A Literate Program for Converting Tables to LongForm Dataframes

Jimmy Oh

Department of Statistics
University of Auckland

Abstract

TableToLongForm automatically converts hierarchical Tables intended for a human reader into a simple LongForm Dataframe that is machine readable. It does this by recognising positional queues present in the hierarchical Table (which would normally be interpreted visually by the human brain) to decompose, then reconstruct the data into a LongForm Dataframe. This document provides a gallery of all recognised patterns and structures, with accompanying toy examples, before finally going into depth on the workings of the code itself.

Contents

On Literate Programs

This software is presented as a *literate program* written in the *noweb* format (Ramsey 1994). It serves as both the documentation and container of the literate program. The `noweb` file can be used to produce both the *literate document* and the executable code.

The literate document is separated into *documentation chunks* and named *code chunks*. Each *code chunk* can contain code directly, or contain references to other *code chunks* which act as placeholders for the contents of the respective *code chunk*. The name of each *code chunk* should serve as a short description of the code it contains. Thus each *code chunk* provides an overview of its purpose by either directly containing code, or by containing the names of other *code chunks*. The reader is then free to delve deeper into the respective *code chunks* if desired.

1 Introduction

1.1 Motivation

In recent times there has been a movement toward *Open Data*, particularly for government data[1,2], yet there is still a prevalence of data releases being for direct human consumption, rather than for machine consumption. One symptom of this is the release of data in tabular form that relies on the human ability to identify patterns and discern structure, in order to decipher the data (henceforth referred to as a Table). Such tables are difficult to read and analyse with the computer, signficantly limiting potential applications of this 'open' data.

LongForm is a simple alternative method of releasing the data that, due to its simplicity, is both easy to implement and is machine readable, greatly enhancing potential applications of the data. It is easy to go from a simple format such as a LongForm Dataframe to any number of other forms of presentation, including hierarchical Tables more suitable for direct human consumption. However the converse is rarely true. This is where TableToLongForm comes in, providing a way to automatically convert hierarchical Tables to a simple LongForm Dataframe, thus enabling much greater utilisation of the data.

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	Labour Force Status by Sex by Sing/Comb Ethnic Group (Qrtly−Mar/Jun/Sep/Dec)												
2		Male											
3		European Only								Maori Only			
4		Persons Em	Persons Un	Not in Labo	Working Ag	Labour Forc	Unemploym	Employmen	Total Labou	Persons Em	Persons Un	Not in Labo	Working Ag
5	2007Q4	855.8	20.0	280.0	1155.8	75.8	2.3	74.0	875.8	71.1	6.1	28.1	105.3
6	2008Q1	863.0	25.4	283.5	1171.9	75.8	2.9	73.6	888.5	69.1	7.5	31.4	107.9
7	2008Q2	850.1	26.0	280.7	1156.8	75.7	3.0	73.5	876.1	67.2	5.7	27.4	100.2
8	2008Q3	839.6	29.8	285.9	1155.3	75.2	3.4	72.7	869.4	71.7	8.7	30.7	111.1
9	2008Q4	854.8	29.5	274.7	1158.9	76.3	3.3	73.8	884.2	76.1	8.5	28.5	113.1
10	2009Q1	845.0	35.4	279.4	1159.8	75.9	4.0	72.9	880.4	75.4	8.4	35.7	119.5
11	2009Q2	831.6	34.9	279.7	1146.2	75.6	4.0	72.6	866.5	74.2	9.9	33.1	117.3
12	2009Q3	813.3	42.5	290.4	1146.2	74.7	5.0	71.0	855.8	70.9	10.9	36.0	117.8
13	2009Q4	831.1	40.1	277.0	1148.2	75.9	4.6	72.4	871.2	71.7	13.6	33.2	118.5
14	2010Q1	822.5	36.4	283.2	1142.1	75.2	4.2	72.0	858.9	71.8	11.3	35.3	118.4
15	2010Q2	825.3	39.9	290.1	1155.3	74.9	4.6	71.4	865.2	71.9	13.7	33.7	119.2
16	2010Q3	836.9	31.0	287.1	1155.1	75.1	3.6	72.5	867.9	69.8	13.5	34.1	117.3
17	2010Q4	838.1	39.6	277.1	1154.8	76.0	4.5	72.6	877.7	70.7	14.4	36.4	121.5
18	2011Q1	829.6	36.8	281.2	1147.6	75.5	4.2	72.3	866.4	70.7	13.9	35.3	119.8
19	2011Q2	838.7	41.0	279.0	1158.7	75.9	4.7	72.4	879.6	67.1	10.5	37.1	114.7
20	2011Q3	830.5	34.6	280.2	1145.3	75.5	4.0	72.5	865.1	69.5	13.4	34.9	117.8
21	2011Q4	841.8	34.8	277.5	1154.1	76.0	4.0	72.9	876.6	69.2	12.7	36.1	118.0
22	2012Q1	843.1	43.3	282.7	1169.1	75.8	4.9	72.1	886.3	71.5	11.2	34.6	117.3
23	2012Q2	837.1	38.2	296.5	1171.8	74.7	4.4	71.4	875.3	66.2	10.6	33.0	109.7
24	2012Q3	833.0	38.0	298.4	1169.3	74.5	4.4	71.2	871.0	67.1	13.1	33.0	113.2
25	2012Q4	833.4	41.0	298.2	1172.6	74.6	4.7	71.1	874.4	63.0	12.3	35.4	110.7
26	2013Q1	832.0	35.8	294.7	1162.5	74.6	4.1	71.6	867.8	69.9	12.2	38.2	120.3
27	Table information:												
28	Units:												
29	Persons Employed in Labour Force: Number, Magnitude = Thousands												
30	Persons Unemployed in Labour Force: Number, Magnitude = Thousands												
31	Not in Labour Force: Number, Magnitude = Thousands												
32	Working Age Population: Number, Magnitude = Thousands												
33	Labour Force Participation Rate: Percent, Magnitude = Units												
34	Unemployment Rate: Percent, Magnitude = Units												
35	Employment Rate: Percent, Magnitude = Units												
36	Total Labour Force: Number, Magnitude = Thousands												
37	Footnotes:												
38													
39	Symbols:												
40	.. figure not available												
41	C: Confidential												
42	E: Early Estimate												
43	P: Provisional												

Figure 1: An example of a hierarchical Table. The data is Labour Force Status Survey data from Infoshare, Statistics New Zealand (2013) and in total spans 240 columns, making it suitable for neither man nor machine. However it is relatively tame in terms of how unsuitable for machines Tables can get and so could, with some manual labour, be read by a machine.

[1] "In many countries across the world, discussions, policies and developments are actively emerging around open access to government data." Davies and Bawa (2012a)

[2] "Over 100 OGD [Open Government Data] initiatives are active across the globe, ranging from community-led OGD projects in urban India, to a World Bank sponsored OGD programme in Kenya, government-led developments in Brazil, civil-society initiated work in Russia, and a World Wide Web Foundation supported programme in Ghana." Davies and Bawa (2012b)

	1	2	3	4	5	6	7	8	9	10	11	12	13
1				Persons En	Persons Un	Not in Labo	Working Ag	Labour Forc	Unemploym	Employmen	Total Labour Force		
2	Male	European C	2007Q4	855.8	20	280	1155.8	75.8	2.3	74	875.8		
3	Male	European C	2008Q1	863	25.4	283.5	1171.9	75.8	2.9	73.6	888.5		
4	Male	European C	2008Q2	850.1	26	280.7	1156.8	75.7	3	73.5	876.1		
5	Male	European C	2008Q3	839.6	29.8	285.9	1155.3	75.2	3.4	72.7	869.4		
6	Male	European C	2008Q4	854.8	29.5	274.7	1158.9	76.3	3.3	73.8	884.2		
7	Male	European C	2009Q1	845	35.4	279.4	1159.8	75.9	4	72.9	880.4		
8	Male	European C	2009Q2	831.6	34.9	279.7	1146.2	75.6	4	72.6	866.5		
9	Male	European C	2009Q3	813.3	42.5	290.4	1146.2	74.7	5	71	855.8		
10	Male	European C	2009Q4	831.1	40.1	277	1148.2	75.9	4.6	72.4	871.2		
11	Male	European C	2010Q1	822.5	36.4	283.2	1142.1	75.2	4.2	72	858.9		
12	Male	European C	2010Q2	825.3	39.9	290.1	1155.3	74.9	4.6	71.4	865.2		
13	Male	European C	2010Q3	836.9	31	287.1	1155.1	75.1	3.6	72.5	867.9		
14	Male	European C	2010Q4	838.1	39.6	277.1	1154.8	76	4.5	72.6	877.7		
15	Male	European C	2011Q1	829.6	36.8	281.2	1147.6	75.5	4.2	72.3	866.4		
16	Male	European C	2011Q2	838.7	41	279	1158.7	75.9	4.7	72.4	879.6		
17	Male	European C	2011Q3	830.5	34.6	280.2	1145.3	75.5	4	72.5	865.1		
18	Male	European C	2011Q4	841.8	34.8	277.5	1154.1	76	4	72.9	876.6		
19	Male	European C	2012Q1	843.1	43.3	282.7	1169.1	75.8	4.9	72.1	886.3		
20	Male	European C	2012Q2	837.1	38.2	296.5	1171.8	74.7	4.4	71.4	875.3		
21	Male	European C	2012Q3	833	38	298.4	1169.3	74.5	4.4	71.2	871		
22	Male	European C	2012Q4	833.4	41	298.2	1172.6	74.6	4.7	71.1	874.4		
23	Male	European C	2013Q1	832	35.8	294.7	1162.5	74.6	4.1	71.6	867.8		
24	Male	Maori Only	2007Q4	71.1	6.1	28.1	105.3	73.4	7.9	67.6	77.2		
25	Male	Maori Only	2008Q1	69.1	7.5	31.4	107.9	71	9.7	64.1	76.6		
26	Male	Maori Only	2008Q2	67.2	5.7	27.4	100.2	72.7	7.8	67	72.8		
27	Male	Maori Only	2008Q3	71.7	8.7	30.7	111.1	72.3	10.8	64.5	80.3		
28	Male	Maori Only	2008Q4	76.1	8.5	28.5	113.1	74.8	10	67.3	84.5		
29	Male	Maori Only	2009Q1	75.4	8.4	35.7	119.5	70.1	10.1	63.1	83.8		
30	Male	Maori Only	2009Q2	74.2	9.9	33.1	117.3	71.8	11.8	63.3	84.2		
31	Male	Maori Only	2009Q3	70.9	10.9	36	117.8	69.5	13.4	60.2	81.8		
32	Male	Maori Only	2009Q4	71.7	13.6	33.2	118.5	71.9	15.9	60.5	85.3		
33	Male	Maori Only	2010Q1	71.8	11.3	35.3	118.4	70.2	13.6	60.6	83.1		
34	Male	Maori Only	2010Q2	71.9	13.7	33.7	119.2	71.8	16	60.3	85.6		
35	Male	Maori Only	2010Q3	69.8	13.5	34.1	117.3	71	16.2	59.5	83.3		
36	Male	Maori Only	2010Q4	70.7	14.4	36.4	121.5	70	16.9	58.2	85.1		
37	Male	Maori Only	2011Q1	70.7	13.9	35.3	119.8	70.6	16.4	59	84.6		
38	Male	Maori Only	2011Q2	67.1	10.5	37.1	114.7	67.6	13.5	58.5	77.6		
39	Male	Maori Only	2011Q3	69.5	13.4	34.9	117.8	70.4	16.1	59	82.9		
40	Male	Maori Only	2011Q4	69.2	12.7	36.1	118	69.4	15.5	58.6	81.8		
41	Male	Maori Only	2012Q1	71.5	11.2	34.6	117.3	70.5	13.6	60.9	82.7		
42	Male	Maori Only	2012Q2	66.2	10.6	33	109.7	69.9	13.8	60.3	76.8		
43	Male	Maori Only	2012Q3	67.1	13.1	33	113.2	70.8	16.4	59.2	80.2		

Figure 2: An example of a LongForm Dataframe. This is the Labour Force Status Survey data after automatic conversion with TableToLongForm. Most, if not all statistical software (including spreadsheet software like Excel) can read, manipulate and run analyses on this without any problems.

1.2 The Plan of Attack

Unless the Table is horrible beyond mortal imagination, it should have some kind of pattern, such that a human will be able to discern the structure and hence understand the data it represents. This code attempts to algorithmically search for such patterns, discern the structure, then reconstruct the data into a LongForm Dataframe. Refer to Section 1.3 for a full gallery of currently recognised patterns.

The task can be seen to consist of three phases:

- Phase One is Identification (Section 3), which involves identifying the rows and columns where the labels and the data can be found.
- Phase Two is Discerning the Parentage (Section 4), which involves identifying the hierarchical structure of the data, based on the row and column labels.
- Phase Three is Reconstruction (Section 5), where we use what we've found in the first two phases to reconstruct the data into a LongForm Dataframe.

1.3 Recognised Patterns

Here we list, with toy examples, all the recognised patterns and structures. TableToLongForm should be able to process any combination of these patterns to automatically convert many different types of tables. The last example is a *complete* example that contains most of the recognised patterns in a single horrible table.

1.3.1 By Empty Below

	1	2	3	4	5	6
1			Column 1	Column 2	Column 3	Column 4
2	Row Parent1	Row Child1	10	20	30	40
3		Row Child2	11	21	31	41
4	Row Parent2	Row Child1	12	22	32	42
5		Row Child2	13	23	33	43

The most simple type of parentage, here the *parent* and *children* are in different columns and we can see which of the children belong to which parent through the use of empty space below each parent.

	1	2	3	4	5	6
1			Column 1	Column 2	Column 3	Column 4
2	Row Parent1	Row Child1	10	20	30	40
3	Row Parent1	Row Child2	11	21	31	41
4	Row Parent2	Row Child1	12	22	32	42
5	Row Parent2	Row Child2	13	23	33	43

1.3.2 By Empty Below Transposed

	1	2	3	4	5
1		Row Parent1		Row Parent2	
2		Row Child1	Row Child2	Row Child1	Row Child2
3	Column 1	10	11	12	13
4	Column 2	20	21	22	23
5	Column 3	30	31	32	33
6	Column 4	40	41	42	43

We note that parentage patterns recognised for row labels can often be applied to the transpose of column labels. This is how TableToLongForm deciphers most column labels, with some exceptions.

	1	2	3	4	5
1			Row Child1	Row Child2	
2	Row Parent1	Column 1	10	11	
3	Row Parent1	Column 2	20	21	
4	Row Parent1	Column 3	30	31	
5	Row Parent1	Column 4	40	41	
6	Row Parent2	Column 1	12	13	
7	Row Parent2	Column 2	22	23	
8	Row Parent2	Column 3	32	33	
9	Row Parent2	Column 4	42	43	

1.3.3 By Empty Right 1

	1	2	3	4	5	6	7
1				Column 1	Column 2	Column 3	Column 4
2	Row Parent1			10	20	30	40
3	Row Child1	Row Child−Child1		11	21	31	41
4	Row Child2	Row Child−Child2		12	22	32	42
5	Row Parent2			13	23	33	43
6	Row Child1	Row Child−Child1		14	24	34	44
7		Row Child−Child2		15	25	35	45

In this situation we have children in the same column as their parent. We can still recognise these as children if the children have children (*Child-Child*) in a different column, while the parent does not (and hence is Empty Right).

Note the values pertaining to the Parent (if any) are discarded. This is because they are assumed to simply represent the sum of their children's values. It is planned for a sum-check to be implemented later to make this more robust.

	1	2	3	4	5	6	7
1				Column 1	Column 2	Column 3	Column 4
2	Row Parent1	Row Child1	Row Child−Ch	11	21	31	41
3	Row Parent1	Row Child2	Row Child−Ch	12	22	32	42
4	Row Parent2	Row Child1	Row Child−Ch	14	24	34	44
5	Row Parent2	Row Child1	Row Child−Ch	15	25	35	45

1.3.4 By Empty Right 2

	1	2	3	4	5	6
1			Column 1	Column 2	Column 3	Column 4
2	Row Parent1		10	20	30	40
3		Row Child1	11	21	31	41
4		Row Child2	12	22	32	42
5	Row Parent2		13	23	33	43
6		Row Child1	14	24	34	44
7		Row Child2	15	25	35	45

Here we have both Empty Below and Empty Right. Either algorithm can handle this situation, but simply due to the ordering of the algorithms such situations are handled as Empty Right.

	1	2	3	4	5	6
1			Column 1	Column 2	Column 3	Column 4
2	Row Parent1	Row Child1	11	21	31	41
3	Row Parent1	Row Child2	12	22	32	42
4	Row Parent2	Row Child1	14	24	34	44
5	Row Parent2	Row Child2	15	25	35	45

1.3.5 By Empty Right 3

	1	2	3	4	5	6	7	8
1				Column 1	Column 2	Column 3	Column 4	
2	Row Super−Parent1			10	20	30	40	
3	Row Parent1			11	21	31	41	
4	Row Child1	Row Child−Child1		12	22	32	42	
5	Row Parent2			13	23	33	43	
6	Row Child1	Row Child−Child1		14	24	34	44	
7	Row Super−Parent2			15	25	35	45	
8	Row Parent1			16	26	36	46	
9	Row Child1	Row Child−Child1		17	27	37	47	
10	Row Parent2			18	28	38	48	
11	Row Child1	Row Child−Child1		19	29	39	49	

The "parent-child in the same column" situation can be extended further. Here we have parents (*Super-Parent*) who have children (*Parent*), who each further have children (*Child*), all in the same column. Such situations can still be recognised if the lowest-level children in the column (*Child*) have children in a different column (*Child-Child*), while its direct parents (*Parent*) each have children in the same column (*Child*) but not in a different column (is Empty Right), and the top-most parents (*Super-Parents*) also have no children in a different column (is also Empty Right).

The algorithm cannot currently handle super-super-parents.

	1	2	3	4	5	6	7	8
1					Column 1	Column 2	Column 3	Column 4
2	Row Super−P	Row Parent1	Row Child1	Row Child−Ch	12	22	32	42
3	Row Super−P	Row Parent2	Row Child1	Row Child−Ch	14	24	34	44
4	Row Super−P	Row Parent1	Row Child1	Row Child−Ch	17	27	37	47
5	Row Super−P	Row Parent2	Row Child1	Row Child−Ch	19	29	39	49

1.3.6 Multi-row Column Label

	1	2	3	4	5
1		Column	Column	Column	Column
2		Child1	Child2	Child3	Child4
3	Row 1	10	20	30	40
4	Row 2	11	21	31	41
5	Row 3	12	22	32	42
6	Row 4	13	23	33	43

Often column labels are physically split over multiple rows rather than making use of line breaks in the same cell. In such occurences, any row not identified as a parent are collapsed into a single row of labels. It is eventually planned for pattern recognition to be used here to make this collapsing smarter.

	1	2	3	4	5
1		Column Child1	Column Child2	Column Child3	Column Child4
2	Row 1	10	20	30	40
3	Row 2	11	21	31	41
4	Row 3	12	22	32	42
5	Row 4	13	23	33	43

1.3.7 Misaligned Column Label

	1	2	3	4	5	6	7	8	9
1			Column Parent1				Column Parent2		
2		Col Child1	Col Child2	Col Child3	Col Child4	Col Child1	Col Child2	Col Child3	Col Child4
3	Row 1	10	20	30	40	50	60	70	80
4	Row 2	11	21	31	41	51	61	71	81
5	Row 3	12	22	32	42	52	62	72	82
6	Row 4	13	23	33	43	53	63	73	83

Often column parents are physically centred over their children (N.B. where a spreadsheet's cell-merge feature is used to do the centering, the actual value is usually stored in the top-left cell and hence causes no problems). TableToLongForm makes use of pattern recognition to identify repeating patterns in the labels of the children, to help discern the correct parent for the children.

	1	2	3	4	5	6	7	8	9
1			Col Child1	Col Child2	Col Child3	Col Child4			
2	Column Paren	Row 1	10	20	30	40			
3	Column Paren	Row 2	11	21	31	41			
4	Column Paren	Row 3	12	22	32	42			
5	Column Paren	Row 4	13	23	33	43			
6	Column Paren	Row 1	50	60	70	80			
7	Column Paren	Row 2	51	61	71	81			
8	Column Paren	Row 3	52	62	72	82			
9	Column Paren	Row 4	53	63	73	83			

1.3.8 Find Single Table

	1	2	3	4	5
1	MISC INFORMATION				
2	MISC INFORMATION				
3		Column 1	Column 2	Column 3	Column 4
4	Row 1	10	20	30	40
5	Row 2	11	21	31	41
6	Row 3	12	22	32	42
7	Row 4	13	23	33	43
8	MISC INFORMATION		MISC INFORMATION		
9	MISC INFORMATION		MISC INFORMATION		

A table is often found amongst miscellaneous information we do not want. TableToLongForm is intended to have several algorithms to identify not only a single table, but multiple tables on the same 'page'. Currently however, it can only identify a single table per 'page' by searching for a block (rectangular region) of numbers, which is assumed to be our table of data.

	1	2	3	4	5
1		Column 1	Column 2	Column 3	Column 4
2	Row 1	10	20	30	40
3	Row 2	11	21	31	41
4	Row 3	12	22	32	42
5	Row 4	13	23	33	43

1.3.9 Complete Example

	1	2	3	4	5	6	7	8	9	10	11
1	MISC INFORMATION										
2					Column Parent1				Column Parent2		
3				Column	Column	Column	Column	Column	Column	Column	Column
4				Child1	Child2	Child3	Child4	Child1	Child2	Child3	Child4
5	Row Super−Parent			10	20	30	40	50	60	70	80
6	Row Parent1			11	21	31	41	51	61	71	81
7	Row Child1	Row Child−Child1		12	22	32	42	52	62	72	82
8		Row Child−Child2		13	23	33	43	53	63	73	83
9	Row Child2	Row Child−Child1		14	24	34	44	54	64	74	84
10		Row Child−Child2		15	25	35	45	55	65	75	85
11	Row Parent2			16	26	36	46	56	66	76	86
12	Row Child1	Row Child−Child1		17	27	37	47	57	67	77	87
13		Row Child−Child2		18	28	38	48	58	68	78	88
14	Row Child2	Row Child−Child2		19	29	39	49	59	69	79	89
15	MISC INFORMATION										
16	MISC INFORMATION										

A complete example containing a combination of many of the patterns listed above.

	1	2	3	4	5	6	7	8	9	10	11
1						Column Child1	Column Child2	Column Child3	Column Child4		
2	Column Paren	Row Super−P	Row Parent1	Row Child1	Row Child−Ch	12	22	32	42		
3	Column Paren	Row Super−P	Row Parent1	Row Child1	Row Child−Ch	13	23	33	43		
4	Column Paren	Row Super−P	Row Parent1	Row Child2	Row Child−Ch	14	24	34	44		
5	Column Paren	Row Super−P	Row Parent1	Row Child2	Row Child−Ch	15	25	35	45		
6	Column Paren	Row Super−P	Row Parent2	Row Child1	Row Child−Ch	17	27	37	47		
7	Column Paren	Row Super−P	Row Parent2	Row Child1	Row Child−Ch	18	28	38	48		
8	Column Paren	Row Super−P	Row Parent2	Row Child2	Row Child−Ch	19	29	39	49		
9	Column Paren	Row Super−P	Row Parent1	Row Child1	Row Child−Ch	52	62	72	82		
10	Column Paren	Row Super−P	Row Parent1	Row Child1	Row Child−Ch	53	63	73	83		
11	Column Paren	Row Super−P	Row Parent1	Row Child2	Row Child−Ch	54	64	74	84		
12	Column Paren	Row Super−P	Row Parent1	Row Child2	Row Child−Ch	55	65	75	85		
13	Column Paren	Row Super−P	Row Parent2	Row Child1	Row Child−Ch	57	67	77	87		
14	Column Paren	Row Super−P	Row Parent2	Row Child1	Row Child−Ch	58	68	78	88		
15	Column Paren	Row Super−P	Row Parent2	Row Child2	Row Child−Ch	59	69	79	89		

2 Code Overview

TableToLongForm is structured as follows.

8 ⟨*TableToLongForm.R* 8⟩≡
    ⟨*document header* 9a⟩
    ⟨*Front End* 9b⟩
    ⟨*Identification* 11a⟩
    ⟨*Discern Parentage* 16b⟩
    ⟨*Reconstruction* 22b⟩
    ⟨*Back End* 10⟩

This code is written to file `TableToLongForm.R`.

We place a document header at the top of the extracted code to encourage people to read the literate description rather than attempting to study the code alone.

9a ⟨*document header* 9a⟩≡

```
##-----------------------------------------------------------------------
## The code in this .R file is machine generated from the literate
##  program, TableToLongForm.Rnw
## Documentation can be found in the literate description for this
##  program, TableToLongForm.pdf
##-----------------------------------------------------------------------
```

2.1 Front End

The main function `TableToLongForm` is defined here. For most users this is the only function they will call. However, the majority of the supporting functions are not hidden and therefore can easily be viewed and/or modified by users.

9b ⟨*Front End* 9b⟩≡

```
TableToLongForm =
  function(datamat, IdentResult = NULL,
           fulloutput = FALSE, diag = FALSE, diagname = NULL){
    if(diag){
      if(is.null(diagname)) diagname = deparse(substitute(datamat))
      assign("TCRunout", file(paste0(diagname, ".TCRunout"),
                              "w"), envir = .GlobalEnv)
      on.exit({
        close(TCRunout)
        rm("TCRunout", envir = .GlobalEnv)
      })
    }

    fullout = ReconsMain(datamat, IdentResult)
    if(fulloutput) fullout else fullout$datafr
  }
```

2.2 Back End

Various code, mainly to help produce diagnostic output, can be ignored by most users.

print.plist A print method for class `plist`, which are nested lists with a numeric vector at the lowest level; `print.default` is rather inefficient in displaying such nested lists.

TCRsink Sinks the output to `TCRunout` for diagnostic output. Requires the existence of `TCRunout` which is created by the main function `TableToLongForm` when `diag = TRUE`.

Spaces may be introduced by `match.call`, thus any spaces in the args of *variables to sink* (that is, the arguments supplied via `...`) are removed without warning.

10 ⟨*Back End* 10⟩≡
```
print.plist = function(plist){
  plistC = function(plist){
    pLoc = attr(plist, "Loc")
    if(is.list(plist)){
      namevec = names(plist)
      if(!is.null(pLoc))
        namevec = paste0(names(plist),
          " (", pLoc[,"rows"], ", ", pLoc[,"cols"], ")")
      namelist = as.list(namevec)
      for(i in 1:length(namelist))
        namelist[[i]] =
          c(paste("+", namelist[[i]]),
            paste("-", plistC(plist[[i]])))
      do.call(c, namelist)
    } else{
      if(!is.null(names(plist))){
        namevec = names(plist)
        if(!is.null(pLoc))
          namevec = paste0(names(plist),
            " (", plist, ", ", pLoc[,"cols"], ")")
        paste("+", namevec)
      } else paste(plist, collapse = " ")
    }
    }
  cat(plistC(plist), sep = "\n")
}

attrLoc =
  function(plist, rows = NULL, cols = NULL){
    attr(plist, "Loc") = cbind(rows, cols)
    class(plist) = "plist"
    plist
  }

TCRsink =
  function(ID, ...)
  if(exists("TCRunout", envir = .GlobalEnv)){
    varlist = list(...)
    names(varlist) = gsub(" ", "", as.character(match.call()[-(1:2)]))
    sink(TCRunout)
    for(i in 1:length(varlist)){
      cat("###TCR", ID, names(varlist)[i], "\n")
      print(varlist[[i]])
    }
    sink()
```

```
  }
```

Defines:
`attrLoc`, used in chunks 18–22.
`TCRsink`, used in chunks 11–14 and 18–24.

3 Identification

We separate the Identification functions into two groups.

Ident Main contains the main function that is called by the *Front End* function.

Ident Low Level contains supporting functions called by the *Ident Main* function.

11a ⟨*Identification* 11a⟩≡
⟨*Ident Main* 11b⟩
⟨*Ident Low Level* 14d⟩

3.1 Identification - Main Function

The purpose of the `IdentMain` function is to identify where in the file the data is found and where the accompanying labels are, while ignoring any extraneous information we do not want. It should also identify the presence of multiple tables in the same file.

It is intended for this procedure to involve a number of Identification algorithms that are used for a high degree of reliability and flexibility, but at this stage there is only a single algorithm.

The algorithms used are:

- Ident by Most Common Boundary.

Algorithms planned for the near future are:

- Ident by Runs.

The output of `IdentMain` will be a list containing two elements, `rows` and `cols`, each of which is a list containing these two elements:

label - a vector of the rows or columns where the labels are found.

data - a vector of the rows or columns where the data are found.

11b ⟨*Ident Main* 11b⟩≡

```
IdentMain =
  function(datamat){
    ⟨Ident by Most Common Boundary 12a⟩
    ⟨Group Column Labels 13b⟩
    TCRsink("IM", rowslist, colslist)
    list(rows = rowslist, cols = colslist)
  }
```

Defines:
`IdentMain`, used in chunk 23b.
Uses `TCRsink` 10.

Example values for **ToyExComplete.csv**

```
> rowslist
$label
[1] 1 2 3 4

$data
 [1]  5  6  7  8  9 10 11 12 13 14


> colslist
$label
[1] 1 2

$data
$data[1]
[1] 4 5 6 7

$data[2]
[1]  8  9 10 11
```

3.1.1 Ident By Most Common Boundary

The `IdentMostCommonBoundary` Low Level function is used to find the most common start and end rows and columns (the boundary) to search for a block (rectangular region) of numbers, which is assumed to be our table of data.

12a ⟨*Ident by Most Common Boundary* 12a⟩≡

```
⟨Get Non empty rows and cols 12b⟩
⟨Call Ident MostCommonBoundary 12c⟩
⟨Construct rowslist and colslist 13a⟩
```

12b ⟨*Get Non empty rows and cols* 12b⟩≡

```
rowNonempty = (1:nrow(datamat))[IdentNonEmpty(datamat, 1)]
colNonempty = (1:ncol(datamat))[IdentNonEmpty(datamat, 2)]
```

Uses `IdentNonEmpty` 15a.

12c ⟨*Call Ident MostCommonBoundary* 12c⟩≡

```
rowData = IdentMostCommonBoundary(datamat, 2)
colData = IdentMostCommonBoundary(datamat, 1)
## Temporary fix for first col being all numbers (e.g. years)
if(colData[1] == 1) colData[1] = 2
TCRsink("CIMCB", rowData, colData)
```

Uses `IdentMostCommonBoundary` 16a and `TCRsink` 10.

Example values for **ToyExComplete.csv**

```
> rowData
[1]  5 14

> colData
[1]  4 11
```

We construct the interim `rowslist` taking every non-empty row before the most common start of the numbers block (`rowData[1]`) and assigning these to the `label` region. The numbers block (which is bounded by `rowData[1]` and `rowData[2]`) is assigned to the `data` region. The interim `colslist` is constructed in the same manner.

13a ⟨*Construct rowslist and colslist* 13a⟩≡

```
rowslist = list(label = rowNonempty[rowNonempty < rowData[1]],
                data = rowNonempty[(rowNonempty >= rowData[1]) &
                                   (rowNonempty <= rowData[2])])
colslist = list(label = colNonempty[colNonempty < colData[1]],
                data = colNonempty[(colNonempty >= colData[1]) &
                                   (colNonempty <= colData[2])])
TCRsink("CRAC", rowslist, colslist)
```

Uses `TCRsink` 10.

Example values for **ToyExComplete.csv**

```
> rowslist
$label
[1] 1 2 3 4

$data
 [1]  5  6  7  8  9 10 11 12 13 14


> colslist
$label
[1] 1 2

$data
[1]  4  5  6  7  8  9 10 11
```

3.1.2 Group Column Labels

We look for a repeating pattern in the column labels to handle cases of Misaligned Column Label (see Section 1.3.7).

13b ⟨*Group Column Labels* 13b⟩≡

⟨*Generate Pattern vector* 14a⟩
⟨*Take Largest Pattern* 14b⟩
⟨*Group by Pattern* 14c⟩

We loop through each row of the labels region and check for a pattern in either the contents of the cells (if they are all non-empty), or a pattern in which cells are empty (if any cells are empty), and store the results in `Patvec` (Pattern Vector).

14a ⟨*Generate Pattern vector* 14a⟩≡

```
curcol = colslist$data
Patvec = NULL
for(currow in rowslist$label){
  curlabel = datamat[currow, curcol]
  if(any(is.na(curlabel))){
    if(!all(is.na(curlabel)))
      Patvec = c(Patvec, IdentPattern(is.na(curlabel)))
  } else Patvec = c(Patvec, IdentPattern(curlabel))
}
TCRsink("GPV", Patvec)
```

Uses `IdentPattern` 15b and `TCRsink` 10.

Example values for **ToyExComplete.csv**

```
> Patvec
[1] 4 1 4
```

Where multiple patterns are found, we assume the shorter patterns are patterns of families that are children to the parents of the largest pattern. Thus we always take the largest pattern found.

Some problems with this and next chunk, refer to `BUG-ID 2`.

14b ⟨*Take Largest Pattern* 14b⟩≡

```
Patvec = max(Patvec)
```

If a pattern is found (`NA` = No patterns found), we group the columns into separate elements in a list and update the `colslist`. For easy handling later, irrespective of whether a pattern is found, `colslist$data` is a list.

14c ⟨*Group by Pattern* 14c⟩≡

```
if(!is.na(Patvec)){
  megacolnum = length(curcol)/Patvec
  megacollist = list()
  for(i in 1:megacolnum)
    megacollist =
      c(megacollist, list(curcol[1:Patvec + Patvec * (i - 1)]))
  colslist$data = megacollist
} else colslist$data = list(curcol)
```

3.2 Identification - Low Level Functions

Here we discuss the low level functions that are called by the main Identification function. Each chunk corresponds to a separate low level function.

14d ⟨*Ident Low Level* 14d⟩≡

⟨*Ident Non Empty* 15a⟩
⟨*Ident Pattern* 15b⟩
⟨*Ident Most Common Boundary* 16a⟩

3.2.1 IdentNonEmpty

Given a matrix (`datamat`) and a margin (1 for rows, 2 for columns), return a vector giving the indices of non-empty rows or columns. Can specify a different empty identifying function (default `is.na`). Procedure:

1. Compute `isnonempty`, a logical vector about whether the rows or cols are not empty.
2. Use `which` on `isnonempty` to get indices.

15a ⟨*Ident Non Empty* 15a⟩≡

```
IdentNonEmpty =
  function(datamat, margin, emptyident = is.na){
    isnonempty = apply(datamat, margin, function(x) !all(emptyident(x)))
    which(isnonempty)
  }
```

Defines:
`IdentNonEmpty`, used in chunks 12b and 23c.

3.2.2 IdentPattern

Attempt to discern a repeating pattern in `vec`, which can be a vector of any type (which is coerced to `character`). The returned value, `res` is either `NA` if no pattern is found. Or it is the grouping number for the repeating pattern, e.g.

- `vec` = 1 1 1 1, then `res` = 1
- `vec` = 3 4 3 4, then `res` = 2
- `vec` = 1 2 3 1, then `res` = `NA`

`IdentPattern` does this fairly efficiently by use of regular expressions. It combines the first `i` elements of `vec` and collapses this into a single string. A `grep` is then called on the entire `vec` that has been collapsed into a single string, checking to see if the entire string can be matched to some repeat of the aforementioned collapsed string of the first `i` elements.

For the moment it is possible for this to fail (and can even be intentionally gamed by providing something like `vec = c(12, 1, 2)`, which will return a pattern of 1, when it should return `NA`), so it should be changed to be more reliable (though less efficient).

15b ⟨*Ident Pattern* 15b⟩≡

```
IdentPattern =
  function(vec){
    len = length(vec)
    res = NA
    for(i in 1:floor(len/2)){
      curseg = paste("^(", paste(vec[1:i], collapse = ""),
        ")+$", sep = "")
      if(nchar(curseg) > 2559){
        warning("Label lengths too long for regular expressions to ",
                "work. IdentPattern has been aborted. A pattern may ",
                "exist but it cannot be found with the current ",
                "algorithm.")
        break
      } else if(length(grep(curseg, paste(vec, collapse = ""))) > 0){
        res = i
        break
      }
    }
    res
  }
```

Defines:
`IdentPattern`, used in chunk 14a.

3.2.3 Ident Most Common Boundary

Search for the most common first and last rows/cols to identify a block (rectangular region) of numbers. Procedure:

1. Suppose `margin = 2`, then loop through each column and search for cells containing numbers.
2. Compute the first row with a number for each column (`nstarts`), and do the same for the last row (`nends`).
3. Return the most common first and last rows.

16a ⟨*Ident Most Common Boundary* 16a⟩≡
```
IdentMostCommonBoundary =
  function(datamat, margin){
    isnumber = suppressWarnings(apply(datamat, margin,
      function(x) which(!is.na(as.numeric(x)))))
    nstarts = table(sapply(isnumber,
      function(x) if(length(x) > 0) min(x) else NA))
    nends = table(sapply(isnumber,
      function(x) if(length(x) > 0) max(x) else NA))
    as.numeric(names(c(which.max(nstarts), which.max(rev(nends)))))
  }
```
Defines:
`IdentMostCommonBoundary`, used in chunk 12c.

4 Discern Parentage

We separate the Parentage functions into three groups.

Pare Front is a simple 'front-end' function that makes the appropriate first call to `PareMain`, and is the function called by the *Front End* function.

Pare Col A specialised front-end to `PareFront` that handles various fringe cases for Discering Parentage for Column Labels, before eventually calling `PareFront`.

Pare Main contains the main function that recursively call itself until the all parentage is discerned.

Pare Low Level contains supporting functions called by the *Pare Main* function.

See the section on the main function (Section 4.2) for details on the purpose of the *Discern Parentage* stage.

16b ⟨*Discern Parentage* 16b⟩≡
⟨*Pare Front* 17a⟩
⟨*Pare Col* 17b⟩
⟨*Pare Main* 18c⟩
⟨*Pare Low Level* 20b⟩

4.0.4 plist

explanation.

4.1 Parentage - Front End Function

This front end function takes the `datamat` and constructs an initialising `plist` (Parentage List), which is used to make the first call to the main function.

17a ⟨*Pare Front* 17a⟩≡

```
PareFront =
  function(datamat)
  PareMain(datamat = datamat, plist =
           list(rows = 1:nrow(datamat), cols = 1:ncol(datamat)))
```

Defines:
`PareFront`, used in chunks 18b and 23c.
Uses `PareMain` 18c.

4.1.1 Pare Col

The Parentage functions were initially designed to work with Row Labels only, however we can also use them to discern the parentage of Col Labels once we handle a few differences. We define a front-end to the front-end function called `PareCol` to do this.

17b ⟨*Pare Col* 17b⟩≡

```
PareCol =
  function(datamat, datacols, labelrows){
    ⟨Case Misaligned Col Parent 17c⟩
    datacols = unlist(datacols)
    ⟨Collapse Fullrow Labels 18a⟩
  ⟨Call Pare Front 18b⟩
}
```

Defines:
`PareCol`, used in chunk 24.
Uses `datacols` 23b and `labelrows` 23b.

Unlike with Row Labels where the parents are reliably in the top-left corner of their family, Col Label parents are sometimes 'misaligned'. In some cases this arises as Col Label parents might be centred over their family. Other times, it happens for no apparent logical reason.

Regardless of the cause, we need to correct for this. During the Identification phase, we Grouped the Column Labels based on repeating patterns. We use these groupings to identify the families, and if the parent is not found where it should be, we simply shift it over to the right place.

17c ⟨*Case Misaligned Col Parent* 17c⟩≡

```
for(j in 1:length(datacols)){
  curfamily = datamat[labelrows, datacols[[j]], drop = FALSE]
  firstcolempty = is.na(curfamily[,1])
  if(any(firstcolempty))
    for(i in which(firstcolempty)){
      notempty = !is.na(curfamily[i,])
      if(sum(notempty) == 1){
        curfamily[i, 1] = curfamily[i, notempty]
        curfamily[i, notempty] = NA
      }
    }
  datamat[labelrows, datacols[[j]]] = curfamily
}
```

Uses `datacols` 23b and `labelrows` 23b.

It is also quite common for Col Labels that are too wide to be physically split over multiple rows to manage the width of the labels. For now, we simply assume that any rows that are not full (and hence not parents) should all really be a single row of children, and collapse these.

18a ⟨*Collapse Fullrow Labels* 18a⟩≡

```
notfullrows = apply(datamat[labelrows, datacols, drop = FALSE], 1,
  function(x) any(is.na(x)))
if(any(diff(notfullrows) > 1))
  warning("full rows followed by not full rows!")
pastestring = ""
pasterows = which(!notfullrows)
for(i in 1:length(pasterows))
  pastestring[i] = paste("datamat[labelrows[", pasterows[i],
               "], datacols]", sep = "")
collapsedlabels = eval(parse(text = paste("paste(",
                               paste(pastestring, collapse = ", "),
                               ")", sep = "")))
```

Uses `datacols` 23b and `labelrows` 23b.

Once the above is handled, we can simply transpose our Col Labels and call `PareFront` on it.

18b ⟨*Call Pare Front* 18b⟩≡

```
labeldatamat = rbind(datamat[labelrows[notfullrows], datacols],
  collapsedlabels)
PareFront(t(labeldatamat))
```

Uses `datacols` 23b, `labelrows` 23b, and `PareFront` 17a.

4.2 Parentage - Main Function

The purpose of the `PareMain` function is to identify (or *Discern*, to better differentiate this stage from the *Identification* stage) hierarchical relationships (the *Parentage*) in the data.

It first makes various checks for fringe cases, then calls various detection algorithms (`Pare Low Levels`) to discern the parentage.

18c ⟨*Pare Main* 18c⟩≡

```
PareMain =
  function(datamat, plist){
    ⟨If only one column 18d⟩
    ⟨If first column empty 19a⟩
    ⟨If only one row 19b⟩
    ⟨If first cell empty 19c⟩
    ⟨Otherwise call Pare Low Levels 20a⟩
    class(res) = "plist"
    res
  }
```

Defines:
`PareMain`, used in chunks 17a, 19a, and 20a.

If only one column is found then this means we are in the right-most column (or there was only one column to begin with), and hence the currently examined cells cannot be parents. We return the rows of these children as a vector, with names that correspond to their labels.

18d ⟨*If only one column* 18d⟩≡

```
if(length(plist$cols) == 1){
  res = structure(plist$rows, .Names = datamat[plist$rows, plist$cols])
  res = attrLoc(res, cols = plist$col)
  TCRsink("IOOC", plist, res)
}
```

Uses `attrLoc` 10 and `TCRsink` 10.

Example values for **ToyExComplete.csv**

```
> plist
$rows
[1] 3 4

$cols
[1] 2


> res
+ Row Child-Child1 (3, 2)
+ Row Child-Child2 (4, 2)
```

If the first column is found to be empty, then we will shift to the next column (which we know exists because we passed the check for only one column).

19a ⟨*If first column empty* 19a⟩≡

```
else if(all(is.na(datamat[plist$rows, plist$cols[1]]))){
  plist$cols = plist$cols[-1]
  res = PareMain(datamat, plist)
}
```

Uses `PareMain` 18c.

If only one row is found then our row is a parent to itself (we know there are children in the row as we passed the check for only one column). We return the row as a numeric vector, nested in a list using correct parentage and names of the parentage within the row.

19b ⟨*If only one row* 19b⟩≡

```
else if(length(plist$rows) == 1){
  res = structure(plist$rows,
    .Names = datamat[plist$rows, plist$cols[length(plist$cols)]])
  res = attrLoc(res, cols = plist$cols[length(plist$cols)])
  for(i in (length(plist$cols) - 1):1){
    res = list(res)
    names(res) = datamat[plist$rows, plist$cols[i]]
    res = attrLoc(res, rows = plist$rows, cols = plist$cols[i])
  }
  TCRsink("IOOR", plist, res)
}
```

Uses `attrLoc` 10 and `TCRsink` 10.

Example values for **ToyExComplete.csv**

```
> res
Never occurs
```

If the first cell is empty, after all previous checks, then this is an unrecognised format and we return a warning message.

19c ⟨*If first cell empty* 19c⟩≡

```
else if(is.na(datamat[plist$rows[1], plist$cols[1]])){
  warning("cell[1, 1] is empty")
  print(plist)
  res = NA
}
```

If we have passed all the checks, we can then call the Low Level `Pare` functions. We first call `ByEmptyRight` to check for *empty right* situations. If none are found, it returns `NA`, in which case we try `ByEmptyBelow` instead.

We then loop through each element of the returned list and call the main function, as per the recursive nature of the function.

20a ⟨*Otherwise call Pare Low Levels* 20a⟩≡
```
else{
  res = PareByEmptyRight(datamat, plist)
  if(any(is.na(res)))
    res = PareByEmptyBelow(datamat, plist)
  for(i in 1:length(res))
    res[[i]] = PareMain(datamat, res[[i]])
  res
}
```
Uses `PareByEmptyBelow` 22a, `PareByEmptyRight` 20c, and `PareMain` 18c.

4.3 Parentage - Low Level Functions

The Low Level Parentage functions are called by the Main Parentage function. In particular, `ByEmptyRight` is always called first. Then `ByEmptyBelow` is called on the results of the above.

20b ⟨*Pare Low Level* 20b⟩≡
```
⟨Pare By Empty Right 20c⟩
⟨Pare By Empty Below 22a⟩
```

4.3.1 Pare By Empty Right

We check to see if we have an *empty right* situation. If we do not, we return `NA`.

20c ⟨*Pare By Empty Right* 20c⟩≡
```
PareByEmptyRight =
  function(datamat, plist)
  with(plist,
       if(all(is.na(datamat[rows[1], cols[-1]]))){
         ⟨Check for Other Empty Rights 20d⟩
         ⟨Case Single Empty Right 20e⟩
         ⟨Case Multiple Empty Rights 21⟩
         res
       } else NA)
```
Defines:
`PareByEmptyRight`, used in chunk 20a.

20d ⟨*Check for Other Empty Rights* 20d⟩≡
```
emptyrights = apply(datamat[rows, cols[-1], drop = FALSE], 1,
  function(x) all(is.na(x)))
rowemptyright = rows[emptyrights]
```

In the case of only a single empty right, we know there is only a single parent, which is the first line. Thus we take everything except the first line (which will be the rows of the children of this parent) and pass this through with correct naming.

20e ⟨*Case Single Empty Right* 20e⟩≡
```
if(length(rowemptyright) == 1){
  res = list(list(rows = rows[-1], cols = cols))
  names(res) = datamat[rows[1], cols[1]]
  res = attrLoc(res, rows = rows[1], cols = cols[1])
  TCRsink("CSER", res)
}
```
Uses `attrLoc` 10 and `TCRsink` 10.

1	***New Zealand***	
2	**Auckland**	
3	Accounting	Male
4		Female
5	Economics	Male
6		Female
7	Statistics	Male
8		Female
9	**Wellington**	
10	Economics	Male
11		Female
12	Statistics	Male
13		Female
14	***Australia***	
15	**Sydney**	
16	Accounting	Male
17		Female
18	Economics	Male
19		Female

Consider the toy example on the left.
In this case we do not have a simple `ByEmptyRight` structure. We have *super-parents* in the form of countries (New Zealand and Australia), and also *parents* in the form of cities (Auckland, Wellington and Sydney). To handle situations such as this, we must **Check for Other Empty Rights**.
If only a **Single Empty Right** is found, the situation is simple and we simply pass on the children of the single parent for the next iteration of `PareMain`.
However, if **Multiple Empty Rights** are found, we must identify the super-parents, and pass on the *children* of these super-parents (which would, in turn, contain parents and their children) as a list, to be handled in the next iteration of `PareMain`. In this example, we would have a list of length 2. The first element of the list would contain the `plist` with `rows` 2 to 13 (corresponding to the children of the New Zealand super-parent). The second element would have `rows` 15 to 19.

Example values for **ToyExComplete.csv**

```
> res
Never occurs
```

In the case of multiple empty rights, we first call `diff` to compute the gap in rows between the empty rights. If the value of `rowdiff[i]` is 1, this means there is no gap between the i^{th} `rowemptyright` and the (i + 1) `rowemptyright`. This happens with *super-parents* as described in the example above. In this case, we gather these super-parents and ignore all other `rowemptyright` (the parents inside the super-parents will be handled at the next iteration of `PareMain`). Note, we assume there are never any super-super-parents (i.e. we can only handle a maximum of 2-levels of parentage in the same column).

Whether or not super-parents were identified, we compute the rows for the children of each parent (or super-parent) identified by `rowemptyright` and pass this through as a list, with correct naming.

21 ⟨*Case Multiple Empty Rights* 21⟩≡

```
else{
  rowdiff = diff(rowemptyright)
  if(any(rowdiff == 1))
    rowemptyright = rowemptyright[c(rowdiff == 1, FALSE)]

  rowstart = pmin(rowemptyright + 1, max(rows))
  rowend = c(pmax(rowemptyright[-1] - 1, min(rows)), max(rows))

  res = list()
  for(i in 1:length(rowstart))
    res[i] = list(list(rows = rowstart[i]:rowend[i], cols = cols))
  names(res) = datamat[rowemptyright, cols[1]]
  res = attrLoc(res, rows = rowemptyright, cols = cols[1])
  TCRsink("CMER", res)
}
```

Uses `attrLoc` 10 and `TCRsink` 10.

Example values for **ToyExComplete.csv**

```
> res
+ Row Super-Parent (1, 1)
- + rows
- - 2 3 4 5 6 7 8 9 10
- + cols
- - 1 2
```

4.3.2 Pare By Empty Below

We check which cells are empty below (there should be at least 1 based on previous checks). Based on this, we compute the rows for the children of each parent and pass this through as a list, with correct naming.

22a ⟨*Pare By Empty Below* 22a⟩≡

```
ParеByEmptyBelow =
  function(datamat, plist)
  with(plist, {
    emptybelow = is.na(datamat[rows, cols[1]])
    rowstart = rows[!emptybelow]
    rowend = c(rowstart[-1] - 1, max(rows))
    res = list()
    for(i in 1:length(rowstart))
      res[i] = list(list(rows = rowstart[i]:rowend[i], cols = cols[-1]))
    names(res) = datamat[rowstart, cols[1]]
    res = attrLoc(res, rows = rowstart, cols = cols[1])
    TCRsink("PBEB", res)
    res
  })
```

Defines:
`ParеByEmptyBelow`, used in chunk 20a.
Uses `attrLoc` 10 and `TCRsink` 10.

Example values for **ToyExComplete.csv**

```
> res
+ Row Child1 (3, 1)
- + rows
- - 3 4
- + cols
- - 2
+ Row Child2 (5, 1)
- + rows
- - 5 6
- + cols
- - 2
```

5 Reconstruction

We separate the Reconstruction functions into two groups.

Recons Main contains the main function that is called by the *Front End* function.

Recons Low Level contains supporting functions called by the *Recons Main* function.

22b ⟨*Reconstruction* 22b⟩≡
⟨*Recons Main* 23a⟩
⟨*Recons Low Level* 25a⟩

5.1 Reconstruction - Main Function

The `ReconsMain` function is, in a manner of speaking, the true `TableToLongForm` function, as it makes the calls to `IdentMain` and `PareFront`, in conjunction with its own `Recons Low Level` functions, to carry out the conversion.

23a ⟨*Recons Main* 23a⟩≡

```
ReconsMain =
  function(datamat, IdentResult){
    ⟨Call Ident Main 23b⟩
    ⟨Reconstruct Row Labels 23c⟩
    ⟨Reconstruct Col Labels 24⟩
  }
```

Call `IdentMain` and assign them meaningful names for convenience, the `labelcols` being the columns where the labels can be found, etc. These should all be a `vector`, except `datacols` which is a `list`.

23b ⟨*Call Ident Main* 23b⟩≡

```
if(is.null(IdentResult))
  IdentResult = IdentMain(datamat)
labelcols = IdentResult$cols$label
datacols = IdentResult$cols$data
labelrows = IdentResult$rows$label
datarows = IdentResult$rows$data
```

Defines:
`labelcols`, used in chunk 23c.
`datacols`, used in chunks 17, 18, and 24.
`labelrows`, used in chunks 17, 18, and 24.
`datarows`, used in chunks 23c and 24.
Uses `IdentMain` 11b.

We create a subset of `datamat` that contains just the Row Labels. We also remove any columns that are completely empty (N.B. this may no longer be necessary, need to do some testing).

We call `PareFront` on our subset to discern the parentage of the Row Labels. We then use this to reconstruct the portion of the LongForm Dataframe relating to the Row Labels and assign this to `rowvecs`.

23c ⟨*Reconstruct Row Labels* 23c⟩≡

```
datamatRowLabels = datamat[datarows, labelcols, drop = FALSE]
datamatRowLabels = datamatRowLabels[,
  IdentNonEmpty(datamatRowLabels, 2), drop = FALSE]
rowplist = PareFront(datamatRowLabels)
rowvecs = ReconsRowLabels(rowplist)
TCRsink("RRL", rowplist, rowvecs[1:4,])
```

Defines:
`rowplist`, used in chunk 24.
`rowvecs`, used in chunks 24–27.
Uses `datarows` 23b, `IdentNonEmpty` 15a, `labelcols` 23b, `PareFront` 17a, `ReconsRowLabels` 25b, and `TCRsink` 10.

Example values for **ToyExComplete.csv**

```
> rowplist
+ Row Super-Parent (1, 1)
- + Row Parent1 (2, 1)
- - + Row Child1 (3, 1)
- - - + Row Child-Child1 (3, 2)
- - - + Row Child-Child2 (4, 2)
- - + Row Child2 (5, 1)
- - - + Row Child-Child1 (5, 2)
- - - + Row Child-Child2 (6, 2)
- + Row Parent2 (7, 1)
- - + Row Child1 (8, 1)
- - - + Row Child-Child1 (8, 2)
- - - + Row Child-Child2 (9, 2)
- - + Row Child2 (10, 1)
- - - + Row Child-Child2 (10, 2)

> rowvecs[1:4,]
 [,1]               [,2]          [,3]         [,4]
 "Row Super-Parent" "Row Parent1" "Row Child1" "Row Child-Child1"
 "Row Super-Parent" "Row Parent1" "Row Child1" "Row Child-Child2"
 "Row Super-Parent" "Row Parent1" "Row Child2" "Row Child-Child1"
 "Row Super-Parent" "Row Parent1" "Row Child2" "Row Child-Child2"
```

Due to the various fringe cases that exist with Col Labels, we cannot simply call `PareFront` and must instead call `PareCol`. We then create a subset of `datamat` that contains just the Col Labels and call `ReconsColLabels` which in truth reconstruct the entire LongForm Dataframe by making use of the `rowvecs` generated above.

24 ⟨*Reconstruct Col Labels* 24⟩≡

```
colplist = PareCol(datamat, datacols, labelrows)
datamatColLabels = datamat[datarows[unlist(rowplist)], unlist(datacols)]
res = ReconsColLabels(colplist, datamatColLabels, rowvecs)
TCRsink("RCL", colplist, res[1:4,])
list(datafr = res, datamat = datamat, IdentResult = IdentResult,
     rowplist = rowplist, colplist = colplist)
```

Uses `datacols` 23b, `datarows` 23b, `labelrows` 23b, `PareCol` 17b, `ReconsColLabels` 26a, `rowplist` 23c, `rowvecs` 23c, and `TCRsink` 10.

Example values for **ToyExComplete.csv**

```
> colplist
+ Column Parent1 (1, 2)
- + Column Child1 (1, 3)
- + Column Child2 (2, 3)
- + Column Child3 (3, 3)
- + Column Child4 (4, 3)
+ Column Parent2 (5, 2)
- + Column Child1 (5, 3)
- + Column Child2 (6, 3)
- + Column Child3 (7, 3)
- + Column Child4 (8, 3)

> res[1:4,]
         UNKNOWN          UNKNOWN      UNKNOWN    UNKNOWN          UNKNOWN
1 Column Parent1 Row Super-Parent Row Parent1 Row Child1 Row Child-Child1
2 Column Parent1 Row Super-Parent Row Parent1 Row Child1 Row Child-Child2
3 Column Parent1 Row Super-Parent Row Parent1 Row Child2 Row Child-Child1
4 Column Parent1 Row Super-Parent Row Parent1 Row Child2 Row Child-Child2
  Column Child1 Column Child2 Column Child3 Column Child4
1            12            22            32            42
2            13            23            33            43
3            14            24            34            44
4            15            25            35            45
```

5.2 Reconstruction - Low Level Functions

The Low Level Reconstruction functions are called by the Main Reconstruction function. In particular, `ReconsRowLabels` is always called first and its results are one of the arguments for `ReconsColLabels`, which finishes the reconstruction of the entire LongForm Dataframe.

25a ⟨*Recons Low Level* 25a⟩≡
⟨*Recons Row Labels* 25b⟩
⟨*Recons Column Labels* 26a⟩

5.2.1 Reconstruction - Row Labels

`ReconsRowLabels` iterates down the row parentage list (`plist`) recursively, extracting the names and using this to construct the columns of the finished LongForm Dataframe corresponding to the row labels. The final output is what was shown in the *Reconstruct Row Labels* chunk above as `rowvecs[1:4,]`.

25b ⟨*Recons Row Labels* 25b⟩≡

```
ReconsRowLabels =
  function(plist)
  if(is.list(plist)){
    rowvecs = as.list(names(plist))
    for(i in 1:length(rowvecs))
      rowvecs[[i]] = cbind(rowvecs[[i]], ReconsRowLabels(plist[[i]]))
    do.call(rbind, rowvecs)
  } else as.matrix(names(plist))
```

Defines:
`ReconsRowLabels`, used in chunk 23c.
Uses `rowvecs` 23c.

Example values for **ToyExComplete.csv**

```
> rowvecs[1:4,]
 [,1]               [,2]          [,3]         [,4]
 "Row Super-Parent" "Row Parent1" "Row Child1" "Row Child-Child1"
 "Row Super-Parent" "Row Parent1" "Row Child1" "Row Child-Child2"
 "Row Super-Parent" "Row Parent1" "Row Child2" "Row Child-Child1"
 "Row Super-Parent" "Row Parent1" "Row Child2" "Row Child-Child2"
```

5.2.2 Reconstruction - Column Labels

As with the row labels, `ReconsColLabels` iterates down the column parentage list (`plist`) recursively. We also need to handle the parents differently from the lowest level child. The final output is what was shown in the *Reconstruct Col Labels* chunk above as `res[1:4,]`.

26a ⟨*Recons Column Labels* 26a⟩≡

```
ReconsColLabels =
  function(plist, datamat, rowvecs){
    ⟨Recons Col Parents 26b⟩
    ⟨Recons Col Children 27⟩
    datfr
  }
```

Defines:
`ReconsColLabels`, used in chunks 24 and 26b.
Uses `rowvecs` 23c.

Example values for **ToyExComplete.csv**

```
> res[1:4,]
         UNKNOWN          UNKNOWN     UNKNOWN    UNKNOWN          UNKNOWN
1 Column Parent1 Row Super-Parent Row Parent1 Row Child1 Row Child-Child1
2 Column Parent1 Row Super-Parent Row Parent1 Row Child1 Row Child-Child2
3 Column Parent1 Row Super-Parent Row Parent1 Row Child2 Row Child-Child1
4 Column Parent1 Row Super-Parent Row Parent1 Row Child2 Row Child-Child2
  Column Child1 Column Child2 Column Child3 Column Child4
1            12            22            32            42
2            13            23            33            43
3            14            24            34            44
4            15            25            35            45
```

Any parents are used to construct additional columns of factors (the labels of the parents) for the LongForm Dataframe, which is attached to the portion previously constructed in `ReconsRowLabels`.

26b ⟨*Recons Col Parents* 26b⟩≡

```
if(is.list(plist)){
  colvecs = as.list(names(plist))
  for(i in 1:length(colvecs)){
    colvecs[[i]] = cbind(colvecs[[i]],
              ReconsColLabels(plist[[i]], datamat, rowvecs))
    colnames(colvecs[[i]])[1] = "UNKNOWN"
  }
  datfr = do.call(rbind, colvecs)
}
```

Uses `ReconsColLabels` 26a and `rowvecs` 23c.

For the lowest level child, we extract the relevant ‘data bits’ from the original table and bind it to our Dataframe, using the lowest level child as the labels of these columns of data values.

27 ⟨*Recons Col Children* 27⟩≡

```
else{
  datbit = datamat[,plist]
  mode(datbit) = "numeric"
  ## Specify row.names to avoid annoying warnings
  datfr =
    cbind(as.data.frame(rowvecs, row.names = 1:nrow(rowvecs)), datbit)
  colnames(datfr) =
    c(rep("UNKNOWN", length = ncol(rowvecs)), names(plist))
}
```

Uses `rowvecs` 23c.

6 Chunk Index

7 Identifier Index

Numbers indicate the chunks in which the function appears. Underline indicates the chunk where the function is defined.

References

Davies, T., Bawa, Z., 2012a. The Journal of Community Informatics 8 (2).
URL http://ci-journal.net/index.php/ciej/issue/view/41

Davies, T., Bawa, Z., 2012b. The Promises and Perils of Open Government Data (OGD). The Journal of Community Informatics 8 (2).
URL http://ci-journal.net/index.php/ciej/article/view/929

New Zealand Qualifications Authority, 2012. Secondary school statistics.
URL http://www.nzqa.govt.nz/studying-in-new-zealand/secondary-school-and-ncea/secondary-schoo

R Core Team, 2012. R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria, ISBN 3-900051-07-0.
URL http://www.R-project.org/

Ramsey, N., Sept 1994. Literate programming simplified. IEEE Software 11 (5), 97–105.
URL http://www.cs.tufts.edu/~nr/noweb/

Statistics New Zealand, 2013. Infoshare.
URL http://www.stats.govt.nz/infoshare/