and J. Wattecamps. Automatically locating, extracting and analyzing tabular data. In *Proc. Twenty-first Int'l ACM SIGIR Conf. Research and Development in Information Retrieval*, pages 347–348, Melbourne, Australia, 1998.

[68] S. Krishnamoorthy, G. Nagy, S. Seth, and M. Viswanathan. Syntactic segmentation and labelling of digitized pages from technical journals. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 15(7):737–747, 1993.

[69] S.W. Lam, L. Javanbakht, and S.N. Srihari. Anatomy of a form reader. In *Proc. Second Int'l Conf. Document Analysis and Recognition*, pages 506–509, Tsukuba Science City, Japan, 1993.

[70] A. Laurentini and P. Viada. Identifying and understanding tabular material in compound documents. In *Proc. Eleventh Int'l Conf. Pattern Recognition*, pages 405–409, The Hague, Netherlands, 1992.

[71] S. Lewandowksy and I. Spence. The perception of statistical graphs. *Sociological Methods and Research*, 18(2 & 3):200–242, 1989.

[72] J. Liang. *Document Structure Analysis and Performance Evaluation.* PhD thesis, University of WA, 1999.

[73] D. Lopresti. Exploiting WWW resources in experimental document analysis research. In *Lecture Notes in Computer Science*, volume 2423, pages 532–543, Berlin, 2002. Springer-Verlag.

[74] D. Lopresti and G. Nagy. Automated table processing: An (opinionated) survey. In *Proc. Third Int'l Workshop on Graphics Recognition*, pages 109–134, Jaipur, India, 1999.

[75] D. Lopresti and G. Nagy. A tabular survey of automated table processing. In *Lecture Notes in Computer Science*, volume 1941, pages 93–120. Springer-Verlag, Berlin, 2000.

[76] D. Lopresti and G. Wilfong. Evaluating document analysis results via graph probing. In *Proc. Sixth Int'l Conf. Document Analysis and Recognition*, pages 116–120, Seattle,WA, 2001.

[77] S. Mao and T. Kanungo. Empirical performance evaluation methodology and its application to page segmentation algorithms. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 23(3):242–256, 2001.

[78] P. Morris. *Introduction to Game Theory.* Springer-Verlag, New York, 1994.

[79] G. Nagy. Twenty years of document image analysis in PAMI. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 22(1):38–62, 2000.

[80] G. Nagy and S. Seth. Hierarchical representation of optically scanned documents. In *Proc. Seventh Int'l Conf. Pattern Recognition*, pages 347–349, Montreal, Canada, 1984.

[81] J.A. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, (7):308–313, 1965.

[82] H.T. Ng, C.Y. Lim, and J.L.T. Koo. Learning to recognize tables in free text. In *Proc. Thirty-Seventh Ann. Meet. Assn. Computational Linguistics*, pages 443–450, College Park, MD, 1999.

[83] L. O'Gorman. Image and document processing techniques for the RightPages electronic library system. In *Proc. Eleventh Int'l Conf. Pattern Recognition*, pages 260–263, The Hague, Netherlands, 1992.

[84] L. O'Gorman. The document spectrum for page layout analysis. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 15(11):1162–1173, 1993.

[85] J.K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, MA, 1994.

[86] L.I. Perlovsky. Conundrum of combinatorial complexity. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 20(6):666–670, 1998.

[87] C. Peterman, C.H. Chang, and H. Alam. A system for table understanding. In *Proc. Document Image Understanding Technology*, pages 55–62, Annapolis, MD, 1997.

[88] M. Petrou. Learning in pattern recognition. In *Lecture Notes in Computer Science*, volume 1715, pages 1–12. Springer-Verlag, Berlin, 1999.

[89] I. Phillips, S. Chen, and R. Haralick. CD-ROM document database standard. In *Proc. Second Int'l Conf. Document Analysis and Recognition*, pages 478–483, Tsukuba Science City, Japan, 1993.

[90] I. Phillips and A.K. Chhabra. Empirical performance evaluation of graphics recognition systems. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 21(9):849–870, 1999.

[91] A. Pizano. Extracting line features from images of business forms and tables. In *Proc. Eleventh Int'l Conf. Pattern Recognition*, pages 399–403, The Hague, Netherlands, 1992.

[92] R. Plamandon and S.N. Srihari. On-line and off-line handwriting recognition: A comprehensive survey. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 22(1):63–84, 2000.

[93] P. Pyreddy and W.B. Croft. Tintin: A system for retrieval in text tables. In *Proc. Second Int'l Conf. Digital Libraries*, pages 193–200, Austin, TX, 1997.

[94] J.R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Francisco, CA, 1993.

[95] RAF Technology, Redmond, WA. *Illuminator User's Manual*, 1995.

[96] M.A. Rahgozar and R. Cooperman. A graph-based table recognition system. In *Proc. Document Recognition III (IS&T/SPIE Electronic Imaging)*, volume 2660, pages 192–203, San Jose, CA, 1996.

[97] C. Ramey and B. Fox. *GNU Bash Reference Manual*. Network Theory Ltd., Bristol, UK, 2002.

[98] I. Redeke. Hierarchical interpretation of business charts for blind computer users using uml. In *Proc. Fourth Int'l IAPR Workshop on Graphics Recognition*, pages 440–454, Kingston, Canada, 2001.

[99] A. Rosenfeld and A.C. Kak. *Digital Picture Processing (2 vols).* Academic Press, Orlando, FL, 1982.

[100] D. Rus and D. Subramanian. Customizing information capture and access. *ACM Trans. Information Systems*, 15(1):67–101, 1997.

[101] T. Sato and B.V. Smith. Xfig user manual version 3.2.4. Available online at http://www.xfig.org/userman, December 2002.

[102] L. Seong-Whan and R. Dae-Seok. Parameter-free geometric document layout analysis. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 23(11):1240–1256, 2001.

[103] J.H. Shamillian, H.S. Baird, and T.L. Wood. A retargetable table reader. In *Proc. Fourth Int'l Conf. Document Analysis and Recognition*, pages 158–163, Ulm, Germany, 1997.

[104] S. Shapiro. *Thinking About Mathematics.* Oxford University Press, Oxford, England, 2000.

[105] C. Simonyi. *Meta-Programming: A Software Production Method.* PhD thesis, Stanford University, USA, 1976.

[106] S. Souafi-Bensafi, M. Parizeau, F. Lebourgeois, and H. Emptoz. Bayesian networks classifiers applied to documents. In *Proc. Sixth Int'l Conf. Document Analysis and Recognition*, pages 508–511, Seattle, WA, 2001.

[107] A. Takasu, S. Satoh, and E. Katsura. A document understanding method for database construction of an electronic library. In *Proc. Twelfth Int'l Conf. Pattern Recognition*, pages 463–466, Jerusalem, Israel, 1994.

[108] A. Takasu, S. Satoh, and E. Katsura. A rule learning method for academic document image processing. In *Proc. Third Int'l Conf. Document Analysis and Recognition*, pages 239–242, Montreal, Canada, 1995.

[109] W. Tersteegen and C. Wenzel. Scantab: Table recognition by reference tables. In *Proc. Third Workshop on Document Analysis Systems*, Nagano, Japan, 1998.

[110] K.M. Tubbs and D.W. Embley. Recognizing records from the extracted cells of microfilm tables. In *Proc. ACM Symp. Document Engineering*, pages 149–156, McLean, VA, 2002.

[111] S. Tupaj, Z. Shi, C.H. Chang, and H. Alam. Extracting tabular information from text files. URL: http://citeseer.nj.nec.com/tupaj96extracting.html, 1996.

[112] E. Turolla, A. Belaid, and A. Belaid. Form item extraction based on line searching. In *Lecture Notes in Computer Science*, volume 1072, pages 69–79, Berlin, 1996. Springer-Verlag.

[113] M. Viswanathan, E. Green, and M.S. Krishnamoorthy. Document recognition: An attribute grammar approach. In *Proc. Document Recognition III (IS&T/SPIE Electronic Imaging)*, volume 2660, pages 101–111, San Jose, CA, 1996.

[114] X. Wang. *Tabular Abstraction, Editing and Formatting*. PhD thesis, University of Waterloo, Canada, 1996.

[115] Y. Wang, R. Haralick, and I.T. Phillips. Zone content classification and its performance evaluation. In *Proc. Sixth Int'l Conf. Document Analysis and Recognition*, pages 540–544, Seattle,WA, 2001.

[116] Y. Wang and J. Hu. Detecting tables in HTML documents. In *Lecture Notes in Computer Science*, volume 2423, pages 249–260, Berlin, 2002. Springer-Verlag.

[117] Y. Wang, I.T. Phillips, and R. Haralick. Automatic table ground truth generation and a background-analysis-based table structure extraction method. In *Proc. Sixth Int'l Conf. Document Analysis and Recognition*, pages 528–532, Seattle,WA, 2001.

[118] Y. Wang, T. Phillips, and R.M. Haralick. Table detection via probability optimization. In *Lecture Notes in Computer Science*, volume 2423, pages 272–282, Berlin, 2002. Springer-Verlag.

[119] T. Watanabe, Q. Luo, and N. Sugie. Layout recognition of multi-kinds of table-form documents. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 17(4):432–445, 1995.

[120] K.Y. Wong, R.G. Casey, and F.M. Wahl. Document analysis system. *IBM Journal of Research and Development*, 26(6):647–656, 1982.

[121] M. Yoshida, K. Torisawa, and J. Tsujii. A method to integrate tables of the world wide web. In *Proc. First Int'l Workshop on Web Document Analysis*, pages 31–34, Seattle, WA, 2001.

[122] B. Yu and A.K. Jain. A generic system for form dropout. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 18(11):1127–1134, 1996.

[123] R. Zanibbi, D. Blostein, and J.R. Cordy. Recognizing mathematical expressions using tree transformation. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 24(11):1455–1467, 2002.

[124] K. Zuyev. Table image segmentation. In *Proc. Fourth Int'l Conf. Document Analysis and Recognition*, pages 705–708, Ulm, Germany, 1997.

Appendix A

RSL Operation Summary

We briefly summarize the RSL operation set in this section. More detailed descriptions are provided separately in Section 3.9. With the exception of the operations *accept interpretations*, *reject interpretations*, *for interpretations*, *write*, and *print*, RSL operations are applied independently to each candidate interpretation (see Section 3.3). Terms used in the operation summaries are described in Figure A.1.

Region Creation and Classification

create { *region type* } **regions using**
external_function(*parameter list*)
observation specification

Create regions of the specified type as determined by an external function. Region locations are specified by bounding boxes.

replace { *region type* } **regions using**
external_function(*parameter list*)
observation specification

Create new regions to replace existing regions of the specified type. 'Replaced' regions have their *region type* classification rejected.

classify { *set of region type* } **regions as** { *region type* }

Classify all regions in the specified set of region types as the specified region type.

classify { *set of region type* } **regions as** { *set of region type* } **using**
external_function(*parameter list*)
observation specification

region type(s)	identifier(s) representing a region type (or comma-separated list of region types) that have been specified in the **model regions** section of the RSL Strategy
relation(s)	identifier(s) representing a relation type (or comma-separated list of relation types) that has been specified in the **model relations** section of the RSL Strategy
node	a node (region) name
nodes	a comma separated list of node names (e.g. 'node1, node2')
pair	an ordered pair of node names
pairs	a comma separated list of ordered node name pairs (e.g. '(node1,node2), (node1,node3)')
interpretation	name of a candidate interpretation (e.g. I0, I1)
confidence	a numerical confidence value or statistic, or the identifier 'none,' used to indicate that no confidence value was produced
external_function	an external inferencing function, as described in Section 3.6.1
parameter list	a comma-separated list of zero or more recognition parameter names defined in the ***r*ecognition parameters** section of the *header*
observation specification	an observation specification section, as described in Section 3.7

Figure A.1: Terminology in Operation Summaries

Regions associated with a set of region types are classified as one or none of the second set of region types, as determined by an external function. If assigned multiple "possible" classes, the "maximum confidence" result is used for a region.

Region Segmentation

segment { *set of region type* } **regions into** { *region type* }

Create new regions of the specified type from existing regions associated with a set of region types.

segment { *set of region type* } **regions into** { *region type* } **using**
external_function(*parameter list*)
observation specification

Create new regions of the specified type from existing regions associated with a set of region types as determined by an external function.

resegment { *set of region type* } **regions into** { *region type* } **using**
external_function(*parameter list*)
observation specification

Revise existing regions of the specified region type to contain regions associated with the a set of region types as determined by an external function.

merge { *region type* } **regions using**
external_function(*parameter list*)
observation specification

Create new regions of the specified type by combining two or more existing regions of the same type. 'Merged' regions have their *region type* classification rejected.

Relations on Regions

relate { *region type [, region type]* } **regions with** { *relation* } **using**
external_function(*parameter list*)
observation specification

Define edges of the specified relation from regions of one or two region types, as determined by an external function.

Rejecting Region Type and Relation Hypotheses

reject { *set of region type* } **classifications**

All region classifications in the given set of region types are rejected.

reject { *set of region type* } **classifications using**
external_function(*parameter list*)
observation specification

An external function is used to determine which region classifications in the given set of region types are to be rejected.

reject { *set of relations* } **relations**

All edges in the passed set of relations are rejected.

reject { *set of relations* } **relations using**
external_function(*parameter list*)
observation specification

An external function is used to determine which edges in the given set of relations are to be rejected.

Accepting and Rejecting Interpretations

accept interpretations

All candidate interpretations are added to the set of accepted interpretations and removed from the set of candidate interpretations.

accept interpretations using
external_function(*parameter list*)
observation specification

An external function is used to determine which candidate interpretations are to be removed from the set of candidate interpretations and added to the set of accepted interpretations.

reject interpretations

All candidate interpretations are rejected, and removed from the set of candidate interpretations.

reject interpretations using
external_function(*parameter list*)
observation specification

An external function is used to determine which candidate interpretations are to be rejected and removed from the set of candidate interpretations.

Conditional Application of Strategies

for interpretations using
external_function(*parameter list*)
observation specification

Apply a strategy function only to candidate interpretations that meet a condition specified in an external function; candidates for which the condition fails are left as-is. If used, a 'for' statement is the first statement of a strategy function.

Parameter Adaptation

adapt { *adaptive parameter list* } **using**
external_function(*parameter list*)
observation specification

Alter the value of one or more adaptive parameters as determine by an external function. Adapted parameter values replace the parameter value in the current and nested scopes (i.e. no side effects are permitted)

File Output

The following commands write the current adaptive parameters (aparams), normalized interpretation tree structure (tree), candidate interpretations (current), or accepted interpretations (accepted) to a file.

write aparams "*file name*"

write tree "*file name*"

write current "*file name*"

write accepted "*file name*"

Terminal Output

The following commands print the current adaptive parameters (aparams), normalized interpretation tree structure (tree), candidate interpretations (current), or accepted interpretations (accepted) to the standard error stream.

print aparams

print tree

print current

print accepted

Appendix B

RSL Syntax

In this appendix we present a context-free grammar for RSL using the TXL grammar syntax. We summarize the TXL grammar syntax below. The complete syntax for TXL grammars and the TXL programming language are provided in the TXL Programming Language Manual[23].

B.1 TXL Grammar Syntax

Nonterminals

Nonterminal **A** is referred to using square brackets, as **[A]**

Rule Definition

The rule '**A** → **b A**' with terminal *b* is represented as:

define *A*
'b [A]
end define

Tokens and Comments

tokens . . . end tokens	defines additional input token types (non-terminals)
keys . . . end keys	defines keywords
comments . . . end comments	defines comment symbols or sequences

Non-terminal Modifiers

[repeat X]	zero or more X non-terminals
[list X]	comma-separated list of zero or more X non-terminals
X+	1 or more X non-terminals
X*	0 or more X non-terminals
opt X	0 or more X non-terminals

Built-in Nonterminal Types

[id]	identifier
[number]	floating-point number
[stringlit]	string literal (e.g. "This is a literal")
[empty]	ϵ in traditional context-free grammar specifications

B.2 RSL Grammar

Please note that both TXL and RSL keywords are shown in bold in this listing.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Terminal (token) definitions, keywords, comments in RSL
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
tokens
    % Adaptive, static parameters.
    aparam "a[\A][\a\i\d\u]*[']*"
    sparam "s[\A][\a\i\d\u]*[']*"
end tokens

keys
    'model 'recognition 'parameters 'strategy 'end
    'for 'accept 'reject 'classify 'segment 'resegment 'classify
    'merge 'create 'replace 'relate 'print 'write
end keys

comments
    '%
end comments

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% RSL types
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
define scope_types
    [basic_type_list]
end define

define basic_type_list
    '{ [list id+] '}
end define

define basic_type
    '{ [id] '}
end define

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Parameters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
define parameter_list
    [repeat parameter_def]
end define

define parameter_def
    [param_name] [param_value]
end define

define param_value
        [number]
    |   [stringlit]
    |   '- [number]  % *for reading negative numbers
end define

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Header
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
define header
    [model_regions]
    [model_relations]
    [recognition_parameters]
end define

define model_regions
    'model 'regions
        [repeat id]
    'end 'regions
end define

define model_relations
    'model 'relations
        [repeat id]
    'end 'relations
end define

define recognition_parameters
    'recognition 'parameters
        [parameter_list]
    'end 'parameters
end define

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Strategies
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
define strategy
    'strategy [id]
```

```
        [opt for_op]
        [repeat strategy_op+]
    'end 'strategy
end define

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Conditional statement
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
define for_op
    'for 'interpretations
        [external_call] [interp_observation]
end define

define strategy_op
          [inference_operation]
    |     [parameter_transform]
    |     [strategy_name]
    |     [print_statement]
    |     [write_statement]
end define

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Inference Operations
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
define inference_operation
          [create_op]       | [replace_op] | [class_op]    | [segment_op]
    |     [resegment_op]    | [merge_op]   | [relate_op]   | [reject_class]
    |     [reject_relation] | [accept_interp]  | [reject_interp]
end define

define create_op
    'create [basic_type] 'regions
        [external_call] [interp_observation]
end define

define replace_op
    'replace [scope_types] 'regions
        [external_call] [interp_observation]
end define

define class_op
        'classify [scope_types] 'regions 'as  [basic_type]
    |   'classify [scope_types] 'regions 'as [basic_type_list]
            [external_call]    [interp_observation]
end define

define segment_op
        'segment [scope_types] 'regions 'into [basic_type] 'regions
```

```
    |   'segment [scope_types] 'regions 'into [basic_type] 'regions
            [external_call] [interp_observation]
end define

define resegment_op
    'resegment [scope_types] 'regions 'into [basic_type] 'regions
        [external_call] [interp_observation]
end define

define merge_op
    'merge [scope_types] 'regions
        [external_call] [interp_observation]
end define

define relate_op
        'relate [scope_types] 'regions 'with [basic_type]
    |   'relate [scope_types] 'regions 'with [basic_type]
            [external_call]    [interp_observation]
end define

define reject_class
        'reject [scope_types] 'classifications
    |   'reject [scope_types] 'classifications
            [external_call] [interp_observation]
end define

define reject_relation
        'reject [scope_types] 'relations
    |   'reject [scope_types] 'relations
            [external_call] [interp_observation]
end define

define accept_interp
        'accept 'interpretations
    |   'accept 'interpretations
            [external_call] [interp_observation]
end define

define reject_interp
        'reject 'interpretations
    |   'reject 'interpretations
            [external_call] [interp_observation]
end define

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% External functions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
define external_call
```

```
    'using [function_name] '(  [list param_name] ')
end define

define function_name
    [id]
end define

%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Observation specifications
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
define interp_observation
        'observing [opt region_observe] [opt edge_observe]
    |     [empty]
end define

define region_observe
    [basic_type_list] 'regions
end define

define edge_observe
    [basic_type_list] 'relations
end define

%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Parameter transform ('adapt')
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
define parameter_transform
    'adapt [list param_name]
        [external_call] [interp_observation]
end define

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Output operations
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
define print_statement
        'print [stringlit] | 'print 'aparams   | 'print 'tree
    |   'print 'current    | 'print 'accepted
end define

define write_statement
        'write 'aparams [stringlit] | 'write 'tree [stringlit]
    |   'write 'current [stringlit] | 'write 'accepted [stringlit]
    |   'write 'results [stringlit]
end define
```

Appendix C

Handley's Structure Recognition Algorithm in RSL

The RSL strategy shown below has been used to implement Handley's table structure recognition algorithm[43]. This strategy is discussed in detail in Chapter 5.

RSL Strategy Listing

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% NewHandley.rsl
%    - Implementation of John Handley's Table Structure Recognition
%      Algorithm (from Document Recognition and Retrieval VIII, 2001)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Revision History
% v 1.0.0 Created by: Richard Zanibbi, Oct 02 2004 16:14:27
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

model regions
    % input types
    Image Word Line

    % line types
    Hline Underline Invisible_Hline
    Vline Invisible_Vline

    % Word and Cell group types
    Cell Header_Cell Block

    % Cell group types
    Row Column

    % Table region types
    Fully_Ruled_Table Table_Frame
end regions

model relations
```

```
    % NOTE: 'contains' is implicitly observed in all RSL operations;
    % it is the region structure relation type.
    contains

    % Defining "horizontally close" Cells
    close_to

    % Adjacencies between lines and Cells
    adjacent_left adjacent_right adjacent_top adjacent_bottom
end relations

recognition parameters
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % String parameters
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    sLeft "left"
    sRight "right"
    sTop "top"
    sBottom "bottom"

    sCloseTo "close_to" % a relation type name.

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Numerical parameters (listed
    % by strategy in which they first
    % appear)
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    sScanResolution 300 % dpi

    % Steps 1 and 2
    sHlineMinAspectRatio 3
    sNumberFullyRuledCols 2
    sMaxUnderlineLength 10 % mm
    sUnderlineWordMaxSeparation 12.8 % mm

    % Step 4
    sMinWordsInHorProj 2

    % Step 5
    sLinearCombinationX 2
    sLinearCombinationY 1

    % Step 6
    sVerProjectTopRange 6.4
    sVerProjectTopWeight 1
    sVerProjectDefaultWeight 2
    sColumnMinGap 1.5
    sColumnHistogramThreshold  2

    % Step 10
    sNumberColsInSpanHeader 2

    % Step 11
    sRowMinGap 1.5
    sMinNumberRowsForStyle1 3
    sMaxMultiLineSeparation 5

    % Step 12
    sMaxNumberSandwhichRows 3

    % Step 13
```

```
    sSecondHorProjectThreshold 1

    % Step 15
    sVlineMinGap 4.5 %mm
    sHlineMinGap 1.5 %mm
end parameters

strategy main
    steps1and2
    step3
    print "Steps 1-3 complete."

    step4
    step5
    step6
    step7
    step8
    step9
    step10
    step11
    step12
    step13
    step14
    step15
    step16
    print "Steps 4-16 complete."

    % Accept all current interpretations at end.
    accept interpretations
end strategy

strategy steps1and2
    % Label all Words as Cells.
    classify { Word } regions as { Cell }

    % Create initial table frame as BB of Cells.
    create { Table_Frame } regions using
        createRegionFromBB()
        observing
            { Cell } regions

    % Classify all lines as horizontal or vertical.
    classify { Line } regions as { Hline, Vline } using
        classLineDirection(sHlineMinAspectRatio)

    % Label Underlines (in horizontal line list)
    classify { Hline } regions as { Underline } using
        classifyUnderlines(sMaxUnderlineLength, sUnderlineWordMaxSeparation,
            sScanResolution)
        observing
            { Word } regions

    % Remove Underlines from the list of horizontal lines.
    reject { Hline } classifications using
        removeUnderlines()
        observing
            { Underline } regions
end strategy

strategy step3
    % Determine the adjacent lines of the Cells.
```

```
    analyzeLineAdjacency

    % Analysis for fully ruled tables (only)
    fullyRuledAnalysis
end strategy

strategy analyzeLineAdjacency
    % Reject the existing set of line adjacency relations.
    reject { adjacent_left, adjacent_right, adjacent_top,
        adjacent_bottom } relations

    % Then find adjacent lines closest to each side of a Cell's
    % bounding box.
    relate { Vline, Cell } regions with { adjacent_left } using
        inferCellBoundingLines(sLeft)

    relate { Vline, Cell } regions with { adjacent_right } using
        inferCellBoundingLines(sRight)

    relate { Hline, Cell } regions with { adjacent_top } using
        inferCellBoundingLines(sTop)

    relate { Hline, Cell } regions with { adjacent_bottom } using
        inferCellBoundingLines(sBottom)
end strategy

strategy fullyRuledAnalysis
    % Only apply this strategy if all Cells are adjacent to a line
    % in all four directions, and there are sNumberFullyRuledCols
    % detected (Note: I've done this using the number of vertical
    % lines and the number of vertical lines bounding Cells).
    for interpretations using
        skipNonFullyRuledInterps(sNumberFullyRuledCols)
        observing
            { Cell, Vline, Hline } regions
            { adjacent_left, adjacent_right, adjacent_top,
              adjacent_bottom } relations

    % Merge Cells with the same bounding lines.
    merge { Cell } regions using
        mergeRegionsSharingLineBounds()
        observing
            { Vline, Word } regions
            { adjacent_left, adjacent_right, adjacent_top,
              adjacent_bottom } relations

    % Define a "fully ruled" table region using the bounding box
    % of all Cells, and vertical and horizontal table lines.
    create { Fully_Ruled_Table } regions using
        createRegionFromBB()
        observing
            { Vline, Hline, Cell } regions

    % Accept the resulting interpretation (this also removes the
    % interpretations from the list of current interpretations)
    accept interpretations
end strategy

strategy step4
    % Produce a first estimate of Row locations using a horiztonal
    % projection of Word and horizontal line bounding boxes.
```

```
    create { Row } regions using
        initialRowProjection(sMinWordsInHorProj)
        observing
            { Word, Hline, Table_Frame } regions
end strategy

strategy step5
    % Define Cells as being "close_to" one another if they
    % are horizontally adjacent and within sColumnMinGap mm
    % of one another in the X direction (see J. Handley's paper
    % for a description of the adjacency metric)
    relate { Cell } regions with { close_to } using
        relateCloseCells(sColumnMinGap, sLinearCombinationX,
            sLinearCombinationY,sScanResolution)
        observing
            { Cell } regions
            { adjacent_left, adjacent_right } relations

    % All Cells that are horizontally "close_to" one another
    % are merged into new Cells.  We never merge across lines,
    % so we observe vertical lines and adjacency relations here.
    merge { Cell } regions using
        mergeClosure(sCloseTo)
        observing
            { Vline } regions
            { adjacent_left, adjacent_right, close_to } relations
end strategy

strategy step6
    % Produce an initial estimate of Column locations using
    % a weighted vertical bounding box projection of Cells
    % and vertical lines.
    create { Column } regions using
        initialColumnProjection(sVerProjectTopRange,
            sVerProjectTopWeight,
            sVerProjectDefaultWeight,
            sColumnHistogramThreshold,sScanResolution)
        observing
            { Cell, Vline, Table_Frame } regions

    % Merge Columns that are within sColumnMinGap of one another
    % in the X direction.
    merge { Column } regions using
        mergeHorCloseRegions(sColumnMinGap,sScanResolution)
        observing
            { Vline } regions
end strategy

strategy step7
    % Define "Blocks" of Cells to use in a modified projection.
    % These will be used to help improve our Column estimate.
    segment { Cell } regions into { Block } regions using
        mergeXCentersInBB()
        observing
            { Column } regions

    % Reject the old Columns.
    reject { Column } classifications

    % Perform another weighted projection, this time using Cells,
    % vertical lines, and the new Blocks.
```

```
    create { Column } regions using
        secondColumnProjection(sVerProjectTopRange,
            sVerProjectTopWeight,
            sVerProjectDefaultWeight,
            sColumnHistogramThreshold,sScanResolution)
        observing
            { Cell, Block, Vline, Table_Frame } regions

    % Merge the new Columns that are within sColumnMinGap of one another
    % in the X direction.
    merge { Column } regions using
        mergeHorCloseRegions(sColumnMinGap,sScanResolution)
        observing
            { Vline } regions
end strategy

strategy step8
    % Assign each Cell to the Column with which it has
    % "relative horizontal maximum overlap" (see the paper)
    % (none, if this value is 0)
    resegment { Cell } regions into { Column } regions using
        assignCellsToColumns()
        observing
            { Column } regions

    % Assign each Cell to the Row with which it has
    % "relative vertical maximum overlap" (again, see the paper)
    % (no assignment if this value is 0)
    resegment { Cell } regions into { Row } regions using
        assignCellsToRows()
        observing
            { Row } regions

    % Assign Cells with Columns but no Row to the Row above.
    resegment { Cell } regions into { Row } regions using
        repairRowStructure()
        observing
            { Row, Column } regions

    % Assign Cells with a Row but no Column to a new Column.
    segment { Cell } regions into { Column } regions using
        repairColumnStructure()
        observing
            { Column, Row } regions
end strategy

strategy step9
    % Need new line adjacenecies for the new Cells.
    analyzeLineAdjacency

    % Merge Cells at the same Row and Column position.
    % Check line adjacencies to insure we don't merge across
    % a line.
    merge { Cell } regions using
        mergeCellsAtSameGridPosition()
        observing
            { Column, Row } regions
            { adjacent_left, adjacent_right, adjacent_top, adjacent_bottom }
                relations

    % Rows containing Cells that have been merged have these Cells replaced
```

```
    % with the parent merged regions.
    resegment { Cell } regions into { Row } regions using
        reviseMergedRegions()
        observing
            { Row } regions

    % Columns containing Cells that have been merged have these Cells
    % with the parent merged regions.
    resegment { Cell } regions into { Column } regions using
        reviseMergedRegions()
        observing
            { Column } regions
end strategy

strategy step10
    % Cells in the first Row that span sNumberColsInSpanHeader
    % Columns are classified as a Header.
    classify { Cell } regions as { Header_Cell } using
        labelHeaders(sNumberColsInSpanHeader)
        observing
            { Row, Column } regions
end strategy

strategy step11
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Part 1
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Update the line adjacency for the current Cells.
    analyzeLineAdjacency

    % Merge Cells that span Rows at the top of the table.
    merge { Cell } regions using
        mergeSpanningCells(sMinNumberRowsForStyle1,sRowMinGap,
            sScanResolution)
        observing
            { Row, Column, Header_Cell, Hline } regions
            { adjacent_top, adjacent_bottom } relations

    % Update Row structure to reflect the merged Cells (NOTE:
    % spanned Cells are members of all Rows containing their children)
    resegment { Cell } regions into { Row } regions using
        reviseMergedRegions()

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Part 2
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Merge Cells in a Row with Cells in the Row above if there
    % is no Cell in the leftmost Column, and the Rows are within
    % sMaxMultiLineSeparation of one another.
    merge { Cell } regions using
        mergeCellsBelow(sMaxMultiLineSeparation,sScanResolution)
        observing
            { Column, Row, Hline } regions
            { adjacent_top, adjacent_bottom } relations

    % Merge the Rows whose Cells were merged in the previous step.
    merge { Row } regions using
        mergeRowsWithCellsMergedInLastStep()
        observing
            { Cell } regions
```

```
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Part 3
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Third step; merge lonely Cells in the first Column with the
    % Cell below it.
    merge { Cell } regions using
        mergeLoneCellsInRow()
        observing
            { Column, Row, Hline } regions
            { adjacent_top, adjacent_bottom } relations
end strategy

strategy step12
    % Update the line adjacency for Cells again.
    analyzeLineAdjacency

    % Merge up to sMaxNumberSandwhichRows at the top of the table
    % if they are "sandwhiched" by two horizontal lines.
    merge { Cell } regions using
        mergeSandwhichedCells(sMaxNumberSandwhichRows)
        observing
            { Row, Column } regions
            { adjacent_top, adjacent_bottom } relations
end strategy

strategy step13
    % Reject the old Row estimates.
    reject { Row } classifications

    % As in step 4, use a horizontal projection of Cells and horizontal
    % lines to define Row locations (this time with a smaller threshold)
    create { Row } regions using
        initialRowProjection(sSecondHorProjectThreshold)
        observing
            { Cell, Hline, Table_Frame } regions
end strategy

strategy step14
    % Reject the current table frame...
    reject { Table_Frame } classifications

    % And create a new one using the bounding box of the Cells and
    % vertical and horizontal lines.
    create { Table_Frame } regions using
        createRegionFromBB()
        observing
            { Cell, Hline, Vline } regions
end strategy

strategy step15
    % Create "invisible" line separators at Row gaps.
    create { Invisible_Hline } regions using
        createLinesAtRowGaps()
        observing
            {Row, Table_Frame} regions

    % Create "invisible" line separator at Column gaps.
    create { Invisible_Vline } regions using
        createLinesAtColumnGaps()
        observing
            { Column, Table_Frame } regions
```

```
    % Reject "invisible" Row separators that are close to
    % a real line separating Rows (within sHlineMinGap mm)
    reject { Invisible_Hline } classifications using
        rejectRealInvHlines(sHlineMinGap,sScanResolution)
        observing
            { Hline, Row } regions

    % Similarly for Column separators, remove "Invisible"
    % Column separators that are close to a real line separating
    % Columns.
    reject { Invisible_Vline } classifications using
        rejectRealInvVlines(sVlineMinGap,sScanResolution)
        observing
            { Vline, Column } regions

    % Replace real double-rulings separating Rows with
    % a line located at their vertical mid-point.
    replace { Hline } regions using
        replaceCloseHlines(sHlineMinGap,sScanResolution)
        observing
            { Invisible_Hline, Row } regions

    % Replace real double-ruling separating Columns with
    % a line located at their horizontal mid-point.
    replace { Vline } regions using
        replaceCloseVlines(sVlineMinGap,sScanResolution)
        observing
            { Invisible_Vline, Column } regions
end strategy

strategy step16
    % For the last time, update the line and Cell adjacency.
    % This time we use a different function, as we want to use
    % real and "invisible" rulings.
    analyzeFinalLineAdjacency

    merge { Cell } regions using
        mergeRegionsSharingLineBounds()
        observing
            { adjacent_left, adjacent_right, adjacent_top,
              adjacent_bottom } relations

end strategy

strategy analyzeFinalLineAdjacency
    % This is similar to 'analyzeLineAdjacency', but takes invisible
    % lines into account.
    reject { adjacent_left, adjacent_right, adjacent_top,
        adjacent_bottom } relations

    relate { Vline, Cell } regions with { adjacent_left } using
        inferFinalCellBoundingLines(sLeft)
        observing
            { Invisible_Vline } regions

    relate { Vline, Cell } regions with { adjacent_right } using
        inferFinalCellBoundingLines(sRight)
        observing
            { Invisible_Vline } regions
```

```
    relate { Invisible_Vline, Cell } regions with { adjacent_left } using
        inferFinalCellBoundingLines(sLeft)
        observing
            {Vline } regions

    relate { Invisible_Vline, Cell } regions with { adjacent_right } using
        inferFinalCellBoundingLines(sRight)
        observing
            { Vline } regions

    relate { Hline, Cell } regions with { adjacent_top } using
        inferFinalCellBoundingLines(sTop)
        observing
            { Invisible_Hline } regions

    relate { Hline, Cell } regions with { adjacent_bottom } using
        inferFinalCellBoundingLines(sBottom)
        observing
            { Invisible_Hline } regions

    relate { Invisible_Hline, Cell } regions with { adjacent_top } using
        inferFinalCellBoundingLines(sTop)
        observing
            { Hline } regions

    relate { Invisible_Hline, Cell } regions with { adjacent_bottom } using
        inferFinalCellBoundingLines(sBottom)
        observing
            { Hline } regions
end strategy
```

Appendix D

Hu et al.'s Structure Recognition Algorithm in RSL

The RSL strategy shown below has been used to implement Hu et al.'s table structure recognition algorithm[53]. Note that we have adapted their algorithm slightly in order to process inputs containing word regions from images rather than ASCII text files, as in the original. This strategy is discussed in detail in Chapter 5.

RSL Strategy

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% HuEtAl.rsl
%   - Implementation of Hu et. al.'s table structure recognition
%     algorithm from Document Recognition and Retrieval VIII, 2001.
%   - NOTE: This implementation adapts the original ASCII-based strategy
%     to work with regions defined in Image files.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Revision History
% v 1.0.0 Original Version: Richard Zanibbi, Oct 01 2004 12:29:28
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

model regions
    % Input region types: note that lines are unused.
    Image Word line

    % Types associated with Words
    Alpha_Word NonAlpha_Word

    % For Cluster tree.
    Cluster

    % Textline and types associated with Textlines.
    Textline Headerline Boxhead Core_Line Partial_Line
    Consistent_Line Inconsistent_Line
```

```
    % Cells and types associated with Cells.
    Cell Column_Header Row_Header Stub_Head

    % Rows, Columns, and Stub (a Column type)
    Row Column Alpha_Column NonAlpha_Column Stub
    Final_Row Final_Column
end regions

model relations
    % Default region containment relation.
    contains

    % For indexing structure.
    indexes
end relations

recognition parameters
    % Main strategy
    sThreshold  1   % for creating text lines from hor. projections
    sOverlap 0.5    % overlap threshold for merging overlapping regions.
    sScanResolution 300 % dpi

    % Column analysis parameters
    sAlpha 0.8        % a weight in [0,1]
    sG 2.0            % a threshold, in MM

    % Boxhead analysis
    sMaxBoxheadHeight   30 %6.4 % mm; based on v. proj.
                               % hard-valued approximation of '5 Textlines'

    % Column Header analysis
    sMinColumnSeparation 2.0 %1.5     % approximation of "two spaces" in mm

    % Row analysis
    sMaxRowLineSeparation 4.5 % approximation in MM.
end parameters

strategy main
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Segment Words into Textlines
    % (to "fake" ASCII text)
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    createTextlines

    columnAnalysis
    boxheadAnalysis

    indexAnalysis
    rowAnalysis
    bodyCellCreation

    accept interpretations
end strategy

strategy createTextlines
    create { Textline } regions using
        createRegionsFromProjections(sThreshold,sOverlap)
        observing
            { Word } regions

    resegment { Word } regions into { Textline } regions using
```

```
        mergeYCentersInBB()
        observing
            { Textline } regions
end strategy

strategy columnAnalysis
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Build the hierarchical Cluster
    % tree
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    classify { Word } regions as { Cluster }

    buildClusterTree

    print "Cluster tree is complete."

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Segment Columns by cutting the
    % Cluster tree
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    classify { Cluster } regions as { Column } using
        cutClusterTree(sAlpha,sG,sScanResolution)
        observing
            { Textline } regions

    print "Cluster tree has been cut."
end strategy

strategy buildClusterTree
    % Using literal, recursive description from the paper.
    for interpretations using
        filterForCompleteClusterTrees()
        observing
            { Cluster } regions

    segment { Cluster } regions into { Cluster } regions using
        joinClosestClusterPair()

    buildClusterTree
end strategy

strategy boxheadAnalysis
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Separate the table body from
    % the Boxhead, if present.
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    classify { Word } regions as { Alpha_Word, NonAlpha_Word } using
        classifyWordTokenType()

    classify { Column } regions as { Alpha_Column, NonAlpha_Column } using
        classifyDominantColumnTokenType()
        observing
            { Alpha_Word, NonAlpha_Word } regions

    classify { Textline } regions as { Consistent_Line, Inconsistent_Line } using
        classifyTextlineConsistency()
        observing
            { Alpha_Word, NonAlpha_Word, Alpha_Column, NonAlpha_Column } regions

    segment { Inconsistent_Line } regions into { Boxhead } regions using
        putInConsistentLinesInBoxhead(sMaxBoxheadHeight,sScanResolution)
```

```
        observing
            { Consistent_Line } regions

    resegment { Word } regions into { Column } regions using
        removeBoxheadWordsFromColumns()
        observing
            { Boxhead, Column } regions

end strategy


strategy indexAnalysis
    % Define Header Cells as groups of 'close' Words
    % along Textlines in the Boxhead.
    segment { Word } regions into { Column_Header } regions using
        cutLineIntoPhrases(sMinColumnSeparation, sScanResolution)
        observing
            { Inconsistent_Line, Boxhead } regions

    % Associate Header cells closest to Columns...
    relate { Column_Header, Column } regions with { indexes } using
        determineLowestColumnHeaders()
        observing
            { Boxhead, Inconsistent_Line, Word } regions

    % .. and (recursively) the parent Headers of the
    %    Headers.
    relate { Column_Header } regions with { indexes } using
        determineHeaderNesting()
        observing
            { Boxhead, Inconsistent_Line, Word, Column } regions
            { indexes } relations

    % Leftmost Column is assumed to be a Stub.
    classify { Column } regions as { Stub } using
        classifyLeftmost()
end strategy

strategy rowAnalysis
    % Determine whether lines end or (usually) continue a Row.
    classify { Textline } regions as { Core_Line, Partial_Line } using
        classifyTextlineRowType()
        observing
            { Word, Column, Boxhead } regions

    % Use simple rule to combine these (partial lines always associated
    % with core lines above if not interrupted by a "blank" line)
    segment { Textline } regions into { Row } regions using
        groupPartialToCoreLines(sMaxRowLineSeparation, sScanResolution)
        observing
            { Boxhead, Core_Line, Partial_Line } regions
end strategy

strategy bodyCellCreation
    % Create body Cells.
    segment { Word } regions into { Cell } regions using
        segmentWordsInSameRowAndColumnAsCell()
        observing
            { Row, Column } regions

    % All Column Headers are also Cells; record this.
```

```
    classify { Column_Header } regions as { Cell }

    % Assign Cells to their appropriate Rows
    % and Columns; use a new label to make book-keeping
    % easier (define new, reject old)
    segment { Cell } regions into { Final_Column } regions using
        assignCellsToRegion()
        observing
            { Word, Column } regions

    % Define Row Headers.
    classify { Cell } regions as { Row_Header } using
        assignRowHeaders()
        observing
            { Final_Column } regions

    % Reject old Columns.
    reject { Column } classifications

    segment { Cell } regions into { Final_Row } regions using
        assignCellsToRegion()
        observing
            { Word, Row } regions

    reject { Row } classifications
end strategy
```

Appendix E

Table Cell Interpretations

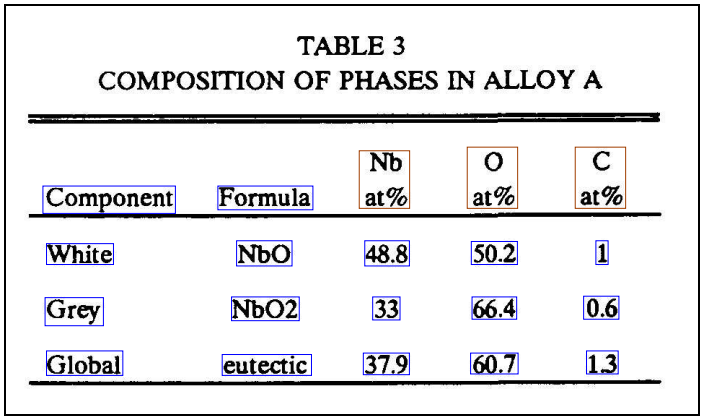


TABLE 3
COMPOSITION OF PHASES IN ALLOY A

Component	Formula	Nb at%	O at%	C at%
White	NbO	48.8	50.2	1
Grey	NbO2	33	66.4	0.6
Global	eutectic	37.9	60.7	1.3

(a) Author and Handley Algorithm Interpretation

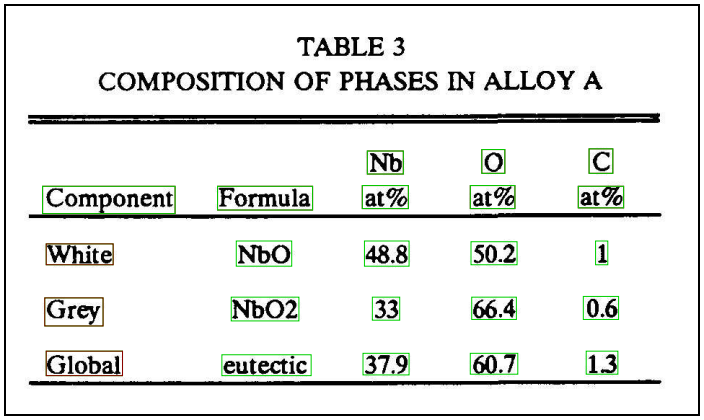


TABLE 3
COMPOSITION OF PHASES IN ALLOY A

Component	Formula	Nb at%	O at%	C at%
White	NbO	48.8	50.2	1
Grey	NbO2	33	66.4	0.6
Global	eutectic	37.9	60.7	1.3

(b) Output Interpretation from Hu et al.'s Algorithm

Figure E.1: Cell Interpretations for Table in UW-I d05d

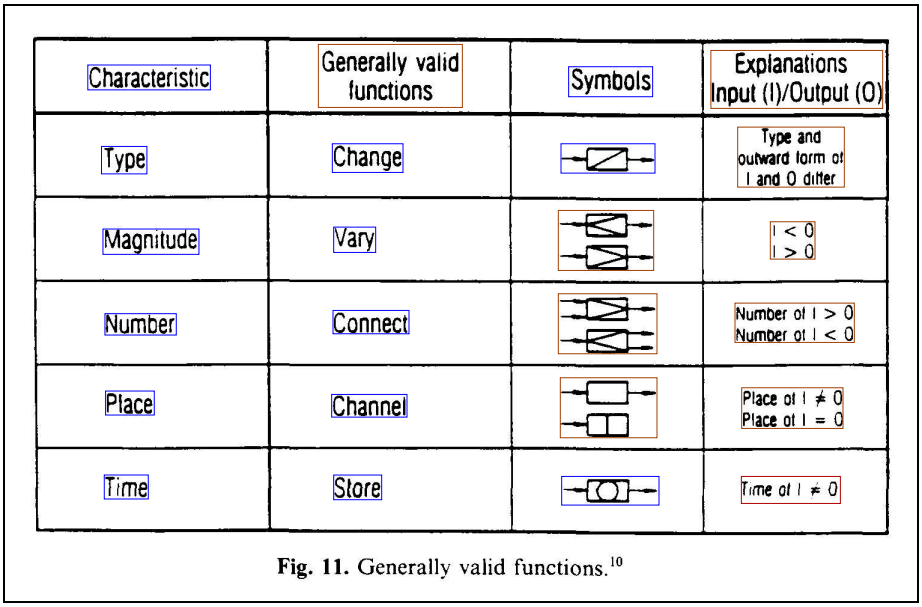


Fig. 11. Generally valid functions.[10]

(a) Author and Handley Algorithm Interpretation

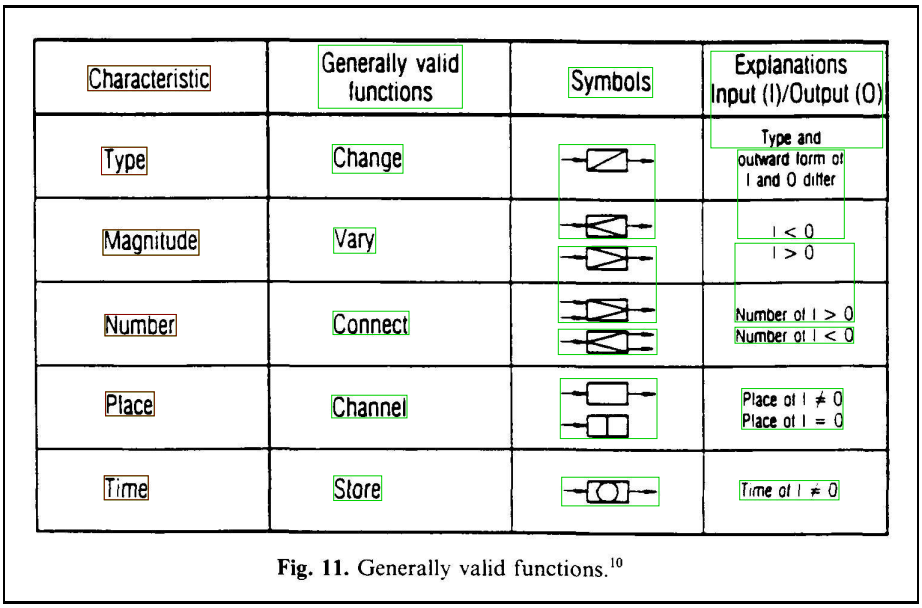


Fig. 11. Generally valid functions.[10]

(b) Output Interpretation from Hu et al.'s Algorithm

Figure E.2: Cell Interpretations for Table in UW-I v002

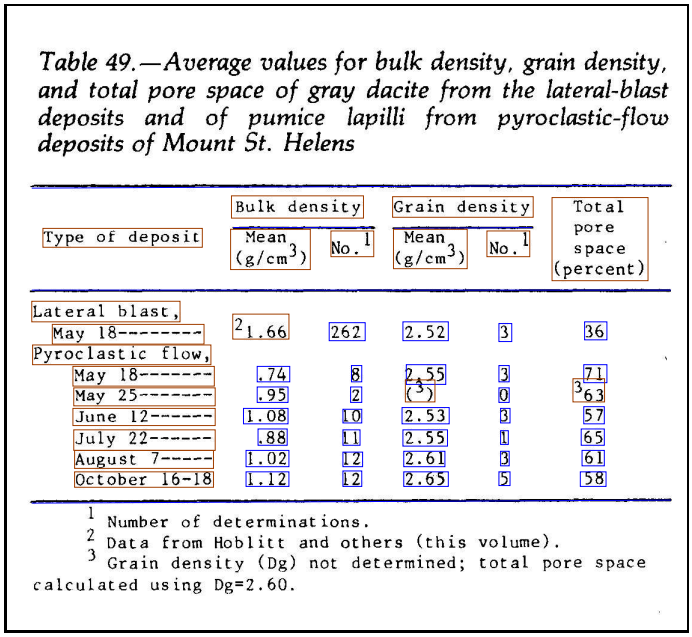


Table 49.—Average values for bulk density, grain density, and total pore space of gray dacite from the lateral-blast deposits and of pumice lapilli from pyroclastic-flow deposits of Mount St. Helens

Type of deposit	Bulk density		Grain density		Total pore space (percent)
	Mean (g/cm^3)	No.[1]	Mean (g/cm^3)	No.[1]	
Lateral blast, May 18--------	[2]1.66	262	2.52	3	36
Pyroclastic flow,					
May 18-------	.74	8	2.55	3	71
May 25-------	.95	2	([3])	0	[3]63
June 12------	1.08	10	2.53	3	57
July 22------	.88	11	2.55	1	65
August 7-----	1.02	12	2.61	3	61
October 16-18	1.12	12	2.65	5	58

[1] Number of determinations.
[2] Data from Hoblitt and others (this volume).
[3] Grain density (Dg) not determined; total pore space calculated using Dg=2.60.

(a) Author

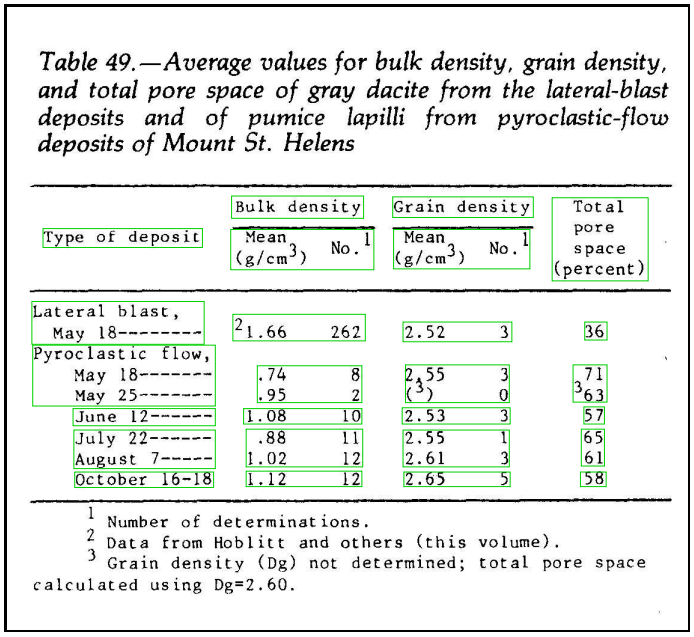


Table 49.—Average values for bulk density, grain density, and total pore space of gray dacite from the lateral-blast deposits and of pumice lapilli from pyroclastic-flow deposits of Mount St. Helens

Type of deposit	Bulk density		Grain density		Total pore space (percent)
	Mean (g/cm^3)	No.[1]	Mean (g/cm^3)	No.[1]	
Lateral blast, May 18--------	[2]1.66	262	2.52	3	36
Pyroclastic flow,					
May 18-------	.74	8	2.55	3	71
May 25-------	.95	2	([3])	0	[3]63
June 12------	1.08	10	2.53	3	57
July 22------	.88	11	2.55	1	65
August 7-----	1.02	12	2.61	3	61
October 16-18	1.12	12	2.65	5	58

[1] Number of determinations.
[2] Data from Hoblitt and others (this volume).
[3] Grain density (Dg) not determined; total pore space calculated using Dg=2.60.

(b) Handley Algorithm

Figure E.3: Author and Handley Algorithm Cell Interpretations for Table in UW-I a038

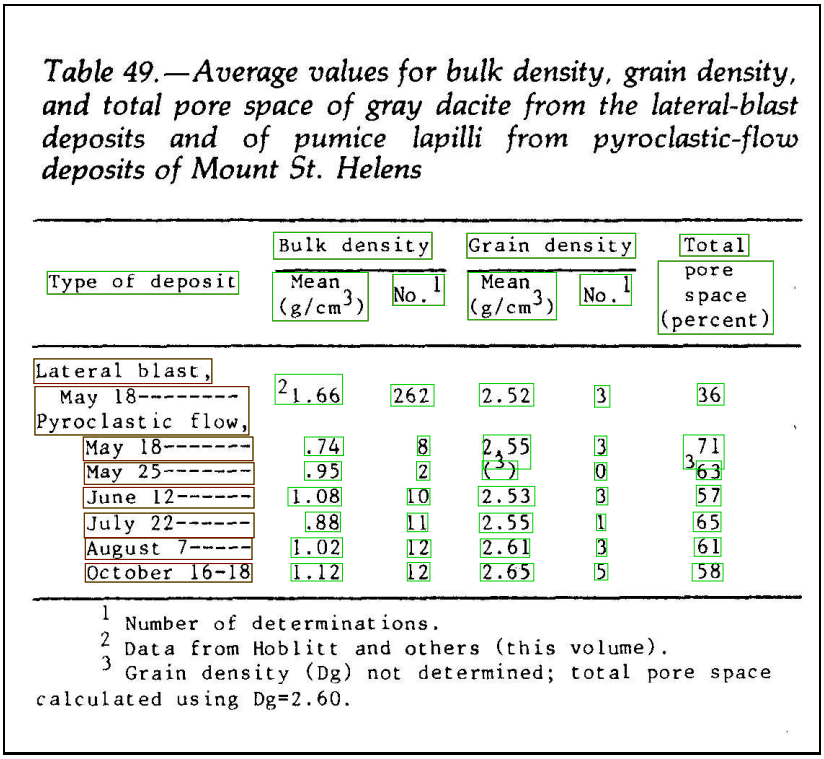


Table 49.—Average values for bulk density, grain density, and total pore space of gray dacite from the lateral-blast deposits and of pumice lapilli from pyroclastic-flow deposits of Mount St. Helens

Type of deposit	Bulk density		Grain density		Total pore space (percent)
	Mean (g/cm^3)	No.[1]	Mean (g/cm^3)	No.[1]	
Lateral blast, May 18--------[2]	1.66	262	2.52	3	36
Pyroclastic flow,					
May 18-------	.74	8	2.55	3	71
May 25-------	.95	2	(3)	0	63
June 12------	1.08	10	2.53	3	57
July 22------	.88	11	2.55	1	65
August 7-----	1.02	12	2.61	3	61
October 16-18	1.12	12	2.65	5	58

[1] Number of determinations.
[2] Data from Hoblitt and others (this volume).
[3] Grain density (Dg) not determined; total pore space calculated using Dg=2.60.

Figure E.4: Hu et al. Algorithm Cell Interpretation for Table in UW-I a038

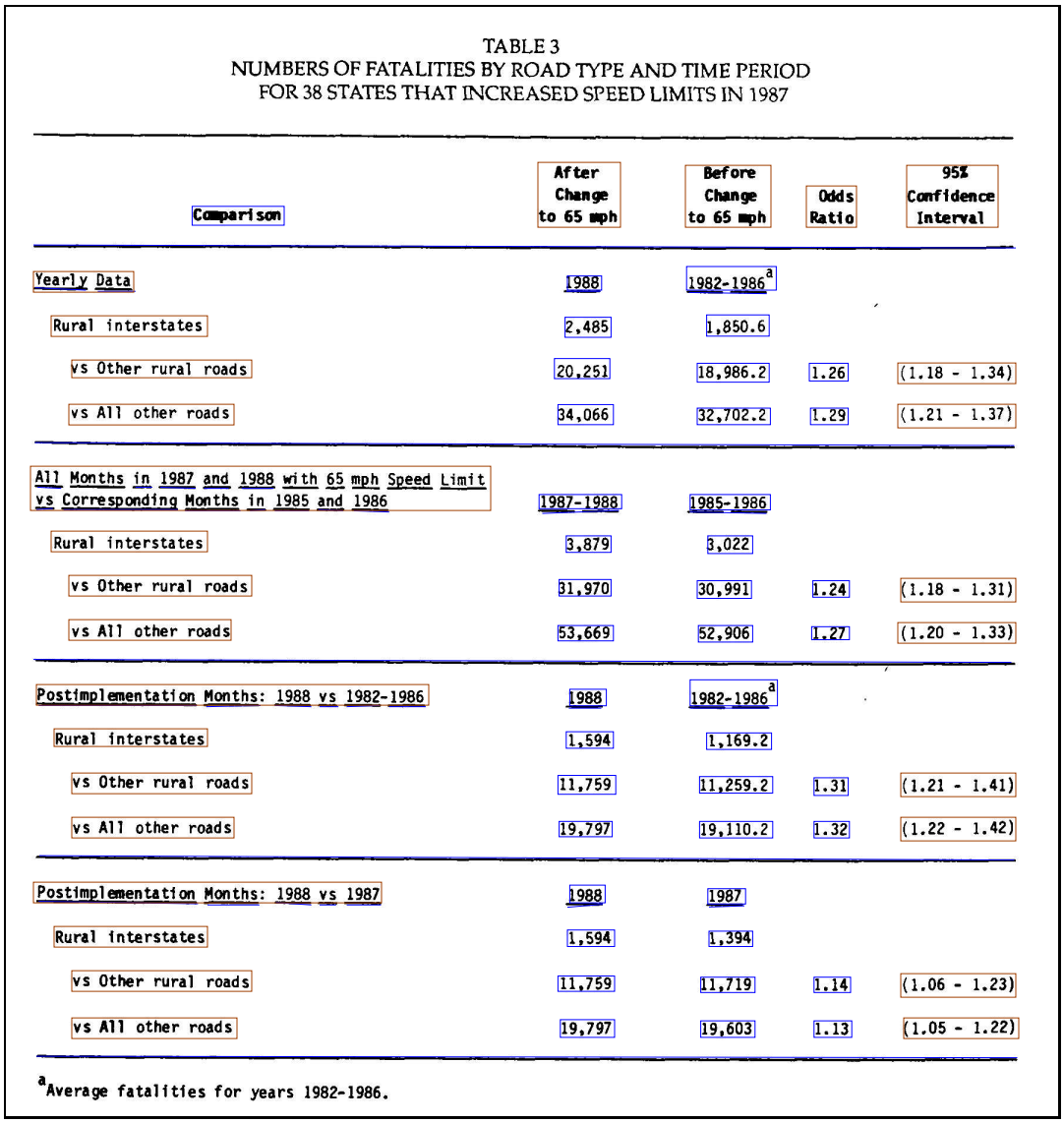


TABLE 3
NUMBERS OF FATALITIES BY ROAD TYPE AND TIME PERIOD
FOR 38 STATES THAT INCREASED SPEED LIMITS IN 1987

Comparison	After Change to 65 mph	Before Change to 65 mph	Odds Ratio	95% Confidence Interval
Yearly Data	1988	1982-1986[a]		
Rural interstates	2,485	1,850.6		
vs Other rural roads	20,251	18,986.2	1.26	(1.18 - 1.34)
vs All other roads	34,066	32,702.2	1.29	(1.21 - 1.37)
All Months in 1987 and 1988 with 65 mph Speed Limit vs Corresponding Months in 1985 and 1986	1987-1988	1985-1986		
Rural interstates	3,879	3,022		
vs Other rural roads	31,970	30,991	1.24	(1.18 - 1.31)
vs All other roads	53,669	52,906	1.27	(1.20 - 1.33)
Postimplementation Months: 1988 vs 1982-1986	1988	1982-1986[a]		
Rural interstates	1,594	1,169.2		
vs Other rural roads	11,759	11,259.2	1.31	(1.21 - 1.41)
vs All other roads	19,797	19,110.2	1.32	(1.22 - 1.42)
Postimplementation Months: 1988 vs 1987	1988	1987		
Rural interstates	1,594	1,394		
vs Other rural roads	11,759	11,719	1.14	(1.06 - 1.23)
vs All other roads	19,797	19,603	1.13	(1.05 - 1.22)

[a]Average fatalities for years 1982-1986.

Figure E.5: Author's Cell Interpretation for Table in UW-I a04g

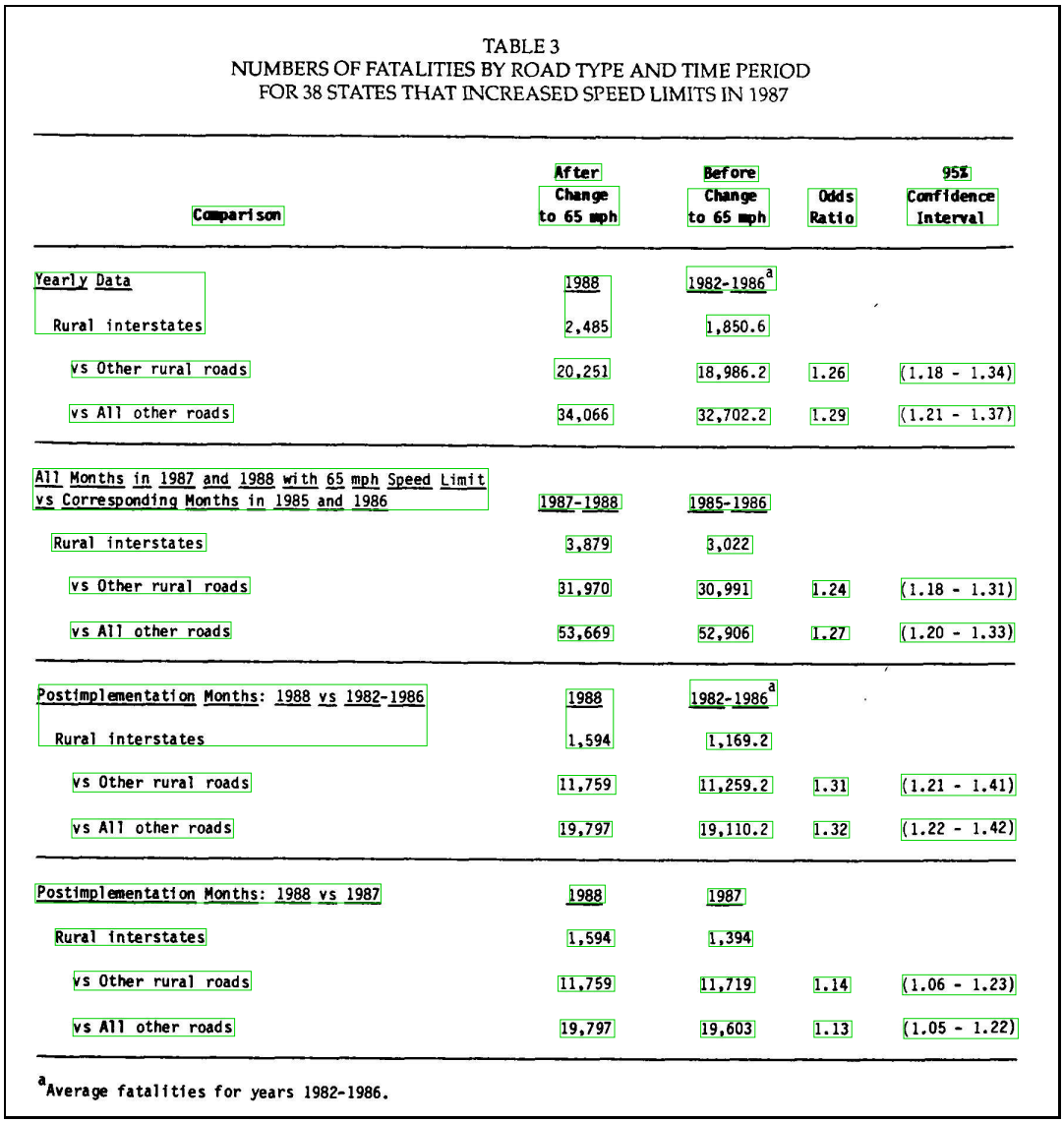


TABLE 3
NUMBERS OF FATALITIES BY ROAD TYPE AND TIME PERIOD
FOR 38 STATES THAT INCREASED SPEED LIMITS IN 1987

Comparison	After Change to 65 mph	Before Change to 65 mph	Odds Ratio	95% Confidence Interval
Yearly Data	1988	1982-1986[a]		
Rural interstates	2,485	1,850.6		
vs Other rural roads	20,251	18,986.2	1.26	(1.18 - 1.34)
vs All other roads	34,066	32,702.2	1.29	(1.21 - 1.37)
All Months in 1987 and 1988 with 65 mph Speed Limit vs Corresponding Months in 1985 and 1986	1987-1988	1985-1986		
Rural interstates	3,879	3,022		
vs Other rural roads	31,970	30,991	1.24	(1.18 - 1.31)
vs All other roads	53,669	52,906	1.27	(1.20 - 1.33)
Postimplementation Months: 1988 vs 1982-1986	1988	1982-1986[a]		
Rural interstates	1,594	1,169.2		
vs Other rural roads	11,759	11,259.2	1.31	(1.21 - 1.41)
vs All other roads	19,797	19,110.2	1.32	(1.22 - 1.42)
Postimplementation Months: 1988 vs 1987	1988	1987		
Rural interstates	1,594	1,394		
vs Other rural roads	11,759	11,719	1.14	(1.06 - 1.23)
vs All other roads	19,797	19,603	1.13	(1.05 - 1.22)

[a]Average fatalities for years 1982-1986.

Figure E.6: Handley Algorithm Cell Interpretation for Table in UW-I a04g

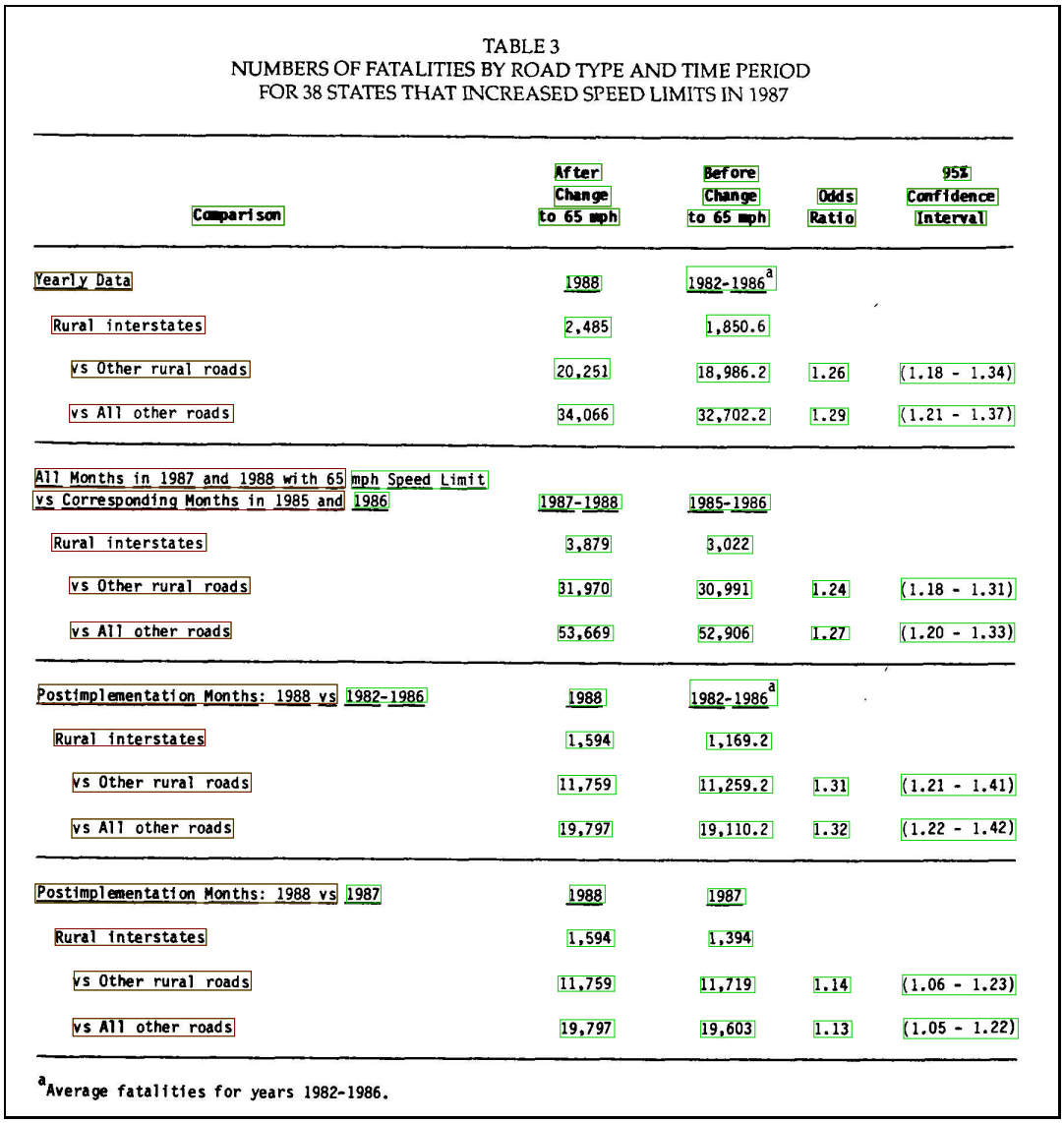


TABLE 3
NUMBERS OF FATALITIES BY ROAD TYPE AND TIME PERIOD
FOR 38 STATES THAT INCREASED SPEED LIMITS IN 1987

Comparison	After Change to 65 mph	Before Change to 65 mph	Odds Ratio	95% Confidence Interval
Yearly Data	1988	1982-1986[a]		
Rural interstates	2,485	1,850.6		
vs Other rural roads	20,251	18,986.2	1.26	(1.18 - 1.34)
vs All other roads	34,066	32,702.2	1.29	(1.21 - 1.37)
All Months in 1987 and 1988 with 65 mph Speed Limit vs Corresponding Months in 1985 and 1986	1987-1988	1985-1986		
Rural interstates	3,879	3,022		
vs Other rural roads	31,970	30,991	1.24	(1.18 - 1.31)
vs All other roads	53,669	52,906	1.27	(1.20 - 1.33)
Postimplementation Months: 1988 vs 1982-1986	1988	1982-1986[a]		
Rural interstates	1,594	1,169.2		
vs Other rural roads	11,759	11,259.2	1.31	(1.21 - 1.41)
vs All other roads	19,797	19,110.2	1.32	(1.22 - 1.42)
Postimplementation Months: 1988 vs 1987	1988	1987		
Rural interstates	1,594	1,394		
vs Other rural roads	11,759	11,719	1.14	(1.06 - 1.23)
vs All other roads	19,797	19,603	1.13	(1.05 - 1.22)

[a]Average fatalities for years 1982-1986.

Figure E.7: Hu et al. Algorithm Cell Interpretation for Table in UW-I a04g

Table 1. Description of the basal friction coefficients for 3 model types. For type I models the friction coefficient μ depends on the spatial variable, for type II models μ is velocity dependent, whereas for type III models the friction coefficient is both, position and velocity dependent

Models Type I	Models Type II	Models Type I ∪ II
1) $\mu = \begin{cases} \mu_F, & \xi_F < \xi \\ \frac{\mu_F - \mu_R}{\xi_F - \xi_R}\xi + \frac{\mu_R\xi_F - \mu_F\xi_R}{\xi_F - \xi_R}, & \xi_R \leqq \xi \leqq \xi_F \\ \mu_R, & \xi < \xi_R \end{cases}$ where μ_F and μ_R are constants independent of ξ.	3) $\mu = \mu_0(1 + c\bar{u})$, where μ_0 and c are constants independent of ξ and u. $\mu = \mu(\bar{u}) = f(\bar{u})\,\mu_0$, where $f(0) = 1$. In particular,	7) $\mu = f(\bar{u}) \begin{cases} \mu_F^0, & \xi_F < \xi \\ \frac{\mu_F^0 - \mu_R^0}{\xi_F - \xi_R}\xi + \frac{\mu_F^0 + \mu_R^0}{2}, & \xi_R \leqq \xi \leqq \xi_F \\ \mu_R^0, & \xi < \xi_R \end{cases}$ where $f(\bar{u})$ is given by model 4), 5), or 6).
2) $\mu = a(\xi - \xi_F) + \mu_F, \quad \xi_R \leqq \xi \leqq \xi_F$ $a = \frac{d\mu}{d\xi} = \frac{\mu_F - \mu_R}{\xi_F - \xi_R}$, where μ_F and a are constants independent of ξ.	4) $f(\bar{u}) = 1 + c\bar{u}$, 5) $f(\bar{u}) = 1 + c\bar{u}^2$, 6) $f(\bar{u}) = \left(1 - \left(\frac{\mu_\infty}{\mu_0}\right) e^{-c\lvert\bar{u}\rvert} + \frac{\mu_\infty}{\mu_0}\right)$, where c, μ_0, μ_∞ are constants independent of $\bar{u}$ and ξ.	8) $\mu = f(\bar{u})\,[a_0(\xi - \xi_F) + \mu_F^0], \quad \xi_R \leqq \xi \leqq \xi_F$, where $f(\bar{u})$ is given by model 4), 5) or 6).

Figure E.8: Author Cell Interpretation for UW-I Table a002

Table 1. Description of the basal friction coefficients for 3 model types. For type I models the friction coefficient μ depends on the spatial variable, for type II models μ is velocity dependent, whereas for type III models the friction coefficient is both, position and velocity dependent

Models Type I	Models Type II	Models Type I ∪ II
1) $\mu = \begin{cases} \mu_F, & \xi_F < \xi \\ \frac{\mu_F - \mu_R}{\xi_F - \xi_R}\xi + \frac{\mu_R\xi_F - \mu_F\xi_R}{\xi_F - \xi_R}, & \xi_R \leqq \xi \leqq \xi_F \\ \mu_R, & \xi < \xi_R \end{cases}$ where μ_F and μ_R are constants independent of ξ.	3) $\mu = \mu_0(1 + c\bar{u})$, where μ_0 and c are constants independent of ξ and u. $\mu = \mu(\bar{u}) = f(\bar{u})\,\mu_0$, where $f(0) = 1$. In particular,	7) $\mu = f(\bar{u}) \begin{cases} \mu_F^0, & \xi_F < \xi \\ \frac{\mu_F^0 - \mu_R^0}{\xi_F - \xi_R}\xi + \frac{\mu_F^0 + \mu_R^0}{2}, & \xi_R \leqq \xi \leqq \xi_F \\ \mu_R^0, & \xi < \xi_R \end{cases}$ where $f(\bar{u})$ is given by model 4), 5), or 6).
2) $\mu = a(\xi - \xi_F) + \mu_F, \quad \xi_R \leqq \xi \leqq \xi_F$ $a = \frac{d\mu}{d\xi} = \frac{\mu_F - \mu_R}{\xi_F - \xi_R}$, where μ_F and a are constants independent of ξ.	4) $f(\bar{u}) = 1 + c\bar{u}$, 5) $f(\bar{u}) = 1 + c\bar{u}^2$, 6) $f(\bar{u}) = \left(1 - \left(\frac{\mu_\infty}{\mu_0}\right) e^{-c\lvert\bar{u}\rvert} + \frac{\mu_\infty}{\mu_0}\right)$, where c, μ_0, μ_∞ are constants independent of $\bar{u}$ and ξ.	8) $\mu = f(\bar{u})\,[a_0(\xi - \xi_F) + \mu_F^0], \quad \xi_R \leqq \xi \leqq \xi_F$, where $f(\bar{u})$ is given by model 4), 5) or 6).

Figure E.9: Handley Algorithm Cell Interpretation for UW-I Table a002

Table 1. Description of the basal friction coefficients for 3 model types. For type I models the friction coefficient μ depends on the spatial variable, for type II models μ is velocity dependent, whereas for type III models the friction coefficient is both, position and velocity dependent

Models Type I	Models Type II	Models Type I ▽ II
1) $\mu = \begin{cases} \mu_F, & \xi_F < \xi \\ \frac{\mu_F - \mu_R}{\xi_F - \xi_R}\xi + \frac{\mu_R\xi_F - \mu_F\xi_R}{\xi_F - \xi_R}, & \xi_R \leqq \xi \leqq \xi_F \\ \mu_R, & \xi < \xi_R \end{cases}$ where μ_F and μ_R are constants independent of ξ.	3) $\mu = \mu_0(1 + c\bar{u})$, where μ_0 and c are constants independent of ξ and u. $\mu = \mu(\bar{u}) = f(\bar{u})\,\mu_0$, where $f(0) = 1$. In particular,	7) $\mu = f(\bar{u}) \begin{cases} \mu_F^0, & \xi_F < \xi \\ \frac{\mu_F^0 - \mu_R^0}{\xi_F - \xi_R}\xi + \frac{\mu_F^0 + \mu_R^0}{2}, & \xi_R \leqq \xi \leqq \xi_F \\ \mu_R^0, & \xi < \xi_R \end{cases}$ where $f(\bar{u})$ is given by model 4), 5), or 6).
2) $\mu = a(\xi - \xi_F) + \mu_F, \quad \xi_R \leqq \xi \leqq \xi_F$ $a = \frac{d\mu}{d\xi} = \frac{\mu_F - \mu_R}{\xi_F - \xi_R}$, where μ_F and a are constants independent of ξ.	4) $f(\bar{u}) = 1 + c\bar{u}$, 5) $f(\bar{u}) = 1 + c\bar{u}^2$, 6) $f(\bar{u}) = \left(1 - \left(\frac{\mu_\infty}{\mu_0}\right) e^{-c\lvert\bar{u}\rvert} + \frac{\mu_\infty}{\mu_0}\right)$, where c, μ_0, μ_∞ are constants independent of $\bar{u}$ and ξ.	8) $\mu = f(\bar{u})\,[a_0(\xi - \xi_F) + \mu_F^0], \quad \xi_R \leqq \xi \leqq \xi_F$, where $f(\bar{u})$ is given by model 4), 5) or 6).

Figure E.10: Hu et al. Algorithm Cell Interpretation for UW-I Table a002