Inferring Templates from Spreadsheets

by

Seyed Kamrooz Ghazinour Naini

Submitted in partial fulfilment of the requirements for the degree
of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
September 2011

DALHOUSIE UNIVERSITY

FACULTY OF COMPUTER SCIENCE

The undersigned hereby certify that they have read and recommend to the Faculty of Graduate Studies for acceptance a thesis entitled "Inferring Templates from Spreadsheets" by Seyed Kamrooz Ghazinour Naini in partial fulfilment of the requirements for the degree of Master of Computer Science.

Dated: September 15, 2011

Supervisor:	_________________________________
Readers:	_________________________________
	_________________________________

DALHOUSIE UNIVERSITY

DATE: September 15, 2011

AUTHOR: Seyed Kamrooz Ghazinour Naini

TITLE: Inferring Templates from Spreadsheets

DEPARTMENT OR SCHOOL: Faculty of Computer Science

DEGREE: MCSc CONVOCATION: May YEAR: 2012

____
Signature of Author

Table of Content

List of Tables

List of Figures

Abstract

Spreadsheets for critical applications, such as financial reporting, are widely created and used by many people with no expertise in programming or software development. It is well known, however, that creating spreadsheets is an error-prone process. Several methodologies have been designed to reduce these errors. In this thesis we characterise the patterns and functional relationships among the formula cells and the corresponding data cells that commonly occur in spreadsheets, and show how the patterns occurring in a given sheet can be generalised to produce a template structure representing the family of spreadsheets of which the given sheet is a member. Finally, we show how this generalisation can be translated into an L-sheets program from which instances of this family can be generated.

List of Abbreviations and Symbols Used

LANPAR	LANguage for Programming Arrays at Random
VisiCalc	Visible Calculator
WYSIWYT	What You See Is What You Test
ViTSL	Visual Template Specification Language

Acknowledgement

My utmost gratitude goes to my supervisor, Dr. Phil Cox who has the attitude and substance of a genius. I would like to thank him for his expertise, kindness, patience and precision with me in all those long discussions that we had in his office about the technical issues of my thesis. Also I would like to thank him for teaching me how to convey my ideas in writing.

I would like to thank Dr.Vlado Keselj and Dr.Karen Jin for being members of the examining committee.

Finally, I would like to thank my mother who encouraged and supported me throughout my studies.

Chapter 1 Introduction

Spreadsheets are mathematical software systems, used in many areas such as finance, education and science. Nowadays, many people use spreadsheets as necessary tools for doing their tasks. For example, employees use them to manage their work and give reports to their bosses and managers, use them to evaluate their companies' performance. There are many reasons that spreadsheet applications are popular. They are relatively easy to understand and have flexible and user-friendly environments. Spreadsheets are capable of a wide range of tasks from simple household budgets to complex financial forecasting and modeling of complex engineering systems.

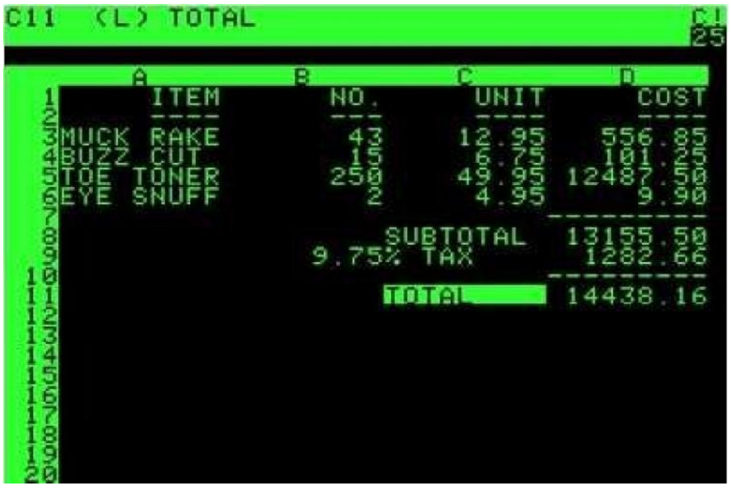
1.1 History of Spreadsheets

In 1961, Mattessich proposed and developed a computerized spreadsheet system, which he successfully applied to accounting and budgeting computations. Like spreadsheets, Mattessich's system arranged formulae and data in arrays; however, these arrays were not interactive, but processed in batch mode by a FORTRAN program [1].

After Mattessich's attempt, Pardo and Landau co-invented LANPAR (LANguage for Programming Arrays at Random) in 1969. Like Mattessich's system, LANPAR was a batch processing program for arrays of formulae and data, but because it was customized for business applications, achieved greater commercial success, and was used for budgeting at companies such as Bell Canada, AT&T, and General Motors. The main contribution of LANPAR was an algorithm for compiling grid-based computations, which received a US patent (no. 4,398,249) in August 1982 [2].

Although Mattessich, Pardo and Landau were the forefathers of spreadsheets systems, none of them influenced spreadsheet concepts and popularity more than Bricklin and Frankston, the inventors of VisiCalc. Dan Bricklin, who was a Harvard Business School student, thinking about automation of the kinds of grid-based presentation of calculations which he was required to do as a business student, created the first prototype of his spreadsheet system in 1978. The program let the users input and edit data in a matrix, composed of five columns and twenty rows [3]. Bricklin then partnered with Bob Frankston whose programming expertise significantly improved the efficiency of Bricklin's implementation, leading to a viable PC-based implementation of Bricklin's original concept.

Following the addition of Daniel Fylstra, a third partner with marketing expertise, the team established Software Arts Corporation in 1979 and implemented a commercial version, called VisiCalc ("visible calculator"), released for the Apple II personal computer in 1979 [4]. VisiCalc enjoyed considerable commercial success, selling close to one million copies [3]. Figure 1.1, illustrates the VisiCalc environment.

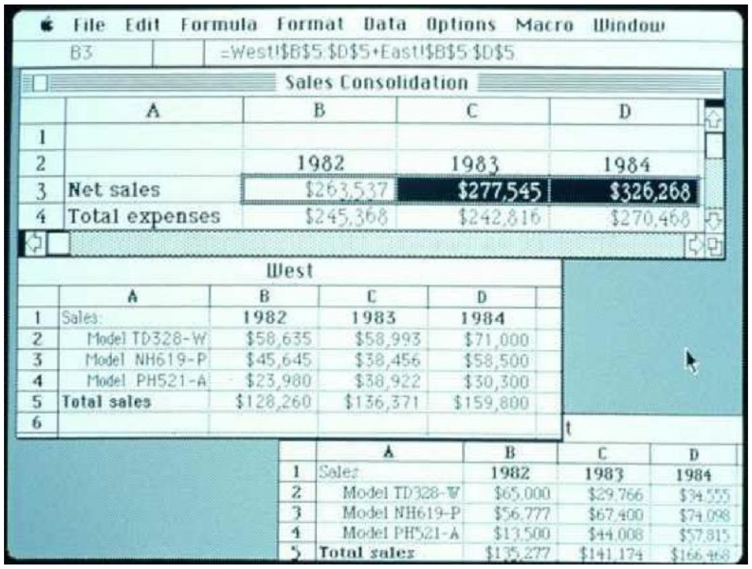


```
C11  (L) TOTAL                                  C!
                                                25
         A         B          C          D
 1       ITEM        NO.       UNIT       COST
 2       ----        ---       ----       ----
 3MUCK  RAKE          43      12.95     556.85
 4BUZZ  CUT           15       6.75     101.25
 5TOE  TONER         250      49.95   12487.50
 6EYE  SNUFF           2       4.95       9.90
 7                                   ---------
 8                     SUBTOTAL       13155.50
 9                9.75% TAX            1282.66
10                                   ---------
11                   TOTAL            14438.16
12
13
14
15
16
17
18
19
20
```

Figure 1.1 VisiCalc environments from [5]

In 1980, Kapor and Sachs developed Lotus 1-2-3 spreadsheet system. Although they added more tools and facilities such as cell ranging and macro definition, Lotus 1-2-3 did not add any special innovative feature to the core concept of VisiCalc [3].

After Lotus 1-2-3, Microsoft was the next to produce a spreadsheet system, introducing Excel 1.0 for the Apple Macintosh in 1984, the first spreadsheet system for a graphical operating system. With the release of the Windows operating system in 1987, Excel, which was prepared for Windows, took over Lotus 1-2-3's place as number one in the market. Figure 1.2 illustrates the first version of Excel.



B3 =West!$B$5:$D$5+East!$B$5:$D$5

Sales Consolidation

	A	B	C	D
1				
2		1982	1983	1984
3	Net sales	$263,537	$277,545	$326,268
4	Total expenses	$245,368	$242,816	$270,468

West

	A	B	C	D
1	Sales:	1982	1983	1984
2	Model TD328-W	$58,635	$58,993	$71,000
3	Model NH619-P	$45,645	$38,456	$58,500
4	Model PH521-A	$23,980	$38,922	$30,300
5	Total sales	$128,260	$136,371	$159,800
6				

	A	B	C	D
1	Sales:	1982	1983	1984
2	Model TD328-W	$65,000	$29,766	$34,555
3	Model NH619-P	$56,777	$67,400	$74,098
4	Model PH521-A	$13,500	$44,008	$57,815
5	Total sales	$135,277	$141,174	$166,468

Figure 1.2 Excel 1.0 for Apple Macintosh from [6]

1.2 Errors in Spreadsheets

According to a census conducted by the U.S Bureau of Labor statistics in 2005, 11 million people used spreadsheets in the US while there were only 2.75 million computer programmers [7]. Researchers have estimated that each year, millions of new users from a wide range of age groups and education levels become spreadsheet users. Therefore, a rising area of research is focused on giving facilities to the end-users of spreadsheets systems, in order to make spreadsheets more powerful and more accurate to less prone to errors.

The majority of research in this area has been focused on creating new methods or algorithms in order to optimize or enhance spreadsheet environments. However, researchers must be aware that the majority of spreadsheets users are end-users (which novices or expert for working with spreadsheets), not software engineers [8]. Software engineering practices are pivotal in ensuring correct spreadsheets, as they enforce design, development, and rigorous testing phases. End-users, on the other hand, tend to ignore the design and testing phases. Empirical studies have repeatedly shown that end-users are reluctant to use, or outright reject, software engineering methods for spreadsheet development [8], one of the main reasons for the high incidence of errors in spreadsheets.

Error-proneness of spreadsheets can cause catastrophic effects. For example a simple error in value conversion in the $320M Mars Climate Orbiter, caused one software component to send data to the thrusters in pounds whereas the Orbiter was expecting it in newtons, resulting in huge financial damage. Similarly, in another project, misplaced parentheses in a formula caused a profit of $200M to be computed, when the profit was actually only $25M [9].

Consequently, a major focus of research is reducing the error-proneness of spreadsheets. One of the approaches has been to develop algorithms or tools in spreadsheets [10,11,12,13,14]. These algorithms or tools have different strategies for dealing with errors, ranging from automatically correcting them, providing the user with suggestions for how to fix the errors, to simply alerting the user to the problems [15,16,17,18,19,20].

Another popular approach is to extract the underlying structure of a spreadsheet in order to helps the users to :

- Detect irregularities, which may signify errors.
- Help users to more easily understand the structure and purpose of the spreadsheet.

- Adopt a more object-oriented approach to spreadsheets.

Many errors in spreadsheet formulae might be avoided if spreadsheets are created from correct structures (templates). That is, if the user first designs the relations between formulae and data cells and then creates spreadsheets from such templates, the resulting sheets may be less likely to include errors. While such a template can be manually extracted from an existing spreadsheet, spreadsheet end-users are likely to be reluctant to do so, for two reasons:

- Extracting templates from spreadsheets is time-consuming especially for large spreadsheets.
- Although an end-user may be familiar with the application domain, he or she is unlikely to understand the software engineering principles required to create a template.

There are several algorithms for detecting errors in and inferring templates from spreadsheets. We will review some of the error-detection algorithms and two methodologies of inferring templates from spreadsheets in chapter 2. Chapter 3 presents the main body of our work. We propose a novel methodology to infer templates from existing spreadsheets. Chapter 4 compares our methodology with the related work and suggests some possible future work.

Chapter 2 Detection and Prevention of Errors in Spreadsheets

As discussed in the first chapter, spreadsheets are user-friendly applications for end-users, ranging in experience from novice to expert. These end-users are not necessarily computer scientists and most of them are not familiar with programming languages or software engineering principles such as testing. Therefore, existing spreadsheets contain many errors [13,21,22,23,24]. Some auditing papers report that 90% or more of spreadsheets contain errors [25]. Some of the consequences, as reported in [14], include financial damages, loss of share value, career damages, and loss of shareholder confidence.

Consequently, tools have been developed to help the user input the correct data and formulae into the correct cells. Likewise, algorithms have been created to help the user detect errors in spreadsheets.

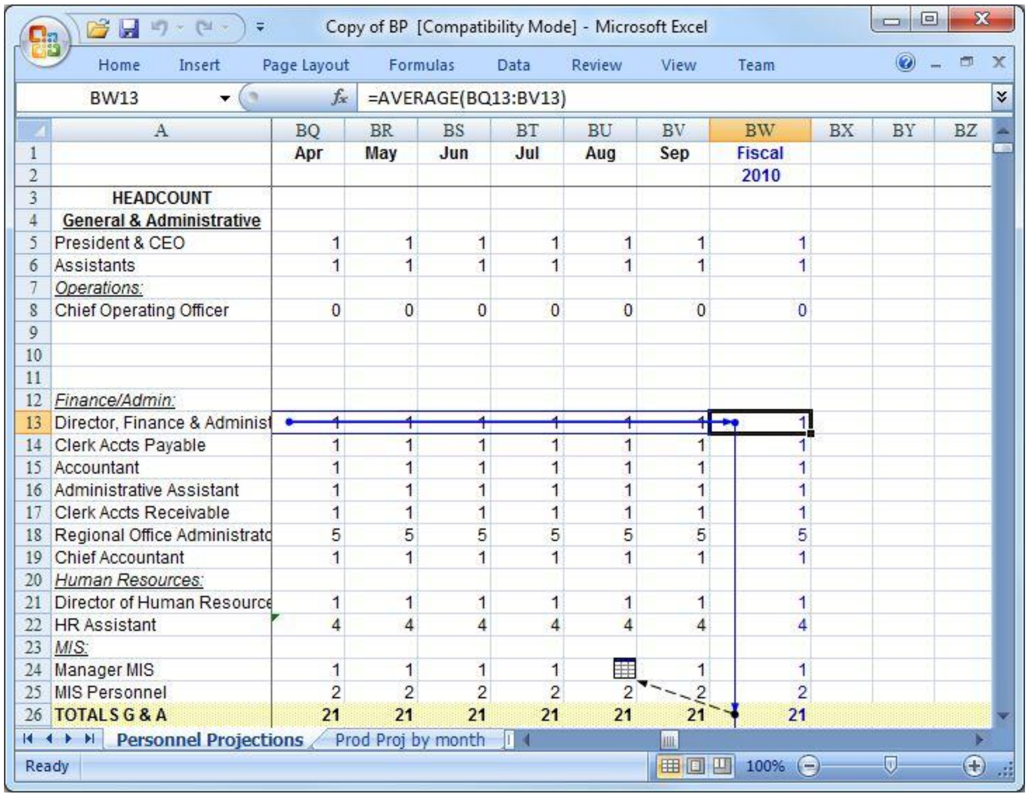
These algorithms and tools can be classified into three categories, which we will discuss in the next three subsections:

- Auditing tools [18, 26, 27, 28]: these help the user input the data and formulae into the correct cells.
- Testing tools and testing strategies [21,29,30,31]: used for debugging.

  Error detection algorithms [8,13,16]: these either detect errors, or help the user to detect them.

2.1 Auditing Tools

Auditing tools can help users find dependencies among cells. For example, in Figure 2.1, the box around cells BR5 to BR25 together with the arrow pointing to cell BR26 show that BR26 contains a formula that refers to the cells in the box. Similarly, cell BW13, refers to cells BQ13 to BW13, and cell BW13 is used in BW26. These auditing tools make visible the dependencies between cells, which would otherwise be hidden.



BW13 =AVERAGE(BQ13:BV13)

	A	BQ	BR	BS	BT	BU	BV	BW	BX	BY	BZ
1		Apr	May	Jun	Jul	Aug	Sep	Fiscal			
2								2010			
3	HEADCOUNT										
4	General & Administrative										
5	President & CEO	1	1	1	1	1	1	1			
6	Assistants	1	1	1	1	1	1	1			
7	Operations:										
8	Chief Operating Officer	0	0	0	0	0	0	0			
9											
10											
11											
12	Finance/Admin:										
13	Director, Finance & Administ	1	1	1	1	1	1	1			
14	Clerk Accts Payable	1	1	1	1	1	1	1			
15	Accountant	1	1	1	1	1	1	1			
16	Administrative Assistant	1	1	1	1	1	1	1			
17	Clerk Accts Receivable	1	1	1	1	1	1	1			
18	Regional Office Administrato	5	5	5	5	5	5	5			
19	Chief Accountant	1	1	1	1	1	1	1			
20	Human Resources:										
21	Director of Human Resource	1	1	1	1	1	1	1			
22	HR Assistant	4	4	4	4	4	4	4			
23	MIS:										
24	Manager MIS	1	1	1	1		1	1			
25	MIS Personnel	2	2	2	2	2	2	2			
26	TOTALS G & A	21	21	21	21	21	21	21			

Figure 2.1 Microsoft Excel 2007 built-in auditing tool

Although there are many features in the auditing tools built into some spreadsheet systems (such as Microsoft Excel), many third parties provide add-ons to these auditing tools to make them more powerful.

As discussed, auditing tools just act as a guide for the spreadsheet user and cannot detect or correct errors themselves. Some auditing tools available for Microsoft Excel 2007 are:

- ExcelSmartTools [27] Using the auditing tools provided with Excel, the user selects one cell at a time and requests that its dependencies be displayed.

ExcelSmartTool improves this feature by allowing the user to select several cells and have their dependencies displayed.

- ExcelSpreadsheet Auditor 2 [31] compares formula cells of a worksheet and highlights in the same colour cells which have similar formulae.

2.2 Testing Tools

Most expert programmers claim that the testing and debugging of code takes the majority of their time. A study conducted in the US by the National Institute of Standards and Technology shows that software engineers usually spend 70-80% of their time testing and debugging code, and on average, it takes 17.4 hours to find and fix an error [29]. Clearly, testing and debugging are significant and time-consuming tasks for programmers, and are likely to be equally so for spreadsheet users. Spreadsheet users, however, are not programmers or software engineers, so have little understanding of the importance of these aspects of software development. Furthermore, their focus is on the details of their problem domain (e.g. accounting), not programming details: they just want to compute the numbers they need as quickly as possible. So if spreadsheet testing tools are to be effective, these tools must make the process as simple and intuitive as possible.

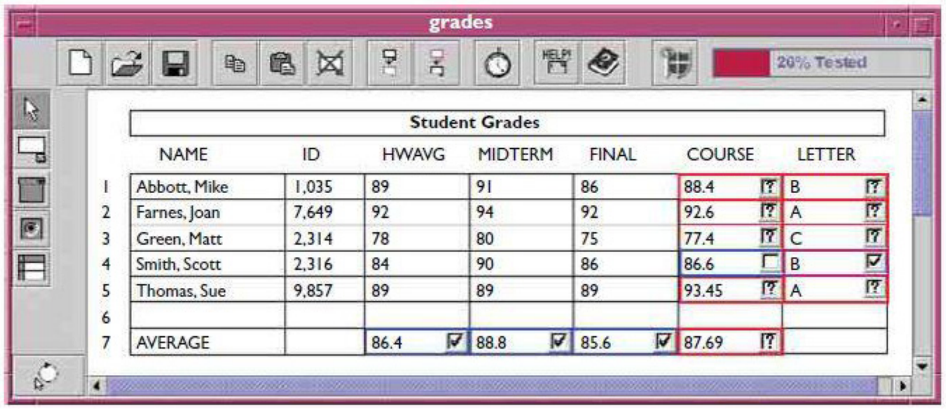
There are three main questions that spreadsheet testing must address:

- Does the spreadsheet contain errors?
- In which cells are the errors located?
- How can the errors be corrected?

In the following subsections, we will introduce some popular methodologies for testing spreadsheets.

2.2.1 “What You See Is What You Test” (WYSIWYT)

The WYSIWYT methodology [21] helps the user to keep track of the extent to which each cell in a spreadsheet has been tested. When the user observes that a correct value has been computed in a cell, he or she can validate it. When a cell is validated, the definition-use test adequacy criterion [29] is used to determine the degree of testedness of related cells, which is indicated by colouring cell borders in shades ranging from red to blue.



Student Grades

	NAME	ID	HWAVG	MIDTERM	FINAL	COURSE	LETTER
1	Abbott, Mike	1,035	89	91	86	88.4 ?	B ?
2	Farnes, Joan	7,649	92	94	92	92.6 ?	A ?
3	Green, Matt	2,314	78	80	75	77.4 ?	C ?
4	Smith, Scott	2,316	84	90	86	86.6 ☐	B ✓
5	Thomas, Sue	9,857	89	89	89	93.45 ?	A ?
6							
7	AVERAGE		86.4 ✓	88.8 ✓	85.6 ✓	87.69 ?	

Figure 2.2 Student grades spreadsheet (From [28] page 54)

For example consider Figure 2.2 which is a student grades spreadsheet. The values displayed in the cells with checkmarks have been validated for the current inputs. If a cell's checkbox is empty or contains a question mark, its value has not been validated for the current inputs. From the border colors, the user is kept informed of which areas of the spreadsheet are tested and to what extent. Thus, in Figure 2.2, the "Letter" cell of row 4 is partially blue (purple), because some of the dependencies ending at that cell have now been tested, but the "Final" cell of row 7 is blue which means it is fully tested.

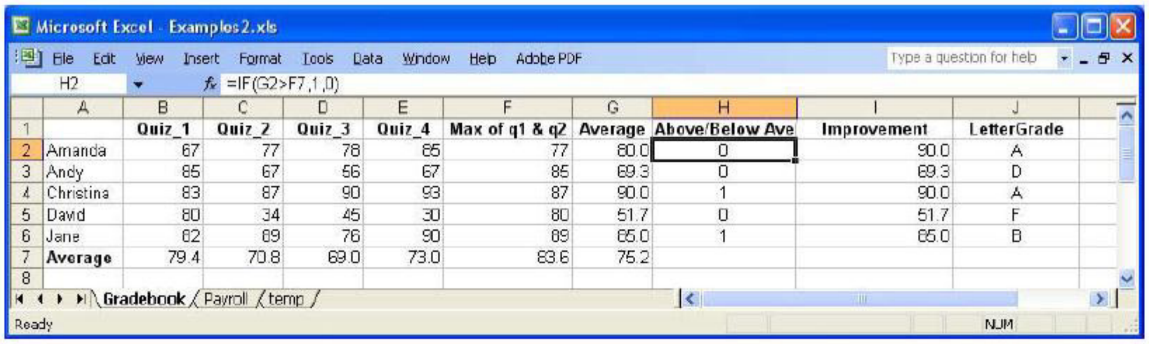
Testing a program “perfectly” (well enough to guarantee detecting all faults) generally requires many test cases, which the user may be unable to generate manually.

Therefore, several mechanisms have been proposed to generate test cases. “Help me Test”, for example, first constructs a chain of dependencies for a specific cell, then iteratively explores portions of these chains, applying constrained linear searches over the spreadsheet’s input space and data gathered through iterative executions [29]. With this process “Help me test” derives test input values and suggests them to the user. Another mechanism for automatically generating test cases will be described in detail in subsection 2.2.3.

2.2.2 Goal-Directed Debugging

Goal-directed debugging is a spreadsheet debugging methodology [32]. In this technique, if the user sees an incorrect value for a cell, he or she provides the correct value, or range of values, for the cell. Then the system uses this information to deduce possible values or formula changes to cells that directly influence the value of the erroneous cell and to achieve the correct value in the target cell [32]. After that the system uses heuristics to rank the change suggestions from most likely to least likely. For example consider Figure 2.3, which shows a spreadsheet used to store the grades of students in a course. A value of 1 in column H indicates that a student’s mark is above the class average for the course in cell G7. The user, seeing that the value for H2 is incorrect, right-clicks on the cell and chooses “value expectation” from the pop-up menu that appears. This produces the dialogue in Figure 2.4, in which the user enters the value that he or she thinks the cell should have. The system generates change suggestions based on this value, and displays them in a pop-up menu if the cell is again right-clicked, as shown in Figure 2.5. The user can choose an appropriate item, ask for more suggestions or ignore the suggestion values.

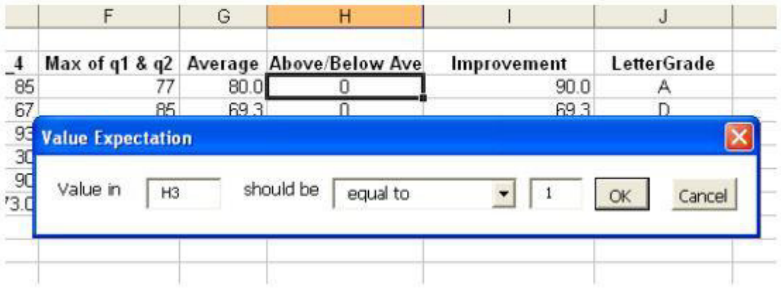
This methodology works well for the spreadsheets that have one error, but its effectiveness decreases as the number of errors increases. Also, since change suggestions are based on values provided by the user, their usefulness is limited if the user makes mistakes. On the other hand, the users claim that most of the time the correct suggestion is ranked within the top two [33].


Microsoft Excel - Examples2.xls
File Edit View Insert Format Tools Data Window Help Adobe PDF
Type a question for help
Gradebook Payroll temp
Ready
NUM

H2 =IF(G2>F7,1,0)

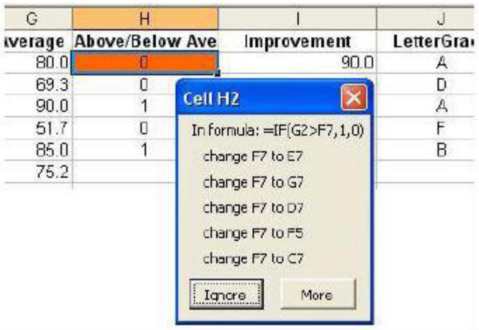
	A	B	C	D	E	F	G	H	I	J
1		Quiz_1	Quiz_2	Quiz_3	Quiz_4	Max of q1 & q2	Average	Above/Below Ave	Improvement	LetterGrade
2	Amanda	67	77	78	85	77	80.0	0	90.0	A
3	Andy	85	67	56	67	85	69.3	0	69.3	D
4	Christina	83	87	90	93	87	90.0	1	90.0	A
5	David	80	34	45	30	80	51.7	0	51.7	F
6	Jane	82	89	76	90	89	85.0	1	85.0	B
7	Average	79.4	70.8	69.0	73.0	83.6	75.2			
8										

Figure 2.3 Store students' grades spreadsheet (from [32] page 2)



Value in H3 should be equal to 1

Figure 2.4 The user suggests a value (from [33] page 132)

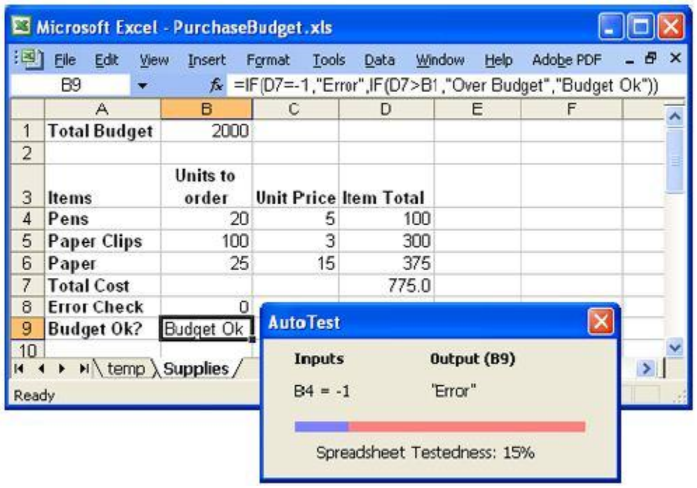


In formula: =IF(G2>F7,1,0)
change F7 to E7
change F7 to G7
change F7 to D7
change F7 to F5
change F7 to C7

Figure 2.5 Change suggestion based on H2 value (from [33] page 132)

2.2.3 AutoTest

AutoTest is a system for generating a test case automatically [34]. When the user selects a cell to be tested, the system generates test values for input cells on which the selected cell depends, and displays the value that the selected cell would have if these input values were imposed. The user marks the generated test case as valid or invalid, to indicate that the displayed value does or does not match the expected value, and in the latter case, modifies the formula. If the user cannot decide whether the displayed value is correct or not, he or she ignores the test case. As testing proceeds, the system displays a progress bar as shown in Figure 2.6 to indicate the degree to which the spreadsheet is tested.


Microsoft Excel - PurchaseBudget.xls
File Edit View Insert Format Tools Data Window Help Adobe PDF
temp Supplies
Ready

B9 =IF(D7=-1,"Error",IF(D7>B1,"Over Budget","Budget Ok"))

	A	B	C	D	E	F
1	Total Budget	2000				
2						
3	Items	Units to order	Unit Price	Item Total		
4	Pens	20	5	100		
5	Paper Clips	100	3	300		
6	Paper	25	15	375		
7	Total Cost			775.0		
8	Error Check	0				
9	Budget Ok?	Budget Ok				
10						

AutoTest

Inputs	Output (B9)
B4 = -1	"Error"

Spreadsheet Testedness: 15%

Figure 2.6 Testing with AutoTest (from [34] page.133)

2.2.4 Test-Driven Goal-Directed Debugging

This approach, a combination of Goal-Directed debugging and AutoTest algorithms [33], automatically generates test cases for a spreadsheet cell, and depending on feedback from the user, suggests a new set of test cases for another cell, or proposes changes for the tested cell. As discussed in previous subsections (2.2.2 and 2.2.3) both AutoTest and

Goal-Directed debugging have some problems, which the other addresses.

The Test-Driven Goal-Directed debugging method applies these two methods as follows [33]:

- AutoTest creates values for input cells to test a particular formula cell.
- The user confirms or rejects the value for the formula cell that results from the test values.
- If the user rejects the formula cell value, Goal-Directed debugging generates change suggestions for related cells.
- All the test cases and change suggestions are stored for the user to view.

Experiments with this method indicate that it is a more efficient and effective approach than either of the two methods alone. For more information, the reader is encouraged to review [33].

2.3 Error-detecting Algorithms

As discussed above auditing tools simply show the hidden dependencies between cells and testing tools help the user to make sure a spreadsheet is thoroughly tested, and guide the user in his or her search for errors. In the following subsections, we discuss algorithms for automatically detecting errors.

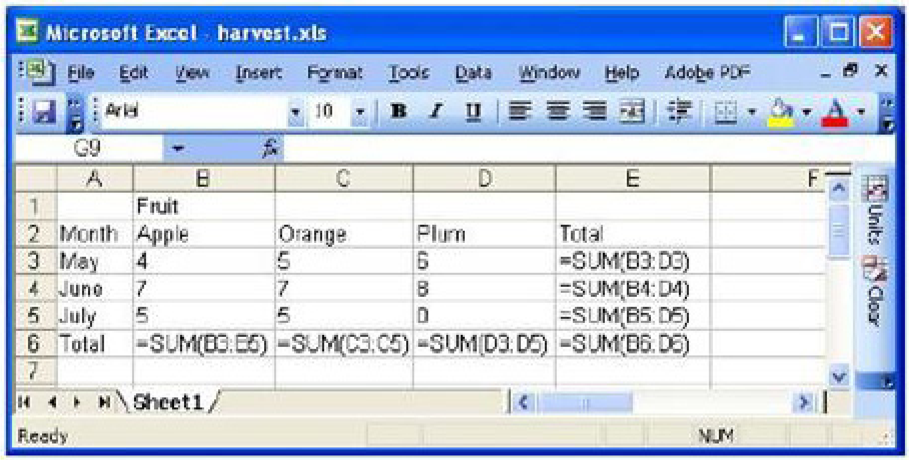
2.3.1 UCheck

UCheck [13] is an algorithm for automatically detecting errors in spreadsheets. It is based on unit reasoning, extracting information from labels and headers to check the consistency of cells [15,20].

As Figure 2.7 shows, cell D4 contains a number which counts plums harvested in June, information which can be inferred from the labels in cells D2 and A4.

UCheck classifies spreadsheet cells into the following groups [13]:

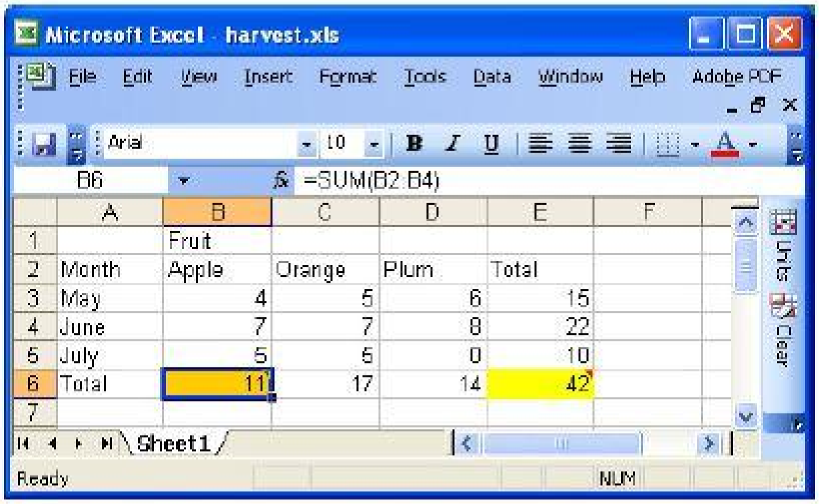
- Header: those cells that contain strings which describe the other cells.
- Footer: usually cells at the end of rows or bottoms of columns containing some sort of aggregation formulae.
- Core: the data cells.
- Filler: empty cells that separate regions in a spreadsheet.


Microsoft Excel - harvest.xls
File Edit View Insert Format Tools Data Window Help Adobe PDF
Arial 10
G9
Sheet1
Ready
NUM
Units
Clear

	A	B	C	D	E	F
1		Fruit				
2	Month	Apple	Orange	Plum	Total	
3	May	4	5	6	=SUM(B3:D3)	
4	June	7	7	8	=SUM(B4:D4)	
5	July	5	5	0	=SUM(B5:D5)	
6	Total	=SUM(B3:B5)	=SUM(C3:C5)	=SUM(D3:D5)	=SUM(B6:D6)	
7						

Figure 2.7 Extracting information from labels and headers (From [13] page 72)

UCheck provides a header inference framework to extract headers from spreadsheets. Since the layout of spreadsheets varies widely, it is impossible for a single algorithm to work equally well in all cases. Therefore, the framework consists of four algorithms which detect special arrangements of cells to classify them, and infer the headers to be used in the following operations:


Microsoft Excel - harvest.xls

B6 =SUM(B2:B4)

	A	B	C	D	E	F
1		Fruit				
2	Month	Apple	Orange	Plum	Total	
3	May	4	5	6	15	
4	June	7	7	8	22	
5	July	5	5	0	10	
6	Total	11	17	14	42	
7						

Figure 2.8 Wrong range for B6 formula cell (From [13] page 87)

Fence identification: detects blank cells (soft fence) in the data regions. Also this algorithm has the ability to detect hard fences, those columns that are repeated with their headers.

Content-Based cell classification: classifies cells according to their values. For example if a cell contains a number, it is considered as a core cell; if it contains a string it is considered to be a header. For example, in Figure 2.7, cells D3, D4, D5 are core cells.

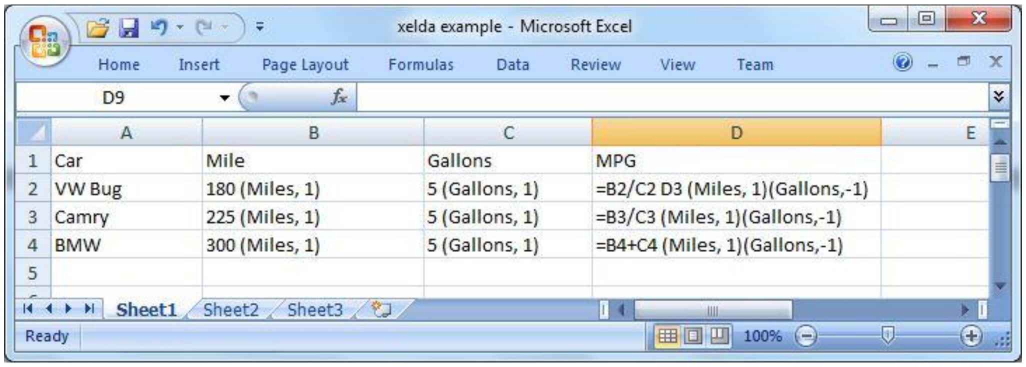
Region-based cell classification: infers the types of cells according to their positions. For example if a cell is located in the leftmost column of a spreadsheet and has a string value, it is classified as a header. For example, in Figure 2.7, cells A2-A6 are header cells.

Footer to core expansion: detects aggregation formulae, then classifies all data cells which are used in formula cells as core cells. For example, in Figure 2.7, cell B6 is a footer cell and cells B3, B4 and B5 are core cells.

Once header cells have been identified, UCheck uses them to infer the units of the formula cells. If a formula cell does not have a well formed unit, there is an error with several possible causes, such as an incorrect reference to another cell or an incorrect range. For example, in Figure 2.8, the headers of B2, B3 and B4 are Fruit, Month[May]&Fruit[Apple] and Month[June]&Fruit[Apple] respectively. Since the formula in B6 is SUM(B2:B4), UCheck infers for B6 the header Fruit | Month[May]&Fruit[Apple] | Month[June]&Fruit[Apple] which is not well-formed since the last two alternatives are compatible with each other but not with the first. This error results from the incorrect range B2:B4 in the formula of B6.

2.3.2 XeLda

XeLda [26] checks the dimensions of formula cells for any inconsistencies. Dimensions are units of measurement expressed as types familiar to end users [8]. For example, for a dimension such as length, there are several units of measurement such as cm, metre and ft. XeLda requires users to annotate all cells with units, including formula cells. For example in Figure 2.9, the cells B2 and B3 and B4 are annotated (Miles,1)



	A	B	C	D	E
1	Car	Mile	Gallons	MPG	
2	VW Bug	180 (Miles, 1)	5 (Gallons, 1)	=B2/C2 D3 (Miles, 1)(Gallons,-1)	
3	Camry	225 (Miles, 1)	5 (Gallons, 1)	=B3/C3 (Miles, 1)(Gallons,-1)	
4	BMW	300 (Miles, 1)	5 (Gallons, 1)	=B4+C4 (Miles, 1)(Gallons,-1)	
5					

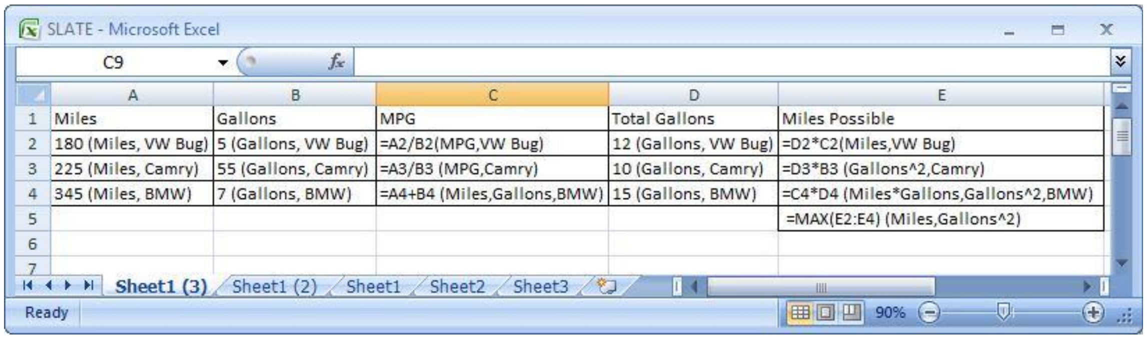
Figure 2.9 XeLda example (from [8] page 280)

meaning that the dimension is "Mile" with exponent 1. Suppose, the user decides to annotate D4 (Mile,1)(Gallon,-1) because the column's header is MPG (Mile per Gallon). Since the formula in D4 is B4 + C4 and B4 and C4 are annotated (Mile, 1) and (Gallons, 1), XeLda infers for D4 the dimension (Mile, 1) (Gallon,1) contradicting the user's annotation.

Since XeLda does not rely on any automatic processes such as header inference. its performance cannot be impaired by faults in such processes. On the other hand, it incurs a heavy overhead for the user, who must annotate every cell.

2.3.3 SLATE

SLATE [35], like XeLda, requires the user to annotate each cell with a value, dimension, and label, even those cells containing no references to other cells, before the analysis can begin. For example a cell referring to 100 kilograms of oranges is annotated as (100, Kg, apples). After the user annotates the cells, SLATE analyzes the cells with formulae containing references and infers dimensions and labels for these cells from the dimensions and labels of their data cells. SLATE does not show the errors explicitly. But if the unit and label which the process determines for a cell are different from the user annotation, the system replaces them with the new dimension and label. For example, in Figure 2.10, first the user annotates the formula cell C4 as (A4+B4, Miles/Gallon, BMW). But after the system determines the unit and label for C4, the annotation for cell C4 will be (A4+B4, Miles, Gallons, BMW), which is different from the annotation that user has written.



	A	B	C	D	E
1	Miles	Gallons	MPG	Total Gallons	Miles Possible
2	180 (Miles, VW Bug)	5 (Gallons, VW Bug)	=A2/B2(MPG,VW Bug)	12 (Gallons, VW Bug)	=D2*C2(Miles,VW Bug)
3	225 (Miles, Camry)	55 (Gallons, Camry)	=A3/B3 (MPG,Camry)	10 (Gallons, Camry)	=D3*B3 (Gallons^2,Camry)
4	345 (Miles, BMW)	7 (Gallons, BMW)	=A4+B4 (Miles,Gallons,BMW)	15 (Gallons, BMW)	=C4*D4 (Miles*Gallons,Gallons^2,BMW)
5					=MAX(E2:E4) (Miles,Gallons^2)
6					
7					

Figure 2.10 An example for SLATE result (from [8] page 280)

2.3.4 Automatic Detection of Dimension Errors

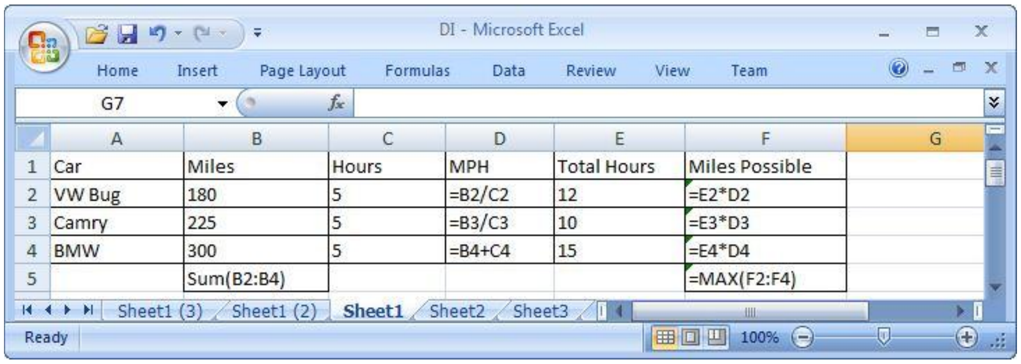
SLATE and XeLda, are examples of algorithms which detect errors using dimension information supplied by the user. In this section, we describe a system that infers the dimension information automatically, rather than requiring the user to annotate it [8]. This procedure is divided to the following steps:

- Header inference
- Label analysis
- Dimension inference

Header inference, which is discussed in 2.3.1, determines headers for each cell. For example in Figure 2.11, the system determines that C1 and B4 are headers for cell C4. In step 2 the algorithm tries to extract dimension from each header according to the following process [8]: a) split headers into separate words. b) Extracting stems of the words c) if the algorithm finds any appropriate stems, which are matched with any of the dimensions, then combine them with each other. For example in Figure 2.11, C1 and B4 are headers for C4, therefore the label analysis determines that "hours" which is converted to "hour" is a dimension for time.

The third step is automatic dimension inference. In this step the system infers the dimensions of each formula cell according to some specific rules [8] and compares them with the dimensions which are extracted from the headers of the formula cells. If these dimensions are not the same, the system reports an error to the user. For example as you can see in Figure 2.11, cell D4 has formula B4+C4; dimension inference deduces from the formula, that the dimensions for D4 are "mile" and "hour" (which are deduced from B4 and C4). But the dimensions which are deduced from header (D1) are mile and hour $^{-1}$ (-1 is derived from the "per" word in the header). Therefore when the algorithm compares these two dimensions, it detects that they are not similar and it reports an error to the user.

In this method, in contrast to XeLda and SLATE, the user does not annotate cells with labels and dimensions, and the system extracts dimensions automatically from the headers of the cells. Therefore it requires less workload for users to enter data in the spreadsheets.


DI - Microsoft Excel
Home Insert Page Layout Formulas Data Review View Team
G7
Sheet1 (3) Sheet1 (2) Sheet1 Sheet2 Sheet3
Ready
100%

	A	B	C	D	E	F	G
1	Car	Miles	Hours	MPH	Total Hours	Miles Possible	
2	VW Bug	180	5	=B2/C2	12	=E2*D2	
3	Camry	225	5	=B3/C3	10	=E3*D3	
4	BMW	300	5	=B4+C4	15	=E4*D4	
5		Sum(B2:B4)				=MAX(F2:F4)	

Figure 2.11 An example for automatic detection (from [8] page 271)

2.4 Templates

The methodologies discussed so far, are related to finding errors in spreadsheets. However, an alternative approach to the problem of spreadsheet errors is to prevent them from occurring in the first place.

A solution to this problem is provided by techniques for analysing existing spreadsheets to extract their structure and automatically create templates. We will discuss two such mechanisms. The first derives templates expressed in the ViTSL template language [36], while the second produces representations of spreadsheet structure in the ClassSheets language, from which a ViTSL representation can also be generated.

2.4.1 Inferring ViTSL Templates

ViTSL (an acronym for visual template specification language) is a visual language for creating spreadsheet templates, which can then be applied within Excel to obtain spreadsheets that conform to the templates using an Excel add-on called Gencel. The architecture of ViTSL/Gencel is illustrated in Figure 2.12.

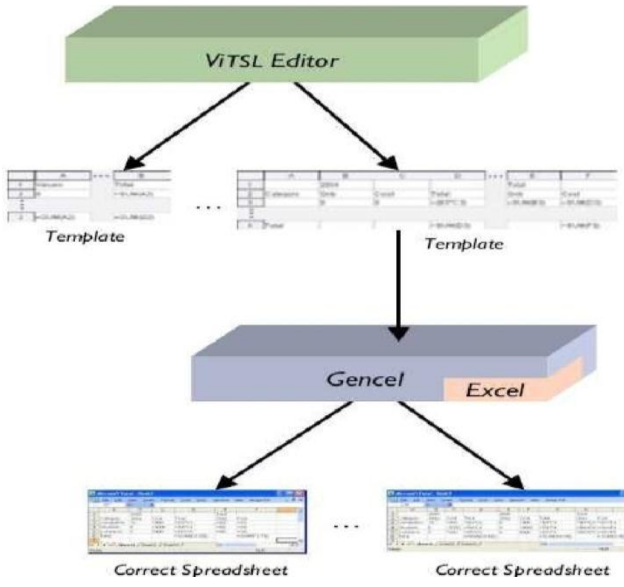
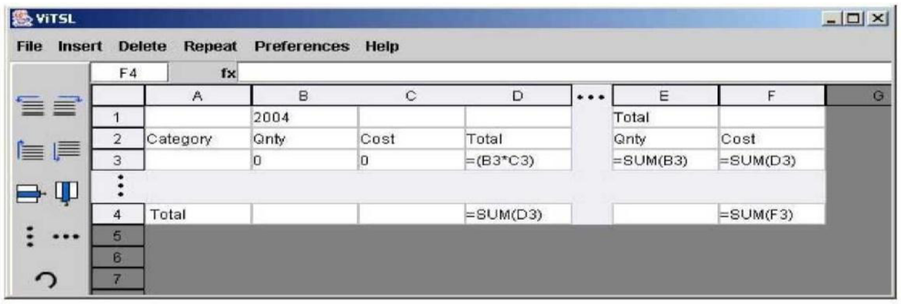


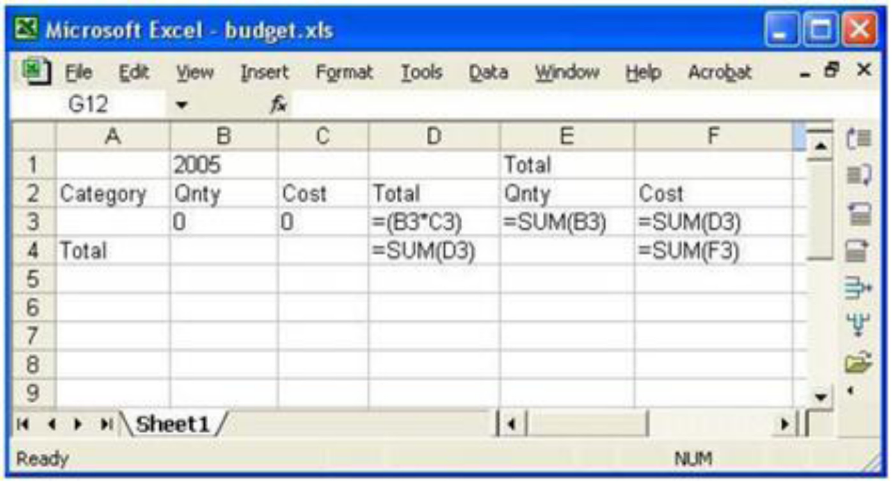
Figure 2.12 ViTSL/Gencel architecture (from [18] page 2)

There are two significant constructs in ViTSL. A vex group, represented by a vertical line of dots, indicates that a group of consecutive rows can be repeated in the vertical direction. Similarly a hex group, represented by horizontally arranged dots, indicates that a group of consecutive columns can be repeated. For example as shown in Figure 3.13, the vex group dots below row 3 indicate that this row can be repeated, and the hex group dots to the right of column D indicate that columns B, C and D can also be repeated. Figure 2.14 depicts a spreadsheet obtained by applying this template in Excel, using the Gencel tool.



	A	B	C	D	⋯	E	F	G
1		2004				Total		
2	Category	Qnty	Cost	Total		Qnty	Cost	
3		0	0	=(B3*C3)		=SUM(B3)	=SUM(D3)	
⋮								
4	Total			=SUM(D3)			=SUM(F3)	
5								
6								
7								

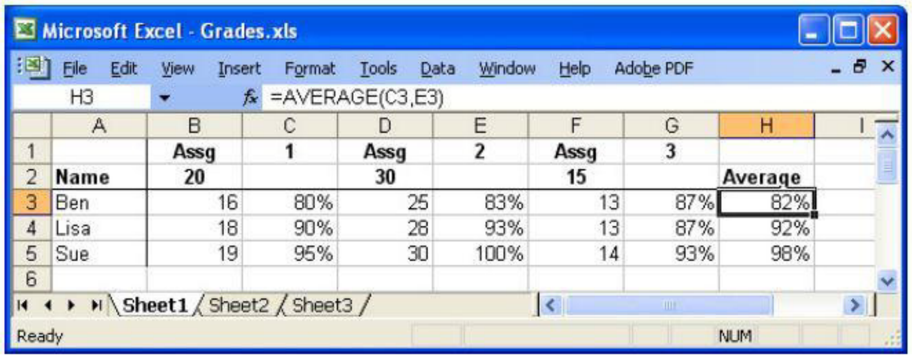
Figure 2.13 A template created in the ViTSL editor (from [18] page 3)


Microsoft Excel - budget.xls
File Edit View Insert Format Tools Data Window Help Acrobat
G12
fx
Sheet1
Ready
NUM

	A	B	C	D	E	F
1		2005			Total	
2	Category	Qnty	Cost	Total	Qnty	Cost
3		0	0	=(B3*C3)	=SUM(B3)	=SUM(D3)
4	Total			=SUM(D3)		=SUM(F3)
5						
6						
7						
8						
9						

Figure 2.14 Gencel spreadsheet (from [18] page 3)

The vex and hex groups in a ViTSL template define rectangular tables, for example, the rectangle from A1 to H5 in Figure 2.15. Hence, the first step in the method for automatically creating ViTSL templates, reported in [36], detects such tables, using some spatial analysis algorithms from UCheck [13].

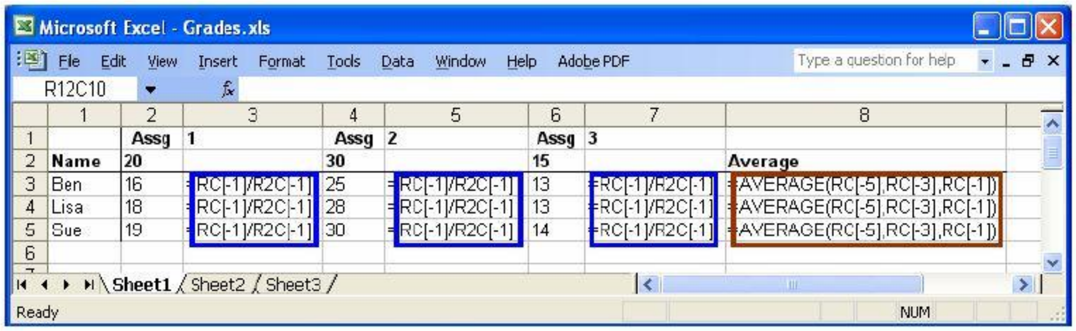

Microsoft Excel - Grades.xls
File Edit View Insert Format Tools Data Window Help Adobe PDF
H3
fx
Sheet1 Sheet2 Sheet3
Ready
NUM

=AVERAGE(C3,E3)

	A	B	C	D	E	F	G	H	I
1		**Assg**	**1**	**Assg**	**2**	**Assg**	**3**		
2	**Name**	**20**		**30**		**15**		**Average**	
3	Ben	16	80%	25	83%	13	87%	82%	
4	Lisa	18	90%	28	93%	13	87%	92%	
5	Sue	19	95%	30	100%	14	93%	98%	
6									

Figure 2.15 Grade sheet example (from [36] page 184)

Once a table has been found, the next step is to find formulae in the table which are "cp-similar", meaning that one formula would result from copying and pasting the other [21]. As Figure 2.16 shows, there are two sets of cp-similar cells in the grade sheet example in Figure 3.4, those outlined in columns 3, 5 and 7, and those outlined in column 8.


Microsoft Excel - Grades.xls
File Edit View Insert Format Tools Data Window Help Adobe PDF
Type a question for help
R12C10
Sheet1 Sheet2 Sheet3
Ready
NUM

	1	2	3	4	5	6	7	8
1		**Assg**	**1**	**Assg**	**2**	**Assg**	**3**	
2	**Name**	**20**		**30**		**15**		**Average**
3	Ben	16	=RC[-1]/R2C[-1]	25	=RC[-1]/R2C[-1]	13	=RC[-1]/R2C[-1]	=AVERAGE(RC[-5],RC[-3],RC[-1])
4	Lisa	18	=RC[-1]/R2C[-1]	28	=RC[-1]/R2C[-1]	13	=RC[-1]/R2C[-1]	=AVERAGE(RC[-5],RC[-3],RC[-1])
5	Sue	19	=RC[-1]/R2C[-1]	30	=RC[-1]/R2C[-1]	14	=RC[-1]/R2C[-1]	=AVERAGE(RC[-5],RC[-3],RC[-1])
6								

Figure 2.16 cp-similar regions in grade sheet (from [36] page 186)

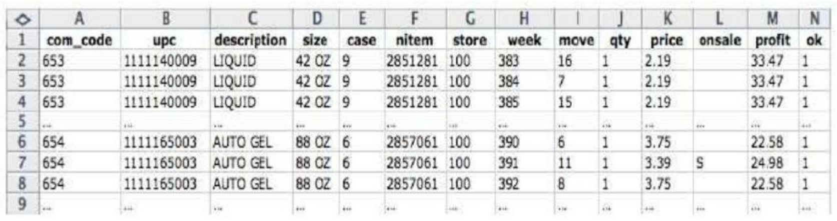
After detecting rectangles of cp-similar cells in a table, the system tries to overlay them, along with the rectangles of data cells they refer to, to find both horizontal and vertical repetitions. For example, in Figure 2.16, the cp-similar cells in column 3 and their referenced data cells in column 2 can be overlaid with the corresponding cells in columns 5 and 4, and the corresponding cells in columns 7 and 6. This repetition of columns provides the basis for creating a hex group. Similarly, the cells from column 2 to column 8 in row 3 can be overlaid with the corresponding cells in rows 4 and 5, leading to creation of a vex group.

The algorithm can ignore some of trivial errors during the overlay process. For example, when two rows could be overlaid but do not have the same type of data cells in specific locations, the system can carry out the overlaying (in the absence of better options) and report the number of violations.

This method for inferring templates from spreadsheets has some shortcomings. For example it cannot extract templates from those spreadsheets in which hex and vex groups cannot be created.

2.4.2 ClassSheets

Like ViTSL, ClassSheets is a spreadsheet-like language with which a spreadsheet developer can build templates from which spreadsheet applications can be generated [38]. Unlike a ViTSL template, which relies on vertical and horizontal repetition of related rows and columns to express structure, a ClassSheets template captures structure using object-oriented classes and database concepts [37]. In this section, we will describe the structure of ClassSheets with the example in Figure 2.17, a spreadsheet of sales records.



◇	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	com_code	upc	description	size	case	nitem	store	week	move	qty	price	onsale	profit	ok
2	653	1111140009	LIQUID	42 OZ	9	2851281	100	383	16	1	2.19		33.47	1
3	653	1111140009	LIQUID	42 OZ	9	2851281	100	384	7	1	2.19		33.47	1
4	653	1111140009	LIQUID	42 OZ	9	2851281	100	385	15	1	2.19		33.47	1
5	...	...	...	...	...	...	...	...	...	...	...	...	...	...
6	654	1111165003	AUTO GEL	88 OZ	6	2857061	100	390	6	1	3.75		22.58	1
7	654	1111165003	AUTO GEL	88 OZ	6	2857061	100	391	11	1	3.39	S	24.98	1
8	654	1111165003	AUTO GEL	88 OZ	6	2857061	100	392	8	1	3.75		22.58	1
9	...	...	...	...	...	...	...	...	...	...	...	...	...	...

Figure 2.17 a spreadsheet for sales records. (From [38] page 220)

The first step in the process of extracting a ClassSheet template is detecting functional dependencies between attributes or sets of attributes in a spreadsheet, where an attribute is a column label. For example in Figure 3.6, “upc”, “size”, “week” are the attributes. Sets, which are defined by users, consist of one or many attributes. For example Dish and StoreWeek can be considered as two sets, which Dish has “upc”, “com_code”, “description”, “size” and “case” as attributes and StoreWeek consists of “store” and “week”.

Functional dependencies are determined using standard rules, described in [38].

After determining the functional dependencies, the algorithm categorizes attributes and according to standard relational database concepts, selects primary keys and foreign keys for each of sets. For example, Figure 2.18 illustrates sets and attributes of the spreadsheet of sales records. In Figure 2.18, attributes which specify with underline are primary keys for the sets.

StoreWeek(*<u>Store,Week</u>*)

Dish(*<u>upc</u>,com_code, description, size,case,nitem*)

Sale(*<u>upc,store,week</u>,move,profit,price,onsale,qty,ok*)

Figure 2.18 Sets and attributes of the sale system (from [38] page 5)

In the next step, a directed graph is constructed according to the relations among the sets discussed above. Figure 2.19, depicts the directed graph for the sales records system.

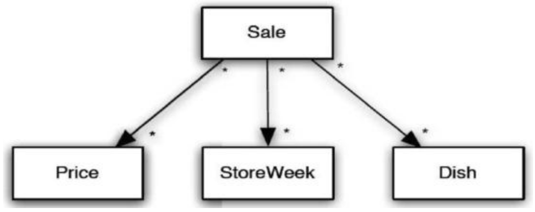
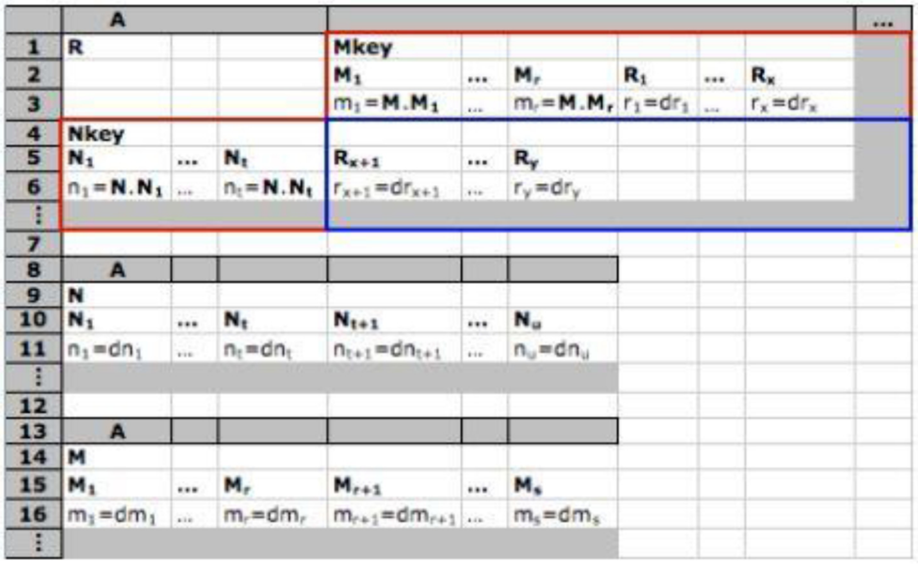


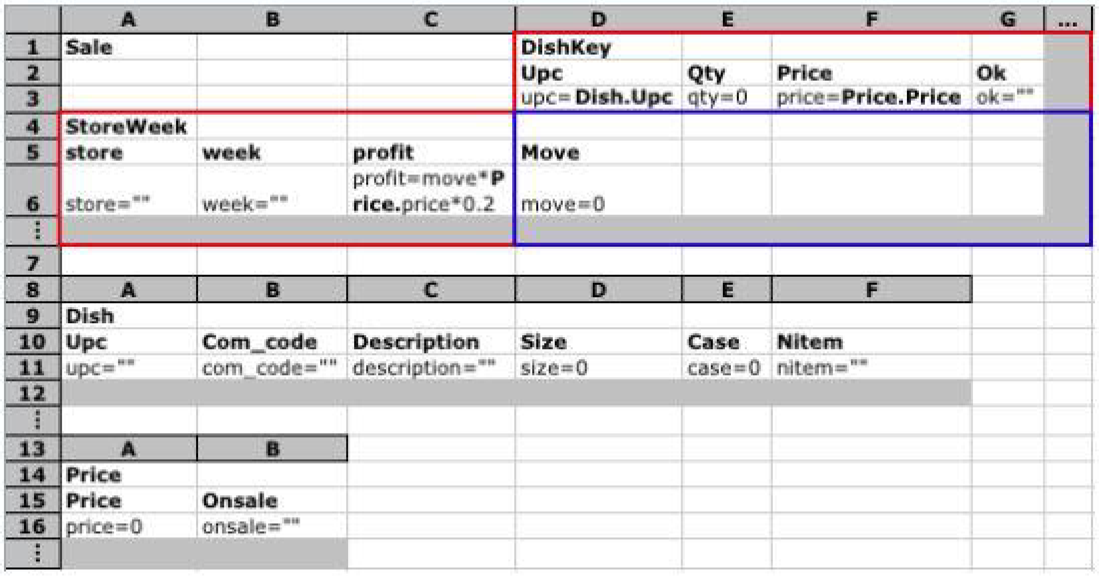
Figure 2.19 Directed graph for the sale system. (From [38] page5)

In the next step the system translates directed graph nodes into a ClassSheets structure as shown in Figure 2.20. In this Figure, R is the root of the directed graph, M and N represent sets of nodes of the directed graph, M_1,..,M_r represent primary keys of M, N_1,…,N_r represent primary keys of N, M_{r+1},..,M_u and N_{r+1},..,N_u represent the rest of attributes in the sets M and N respectively. R_1,…,R_y represent the attributes of set R and dn_1,..dn_u, dm_1,..dm_u and dr_1...dr_u are values of the corresponding attributes of sets N,M and R respectively. Therefore, as Figure 2.21 shows, for the sales system, R, M and N are equal to Sale, Dish and StoreWeek respectively. Figure 2.21 is the ClassSheets structure for the sales system when the system puts primary keys and attributes of M, N and R in the appropriate cells.



	A									...
1	R			Mkey						
2				M_1	...	M_r	R_1	...	R_x	
3				m_1=M.M_1	...	m_r=M.M_r	$r_1=dr_1$	...	$r_x=dr_x$	
4	Nkey									
5	N_1	...	N_t	R_{x+1}	...	R_y				
6	n_1=N.N_1	...	n_t=N.N_t	$r_{x+1}=dr_{x+1}$	...	$r_y=dr_y$				
⋮										
7										
8	A									
9	N									
10	N_1	...	N_t	N_{t+1}	...	N_u				
11	$n_1=dn_1$	...	$n_t=dn_t$	$n_{t+1}=dn_{t+1}$	...	$n_u=dn_u$				
⋮										
12										
13	A									
14	M									
15	M_1	...	M_r	M_{r+1}	...	M_s				
16	$m_1=dm_1$	...	$m_r=dm_r$	$m_{r+1}=dm_{r+1}$	...	$m_s=dm_s$				
⋮										

Figure 2.20 General structures of ClassSheets (from [38] page 6)



	A	B	C	D	E	F	G	...
1	Sale			DishKey				
2				Upc	Qty	Price	Ok	
3				upc=Dish.Upc	qty=0	price=Price.Price	ok=""	
4	StoreWeek							
5	store	week	profit	Move				
6	store=""	week=""	profit=move*Price.price*0.2	move=0				
⋮								
7								
8	A	B	C	D	E	F		
9	Dish							
10	Upc	Com_code	Description	Size	Case	Nitem		
11	upc=""	com_code=""	description=""	size=0	case=0	nitem=""		
12								
⋮								
13	A	B						
14	Price							
15	Price	Onsale						
16	price=0	onsale=""						
⋮								

Figure 2.21 A ClassSheets structure for the sale system (from [38] page 6)

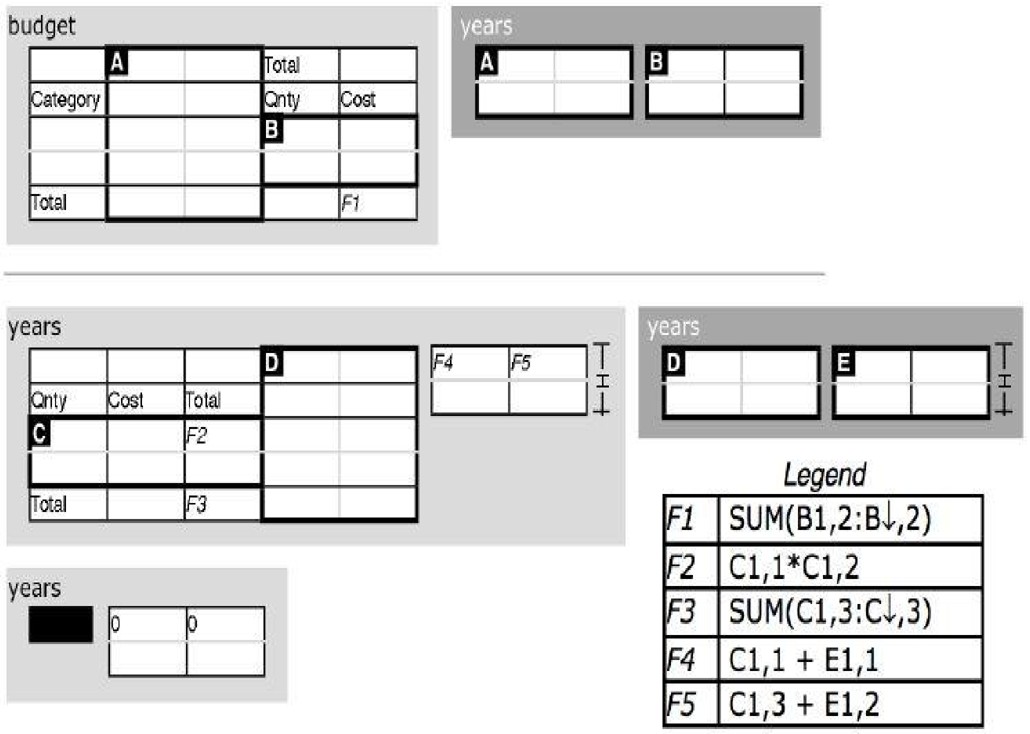
Chapter 3 Inferring Templates from Spreadsheets

As discussed in chapter 2, extracting templates from spreadsheets is a difficult process because spreadsheet design does not obey any specific rules. There are, however, some methods for extracting templates from spreadsheets that we described in chapter 2 [36,38]. In this chapter we will introduce our technique for extracting templates from spreadsheets and converting them into programs in *L-sheets*, which we will describe in the next section. It is important to note that, our goal is to define the patterns that can be used to build templates, not to provide algorithms to extract these patterns, but it should be clear that such algorithms can be constructed.

Our method for extracting templates from spreadsheets focuses on formula cells and the data cells they refer to, ignoring other features such as labels. We will discuss this focus further in Chapter 5.

3.1 L-sheets

L-sheets is a spreadsheet extension that enhances the programmability of spreadsheets with logic programming. An L-sheets application consists of *worksheets* like those in Microsoft Excel, and *program sheets*. A program sheet consists of a set of *templates*, each of which is a sequence of *cases*. For example, Figure 3.1 depicts two templates, **budget**, consisting of one case, and **years** consisting of two cases. A case is made up of a *head* and a *body*, which are respectively, a *form* and a sequence of forms. A form has a name, and contains several *parameters* which are arrays. For example, in the Figure, the body of the single case of the budget template has just one form, which is named years and has two parameters. Head forms have a pale grey background while body forms are lighter grey arrays in forms are depicted as rectangles divided by horizontal and vertical lines. A vertical grey line indicates that an array represents a rectangle of any width in a worksheet, and a horizontal grey line indicates that an array represents a rectangle of any height in a worksheet.



F1	SUM(B1,2:B↓,2)
F2	C1,1*C1,2
F3	SUM(C1,3:C↓,3)
F4	C1,1 + E1,1
F5	C1,3 + E1,2

Figure 3.1 Program sheet for Budget example (from [39] p.93)

Arrays in forms may be divided into *subarrays*, which may be named. For example, the array in the head form of the single case of the **budget** definition includes embedded subarrays named **A** and **B**. A subarray may contain a constant or a formula. In our example, the bottom left and bottom right subarrays of the parameter array of the head of the single case of the **budget** template contain, respectively, the constant Total, and the formula SUM(B1,2:B↓,2).

Note that in this description of L-sheets, we have used the words "template" and "form" rather than "definition" and "template" as in [39,40], so that, as in [36] we can use "template" to mean a specification of spreadsheet structure.

The annotations F1, F2… are not part of the program, but have been added to indicate cells which contain formulae. In a program sheet formula a reference consists of a subarray name followed by a row and column of that subarray. For example, the formula SUM(B1,2:B↓ ,2) refers to the rectangle consisting of all cells from the top to the bottom of column 2 of subarray **B**.

L-sheets programs can be represented textually, which will be convenient later in this chapter, using a Prolog-like notation in which each case is represented as a clause. Since we ignore labels when extracting the structure from a spreadsheet, we omit them also from this textual representation. To illustrate, the representation of the single case of the **budget** template in Figure 3.1 is:

budget(X) :- **years**(Y,Z).

where X, Y and Z represent the parameters of the two forms. A parameter is represented by a pair of lists; for example X is the following pair

(

((∅,1), ({A},*), ({B},1), ({=SUM(B1,2:B↓,2),B},1)),

(({A},1),({},1), ({B},*), ({=SUM(B1,2:B↓,2)},1)}

)

The first list represents the division of the array into vertical strips. In this example, the first vertical strip, represented by the pair (∅,1), spans nothing significant and is one column wide; the second strip, represented by ({A},*), spans subarray A, and is any width; the third spans subarray B and is 1 column wide; and the last strip spans a formula and the subarray B and is one column wide. In the same way, the second list represents the division into horizontal strips.

3.2 Representing a Spreadsheet

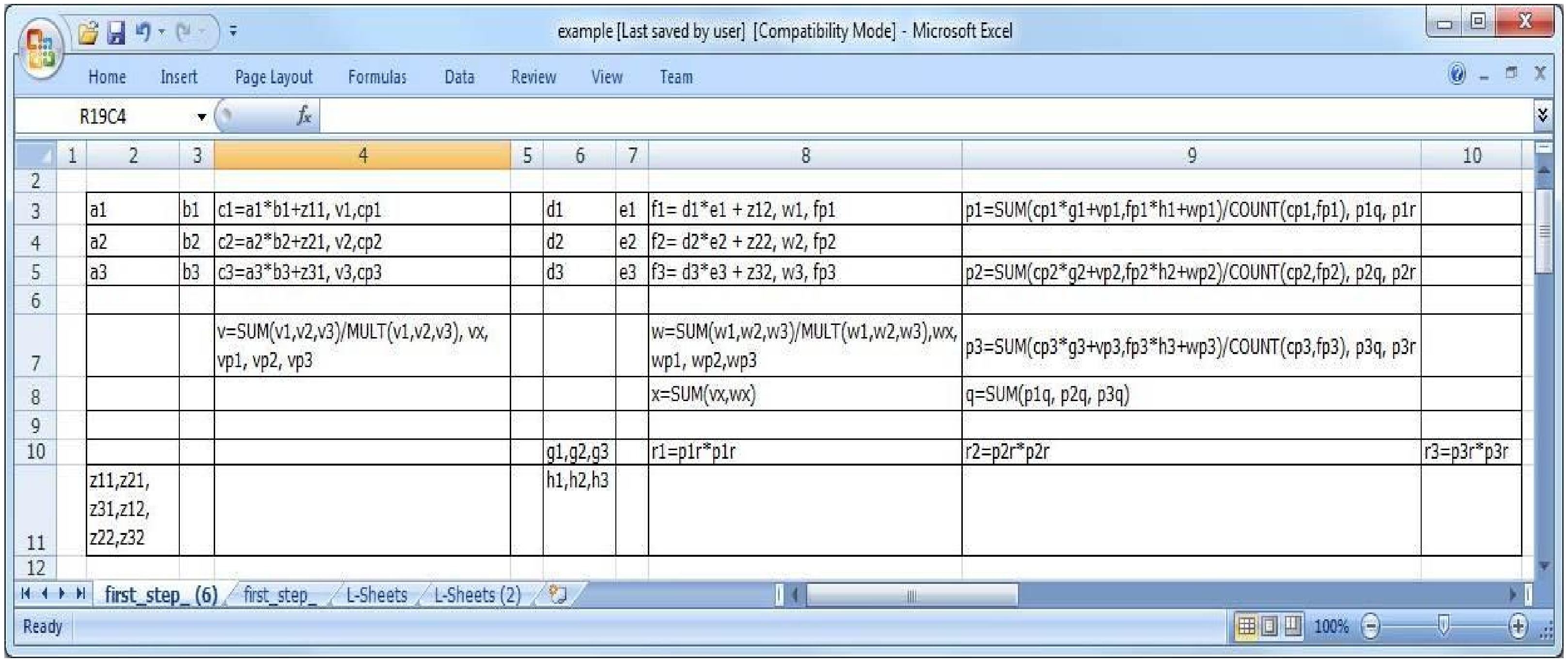
As mentioned above, our method for extracting spreadsheet structure focuses on formula cells and the cells they refer to. To expedite our analysis, we build a representation of a spreadsheet as a set of *characteristics*, where a characteristic corresponds either to a formula cell, or a cell referred to by a formula cell. Note that in this representation, there may be several characteristics corresponding to one cell. For example, if a cell contains a formula and is also referred to by two other formula cells, there will be three characteristics corresponding to it.

If x is a cell containing a formula f, let $f_1,\ldots,f_n$ denote the cells referenced by f in the order of their first occurrences in f, and let f' denote the expression obtained by replacing each occurrence of f_i in f by $<i>$ for each i ($1 \le i \le n$).

Definition 1: If **S** is a spreadsheet, let x be a cell of S that either contains a formula f, or is referenced by a formula f in a cell y, then a *characteristic of* **S** *corresponding to* x is a 6-tuple of the form (*t, l, b, r, ex, rel*) where

- $t = b$ = the index of the row of **S** in which x occurs,
- $l = r$ = the index of the column of **S** in which x occurs,
- *ex* is the expression $\approx(<0>, f')$
- *rel* is the sequence ($c_0, c_1, \ldots, c_n$) where c_i is the characteristic of **S** corresponding to f_i for each i ($1 \le i \le n$), and c_0 is the characteristic of **S** corresponding to the cell containing f.

We will use names of the components of a characteristic as functions: for example, if c is a characteristic, *rel*(c) will denote the sixth component of c. Also, we will denote by



example [Last saved by user] [Compatibility Mode] - Microsoft Excel

R19C4

	1	2	3	4	5	6	7	8	9	10
2										
3		a1	b1	c1=a1*b1+z11, v1,cp1		d1	e1	f1= d1*e1 + z12, w1, fp1	p1=SUM(cp1*g1+vp1,fp1*h1+wp1)/COUNT(cp1,fp1), p1q, p1r	
4		a2	b2	c2=a2*b2+z21, v2,cp2		d2	e2	f2= d2*e2 + z22, w2, fp2		
5		a3	b3	c3=a3*b3+z31, v3,cp3		d3	e3	f3= d3*e3 + z32, w3, fp3	p2=SUM(cp2*g2+vp2,fp2*h2+wp2)/COUNT(cp2,fp2), p2q, p2r	
6										
7				v=SUM(v1,v2,v3)/MULT(v1,v2,v3), vx, vp1, vp2, vp3				w=SUM(w1,w2,w3)/MULT(w1,w2,w3),wx, wp1, wp2,wp3	p3=SUM(cp3*g3+vp3,fp3*h3+wp3)/COUNT(cp3,fp3), p3q, p3r	
8								x=SUM(vx,wx)	q=SUM(p1q, p2q, p3q)	
9										
10						g1,g2,g3		r1=p1r*p1r	r2=p2r*p2r	r3=p3r*p3r
11		z11,z21, z31,z12, z22,z32				h1,h2,h3				
12										

Figure 3.2 An example of a spreadsheet

rect(c) the tuple (*t*(c), l(c), *b*(c), *r*(c)). We will apply these conventions to other entities later on.

Consider Figure 3.2, characteristics are shown for each formula cell and its data cells. In this Figure, cells contain names that we will use to identify characteristics in our discussion, as well as formulae, expressed in terms of these names. For example cell R3C4 contains a formula which refers to R3C2, R3C3 and R11C2. The role of R3C4 as a formula cell is represented by the characteristic c1, and the roles of R3C2, R3C3 and R11C2 as data cells referred to by R3C4 are represented by the characteristics a1, b1 and z11. The characteristics a1, b1, c1 and z11 are as follows:

a1=(3,2,3,2, ≈(<0>, (<1>*<2>+<3>)),(c1,a1,b1,z11))

b1=(3,3,3,3, ≈(<0>,<1>*<2>+<3>)),(c1,a1,b1,z11))

c1=(3,4,3,4, ≈(<0>,<1>*<2>+<3>)),(c1,a1,b1,z11))

z11=(11,2,11,2, ≈(<0>,<1>*<2>+<3>)),(c1,a1,b1,z11))

Note that if a cell is referenced by several different formula cells, there will be a distinct characteristic corresponding to it for each reference. In addition, if the cell contains a formula, there will a characteristic corresponding to it, representing its role as a formula cell. For example, in addition to c1, discussed above, R3C4 has two characteristics, v1 and cp1, corresponding to references from R7C4 and R3C9 respectively.

3.3 Finding Patterns in a Spreadsheet

From now on, we will use the word 'spreadsheet' to mean both the familiar rectangular grid of cells, and the set of all characteristics derived from a spreadsheet.

The patterns of interest in a spreadsheet involve repetitions of groups of cells linked by formulae, and regularly spaced either horizontally or vertically. Such repetitions are frequently accompanied by cells containing formulae which compute summaries of the values of repeated cells. In general, a summary formula can contain aggregator functions, such as +, which can have any number of arguments. We start our analysis with some technical definitions that we can use to organise the list of arguments of a function into subsequences, then use these definitions to help characterise summaries.

Definition 2: If t, k and p are positive integers such that $tk \leq p$, let K be a sequence of length k+1 of sequences of integers between 1 and p, such that the elements of K are mutually disjoint, $|K_{k+1}| = p - tk$, $|K_i| = t$ for each i ($1 \leq i \leq k$), and for each i ($1 \leq i \leq k+1$) the elements of K_i are distinct. K is called *a ktp-division*. Each element of K_{k+1} is called a *remainder* of K.

If X is any sequence of length p and K is a *ktp*-division, let Y be the sequence of sequences of elements of X such that $Y_{i,j} = X_{K_{i,j}}$ for all i ($1 \leq i \leq k$) and j ($1 \leq j \leq t$), and $Y_{k+1,j} = X_{K_{k+1,j}}$ for all j ($1 \leq j \leq p - tk$). Y is called the *kt-division of X induced by K*, and each element of Y_{k+1} is called a *remainder of Y.*

For example, ((12,5,8,2),(1,11,6,9),(15,13,10,4),(3,14,7)) is a 3-4-15 division and induces the 3-4-division ((L, E, H, B), (A, K, F, I,), (O, M, J, D), (C, N, G)) of (A, B, C, D, E, F, G, H, I, J, K, L, M, N, O).

Definition 3: A binary function f is called an *aggregator* iff for some a, called the *identity* of f, $f(a,x) = x$ for all x. If f is an aggregator with identity a, for each $n \geq 0$, we define an n-ary function f^n as follows:

$$f^0 = a$$

$$f^n(x_1,\ldots,x_n) = f(f^{n-1}(x_1,\ldots,x_{n-1}),x_n)$$

If f is a p-ary function and K is a ktp-division, then f is a *repetition with respect to K* iff there exists a function g, and for some $q \geq 1$ there exist functions $e_1,\ldots, e_q$, and aggregators $h_1,\ldots, h_q$ such that for all X

$$f(X) = g(h_1{}^k(e_1(Y_1, Z_1), \ldots, e_1(Y_k, Z_1)),$$

$$\ldots,$$

$$h_q{}^k(e_q(Y_1, Z_q), \ldots, e_q(Y_k, Z_q)), Z_0)$$

where Y is the kt-division of X induced by K, for each i, Z_i is a sequence of distinct elements of Y_{k+1}, and $Z_0,\ldots, Zq$ together include all elements of Y_{k+1}.

Let c be a characteristic such that ex(c) = ≈(<0>,f) where f has arity p and is a repetition with respect to some ktp-division K. Let Y be the kt-division of (rel(c)$_1$,..., rel(c)$_p$) induced by K. Then c is called a *summary* of the sequence of characteristics ($Y_{1,i}, \ldots, Y_{k,i}$) for each i ($1 \leq i \leq$t) , and each element of Y_{k+1} is called a *peripheral* of c.

For example in Figure 3.2, R3C9 is a summary of each of the sequences of characteristics (cp1,fp1), (g1,h1) and (vp1,wp1). For this cell, SUM and COUNT are aggregators and division (/) is the g function. Multiplication in (cp1 * g1) and (fp1 * h1) are e_1 and e_2 functions and there are no arguments to the function other than the characteristics in the summarised sequences.

The patterns we are looking for include repeated groups of functionally related cells. The next definition requires the cells within one group in a repetition to have the same relationships as the corresponding cells within another group.

Definition 4: Two disjoint sequences ($c_1,\ldots,c_n$) and ($d_1,\ldots,d_n$) of distinct characteristics are *i-compatible*, where $1 \leq i \leq n$, iff

- $ex(c_i) = ex(d_i)$
- for some k ($1 \leq k \leq n$), $rel(c_i)_0 = c_k$ and $rel(d_i)_0 = d_k$
- $\forall j$ ($1 \leq j \leq |rel(c_i)|$) either $rect(rel(c_i)_j) = rect(rel(d_i)_j)$ or for some k ($1 \leq k \leq n$), $rel(c_i)_j = c_k$ and $rel(d_i)_j = d_k$

For example in Figure 3.2, the sequences (a1,b1,c1,v1,cp1,d1,e1,f1,w1,fp1) and (a2,b2,c2,v2,cp2,d2,e2,f2,w2,fp2) are 1-compatible because :

- ex(a1) = ex(a2)
- $rel(a1)_0$ = c1 and $rel(a2)_0$ = c2, where c1 and c2 are the third elements of the sequences

These sequences are also i-compatible for every value of i from 2 to 10. But the sequences (a1,b1,c1,v1,cp1,d1,e1,f1,w1,fp1,p1) and (a1,b1,c1,v1,cp1,d1,e1,f1,w1,fp1,p2) are not 11-compatible because the third condition of the compatibility definition is not satisfied.

Compatibility deals with the functional similarity between repeating groups of characteristics, an important property of the patterns we are interested in. These patterns, however, also require repeated cells to satisfy certain geometrical requirements, in particular, vertical or horizontal alignment, and even spacing, which are dealt with in the following definitions.

Definition 5: A nonempty sequence of distinct characteristics ($c_1, c_2, \ldots, c_n$) is called a *horizontal match* iff $t(c_i) = t(c_j)$ and $ex(c_i) = ex(c_j)$ for all i, j ($1 \leq i \leq j \leq n$); and there exists $d\geq 1$ such that for all i ($1\leq i<n$), $l(c_{i+1})-r(c_i) = d$. The constant d is called the *gap* of the match.

We define *vertical match* similarly. A *match* is either a horizontal match or a vertical match.

A nonempty set C of matches is said to be *acceptable* if all the matches in C are the same length, and no characteristic occurs more than once in C.

In Figure 3.2, each of M1 = (a1, b1, d1) and M2 = (r1, r2, r3) is a horizontal match and each of M3 = (p1, p2, p3), M4 = (a1, a2, a3) and M5 = (b2, b3) is a vertical match. C = {M1, M2, M3} is an acceptable set. M1 and M4 cannot be in the same acceptable set since they overlap, and M5 cannot be an acceptable set with any of the others since it is shorter.

Now we define patterns consisting of sequences of repeated items.

Definition 6: If $C = \{C_1, \ldots, C_m\}$ is an acceptable set of matches, then for each i ($1\leq i\leq|C_1|$), the sequence of characteristics $(C_{1,i}, \ldots, C_{m,i})$ is called a *slice* of C.

A *linear repeat* is a set R of acceptable matches such that for each i ($1\leq i\leq|R|$) and for any two slices d and e of R, either d and e are i-compatible or there is a characteristic which is a summary of R_i.

The length of a linear repeat R is the length of its matches.

In Figure 3.2, let Q be the set of matches {(a1,a2,a3), (b1,b2,b3), (c1,c2,c3), (v1,v2,v3), (cp1,cp2,cp3), (d1,d2,d3), (e1,e2,e3), (f1,f2,f3), (w1,w2,w3), (fp1,fp2,fp3), (p1,p2,p3), (p1q,p2q,p3q), (p1r,p2r,p3r), (r1,r2,r3)}, then Q is acceptable since the matches in Q are disjoint and the same length, any two slices of Q are i-compatible for i = 1, 2, 3, 5, 6, 7, 8, 10, 11, 12, 13 and 14, and there is a summary for Q_4 and a summary for

Q_9. Note that any subset of Q is a linear repeat, as is the set of matches we get by removing the first (or second or third) characteristic from each match in Q. Note also that a linear repeat may include both horizontal and vertical matches; for example, all matches in Q are horizontal except the last.

The rectangle of cells from a1 to g3 contains a horizontal repeating pattern and a vertical repeating pattern, so the next three definitions complete the definition of repeats to account for such rectangles of two-dimensional patterns.

Definition 7: If H is a set of horizontal matches and V is a set of vertical matches, or vice versa, such that the set of characteristics in H is equal to the set of characteristics in V, then a match of H is said to be *major with respect to* V iff it contains a characteristic that occurs in the first slice of V.

For example in Figure 3.2, H = {(a1,b1,c1,d1,e1,f1), (a2,b2,c2,d2,e2,f2), (a3,b3,c3,d3,e3,f3)} is a set of horizontal matches and V = {(a1,a2,a3), (b1,b2,b3), (c1,c2,c3), (d1,d2,d3), (e1,e2,e3), (f1,f2,f3)} is a set of vertical matches then the first element of H is the major with respect to V because characteristics in the first element of H are in the first slice of the V.

Definition 8: A set of matches is *consistent* if they are either all vertical or all horizontal and have the same gap.

For example in Figure 3.2, H = {(a1,d1),(a2,d2),(a3,d3)} is consistent because all of its elements are horizontal matches and the gap is 3 cells but H = {(a1,a2), (b1,b3), (c1,c3)} is not consistent because its elements have different gaps.

Definition 9: A *candidate* is a pair (P,Q) of linear repeats such that at least one of P and Q is of length $\geq \theta$, and there exist partitions $\{\mathcal{P}_1, \ldots, \mathcal{P}_n, \mathcal{P}_{n+1}, \ldots, \mathcal{P}_t\}$ and $\{\mathcal{Q}_1, \ldots,$

$\mathbb{Q}_n$, $\mathbb{Q}_{n+1}$, …, $\mathbb{Q}_s$}, of P and Q, which are the coarsest partitions of P and Q into consistent sets such that

- for each i ($1 \le i \le n$), the set of characteristics in $\mathbb{P}_i$ is equal to the set of characteristics in $\mathbb{Q}_i$, and either the matches in $\mathbb{P}_i$ are vertical and those in $\mathbb{Q}_i$ are horizontal, or vice versa. ($\mathbb{P}_i$,$\mathbb{Q}_i$) is called a *1,2-block* of (P,Q).
- if $n<i \le t$, then $\mathbb{P}_i$ consists of a single match and either
    - $\mathbb{P}_{i,1,1}$ is a summary of some matches, each of which is either major with respect to $\mathbb{Q}_j$ for some j ($1 \le j \le n$), or contained in $\mathbb{Q}_j$ for some j ($n<j \le s$)
    - $rel(\mathbb{P}_{i,1,1})_0 \neq \mathbb{P}_{i,1,1}$; or
    - each x in $rel(\mathbb{P}_{i,1,1})$ either does not occur in (P,Q), or occurs in the first slice of $\mathbb{Q}_j$ for some j ($1<j \le n$), or in $\mathbb{P}_j$ for some j ($n<j \le t$).
- if $n<i \le s$, then $\mathbb{Q}_i$ consists of a single match and either
    - $\mathbb{Q}_{i,1,1}$ is a summary of some matches, each of which is either major with respect to $\mathbb{P}_j$ for some j ($1 \le j \le n$) or contained in $\mathbb{P}_j$ for some j ($n<j \le t$)
    - $rel(\mathbb{Q}_{i,1,1})_0 \neq \mathbb{Q}_{i,1,1}$; or
    - each x in $rel(\mathbb{Q}_{i,1,1})$ either does not occur in (P,Q), or occurs in the first slice of $\mathbb{P}_j$ for some j ($1<j \le n$), or in $\mathbb{Q}_j$ for some j ($n<j \le s$).

The constant $\theta \ge 2$ is called the *repeat threshold*. For each i, where $n<i \le t$, $\mathbb{P}_i$ is called a *1-block* of (P,Q). For each i, where $n<i \le s$, $\mathbb{Q}_i$ is called a *2-block* of (P,Q).

A candidate (P,Q) is a *repeat* iff

- there is no candidate (R,Q) such that either $P \subset R$, or $|P|=|R|$ and P_i is a subsequence of R_i for each i ($1 \le i \le |P|$); and

- there is no candidate (P,R) such that either $Q \subset R$, or $|Q|=|R|$ and Q_i is a subsequence of R_i for each i ($1 \leq i \leq |Q|$).

In Figure 3.2, let P and Q be the following sets of matches

P = {(a1,d1), (b1,e1), (c1,f1), (v1,w1), (cp1,fp1), (a2,d2), (b2,e2), (c2,f2), (v2,w2), (cp2,fp2), (a3,d3), (b3,e3), (c3,f3), (v3,w3), (cp3,fp3), (v,w), (vx,wx), (vp1,wp1), (vp2,wp2), (vp3,wp3), (g1,h1), (g2,h2), (g3, h3)}.

Q = {(a1,a2,a3), (b1,b2,b3), (c1,c2,c3), (v1,v2,v3), (cp1,cp2,cp3), (d1,d2,d3), (e1,e2,e3), (f1,f2,f3), (w1,w2,w3), (fp1,fp2,fp3), (p1,p2,p3), (p1q,p2q,p3q), (p1r,p2r,p3r), (r1,r2,r3)}.

As discussed above, Q is a linear repeat. We leave it to the reader to establish that P is also a linear repeat.

Now let $\mathcal{P}$ and $\mathcal{Q}$ be partitions of P and Q, as follows:

$\mathcal{P} = \{\{P_1, \ldots, P_{20}\}, \{P_{21}\}, \ldots, \{P_{23}\}\}$

$\mathcal{Q} = \{\{Q_1, \ldots, Q_{10}\}, \{Q_{11}\}, \ldots, \{Q_{14}\}\}$

The sets of characteristics in $\mathcal{P}_1$ and $\mathcal{Q}_1$ are the same, the matches in $\mathcal{P}_1$ are horizontal, the matches in $\mathcal{Q}_1$ are vertical, each of the other sets in $\mathcal{P}$ and $\mathcal{Q}$ consists of a single match, and all the sets in $\mathcal{P}$ and $\mathcal{Q}$ are consistent.

To verify that (P,Q) is a candidate, we need to check that each of $\mathcal{P}_2$, $\mathcal{P}_3$, $\mathcal{P}_4$, $\mathcal{Q}_2$, $\mathcal{Q}_3$, $\mathcal{Q}_4$ and $\mathcal{Q}_5$ satisfy the conditions. For example, $rel(\mathcal{P}_{2,1,1})_0 = rel(P_{21,1})_0 = rel(\text{g1}) \neq \text{g1}$; hence $\mathcal{P}_2$ satisfies the conditions. Now $\mathcal{Q}_{2,1,1} = Q_{11,1} = \text{p1}$, which is a summary of P_{21}, P_5 and P_{18}. The first of these is contained in one of the singleton sets in $\mathcal{P}$, while each of the other two is major wrt $\mathcal{Q}_1$. Hence $\mathcal{Q}_2$ satisfies the conditions. Lastly, considering $\mathcal{Q}_5$, we see that *rel*($\mathbb{Q}_{5,1,1}$) = *rel*($Q_{14,1}$) = *rel*(r1) =(p1r), so since p1r occurs in the singleton set $\mathbb{Q}_{13}$ of $\mathbb{Q}$, $\mathbb{Q}_5$ satisfies the conditions. We leave it to the reader to verify that remaining four singleton sets conform to the definition.

It is easy to see that there are no coarser partitions of P and Q that satisfy these conditions, since the set of characteristics occurring in {P_{21}}, …, {P_{23}} is disjoint from the set of characteristics occurring in {Q_{11}}, …, {Q_{14}}. It is also clear that the candidate (P,Q) is a repeat, since P and Q contain all characteristics representing repeated items.

3.4 Representing the Structure of a Repeat

Once a repeat has been identified, we want to build a structure that represents a generalisation of it. For this purpose we define groups of characteristics according to the roles they play in the repeat. These groups contain references to other groups, corresponding to functional relationships, and groups that have a geometrical relationship are combined. The groups and the relationships between them can be represented as a tree, like that in Figure 3.3, which corresponds to the repeat in Figure 3.2, discussed in the last example. We suggest that the reader identify in it examples of each of the definitions that follow. We will describe the conventions used in drawing the tree during the presentation of the definitions.

Definition 10: A *group* of a repeat is either an *rr*, *r1*s, *r2s*, *r*1, *r2, s1, s2, a, b, r1a, r2a,* or *e-* group.

If c is a characteristic such that c does not occur in repeat (P,Q), and either *rel*(c)$_0$ occurs in (P,Q), or *rel*(c)$_0$ is a summary of some matches in the 1-blocks of (P,Q), or *rel*(c)$_0$ is a summary of some matches in the 2-blocks of (P,Q), then the *e-group*

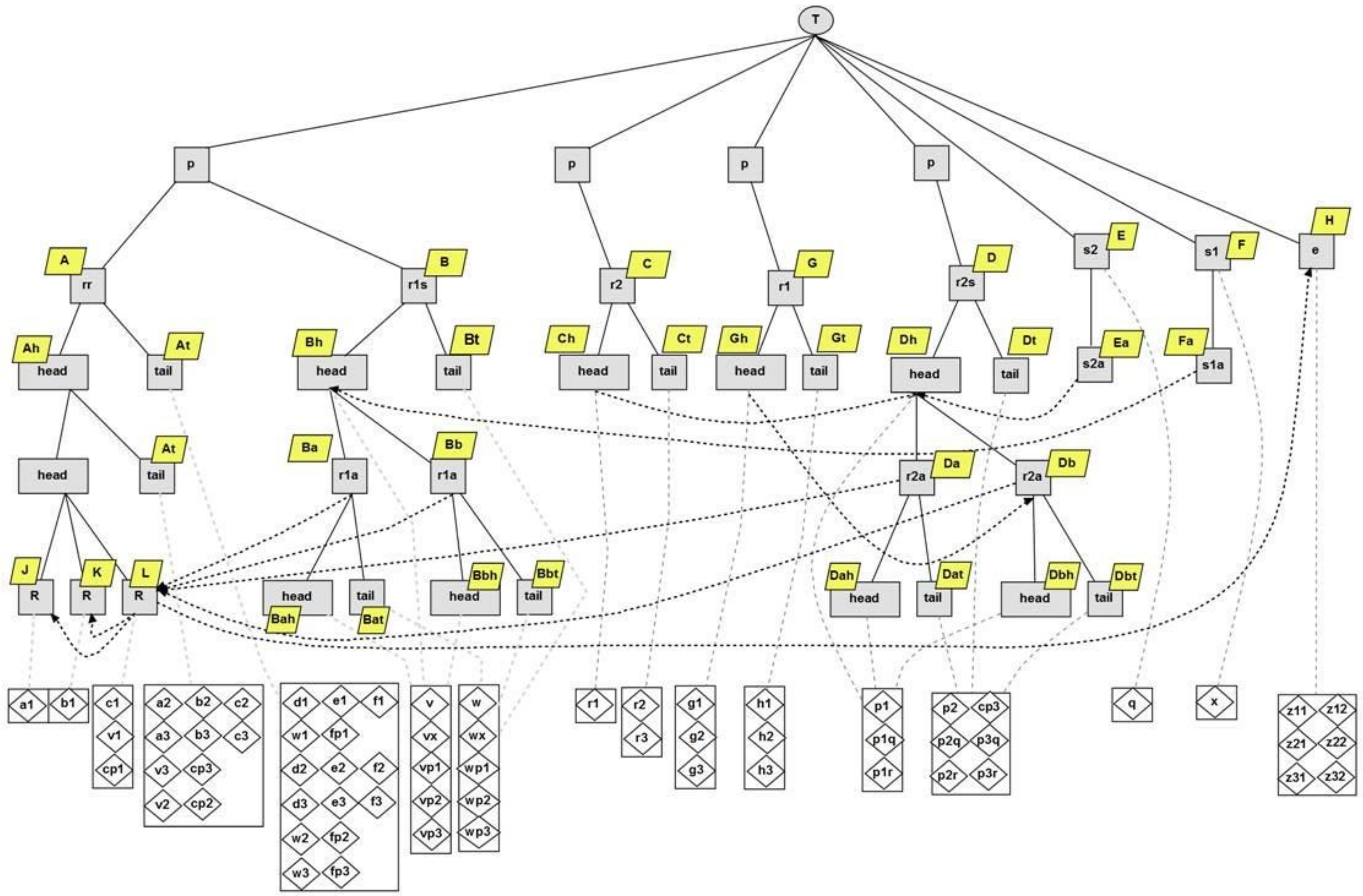


Figure 3.3 Tree structure for figure 3.2

of (P,Q) *corresponding to* c is the set of characteristics {d | *rect*(d)=*rect*(c) and d satisfies the same conditions as c with respect to (P,Q)}.

Figure 3.3, has an e-group corresponding to {z11,z12,z21,z22,z31,z32}.

In Figure 3.3, each group is drawn as a grey rectangle annotated with the group type. The dotted lines below groups indicate the sets of characteristics that they encompass, represented by the boxes, while the characteristics that comprise these sets are represented by the diamonds inside the boxes.

Definition 11: Let c be a characteristic that does not occur in repeat (P,Q), and is a summary of some matches in the 1-blocks of P, and *ex*(c) = $\approx(<0>, g(h_1^k(e_1(Y_1, Z_1), \ldots, e_1(Y_k, Z_1)), \ldots, h_q^k(e_q(Y_1, Z_q), \ldots, e_q(Y_k, Z_q)), Z_0)$; then for $1 \le i \le q$, the ith *s1a-group of* (P,Q) *corresponding to* c is the triple A = (*ex*, *id*, *cont*), where

- *cont* = {c}
- *id* is the identity of the aggregator function h_i
- $ex = h_i(n(\text{A}), e_i(n(\text{y}_1), \ldots, n(\text{y}_{|Y_l|}), n(\text{z}_1), \ldots, n(\text{z}_{|z_i|}))$

where for $1 \le j \le |Y_1|$, y_j is the significant subarray (see below) of (P,Q) such that $rel(\text{c})_{Y_{1,j}} \in cont(\text{y}_j)$; and for $1 \le j \le |Z_i|$, z_j is the significant subarray of (P,Q) such that $rel(\text{c})_{Z_{i,j}} \in cont(\text{z}_j)$.

We define *s2a-group* analogously.

The function *n* in this definition associates names with certain subarrays, and, together with another naming function n_1, will be defined later.

Definition 12: Let c be a characteristic that does not occur in repeat (P,Q), and is a summary of some matches in the 1-blocks of (P,Q), and *ex*(c) = $\approx(<0>, g(h_1^k(e_1(Y_1,$

Z_1), …, $e_1(Y_k, Z_1)$),…, $h_q^k(e_q(Y_1, Z_q), …, e_q(Y_k, Z_q)), Z_0$); then the *s1-group of* (P,Q) *corresponding to* c is the pair (*ex, cont*) such that

- *cont* = {c}
- *ex* = $g(n(A_1), …, n(A_q), n(s_1), …, n(s_{|Z_0|}))$

where for $1 \leq j \leq |Z_0|$, s_j is the significant subarray (see below) of (P,Q) such that $rel(c)_{Z_{0,j}} \in cont(s_j)$

We define *s2-group* analogously.

For example, in Figure 3.3, s1-group corresponds to x and s1a-group is associated with the s1-group via the *ex* component. Because the aggregator in s1a refers to vx, a dotted arrow from s1a to *head* of r1s-group shows this dependency.

Definition 13: If C is a 1-block of (P,Q) and $C_{1,1}$ is not a summary, the *r1-group of* (P,Q) *corresponding to* C is the triple (*ex*, *head*, *tail*) such that

- *tail* is the set of all characteristics in C_1 except the first
- *head* = {$C_{1,1}$}
- *ex* = $f(n(s_1),…, n(s_m))$ if $C_{1,1} = rel(C_{1,1})_0$, $ex(C_{1,1}) = \approx(<0>, f(<1>,…,<m>)$, and for $1 \leq j \leq m$, s_j is the significant subarray (see below) of (P,Q) such that $rel(C_{1,1})_j \in cont(s_j)$, and *ex* = **nil** otherwise.

We define *r2-group* analogously.

For example, in Figure 3.3, the *head* and the *tail* of the r1-group corresponding to {g1,g2,g3} and {h1,h2,h3} respectively.

Definition 14: Let C be a 1-block of (P,Q), $C_{1,1}$ is a summary, and $ex(C_{1,1}) = \approx(<0>, g(h_1^k(e_1(Y_1, Z_1), …, e_1(Y_k, Z_1)),…, h_q^k(e_q(Y_1, Z_q), …, e_q(Y_k, Z_q)), Z_0)$; then for $1 \leq i \leq q$, the i^{th} *r1a-group of* (P,Q) *derived from* C is the 4-tuple A = (*ex*, *id*, *head*, *tail*), where

- *tail* is the set of all characteristics in C_1 except the first
- *head* = $\{C_{1,1}\}$
- *id* is the identity of the aggregator function h_i
- $ex = h_i(n(head(A)), e_i(n(y_1),\ldots, n(y_{|Y_1|}), n(z_1), \ldots, n(z_{|z_i|}))$

where for $1\leq j\leq|Y_1|$, y_j is the significant subarray (see below) of (P,Q) such that $rel(C_{1,1})_{Y_{1,j}} \in cont(y_j)$; and for $1\leq j\leq|Z_i|$, z_j is the significant subarray of (P,Q) such that $rel(C_{1,1})_{Z_{i,j}} \in cont(z_j)$.

We define *r2a-group* similarly.

Definition 15: If C is a 1-block of (P,Q), $C_{1,1}$ is a summary, and $ex(C_{1,1}) = \approx(<0>, g(h_1^k(e_1(Y_1, Z_1), \ldots, e_1(Y_k, Z_1)),\ldots, h_q^k(e_q(Y_1, Z_q), \ldots, e_q(Y_k, Z_q)), Z_0)$; then the *r1s group of* (P,Q) *corresponding to* C is the 3-tuple (*ex*, *head*, *tail*) such that

- *tail* is the set of all characteristics in C_1 except the first
- *head* = $\{C_{1,1}\}$
- $ex = g(n(head(A_1)), \ldots, n(head(A_q)), n(s_1), \ldots, n(s_k))$

where for each i ($1\leq i\leq q$), A_i is the i^{th} *r1a-group corresponding to* C, and for $1\leq j\leq|Z_i|$, s_j is the significant subarray (see below) of (P,Q) such that $rel(C1,1)_{Z_{i,j}} \in cont(s_j)$.

We define *r2s-group* similarly.

For example, in Figure 3.3 the *head* and *tail* of the r1s-group correspond to {v,vx,vp1,vp2,vp3} and {w,wx,wp1,wp2,wp3} respectively. The r1a-groups are associated with the r1s-group via the *ex* components. The dotted lines from r1a-groups to the other groups show that these aggregators are dependent on the other groups.

Definition 16: If (R,S) is a 1,2-block of (P,Q), a *b-group of* (R,S) is a pair of the form (*ex, cont*) such that, for some y in the first slice of R and the first slice of S:

- *cont* = { x | *rect*(x)=*rect*(y) and x occurs in the first slice of R and the first slice of S}
- *ex* = *f*(*n*(s_1),…, *n*(s_m)) if for some x ∈ *cont*, x=*rel*($x)_0$, *ex*(x) = ≈(<0>, *f*(<1>,…,<*m*>), and for $1\leq j\leq m$, s_j is the significant subarray (see below) of (P,Q) such that *rel*($x)_j$ ∈cont(s_j). *ex* = **nil** otherwise.

Definition 17: If (R,S) is a 1,2-block of (P,Q), the *rr-group of* (P,Q) *corresponding to* (R,S) is the pair (*head, tail*), where :

- *tail* = { x | characteristic x occurs in some slice of R other than the first }
- *head* is a pair of the form (*head, tail*) where
    - *tail* = { x | characteristic x occurs in the first slice of R, and some slice of S other than the first }
    - *head* is the set of all *b*-groups of (R,S).

For example in Figure 3.3, the *head* of rr-group consists of three b-groups which are {a1},{b1},{c1,v1,cp1} and the *tail* of the *head* of the rr-group corresponds to {a2, b2, c2, v2, cp2, a3, b3, c3, v3, cp3}. Also the *tail* of the rr-group corresponds to {d1, e1, f1, w1, fp1, d2, e2, f2, w2, fp2, d3, e3, f3, w3, fp3}.

Definition 18: A *significant subarray* of a repeat (P,Q) is either an *e*-group*,* a,*b*-group, or the head of an *r1, r2, r1s, r2s, r1a* or *r2a*-group. A *named subarray* is either a significant subarray, a group, the head or tail of a group, or the tail of the head of an *rr*-group.

The functions *n* and n_1 are one-to-one functions from the named subarrays of (P,Q) to two disjoint sets of symbols .

Definition 19: If A and B are groups of a repeat (P,Q), then A and B are *consistent* iff

either A and B both correspond to 1-blocks, the matches of which are consistent;

or A and B both correspond to 2-blocks, the matches of which are consistent;

or A corresponds to a 1-block X, B to a 1,2-block (R,S), or vice versa, and X∪R is consistent;

or A corresponds to a 2-block X, B to a 1,2-block (R,S), or vice versa, and X∪S is consistent;

or there is a group X such that A and X are consistent, and B and X are consistent.

Consistency is an equivalence relation on the set of all *rr*, *r1s*, *r2s*, *r1* and *r2*-groups. A *cluster* of (P,Q) is an equivalence class under this relation.

Definition 20: The *model* for a repeat is the set of all groups and *clusters* of the repeat.

3.5 Generating an L-sheets Program

The model of a repeat captures the essential elements of the repeat. In this section, we show how to generate an L-sheets program from it. We start by defining various functions that operate on the model. These functions are used to generate sets of tuples that are processed by an algorithm called `parameter`, listed in Appendix A, that generates the parameters of the forms that make up the L-sheets program.

The input to this algorithm is a set of 7-tuples of the form (*c,t,l,b,r,h,w*), where the first component is a set of strings, which is either empty or contains name or content of a named subarray of the model. The remaining components provide geometrical information about the named subarray. The algorithm incrementally builds two lists representing an L-sheets parameter as described in section 3.1, by selecting an element from the input set and adding corresponding elements to each output list. If the selected input element overlaps with an element of one of the current output lists, the overlapped output list element is removed and new elements added corresponding to the overlapped and non-overlapped parts of the input tuple and the overlapped output list element.

Table 3.1 defines functions that return the geometrical properties of an entity. For example, consider the r2-group C in Figure 3.3. To compute *h*(C), we note that *l*(*head*(C)) = *l*(Ch) = *min*{ *l*(r1) } = 7, and *l*(*tail*(C)) = *l*(Ct) = *min*{ *l*(r2), *l*(r3) } = 8. Hence *l*(*head*(C)) ≠ *l*(*tail*(C)), so that h(C) = 1. Similarly, to compute *w*(C), we note that *t*(*head*(C)) = *t*(Ch) = *min* { *t*(r1) } = 9, and *t*(*tail*(C)) = *t*(Ct) = *min* { *t*(r2), *t*(r3) } = 9, so since these values are equal, *w*(C) = *, indicating that C represents a group of horizontally repeating items.

	x is *s1*, *s2*, *s1a*, or *s2a*	x∈ *head*(*head*(y)) where y is *rr*	x is *e* or *b*	x=*head*(y) where y is *r1*, *r2*, *r1s*, *r2s*, r1a, or *r2a*	x=*tail*(y) where y is *r1*, *r2*, *r1s*, *r2s*, r1a, *r2a* or *rr*	x= *tail*(*head*(y)) where y is *rr*	x= *head*(*head*(y)) where y is *rr*
t(x)	*t*(*cont*(x))		min { *t*(z) | z∈x }				
l(x)	*l*(*cont*(x))		min { *l*(z) | z∈x }				
b(x)	*b*(*cont*(x))		max { *b*(z) | z∈x }				
r(x)	*r*(*cont*(x))		max { *r*(z) | z∈x }				
h(x)	1				*h*(y)	*h*(*head*(y))	*b*(x)–*t*(x)+1
w(x)	1				*w*(y)	*w*(*head*(y))	*r*(x)–*l*(x)+1

	x=*head*(y) where y is *rr*	x is *r1*, *r2*, *r1s*, *r2s*, *r1a*, or *r2a*	x is *rr*
t(x)	*t*(*head*(*x*))	*t*(*head*(x))	
l(x)	*l*(*head*(*x*))	*l*(*head*(x))	
b(x)	*b*(*tail*(x))	*b*(*tail*(x))	
r(x)	*r*(*tail*(x))	*r*(*tail*(x))	
h(x)	* if *l*(*head*(x)) = *l*(*tail*(x)) *b*(x)–*t*(x)+1 otherwise	* if *l*(*head*(x)) = *l*(*tail*(x)) 1 otherwise	*
w(x)	* if *t*(*head*(x)) = *t*(*tail*(x)) *r*(x)–*l*(x)+1 otherwise	* if *t*(*head*(x)) = *t*(*tail*(x)) 1 otherwise	*

Table 3.1 *t,l,b,r,h,w* functions for each group or the head and tail of the groups

The remaining functions, defined in Table 3.2, compute the sets of tuples that are provided as input to the `parameter` function. To simplify the presentation, we use *geom*(x) as an abbreviation for *t*(x),*l*(x),*b*(x),*r*(x),*h*(x),*w*(x).

x	***name*(x)**	***name*$_1$(x)**
named subarray	{({*n*(x)}, *geom*(x))}	{({*n*$_1$(x)}, *geom*(x))}

x	***a*(*x*)**	***c*(x)**	***b*(x)**
rr	{({*n*(x)}, *geom*(*head*(*head*(x)))}, ({*n*(x),*geom*(*tail*(x))}})	–	∅
r1a	–	{({*ex*(x)}, *geom*(*head*(x)))}	{({*id*(x)}, *geom*(*head*(x)))}
r1, r2, r1s, r2s, r2a	{({*n*(x)}, *geom*(*head*((x)))}, ({*n*(x),*geom*(*tail*(x))}})	{({*ex*(x)}, *geom*(*head*(x)))}	∅
e	–	{(∅, *geom*(x))}	{(∅, *geom*(x))}
b, s1, s2	–	{({*ex*(x)}, *geom*(x))} if *ex*(x)≠**nil** {(∅, *geom*(x))} otherwise	–
s1a, s2a	–	{({*ex*(x)}, *geom*(x))}	{({*id*(x)}, *geom*(x))}

x	*m*(x)
parameter	$\{ z \mid z \in a(y) \text{ for some } y \in x \}$
s1, s2	*c*(x)
rr,r1,r2,r1s,r2s,r2a	*a*(x)
any other group	*name*(x)

x	*f*(x)	*f1*(x)
r2, r2s	*c*(x) ∪ *name*(*head*(x)) ∪ *name*(*tail*(x))	*name*(*tail*(x))
r2a	*name*(*head*(x)) ∪ *name*(*tail*(x))	*name*(*tail*(x))
s2a	*c*(x)	*name*(x)
e	*name*(x)	*name*(x)

x	*p*(x)	*p*1(x)	*p*2(x)	*p*3(x)
rr	*name*(head*(*x)) ∪ *name*(*tail*(x))	*name*(*tail*(x))	*name*(*head*(x))	*b*(x)
r1, r1s	*c*(x) ∪ *name*(*head*(x)) ∪ *name*(*tail*(x))	*name*(*tail*(x))	–	*b*(x)
r2a	*name*(x)	$name_1$(x)	$name_2$(x)	{({*id*(x)},*geom*(x))}
s1a	*c*(x)	*name*(x)	–	*b*(x)
e	*name*(x)	*name*(x)	*name*(x)	*b*(x)
r1a	–	–	*name*(x)	*b*(x)

x	*q*(x)	q_1(x)	q_2(x)	q_3(x)
rr	$\{ c(y) \mid y \in head(head(x))\} \cup \{ name(y) \mid y \in head(head(x))\} \cup$ *name*(*tail*(*head*(x)))	-	*name*(*tail*(*head*(x)))	-
r1a	*c*(x)	-	*name*(*head*(x))	-
r2a	*c*(x) ∪ $name_1$(*tail*(x))	*name*(*head*(x)) ∪ *name*(*head*(x))	$name_1$(*tail*(x))	*name*(*tail*(x))
r1,r1s	*name*(*head*(*x*))	-	*name*(*head*(x))	-
e	*name*(x)	*n*(x)	*name*(x)	-

Table 3.2 Necessary functions for creating L-sheets programs.

We now show how the L-sheets schema for a repeat can be constructed from the model. We assume an arbitrary but fixed ordering of the clusters and groups of the model, and that the following sets which occur in the schema presented below are ordered accordingly.

P = the set of all clusters, *s1*, *s2*, and *e*-groups.

Y = the set of all *rr*, *r1*, *r1s*, *s1a* and *e*-groups.

Z = the set of all *r2*, *r2s*, *s2a* and *e*-groups.

W = the set of all *rr, r1, r1s, r1a* and *e*-groups.

V = the set of all *r2a*-groups.

Other conventions used in the presentation of this L-sheets schema are as follows. First, a term consisting of a function applied to one the above sets of groups and clusters produces the list obtained by applying the function to each element of the set, ordered as noted above. Second, although each of the functions defined in the tables produces a set to be processed by the parameter algorithm, for simplicity, we use it in the L-sheets schema to denote the output of the parameter algorithm applied to the set produced by the function. Finally, concatenation of lists is denoted by •. The L-sheets schema for a model is as follows.

T(*m*(P)) :- T1(*m*(Y)•*m*(V)), F(*m*(Z)•*m*(V)).

F(*f*(Z)•*f*(V)) :- F(*f*1(Z)• *f*1(V)).

F(*b*(Z)•*b*(V)).

T1(*p*(Y)•*p*(V)) :- T1(*p*1(Y)•*p*1(V)), T2(*p*2(W)•*p*(V)•*p*1(V)).

T1(*b*(Y)•*p*3(V)).

T2(*q*(W)•*q*(V)•*q*1(V)) :- T2(*q*2(W)•*q*2(V)•*q*3(V)).

T2(*b*(W)•*b*(V)•*b*(V)).

Figure 3.4 depicts the L-sheets program obtained from the spreadsheet in Figure 3.2. Each array in this program is labelled with the type of entity in the model in Figure 3.3

from which it was generated. These labels are not part of the program.

To conclude, we note that some spreadsheets may include only linear repeats, rather than the more general two-dimensional ones, and show how to derive L-sheets schema for such cases.

If (P,Q) is a repeat and Q is smaller than the threshold θ, then the following groups will not occur: *rr, r1s, r1a, r2, r2s, r2a, s2, s2a*. The sets of items that occur in the argument lists of the forms of the L-sheets schema are modified to take these changes into account, as below.

P = the set of all clusters, *s1*, and *e*-groups.

Y = the set of all *r1*, *s1a* and *e*-groups.

Z = the set of all *e*-groups.

W = the set of all *r1* and *e*-groups.

V = ∅.

Any term constructed from a set which is empty or includes only e-groups, can be deleted, as can any form, all arguments of which are deleted. Hence, the resulting L-sheets schema is as follows.

T(*m*(P)) :- T1(*m*(Y)~~•*m*(V)), F(*m*(Z)•*m*(V)).~~

~~F(*f*(Z)•*f*(V)) :- F(*f*1(Z)• *f*1(V)).~~

~~F(*b*(Z)•*b*(V)).~~

T1(*p*(Y)~~•*p*(V)~~) :- T1(*p*1(Y)•*p*1(V)), T2(*p*2(W)~~•*p*(V)•*p*1(V)~~).

T1(*b*(Y)~~•*p*3(V)~~).

T2(*q*(W)~~•*q*(V)•*q*1(V)~~) :- T2(*q*2(W)~~•*q*2(V)•*q*3(V)~~).

T2(*b*(W)~~•*b*(V)•*b*(V)~~).

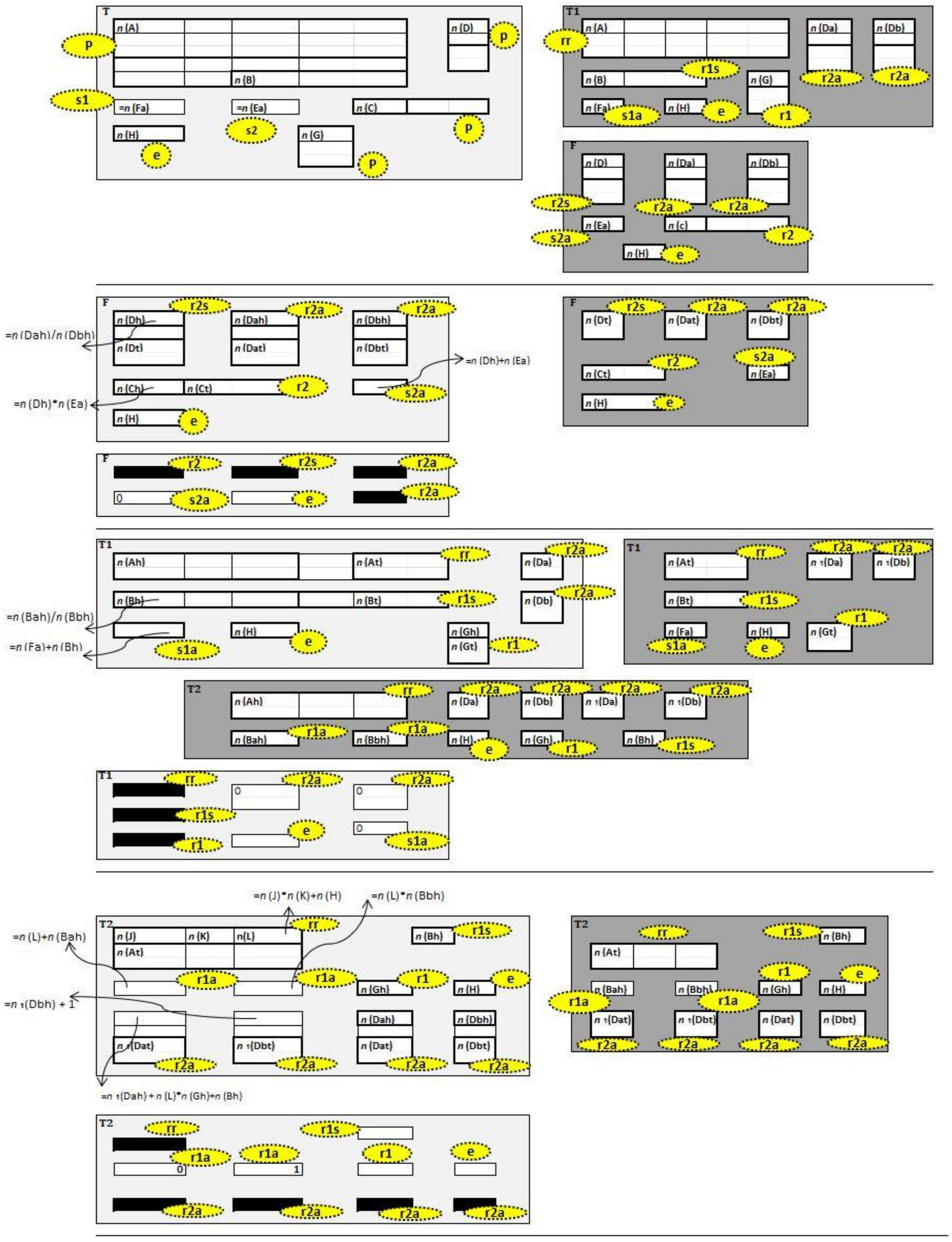


Figure 3.4 L-sheets program for Figure 3.2

Any term constructed from a set which is empty or includes only e-groups, can be deleted, as can any form, all arguments of which are deleted. Hence, the resulting L-sheets schema is as follows.

T(*m*(P)) :- T1(*m*(Y)~~•*m*(V)), F(*m*(Z)•*m*(V)).~~

~~F(*f*(Z)•*f*(V)) :- F(*f*1(Z)• *f*1(V)).~~

~~F(*b*(Z)•*b*(V)).~~

T1(*p*(Y)~~•*p*(V)~~) :- T1(*p*1(Y)•*p*1(V)), T2(*p*2(W)~~•*p*(V)•*p*1(V)~~).

T1(*b*(Y)~~•*p*3(V)~~).

T2(*q*(W)~~•*q*(V)•*q*1(V)~~) :- T2(*q*2(W)~~•*q*2(V)•*q*3(V)~~).

T2(*b*(W)~~•*b*(V)•*b*(V)~~).

Finally, we note that the functions *b* and *q* do not insert any formulae into the arrays they create from r1 and e-groups. Hence execution of T2 accomplishes nothing, so all forms involving T2, and the templates for T2, can be deleted. Hence the schema reduces to:

(1) T(m(P)) :- T1(m(Y)).

T1(p(Y)) :- T1(p1(Y)).

T1(p3(Y)).

Now consider the other case, when (P,Q) is a repeat and P is smaller than the threshold. In this case the following groups will not occur: rr, r2s, r2a, r1, r1s, r1a, s1, s1a. Modifying the sets of items argument lists of the forms of the L-sheets schema, we obtain:

P = the set of all clusters, *s2* and *e*-groups.

Y = the set of all *e*-groups.

Z = the set of all *r2*, *s2a* and *e*-groups.

W = the set of all *e*-groups.

V = ø.

Deleting terms and forms from the L-sheets schema, as described above, reduce it as follows.

T(*m*(P)) :- ~~T1(*m*(Y)•*m*(V)),~~ F(*m*(Z)~~•*m*(V)~~).

F(*f*(Z)~~•*f*(V)~~) :- F(*f*1(Z)~~• *f*1(V)~~).

F(*b*(Z)~~•*b*(V)~~).

~~T1(*p*(Y)•*p*(V)) :- T1(*p*1(Y)•*p*1(V)), T2(*p*2(W)•*p*(V)•*p*1(V)).~~

~~T1(*b*(Y)•*p*3(V)).~~

~~T2(*q*(W)•*q*(V)•*q*1(V)) :- T2(*q*2(W)•*q*2(V)•*q*3(V)).~~

~~T2(*b*(W)•*b*(V)•*b*(V)).~~

This reduces to:

(2) T(m(P)) :- F(m(Z)).

F(f(Z)) :- F(f1(Z)).

F(b(Z)).

By comparing the textual parameters in (1) and (2), Table 3.3 is derived.

Hence, the templates T1 in (1) and F in (2) are equivalent, and therefore templates T in (1) and T in (2) are also equivalent.

Finally, we note that each linear repeat is represented in the derived L-sheets program by recursion. This is not necessary if every match in a linear repeat has a gap of 1. The parameters derived from the groups of such a linear repeat could be more simply represented in the L-sheets program as arrays of variable height or width, removing the need for one level of recursion.

	(1) $y \in Y$	**(2) $z \in Z$**	
	$y = r1$, $z =$	$y = s1a$, $z = s2a$	$y = e$, $z = e$
$m(y)$	$a(y)$	$name(y)$	$name(y)$
$m(z)$	$a(z)$	$name(z)$	$name(z)$
$p(y)$	$c(y) \cup name(head(y)) \cup name(tail(y))$	$c(y)$	$name(y)$
$f(z)$	$c(z) \cup name(head(z)) \cup name(tail(z))$	$c(z)$	$name(z)$
$p_1(y)$	$name(tail(y))$	$name(y)$	$name(y)$
$f_1(z)$	$name(tail(z))$	$name(z)$	$name(z)$
$p_3(y)$	$\emptyset$	$\{(\{id(y)\}, geom(y))\}$	$\{(\emptyset, geom(y))\}$
$b(z)$	$\emptyset$	$\{(\{id(z)\}, geom(z))\}$	$\{(\emptyset, geom(z))\}$

Table 3.3 comparing the textual parameters computed in (1) and (2)

Chapter 4 Conclusion

4.1 Evaluation

In this chapter, we will compare our method for inferring templates with the process for inferring ViTSL templates discussed in chapter 2 [36]. In this thesis we do not compare our method for inferring templates with ClassSheets template extraction process [38], discussed in chapter 2, because it deals not with formulae, but with data dependencies, so it aims to discover relational tables in spreadsheets used as databases.

a) The ViTSL template language requires cells which are repeated together to be aligned, either horizontally or vertically, as in the budget example shown in Figure 3.2. Hence it cannot represent repeats in which related repeated items are not aligned, for example the sequences of cells R3C2, R3C6 and R10C6, 11C6 in Figure 3.2. Note that this is a restriction imposed by the ViTSL template language, rather than the associated process for inferring templates. Our definitions allow related, repeated items not only to be unaligned, but also to be orthogonal.

b) Our methodology allows the inferring of templates from spreadsheets when matches are overlapped. In other words, we can extract templates from those spreadsheets when data or formula cells are referred from different formula cells more than once. Templates from these kind of spreadsheets cannot be extracted by inferring templates for ViTSL process. A good example could be a spreadsheet which shows the Fibonacci numbers [41].

c) L-sheets can depict separate parameters, with no spatial relationship. For example in Figure 3.2, cells R10C8, R10C9 and R10C10 refer to R3C9, R5C9 and R7C9 respectively but the corresponding cells do not have any spatial relationships. As Figure 3.4 shows, two parameters which encompass these cells are not geometrically related. But in ViTSL, regions with no spatial relationship should be considered in one template which can mislead the user.

There are some directions for furthering this work, including the following items:

1- Devising and implementing the algorithms for extracting the structures defined in chapter 3 from spreadsheets.

2- The process for extracting spreadsheet structure, inherent in our definitions, could be augmented with algorithms that detect and semantically analyse headers, such as those in UCheck, described in Chapter 2. These operations the system may apply some error detection algorithms such as UCheck or automatic detection of dimension errors (which are described in chapter 2).

Bibliography

[1] C. Marcel. Mattesich (R.) - simulation of the firm through a budget computer program. *Revue Économique 17(4),* pp. 692-693.Jan 1966 Available: http://ideas.repec.org/a/prs/reveco/reco_0035-2764_1966_num_17_4_407724_t1_0692_0000_001.html.

[2] LANPAR information. Internet: www.renepardo.com/articles/spreadsheet.pdf [Nov, 24, 2010]

[3] Power D. J. A brief history of spreadsheets. Internet: http://dssresources.com/history/sshistory.html [Dec, 12 , 2010]

[4] Bill Jelen. "25 Cool Excel Downloads", Holy Macro, pp.52-70. May 2005

[5] VisiCalc picture. Internet : http://listverse.files.wordpress.com/2008/09/visicalc.png [Nov. 23, 2010].

[6] Excel 1.0 picture. Internet: http://royal.pingdom.com/2009/06/17/first-version-of-todays-most-popular-applications-a-visual-tour/ [June, 17, 2009]

[7] C. Scaffidi, M. Shaw and B. Myers. Estimating the numbers of end users and end user programmers. Presented at Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing. Available: http://portal.acm.org/citation.cfm?id=1092357.1092394.

[8] C. Chambers and M. Erwig. Automatic detection of dimension errors in spreadsheets. *J.Vis.Lang.Comput. 20(4),* pp. 269-283, Aug 2009. Available: http://portal.acm.org/citation.cfm?id=1568786.1568909.

[9] Mars climate orbiter team finds likely cause of loss. Internet: http://mars.jpl.nasa.gov/msp98/news/mco990930.html [Dec, 12,2010].

[10] J. R. Ruthruff, S. Prabhakararao, J. Reichwein, C. Cook, E. Creswick and M. Burnett. Interactive, visual fault localization support for end-user programmers. *J.Vis.Lang.Comput. 16(1-2),* pp. 3-40. Feb 2005. Available: http://dx.doi.org/10.1016/j.jvlc.2004.07.001.

[11] A. J. Ko and B. A. Myers. Development and evaluation of a model of programming errors. Presented at Human Centric Computing Languages and Environments. Proceedings IEEE Symposium on. pp 7-14, USA, Oct 2003

[12] A. J. Ko and B. A. Myers. Designing the whyline: A debugging interface for asking questions about program behavior. Presented at CHI '04: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.2004 Available: http://www.cs.cmu.edu/\~{}ajko/papers/DesigningTheWhyline.pdf

[13] R. Abraham and M. Erwig. UCheck: A spreadsheet type checker for end users. *J.Vis.Lang.Comput. 18(1),* pp. 71-95. Feb 2007. Available: http://portal.acm.org/citation.cfm?id=1223500.1223552

[14] C. Chambers and M. Erwig. Combining Spatial and Semantic Label Analysis. IEEE Symposium on VL/HCC, Presented at Visual Languages and Human-Centric Computing, pp. 235-232.Oct 2009.

[15] R. Abraham and M. Erwig. Header and unit inference for spreadsheets through spatial analyses. Presented at Visual Languages and Human Centric Computing, IEEE Symposium on. pp165-172.2004

[16] R. Mittermeir and M. Clermont. Finding high-level structures in spreadsheet programs. Presented at Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02). Pp 221-230.Washington, DC, USA 2002. Available: http://portal.acm.org/citation.cfm?id=882506.885148

[17] R. R. Panko. Applying code inspection to spreadsheet testing. *J. Manage. Inf. Syst. 16(2),* pp. 159-176.Sept 1999. Available: http://portal.acm.org/citation.cfm?id=1189438.1189448.

[18] R. Abraham, M. Erwig, S. Kollmansberger and E. Seifert. Visual specifications of correct spreadsheets. Presented at Visual Languages and Human-Centric Computing, 2005 IEEE Symposium on.

[19] J. Sajaniemi. Modeling spreadsheet audit: A rigorous approach to automatic visualization. 1998 Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.35.2163

[20] M. Erwig and M. M. Burnett. Adding apples and oranges. Presented at Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages. Available: http://portal.acm.org/citation.cfm?id=645772.667957.

[21] M. Burnett, A. Sheretov, B. Ren and G. Rothermel. Testing homogeneous spreadsheet grids with the what you see is what you test methodology. *IEEE Trans.Softw.Eng. 28(6),* pp. 576-594. June 2002. Available: http://portal.acm.org/citation.cfm?id=570528.570531.

[22] T. SH Teo and M. Tan. Quantitative and qualitative errors in spreadsheet development. Presented at Proceedings of the 30th Hawaii International Conference on System Sciences: Information System Track-Organizational Systems and Technology - Volume 3. pp149-150. USA 1997. Available: http://portal.acm.org/citation.cfm?id=938435.938681.

[23] D. Chadwick, B. Knight and K. Rajalingham. Quality control in spreadsheets: A visual approach using color codings to reduce errors in formulae. *Software Quality Control 9(2),* pp. 133-143. June 2001. Available: http://portal.acm.org/citation.cfm?id=599123.599202.

[24] M. Tukiainen. Uncovering effects of programming paradigms: Errors in two spreadsheet systems. Presented at In 12th Workshop of the Psychology of Programming Interest Group (PPIG).

[25] K. Rajalingham, D.R. Chadwick, B. Knight,(2001). Classification of spreadsheet errors, in: Symposium of the European Spreadsheet Risks Interest Group (EuSpRIG)

[26] T. Antoniu, P. A. Steckler, S. Krishnamurthi, E. Neuwirth and M. Felleisen. Validating the unit correctness of spreadsheet programs. Presented at Proceedings of the 26th International Conference on Software Engineering.pp 439-448. USA 2004 Available: http://portal.acm.org/citation.cfm?id=998675.999448.

[27] Excel smart tools information. Internet: http://www.ozgrid.com/Services/ExcelSmartTools.html [Nov, 24, 2010]

[28] M. Burnett, C. Cook and G. Rothermel. End-user software engineering. *Commun ACM 47(9),* pp. 53-58.Sept 2004. Available: http://doi.acm.org/10.1145/1015864.1015889.

[29] G. Rothermel, M. Burnett, L. Li, C. Dupuis and A. Sheretov. A methodology for testing spreadsheets. *ACM Trans.Softw.Eng.Methodol. 10(1),* pp. 110-147.Jan 2001 Available: http://doi.acm.org/10.1145/366378.366385.

[30] R. Abraham and M. Erwig.(2007) GoalDebug: A spreadsheet debugger for end users. Presented at Software Engineering 29th International Conference., pp 251-260. USA ICSE 2007.

[31] Excel spreadsheet auditor components. Internet: http://www.freedownloadscenter.com/Business/Finance/Excel_Spreadsheet_Auditor.html [Nov, 24, 2010]

[32] R. Abraham and M. Erwig. Goal-directed debugging of spreadsheets. Presented at Proceedings of the IEEE Symposium on Visual Languages and Human-Centric

Computing. pp 37-44. USA 2005 Available: http://portal.acm.org/citation.cfm?id=1092357.1092375.

[33] R. Abraham and M. Erwig. Test-driven goal-directed debugging in spreadsheets. Presented at Visual Languages and Human-Centric Computing VL/HCC. IEEE Symposium on. pp.131-138, Sept. 2008

[34] R. Abraham and M. Erwig. AutoTest: A Tool for Automatic Test Case Generation in Spreadsheets. Presented at Visual Languages and Human-Centric Computing IEEE Symposium on. pp 33-50. USA 2006

[35] M. J. Coblenz, A. J. Ko and B. A. Myers. Using objects of measurement to detect spreadsheet errors. Presented at Visual Languages and Human-Centric Computing, IEEE Symposium on. pp 314-316. USA 2005

[36] R. Abraham and M. Erwig. Inferring templates from spreadsheets. Presented at Proceedings of the 28th International Conference on Software Engineering. pp 182-191. China 2006. Available: http://doi.acm.org/10.1145/1134285.1134312.

[37] G. Engels and M. Erwig. ClassSheets: Automatic generation of spreadsheet applications from object-oriented specifications. Presented at Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering.pp.168-174. Mar 1994. Available: http://doi.acm.org/10.1145/1101908.1101929.

[38] J. Cunha, M. Erwig and J. Saraiva. Automatically inferring ClassSheet models from spreadsheets. *Visual Languages and Human-Centric Computing, IEEE Symposium on* pp. 93-100.Sept 2010.

[39] P. T. Cox. Enhancing the programmability of spreadsheets with logic programming. Presented at Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing. pp 87-94. 2007. Available: http://dx.doi.org/10.1109/VLHCC.2007.18.

[40] P. Cox and P. Nicholson. Unification of arrays in spreadsheets with logic programming. In *Practical Aspects of Declarative Languages*, pp 100-115. USA, Jan 2008 Available: http://dx.doi.org/10.1007/978-3-540-77442-6_8.

[41] How To Create Fibonacci Numbers in Excel. Internet: http://spreadsheets.about.com/od/spreadsheetlessons/qt/Fibonacci_Numbe.htm [Aug, 30,2011]

Appendix A : Generating L-sheets Parameters Algorithm

This Prolog program was provided by Dr.Phil Cox

```
/*-------- Make parameter from a set of rectangles ----------*/

parameter(Rects,(HP,VP)) :-
        build(h,Rects,[],H), quicksort(H,HS), compress(HS,HP),
        build(v,Rects,[],V), quicksort(V,VS), compress(VS,VP).

/*-------- Remove position information from rectangle -------*/

compress([],[]).
compress([(N,_,_,W)|R],[(N,W)|RC]) :-
        compress(R,RC).

/*-------- Build list of strips - first parameter indicates direction -
------*/

build(_,[],Strips,Strips).
build(D,[Rect|Rest],Strips,StripsOut) :-
        extract(D,Rect,N,Begin,End,Size),
        insert((N,Begin,End,Size),Strips,Strips1),
        build(D,Rest,Strips1,StripsOut).

/*-------- Gets the name and column or row range of rectangle -------*/

extract(h,(N,_,L,_,R,_,W),N,L,R,W).
extract(v,(N,T,_,B,_R,H,_),N,T,B,H).

/*-------- Incorporate a rectangle into a set of strips (columns or
rows)  -------*/

insert(X,[],[X]).
insert((N,B,E,S),[(N1,B1,E1,S1)|Rest],[(N1,B1,E1,S1)|StripsOut]) :-
        ( E < B1 ; B > E1 ), !,
        insert((N,B,E,S),Rest,StripsOut).
insert((N,B,E,S),[(N1,B1,E1,S1)|StripsIn],StripsOut) :-
        E > E1,
        split(S,S1,E1-B+1,E-E1,Sn1,Sn2), !,
        Bn is E1+1,
        insert((N,Bn,E,Sn2),StripsIn,Strips1), !,
        insert((N,B,E1,Sn1),[(N1,B1,E1,S1)|Strips1],StripsOut).
insert((N,B,E,S),[(N1,B1,E1,S1)|StripsIn],StripsOut) :-
        B < B1,
        split(S,S1, E-B1+1, B1-B, Sn2, Sn1), !,
        En is B1-1,
        insert((N,B,En,Sn1),StripsIn,Strips1), !,
        insert((N,B1,E,Sn2),[(N1,B1,E1,S1)|Strips1],StripsOut).
insert((N,B,E,S),[(N1,B1,E1,S1)|StripsIn],[(N1,Bn,E1,Sn2)|StripsOut])
:-
        E < E1,
        split(S,S1, E-B1+1, E1-E, Sn1, Sn2), !,
        Bn is E+1,
        insert((N,B,E,S),[(N1,B1,E,Sn1)|StripsIn], StripsOut).
insert((N,B,E,S),[(N1,B1,E,S1)|StripsIn],[(N1,B1,En,Sn1)|StripsOut]) :-
        B > B1,
        split(S,S1, E-B+1, B-B1, Sn2, Sn1), !,
        En is B-1,
        insert((N,B,E,S),[(N1,B,E,Sn2)|StripsIn],StripsOut).
insert((N,B,E,S),[(N1,B,E,S)|StripsIn],[(N2,B,E,S)|StripsIn]) :-
        concat(N,N1,N2).

/*-------- Computes sizes of parts of a split strip -------*/

split('*','*',_,R,'*',F) :- F is R, !.
split('*',_,L,_,F,'*') :- F is L, !.
split(_,_,L,R,S1,S2) :- S1 is L, S2 is R.

/*-------- Sorts strips from left to right (top to bottom)  -------*/

quicksort([],[]).
quicksort([X|Y],Z) :-
        partition(X,Y,Y1,Y2),
        quicksort(Y1,Z1),
        quicksort(Y2,Z2),
        concat(Z1,[X|Z2],Z).

partition(_,[],[],[]).
partition((_,F1,_,_),[(N,F2,Y,S)|R],[(N,F2,Y,S)|R1],R2) :-
        F2 < F1, !, partition((_,F1,_,_),R,R1,R2).
partition(X,[F|R],R1,[F|R2]) :-
        partition(X,R,R1,R2).

concat([],X,X).
concat(X,[],X).
concat([X|R],Y,[X|Z]) :- concat(R,Y,Z).
```