UNIVERSITY OF JOENSUU
COMPUTER SCIENCE
DISSERTATIONS 5

MARKKU TUKIAINEN

DEVELOPING A NEW MODEL OF SPREADSHEET CALCULATION: A GOALS AND PLANS APPROACH

ACADEMIC DISSERTATION

To be presented, with the permission of the Faculty of Science of the University of Joensuu, for public criticism in Auditorium Louhela of the Carelian Science Park, Länsikatu 15, Joensuu, on June 29th, 2001, at 12 noon.

UNIVERSITY OF JOENSUU
2001

Julkaisija Publisher	Joensuun yliopisto University of Joensuu
Toimittaja Editor	Jussi Parkkinen
Vaihto	Joensuun yliopiston kirjasto, vaihdot PL 107, 80101 JOENSUU Puh. 013-251 2677, fax. 013-251 2691
Exchanges	Joensuu University Library, Exchanges P.O.Box 107, FIN-80101 JOENSUU, Finland Tel. +358-13-251 2677, Fax. +358-13-251 2691 Email: Riitta.Porkka@joensuu.fi
Myynti	Joensuun yliopiston kirjasto, julkaisujen myynti PL 107, 80101 JOENSUU Puh. 013-251 2652, 251 2662, fax. 013-251 2691 Email: Armi.Lavikainen@joensuu.fi
Sale	Joensuu University Library, sale of publications P.O.Box 107, FIN-80101 JOENSUU Tel. +358-13-251 2652, fax +358-13-251 2691 Email: Armi.Lavikainen@joensuu.fi

ISSN 1238-6944

ISBN 952-458-026-8

UDK 681.3.02

Computing Reviews (1998) Classification: H.1.2, H.4.1, I.2.4

Yliopistopaino

Joensuu 2001

DEVELOPING A NEW MODEL OF SPREADSHEET CALCULATION: A GOALS AND PLANS APPROACH

Markku Tukiainen

Department of Computer Science
University of Joensuu
P.O.Box 111, FIN-80101 Joensuu, Finland
Markku.Tukiainen@cs.joensuu.fi

University of Joensuu. Computer Science, Dissertations 5
Joensuu, June 2001, 151 pages
ISSN 1238-6944, ISBN 952-458-026-8

Keywords: spreadsheet calculation, goals and plans, empirical studies

This thesis investigates a new approach to spreadsheet calculation. This is done through the schema-based theory of programming knowledge and an empirical investigation of spreadsheet calculation. More specifically, the thesis presents a model of spreadsheet programming knowledge, a tool that utilizes the model, and an empirical comparison of this model with the traditional model of spreadsheet calculation.

The thesis starts with an introduction to the use and the nature of spreadsheet calculation. The current spreadsheet model, the development of applications with the current model, and systems utilizing the model are described. After this, the schema-based goals-and-plans concept of programming is presented in enough detail needed for understanding the spreadsheet studies presented. This introduction is followed by four papers. Together, these papers present a theory of spreadsheet calculation knowledge representation, a model of spreadsheet calculation, support for spreadsheet calculation tool design, and a report of testing the new model empirically.

The first paper describes the theoretical background of this research and develops a model, the DGP/S model, for spreadsheet knowledge. The second paper presents BASSET, a spreadsheet system based on this model. The third paper presents an empirical study comparing the traditional spreadsheet calculation paradigm with the approach developed in this thesis. The fourth paper considers some of the criticism of the goals-and-plans approach of programming knowledge. It describes the main points of the criticism and argues that this criticism seems to be partially based on some misconceptions of the goals-and-plans approach. The thesis also includes an appendix that describes the spreadsheet goals and plans which we found in our analysis of spreadsheet applications.

Acknowledgements

This thesis is the result of research carried out at the Department of Computer Science of the University of Joensuu during the years 1989-1999.

I am deeply grateful to my supervisor, Professor Jorma Sajaniemi, for his invaluable advice and comments. For their enthusiasm for spreadsheet calculation, I am grateful to the members of the spreadsheet research groups, Pertti Saariluoma, Kari Hassinen and Jarmo Väisänen in particular.

The design and implementation of the BASSET spreadsheet system was a long project. Of the many people involved in the process, I gratefully mention Jorma Sajaniemi, Kari Hassinen, Jarmo Väisänen, Pasi Rikala, Tapani Reijonen, Juha Salonen, Petri Laitinen and Juhani Rautiainen.

Professors Kari Kuutti and Kari-Jouko Räihä kindly accepted the role of a reviewer. I wish to thank for their time and efforts.

My work was financially supported by the Department of Computer Science and the Faculty of Science at the University of Joensuu.

During the whole time of my work at the University, I have had many friends and colleagues with whom I have shared the joys and the sorrows. Thank you for everything.

I warmly thank my wife Reetta for sharing her life with me, being my best critique during my career and for giving us two beautiful children, Tapio and Kerttu.

Joensuu, June 6th 2001, SDG.

Table of Contents

INTRODUCTION 1

1.1 OVERVIEW OF THE THESIS 1
1.1.1 Motivation 2
1.1.2 The objective of the study 3
1.1.3 The structure of the thesis 4
1.1.4 The main contributions of the thesis 5
1.2 SPREADSHEET CALCULATION 5
1.2.1 End-user programming 6
1.2.2 The current spreadsheet model 7
1.2.3 Applications of the current spreadsheet model 9
1.2.4 Evaluation of the current spreadsheet model 11
1.2.5 Current spreadsheet systems 13
1.3 GOALS AND PLANS IN PROGRAMMING 15
1.3.1 Human memory and expertise 15
1.3.2 Programming goals and plans 17
1.3.3 Plans and programming expertise 22
1.3.4 The creation and use of plans 24
1.3.5 Related developments to goals-and-plans approach 25
1.4 GOALS AND PLANS IN SPREADSHEET CALCULATION 27

LIST OF ORIGINAL PUBLICATIONS INCLUDED IN THIS THESIS:

2. GOALS AND PLANS IN SPREADSHEET CALCULATION
Report A-1999-1, Department of Computer Science, University of Joensuu, 1999.

3. ASSET: A STRUCTURED SPREADSHEET CALCULATION SYSTEM
Machine-Mediated Learning 5(2), 63-76, Lawrence Erlbaum Associates, 1996.

4. COMPARING TWO SPREADSHEET CALCULATION PARADIGMS: AN EMPIRICAL STUDY WITH NOVICE USERS
Interacting with Computers 13(4), 427-446, Elsevier Science B.V., 2001.

5. GOALS AND PLANS IN SPREADSHEETS AND OTHER PROGRAMMING TOOLS
Proceedings of the 8th Annual Workshop of the Psychology of Programming Interest Group, April 10-12, Ghent, Belgium, pp. 114-122, 1996.

APPENDIX A: SPREADSHEET GOAL AND PLAN CATALOG: ADDITIVE AND MULTIPLICATIVE COMPUTATIONAL GOALS AND PLANS IN SPREADSHEET CALCULATION
Report A-1996-4, Department of Computer Science, University of Joensuu, 1996.

The publications are reprinted with kind permissions from the publishers.

Introduction

This thesis investigates a new approach to spreadsheet calculation. This is done through the schema-based theory of programming knowledge and an empirical investigation of spreadsheet calculation. More specifically, the thesis presents a model of spreadsheet programming knowledge, a tool that utilizes the model, and an empirical comparison of this model with the traditional model of spreadsheet calculation.

1.1 Overview of the thesis

The approach presented in this thesis is based on the spreadsheet calculation research carried on at the department of Computer Science of the University of Joensuu during the years 1988-2000. The results of this research have been published in a series of publications (Hassinen et al., 1988, Saariluoma and Sajaniemi, 1989, Sajaniemi, 1989, Sajaniemi and Pekkanen, 1988, Sajaniemi and Tukiainen, 1996, Sajaniemi et al., 1999, Sajaniemi et al., 1992, Tukiainen, 1996, Tukiainen, 2001, Tukiainen and Sajaniemi, 1996). This thesis includes the publications "*Goals and Plans in Spreadsheets and Other Programming Tools*" (Sajaniemi and Tukiainen, 1996), "*Spreadsheet Goal and Plan Catalog: Additive and Multiplicative Computational Goals and Plans in Spreadsheet Calculation*" (Tukiainen and Sajaniemi, 1996), "*ASSET: A Structured Spreadsheet Calculation System*" (Tukiainen, 1996), "*Goals and Plans in Spreadsheet Calculation*" (Sajaniemi et al., 1999) and "*Comparing two spreadsheet calculation paradigms: An empirical study with novice users*" (Tukiainen, 2001) in full.

A major part of the research behind this thesis was done in co-operation with Professor Jorma Sajaniemi during the years 1989-2000. The other researchers participating in the spreadsheet calculation research group were Kari Hassinen (1988-1990, University of Joensuu, Department of Computer Science), Professor Pertti Saariluoma (1988-1994, University of Helsinki, Department of Psychology) and Jarmo Väisänen (1988-1989, University of Joensuu, Department of Computer Science). The research was partially supported by the Academy of Finland and the Faculty of Sciences of the University of Joensuu.

1.1.1 Motivation

"Spreadsheets make it easy to do complex calculation – and even easier to do them incorrectly"
(Richard Scoville, PC-Bible, Chapter 15: Spreadsheets, p. 403)

Spreadsheets are used daily by millions of users, which probably makes spreadsheet programs the second most used type of application program, next to word processing. Some user studies, e.g. the 1993 study by McLean et al. among American grocery manufacturers, have found spreadsheets the most utilized and significant application for their subjects. But despite the wide use of spreadsheet systems, research literature in the field is scarce (Hendry and Green, 1994). Another conspicuous feature of the literature is that some of the same research questions have led to dramatically varying results. For example, in their studies of the use of spreadsheet formulae, Brown and Gould (1987) found that as many as 44 % of the spreadsheets created by experienced users contained programming errors, whereas Nardi and Miller (1990, 1991) suggest that there are no difficulties in using spreadsheet formula language.

Spreadsheet calculation has been called the success story of making programming easier (Lewis and Olson, 1987). Among its merits, the accessible two-dimensional layout with task specific aggregate operations and simple control structures without programming 'games', such as a need to declare variables, have been mentioned many times by advocates of spreadsheet calculation. Yet a remarkable portion of the professionally audited spreadsheet applications have contained errors. Panko (2000b) reports that in a collection of seven field audits involving 367 real world spreadsheets, 24 % of the spreadsheets contained errors. In audits carried out since 1997, as many as 91 % of the 54 spreadsheets audited contained errors.

The strengths of the traditional spreadsheets have been attributed to their familiar and concrete representation of data values and formulae, suppression of the inner world of traditional programming, automatic consistency maintenance, absence of control model, low viscosity, aggregate operations, and immediate feedback. These virtues have been described in many publications concerned with the spreadsheet model (see for example Nardi and Miller, 1990 or Napier et al., 1989). But the limitations of the spreadsheet model, i.e., the difficulty of debugging or redesign and the lack of modularity and abstraction, have been mentioned less often.

People do make errors when using current spreadsheet systems. Even in relatively simple calculations in the domain areas which are the most suitable for the spreadsheet model of computation, e.g. business and administrative computing, users make a noticeable number of errors (Butler, 2000, Panko, 2000a). One cause for these errors could be the lack of support

for structuring spreadsheet applications within the systems. The current spreadsheet systems do not contain adequate tools for structuring data for computations and creating abstractions for further use in the computations. The only means of creating abstractions is the naming of cell ranges and formulae. They relieve the user from the burden of changing multiple references to an area or updating many instances of the same formula. But the maintenance of these abstractions in cases where changes to the model are needed is left solely for the user to do. The systems do not offer much assistance in such situations.

Spreadsheet users make many kinds of errors. These can be grouped under at least three different categories. First, there are mechanical errors, often called slips. These include simple typing errors and errors in giving the right reference for a formula. Such errors can usually be found through a thorough investigation of the spreadsheet application. Second, there are logical errors. These are errors in problem domain understanding, such as a failure of the spreadsheet application to specify the relationships in the problem domain correctly. The identification of such errors is more difficult and requires assistance from other domain knowledgeable users. The third category is omission errors, which manifest themselves as missing parts in the spreadsheet model. These are the most difficult to detect (Panko, 2000b). One common problem in current spreadsheet systems is the updating of existing applications. When the user updates a part of the application he/she can easily omit the necessary updates in the other parts.

1.1.2 The objective of the study

In the previous section, the following issues were identified to cause problems in spreadsheet calculation:

- lack of modularity
- lack of abstraction
- difficulty of debugging
- difficulty of redesign

In this study we wanted to develop better data structuring and abstraction mechanisms for the spreadsheet model. These were expected to remedy its lack of modularity and abstraction. We also wanted to make the connection between the data and the computation in the spreadsheet model more transparent. This was expected to help the users in debugging and redesigning their spreadsheet applications.

This thesis explores spreadsheet calculation at the level of spreadsheet formula language, leaving out all other aspects of current spreadsheet systems, such as their graph making or database functions. In this sense, spreadsheet formula language represents a spreadsheet calculation paradigm for us, i.e., we will present a detailed description of the repertoires of

current spreadsheet systems in terms of spreadsheet formula language. It is also to be noted that spreadsheet formula language does not include any macro language intended for automating recurring tasks, such as creating weekly reports, or customizing the spreadsheet environment, such as creating input dialogs for the spreadsheet application. The macro languages of current spreadsheet systems vary a great deal, ranging from full function general programming languages to recording systems of user operations. There has been no thorough empirical research on macro language usability, but the comments presented in public range from “unclear syntax”, “confusing use of reference”, “don't understand what is happening” to a simple “AARGGH” (Gray et al., 1992). We left the macro languages out of this study because we wanted to concentrate on the structuring of the data and its connections to the computation at the level of the formulae.

The published empirical studies of programming include many studies of different programming languages, mainly procedural paradigm languages such as Pascal, but only a few studies pertaining to spreadsheets or any other end-user programming languages (Gray et al., 1992). This thesis is an attempt to contribute to the understanding of spreadsheet formula language by suggesting modifications to the existing model of spreadsheet calculation. In other words, we suggest a new approach to connecting the data and the computations in the spreadsheet so as to include automatic maintenance of the integrity of the computations. This thesis also attempts to contribute to the understanding of spreadsheet calculation by presenting a new model of spreadsheet knowledge and by using this model to analyze a set of real life spreadsheet applications. This analysis produces a set of task specific operations at a level different from that of the current spreadsheet systems.

1.1.3 The structure of the thesis

This thesis starts with an introduction to the use and the nature of spreadsheet calculation. The current spreadsheet model, the development of applications with the current model, and systems utilizing the model are described. In the next section the goals-and-plans concept of programming is presented in enough detail needed for understanding the spreadsheet studies presented. Section 1.4 briefly describes the contents of the papers included in this thesis. This introduction is followed by the four papers. Together, these papers present a theory of spreadsheet calculation knowledge representation, a model of spreadsheet calculation, support for spreadsheet calculation tool design, and a report of testing the new model empirically.

The first paper describes the theoretical background of this research and develops a model, the DGP/S model, for spreadsheet knowledge. The second paper presents BASSET, a spreadsheet system based on this model. The third paper presents an empirical study comparing the traditional spreadsheet calculation paradigm with the approach developed in this thesis. The fourth paper considers some of the criticism of the goals-and-plans approach

of programming knowledge. It describes the main points of the criticism and argues that this criticism seems to be partially based on some misconceptions of the goals-and-plans approach. The thesis also includes an appendix that describes the spreadsheet goals and plans which we found in our analysis of spreadsheet applications.

1.1.4 The main contributions of the thesis

The main contributions of this work can be summarized as:

- An empirical analysis of goal and plan structure of real life spreadsheet applications
- A development of a new model of spreadsheet calculation
- A development of a tool based on the new model of spreadsheet calculation
- An empirical analysis of the new model

1.2 Spreadsheet calculation

Spreadsheet calculation provides an efficient way to enter a set of numbers, perform certain computations, and display the results. For the knowledgeable end-user, a typical spreadsheet system also offers a means of constructing new computer applications. It is reasonable to argue that spreadsheet calculation has lowered the threshold to some level of programming for many users (Lewis and Olson, 1987). In this section we examine spreadsheet calculation as an end-user programming environment and describe the current spreadsheet model and its applications. We conclude with an evaluation of the current model and with some examples of systems implementing the model.

In the following, the term *spreadsheet model* refers to an overall structure of spreadsheet calculation, i.e., the way it is implemented as a calculation tool and the way the users interact with the model. *Spreadsheet application* refers to a concrete application which a user has implemented within the spreadsheet system, e.g. a company budget. *Spreadsheet system* refers to a concrete spreadsheet program such as Microsoft Excel or Lotus 1-2-3.

1.2.1 End-user programming

Spreadsheet application development is characterized as a typical end-user programming activity (Gray et al., 1992). An end-user is usually defined as an intelligent, task oriented non-programmer, such as an accountant or a working scientist, who has computational needs and wants to make serious use of computers (Lewis and Olson, 1987).

End-user programming can be defined by its objective: creating an application that serves some purpose of the user (Nardi, 1993). In the light of this definition, a whole repertoire of methods to adapt applications can be included in the domain of end-user programming, such as visual simulation environments (Repenning and Sumner, 1995), programming by example systems (Cypher, 1993), and authoring systems (Nardi, 1993). In this sense, actions which seem to be quite far removed from traditional programming are regarded as end-user programming, even such actions as form filling dialogues for setting preferences for applications or defining styles in word processing.

The problem of using general purpose programming languages in end-user programming is that end-users do not usually have the time or the interest to learn the language. General purpose languages are designed to be expressive, so that they can be applied to a great variety of problems. They therefore include low level primitives, from which the professional programmer can create higher level abstractions for use in the further development of the programs. But for end-users the low level primitives do not offer any help towards expressing their domain related problems in the programming language. It is commonly accepted, then, that general purpose programming languages are not the solution to the programming needs of the end-user (Goodell and Traynor, 1997).

In end-user programming systems, the expressiveness of general purpose programming languages is traded for ease of use, which is achieved by means of a variety of metaphors, or programming conventions (Goodell and Traynor, 1997). In this sense, a metaphor means a system for specifying computations, such as a sequence of actions in an application or the use of a task specific language. The possible metaphors vary, however, in their power and their limitations according to the users and their domain related purposes, which makes it hard to define a single best method of end-user programming. Accordingly, we have seen a variety of end-user programming techniques, including Programming by Demonstration (Cypher, 1993), visual programming (Green and Petre, 1996) and many task specific languages (Nardi, 1993).

Nardi (1993) argues that the reason for the success of task specific languages, such as spreadsheets, CAD systems and statistical packages, is that the language primitives are application level primitives, which are already familiar to the users. In spreadsheet calculation, these application level primitives provide the user with the means of constructing applications within the application domain. Spreadsheet formula language is simple, and it is also felt to be

simple by the end-users because it enables them to do the things they do in their everyday lives, such as adding, subtracting, and rounding numbers.

1.2.2 The current spreadsheet model

In the current spreadsheet model, the data values and computations are laid out in a simple two-dimensional grid, easy for the user to view and modify. Some of the current spreadsheet systems advertise three-dimensionality, by which they mean that there can be a stack of two-dimensional grids in the same workspace (i.e., a file). The two-dimensional grid resembles a cross ruled sheet of paper, thus giving the user a familiar representation to work with. Users may feel that they are working directly on their problem domain task rather than using a computer.

The use of the spreadsheet grid has not changed much since the introduction of VisiCalc in 1979. The grid is still divided into columns, which are labeled by upper case characters starting from A, and into rows, which are numbered starting from 1. The cells are referenced by both column and row reference, e.g. cell A1. A cell can contain a numeric value, a label, or a formula. Labels are distinguished from formulae by the first character typed: a special character, such as +, =, or @, denotes a formula. A cell has both a value and a format. The format defines how the value of the cell is shown to the user.

Computations are specified by means of writing formulae, which contain cell references, arithmetic operators, and function calls. The cell references can be relative, absolute, or mixed (when the row reference is absolute, the column reference is relative, and vice versa). When a formula is copied, the relative references change according to the distance copied; for example, if the formula =B4+7 is copied into a location two cells down and three cells to the right, the copied formula will read =E6+7. With absolute references, neither the column nor the row reference will change when the formula is copied. With mixed references, the relative component (either the row or the column reference) changes according to the distance copied but the absolute component remains the same. The set of common functions in a spreadsheet system contains several hundred functions. The typical classes of functions include date and time, financial, statistical, mathematical, database, logical, and look up function classes.

The user interaction style of current spreadsheet systems is one of direct manipulation interaction. The user points to a location in the grid and changes the value of that cell by typing the new value. The spreadsheet system automatically recalculates the formulae in the sheet when a new value is entered. A common way to construct computations with a spreadsheet system is to enter some data first, then make a formula once, and then copy it into other locations as needed. In a study of command use in spreadsheet calculation, Guadagno et al. (1990) found that one of the most likely command sequences was the Copy command followed by either the Copy or the Move command (it should be noted here that the system

used was Lotus 1-2-3, in which the Copy command includes the Paste command). The Copy and the Move commands are the normal means to work with the computations and the layout of the spreadsheet. The Move command is needed to add space for new computational elements or explanatory application headers.

Spreadsheet formula language is a task specific end-user programming language. It provides the user with task specific programming primitives and simple control structures. It offers rather a small number of primitives only, but these primitives map directly onto operations that users typically need, such as taking averages, rounding numbers, and performing basic arithmetic operations, e.g. calculating percentages. Spreadsheet formula language thus makes spreadsheet operations accessible to end-user programmers, not only because these operations are familiar from the users' task domain but because the users can manipulate the operations without the usual programming domain activities, such as allocating memory to the variables or compiling the application.

Many studies have found that the control structures of general purpose programming languages are difficult for novice users to understand (Hoc, 1989, Lewis and Olson, 1987, Soloway et al., 1983b). The control structures are essential for programming activity, which makes a barrier for novice programmers to conquer before they can master the language. Spreadsheets provide the user with much simpler control structures than the general programming languages. The sequence of operations in spreadsheets is managed by the cell dependencies in the formulae. This presents an implicit control flow, which links the cells together. The only control structures in spreadsheet calculation are iteration and selection.

Iteration takes two forms in spreadsheet calculation. First, spreadsheet functions can iterate an operation over a range of cells. For example, the user can sum the values of ten adjacent cells by issuing a formula SUM(A1..A10). The other form of iteration is copying a formula. The user can iterate the formula SUM(A1..A10) to the second column of the sheet by copying the formula in a row from the first to the second column. The cell references in the formula will change to accord with the position of the new formula (i.e., the new formula will be SUM(B1..B10)).

Selection is expressed by the conditional function IF in spreadsheet calculation. An example would be IF(B1<>0; A1/B1;0). According to this formula, if the value in cell B1 is not 0, then the function returns the value of cell A1 divided by the value of cell B1; otherwise the function returns the value 0. This is a normal means of guarding against division by zero errors in spreadsheets. It could be argued that selection is easier to understand in spreadsheet calculation than it is in general purpose programming languages because in spreadsheet calculation, selection does not transfer control from one part of the spreadsheet to another. Its effects are local, pertaining only to the cell where the formula resides. On the other hand, the user sometimes has to allocate a new area in the sheet for the selection, for example when computing a guarded sum of a set of numbers including only those values that exceed 10.

1.2.3 Applications of the current spreadsheet model

Spreadsheets are used in a large variety of applications. They include specific computational needs such as mathematical modeling (Arganbright, 1986) or learning physics (Misner and Cooney, 1991). Even so, it can be argued that spreadsheets are most commonly used for the purposes that the spreadsheet paradigm was originally designed for, i.e., for tasks that could be performed without computers, by means of paper spreadsheets. These task types include financial applications, such as budgeting, and administrative applications, such as managing small businesses. The common characteristic of these tasks is that their complexity lies in the relationships between the entities in the problem domain itself, not in the programming needed to create the formulae that model these relationships (Nardi, 1993).

Sajaniemi and Pekkanen (1988) did a quantitative analysis of a total of 135 spreadsheets collected from Finnish business companies and governmental agencies. In their summary they conclude that the spreadsheets contained very little calculation. On average, only 5 % of the cells contained formulae, and the sheets used a small number of distinct operators (average 4; range 0-10) and functions (average 2; range 0-8). In 90 per cent of the cases, the operators used were the four basic arithmetic operators (+, /, *, -). The three most commonly used functions IF, SUM and ROUND covered about 80 % of all function use. The formulae were quite short, containing 3 references to cells and 2 operators on average. The average length of the formulae was 22 characters and only every other formula contained a function call.

Other researchers have obtained similar results. In a series of studies of spreadsheet use at work settings, Nardi and Miller (1990, 1991) interviewed 11 spreadsheet users extensively, auditing several spreadsheets in each interview. They conclude that in the spreadsheet applications they studied, they found many complex applications expressive of rich domain semantics. For an example of the types of application Nardi and Miller are referring to, see Figures 1 and 2. The computations in these models, however, are quite simple. Nardi (1993) found that most users made use of fewer than ten functions in their applications; these typically included the basic arithmetic and rounding functions.

BogusCo, International
Changes in Working Capital
FY 88

	Balance Sep -87	Difference	Balance Sep -88	Total Net Change	Translation
Short term investments	1 000	248	1 248	248	0
Receivables	85 626	2 786	88 412	2 446	340
Inventories	54 460	-4 862	49 598	-5 343	481
Prepaid Expenses	1 373	62	1 435	61	1
Deferred Income Taxes	1 949	3 932	5 881	3 932	0
Total Current Assets	144 408	2 166	146 574	1 344	822
Bank Notes Payable	-25 937	1 779	-24 158	1 869	-90
Current Portion LTD	-1 838	1 205	-633	1 187	18
Trade Accounts Payable	-8 012	432	-7 580	413	19
Other Accrued Expense	-21 314	-6 089	-27 403	-6 206	117
Warranty Reserve	-6 941	-157	-7 098	-133	-24
ROUNDING	-1	1			1
Total Current Liabilities	-64 043	-2 829	-66 872	-2 870	41
TOTAL	80 365	-663	79 702	-1 526	863
Notes Payable		1 779		1 869	-90
Long term Debt		1 205		1 187	18
Other WC		-3 647		-4 582	935
Total		-663		-1 526	863
Other WC		-3 647		-4 582	935
Minority interest		804		804	
Total Other WC		-2 843		-3 778	935
Other accrued expenses		-6 206			
Minority interest		804			
Net other accrued expenses		-5 402			

Figure 1: A typical spreadsheet application from Nardi (1993, p.42)

Another large collection, nearly 300 spreadsheet applications, can be found in Butler (2000), which contains spreadsheets used by UK taxpayers to calculate their liabilities. Officers of H. M. Customs and Excise have been conducting audits of these applications since 1985. In the latest audit run, 21 sheets were audited, their average sizes being about 6000 constants and 700 formulae in a single sheet. In spite of the large sizes of the applications, the computations were relatively simple (Butler, 2000).

BogusCo, International					
Changes in Working Capit					
FY 88					
	Balance Sep -87	Difference	Balance Sep -88	Total Net Change	Translation
Short term investments	10[illegible]8	=[illegible]7	12[illegible]	24[illegible]	=[illegible]E7
Receivables	85626	=D8-B8	88412	2446	=C8-E8
Inventories	54460	=D9-B9	49598	-5343	=C9-E9
Prepaid Expenses	1373	=D10-B10	1435	61	=C10-E10
Deferred Income Taxes	1949	=D11-B11	5881	3932	=C11-E11
Total Current Assets	=SUM(B7:B11)	=SUM(C7:C11)	=SUM(D7:D11)	=SUM(E7:E11)	=SUM(F7:F11)
Bank Notes Payable	-2[illegible]37	=D15-B15	-24158	1869	=C15-E15
Current Portion LTD	-1838	=D16-B16	-633	1187	=C16-E16
Trade Accounts Payable	-8012	=D17-B17	-7580	413	=C17-E17
Other Accrued Expense	-21814	=D[illegible]-B18	-27403	-62[illegible]6	=[illegible]8-E18
Warranty Reserve	-6941	=D19-B19	-7098	-133	=C19-E19
ROUNDING	-1	=D20-B20			=C20-E20
Total Current Liabilities	=SUM(B15:B20)	=SUM(C15:C20)	=SUM(D15:D20)	=SUM(E15:E20)	=SUM(F15:F20)
TOTAL	=B12+B21	=C12+C21	=D12+D21	=E12+E21	=F12+F21
Notes Payable		=C15		=E15	=F15
Long term Debt		=C16		=E16	=F16
Other WC		=C12+C17+C18+C		=E12+E17+E18+I	=F12+F17+F18+F
Total		=SUM(C24:C26)		=SUM(E24:E26)	=SUM(F24:F26)
Other WC		=C26		=E26	=F[illegible]
Minority interest		804[illegible]		=C30	
Total Other WC		=SUM(C29:C30)		=SUM(E29:E30)	=SUM(F29:F30)
Other accrued expenses		=E[illegible]			
Minority interest		=C30			
Net other accrued expens		=SUM(C33:C34)			

Figure 2: The same spreadsheet application from Nardi with formulae and some annotations showing calculation relations (1993, p.43)

1.2.4 Evaluation of the current spreadsheet model

Spreadsheet systems have extended computing power to people who were not computer users before. The apparent success of spreadsheet systems has been lauded in many publications and magazines. This has resulted in an emphasis on the strengths of the spreadsheet model, probably decreasing the attention paid to the weaknesses of the model.

The most commonly noted strength of the current spreadsheet model is the direct manipulation interface of the spreadsheet (Hendry and Green, 1994, Lewis and Olson, 1987, Nardi, 1993). The users can access data values and computations directly, using a concrete and familiar two-dimensional array metaphor. This user interface metaphor suppresses the

traditional inner world of computation, the world in which the processing is done but to which the user has no direct access. The input data and the computations in a spreadsheet seem to be directly accessible both to the underlying computational system and to the user. The user does not have to program special mechanisms for input and for output.

Another strength of the current model is its automatic consistency maintenance. The user specifies the way the cells depend on each other, and the system maintains these dependencies. Because of this consistency maintenance, the spreadsheet user does not usually need to be aware of the explicit control flow in the application. Changes in a part of the sheet are reflected through the computation dependencies, and the output values are automatically updated and shown to the user. This immediate feedback of changes to the spreadsheet gives the user a feeling of working directly with the task.

Still another frequently mentioned strength of the current spreadsheet model is the level of primitive operations (Lewis and Olson, 1987, Nardi, 1993). Spreadsheet systems offer operations such as “count” or “average”, which are aggregate operations in comparison to those of traditional programming languages. In the traditional programming languages these operations have to be built out of many lower level primitive operations and control structures.

The weaknesses of the current spreadsheet model fall into two categories. The first one deals with the representational aspects of the model and the second one with the construction of the computations within the model. The former category includes the difficulty of debugging spreadsheet applications and of reusing old applications. The latter category includes the lack of modularity, the lack of abstraction, and the lack of proper connection between the data and the computations.

Spreadsheets do contain errors that often go unnoticed (Butler, 2000, Panko, 2000b). One reason for this is that spreadsheets are not easy to debug. Spreadsheet systems usually show the values of all formulae at the interface, but the actual cell contents (e.g., a formula) are shown only one at a time and in a special place at the interface (the formula line). This makes it difficult to grasp the computation as a whole and to locate a possible error in the computation. Because of this, there are attempts in spreadsheet systems to make the computation networks more transparent by including different auditing mechanisms (Sajaniemi, 2000).

The same factors that make spreadsheets hard to debug also make them hard to reuse. In reuse, the user has to deduce the working of a sheet at a level which makes modifications possible. Moreover, spreadsheets are often regarded as short timespan scratch pads, with the consequence that users do not usually document their operations or make comments on the sheet.

In a spreadsheet, the computations are all at the cell level, i.e., all formulae are written into cells. In traditional programming, in contrast, the modularity makes it possible to implement, debug and test an application in smaller parts. It also makes the reuse of code easier because

these smaller independent modules are likely to be needed in different applications. In spreadsheet calculation there are macro facilities which could be used to implement modules, but here the same problems arise as with general programming languages. The user has to work with parameters, local variables, and return values. Anecdotal evidence suggests that the use of macros is hard for end-users (Gray et al., 1992).

Abstraction, a powerful mechanism in traditional programming, allows a programmer to build and use programming constructs at the level appropriate for the task at hand. Software can be layered to a hierarchy of constructs, and different levels can be used for different tasks. For example, when an application programmer is building a graphical user interface, he does not need to worry about the details of different physical interaction devices because the device driver and the layers of the window system hide these details from him. Abstraction in spreadsheets, in contrast, is at a fixed level. Current spreadsheet systems have various features for making summaries of large data sets. These features are called, depending on the particular system used, consolidating, cross-tabbing, grouping, or creating PivotTables. They are designed mainly for reporting purposes, and some systems even replace references to the original data by constant values.

When the user inserts new cells into a referenced area, the automatic updating of the references in the formulae may cause him/her to misunderstand the connection between the data and the computations. If the insertion is done at the edges of the referenced area, the references are not updated automatically. Also, when the sizes of the input data areas change, the systems do not enable an explicit showing of the associated computations, although the change might affect these computations also.

1.2.5 Current spreadsheet systems

The way computations are defined in current spreadsheet systems has not changed much since the introduction of VisiCalc in 1979. The complexity of the interface and the number of new features such easy graphing and automatic help agents have increased, but the formula languages have remained relatively unchanged. There are innumerable spreadsheet systems in the world; some estimates set the number of commercial spreadsheet systems at the market as high as about 700. All main office application packages contain a spreadsheet system. The repertoires of the main spreadsheet systems currently used (latest version in parenthesis), namely Excel (Microsoft Excel 2000), Lotus 1-2-3 (SmartSuite 1-2-3 Millennium Edition r9.5) and Quatro (WordPerfect Office Quatro Pro 9), all offer quite similar sets of features. They include extensive character formatting, automatic layout formatting, analyzing and reporting facilities, chart and graph making, database features, and macro languages. The formula languages of all these systems follow the method described in the previous sections.

There have been some minor changes in these systems regarding the specification of the computations, mainly in the naming of ranges, and the invoking of the calculations. The latest version of one of the systems, Excel, supports implicit range naming, which allows the user to use self specified row and column headings in the computations. For example, if a user has a table of figures with the column heading January, the formula SUM(January) will calculate the sum total of this column. However, a mechanism of this kind is hard to define so as to always give expected results. The example system automatically deduces the area that the computation should use, i.e., the system searches the label starting from the cell in which the user is writing the formula, finding the nearest instance of the label, and then starting to acquire the reference area from that label cell towards the formula cell. If the search finds an empty cell, it stops and the program constructs a reference area. This may not always be the intent of the user, as he may believe that the whole area is included in the calculation.

Another change that has occurred concerns the invoking of calculations. The systems mentioned above include a feature, called QuickSum or AutoSum, for easy implementation of sum functions. The user selects an area in the sheet, then presses the AutoSum button and gets, by default, columnwise sums. Alternatively, the user can select a data area and empty cells on one side of the area and get rowwise sums. However, just as in the case of normal references, the addition of new values is hard to handle. If the user adds a new row of values to the area, no new rowwise AutoSum formula is added to the sheet, and if the new added row is the first one in the area, the columnwise AutoSum formulae are not updated to reflect this change.

There have been a few attempts to introduce spreadsheet systems that differ from the traditional genre. One trend has been to try and change the spreadsheet model to include more structure. The products developed in this line include, in chronological order, Trapeze for the Macintosh systems, Javelin for the DOS systems, and CA-Compete and Lotus Improve for the Windows system. More structure has usually meant that qualifiers of the data model have been introduced as dimensions of the spreadsheet application. The resulting spreadsheet application may have included as many as twelve dimensions, which have enabled the application to be viewed from different perspectives depicting different (business) viewpoints of the modeled data. For various reasons, which include marketing and end-user satisfaction problems, these attempts have all failed. From the usability point of view, these systems failed because they tried to replace the existing two-dimensional spreadsheet model with a host of other concepts, because the usability of the user interface was poor, and because the multidimensional model building was cognitively hard for the users.

Other systems have attempted to introduce various data manipulation techniques, e.g., Mesa and Mesa/2 for OS/2, or various scripting language facilities for incorporating new functionality and usability into the system, e.g., WingZ with the HyperScript language, Lotus HAL with a limited, natural language based command language, and a more academic project, the Analytic Spreadsheet Package, with SmallTalk as the formula language (Piersol, 1986).

One system, Spreadsheet 2000 by the Casady & Green Company, has tried to replace the textual nature of specifying calculations with a more visual programming style. All the elements in the user interface are visual, the input data and the output areas (boxes) are floating elements over the sheet, the calculations are constructed by means of visual elements for operators, and the operands and operators are connected by wires showing the flow of control. The approach has a number of benefits but also a number of drawbacks. For an example of the latter, in larger applications it becomes difficult to maintain the wire box notations. For further discussion of the usability of visual programming environments see, e.g., Green and Petre 1996.

1.3 Goals and plans in programming

In spreadsheet calculation, the programming is done by writing formulae which connect the data cells to the computations. The success of spreadsheet calculation has been attributed to the ease of formula specification. Many of the programming ‘games’ needed in traditional programming have been avoided in spreadsheet calculation (Lewis and Olson, 1987). Even so, the constraints of human cognition which govern more traditional programming govern spreadsheet programming just the same, and the same rules of expertise prevail.

Expert programmers know more about programming than novices do, they perform faster and more accurately, and they produce similar solutions to programming problems. The differences between novice and expert programmers have raised questions about the nature of the phenomena that might explain these differences. One such attempt is the schema-based goals-and-plans approach. Goals and plans have been used to describe the knowledge required in common programming tasks. In this section we present the goals-and-plans approach to programming. The section starts with an introduction, which deals with the development of human memory structures and expertise. The goals-and-plans approach to programming is then described and related to programming expertise. Next, the cognitive processes of plan creation and use are described, and the section concludes with an account of developments related to the goals-and-plans approach.

1.3.1 Human memory and expertise

Chunking is a well-known psychological phenomenon of human memory. The term chunk was introduced by Miller (1956), who argued that the capacity of our working memory is limited by the number of units that can be active at the same time, and that the number of these

units or chunks is about seven. The physical size and complexity of these chunks, however, is not limited, i.e., they can contain letters, words, sentences, or more abstract representations. Thus the varying results obtained from empirical studies of memorizing nonsense syllables, word lists and other list items were nicely explained by Miller.

De Groot (1965) was the first to understand the interaction of expertise and chunking. He showed different chess positions to chess players for 3-7 seconds, then removed the chess pieces and asked the subjects to reconstruct the position. Some of the positions were taken from real games and some were randomly ordered. He found that the number of recalled positions increased with expertise in chess when real game positions were displayed. The master level players were able to remember from 20 to 25 pieces in position after 5 seconds of display, while the novices remembered from 4 to 5 pieces only. In random orders this effect was not found. All the players could remember only a few pieces, and the difference between the expert and the novice was only about 2 pieces.

This experiment has been replicated many times, and the same results have been obtained. It has also been verified that it is the expertise and not the player’s age which produces the result (Chi, 1978) and that the expert’s recalling does not disappear even after a 30 seconds’ delay with an interfering task (Charness, 1976). The explanation for the phenomenon is simple. Expert chess players have learned numerous specific chess positions during their many years of playing chess. Now when they see a complex position they can code it rapidly into a few easily remembered chunks. It has also been shown that chunks grow slowly with increasing expertise in the area (Chase and Simon, 1973a).

De Groot’s experiment has been adopted to many domains, e.g., to bridge, go, basketball, baseball, music, architecture, electronics and computer programming (Akin, 1980, Allard et al., 1980, Charness, 1979, Chiesi et al., 1979, Eagan and Schwartz, 1979, Engel and Bukstel, 1978, McKeithen et al., 1981, Reitman, 1976, Shneiderman, 1976, Sloboda, 1976). The results have been quite similar. Clearly, chunking is a way to get round the physical limitations of the working memory. Experts have no better or larger working memories, but they have a great deal of larger and more complex memory chunks than novices do.

Between the expert’s chunks and the novice’s chunks there is a qualitative difference also. Chess experts’ chunks were found to frequently consist of pieces that form either attack or defense positions (Chase and Simon, 1973b). In physics, experts classified problems according to underlying physical principles such as Newton’s Laws, while novices classified the same problems in terms of some surface features such as inclined planes (Chi et al., 1981). In programming, experts’ better recall of ordered as opposed to scrambled programs has been demonstrated with different programming languages, including BASIC (Barfield, 1986), FORTRAN (Shneiderman, 1976), and ALGOL (McKeithen et al., 1981). The differences in qualitative knowledge organization have also been demonstrated in programming experiments with novice, intermediate, and expert ALGOL programmers (McKeithen et al., 1981) and novice and expert PPL programmers (Adelson, 1981).

Programming involves knowledge of different domains, including at least the programming domain and the problem domain (Brooks, 1983). The problem domain description of a programming problem represents the desired goal or the decomposition of the goal into several subgoals, and it forms the highest level description of the problem. It is from this description that the programmer starts to design his/her solution to the problem. By means of some design strategy he/she develops a complex goal structure to solve the problem. Some of the goals he/she introduces because of the programming domain and some because of the problem domain. At some level these goals can be achieved with programming language structures, and all these structures must have some representations in the programmer's mind. Expert programmers must have many large memory structures about programming concepts.

1.3.2 Programming goals and plans

Schema theory is a theory of knowledge organization in human memory and of the processes involved in using that knowledge (Detienne, 1990). A schema (Bartlett, 1932) is a human memory structure which consists of a set of propositions organized by their semantic content. For example, we could imagine a schema of the human face, the propositions of which would be two ears, two eyes, a nose, a mouth, and a certain semantic structure, i.e., an oval structure where the eyes would be above the nose, the nose above the mouth, and the ears on the sides at about the level of the eyes. The schema concept has been a fruitful one for explaining many phenomena involving expectations and commonalities in human cognitive behavior. Since the schema concept has a dominant role in modern cognitive studies, many variants of it have been developed, e.g., scripts (Schank and Abelson, 1977), frames (Minsky, 1975), MOPs (Schank, 1982), and plans (Soloway and Ehrlich, 1984).

```
Frame: Counter_Variable Plan
Description: counts occurrences of an action
Initialization: Counter := 0;
How Used: Counter := Counter + 1;
```

Figure 3: Plan for the counter (adapted from Soloway et al., 1982).

The schema-based theory of programming knowledge has been developed mainly by Soloway and his research group (Detienne, 1990). In the terminology of Soloway's group, schemas are called programming plans. Their plan concept is role-based, i.e., plans fill roles in programs as actors fill roles in a play. Their plans represent stereotypic action sequences in programming, and the concept has a rich frame like semantic structure. Plan descriptions define the common roles of code fragments in the programs (Soloway et al., 1982). For example, the counter (see Figure 3) is an example of a type of plan Soloway is talking about.

The research of Soloway's group has followed two main tracks: studies of expert programming knowledge by means of plan analysis, and studies of novice programming errors and misconceptions. They have related the errors and misconceptions found in novice programming to expert programming knowledge in order to explain why novices make certain kinds of errors. They have developed a set of software tools to test and utilize their model of plan-based programming: PROUST, which diagnoses novice programs for typical programming plan errors (Johnson, 1986), MARCEL, which simulates the novice program creation process (Spohrer, 1989) and GPC-editor (Goal-Plan-Code editor), which is an editing environment that supports problem decomposition by means of a goal library and problem solution by means of a plan library (Weaver et al., 1989).

Working as their own expert group of Pascal programmers, Soloway's group have identified a set of programming plans used in elementary looping problems (Soloway et al., 1982). They distinguished between variable plans and control flow plans (Ehrlich and Soloway, 1984). Variable plans describe the function or role of the variable and its manner of use in programming. The plan is composed of a set of slot types (descriptions, variables) and a set of slot fillers (specific values for the instantiation of the plan). An example of a variable plan is shown in Figure 4. Control flow plans describe common looping structures for the accumulation of values and for keeping track of the number of instances of values.

```
Frame: Running_Total Variable Plan
Description: builds up a value one step at a time
Initialization: Running_total := 0
How Used: Running_total := Combine(Running_total, New_value)
```

Figure 4: Tactical plan for a running total variable (Soloway et al., 1982).

They also classified their plans into three categories: strategic, tactical, and implementation plans. Strategic plans specify a global method used in an algorithm. For example, the Read/Process looping strategy specifies the "first read a value, then process it" method. Tactical plans specify a local strategy for solving a problem. For example, the Counter_Controlled Running_Total Loop Plan stipulates that the iteration for accumulating a total sum is controlled by the value of the Counter variable; see Figure 5. These two plan types are argued to be independent of programming language. Implementation plans specify programming language implementations of the tactical and the strategic plans. It should be noted that Soloway's group present their plans at the level needed for the specific set of programming problems, i.e., elementary looping problems in this case. It is possible, however, to think of some other level of description. For example, the Counter_Controlled Running_Total Loop Plan could include many separate plans, such as a Counter Variable Plan, a Running Total Variable Plan, and a Running Total Loop Plan.

The plans are linked together by the relations of specialization ("instance-of") and implementation ("uses") (Soloway et al., 1982). For an example of specialization, the Running_Total Loop Plan can be specialized into three plans: the Total_Controlled, the Counter_Controlled, or the New_Value_Controlled Running_Total Loop Plan. These plans differ from each other in the variable that controls the iteration. The implementation relation links the tactical and strategic plans to their programming language dependent realizations. The programming language Soloway 's group mainly uses is Pascal.

```
Frame: Running_Total Loop Plan
Description: build up a running total loop in the loop,
      optionally counting the number of iterations
Variables:  Counter, running_total, and New_value
Set Up: initialize variables
Action in Body: Read, Count, Total
```

Figure 5: Tactical plan for a running total loop (Soloway et al., 1982).

Soloway's research has concentrated on static representations of programming knowledge rather than the process of program writing. One exception is Soloway, Bonar and Ehrlich (1983b), which studied the strategies of implementing looping in Pascal and found that Pascal's looping structures may not reflect the preferred user strategies for looping. Instead of continuing this line of research, however, Soloway's group turned their attention from strategies to expert programming knowledge.

The explanation given by Soloway's group for the programming process is fairly straightforward: "Faced with a new problem, an expert retrieves plans from his/her knowledge base which have proven useful in similar situations, and then weaves them together to fit the demands of the new problem" (Soloway et al., 1982).

Table 1: Rules of programming discourse (Soloway and Ehrlich, 1984).

```
(1)  Variable names should reflect function.

(2)  Don't include code that won't be used.

(2a) If there is a test for a condition, then the condition
     must have the potential of being true.

(3)  A variable that is initialized via an assignment
     statement should be updated via an assignment statement.

(4)  Don't do double duty with code in a non-obvious way.

(5)  An IF should be used when a statement body is guaranteed
     to be executed only once, and a WHILE used when
     a statement body may need to be repeatedly executed.
```

They have argued that the composition of plans is governed by the rules of programming discourse (Soloway and Ehrlich, 1984). Their set of specific programming discourse rules is presented in Table 1.

Experts are usually not aware of using programming knowledge of this kind, and this is why the term tacit knowledge is often used (Ehrlich and Soloway, 1984). Soloway's group has verified the existence of such tacit knowledge by conducting a series of empirical studies with programmers of different skill levels (Soloway and Ehrlich, 1984). In one type of study they used fill-in-the-blank problems. They presented the subjects with short programs that lacked one or two lines of code and asked them to fill in the blank lines with statements that best completed the program. They constructed two types of programs: plan like programs, which used typical programming plans and followed the rules of programming discourse, and unplan like programs, which violated one or two of the rules of programming discourse.

These studies supported the claim that expert programmers have, and make use of, programming plans and programming discourse rules (Soloway and Ehrlich, 1984, Ehrlich and Soloway, 1984). Experts performed significantly better than novices in the plan like programs, whereas in the unplan like programs the performance of experts was reduced to essentially that of novices. Furthermore, in a recall study the experts recalled the critical lines (i.e., the lines that make a program either plan like or unplan like) better and earlier in the plan like programs than in the unplan like programs. This accords with the well known psychological fact that the representatives of a particular category are recalled earlier than other items.

In their description of the problem solving processes involved in programming, Soloway's group declares that program planning, especially problem decomposition, is quite a top-down breadth-first process (Adelson and Soloway, 1985). They represent the space of possible correct solutions to a programming problem by a Goal and Plan Tree (GAP tree) (Spohrer et al., 1985a, Spohrer et al., 1985b). A GAP tree contains a set of goals and subgoals needed for a solution and all the different plans that could fulfill the intentions of the goals. So, the first level of a GAP tree contains the main goal of the problem, i.e., the definition of the programming problem, and at the second level there is a decomposition of the main goal into a set of subgoals that need to be achieved. The third level contains all the different plans for each subgoal, and at the fourth level these plans may have a set of subgoals to be met in order to carry out the whole plan. This division into goals and plans continues down to the very lowest level, which contains the implementation plans for various programming language constructs. If we are looking for any one good solution to the original problem, we can find it as a solution subtree in the GAP tree.

One direction that Soloway's group has pursued in their research is the very first programming experiences of novice Pascal programmers (Bonar, 1985a). They found that novice programmers used step-by-step natural language procedural knowledge extensively when confronted by impasses in the programming task. The novices followed functional or

surface links from the programming domain to that of natural language problem solving and then solved the problem in this familiar domain. The other type of impasse resolving used by the novices was generalizing or specializing from programming plans already learned. Unfortunately, these behaviors often produced errors in the solutions. An especially hard task for the novices seemed to be plan composition, i.e., the putting together of spatially dispersed plans and plan components to make a correct working solution. The novices seemed to know all the necessary plans but not how to put them together (Spohrer and Soloway, 1986b).

In their studies of novice programmers, Soloway’s group concluded that it may not be true, as is commonly thought, that most programming errors made by novices are due to their misunderstanding of programming language semantics (Spohrer and Soloway, 1986a). They analyzed syntactically correct novice programs by means of a goal/plan analysis and identified reasons for the bugs found in the programs (Spohrer et al., 1985b). Their analysis was based on inferred GAP-tree solutions, obtained by collecting about 50 novice programs per problem and then identifying all different plans for the subgoals of the problems. They then identified the actual plans used in each of the novices’ programs and characterized the differences between the novice’s actual implementations and the correct implementations of the plans. They were able to locate the differences at some particular part of the plan, i.e., at the input, output, initialize, update, guard, syntax (i.e., delimiters for the scope of a block of code), or the whole plan (i.e., many components of the plan wrong).

The components of a plan could be wrong in many ways, i.e., a component could be missing, malformed, misplaced, or spurious (i.e., present where none is required). Soloway's group classified all the errors, but the analysis still did not tell what the programmer's misconception had been. Similarly, the analysis showed where a bug was but did not indicate why it occurred. So they refined the analysis to account for the problem dependent solution GAP-tree and to identify the actual goal which contained the bug. Now they were able to tell what type of error the bug was and in which component of which plan it occurred. By this means they were able to infer what the misconception of the programmer had been. For example, the novice programmer's failing to conceive all the required subgoals or omitting some part of the plan could have been caused by the cognitive complexity of the plan creation process. Furthermore, they found that several bugs in a single program often originated from a single underlying misconception and that some plans (towards the same goal) were harder than others (Spohrer et al., 1985a, Spohrer et al., 1985b).

What emerged as especially hard situations were those in which a single plan was used to achieve multiple goals. For situations of this kind Soloway’s group used the notion of merged goals/plans (Spohrer et al., 1985c). A typical example of novice plan merging is a situation in which the program has to input numerous values of different types (e.g. sex, age, and height), guard against invalid data, and retry for new values for invalid data. Novices tend to merge all the inputting together, i.e., they will include all the different types of values into the same input statement. This makes the guarding and retrying very hard; for example, in

guarding one has to write many Boolean expressions containing checks for certain input values, upper bounds, lower bounds, etc. Furthermore, Soloway's group found that merging can lead to goal dropouts (i.e., some subgoal being left out when goals are merged) and goal fragmentation (i.e., parts of a subsequent plan being merged into the previous plan) (Spohrer et al., 1985b). Errors of this kind they related either to a previously learned erroneous plan or to cognitively complex processing.

Detienne (1993, 1995) extended the goals-and-plans approach to the object oriented programming paradigm. She conducted an empirical study of eight professional programmers, some of them experienced in object oriented programming and some not, concentrating on the transfer effects of procedural programming knowledge on object oriented programming (Detienne, 1995).

Detienne's results suggested that the plans used for the implementation and design of programs varied according to the subject's experience of programming language. If the programmer had more experience with object oriented programming, he/she organized the plan development at the program design phase around object oriented plans, integrating actions and objects. If he/she had less experience with object oriented programming, he/she used plans related to procedural programming languages, grouping actions in the order of execution, a method typical for the procedural paradigm.

1.3.3 Plans and programming expertise

It has been suggested that the formation of goal-and-plan knowledge in programming language cannot fully explain the development of programming expertise (Davies, 1993). For example, by using some programming discourse rules, programmers seem to create expectations on how programs should be written (Soloway and Ehrlich, 1984). Davies (1990a, 1990b) has done a series of studies focusing on other types of programming knowledge and the development of programming expertise. His main argument is that the theories of programming plan knowledge are theories of declarative knowledge only and have therefore failed to account for how this knowledge is acquired and used.

Davies (1990a) claims that the development and use of programming plans may be bound to the particular learning experiences of the programmer. He tested two groups of BASIC programmers, one of which had had a course in program design and the other one not. The programmers had all had at least 18 months of experience with BASIC. Davies presented them with short programs with bugs of different types and asked them to find the errors in the programs. The bugs were either surface-level, control-level, or plan-level ones. The programs also contained different types of cues to the bugs: a control structure cue, a plan structure cue, or no cue.

The overall bug finding performances of the design experienced and the non design experienced group did not differ in regard to the bug or cue situation. However, the design experienced group performed significantly better in situations in which the plan structures were cued or the bugs were plan related. In a related study, Davies (1990a) used the recall method in longitudinal settings. Two groups were tested for recall of critical lines in plan like and unplan like programs. In the first recall experiment there were no significant differences between the two groups. Then, one group was taught program design for 5 months and both groups were taught BASIC programming for 5 months. In the second recall trial, the group which had been taught program design recalled critical lines from plan like programs significantly better than the other group. Davies concluded that experience in design affects the nature and development of programming plans.

Davies (1990b) also claims that the existence of programming plans does not describe skill differences in program comprehension adequately. He conducted an experiment where different skill level groups were presented programs with blank lines and three different code fragments to fill in the blank lines. The code fragments either corresponded to the plan structure or violated it in some way, or alternatively, they corresponded to a discourse rule or violated it in some way. For each blank, the subjects had to choose the best suited fragment to fill in the blank, and the second and the third best. The result was that for plan structure programs, both the intermediates and the experts selected the fragment conforming to the plan structure significantly most often, but the novices did not. The novices had 2 months of programming experience, the intermediates had 9 months, and the experts were professional programmers. On the discourse rule programs, the experts selected the fragment conforming to the discourse rule significantly most often, but the intermediates and the novices did not. From this Davies concluded that the plan structure appears to prevail at the intermediate as well as the expert level of programming skill, and that expertise would appear to be related to both the use of programming discourse rules and the use of programming plans.

In a later study Davies (1994) demonstrated that the recognition of certain salient plan features depends on the skill level of the subject. These salient features are often called focal lines (Rist, 1986, Rist, 1995); i.e. lines which directly encode the goal of the plan. The subjects participating in Davies's 1994 study were assigned to three groups according to their level of programming expertise. He presented the subjects with short Pascal programs similar to the programs Soloway’s group had used in their studies and asked them to memorize the programs. After 10 minutes of memorization the subjects were shown lines of code on a VDU, one line at a time, and were asked to press the *yes* or the *no* button to indicate whether the line was from the program or not. It was shown that the experts reacted significantly more correctly and faster to the focal lines of the plans than did the novices or the intermediates. This effect was not found in non focal lines, where a linear dependency was observed between the subject's skill level and his/her response.

Davies (1991) has also found evidence of interactions among programming notations, programming skill level, plan utilization, and programming strategy. In a program production study using subjects of three skill levels and two programming languages, BASIC and Pascal, Davies measured the jumps which occurred between and within the plans in code writing sequences and the lengths of the pauses which occurred between the jumps. He concluded that when expertise develops, it decreases the number of within plan jumps, so that the plans are written in a continuous manner, and increases the number of between plan jumps, so that the plans are written one after another. At the intermediate level (i.e., computer science majors of the second and the final year) the programming language used produced differences in the programming strategy. The number of between plan jumps was distinctly higher with Pascal than with BASIC. Davies argued that Pascal notation appears to support greater plan utilization than BASIC notation does and that this support to the programming strategy is particularly strong at the intermediate level.

1.3.4 The creation and use of plans

Rist (1986, 1989, 1990, 1991, 1994, 1995) who has developed the programming plan concept further, has concentrated mainly on the cognitive processes of the creation and use of programming plans. He has introduced the concepts of focus and focal development of a plan. The focus of a plan is the single most important part of the code that immediately implements the goal. For example, if the goal is to count something, then the focus of the plan that implements the counter goal is the code fragment that does the incrementing, "+1". The programmer develops this focus into the focal line of the plan, which in this case, if the name of the counter variable is Counter, will read "Counter := Counter +1". The programmer then extends the plan by initializing the variable, i.e., "Counter := 0". This plan creation process is called the focal development of a basic plan by Rist (1989). He claims that when a programmer is developing his/her programming plan knowledge or is creating a plan for a goal for the first time, the order of writing the lines of code for the program follows the focal development order. When a programmer already has the plan for the goal in her programming knowledge, then the writing of the lines of code takes place in a schematic order, which is the same as the one in which the lines appear in the final program code. Rist has found empirical evidence for these programming processes in several protocol studies (Rist, 1989, Rist, 1991).

```
Plan Creation:
      Extension: write('Enter...');
      Focus: read(number);
      Goal: value of number
Plan Retrieval:
      Initialization: write('Enter...');
      Calculation: read(number);
      Output: value of number
```

Figure 6: Prompt plan according to Rist (1989).

Rist (1989) makes a clear distinction between basic plans and complex plans. His basic plans correspond to Soloway's variable plans and simple looping plans. Rist identifies five common basic plans: the prompt plan (reading input), the label plan (writing output), the running total plan (counting occurrences or calculating a sum), the found plan (a guard for the stopping condition of a loop or execution of a series of statements), and the loop plan (repeating a series of statements). He bases his categorization of basic plans on the process involved: whether the plan is created or used (retrieved from long term memory). An example of Rist's basic plan is shown in Figure 6. His complex plans are made up of basic plans. He has found a difference in the strategy employed by novices and intermediates when merging basic plans into complex plans. Novices add one plan at a time to the solution and make difficult compensations in the existing solution in order to add a new plan. Intermediates create their complex plans mentally and write out the merged complex plan in a schematic order (Rist, 1991).

Rist (1994) has developed some tools for extracting plan structures from programs. These include PARE (= Plan Analysis by Reverse Engineering), which searches plans by starting from the goal (one of the outputs of the program) and tracing back through the data and the control flow. This process is repeated for all the goals in the program. The result (the plan structure) is a directed acyclic graph with multiple roots, one root for each goal. A plan is a branch (including the root) of the plan structure. This definition of plan is quite similar to the definition of program slices used in program debugging (Weiser, 1982). Rist (1990, 1995) has also used his findings to explain program design processes.

1.3.5 Related developments to goals-and-plans approach

Computer science, especially programming, is a science of methods. It is a systematic study of algorithmic processes that describe and transform information. The body of generally known programming knowledge is growing and broadening all the time. Programmers have always talked about programming structures such as algorithms, data structures, software reuse, abstractions, and programming methods. All of these must have some representations in the programmer's mind. Besides goals and plans, these structures have been called

beacons in program comprehension (Brooks, 1983), templates in learning programming (Linn and Clancy, 1992), or slices in program debugging (Weiser, 1982). Besides these empirical studies of programming, there have been studies concerning the acquisition of domain concepts and the reuse of designs and programming solutions.

Knowledge representation, object oriented analysis and design, and conceptual modeling in information systems all rely on the acquisition of domain concepts (Parsons and Wand, 1997). Conceptual modeling is needed for constructing, designing and defining the problem domain and the knowledge content needed to create applicable implementation independent conceptual models. The problem is that these conceptual models include a great number of concepts that are comprehensible only in their organizatorial and social context (Falkenberg et al., 1996). The classification of domain concepts is a major step in the construction of models. The classification can be similarity based (statistical) or goal oriented (Parsons and Wand, 1997). For evaluating the classification schemes or choosing among them, the existing methods offer little help. It is possible to devise different classifications of the same concepts, but the classification should be evaluated with respect to the context or the purpose of constructing the conceptual model. The goals-and-plans approach to programming represents one possible classification of programming knowledge.

One recent notion that has developed in object oriented programming is design patterns. A design pattern is an explicit description of a structure: it names, abstracts, and identifies those key aspects of a common design structure which make the structure useful for creating a reusable object oriented design (Gamma et al., 1995). Although the research has not presented the design patterns as cognitive structures, we speculate that their connection to the schema based programming plan concept is obvious. There has been a plethora of books about the pattern languages of programming, design and analysis; see, for example (Coplien and Schmidt, 1995, Fowler, 1997, Martin et al., 1997, Vlissides et al., 1995).

In their description of the definition process of a design pattern, Gamma et al. (1995) say that to notice that something forms a pattern is the easy part; the hard part is to describe the patterns so that other programmers will understand them. Talking about the value of a pattern catalog in particular, they point out that although experts recognize the value immediately, the only ones who can understand the patterns are those who have already used them. Gamma et al. had to extend the descriptions of design patterns to include examples of use and some sample code. They also started to examine the trade offs of the patterns and the various ways of implementing them. In this way they have made design patterns more accessible to new programmers, they argue. Another change they have made to the descriptions is a greater emphasis on the problem solved by the pattern. The easiest way to understand a pattern, they argue, is to see it as a solution to a common programming problem or as a technique that can be adapted and reused. One could argue that if we substitute the term “goal” for “common programming problem” and the term “plan” for “design pattern”, we will have one

representation of the complex goal-and-plan structure of an expert object oriented programmer.

1.4 Goals and plans in spreadsheet calculation

In the previous sections we have described the main concepts of spreadsheet calculation and programming goals and plans. We have also reviewed some research findings on these topics. The next chapters contain four papers, which elaborate the goals-and-plans approach to spreadsheet calculation.

We started our research with an empirical study of real life spreadsheet applications, intending to focus on the deep structure of spreadsheets and the cognitive fit of current spreadsheet systems to human cognition in programming. We analyzed 135 spreadsheets, collected by Sajaniemi and Pekkanen (1988), which were used by Finnish business companies and governmental agencies. We wrote a program which extracted all the computations (i.e., all individual connected formula graphs) from the spreadsheets and then, in a manual analysis of the applications, determined their physical and logical data structures and their goals and plans by looking at the semantic information in the references of the formulae and in the data labels.

As we wanted to describe the kind of knowledge that can potentially be used in any spreadsheet, our interest in spreadsheet plans was at the lowest levels of the plan generalization hierarchy, i.e., at the same level as Soloway's group's interest in their description of Pascal plans. The structures in the spreadsheets we studied were quite comprehensible, and the intentions of the user were inferrable from the applications. This phase of our research resulted in a collection of spreadsheet goals and plans, a technical report by Markku Tukiainen and Jorma Sajaniemi, which is presented in Appendix A under the title "Spreadsheet Goal and Plan Catalog: Additive and Multiplicative Computational Goals and Plans in Spreadsheet Calculation". It shows that the goals-and-plans approach can be utilized to represent real life spreadsheet calculation knowledge. This finding enabled us to formulate a theory of spreadsheeting knowledge and to develop a new model of spreadsheet calculation. These are presented in Chapters 2 and 3. Chapter 4 reports an empirical comparison of the new model with the traditional spreadsheet model. Finally, Chapter 5 completes our argumentation by discussing the criticism leveled at the goals-and-plans approach.

Chapter 2 contains the paper "Goals and plans in spreadsheet calculation", a technical report by Jorma Sajaniemi, Markku Tukiainen and Jarmo Väisänen. The purpose of this paper is to present a framework for studying spreadsheet knowledge and to apply this

framework to the analysis of real spreadsheets. The framework is devised on the basis of our findings, and it makes a clear distinction between goals and plans. The paper shows what types of goals and plans we have discovered, and compares them to the goals and plans discovered in Pascal. The paper is grounded on the empirical analysis of spreadsheets, combined with background data from user interviews. It develops a theory of goals and plans in the context of spreadsheet calculation and presents a model, the DGP/S model, which describes the structure and role of data structures and computational goals and plans in spreadsheets. This model describes the data structure abstractions which are needed in spreadsheets and the way these abstractions can be made further use of in spreadsheet calculation.

Chapter 3 contains the paper "ASSET: A Structured Spreadsheet Calculation System", a journal article by Markku Tukiainen. It describes a spreadsheet calculation tool based on the new spreadsheet model. The name of the article reflects the fact that the new model is a general one, realizable in many different ways. The actual tool implemented, which is just one example implementation of the new model, is called Basset (version B of A Structured Spreadsheet Experimentation Tool). The description of the tool is quite terse and technical because of the space limitations of the journal. The paper confirms that the implementation of the new spreadsheet calculation model is feasible.

One significant decision concerning the user interface of the new model was to leave the grid out of the sheet. The rationale for this was to emphasize the nature of the data structures. The traditional spreadsheet systems are applauded for their familiar and concrete visual representation of data values and formulae (Lewis and Olson, 1987, Nardi, 1993, Norman, 1986). Nardi and Miller (1990) list two important ways in which this two-dimensional grid facilitates problem solving cognition. First, the tabular structure helps the users to organize their models. They do not have to invent a structure – it is given to them. Second, the tables (i.e., coherent cell ranges in the grid) provide a simple mechanism for segmenting models into smaller parts by leaving empty cells between the segments. Basset supports these two ways directly, without using a grid in the sheet. After creating a structure, a Basset user has a tabular structure in a separate segment, which can be positioned (and repositioned) anywhere in the sheet – with as much space around it as needed.

Chapter 4 contains the paper "Comparing Two Spreadsheet Calculation Paradigms: An Empirical Study with Novice Users", a journal article by Markku Tukiainen. Through an empirical study with novice users, the paper evaluates the model presented in this thesis by comparing the traditional spreadsheet calculation model and the new model based on structured spreadsheet calculation. The results show that the two models produce different error behaviors. The traditional spreadsheet calculation model has been criticized for its low conceptual level, e.g., the lack of abstraction and modularity mechanisms, a feature which has been argued to cause many errors in spreadsheet development and use. The paper supports this view by finding a large number of errors evoked by the traditional spreadsheet paradigm.

Structured spreadsheet calculation raises the conceptual level of spreadsheets. The remarkably low rate of paradigm evoked errors in the group of structured spreadsheet calculation users makes it evident that changing the conceptual level of spreadsheet systems will have a significant effect on the number and type of errors which novice spreadsheet users will make. We believe this shows that the higher conceptual level of structured spreadsheet calculation has successfully overcome some of the problems of the traditional spreadsheet model.

The error types occurring in the use of the traditional spreadsheet calculation model belong to the lower level details of formula construction, whereas in structured spreadsheet calculation the error types belong to a higher goal level. When the user's experience increases, the higher goal level errors are more likely to disappear than the lower detail level errors (slips). This means that it is the experienced users who would benefit more from a switch from traditional spreadsheet calculation to structured spreadsheet calculation.

Finally, chapter 5 contains the paper “Goals and Plans in Spreadsheets and Other Programming Tools”, a conference paper by Jorma Sajaniemi and Markku Tukiainen. Although there have been many studies advocating the goals-and-plans approach, there has also been some criticism of this approach. This paper presents the main points of this criticism and argues that it seems to be partially based on some misconceptions of the goals-and-plans approach. In particular, the paper reflects on the distinction between programming knowledge and program knowledge and shows that the goals-and-plans approach is a valid one for the representation of spreadsheet programming knowledge and for tool design.

Adelson, B. (1981). Problem Solving and the Development of Abstract Categories in Programming Languages. *Memory & Cognition* **9**: 422-433.

Adelson, B. & Soloway, E. (1985). The Role of Domain Experience in Software Design. *IEEE Transactions on Software Engineering* 11: 1351-1360.

Akin, Ö. (1980). *Models of Architectural Knowledge*. London: Pion.

Allard, F., Graham, S. & Paarsalu, M. (1980). Perception in sport: Basketball. *Journal of Sport Psychology* 2: 14-21.

Arganbright, D. (1986). Mathematical modeling with spreadsheets. *Abacus* 3: 18-31.

Barfield, W. (1986). Expert-Novice Differences for Software: Implications for Problem-solving and Knowledge Acquisition. *Behavior and Information Technology* 5: 15-29.

Bartlett, F. C. (1932). *Remembering: A Study in Experimental and Social Psychology*. Cambridge: Cambridge University Press, 1977.

Bonar, J. G. (1985a). *Understanding the Bugs of Novice Programmers*. Ph. D. Thesis, Computer and Information Science, University of Massachusetts.

Brooks, R. (1983). Towards a Theory of the Comprehension of Computer Programs. *International Journal of Man–Machine Studies* 18: 543-554.

Brown, P. S. & Gould, J. D. (1987). An Experimental Study of People Creating Spreadsheets. *ACM Transactions on Office Information Systems* 5(3): 258-272.

Butler, R. J. (2000). *Is this Spreadsheet a Tax Evader? How H. M. Customs & Excise Tax Test Spreadsheet Applications*. The 33rd Hawaii International Conference on System Sciences, Maui, Hawaii, IEEE.

Charness, N. (1976). Memory for Chess Positions: Resistance to Interference. *Journal of Experimental Psychology: Human Learning and Memory* 2: 641-653.

Charness, N. (1979). Components of Skill in Bridge. *Canadian Journal of Psychology* 33: 1-16.

Chase, W. & Simon, H. (1973a). The Mind's Eye in Chess. In Chase, W. *Visual Information Processing*. New York: Academic Press.

Chase, W. & Simon, H. (1973b). Perception in Chess. *Cognitive Psychology* 4: 55-81.

Chi, M. T. H. (1978). Knowledge Structures and Memory Development. In Siegler, R. S. *Children's Thinking: What Develops?* Hillsdale, NJ: Lawrence Erlbaum Associates.

Chi, M. T. H., Feltovich, P. J. & Glaser, R. (1981). Categorization and Representation of Physics Problems by Experts and Novices. *Cognitive Science* 5: 121-152.

Chiesi, H., Spilich, G. & Voss, J. (1979). Acquisition of Domain Related Information in Relation to High and Low Domain of Knowledge. *Journal of Verbal Learning and Verbal Behavior* 18: 257-273.

Coplien, J. O. & Schmidt, D. C., Eds. (1995). *Pattern Languages of Program Design.* Reading (Mass.): Addison-Wesley.

Cypher, A. (1993). *Watch What I Do, Programming by Demonstration*. Cambridge MA: MIT Press.

Davies, S. P. (1990a). The Nature and Development of Programming Plans. *International Journal of Man–Machine Studies* 32: 461-481.

Davies, S. P. (1990b). Plans, Goals and Selection Rules in the Comprehension of Computer Programs. *Behaviour & Information Technology* 9(3): 201-214.

Davies, S. P. (1991). The Role of Notation and Knowledge Representation in the Determination of Programming Strategy: A Framework for Integrating Models of Programming Behavior. *Cognitive Science* 15: 547-572.

Davies, S. P. (1993). The Structure and Content of Programming Knowledge: Disentangling Training and Language Effects in Theories of Skill Development. *International Journal of Human–Computer Interaction* 5(4): 325-346.

Davies, S. P. (1994). Knowledge Restructuring and the Acquisition of Programming Expertise. *International Journal of Human–Computer Studies* 40: 703-726.

de Groot, A. D. (1965). *Thought and Choice in Chess.* The Hague: Mouton.

Detienne, F. (1990). Expert Programming Knowledge: A Schema-Based Approach. In Hoc, J.-M., Green, T. R. G., Samurcay, R., Gilmore, D. J. *Psychology of Programming.* London: Academic Press.

Detienne, F. (1993). Acquiring Experience in Object-oriented Programming: Effects on Design Strategies. In Lemut, E., du Boley, B. & Dettori, G. *Cognitive Models and Intelligent Environments for Learning Programming.* Berlin: Springer-Verlag. 49-58.

Detienne, F. (1995). Design Strategies and Knowledge in Object-Oriented Programming: Effects of Experience. *Human-Computer Interaction* 10: 129-169.

Eagan, D. & Schwartz, E. (1979). Chunking in Recall of Symbolic Drawings. *Memory and Cognition* 7: 149-158.

Ehrlich, K. & Soloway, E. (1984). An Empirical Investigation of the Tacit Plan Knowledge in Programming. In Thomas, J. C. & Schneider, M. L. *Human Factors in Computer Systems.* Norwood, New Jersey: Ablex Publishing Corporation. 113-133.

Engel, R. & Bukstel, L. (1978). Memory Processes among Bridge Players of Different Expertise. *American Journal of Psychology* 91: 673-689.

Falkenberg, E. D., Hesse, W., Lindgreen, P., E., N. B., Han Oei, J. L., Rolland, C., Stamper, R. K., Van Assche, F. J. M., Verrijn-Stuart, A. A. & Voss, K. (1996). *A Framework of Information System Concepts.* IFIP, The FRISCO Report, ISBN 3-901882-01-4.

Fowler, M. (1997). *Analysis Patterns: Reusable Object Models*. Reading, MA: Addison-Wesley Publishing Company.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, MA: Addison-Wesley Publishing Company.

Goodell, H. & Traynor, C. (1997). End-User Computing. *SIGCHI Bulletin* 29(4): 84-86.

Gray, W. D., Spohrer, J. C. & Green, T. R. G. (1992). End-User Programming Language: The CHI'92 Workshop Report. *SIGCHI Bulletin* 25(2): 46-49.

Green, T. R. G. & Petre, M. (1996). Usability Analysis of Visual Programming Environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing* 7: 131-174.

Guadagno, N. S., Lane, D. M., Batsell, R. R. & Napier, H. A. (1990). The predictability of commands in a spreadsheet program. *Interacting with Computers* 2(1): 75-82.

Hassinen, K., Sajaniemi, J. & Väisänen, J. (1988). *Structured Spreadsheet Calculation*. 1988 IEEE Workshop on Languages for Automation, Computer Society Press, 129-133.

Hendry, D. G. & Green, T. R. G. (1994). Creating, Comprehending and Explaining Spreadsheets: a Cognitive Interpretation of What Discretionary Users Think of Spreadsheet Model. *International Journal of Human-Computer Studies* 40: 1033-1065.

Hoc, J. (1989). Do We Really Have Conditional Statements in Our Brains? In Soloway, E. & Spohrer, J. *Studying the Novice Programmer.* Hillsdale NJ, Lawrence Earlbaum Associates. 179-190.

Johnson, W. L. (1986). *Intension-based Diagnosis of Novice Programming Errors*: Morgan Kaufmann Publishers.

Lewis, C. & Olson, G. M. (1987). *Can Principles of Cognition Lower the Barriers to Programming*? Empirical Studies of Programmers: Second Workshop, Washington, D. C., Ablex Publishing Corporation, 248 - 263.

Linn, M. C. & Clancy, M. J. (1992). The Case for Case Studies of Programming Problems. *Communications of the ACM* 35(3): 121-132.

Martin, R. C., Riehle, D., Buschmann, F. & Vlissides, J. (1997). *Pattern Languages of Program Design 3*: Addison-Wesley.

McKeithen, K. B., Reitman, J. S., Rueter, H. H. & Hirtle, S. C. (1981). Knowledge Organization and Skill Differences in Computer Programming. *Cognitive Psychology* 13: 307-325.

McLean, E. R., Kappelman, L. A. & Thompson, J. P. (1993). Converging End-User and Corporate Computing. *Communications of the ACM* 36(12): 79-92.

Miller, G. A. (1956). The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *Psychological Review* **63**: 81-97.

Minsky, M. (1975). A Framework for Representing Knowledge. In Winston, P. H. *The Psychology of Computer Vision.* New York, McGraw-Hill.

Misner, C. W. & Cooney, P. J. (1991). *Spreadsheet Physics*: Addison-Wesley Publishing Company.

Napier, H. A., Lane, D. M., Batsell, R. R. & Guadagno, N. S. (1989). Impact of a Restricted Natural Language Interface on Ease of Learning and Productivity. *Communications of the ACM* 32(10): 1190-1198.

Nardi, B. A. (1993). *A Small Matter of Programming: Perspectives on End User Computing*. London: MIT Press.

Nardi, B. A. & Miller, J. R. (1990). *The Spreadsheet Interface: A Basis for End User Programming*. INTERACT'90, Elsevier Science Publishers B. V. (North Holland), 977-983.

Nardi, B. A. & Miller, J. R. (1991). Twinkling Lights and Nested Loops: Distributed Problem Solving and Spreadsheet Development. *International Journal of Man–Machine Studies* 34: 161-184.

Norman, D. A. (1986). Cognitive Engineering. In Norman, D. A. & Draper, S. W. *User-Centered System Design.* Hillsdale, NJ, Lawrence Erlbaum. 31-61.

Panko, R. (2000a). *Spreadsheet Research (SSR) WWW page.* http://panko.cba.hawaii.edu/ssr/. (5.1.2000).

Panko, R. R. (2000b). *What We Know About Spreadsheet Errors.* College of Business Administration, University of Hawaii, Electronic Paper, http://panko.cba.hawaii.edu/ssr/Mypapers/whatknow.htm (5.1.2000).

Parsons, J. & Wand, Y. (1997). Choosing Classes In Conceptual Modeling. *Communications of ACM* 40(6): 63-69.

Piersol, K. W. (1986). *Object Oriented Spreadsheets: The Analytic Spreadsheet Package*. Proceedings of OOPSLA'96, 385-390.

Reitman, J. (1976). Skilled Perception in Go: Deducing Memory Structures from Inter-response Times. *Cognitive Psychology* 8: 336-356.

Repenning, A. & Sumner, T. (1995). Agentsheets: A Medium for Creating Domain-Oriented Visual Languages. *IEEE Computer* 28(3): 17-25.

Rist, R. S. (1986). *Plans in Programming: Definition, Demonstration, and Development*. Empirical Studies of Programmers, Washington, DC: Ablex Publishing Corporation, 28-47.

Rist, R. S. (1989). Schema Creation in Programming. *Cognitive Science* 13: 389-414.

Rist, R. S. (1990). Variability in Program Design: The Interaction of Process with Knowledge. *International Journal of Man–Machine Studies* 33: 305-322.

Rist, R. S. (1991). Knowledge Creation and Retrieval in Program Design: A Comparison of Novice and Intermediate Student Programmers. *Human-Computer Interaction* 6: 1-46.

Rist, R. S. (1994). Search Through Multiple Representation. In Gilmore, D. J., Winder, R. L. & Detienne, F. *User-Centered Requirements for Software Engineering Environments.* Springer-Verlag.

Rist, R. S. (1995). Program Structure and Design. *Cognitive Science* 19: 507-562.

Saariluoma, P. & Sajaniemi, J. (1989). Visual information chunking in spreadsheet calculation. *International Journal of Man-Machine Studies* 30: 475-488.

Sajaniemi, J. (1989). Goals and Plans as a Basis for User Interfaces in Spreadsheet Calculation. In Pulliainen, K. & Sihvo, H. *West of East.* University of Joensuu. 129-140.

Sajaniemi, J. (2000). Modeling Spreadsheet Audit: A Rigorous Approach to Automatic Visualization. *Journal of Visual Languages and Computing* 11: 49-82.

Sajaniemi, J. & Pekkanen, J. (1988). An Empirical Analysis of Spreadsheet Calculation. *Software–Practice and Experience* 18(6): 583-596.

Sajaniemi, J. & Tukiainen, M. (1996). *Goals and Plans in Spreadsheets and Other Programming Tools*. PPIG'96, Ghent, Belgium, 114-122.

Sajaniemi, J., Tukiainen, M. & Väisänen, J. (1999). *Goals and Plans in Spreadsheet Calculation.* Dept. of Computer Science, University of Joensuu, Technical Report, A-1999-1.

Sajaniemi, J., Väisänen, J., Hassinen, K. & Tukiainen, M. (1992). *B-ASSET-92-kielen määrittely.* Department of Computer Science, University of Joensuu, Technical Report, B-1992-3.

Schank, R. C. (1982). *Dynamic Memory: A Theory of Learning in Computers and People*. Cambridge: Cambridge University Press.

Schank, R. C. & Abelson, R. P. (1977). *Scripts, Plans, Goals and Understanding*. Hillsdale, New Jersey: Lawrence Earlbaum Associates.

Scoville, R. (1994). Spreadsheets. In Knorr, E. *The PC Bible.* Berkeley, California: Peachpit Press.

Shneiderman, B. (1976). Exploratory Experiments in Programmer Behavior. *International Journal of Computer and Information Sciences* 5: 123-143.

Sloboda, J. (1976). Visual Perception of Musical Notation: Registering Pitch Symbols in Memory. *Quarterly Journal of Experimental Psychology* 28: 1-16.

Soloway, E., Bonar, J. & Ehrlich, K. (1983b). Cognitive Strategies and Looping Constructs: An Empirical Study. *Communications of the ACM* 26: 853-860.

Soloway, E. & Ehrlich, K. (1984). Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering* 10: 595-609.

Soloway, E., Ehrlich, K., Bonar, J. & Greenspan, J. (1982). What Do Novices Know About Programming? In Badre, A., Shneiderman, B. *Directions in Human/Computer Interaction.* Ablex Publishing Co, 27-54.

Spohrer, J. C. (1989). *MARCEL: A Generate-Test-and-Debug (GTD) Impasse/Repair Model of Student Programmers*. Ph. D. Thesis, Dept. of Computer Science, Yale University.

Spohrer, J. C., Pope, E., Lipman, M., Sack, W., Freiman, S., Littman, D., Johnson, L. & Soloway, E. (1985c). *Bugs In Novice Programs And Misconceptions In Novice*

Programmers. Proceedings of Computers In Education, Elsevier Science Publishers B. V. (North Holland), 543-552.

Spohrer, J. C. & Soloway, E. (1986a). *Alternatives to Construct-Based Programming Misconceptions*. In Mantei, M., Orbeton, P. (Eds) CHI'86 Human Factors in Computing Systems, Boston (Mass): ACM, 183-191.

Spohrer, J. C. & Soloway, E. (1986b). Novice Mistakes: Are the Folk Wisdoms Correct? *Communications of the ACM* 29(7): 624-632.

Spohrer, J. C., Soloway, E. & Pope, E. (1985a). A Goal/Plan Analysis of Buggy Pascal Programs. *Human-Computer Interaction* 1: 163-207.

Spohrer, J. C., Soloway, E. & Pope, E. (1985b). *Where The Bugs Are*. In Borman, L., Curtis, B. (Eds) CHI'85 Human Factors in Computing Systems, San Francisco: ACM, 47-53.

Tukiainen, M. (1996). ASSET: A Structured Spreadsheet Calculation System. *Machine-Mediated Learning* 5(2): 63-76.

Tukiainen, M. (2001). Comparing two spreadsheet calculation paradigms: An empirical study with novice users. *Interacting with Computers* 13(4): 427-446.

Tukiainen, M. & Sajaniemi, J. (1996). *Spreadsheet Goal and Plan Catalog: Additive and Multiplicative Computational Goals and Plans in Spreadsheet Calculation.* Department of Computer Science, University of Joensuu, Technical Report, A-1996-4.

Vlissides, J. M., Coplien, J. O. & Kerth, N. L., Eds. (1995). *Pattern Languages of Program Design 2.* Addison-Wesley.

Weaver, C. S., Hohmann, L., Ewing, K., Kafai, Y., Guzdial, M., Koziol, D. & Soloway, E. (1989). *The GPCeditor: Supporting the Learning and Using of Synthesis Skills.* Dept. of Electrical Engineering and Computer Science, University of Michigan, Tech. Rep.

Weiser, M. (1982). Programmers Use Slices When Debugging. *Communications of ACM* **25**(7): 446-452.

UNIVERSITY OF JOENSUU
DEPARTMENT OF COMPUTER SCIENCE
Report Series A

Goals and Plans in Spreadsheet Calculation

Jorma Sajaniemi, Markku Tukiainen and Jarmo Väisänen

Report A-1999-1

ACM H.1.2, H.5.2, D.m
UDK 681.3.02, 519.68
ISSN 0789-7316
ISBN 951-708-734-9

Goals and Plans in Spreadsheet Calculation

Jorma Sajaniemi, Markku Tukiainen and Jarmo Väisänen
saja@cs.joensuu.fi, mtuki@cs.joensuu.fi

University of Joensuu
Department of Computer Science
P.O. Box 111, FIN-80101 Joensuu, Finland

Abstract

Programming knowledge can be characterized in the form of goals and plans that describe what must be achieved and how this is done. We have conducted interviews of spreadsheet users and analyzed spreadsheet applications qualitatively. The analysis resulted in a set of basic programming goals and plans describing spreadsheet programming knowledge. This paper introduces a model for spreadsheet programming knowledge description and uses it to present the results of our analysis.

Keywords: Spreadsheet calculation, Goals and plans, Empirical studies of programming

Contents

1. INTRODUCTION......1

2. GOALS AND PLANS IN PROCEDURAL LANGUAGES......3

3. GOAL AND PLAN HIERARCHIES IN SPREADSHEETS AND PROCEDURAL PROGRAMS......5

4. DATA LAYOUT IN SPREADSHEETS......8

4.1. RATIONALE FOR THE MODEL......8
4.2. PROPERTIES OF SPREADSHEET SYSTEMS......9
4.3 THE DGP/S MODEL......11
4.4 REAL SPREADSHEETS IN THE DGP/S MODEL......12

5. COMPUTATIONAL GOALS AND PLANS......14

5.1 RATIONALE FOR THE MODEL......14
5.2 THE DGP/S MODEL......16

6. ANALYSIS OF ADDITIVE AND MULTIPLICATIVE GOALS AND PLANS......18

7. CONCLUSIONS......22

ACKNOWLEDGEMENTS......23
REFERENCES......24

1. Introduction

“Spreadsheets make it easy to do complex calculation–and even easier to do them incorrectly”
(Richard Scoville, PC-Bible, Chapter 15: Spreadsheets, p. 403, 1994)

Spreadsheet programs are probably the second most used type of application programs after word processing, and spreadsheets are used daily by millions of users. Some user studies, for example a study among American grocery manufacturers (McLean et al., 1993), have found spreadsheets the most utilized and significant application for their subjects. Although spreadsheet systems are widely used, the scientific literature is scarse (Hendry and Green, 1994). Moreover, the results obtained for the same research questions have varied dramatically, e.g., Brown and Gould (Brown and Gould, 1987) found that even 44 % of spreadsheets created by experienced users contained errors in formulae, but Nardi and Miller (Nardi, 1993, Nardi and Miller, 1990, Nardi and Miller, 1991) suggest that there are no problems in formulae usage.

Spreadsheet calculation has been said to be the success story of making programming easier (Lewis and Olson, 1987). The strength of spreadsheets have been contributed to familiar and concrete representation of data values and formulae, suppression of the inner world of traditional programming, automatic consistency maintenance, absence of control model, low viscosity, aggregate operations and immediate feedback. These virtues have been described in many publications of the spreadsheet model (Nardi and Miller, 1990). But the limitations of the spreadsheet model, i.e., the difficulty of debugging and redesign, and the lack of modularity and abstractions, have been mentioned less densely. In the literature of empirical studies of programming, we have seen many studies of different programming languages, such as Basic or Prolog, but only a few concerning spreadsheets or other end user programming languages (Gray et al., 1992).

There has been few controlled empirical studies of spreadsheet creation (Green and Navarro, 1995, Olson and Nilsen, 1987-1988) and learning (Hicks et al., 1991, Kerr and Payne, 1994). Some of the studies have compared different spreadsheet systems (Napier et al., 1989) (Baxter and Oatley, 1991) and some the usage of a spreadsheet system (Guadagno et al., 1990, Napier et al., 1992). The results have been varying, e.g., Baxter and Oatley found no difference in learning and using two systems, but Napier et al. did find a difference. There has also been studies of individual spreadsheet users at their work settings (Green and Hendry, 1993, Hendry and Green, 1994), and some studies concentrating on spreadsheet errors during development (Brown and Gould, 1987, Doyle, 1990, Lerch et al., 1989, Panko and Halverson Jr, 1994, Ronen et al., 1989). Panko (1997) has collected a web-site (Spreadsheet Research, SSR) to gather information of research on spreadsheet development, testing, use, and technology. The web-site contains, e.g., lots of references to spreadsheet errors in practical every day usage. The number of erroneous spreadsheets in practice is estimated by industry experts to be about 33 % of all spreadsheets. Saariluoma and Sajaniemi, and Sajaniemi’s research group at the University of Joensuu (1988a, 1988b, 1989, 1991, 1994, 1995) have studied spreadsheets from computational and cognitive perspectives. For more complete review on spreadsheet research see Hendry and Green (Hendry and Green, 1994).

In a problem solving activity, like computer programming, there is always an interaction of user’s cognitive structures (e.g. knowledge chunks) and cognitive processes (e.g. planning). Chunking is a well known psychological phenomenon of human memory

(Miller, 1956) that provides a way to go around the physical limitations of working memory. Experts have no better or larger working memory than novices, but they have much larger and more complex memory chunks. Planning has been defined as the predetermination of a course of action aimed at achieving a goal (Hayes-Roth and Hayes-Roth, 1978). This predetermination of a course of action in the experts of some domain have often been called tacit knowledge of the subject domain (Larkin et al., 1980, Polya, 1973). In the domain of procedural programming this tacit knowledge has been argued to appear in the form of plans which represent many of the stereotypic actions in programs (Ehrlich and Soloway, 1984).

The development of a spreadsheet application is a form of programming; especially it can be regarded as programming carried out by end-users. It consists of understanding the problem (i.e., the goals of the application), devising a plan to achieve the set of goals, and implementing the proposed plan. But it is not trivial to find out how the planning is done. Usually a user of a spreadsheet system has little or no knowledge of system design. Therefore, it is unrealistic to believe that planning in spreadsheet calculation is of stepwise refinement type, and nowadays stepwise refinement is considered unrealistic even in traditional programming (Pennington, 1987). Planning in spreadsheet design is perhaps more like opportunistic planning (Hayes-Roth and Hayes-Roth, 1978), i.e., the planner can have a plan existing at different level of abstraction simultaneously and he or she continually alternates between the levels. Whatever the process of planning is, the user must have cognitive structures for spreadsheet programming knowledge.

The purpose of this paper is to present a framework where spreadsheet knowledge can be studied, and to apply this framework in the analysis of a set of real spreadsheets. The framework was devised on the basis of our findings, and it makes a clear distinction between goals and plans. We will show what types of goals and plans we have discovered, and compare them to the goals and plans discovered in Pascal (Soloway and Ehrlich, 1984, Soloway et al., 1982, Spohrer et al., 1985). Our analysis will be limited to data and computations as they appear in spreadsheets, i.e., we will not considers other aspects of spreadsheet system usage like human-computer interaction in formula construction or the usage of macros and graphics.

Our work started with an analysis of 135 spreadsheets, out of which 101 contained formulae, used by Finnish business companies and governmental agencies (Sajaniemi and Pekkanen, 1988). Based on this analysis, we developed a model, the DGP/S model, that describes the structure and role of data structures, and computational goals and plans in spreadsheets. In this model, spreadsheets are composed of various variable structures and computational plans (like the sums of all values in all columns in a given structure). With this in mind, we set out to explore the spreadsheets and wrote an application to extract all connected computation graphs in spreadsheets. We then analyzed all these computation graphs by looking at the associated spreadsheets and by trying to figure out the intentions of the user. This process soon made us able to identify a collection of reoccurring computations.

The rest of this paper is organized as follows. In chapter 2, we will shortly characterize the ideas of goals and plans in procedural languages as described in Soloway's work. Chapter 3 compares some of these ideas of procedural programming with spreadsheet calculation. Then, in chapter 4, we will introduce the fist part of the DGP/S model by discussing data layout in spreadsheets and especially the ideas of physical and logical data structures. In the next chapter we will complete the introduction of the DGP/S model by describing more closely computational goals and plans and their relation to the data structures. In chapter 6, we will apply the model to describe the spreadsheet programming knowledge we have found in the example spreadsheets. Finally, chapter 7 contains conclusions.

2. Goals and Plans in Procedural Languages

The DGP/S model is based on a so-called *schema-based approach* to expert programming knowledge that emphasizes the role of semantic structures more than the role of syntactic structures (Detienne, 1990). This approach can be used to account for problem solving and understanding activities in programming tasks, e.g., for coding and debugging. The schema-based approach has been developed mainly by Elliot Soloway and his colleaques who conducted a series of studies of novice Pascal programmers' bugs and misconceptions. They argued that a large part of these errors and misconceptions could be explained by analyzing novices' programs in terms of experts' programming structures and by finding out the mismatches. According to them, expert Pascal programmers have and use high-level, plan knowledge that directs their programming activities (Soloway et al., 1982).

The notion of Soloway's programming plan comes from the text comprehension research notion of schema (Soloway and Ehrlich, 1984). A *plan* in Soloway's theory is a procedure or strategy in which the key elements of the process have been abstracted and represented explicitly. Soloway's group identified a set of plans (serving themselves as expert programmers) which they thought to be used in simple Pascal looping programs. Figure 1 presents a set of such basic plans: running-total-variable-plan that captures the notion of a variable responsible for gathering the total sum of individual items, division-by-zero-guard-plan that unifies the actions needed to prevent a division by zero error to occur, etc. The plans are divided in the theory into two main types: control flow plans representing, e.g., looping structures, and variable plans representing variable usage (Ehrlich and Soloway, 1984).

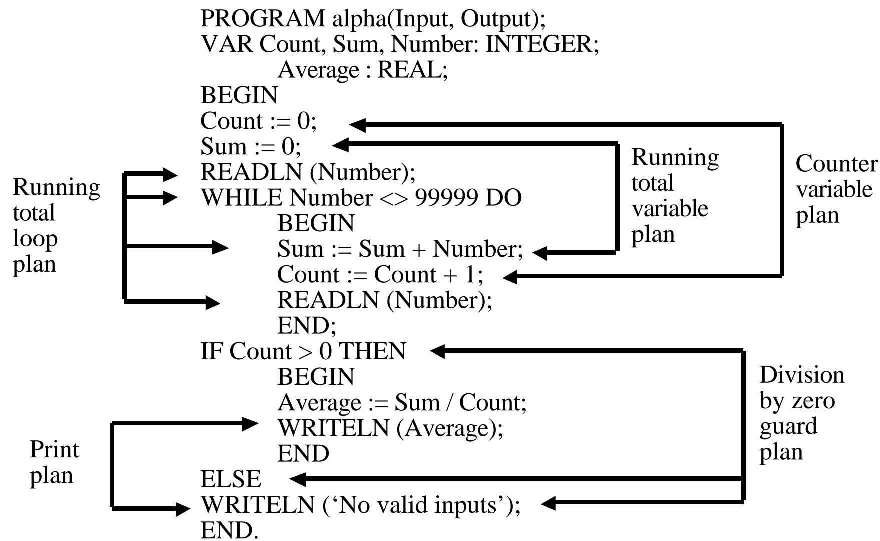

Figure 1: Soloway's basic programming plans (adapted from Curtis, 1988).

In addition to this plan knowledge, Soloway's group argued that expert programmers' knowledge includes "rules of programming discourse", i.e., rules that specify the conventions in programming. For example, the name of a variable should usually agree with its function, and programmers don't include in programs code that will never be executed. These rules specify what experts expect to find in programs and they are analogous to discourse rules in conversation.

Soloway's group validated the existence of this type of plan knowledge by giving experts and novices fill-in-the-blank programs which were either plan-like or unplan-like (i.e., some plan or discourse rule was violated). All programs missed a line of code or a fragment of a line, for example a comparison operator. It was argued that the appropriate plan will 'color' the program and therefore experts will fill in the plan-like answer. The results supported this hypothesis: with plan-like programs, experts performed significantly better than novices, while in the unplan-like case experts' performance dropped to essentially that of novices. In another experiment, experts were asked to recall programs. Again the programs were either plan-like or not. Results were similar to the fill-in-the-blank test: critical lines in plan-like programs were recalled earlier than those in unplan-like programs (Soloway and Ehrlich, 1984).

Gilmore and Green (1987, 1988) have argued that this type of programming plans may not be psychologically real outside Pascal. They studied Pascal and BASIC programmers' bug corrections by providing different types of bugs and different graphical cues to highlight plan and/or control structures in programs. They found out that Pascal programmers benefited of plan structure cues but BASIC programmers did not. According to Gilmore and Green, the experiment could suggest that BASIC programmers did not use programming plans but they concluded that BASIC programmers may use plans which, however, are harder to identify in BASIC than in Pascal. They argued that BASIC is a less "role-expressive" language, i.e., it is not as easy to understand the function or role of a particular statement in a BASIC program as it is in a Pascal program. They suggested also that although the concept of a programming plan may generalize across languages, the actual content of these plans does not.

Davies (1990) suggests that programming plans appear to reflect rather programmers' design related skills than overall skills in programming, or in a certain programming language. In a experiment similar to that of Gilmore and Green, Davies found out that a programmer's design skills had significant impact upon his or her ability to detect plan-related bugs and to use cues about plan structure. Competence in programming did not explain differences in ability to find bugs. Davies (1993) reviews literature emerging from studies concerned with skill acquisition and development of knowledge representation in programming, and he suggests that these studies provide a theory of plans, but say little about the activity of planning, i.e., about the way in which plans are developed and used.

Soloway's group used also a representation called *goal and plan tree* (GAP tree) to represent the space of possible correct solutions to a programming problem (Spohrer et al., 1985). The GAP tree representation makes an explicit distinction between *goals*, i.e., results that must be achieved, and *plans* that are ways to produce the results. A GAP tree starts with all the top-level goals a solution must accomplish and a set of alternative plans that will achieve these goals. Each plan may evoke new subgoals and each of these subgoals has one or more alternative plans, etc. The result is a tree with alternating levels of required goals and alternative plans.

The approach above is often called a schema-based approach, emphasizing the semantic structures of programming and contrasted to control-flow approach emphasizing the syntactic structures (Detienne, 1990). (There exists still further ways to decompose a

problem, for example GOMS modeling (Card et al., 1983)). In this paper, we do not discuss the control-flow approach to program understanding, although we agree that the schema-based approach is not the only approach used by experts. For example, rules of programming discourse seem to control the composition of plan-like Pascal programs but they do not account all of the syntactic structuring of these programs (Soloway and Ehrlich, 1984). This type of discourse rules can be found in all programming tools like spreadsheets or procedural programming languages. One of these programming discourse rules in spreadsheets could be that the visual structure should follow the computational structure. Saariluoma and Sajaniemi (1989) has showed that violating this rule can have serious affects on comprehending the spreadsheet application.

3. Goal and Plan Hierarchies in Spreadsheets and Procedural Programs

By looking at typical problems solved with spreadsheet calculation on one hand and with a procedural programming language, e.g., Pascal, on the other hand, it becomes clear that the problem sets are different. There are, of course, problems that can be solved easily with both tools, e.g., problems that have little input and output and contain non-iterative computations only.. But there exist lots of applications which suit better for one tool only (Casimir, 1992). This is due to the nature of the tools. For example, the problem of finding the shortest route between a set of towns, which is easy to solve in Pascal, is almost impossible to solve with spreadsheets as they lack iterative constructs. On the other hand, the budget of a small company consisting of text and numbers with some additions and subtractions is much easier to produce with a spreadsheet system than in Pascal.

Plan criticism has pondered the possible existence of “natural” plans: plans which would be presumably observed in non-programmers, and which will be mapped onto code structures, e.g., the natural plan for an average is a sum divided by a count (Bonar, 1985) (Davies, 1990). A clear distinction between goals and plans makes this problem easier to understand. A goal, say taking the average of a set of numbers, can be known by non-programmers but any plan to achieve this goal in a programming system is based on the feature repertoire offered by the tool and is, therefore, tool dependent. There are many ways to compute an average, and any of these plans can be realized only by tools that support every detail of the plan. One could argue that the programmer has a “natural” plan for the average: average is the total sum divided by the count. But the fact that an average can be computed in this way is not a plan at all - it is just the mathematical definition of average. In Pascal, an average may be computed in this way but it can be computed as a running average (new average = (old average * count so far + new item) / new count), also. On the other hand, most spreadsheet systems provide a standard function for computing an average. A spreadsheet programmer may indeed obtain the correct result without even knowing that the number of items has any effect on the result. Thus, it is fair to say that average, as a concept, is presumably observed in non-programmers and might henceforth be called a “natural”, tool independent goal, and that there exists many tool dependent plans for this goal. Some of the plans may obey the mathematical definition but it does not mean that they could be realized in all tools. Thus, goals may be natural but plans are tool dependent and hence no plan can be “natural.”

Let us next consider an example of a situation where the size of the effect of a minor change in a programmer’s intention, i.e., goal, varies remarkably between various programming tools. Suppose that the original goal is to compute the total sum of a set of individual numbers. Figure 2 shows typical solutions in Pascal and in some spreadsheet language. Suppose now that instead of the total sum the programmer wants to compute a

guarded sum that includes numbers satisfying some certain criteria, only. In Pascal, it is easy to add a guard to the expression which computes the sum as shown in Figure 2. In spreadsheet calculation, the programmer has to use some other area in the sheet to extract appropriate values. The programmer must even change the type of referencing in formulae because the named area reference to the original numbers (B_area) used in the total sum cannot be utilized by the guards selecting single values in the area.

	Pascal	Spreadsheet

Total Sum	sum:=0;
	for i:=1 to n do
	    sum:=sum+B[i];
		B8: @sum(B_area)
		where B_area is B1:B7

Guarded Sum	sum:=0;
	for i:=1 to n do
	    if A[i]='x' then
	    sum:=sum+B[i];
		C1: @if(A1='x',B1,0)
		C2: @if(A2='x',B2,0)
		:
		C7: @if(A7='x',B7,0)
		C8: @sum(C_area)
		where C_area is C1:C7

Figure 2: Total Sum and Guarded Sum in Pascal and in a spreadsheet language.

This example demonstrates clearly that the general programming notion of guarded sum has entirely different implementations in Pascal and in spreadsheets. It is, therefore, obvious that plans are tool dependent. However, similar tools have many common or similar plans. For example, the guarded sum plan of Pascal is equal to that of Algol 60, and with differences in syntactic notation similar to the programming language C. Thus, plans may be thought to be grouped according to the tools in which they can be utilized. Some plans can be applied in many tools and are, therefore, more general by their nature. Such plans can be thought to be part of general programming language applicable in many programming environments. Others are specific to a programming paradigm, e.g. looping plans can be utilized in procedural languages but not in functional languages. Finally, some plans are highly tool dependent and can be used in a single tool only. Thus, plan groups can be seen to form a generality hierarchy in which more generally applicable plans, i.e., programming plans, appear higher, and more specific language dependent plans appear lower. A programmer with expertise in programming in general and with expertise in Pascal programming may still have poor knowledge of spreadsheet plans.

Programming involves knowledge of different domains, at least those of the programming domain and the problem domain (Brooks, 1983). The problem domain description of the programming problem represents the desired goal or decomposition of this goal into several subgoals and forms the highest level of the goal (and plan) description of the problem. The programming domain knowledge can be further divided into knowledge of general programming concepts and knowledge of certain programming language or tool. In Soloway's research, programming domain knowledge is divided into strategic plans which represent language independent programming knowledge, tactical plans which are kind of semi-language independent programming knowledge, and implementation plans which are programming language dependent realizations of higher level plans (Soloway et al., 1982). This division of plans corresponds closely to our suggestion above, i.e., that plans form a hierarchy according to the number of tools they can be utilized in, and we might call it Soloway's plan generality hierarchy. Soloway's hierarchy is, however, a subset of our plan generalization hierarchy, because it treats procedural languages only. Soloway's all plans were developed for Pascal, and even higher level plans are such that they can be easily implemented in it.

Let us next consider how the various domains and plan hierarchies are reflected in GAP trees. The highest level of a GAP tree consists of the top-level goals, i.e., the result that the user wants to achieve with the program. Such goals belong, of course, to the problem domain. The next level consists of alternative plans for the top-level goals. These plans make the first step to structure the solution and they are quite general by their nature. But all decompositions of the problem may not be appropriate in every tool: even if the plans are general, some plans may not be applicable within the current tool and must therefore be discarded. Thus, the highest plan level contains plans that can be utilized in tools that are similar to the current tool, i.e., plans that belong to the higher levels of the plan generality hierarchy. As we descend the GAP tree, plans become more and more tool dependent, i.e., they appear at lower levels of the plan generality hierarchy, also. Finally, at the lowest level plans must be implemented directly within the tool.

Thus, in a problem decomposition goals on the highest (i.e., problem domain) level are the same for all problem solvers, but all plans are (more or less) language or tool dependent. At any problem decomposition level, the programmer knows what tool he or she uses and implicitly knows the advantages and constrains of the tool. For example, consider the sum of a set of values as the highest level goal, and the corresponding GAP trees in spreadsheet calculation on one hand and in Pascal on the other. In Pascal, we use a looping structure which “goes through” all the values in a set together with a variable to accumulate the sum. In Soloway’s hierarchy of plans, the strategic plan for this looping is either Read/Process (for the Pascal repeat and for loops) or Process/Read (for the Pascal while loop), and the tactical plan depends on which strategic plan was chosen. Similarly, the implementation plan depends on the chosen tactical plan. In spreadsheet calculation, the plan for the sum is to use the SUM function with the appropriate area reference as its argument. Thus, the GAP trees differ in all other respects but the uppermost (problem domain) goal.

Our approach in which all the plans are tool dependent opens a new dimension in our understanding of planning and plans in programming. Programming tools have a strong effect on the set of applicable plans, on the goal and plan hierarchy of the resulting program, and on the difficulty of the task of devising a program for a specific problem domain goal. It is therefore unfruitful to try to find the utilization of the same plans when different tools are used. But it should be possible to find the utilization of the same higher level plans when programmers use similar tools. It may, however, be that higher level plans are harder to recognize and their use is harder to validate in empirical research. But the lack of results demonstrating the use of higher level plans that apply in many tools is not a reason to underestimate the results concerning lower levels plans that can be utilized in a single tool only.

The goals and plans discussed in this paper are those of expert users. Novices’ plan generalization hierarchies are surely different, and their knowledge is fragmented. Bonar (1985) has discussed novice Pascal users’ knowledge and he introduced the term step-by-step natural language procedure knowledge to describe the kind of every day procedural knowledge (e.g., instructions for going to a certain place) that novices use when their Pascal plan knowledge is not sufficient to solve a problem. Novice spreadsheet users are likely to have and use similar knowledge. For example, we have seen a solution to the average problem in which the user has used counting to obtain the number of the values together with summation to obtain the result.

It has been argued (Davies, 1990) that if plans are used by programmers then the number of plans reflect programmers’ expertise in programming, and that such results have not been obtained. The plan generalization hierarchy, however, gives insight into the possible

kinds of expertise. An expert Pascal programmer may be a novice Fortran programmer, but as both languages are procedural, he has many higher level plans in common with expert Fortran programmers even though their lower level plans are different. If the Pascal programmer has never been introduced to spreadsheets he may have very few plans in common with expert spreadsheet programmers. These common plans are most likely high level but, coincidentally, they may be lower level plans, too. As a consequence, any empirical research studying plans in programming must recognize the plan generalization hierarchy and clearly state the level it operates on.

In this paper, we are interested in spreadsheet plans on the same level as Soloway's group described Pascal plans, i.e., on the lowest levels of plan generalization hierarchy. Consequently, plans described in this paper do not represent any general programming knowledge. What we do argue is that expert spreadsheet users use and recognize such plans. Spreadsheets also contain goals and plans which deal with other aspects than computations. We have found, for example, *descriptive* goals and plans which represent space reservations like titles and data labeling. Such goals and plans are not covered in this paper. Further, we will not consider other aspects of spreadsheet systems like macros or graphics.

4. Data Layout in Spreadsheets

In this chapter, we will present the way data is modeled in the DGP/S model. We will start by discussing first the basic rationale for the model deriving from users needs to model application domain data, and by describing secondly the associated properties of current spreadsheet systems. The third section introduces the model, and finally we will give some examples of real world spreadsheet descriptions in the model. The model is based on an analysis of a large number of real spreadsheets. This analysis is described more detailed in chapter 6.

4.1. Rationale for the Model

Spreadsheet systems are mostly used in applications where relations between distinct application domain data items can be described by mutually independent qualifiers (i.e., orthogonal indexes). For example, a collection of sales amounts can be qualified on one hand by the time scale and on the other hand by the product. Thus such data collections can be represented as complete n-dimensional cubes. Due to the limited expressive power of spreadsheet systems, spreadsheets are mostly used in cases where the number of dimensions is at most 2, and in some rare cases 3. Commercial systems which have utilized more than 3 dimensions, e.g. Lotus Improve, have not been proven succesful in use. *The DGP/S model reflects these facts by supporting 0, 1, and 2-dimensional data cubes only.*

Sometimes the qualifiers of a data collection are not totally independent. In the previous example, it may be the case that some products have not been available during the whole time period. Thus the resulting cube is not complete and some data items are missing. We shall later see that such situations end up in one compromise or other when represented in spreadsheets.

Let us consider more closely how a qualifier that is based on time can be used to structure the data items. Usually each time unit, e.g., a year, is constructed of the same number of smaller time units, e.g., months. In this case, the point of time can be thought of as a single qualifier (June 1998) or as two qualifiers (June and 1998). If the division into smaller units

is not the same for all the larger time units then the two qualifier interpretation results in an incomplete cube. For example, if the first qualifier is the month and the second is the number of a day, there is no data item for the combination April and 31. In both cases it is, however, safe to interpret the whole time interval as a single qualifier that has a structure of its own. The interval composed of months from January 1997 to December 1998 can be thought to consist of 24 months which can be grouped to form two years. Similarly, the interval composed of days from the beginning of January to the 15th of August can be thought of as a collection of days that can be grouped to form months having different lengths.

Thus, it is possible to assume in these cases that each has a single qualifier with some internal structure. However, it is not possible say that this interpretation is the only one. Especially, we cannot say that this interpretation is what the spreadsheet user has in her mind when she thinks of the sales figures.

The second qualifier in the previous example, i.e., the product, may also consist of groups of similar products. In contrast to the time scale, different product groups have very seldom the “same” products. For example, if the products are paint jars then a group might consist of all sizes of jars containing red paint, and other groups (i.e. jars containing blue paint) may have the same products, i.e., the same jar sizes. But if the products are bakery items, there hardly is any relationship between product items in the group of breads and the group of cookies.

Thus, it is clear that a single quantifier may have some internal structure that may be regular, as in the case of years and months, or more or less irregular. It is evident that this internal structure of qualifiers may be important for the application. *The DGP/S model supports internal structure of application domain qualifiers.*

4.2. Properties of Spreadsheet Systems

The goal of all application programming is to solve some problem in an application domain, and not just to develop an application that performs some computations. It is a well known phenomenon in software engineering application domain knowledge cannot be transferred directly to programming domain (Brooks, 1983). The writing of a computer program creates just a model of the application domain data. In application programming, it is a common situation that future users of a system will be application domain experts while the design and implementation will be done by programming domain experts. The success of a computer system is many times deemed by how well the designers and programmers have understood the application requirements (Curtis et al., 1988). Several program comprehension models have also noted the importance of understanding application domain functions, data, and data dependencies (see, e.g., Brooks (Brooks, 1983)). There is also evidence of using both representations at the same time. For example, Cobol and FORTRAN programmers attaining high levels of comprehension use a cross-referencing strategy between the program world and the domain world (Pennington, 1987).

In spreadsheet calculation, the application expert and the implementer of a system are usually the same person, but still an application is a model of the real world only. The implementer has to implement the application domain data in the spreadsheet using the spreadsheet system. As a result, she will maintain both representations, one in the application domain and the other in the spreadsheet domain, at the same time.
For example consider the following excerpt taken from a protocol where KM, the author of a spreadsheet number 046 (cf. chapter 6), describes a part of her spreadsheet. The interview was conducted by Sajaniemi (JKS).

> JKS: (...) explain me what this is, in your own words, (...) what this H62 is.
> KM: It is this sum which goes to the finance budget, the amount we need (...) for personnel wages plus the vacation extra so it is the amount we are budgeting for (...)
> JKS: So you are saying it is sum of the wages and the vacation extra.
> KM: Wages plus the vacation extra, yes.

KM describes the cell H62 to contain the sum of the wages and the vacation extra. The wages can be found in the cells H7..H57 and the vacation extra in the cells L7..M57. Thus, it is clear that KM considers for example the wages as a single unit i.e. a application domain structure. The protocol does not, however, make any reference to the actual layout of this data. The application domain data can be laid on the spreadsheet in any convenient way i.e. the application domain structure can be represented by many spreadsheet cell structures.

As another example consider an excerpt of a user protocol taken from Hendry and Green (Hendry and Green, 1994). The user Sue describes a simple task in an application domain which she is unable to solve in spreadsheet domain because she cannot transfer the application domain data structures to spreadsheet system structures. The problem is to compute averages of blocks of numbers, contained in a list of 500 records. Starting at row 1, the records are located so that the 10 blocks are found in rows 1-50, 51-100,..., 451-500, with data in columns A and B. User's goal is to compute the average of column B for each of the 10 blocks, and list the averages in column C.

> Sue: "... say that [in cell C1] I want the averages of [the first block] then it's nice and easy because all I have to do is [use the average function and create a range for it by selecting the first 50 rows of column B] ... I always think at this point, right, I've done it for one cell I ought to be able to copy [that formula down 10 rows and get the averages of the other blocks] ... but of course if you do that ... you wouldn't get what you wanted because [you would get the average of rows 1-50 in C1, 2-51 in C2, 3-52 in C3, and so on]."

The protocol goes on and Sue explains that this is a typical situation were facilities to copy formulae go wrong. The problem here is that Sue sees her data as a logical domain data structure of 10 blocks of 50 records each and she expects the spreadsheet system to perform the calculations according domain data structures. She feels that creating one range of first 50 rows should generalize over the whole 500 records structure and the computation should follow this scheme.

The DGP/S model reflects this distinction of application domain and programming domain in its data modeling mechanism.

Current spreadsheet calculation systems operate typically on spreadsheets which consist of individual cells. These cells contain primitive data and can be used in formulas and operations almost unrestrictedly. The only data structuring facility is grouping cells in form of rectangular areas that can be referenced as a single entity. Individual cells, however, form larger structures in the user's mind when he or she creates an application. Cells must be mentally chunked together for larger structures before spreadsheets can be comprehended and manipulated (Saariluoma and Sajaniemi, 1989). *The DGP/S model restricts its modeling capacity to represent programming domain data collections as rectangular spreadsheet areas in the programming domain.*

4.3 The DGP/S Model

The DGP/S model makes a distinction between *physical data structures* which refer to the surface layout of the spreadsheet, and *logical data structures* which refer to the conceptual data in the application and in the application domain. For example, the logical sequence of twelve elements (e.g., monthly sales) can be physically laid out as two lists containing six cells (two half years), or as four lists containing three cells (four year quarters), or as two lists containing five cells and one list containing two cells (because of the physical width of the screen), or in some other way.

Physical data structures describe the physical layout of related cells As commercial spreadsheet systems promote rectangular collections of cells it is no surprise that physical data structures found in real spreadsheets are rectangular. The need to include textual data (e.g., titles and headers) and intermediate results typically force users to build collections of separated rectangular areas. Logical data structures are structures inherent to the problem domain and they reveal the conceptual relations of individual data items on the logical level.

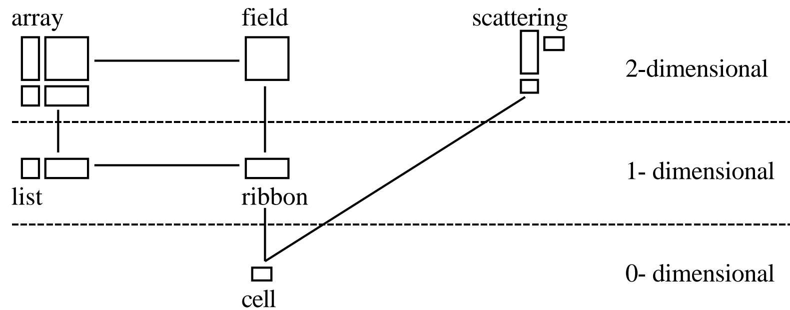

Figure 3: The physical data structures hierarchy.

Figure 3 shows the hierarchy of physical data structures. A *cell* is the smallest unit of all data structures. It is an individual element containing a text, a number or a formula, and occupying a certain location in the spreadsheet. A *ribbon* is a consecutive line of one or more cells located in one row or in one column. A *list* is a line of one or more ribbons, all located in the same row (in case of row-wise ribbons) or in the same column (in case of column-wise ribbons). The lengths of the individual ribbons in a list may be different. A *field* is a rectangular collection of one or more cells and can be considered as a line of row-wise ribbons or as a line of column-wise ribbons. An *array* is a collection of two or more fields that could be thought to be obtained from a larger field by leaving out some rows and/or columns. Thus an array can also be seen as a list of lists. Finally, a *scattering* is any (unstructured) collection of individual cells not covered by any of the other forms.

Logical data structures reveal the conceptual relations of individual data items on the logical level. The hierarchy of logical data structures is shown in Figure 4. A *chamber* is the simplest unit. It is an individual data element describing a text, a number or a value obtained through computations from other data. The only properties of a chamber are its identity (i.e., name) and contents. Especially, a chamber has no location information because, on the logical point of view, the positioning of a piece of information in a spreadsheet is unimportant per se. A *sequence* is an ordered, non-empty set of chambers. Thus, the elements in a sequence can be denoted by their index within the set. A sequence has no information of its position or direction (vertical vs. horizontal). A sequence may be *split*

into two or more parts. The lengths of the parts may vary within a sequence. Sequences describe 1-dimensional application domain data, and the splitting within a sequence models the internal structure of application domain qualifiers. For example, the sales figures within a year can be modeled as a sequence split into 12 parts consisting of 31, 28, 31, ... elements.

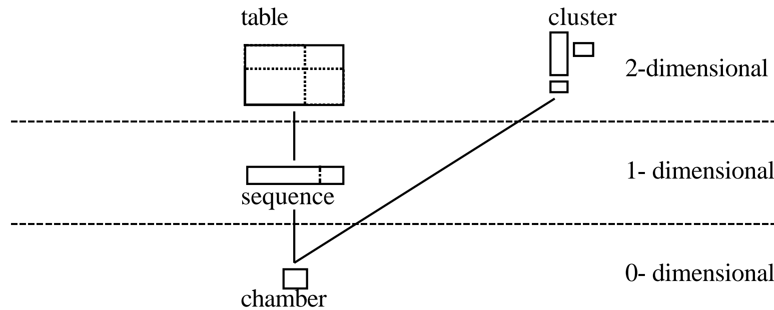

Figure 4: The logical data structures hierarchy.

A *table* is an ordered, non-empty set of similar sequences. Two sequences are similar if they have the same number of elements, or they are split into the same number of parts and the lengths of these parts are pair-wise equal. A table may be *split* into two or more parts of consecutive sequences. Similar to sequences, tables have no position or direction.

Finally, a *cluster* is a logical collection of elements that cannot be structured in any of the previous ways.

In the DGP/S model, each data collection found in a spreadsheet can be classified as an occurrence of some physical data structure, and as an occurrence of some logical data structure. The correspondence between these two forms is created by the user that has created the spreadsheet. Theoretically, there is no formal connection between physical and logical data structures. For example, a sequence may be represented as a scattering, and a cluster may be realized as a ribbon. In practice, however, there exists a strict correspondence between physical and logical data structures. Users generally tend to choose physical data structures that represent logical data structures in a straightforward way. As a consequence, users can mentally process the application area and its representation at the same time which is a prerequisite for efficient working. A logical sequence is nearly always located in a single row or column, i.e., represented as a physical ribbon or a list. A ribbon is more natural for the representation of a sequence, but split sequences are often represented as a physical list of ribbons. Tare usually represented as fields or arrays.

4.4 Real Spreadsheets in the DGP/S Model

In this section, we will describe parts of real spreadsheets with the DGP/S data modeling mechanism. The physical structure of cell collections in spreadsheets made by other users can be revealed by considering data layout and formulae. Logical structures cannot be identified as easily because the role of individual cells must be explained in terms of the application, i.e., the application must be fully understood if logical structures are to be found.

A B

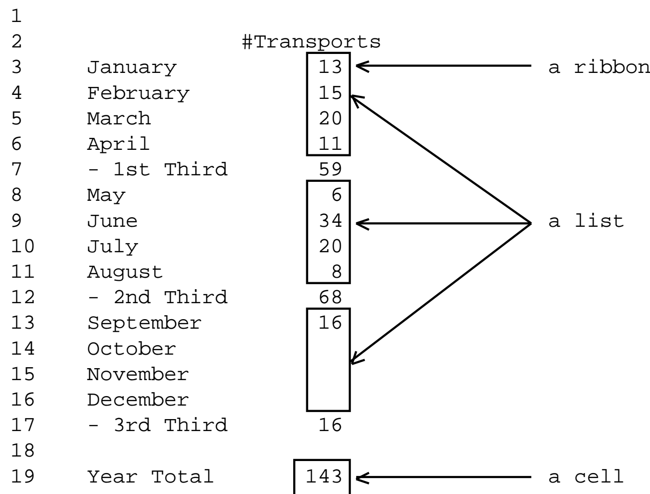

Figure 5: The physical data structures of a spreadsheet 067.

Consider the spreadsheet for the calculation of monthly transports in Figure 5. From the physical point of view, the areas B3..B6, B8..B11 and B13..B16 containing the numbers of transports in year thirds are ribbons. They form up a list that contains transports for the whole year. The cells B7, B12 and B17 calculate transports in each year third, and the cell B19 sums them up. Thus, the cells B7, B12 and B17 form a list of three one-cell ribbons.

From the logical point of view, the three ribbons represent a sequence split into three parts. If the three sums of the thirds were missing, then the sequence would not be considered to be split. In this case, the physical division to three ribbons would be a layout consideration only. But because the three sums do exist, the division to the three thirds is essential to the application domain, also. Thus, it must be reflected in the logical structures, i.e., the ribbons are representations of three parts of the sequence. The cells B7, B12 and B17 represent a non-split sequence, even though its physical representation consists of three separate areas. Finally, the cell B19 represents a chamber.

F	G	H	I	J	
7	200	160	360	90	450
:	:	:	:	:	:
57	100	100	200	50	250

Figure 6: A part of the spreadsheet 046.

Logical data structures cannot always be found by investigating the spreadsheets only. Consider the example in Figure 6. The cells in the column H are sums of the cells in columns F and G on the same row, e.g., H7 contains the formula +F7+G7. Similarly, the cells in the column J are obtained from the values in the previous two columns, e.g., the cell J7 contains the formula +H7+I7.

The non-formula areas can be explained by logical data structures in two ways. Either there is a table of three sequences (F7..F57, G7..G57, I7..I57) or there is a table (F7..G57) of two sequences together with another sequence (I7..I57). By looking at the spreadsheet, it is impossible to say which of these interpretations is the correct one.

The protocol, however, reveals that the latter interpretation is correct:

> KM: (...) So there (the column H) is this wage plus the social security fee. They are the amounts (...).(...) it (J) is the money that is obtained from wages account (H) plus then this estimated pension (I) we need.

The protocol describes that there is a table (F7..G57) that contains sums of money actually paid, and a sequence (I7..I57) that contains estimated sums that are only used as a basis for further calculations.

5. Computational Goals and Plans

We will now describe the way computations are described in the DGP/S model. We will start with a rationale that explains the starting points of the model.

5.1 Rationale for the Model

Although current spreadsheet systems have almost no provision for structuring computations, spreadsheets do contain groups of data items and formulae that have some structure. These structures are created by users who have written the spreadsheets

As an example, Figure 7 contains a section of the spreadsheet 047. The cell E15 contains the formula +E14+B15, the cell F15 contains +F14+C15, and the cell H15 contains +E15-F15.

	B	C	D	E	F	G	H
9				From the beginning of the year			
10				------------------------------			
11	Planned	Actual		Planned	Actual		Difference
12	10	8		10	8		2
13	15	14		25	22		3
14	15	13		40	35		5
15	15	15		55	50		5
16	10	11		65	61		4
17	15			80			
18	13			93			
:	:			:			

Figure 7: A part of the spreadsheet 047.

The following excerpt is taken from a protocol where KM, the author of the spreadsheet, describes the above section.

> JKS: (...) Tell me in your own words, without looking at the formula, what is in the cell H15. So what is that number in the cell (...)?
> KM: It is the difference of the planned and actual figures from the beginning of the year.
> JKS: And how is it calculated?
> (...)

> KM: We calculate it so that we take the planned and subtract the actual, and it has been either positive or negative in our reports, but now it is almost always positive here. But it describes whether we are going up or down.

The protocol shows that KM can describe the cell H15 in two different ways. First, she describes the meaning of the cell in a general way that makes no reference to the various cells used in the formula. This meaning of the cell can be considered as the goal of this cell. Secondly, KM describes the computation expressed by the formula. This can be considered as a plan for the implementation of the goal. The planned and actual figures from the beginning of the year are not, however, input data, and must first be computed. Thus the plan involves further subgoals with various plans to achieve them. *The DGP/S model makes a distinction between goals that describe results to be achieved, and plans that describe ways to compute such results. A goal may have many alternative plans, and a plan may involve many subgoals.*

In chapter 3, we found that lower level goals and plans are tool dependent and do concern application domain goals only indirectly. In chapter 4, we saw further that logical data structures correspond to application domain data, and physical data structures correspond to programming domain data representations of logical structures. Thus, computational goals are usually related to logical data structures while their implementations, i.e., plans, are related to physical data structures. These relationships are presented in Figure 8.

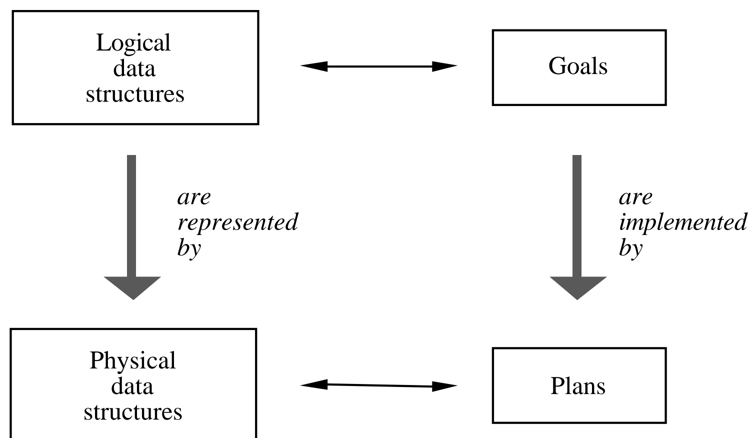

Figure 8: The relationships between data structures, goals and plans.

In spreadsheets, the application domain goals are, however, simple, and hence practically all goals have a direct application level interpretation. Further, spreadsheet users tend to use subgoals that deal with data that has a clear meaning in the application domain, rather than operating with tricky computations using artificial data structures. As a consequence, lower level goals deal with application domain data much more frequently than in procedural programming.

For example, in the protocol above, KM does not describe in detail the whole chain of dependent cells. She does not say that H15 contains the formula +E15-F15, neither does she describe the formulae in the cells E15 and F15. Thus, in this context, these details are

not important to KM. A possible explanation is that KM does not see the goals and plans as properties of individual cells but as properties of larger structures. Especially, KM states both the goal and the plan in terms of application domain data. This phenomenon is not surprising in spreadsheets where lower level goals and plans usually deal with application domain data: it is easier for users to state plans in terms of application data than its possibly mixed representation. *The DGP/S model expresses goals and plans in terms of logical data structures.*

Logical data structures have no direction information: a sequence can be represented vertically or horizontally, and a table has no notion of rows or columns. Some goals may be applied to two-dimensional structures, i.e., tables, to produce a 1-dimensional structure in either of the dimensions, with different results. For example, consider a table describing daily amounts of product deliveries. Total daily deliveries of all products form a sequence, and so does the total deliveries of each product. The table may be represented with days in rows and products in columns, or vice versa. *For simplicity of expression, the DGP/S model uses the terms row-wise and column-wise to describe that goals and plans yield different results depending on the way two-dimensional data structures are indexed.*

In chapter 6, we will see many frequently used plans which accomplishes a set of general computational goals in spreadsheet applications These reoccurring computations concern quite basic and elementary operations, i.e., they occur at the lowest levels of plan generalization hierarchy. They are obtained by studying 101 spreadsheets. With a much larger number of spreadsheets to be studied, there would possibly be larger reoccurring structures to be found. On the other hand, spreadsheet applications are mostly very simple, and hence their goals are simple, too. Thus, this level of goals and plans describes a large portion of all computations within spreadsheets. *The DGP/S model does not state where it can be applied, but it has been used to describe goals and plans in lower levels of the plan generalization hierarchy, only.*

5.2 The DGP/S Model

The DGP/S model uses a special notation to describe *goals* that describe results to be achieved and *plans* that describe ways to compute such results, both at the level of general spreadsheeting knowledge and at the level of a spreadsheet language. Formula templates are written at an abstract spreadsheet programming language. In this language, actual cell references have been abstracted to refer to logical data structures. When a computation refers to a single cell we will use term chamber and write the reference as Cham1, Cham2, etc. Sequences are named as Seq1, Seq2, etc. and tables are denoted by Tab1, Tab2, etc. In some computations, structure types are overloaded, i.e., a chamber can act as a sequence or a sequence can play the role of a table. When a computations can be applied to all types of structures symbols Input and Output are used.

Goal descriptions have the following slots:

G: Goal name reflecting the intension of the goal

D:Description of the goal in the form “Output = Goal_name(Input)” together with a verbal description. This description gives an overview of the intentional aspects of the goal. Output tells the type of the data structure that can result from an application of this goal. Input tells the types of the data structures that can act as input for this goal (if needed).

P: Name of a plan that fulfill the intensions of this goal. Usually the name is formed from the goal name by using an extension starting with the underline character "_". In such a case, the goal name may be left out of this occurrence of the plan name. This slot may occur several times in a goal description.

E: Examples of actual spreadsheets in which the goal has been used

Plan descriptions have the following slots:

P: Plan name reflecting the computation performed by this plan

D:Description of the plan in the form "Output = Plan_name(Input)")" together with a verbal description. This description gives the computational aspects of the plan. Output tells the type of the data structure that can result from an application of this plan. Input tells the types of the data structures that can act as input for this plan (if needed).

T: Template for the computation written at the abstract spreadsheet programming language. Templates are described more thoroughly below.

C: Constrains concerning spatial or size requirements for the application of the plan, e.g., that input structures must be of equal size

S: Spatial variations that produce different results: row-wise or column-wise.

O:Observations and other remarks

R: Required subgoals, if the computation uses other goals

E: Examples of actual spreadsheets in which the plan has been used

A *template* gives the formula which lies in the element(s) of the output structure. The formula is applied to all elements of the output structure once, i.e., over all indexes. If the computation needs to refer specific elements of the input structures, these are denoted as Input.c_i where i is the index of the element. The index n is used to refer to the last element of a structure, e.g., Input.c_n. If all the formulae in the output must be listed separately, then the output elements are given in the form Output.c_n: +Input.c_n. The abstract language may be supposed to contain built-in functions like @SUM and @AVE for sums and averages respectively.

As an example, Figure 9 defines two goals and three plans. The goal TotalSum can be implemented by many plans, one of which is TotalSum_direct. This plan has as its subgoal the SumClause which can be implemented by the operator + or the function SUM.

```
G: TotalSum
        D: c1 = TotalSum(Input)
           TotalSum sums up all the elements in input
           structure to the output chamber
```

```
        P: _direct
           _splitted
           _cumulative
        E: SS049

P: TotalSum_direct
        D: Description: c1 = TotalSum_direct(Input)
           TotalSum_direct sums up all the values of elements
           in the input structure directly
        T: c1: SumClause(Input)
        E: SS049

G: SumClause
        D: c1 = SumClause(Input)
           SumClause sums up all the elements in input
           structure to the output chamber with a
           single addition
        P: _addition
           _sumFunction
        E: SS049

P: SumClause_addition
        D: Description: c1 = SumClause_addition(Input)
           SumClause_addition sums up all the values
           of elements in the input structure
           using the addition operator
        T: c1: +Input.c1+...+Input.cn
        E: SS049

P: SumClause_sumFuntion
        D: Description: c1 = SumClause_sumFuntion(Input)
           SumClause_sumFuntion sums up all the values
           of elements in the input structure
           using the sum function
        T: c1: SUM(Input)
        E: SS090
```

Figure 9: An example of goals and plans using DGP/S notation.

6. Analysis of Additive and Multiplicative Goals and Plans

We have applied the DGP/S model in analysis of goals and plans in real spreadsheets. We analyzed 101 spreadsheets used in Finnish business companies and governmental agencies. (See Sajaniemi & Pekkanen (1988) for a more detailed description of this data.) We were interested in additive and multiplicative computations. A computation is called *additive* if it contains the operators + and -, and the functions SUM and DSUM only. Similarly, a *multiplicative* computation contains the operators * and /, and the functions AVG and DAVG. Such computations cover 91 % of all operators and 27 % of all functions in the 101 spreadsheets.

The sizes of the spreadsheets varied column-wise from 4 to 256, with an average of 38,5 and row-wise from 11 to 1830, the average being 141,4. The most often used function was IF making 32,8 % of all function uses, and ROUND with the usage of 22 %. The function IF was used mostly to prevent the occurrence of division by zero, and to “pretty-printing”, i.e., to prevent a computation totally if some of the referenced values was zero.

We wrote a program to extract connected formula graphs. In formula graphs nodes represent spreadsheet cells, directed edges represent formula references and connected formula graphs are maximal-size connected subgraphs. For example, Figure 10 shows a formula graph describing an artificial computation consisting of addition and multiplication.

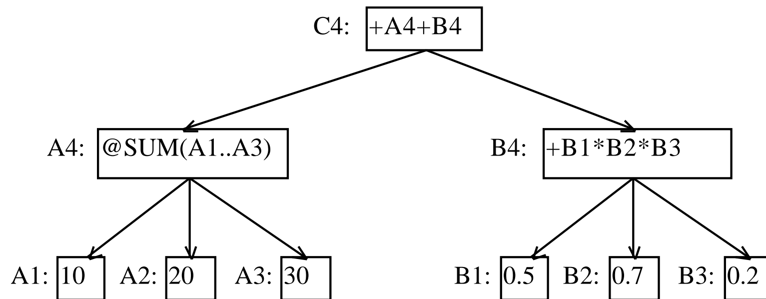

Figure 10: A connected formula graph.

Our analysis program extracted connected formula graphs, their sizes in cells and tried to match spatially equivalent components. The results of the analysis program offered a good starting point for further analysis. We analyzed all these formula graphs by hand, looked for reoccurring computational structures, and tried to figure out the intentions of the programmer. With this method, we soon found out typical data structures (e.g., tables and sequences of cells) and a collection of computational goals (e.g., cumulative sum of a set of numbers) together with plans to achieve these goals (e.g., a certain set of formulae in consecutive cells).

We classified the goals to four groups: application specific goals, general goals, formula goals and merged goals. *Application specific goals* are related to specific application, i.e., the computations are not applicable across different applications. The main goals of each application are not of interest here, because we were looking for more general goals. *General goals* are goals which can be found (and applied) in many applications. *Formula goals* are implicit goals that are used only at the level of formulae. Finally, *merged goals* are goals that try to achieve two or more goals at the same time.

As an example of a complete analysis of one of the spreadsheets (090) that is a typical one in the set of spreadsheets analyzed. It is a financial application showing the assets, operation, net capital employed and annual change in net capital employed in a certain company (Figure 11). The spreadsheet is basically divided into four sections, one per each financial item. We have changed the name of the company and all the constant values in the sheet to preserve anonymity.

	A	B	C	D	E
1	COMPANY LTD				
2					
3	FINANCIAL STATEMENT				
4		85/86	84/85		
5		K$	K$		
6	ASSETS				
7					
8	Income financing				
9	Operating margin	6263.9	13928		
10	Investment subsidy	1976.8	2363.7		
11	Other income	=1501.3-1397	=1956.4-1345.8	← Calculator	
12	Net interests	=-2409.3+1229.6	=-2030.5+2259.3		
13	Taxes	-3581.6	-3228.1		
14		- - - - - - - - - -	- - - - - - - - - -		
15	Income total	=SUM(B9:B14)	=SUM(C9:C14)	← IndexSum	
16	Capital assets sales	6434.8	5180.4		
17	Long-term debt increase	=1115+2000+2065	5966 ←	Calculator	
18	Valuation item change				
19		- - - - - - - - - -	- - - - - - - - - -		
20		=SUM(B15:B19)	=SUM(C15:C19)	← IndexSum	
21					
22					
23	OPERATION				
24					
25	To investments				
26	Buildings	2182.7	2235.9		
27	Machinery	=6885.4+2158.7-3295.3	=8963.3-1353.4+4293.4	← Calculator	
28	Bonds	6287.6			
29	Other	=1229.6+2222.7	=38.5+7.9+6.6	← Calculator	
30		-	-		
31	Investments total	=SUM(B26:B30)	=SUM(C26:C30)	← IndexSum	
32	Long-term debt decrease	=-(3880.9-3799.3)+B17	2849		
33	Payment of dividends	490	300		
34		- - - - - - - - - -	- - - - - - - - - -		
35		=SUM(B30:B34)	=SUM(C30:C34)	← IndexSum	
36	Net margin change	=B62	=C62	← Duplication	
37		- - - - - - - - - -	- - - - - - - - - -		
38		=B35+B36	=C35+C36	← IndexSum	
39		==========	==========		
40					
41	CHANGE IN NET CAPITAL EMP				
42					
43	Cash and funds	=B54-C54	=C54-D54		
44	Short-term financial assets	=B55-C55	=C55-D55	← DifferencePairwise	
45	Floating assets	=B56-C56	=C56-D56		
46	Short-term foreign capital	=B57-C57	=C57-D57		
47		- - - - - - - - - -	- - - - - - - - - -		
48		=SUM(B43:B47)	=SUM(C43:C47)	← IndexSum	
49					
50					
51					
52	NET CAPITAL EMPLOYED				
53					
54	Cash and funds	8667.4	7393.9	3195.6	
55	Short-term financial assets	=33732.1-B54	=26421.2-C54	=11900.8-D54	← Substraction ByConstant
56	Floating assets	9687.9	8634.7	3269.9	
57	Short-term foreign capital	-23090.2	-19050.9	-12299.8	
58		- - - - - - - - - -	- - - - - - - - - -	- - - - - - - -	
59	Net capital 28.2.	=SUM(B54:B58)	=SUM(C54:C58)	=SUM(D54:D58)	← IndexSum
60	Net capital 1.3.	=C59	=D59	←	Duplication
61		- - - - - - - - - -	- - - - - - - - - -		
62		=B59-B60	=C59-C60	←	DifferencePairwise
63					

Figure 11: An example of goal/plan analysis of the spreadsheet 090.

There are two accounting periods: 84/85 and 85/86. Assets (rows 9 to 20) and operations (rows 26 to 38) contain constant values from different sources of information, e.g., bank accounts, loans etc. All the computations refer only to numbers within the accounting period, so all the references in formulae are within the column used. Since the net capital employed (rows 54 to 62) depends on two fiscal years, the user has added a third column

in order to have the correct sum of the end value of last accounting period's net capital employed. In other words, the 84/85 net capital employed formula in the cell C62 is the difference of the cells C59 and C60, where the cell C59 is the value of net capital employed at 28th of February 1985, and the cell C60 is the value of net capital employed at 1st of March 1984. Change in the net capital employed (rows 43 to 48) is the difference of the values of the two accounting periods, so the references in the formulae refer to two columns.

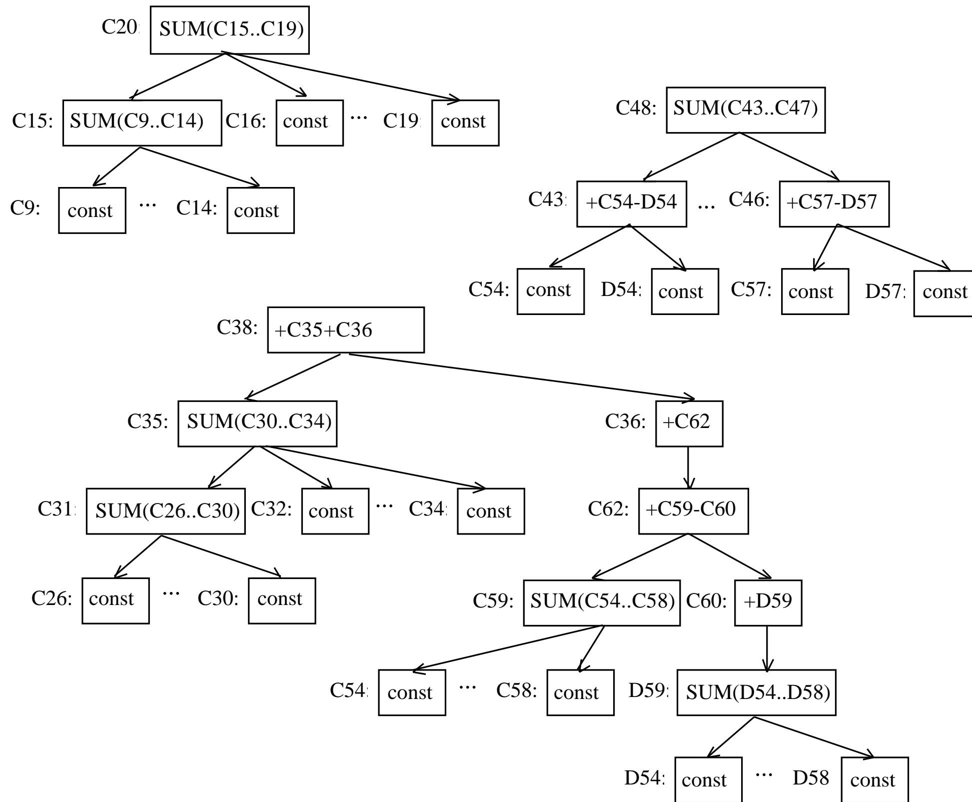

Figure 12: Connected components of the spreadsheet 090.

Figure 12 shows the connected components analysis program extracted from the spreadsheet 090. As a matter of fact, the analysis program did connect the components Assets (root cell B20) and Operations (root cell B38) in the column B because the cell B32 contains a calculator formula which refers to the cell B17. This is incidental and happens only at the accounting period 85/86. The cell B17 is for long-term debt increase and the cell B32 is for long-term debt decrease. We suppose that the company decided to pay it's debt during 85/86 and the author of the spreadsheet found a convenient way to have the amount of debt added to the debt decrease calculator by using duplication.

After running the analysis program we studied the application by hand. We started at the root of each of the connected components, determined the physical and logical data structures and the goals and the plans by looking at the references of the formulae and the semantic information in the labels. Some of the goals were easier to discover than others. In the application 090, we found five different general goals and six different plans. Appendix C contains detailed descriptions of these goals and plans.

The Calculator goal is applied at the cells B11, C11, B12, C12, B17, B27, C27, B29, C29, and B32, the IndexSum goal is applied in the areas B15..C15, B20..C20, B31..C31, B35..C35, B38..C38, B48..C48, and B59..D59, the Duplication goal occurs in the areas B36..C36 and B60..C60, the DifferencePairwise goal in the areas B43..C46 and B62..C62, and the SubtractionByConstant goal in the area B55..D55.

Figure 13 shows the number of applications which included at least one instance of the goal in question. The overall number of different general goals was 39 and overall number of different general plans was 47. All additive and multiplicative goals and plans that we found are listed in the Appendix A and the Appendix B respectively. A more detailed presentation of these goals and plans is given by Tukiainen and Sajaniemi (1996).

TotalSum
IndexSum
DifferencePairwise
Duplication
PercentagePairwise
PercentageProportion
SumPairwise
MultiplyPairwise
DividePairwise
ProportionByPercentage

Figure 13: Application frequencies of ten most common goals.

7. Conclusions

We have shown how the schema-based approach used in procedural programming research can be extended to the domain of spreadsheet calculation. Using the goal and plan approach developed by Soloway et al. we have extracted a collection of reoccurring computations in spreadsheet calculation and identified the user's intentions behind these computations. We have analyzed 101 spreadsheet applications using this framework and identified the goals (user's intentions) and the plans (realizations at the level spreadsheet formula language) used (Tukiainen and Sajaniemi, 1996).

We have also proposed a DGP/S model of spreadsheet calculation which supposes a set of computational goals and plans used in spreadsheet calculation and makes a distinction between physical and logical data structures. This model is derived from user studies and properties of actual spreadsheet applications.

The results presented in this paper can be utilized in future spreadsheet systems to make them reflect more closely users' cognitive structures. For example, Hassinen & al. (1988) have introduced the notion of structured spreadsheet calculation that brings logical data structures and goals to the user-interface of a spreadsheet system. This mechanism has

been further developed by Tukiainen (Tukiainen, 1996). Hassinen (1994) has projected a tool that can automatically find possible errors in spreadsheets. The basic idea of this tool is to search for logical data structures that seem to be in contradiction with the corresponding physical structures. Sajaniemi (1998) has formalized the definition of the connection between computational goals and physical data structures, and introduced an Excel macro that visualizes these connections in spreadsheets.

Acknowledgements:

We wish to thank Kari Hassinen for his deep insights about logical and physical data structures. This work has been partially supported by the Academy of Finland.

References

Baxter, I. & Oatley, K. (1991). Measuring the learnability of spreadsheets in inexperienced users and those with previous spreadsheet experience. *Behaviour & Information Technology* **10**(6): 475-490.

Bonar, J. G. (1985). *Understanding the bugs of novice programmers*. Ph.D. Thesis, Computer and Information Science, University of Massachusetts.

Brooks, R. (1983). Towards a Theory of the Comprehension of Computer Programs. *International Journal of Man-Machine Studies* **18**: 543-554.

Brown, P. S. & Gould, J. D. (1987). An Experimental Study of People Creating Spreadsheets. *ACM Transactions on Office Information Systems* **5**(3): 258-272.

Card, S. K., Moran, T. P. & Newell, A. (1983). *The Psychology of Human-Computer Interaction*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Casimir, R. J. (1992). Real programmers don't use spreadsheets. *ACM SIGPLAN Notices* **27**(6): 10-16.

Curtis, B., Krasner, H. & Iscoe, N. (1988). A Field Study of the Software Design Process for Large Systems. *Communications of the ACM* **31**(11): 1268-1287.

Davies, S. P. (1990). The nature and development of programming plans. *International Journal of Man–Machine Studies* **32**: 461-481.

Detienne, F. (1990). Expert Programming Knowledge: A Schema-based Approach. In Hoc, J.-M., Green, T. R. G., Samurcay, R. & Gilmore, D. J. *Psychology of Programming*. London, Academic Press : 205-222.

Doyle, J. R. (1990). Naive users and the Lotus interface: a field study. *Behaviour & Information Technology* **9**(1): 81-89.

Ehrlich, K. & Soloway, E. (1984). An Empirical Investigation of the Tacit Plan Knowledge in Programming. In Thomas, J. C. & Schneider, M. L. *Human Factors in Computer Systems.* Norwood, New Jersey, Ablex Publishing Corporation : 113-133.

Gray, W. D., Spohrer, J. C. & Green, T. R. G. (1992). End-User programming Language: The CHI'92 Workshop Report. *SIGCHI Bulletin* **25**(2): 46 - 49.

Green, T. R. G. & Hendry, D. G. (1993). CogMap: a Visual Description Language for Spreadsheets. *Journal of Visual Languages and Computing* **4**: 35-54.

Green, T. R. G. & Navarro, R. (1995). *Programming Plans, Imagery, and Visual Programming*. INTERACT'95, Lillehammer, Norway, London: Chapman and Hall.

Guadagno, N. S., Lane, D. M., Batsell, R. R. & Napier, H. A. (1990). The predictability of commands in a spredsheet program. *Interacting with Computers* **2**(1): 75-82.

Hayes-Roth, B. & Hayes-Roth, F. (1978). *Cognitive Processes in Planning.* Rand Corporation, Tech. Report, R-2366-ONR.

Hendry, D. G. & Green, T. R. G. (1994). Creating, comprehending and explaining spreadsheets: a cognitive interpretation of what discretionary users think of spreadsheet model. *International Journal of Human-Computer Studies* **40**: 1033-1065.

Hicks, J. O. J., Hicks, S. A. & Sen, T. K. (1991). Learning spreadsheets: human instruction vs. computer-based instruction. *Behaviour & Information Technology* **10**(6): 491-500.

Kerr, M. P. & Payne, S. J. (1994). Learning to use a spreadsheet by doing and by watching. *Interacting with Computers* **6**(1): 3-22.

Larkin, J., McDermott, J., Simon, D. & Simon, H. (1980). Expert and Novice Performance in Solving Physics Problems. *Science* **208**: 140 - 156.

Lerch, F. J., Mantei, M. M. & Olson, J. R. (1989). *Skilled Financial Planning: The Cost of Translating Ideas into Action*. Human Factors in Computing Systems CHI'89, Austin, Texas, ACM Press.

Lewis, C. & Olson, G. M. (1987). *Can Principles of Cognition Lower the Barriers to Programming*. Empirical Studies of Programmers: Second Workshop, Washington, D. C., Ablex Publishing Corporation.

McLean, E. R., Kappelman, L. A. & Thompson, J. P. (1993). Converging End-User and Corporate Computing. *Communications of the ACM* **36**(12): 79-92.

Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capasity for processing information. *Psychological Review* **63**: 81-97.

Napier, H. A., Batsell, R. R., Lane, D. M. & Guadagno, N. S. (1992). Knowledge of Command Usage In a Spreadsheet Program. *DATA BASE* **23**(1): 13-21.

Napier, H. A., Lane, D. M., Batsell, R. R. & Guadagno, N. S. (1989). Impact of a Restricted Natural Language Interface on Ease of Learning and Productivity. *Communications of the ACM* **32**(10): 1190-1198.

Nardi, B. A. (1993). *A Small Matter of Programming: Perspectives on End User Computing*. London: MIT Press.

Nardi, B. A. & Miller, J. R. (1990). *The Spreadsheet Interface: A basis for End User Programming*. INTERACT'90, Elsevier Science Publishers B. V. (North Holland).

Nardi, B. A. & Miller, J. R. (1991). Twinkling lights and nested loops: distributed problem solving and spreadsheet development. *International Journal of Man–Machine Studies* **34**: 161-184.

Olson, J. R. & Nilsen, E. (1987-1988). Analysis of the Cognition Involved in Spreadsheet Software Interaction. *Human-Computer Interaction* **3**(4): 309-349.

Panko, R. R. & Halverson Jr, R. P. (1994). *Patterns of Errors in Spreadsheet Development.* Decision Science, College of Business Administration, University of Hawaii, Tech. Report, .

Pennington, N. (1987). Comprehension Strategies in Programming. In Olson, G. M., Sheppard, S. & Soloway, E. *Empirical Studies of Programmers: Second Workshop.* Norwood, New Jersey, Ablex Publishing Corporation : 100-113.

Polya, G. (1973). *How to solve it*. Princeton, NJ: Princeton University Press.

Ronen, B., Palley, M. A. & Lucas, H. C. J. (1989). Spreadsheet Analysis and Design. *Communications of the ACM* **32**(1): 84-93.

Saariluoma, P. & Sajaniemi, J. (1989). Visual information chunking in spreadsheet calculation. *International Journal of Man-Machine Studies* **30**: 475-488.

Sajaniemi, J. & Pekkanen, J. (1988). An Empirical Analysis of Spreadsheet Calculation. *Software-Practise and Experience* **18**(6): 583-596.

Soloway, E. & Ehrlich, K. (1984). Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering* **10**: 595-609.

Soloway, E., Ehrlich, K., Bonar, J. & Greenspan, J. (1982). What Do Novices Know About Programming. In Badre, A., Shneiderman, B. *Directions in Human/Computer Interaction.* Ablex Publishing Co. .

Spohrer, J. C., Soloway, E. & Pope, E. (1985). A Goal/Plan Analysis of Buggy Pascal Programs. *Human-Computer Interaction* **1**(2): 163-207.

Tukiainen, M. (1996). ASSET: A Structured Spreadsheet Calculation System. *Machine-Mediated Learning* **5**(2): 63-76.

Tukiainen, M. & Sajaniemi, J. (1996). *Spreadsheet Goal and Plan Catalog: Additive and Multiplicative Computational Goals and Plans in Spreadsheet Calculation.* Dept. of CS, University of Joensuu, Technical Report, A-1996-4.

Appendix A: Basic goals and their descriptions

Goal	Description
AdditionByConstant	adds a constant value to the cells of input sequence. This is not Calculator because it is used for forecasting purposes
Average	calculates an average values of the original cells
ComputeTitles	computes titles out of other numeric year titles
CorrectionByIndexClause	changes the values in input structure to the corrected values of output structure (e.g., changes currency values according one base year) by dividing with base value and multiplying by current year value
CumulativeDifference	sums cumulatively over all the cells of two input sequences and then takes a pair-wise difference of these cumulated sequences to the output sequence
CumulativeSum	sums over all the cells of the input sequence to the output sequence
DecreaseByPercentage	decreases values in input sequence by percentages in another input sequence
DifferenceBetweenPairs	takes a difference between two consecutive cells in input sequence so that first cell in output sequence yields a difference between first and second cell in input sequence
DifferencePairwise	subtracts cells of one sequence from cells of another sequence pair-wise
DividePairwise	divides values of cells of two input structures pair-wise
DivisionByConstant	divides values in input sequence by a constant value
Duplicates	duplicates a value in the input chamber to the output structure
Duplication	duplicates values of the input structure to the output structure
FormatToHundreds	divides the cells of input sequence by 100. Used to format vehicle’s usage kilometers to hundreds
FormatToHours	divides the cells of input sequence by 60. Used to format minutes to hours
FormatToThousands	divides the cells of input sequence by 1000
IncompleteDuplication	duplicates some but not all values of the input sequence to the output sequence
IncreaseByPercentage	increases the values of cells of input sequence with a percentage proportion constant in chamber
IndexSum	sums up all the cells over the indexes of one dimension of the input table to the output sequence
MultiplyByConstant	multiplies the original cells by constant
MultiplyPairwise	multiplies values of cells of two input structures pairwise
Negation	changes the values of the input structure to their negations to the output structure
NumberSeries	cumulates a constant value of 1 to the original cells
PartialTotalSum	sums up all the cells in input structure to the output chamber and then subtracts one of the cells out of this total sum
PartlySum	sums a constant size portions of input sequence
Percentage	gives a percentage of a cell value from total of input structure
PercentageDecrease	gives a percentage decrease rates between two input sequences
PercentageGrowth	gives a percentage growth rate from two input sequences
PercentagePairwise	gives a percentage proportion between pairs of cells of two input sequences
PercentageProportion	gives a percentage proportion between pairs of cells of a input sequence and a chamber
ProportionByPercentage	gives a proportion from input sequence by a constant cell percentage
ProportionPartition	gives a proportion parts of input chamber by proportions in input sequence, eg. input chamber is to be divided to parts which are proportions in input sequence
RepairFormatToThousand	repairs FormatToThousand’s division by multiplying values by 1000
StepwiseIncrement	increases the values of cells of sequence with a constant value in chamber
SubtractionByConstant	subtracts a constant value from the original cells
SubtractionFromConst	subtracts the referenced cell value from a constant

SumPairwise	sums cells of at two sequences pair-wise
TotalSum	sums up all the cells in input structure to the output chamber
UnitProportion	gives a proportion of a measure by another measure, e.g., marks/hour, kg/h or income/capita etc.

Appendix B: Basic plans and their templates

Plan	Template
AdditionByConstant	T: +Seq1.cn+Constant
Average_ byHand	T: +TotalSum(Input)/count(cells in Input)
Average_default	T: +average(Input)
ComputeTitles	T: +Seq1.cn-1900
CorrectionByIndexClause	T: +Input.cn*Cham1/Cham2
CumulativeDifference	T: +DifferencePairwise(Seq1, Seq2)
CumulativeSum_default	T: +Seq1.cn+Seq2.cn-1
DecreaseByPercentage	T: +Seq2.cn*(1-Seq1.cn/100)
DifferenceBetweenPairs	T: Seq2.c1: +Seq1.c2-Seq1.c1
DifferencePairwise	T: +Seq1.cn-Seq2.cn
DividePairwise_default	T: +Input1.cn/ Input2.cn
DivisionByConstant	T: +Seq1.cn/Constant1
Duplicates_default	T: Output.c1: +Cham1,Output.cn: +Output.cn-1
Duplication_default	T: +Cham1
FormatToHundreds	T: +Seq1.cn/100
FormatToHours	T: +Seq1.cn/60
FormatToThousands	T: +Seq1.cn/1000
IncompleteDuplication	T: Seq2.c1: +Seq1.c1
IncreaseByPercentage	T: +Seq1.cn*(1+Cham1/100)
IndexSum_default	T: SumClause(Tab1.index1)
IndexSum_splitting	T: SumClause(Tab1.index1)
MultiplyByConstant	T: +Cham1*Constant1
MultiplyPairwise_default	T: +Input1.cn* Input2.cn
Negation_default	T: -Input.cn
NumberSeries_default	T: Output.cn: +Output.c(n-1)+1
PartialTotalSum	T: +TotalSum (Input)-Input.cx
Percentage	T: +Input.cx/SUM(Input)*100
PercentageDecrease	T: +(Seq1.cn-Seq2.cn)/Seq1.cn*100
PercentageGrowth	T: +Seq2.cn/Seq1.cn*100-100
PercentageGrowth_usingPF	T: +(Seq2.cn- Seq1.cn)/Seq1.cn
PercentagePairwise	T: +Seq1.cn/Seq2.cn*100
PercentagePairwise_usingPF	T: +Seq1.cn/Seq2.cn
PercentageProportion	T: +Seq1.cn/Cham1*100
PercentageProportion_usCR	T: +Seq1.cn/Cham1*Constant1
ProportionByPercentage	T: +Seq1.cn*Cham1/100
ProportionByPercentage_CR	T: +Seq1.cn*Cham1/Constant1
ProportionPartition	T: +Cham1/Cham2*Seq1.cn
RepairFormatToThousands	T: +Input.cn*1000
StepwiseIncrement	T: +Cham1+Seq2.cn-1
SubtractionByConstant	T: +Cham1-Constant1
SubtractionFromConstant	T: +Constant1- Duplication(Cham1)
SumPairwise	T: +Seq1.cn+Seq2.cn
TotalSum_cumulative	T: +Input.cn+Seq2.cn-1
TotalSum_default	T: SumClause(Input)
TotalSum_splitting	T: SumClause(Input)
UnitProportion	T: +Input1.cx/Input2.cx

Appendix C: Goals and plans in the spreadsheet 090

Goal: Calculator
D: Output = Calculator (Input)
Calculator calculates value of a cell using constant operands and basic operators +,-,* and /.
P: _default
E 001, 090, 075

Plan: Calculator_default
D: Output = Calculator_default(Input)
Calculator_default calculates value of a cell using constant operands and basic operators +,-,* and /.
T: No particular form
O: No other operators or functions.
E: 001, 090

Goal: IndexSum
D: Seq1 = IndexSum (Tab1)
IndexSum sums up all the cells over the indexes of one dimension of the input table to the output sequence.
P: _default
P: _splitting
E: 002, 046, 050, 062

Plan: IndexSum_default
D: Seq1 = IndexSum_default (Tab1)
IndexSum_default sums up all the cells over one of the indexes of the input table to the output sequence
T: SumClause(Tab1.index1)
S: Rowwise
S: Columnwise
E: 002, 047, 050

Goal: Duplication
D: Output = Duplication (Input)
Duplication duplicates values of the input structure to the output structure. Input and Output are of equal size.
P: _default
E: 024, 030, 033, 046, 047

Plan: Duplication_default
D: Cham2 = Duplication_default (Cham1)
Duplication_default duplicates value of the input cell to the output cell
T: +Cham1
E: 024, 030, 033

Goal: DifferencePairwise
D: Seq3 = DifferencePairwise (Seq1, Seq2)
DifferencePairwise subtracts cells of Seq2 from cells of Seq1 pairwise
P: _default
E: 049, 090, 065, 066

Plan: DifferencePairwise_default

D: Seq3 = DifferencePairwise_default (Seq1, Seq2)
DifferencePairwise_default subtracts cells of Seq2 from cells of Seq1 pairwise

T: +Seq1.c_n-Seq2.c_n

E: 049, 090, 065

Goal: SubtractionFromConstant

D: Output = SubtractionFromConstant (Input)
SubtractionFromConstant subtracts the referenced cell value from a constant.

P: _default

E: 090

Plan: SubtractionFromConstant_default

D: Cham2 = SubtractionFromConstant_default (Cham1, Constant)
SubtractionFromConstant_default repairs a value by subtracting a duplicate from the original constant

R: Duplication(Cham1)

T: +Constant1- Duplication(Cham1)

E: 090

3. ASSET: A STRUCTURED SPREADSHEET CALCULATION SYSTEM

Machine-Mediated Learning 5(2), 63-76, Lawrence Erlbaum Associates, 1996.

Email me for copies: Markku.Tukiainen@joensuu.fi

4. Comparing Two Spreadsheet Calculation Paradigms: An Empirical Study with Novice Users

Interacting with Computers 13(4), 427-446, Elsevier Science B.V., 2001.

Email me for copies: Markku.Tukiainen@joensuu.fi

Goals and Plans in Spreadsheets and Other Programming Tools

Jorma Sajaniemi and Markku Tukiainen
saja@cs.joensuu.fi, mtuki@cs.joensuu.fi
University of Joensuu
Department of Computer Science
P.O. Box 111, 80101 Joensuu
Finland

EXTENDED ABSTRACT

1. Introduction

Spreadsheet calculation has been said to be the success story of making programming easier (Lewis & Olson, 1987). The strength of spreadsheets has been contributed to familiar and concrete representation of data values and formulae, suppressing the inner world of traditional programming, automatic consistency maintenance, absence of control model, low viscosity, aggregate operations and immediate feedback. These virtues have been described in many publications concerning the spreadsheet model (e.g., Nardi & Miller, 1990; Napier & al., 1990), but there are serious limitations, also. For example, the difficulty of debugging or redesign, and lack of modularity and abstractions, make spreadsheet creation a challenge (Hendry & Green, 1994; Panko & Halverson, 1994; Brown & Gould, 1987).

Although the spreadsheet model has no provision for structuring data or computations, spreadsheets created within this model do contain groups of data items and formulae that have some structure. These structures are caused by programmers (or, more conventionally, users) who have created the spreadsheets. This view follows the ideas of Soloway & al., who described structures they found in Pascal programs in terms of goals and plans (e.g., Soloway & al., 1982; Spohrer & al., 1985). With this in mind we set out to explore actual spreadsheets to see if there are reoccurring uses of computations. We did this by writing a program to extract connected formula graphs from spreadsheets and by analyzing 101 spreadsheets used by Finnish business companies and governmental agencies. (See Sajaniemi and Pekkanen (1988) for a more detailed description of this data.)

We went through all these formula graphs, looked for reoccurring computational structures, and tried to figure out the intentions of the programmer. With this method we soon found out a collection of computational goals (e.g., cumulative sum of a set of numbers) and plans to achieve these goals (e.g., a certain set of formulae in consecutive cells). The purpose of this paper is to clarify main concepts of programming knowledge and study the role of languages and other tools in this knowledge. This is done by presenting examples of the goals and the plans we have found and by comparing those to the goals and the plans discovered in Pascal (e.g., Ehrlich & Soloway, 1984; Spohrer & al., 1985).

2. The Hierarchy of Goals and Plans

In the domain of programming expert tacit programming knowledge has been argued to appear in the form of plans which represent many of the stereotypic action sequences in programs (Ehrlich & Soloway, 1984). This notion of programming plans comes from the

text comprehension research notion of schemas (Soloway & Ehrlich, 1984). The plan in this thinking is a procedure or strategy in which the key elements of the process have been abstracted and represented explicitly. This approach is often called schema-based, emphasizing the semantic structures of programming and contrasted to control-flow approach emphasizing the syntactic structures (Detienne, 1990; Rist, 1989).

Programming involves knowledge of different domains, at least those of the programming domain and the problem domain (Brooks, 1983; Bonar, 1985). The problem domain description of a programming problem represents the desired goal, or decomposition of this goal into several subgoals, and forms the highest level description of the problem. This composition can be expressed as a goal and plan tree (GAP tree) (Spohrer & al., 1985) where every second level describes (sub)goals to be fulfilled and every other level describes alternative plans for each of the goals. A plan implements the goal in terms of programming constructs and further subgoals. Each program that solves the programming problem obeys a pruned tree obtained from the GAP tree by selecting one special plan for each goal.

Programming knowledge can be divided into knowledge of general programming concepts and knowledge of a certain programming language or tool. Soloway & al., (1982) make a distinction between strategic plans which represent language independent programming knowledge, tactical plans which are kind of semi-language independent programming knowledge, and implementation plans which are programming language dependent realizations of higher level plans. Soloway & al. have studied plans more accurately in the lower level of this hierarchy, i.e., plans in Pascal programming. They identified a set of plans (serving as expert programmers) they thought was used in simple Pascal looping programs. The plans were divided into two main kinds: control flow plans representing looping structures and variable plans representing variable usage (Ehrlich & Soloway, 1984).

It has been argued that the notational view of Soloway's plans is too narrow in its perspective and that plans are artifacts of the particular programming language used (Gilmore & Green, 1987; Davies, 1990, 1993). This criticism is based on the following arguments. First, contents of plans have not been generalized to languages other than Pascal and thus it is argued that plans cannot describe general programming knowledge. Second, BASIC programmers have been found to be unable to benefit from cues to plan structure which is claimed to suggest that BASIC programmers do not employ plan-based representations of programs and that programming plans may not even be psychologically real outside Pascal. Third, the development and use of plans are affected by learning experiences and it is argued that they therefore cannot be considered solely as natural strategies. Fourth, although the existence of plans may be used to differentiate novices from experts, intermediate programmers appear to have the same range and number of plan structures as experts.

This criticism seems to be partially based on some misconceptions. First, the distinction between programming knowledge and program knowledge seems to be forgotten. Programming knowledge contains plans that can be used in potentially any program whilst program knowledge concerns some specific program. These two types of knowledge are connected but they are not the same. Programming plans have more general nature and they must be adapted to the requirements of other plans when they are used in some specific program. Programming knowledge seems to contain separate shallow plan hierarchies. Program knowledge, on the contrary, contains one deep plan hierarchy starting from the main goal of the program and extending to the simplest plan at the lowest level. Shallow programming plan hierarchies appear all around the deep program plan hierarchy but they are now adjusted to fit the details of the program. Many papers on plans fail to make this distinction and they claim to study properties of programming knowledge while they are actually studying program knowledge.

Second, unplan-like constructs are critical information for programmers. For example, the plan-like version of initialization for running total is to assign 0 for the variable holding the running total. If a program initializes this variable to -999999 then the reader of this program is alarmed and tries to find a justification for such an exceptional initialization. He may concentrate attention to find the reason and, presumably, remember this exception much better than a plan-like use of a variable. Programming plans are different from schemas of everyday life. It does not matter whether there are books as opposed to other things on a shelf. Therefore, a visitor may recall the existence of non-existent books if the schema of the room includes books (Brewer & Treyens, 1981). But every single detail of a program must be correct if the program is to work. One might expect that given the opportunity to study a program in detail, a programmer will remember the exceptional features that make the program function correctly. Thus, it is not at all clear that plan-like structures would be better remembered than unplan-like ones.

Third, the concept of goals has been often neglected. Goals tell what must be accomplished and plans describe how they can be realized. Plans are language or tool dependent by their very nature. If the tool does not allow some specific construct then a plan cannot use that construct. Goals, on the other hand, can be tool independent or tool dependent. The highest goal of a program represents the desired effect of the program and is tool independent. Lower level subgoals can be tool dependent because they use constructs of the tool or because they appear only in subgoals of plans specific to this tool. Whereas the highest goal of a program belongs to the problem domain, the highest goals of programming hierarchies belong to the programming domain. Plan criticism has pondered the possible existence of "natural" plans, also. The distinction between goals and plans makes this question easier. Goals can be "natural," say taking the average of a set of numbers, but any plan to achieve this goal is tool dependent and hence cannot be "natural" (see chapter 5). Many problems involving plans can be solved by considering the combination of plans and goals instead of just plans.

Fourth, it seems unjustified to claim that plans can describe programming expertise only if they are tool independent. An expert programmer who has used Pascal may find little use of his previous knowledge when he starts to program in another language, for example in the string-processing language Icon (Scholtz & Wiedenbeck, 1990). Nevertheless, he would be considered an expert (Pascal) programmer. If he starts programming in Fortran, he may use some of his programming knowledge because Fortran and Pascal obey the same programming paradigm, i.e., they both are procedural languages. Therefore, we would expect to find programming paradigm dependent programming knowledge, also. But maybe this is just the part of the tool dependent programming knowledge that is common to both tools. The call for general programming knowledge seems to stem from strict interpretation of previous literature. But the psychological significance of plans does not diminish even if different tools result in different plans.

Fifth, the programs used in the experiments are too small to provide ecologically valid results. There is no need to utilize plan knowledge when one has to find errors in programs having some twenty lines. These programs have typically 3-5 variables, some of which are so related that they can be chunked as a single entity. It is then easy to simulate the actions of the program in short-term memory. One has to remember the values of the variables, the current point of execution (remembered by sight), and the starting lines of loops to be utilized at the end of each loop. Moreover, indentation helps in this work because it makes it unnecessary to remember the starting lines of loops as these can be found easily at the end of a loop by letting eyes to roll up to the first line having matching indentation. It is no wonder that indentation helps in this simulation process because it makes short-term memory load smaller. But real programs consist of many pages of program text and dozens

of variables. It is impossible to simulate their execution without external resources and therefore other strategies must be applied in program comprehension and error detection (Littman & al., 1986; Pennington, 1987). Small programs lead to unnatural strategies in experiments resulting in doubtful conclusions.

Sixth, the way plan cues were introduced may not be proper. Indentation is a common way to introduce cues about control structure and one might suppose that the utilization of indentation has become an automatic process requiring only a small amount of attention. Special coloring schemes to introduce cues about plan structure are new to subjects and thus, in spite of practice sessions, their utilization may require more attention resulting in poorer activity. Moreover, coloring may make the program look uncontinuous (especially if colored letters instead of a colored background is used). If different coloring schemes have different effects then other mechanisms for cues should be found.

Finally, the effect of learning experiences does not mean that plans describe natural strategies. Experts may have programming knowledge that is unaffected by teaching but novices and intermediates can acquire plans by themselves (Rist, 1989), or through teaching. The learning material may be descriptions of high-level plans (see for example Gamma & al. (1994) for paradigm dependent goals and tool dependent plans) or general design methods yielding simply a better program chunking ability.

In the following, we will describe spreadsheeting goals and plans. Our analysis here will be limited to computations and we will not consider other aspects of spreadsheet systems like macros or graphics. These goals and plans were found in real-life spreadsheets and thus describe methods utilized by people in their normal work. Our goals and plans describe spreadsheeting knowledge, i.e., knowledge that can be used in potentially any spreadsheet, and concerns the low-level end of goal and plan hierarchy. A certain spreadsheet, on the other hand, has its own goal and plan structure. This structure uses the given goals and plans adapted to the requirements of the specific spreadsheet. We argue that spreadsheeting knowledge contains such goals and plans, but it contains other parts as well. For example, the merging of spreadsheeting plans and their adaptation to each other demands other skills not described by the kind of goals and plans given below.

3. An Example of Goal and Plan Structure

We have recorded some protocols of people explaining their spreadsheet applications. These protocols show evidence of the type of goals and plans we are interested in. As an example consider Figure 1. It contains a part of spreadsheet number SS047. The cell E15 contains the formula +E14+B15, the cell F15 contains +F14+C15, and the cell H15 contains +E15-F15.

	B	C	D	E	F	G	H
9			From the beginning of the year				
10			--------------------------------				
11	Planned	Actual		Planned	Actual		Difference
12	10	8		10	8		2
13	15	14		25	22		3
14	15	13		40	35		5
15	15	15		55	50		5
16	10	11		65	61		4
17	15			80			
18	13			93			

:   :   :

Figure 1: Cumulative difference in a spreadsheet.

Let us consider the following excerpt taken from a protocol (translated from Finnish) where KM, the author of the spreadsheet, describes it.

```
JS:  (...) Tell me in your own words, without looking at the
     formula, what is in cell H15. So what is that number in the
     cell (...)?
KM:  It is the difference of planned and actual from the beginning
     of the year.
JS:  And how is it calculated? (...)
KM:  We calculate it the way we take the planned and subtract
     the actual and it has been either positive or negative in our
     reports, but now it is almost always positive here. But it
     describes whether we are going up or down.
```

The protocol shows that KM can describe the cell H15 in two different ways. First she describes the meaning of the cell in a general way that makes no reference to the various cells used in the formula. This meaning of the cell can be considered as the goal of this cell. Then KM describes the computation expressed by the formula. This can be considered as the plan for the implementation of the goal.

In the protocol, KM does not describe in detail the whole chain of dependent cells. She does not say that H15 contains the formula +E15-F15, neither does she describe the formulae in the cells E15 and F15. One could argue that these details are not important to KM. She does not see her goals and plans as properties of individual cells but as properties of larger structures.

4. Computational Goals and Plans in Spreadsheets

The goals and plans we are going to present here represent lower level plans that are explicitly represented at the level of the programming language used (see Figure 2). Our descriptions do not consider the spatial layout of cells if it has no effect on the computations. For example, the sum of a row-wise or column-wise sequence of cells, and even a rectangular area, can be calculated with the same formula.

Goal	Short description
Duplication	Duplicates a value or values
SplittedSums	Group of sums of parts of related data
TotalSum	Sum of related data
CumulativeSum	Cumulative sequence of sums of parts of related data
UnitProportion	Value divided by another value giving proportion
Percentage	Value shoving percentage proportion of data

Figure 2: Examples of goals

We will present some example goals and plans. Tukiainen and Sajaniemi (1996) give a more detailed listing of additive (i.e., containing operators + and - and functions SUM and DSUM) and multiplicative (i.e., containing operators * and / and functions AVG and DAVG) goals and plans. Additive and multiplicative computations cover 91 % of all operators and 26,9 % of all functions used in the spreadsheets (Sajaniemi & Pekkanen, 1988). In the following, goal descriptions and plan descriptions have following parts:

```
G: Goal_name
        D: Description: Output = Goal_name(Input)
           Verbal description of intention
        P: Plan names
        R: Restrictions
        E: Examples
P: Plan_name
        D: Description: Output = Plan_name(Input)
        T: Template
                C: Constrains
                S: Spatial variations
        O: Observations
        R: Restrictions
        E: Examples
```

Figure 3 defines two goals and three plans. The goal TotalSum can be implemented by many plans, one of which is TotalSum_direct. This plan has as its subgoal the SumClause which can be implemented by the operator + or the function SUM.

```
G: TotalSum
        D: c1 = TotalSum (Input)
           TotalSum sums up all the cells in input
           structure to the output chamber
        P: _direct
           _splitted
           _cumulative
        E: SS049

P: TotalSum_direct
        D: Description: c1 = TotalSum_direct(Input)
           TotalSum_direct sums up all the values of cells
           in the input structure directly
        T: c1: SumClause(Input)
        E: SS049

G: SumClause
        D: c1 = SumClause(Input)
           SumClause sums up all the cells in input
           structure to the output chamber with a
           single addition
        P: _addition
           _sumFunction
        E: SS049

P: SumClause_addition
        D: Description: c1 = SumClause_addition(Input)
           SumClause_addition sums up all the values
           of cells in the input structure
           using addition operator
        T: c1: +Input.c1+...+Input.cn
        E: SS049
```

```
P: SumClause_sumFuntion
        D: Description: c1 = SumClause_sumFuntion(Input)
           SumClause_sumFuntion sums up all the values
           of cells in the input structure
           using sum function
        T: c1: @SUM(Input)
        E: SS090
```

Figure 3: Examples of goals and plans

5. Tool Dependence and Tool Independence

Our interpretation of problem decomposition is that on the highest (i.e., problem domain) level, goals are same for all programmers independent of the tools they may use. The plans, however, are language or tool dependent. At any problem decomposition level, the programmer knows the tool she uses and implicitly considers the possibilities and constrains of the tool. Even upper level plans cannot be the same for tools having different paradigms for program structuring. In a GAP tree plans at all levels are tool dependent. Only by accident they might be the same for different tools.

For example, a simple sum of a set of values in spreadsheet calculation and in Pascal makes the tool dependency in planning clear. A Pascal program needs a looping structure to go through all the values in the set and a sum variable to cumulate the result. In a spreadsheet the sum of values can be calculated using SUM-function having as its argument the area where the values reside. Further, consider selective sum that skips all negative values. In Pascal, the main loop is the same as in the previous example and only a small change in the plan hierarchy is needed. With spreadsheets, however, the previous simple solution breaks down, and the resulting set of plans differs totally from that of the sum.

Taking the average has been suggested as a "natural plan" possessed by even non-programmers (Davies, 1990). In this view the natural plan is to sum up the individual numbers and divide it by the count of the numbers. In our view, however, average is a tool independent goal. The fact that it can be computed by dividing the sum by the count is not a plan at all. It is just the mathematical definition of average. There are tool dependent plans for obtaining the average. In Pascal this plan consists of three (tool independent) subgoals: obtaining the sum, obtaining the count, and making the division. But in spreadsheet calculation the plan is much easier: use the AVG-function. The difference between these two tool dependent plans is striking. In spreadsheets there is no need even to think about how two obtain the count.

Typical problems solved with either spreadsheet calculation or (Pascal) programming are different. Of course there are problems that can be solved easily with both tools, e.g., problems that have little input and output and contain no computations with a variable number of iteration steps. Most applications suit better to different tools (Casimir, 1992). This is due to the nature of the tools, i.e., different aspects of problems can be expressed more easily with the "right" tool. We have seen that different tools are best suited to different problems. Moreover, the power of plans depends radically on the functionality and limitations of the tools. As a consequence, GAP hierarchies are different in different tools. Thus, it seems fair to say that all plans are tool dependent. Similar plans emerge with different tools only by possible similarity of the tools. Especially, tools within the same programming paradigm share many plans resulting in programming paradigm knowledge.

References:

Bonar, J. G. (1985). "Understanding the Bugs of Novice Programmers." Ph.D. Thesis, University of Massachusetts.

Brewer, W. F., Treyens, J. C. (1981). "Role of Schemata in Memory for Places." Cognitive Psychology 13 : 207-230.

Brooks, R. (1983). "Towards a Theory of the comprehension of computer programs." International Journal of Man-Machine Studies 18 : 543-554.

Brown, P. S., Gould., J. D. (1987). "An Experimental Study of People Creating Spreadsheets." ACM Transactions on Office Information Systems 5 (3): 258-272.

Casimir, R. J. (1992). "Real programmers don't use spreadsheets." ACM SIGPLAN Notices 27 (6): 10-16.

Davies, S. P. (1990). "The Nature and Development of Programming Plans." IJMMS 32 : 461-481.

Davies, S. P. (1993). "The Structure and Content of Programming Knowledge: Disentangling Training and Language Effects in Theories of Skill Development." IJHCI 5 : 325-346.

Detienne, F. (1990). Expert Programming Knowledge: A Schema-based Approach. Psychology of Programming Ed. J.-M. Hoc Green, T. R. G., Samurcay, R., Gilmore, D. J. London, Academic Press. 205-222.

Ehrlich, K., Soloway E. (1984). An Empirical Investigation of the Tacit Plan Knowledge in Programming. Human Factors in Computer Systems Ed. J. T. &. M. Scheider. Ablex Publ. Co. 113-133.

Gamma, E., Helm, R., Johnson, R., Vlissides', J. (1994). "Design Patterns. Elements of Reusable Object-Oriented Software."

Gilmore, D. J., Green, T. R. G. (1987). "Are 'programming plans' psychologically real - outside Pascal?" Human-Computer Interaction - INTERACT'87., Elsevier Science Publishers B. V.

Hendry, D. G., Green, T. R. G. (1994). "Creating, comprehending and explaining spreadsheets: a cognitive interpretation of what discretionary users think of spreadsheet model." International Journal of Human-Computer Studies 40 : 1033-1065.

Lewis, C., Olson, G. M. (1987). "Can Principles of Cognition Lower the Barriers to Programming." Empirical Studies of Programmers: Second Workshop, Washington, D. C., Ablex Publishing Corporation.

Littman, D. C., Pinto, J., Letovsky, S., Soloway, E. (1986). "Mental Models and Software Maintenance." Empirical Studies of Programmers, Washington, DC, Ablex Publ. Corporation, 80-98.

Napier, A. H., Lane, D. M., Batsell, R. R. & Guadango, N. S. (1990). "The Predictability of Commands in a Spreadsheet Program." Interacting with Computers 2 (1): 75-82.

Nardie, B. A. &. M., James R. (1990). "The Spreadsheet Interface: A Basis for End User Programming." INTERACT'90,

Panko, R. R., Halverson, R. P. Jr. (1994). "Patterns of Errors in Spreadsheet Development". Technical Report. Decision Science, College of Business Administration, University of Hawaii.

Pennington, N. (1987a). "Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs." Cognitive Psychology 19 : 295-341.

Pennington, N. (1987b). "Comprehension Strategies in Programming". Empirical Studies of Programmers: Second Workshop, Washington, DC, Ablex Publ. Corporation, 100-113.

Rist, R. S. (1989). "Schema Creation in Programming." Cognitive Science 13 : 389-414.

Sajaniemi, J., Pekkanen, J. (1988). "An empirical Analysis of Spreadsheet Calculation." Software-Practice and Experience 18 (6): 583-596.

Scholtz, J., Wiedenbeck, S. (1990). "Learning Second and Subsequent Programming Languages: A Problem of Transfer." International Journal of Human-Computer Interaction 2 (1): 51-72.

Soloway, E., Ehrlich, K., Bonar, J., Greenspan, J. (1982). What Do Novices Know About Programming. Directions in Human/Computer Interaction Ed. A. Badre Shneiderman, B. Ablex Publ. Co. 27-54.

Soloway, E., Ehrlich, K. (1984). "Empirical Studies of Programming Knowledge." IEEE Trans. on Software Engineering 10 (5): 595-609.

Spohrer, J. C., Soloway, E., Pope, E. (1985). "A Goal/Plan Analysis of Buggy Pascal Programs." Human-Computer Interaction 1 (2): 163-207.

Tukiainen, M., Sajaniemi, J. (1996). "Spreadsheet Goal and Plan Catalog". Technical Report. Department of Computer Science, University of Joensuu, Joensuu, in preparation.

UNIVERSITY OF JOENSUU
DEPARTMENT OF COMPUTER SCIENCE
Report Series A

**Spreadsheet Goal and Plan Catalog:
Additive and Multiplicative Computational
Goals and Plans in Spreadsheet Calculation**

Markku Tukiainen, Jorma Sajaniemi

Report A-1996-4

ACM H.1.2, H.5.2, D.m
UDK 681.3.02, 519.68
ISSN 0789-7316
ISBN 951-708-461-7

Spreadsheet Goal and Plan Catalog:
Additive and Multiplicative Computational Goals and Plans in Spreadsheet Calculation

Markku Tukiainen and Jorma Sajaniemi
mtuki@cs.joensuu.fi, saja@cs.joensuu.fi

University of Joensuu
Department of Computer Science
P.O. Box 111
FIN-80101 Joensuu
Finland

Abstract

Spreadsheeting knowledge can be characterized in the form of goals and plans that describe what must be achieved and how this is done. We have analyzed a set of real-life spreadsheets and extracted a collection of computational goals and plans. This report contains a descriptive catalog of goals and plans consisting of addition, subtraction, multiplication, and division. The catalog is based on actual spreadsheet usage and is not intended to be a prescriptive catalog of what spreadsheeting goals and plans should be like.

Keywords: Spreadsheet calculation, Empirical studies of programming

CR categories: H.1.2, H.5.2, D.m

Contents

1. INTRODUCTION 2

2. THE SCOPE OF THIS REPORT 3

3. NOTATIONS 4
- 3.1 Goal Notation 4
- 3.2 Plan Notation 5

4. GOAL CATALOG 6
- 4.1 General Goals 6
- 4.2 Formula Goals 11
- 4.3 Application Specific Goals 12
- 4.4 Merged Goals 14

5. PLAN CATALOG 16
- 5.1 General Plans 16
- 5.2 Formula Plans 23
- 5.3 Application Specific Plans 23
- 5.4 Merged Plans 26

References 28

1. INTRODUCTION

In the domain of programming expert tacit programming knowledge has been argued to appear in the form of plans which represent many of the stereotypic action sequences in programs (Ehrlich & Soloway, 1984). This notion of programming plans comes from the text comprehension research notion of schemas (Soloway & Ehrlich, 1984). The plan in this thinking is a procedure or strategy in which the key elements of the process have been abstracted and represented explicitly. This approach is often called schema-based, emphasizing the semantic structures of programming and contrasted to control-flow approach emphasizing the syntactic structures (Detienne, 1990; Rist, 1989).

Programming involves knowledge of different domains, at least those of the programming domain and the problem domain (Brooks, 1983; Bonar, 1985). The problem domain description of a programming problem represents the desired goal, or decomposition of this goal into several subgoals, and forms the highest level description of the problem. This composition can be expressed as a goal and plan tree (GAP tree) (Spohrer & al., 1985) where every second level describes (sub)goals to be fulfilled and every other level describes alternative plans for each of the goals. A plan implements the goal in terms of programming constructs and further subgoals. Each program that solves the programming problem obeys a pruned tree obtained from the GAP tree by selecting one special plan for each goal.

Programming knowledge can be divided into knowledge of general programming concepts and knowledge of a certain programming language or tool. Soloway & al., (1982) make a distinction between strategic plans which represent language independent programming knowledge, tactical plans which are kind of semi language independent programming knowledge, and implementation plans which are programming language dependent realizations of higher level plans. Soloway & al. have studied plans more accurately in the lower level of this hierarchy, i.e., plans in Pascal programming. They identified a set of plans (serving as expert programmers) they thought was used in simple Pascal looping programs. The plans were divided into two main kinds: control flow plans representing looping structures and variable plans representing variable usage (Ehrlich & Soloway, 1984).

Although the spreadsheet model has no provision for structuring data or computations, spreadsheets created within this model do contain groups of data items and formulae that have some structure. These structures are caused by programmers (or, more conventionally, users) who have created the spreadsheets. With this in mind we set out to explore actual spreadsheets to see if there are reoccurring uses of computations. We did this by writing a program to extract connected formula graphs from spreadsheets and by analyzing 101 spreadsheets used by Finnish business companies and governmental agencies. (See Sajaniemi & Pekkanen (1988) for a more detailed description of this data.)

We went through all these formula graphs, looked for reoccurring computational structures, and tried to figure out the intentions of the programmer. With this method we soon found out a collection of computational goals (e.g., cumulative sum of a set of numbers) and plans to achieve these goals (e.g., a certain set of formulae in consecutive cells). This report contains a descriptive catalog of the goals and plans we found in the spreadsheets. These goals and plans were found in real-life spreadsheets and thus describe methods utilized by people in their normal work. The catalog is based on actual spreadsheet usage and is by no means intended to be a prescriptive catalog of what spreadsheeting goals and plans should be like.

The goals and plans in this catalog describe spreadsheeting knowledge, i.e., knowledge that can be used in potentially any spreadsheet, and concerns the low-level end of goal and plan hierarchy. A certain spreadsheet, on the other hand, has its own goal and plan structure. This structure uses the given goals and plans adapted to the requirements of the specific spreadsheet.

The goals and plans are grouped in the catalog with respect to their generality but this grouping is by no means an absolute one. Goals and plans can be related to each other in many ways which are not reflected in this grouping. For example, two plans may differ only by the fact that one uses a constant value and the other uses a cell to find the value. These plans could be considered to be some kind of transformations of some general plan template but they are still given in the catalog as two independent plans. The grouping of the plans is not based on any formal notion of plan relationships but it is made to increase the readability of this report only.

2. THE SCOPE OF THIS REPORT

This report describes additive and multiplicative computational goals and plans found in a set of real-life spreadsheet applications (Sajaniemi & Pekkanen, 1988). Additive computational goals and plans are strategies or procedures that contain addition and/or subtraction, and multiplicative computational goals and plans are strategies or procedures that contain multiplication and/or division. We will not list goals and plans involving other operators and functions. For example the IF function is often used in spreadsheets and one of its major uses is to guard for division by zero.

The goals are arranged to four groups: application specific goals, general goals, formula goals and merged goals. Application specific goals are related to specific application, ie, the computations are not applicable accross different applications. The main goals of each application are not articulated here, because we were loooking for more general goals. General goals are goals which can be found (and applied) in many applications. Formula goals are implicit goals, they are used only at the level of formulas. Merged goals are goals that try to achieve two or more goals at the same time.

Spreadsheets contain other than computational goals and plans, also. We have found, for example, depictive goals and plans which represent data structures and other space reservations like titles and data labeling. Such goals and plans are not covered in this catalog. Further, we will not consider other aspects of spreadsheet systems like macros or graphics.

3. NOTATIONS

We have developed a special notation to characterize goals and plans both at the level of general spreadheeting knowledge and at the level of the spreadsheet language, i.e., the formula templates written at the abstracted spreadsheet programming language. In this spreadsheet language, the actual cell references have been abstracted to refer to data structures of the types chamber, sequence or table.

When computation refers to a single cell we will use term chamber and write the reference as Cham1, Cham2, etc. When the computation refers to larger data structures (areas) then the terms sequence and table are used. A sequence is one-dimensional uniform collection of cells, i.e., it lies either in a row or in a column of a spreadsheet. Sequences are named as Seq1, Seq2, etc. A table is a two-dimensional uniform collection of cells, i.e., it has at least two rows of cells and two columns of cells. Tables are noted as Tab1, Tab2, etc. In some computations, structure types are overloaded, i.e., a chamber can act as a sequence or a sequence can play the role of a table. Computations which can be applied to all structure types use the names Input and Output for their arguments.

3.1 Goal Notation

Goal descriptions have the following slots:

G: Goal name, the name reflects the intension of the goal

D: Description: Output = Goal_name(Input), an overview of the

intentional aspects of the goal. Output tells what type of data structure can result from realization of this goal. Input tells what type of data structure can act as initial structure for this goal (if any needed).
P: Plan_names, the names of the plans that fulfil the intensions of this goal
E: Examples, of the actual spreadsheet applications from which the goal was extracted

3.2 Plan Notation

Plan descriptions have the following slots:
P: Plan_name, the name reflects the computation performed by this plan
D: Description: Output = Plan_name(Input), a description of the computational aspects of the plan. Output tells what type of data structure can result from application of this plan. Input tells what type of data structure can act as input structure for this plan (if any needed).
T: Template, the computation written at the abstracted spreadheet programming language. Template shows the formula which lies in the cell(s) of the output. The formula begins with plus (+) or minus (-) operator. The formula is applied to all cells of the output structure once, ie, over all indexes. If the computation needs to refer specific cells of the input then it is noted like Input.c_i where i is the index of Input cell. If an index n is used, like Input.c_n it means that n runs from the first index of input to the last index of input. If for some reason we have to tell all the formulae of the output then the ouput cells are noted like Output. c_n: +Input. c_n. Language contains build-in functions @SUM and @AVE for sums and averages.
C: Constrains, spatial or usually size constrains, eg. argument structures must be of equal size
S: SpatialVariations, some of the plans can be laid out horizontally or vertically, eg. IndexSums can be rowwise or columnwise
O: Observations, other remarks
R: Required subgoals, if computation uses other goals
E: Examples of the actual spreadsheet applications from which the plan was extracted

4. GOAL CATALOG

4.1 General Goals

Goal: TotalSum
D: Cham1 = TotalSum (Input)
TotalSum sums up all the cells in input structure to the output chamber Cham1.
P: _default
_splitting
_cumulative
E: 048, 049, 051, 052, 059, 060, 062

Goal: IndexSum
D: Seq1 = IndexSum (Tab1)
IndexSum sums up all the cells over the indexes of one dimension of the input table to the output sequence.
P: _default
P: _splitting
E: 002, 046, 050, 062

Goal: CumulativeSum
D: Seq2 = CumulativeSum (Seq1)
CumulativeSum sums over all the cells of the input sequence Seq1 to the output sequence Seq2.
P: _default
E: 047, 091

Goal: DifferencePairwise
D: Seq3 = DifferencePairwise (Seq1, Seq2)
DifferencePairwise subtracts cells of Seq2 from cells of Seq1 pairwise
P: _default
E: 049, 090, 065, 066

Goal: Duplication
D: Output = Duplication (Input)
Duplication duplicates values of the input structure to the output structure. Input and Output are of equal size.
P: _default
E: 024, 030, 033, 046, 047

Goal: IncompleteDuplication
D: Seq2 = IncompleteDuplication (Seq1)
IncompleteDuplication duplicates some but not all values of the input sequence to the output sequence.
P: _default
E: 090

Goal: Negation
D: Output = Negation (Input)
Negation changes the values of the input structure to their negations to the output structure. Input and Output are of equal size.

P: _default
E: 024, 030, 033

Goal: SubstractionByConstant
D: Output = SubstractionByConstant (Input)
SubstractionByConstant subtracts a constant value from the original cell.
P: _default
E: 090

Goal: SubstractionFromConstant
D: Output = SubstractionFromConstant (Input)
SubstractionFromConstant subtracts the referenced cell value from a constant.
P: _default
E: 090

Goal: MultiplyByConstant
D: Output = MultiplyByConstant (Input)
MultiplyByConstant transforms a value by multiplying the original cell value by constant.
P: _default
E: 016

Goal: UnitProportion
D: Output = UnitProportion (Input1,Input2)
UnitProportion gives a proportion of a measure by another measure. Typical examples are marks/hour, kg/h or income/capita etc.
P: _default
E: 059, 060, 072

Goal: Percentage
D: Output = Percentage (Input)
Percentage gives a percentage of a cell value from total of input structure.
P: _default
E: 065, 072

Goal: PercentagePairwise
D: Seq3 = PercentagePairwise (Seq1, Seq2)
PercentagePairwise gives a percentage proportion between pairs of cells of input sequences Seq1 and Seq2.
P: _default
E: 048

Goal: PercentageProportion
D: Seq2 = PercentageProportion (Seq1, Cham1)
PercentageProportion gives a percentage proportion between pairs of cells of input sequence Seq1 and chamber Cham1.
P: _default
_usingConstantReference
E: 072

Goal: IncreaseByPercentage
D: Seq2 = IncreaseByPercentage (Seq1, Cham1)
IncreaseByPercentage increases the values of cells of input sequence Seq1 with a percentage proportion constant in chamber Cham1.

P: _default
E: 046

Goal: StepwiseIncrement
D: Seq2 = StepwiseIncrement (Cham1)
StepwiseIncrement increases the values of cells of sequence Seq2 with a constant value in chamber Cham1. This looks like CumulativeSum.
P: _default
E: 047, 076

Goal: CumulativeDifference
D: Seq3 = CumulativeDifference (Seq1, Seq2)
CumulativeDifference sums cumulatively over all the cells of the input sequences Seq1 and Seq2 and then takes a pairwise difference of these cumulated sequences to the output sequence Seq3
P: _default
E: 047

Goal: Duplicates
D: Output = Duplicates (Cham1)
Duplicates duplicates a value in the input chamber to the output structure
P: _default
E: 092, 075

Goal: NumberSeries
D: Output = NumberSeries (Input)
NumberSeries cumulates a constant value of 1 to the original cells
P: _default
E: 092

Goal: CorrectionByIndexClause
D: Output = CorrectionByIndexClause (Input, Cham1, Cham2)
CorrectionByIndexClause changes the values in input structure to the corrected values of output structure (eg. changes currency values according one base year) by dividing with base value Cham1 and multiplying by current year value Cham2 of the original cell
P: _default
E: 033, 030, 049

Goal: Average
D: Cham1 = Average (Input)
Average calculates an average value of the original cells
P: _default
_byHand
E: 059, 054

Goal: AdditionByConstant
D: Seq2 = AdditionByConstant (Seq1, Constant)
AdditionByConstant adds a constant value to the cells of input sequence. This is not Calculator because it is used for forecasting purposes in planned budget application.
P: _default
E: 076

Goal: FormatToThousands

D: Seq2 = FormatToThousands (Seq1)

FormatToThousands divides the cells of input sequence by 1000.

P: _default

E: 048, 049

Goal: ComputedTitles
D: Seq2 = ComputedTitles (Seq1)
ComputedTitles computes titles out of other numeric year titles.
P: _default
E: 048

Goal: PartialTotalSum
D: Cham1 = PartialTotalSum (Input)
PartialTotalSum sums up all the cells in input structure to the output chamber Cham1 and then substracts one of the cells out of this total sum. This is usually done because one of the values is so dominating that further analysis are affected most by this value.
P: _default
E: 065

Goal: SumPairwise
D: Seq3 = SumPairwise (Seq1, Seq2)
SumPairwise sums cells of Seq2 from cells of Seq1 pairwise. Usually Seq1 and Seq2 are of equal size.
P: _default
E: 065, 076, 057

Goal: DifferenceBetweenPairs
D: Seq2 = DifferenceBetweenPairs (Seq1)
DifferenceBetweenPairs takes a difference between two consecutive cells in Seq1 so that first cell in Seq2 yields a difference between Seq1.c_1 and Seq1.c_2 and so on
P: _default
E: 072

Goal: MultiplyPairwise
D: Output = MultiplyPairwise (Input1, Input2)
MultiplyPairwise multiplies values of cells of Input1 by values of cells of Input2 pairwise
P: _default
E: 108, 075

Goal: DividePairwise
D: Output = DividePairwise (Input1, Input2)
DividePairwise divides values of cells of Input1 by values of cells of Input2 pairwise. This is probably always UnitProportion.
P: _default
E: 073, 075

Goal: RepairFormatToThousands
D: Output = RepairFormatToThousands (Input)
RepairFormatToThousands repairs FormatToThousand's division by multiplying values by 1000.
P: _default
E: 075

Goal: ProportionByPercentage
D: Seq2 = ProportionByPercentage (Seq1, Cham1)
ProportionByPercentage gives a percentage proportion Cham1 from input sequence Seq1.
P: _default
_usingConstantReference
E: 046, 059, 062

Goal: PercentageGrowth
D: Seq3 = PercentageGrowth (Seq1, Seq2)
PercentageGrowth gives a percentage growth rate from input sequence Seq1 to input sequence Seq2.
P: _default
_usingPercentageFormat
E: 057, 060

Goal: DivisionByConstant
D: Seq2 = DivisionByConstant (Seq1, Constant1)
DivisionByConstant divides values in input sequence Seq1 by Constant1.
P: _default
E: 059

Goal: DecreaseByPercentage
D: Seq3 = DecreaseByPercentage (Seq1, Seq2)
DecreaseByPercentage decreases values in input sequence Seq1 by percentages in Seq2.
P: _default
E: 062

Goal: PercentageDecrease
D: Seq3 = PercentageDecrease (Seq1, Seq2)
PercentageDecrease gives a percentage decrease rate from input sequence Seq1 to input sequence Seq2.
P: _default
E: 062

Goal: ProportionPartition
D: Seq2 = ProportionPartition (Cham1, Cham2, Seq1)
ProportionPartition gives a proportion parts of Cham1 by proportions in input sequence Seq1, eg. Cham1 is to be divede to parts which are proportions in Seq1.
P: _default
E: 061

Goal: PartlySum
D: Seq1= PartlySum (Constant1, Seq2)
PartlySum sums a size Constant1 portions of sequence Seq2.
P: _default
E: 106

Goal: FormatTo100Kilometers
D: Seq2 = FormatTo100Kilometers (Seq1)
FormatTo100Kilometers divides the cells of input sequence by 100.
O: Used to format vehicle's usage kilometers to hundreds.
P: _default
E: 063

Goal: FormatToHours
D: Seq2 = FormatToHours (Seq1)
FormatToHours divides the cells of input sequence by 60.
O: Used to format minutes to hours.
P: _default
E: 063

4.2 Formula Goals

Goal: SumClause
D: Cham1 = SumClause (Input)
SumClause sums up all the values in input structure to the output chamber with a single addition
P: _addition
_sumFunction
E: 049

Goal: Calculator
D: Output = Calculator (Input)
Calculator calculates value of a cell using constant operands and basic operators +,-,* and /.
P: _default
E: 001, 090, 075

4.3 Application Specific Goals

Goal: MeansEndsAnalysis
D: Seq3 = MeansEndsAnalysis (Seq1, Seq2, Cham1)
MeansEndsAnalysis calculates a productivity value using means/ends values of the input sequence Seq1 and corrects it by index values Seq2 and Cham1.
P: _default
E: 016

Goal: TurnTime
D: Cham4 = TurnTime (Cham1, Cham2, Cham3)
TurnTime computes a turn time of store.
P: _default
E: 064

Goal: VacationPay
D: Seq3 = VacationPay (Seq1, Seq2, 50)
VacationPay calculates a vacation time extra payment by dividing the monthly payment in Seq1 with 50 and multiplying this amount by the number of vacation days in Seq2.
P: _default
E: 046

Goal: XeroxCopyPrice
D: Cham6 = XeroxCopyPrice (Cham1, Cham2, Cham3, Cham4, Cham5)
XeroxCopyPrice calculates an approximate xerox copy price for square metre taking account of employee's salary and price of materials.
P: _default
E: 054

Goal: AverageSquareMetreAge
D: Cham1 = AverageSquareMetreAge (Seq1, Seq2)
AverageSquareMetreAge calculates an average age for a number of buildings weighted by the area of buildings. Seq1 holds the areas and Seq2 holds the ages.
P: _default
E: 075

Goal: TemperatureIndex
D: Seq4 = TemperatureIndex (Seq1, Seq2, Seq3, Constant1, Constant2)
TemperatureIndex calculates a monthly normalized temperature index. Seq1 holds the normalized degree-day for current month, Seq2 holds a sum of energy consumption for last twelve months, Seq3 holds a sum of degree-days for last twelve months.
P: _default
E: 106

Goal: EnergyConsumption

D: Seq3 = EnergyConsumption (Seq1, Seq2, Constant1, Constant2, Constant3)

EnergyConsumption calculates a monthly normalized energy consumption. Seq1 holds a sum of energy consumption for last twelve months, Seq2 holds a sum of degree-days for last twelve months.

P: _default

E: 106

Goal: ObtainableSales

D: Cham3 = ObtainableSales (Cham1, Cham2, Constant1, Constant2)

ObtainableSales calculates obtainable sales for a year. Cham1 holds a premis for sales for year, Cham2 holds a turnover for year.

P: _default

E: 069

Goal: DepreciatedValues

D: Seq4 = DepreciatedValues (Seq1, Seq2, Seq3)

DepreciatedValues calculates depreciated values for different things. Seq1 and Seq2 hold values of things, Seq3 holds depreciation value factors.

P: _default

E: 071

Goal: CirculationPeriodFactors

D: Seq2 = CirculationPeriodFactors (Seq1, Cham1, Cham2, Constant1, Constant2)

TurnoverTimeFactors calculates circulation period factors. Seq1 holds the values to be circulated. Cham1 holds a turnover for year, Cham2 holds the number of months in an accounting period.

P: _default

E: 071

Goal: ComputedAreaValue

D: Seq2 = ComputedAreaValue (Seq1, Cham1, Cham2)

ComputedAreaValue calculates a value of build area for company's assets. Seq1 holds the areas to be valuated. Cham1 holds a total sum of the areas, Cham2 holds the weighted value of the area according to geographic location.

P: _default

E: 087

Goal: YearlyCustomerSales

D: Seq5 = YearlyCustomerSales (Seq1, Seq2, Seq3, Seq4, Constant1)

YearlyCustomerSales calculates a yearly customer sales of animal food. Sales are calculated taking into account of percentage of customer sales (Seq1), yearly sales in kilograms (Seq2), price for 100 kilograms (Seq3) and profit percentage (seq4).

P: _default

E: 062

4.4 Merged Goals

MGoal: PercentagePairwise_&_SumClause
D: Seq3 = PercentagePairwise_&_SumClause (Seq1, Tab1)
PercentagePairwise_&_SumClause gives a percentage proportion between pairs of cells of input sequences Seq1 and Seq2. Seq2 is a IndexSum of Tab1 which can be computed only in formula of goal.
P: _default
R: Seq2 is IndexSum of Tab1. Seq1 and Seq2 of equal sizes.
E: 019

MGoal: UnitProportion_&_MultiplyByConstant
D: Output = UnitProportion_&_MultiplyByConstant (Input1, Input2, Constant1)
UnitProportion_&_MultiplyByConstant gives a proportion of a measure by another measure repaired with constant.
P: _default
E: 016

MGoal: FormatToThousands_&_CorrectionByIndexClause
D: Seq2 = FormatToThousands_&_CorrectionByIndexClause (Seq1)
FormatToThousands_&_CorrectionByIndexClause gives index clause corrected sequenced measured in thousands.
P: _default
E: 049

MGoal: DifferencePairwise_&_IndexSum
D: Seq2 = DifferencePairwise (Seq1,_IndexSum(Tab1))
DifferencePairwise_&_IndexSum gives a difference pairwise of input sequence Seq1 and formula goal IndexSum output sequence.
P: _default
E: 073

MGoal: Duplicates_&_CorrectionByIndexClause
D: Seq1 = Duplicates_&_CorrectionByIndexClause (Cham1, Constant1.. Constantn)
Duplicates_&_CorrectionByIndexClause gives a serie of duplicates of Cham1 and corrects some of them with constants Constant1.. Constantn.
P: _default
E: 075

MGoal: UnitProportion_&_ RepairFormatToThousands
D: Cham3= UnitProportion_&_ FormatToThousands (Cham1, Cham2, 1000)
UnitProportion_&_ FormatToThousands gives a proportion of Cham1 and Cham2 and multiplies this value by 1000 in order to correct previous FormatToThousands usage.
P: _default
E: 075

MGoal: ProportionByPercentage_&_ FormatToThousands

D: Seq2= ProportionByPercentage_&_ FormatToThousands (Seq1, Cham1, 100000)

ProportionByPercentage_&_ FormatToThousands gives a percentage proportion in Seq1 of Cham1 and divides this value by 1000 in order to change value to thousands.

P: _default

E: 059

MGoal: Input_&_CorrectionByIndexClause

D: Seq2= Input_&_CorrectionByIndexClause (Cham1, Constant1, Constant2)

Input_&_CorrectionByIndexClause transforms the original input in Constant2 by index correction Cham1 and Constant1. This is novice merged goal. Probable cause could be that novice does not want to see the original inputs and does not know how to hide cells so he puts the inputs to the formulas.

P: _default

E: 048, 049

MGoal: TotalSum_&_FormatToThousands

D: Seq2= TotalSum_&_FormatToThousands (Seq1, 1000)

TotalSum_&_FormatToThousands calculates a total sum of Seq1 and formats it to thousands.

P: _default

E: 033

MGoal: MultiplyPairwise_&_ DifferencePairwise

D: Seq4= MultiplyPairwise_&_ DifferencePairwise (Seq1, Seq2, Seq3)

MultiplyPairwise_&_ DifferencePairwise calculates a pairwise difference of Seq1 and Seq2 and multiplies result pairwise with Seq3.

P: _default

E: 108

MGoal: MultiplyPairwise _&_FormatToThousands

D: Seq3= MultiplyPairwise_&_FormatToThousands (Seq1, Seq2, 1000)

MultiplyPairwise_&_FormatToThousands multiplies Seq1 and Seq2 pairwise and formats Result to thousands.

P: _default

E: 115

5. PLAN CATALOG

5.1 General Plans

Plan: TotalSum_default

D: Cham1 = TotalSum_default (Input)

TotalSum_default sums up all the values of cells in the input structure directly

T: SumClause(Input)

E: 049, 051, 052, 059, 060

Plan: TotalSum_cumulative

D: Cham1 = TotalSum_cumulative (Input)

TotalSum_cumulative cumulates sum over all the values of cells in the input structure

R: Seq1 = CumulativeSum(Input)

T: +Seq1.c_{last}

E: 091

Plan: TotalSum_splitting

D: Cham1 = TotalSum_splitting (Part_1_of_Input)

:

Chamn = TotalSum_splitting(Part_n_of_Input)

TotalSum_splitting sums up all the cells in all the parts of the input structure individually to the output cells

T: SumClause(Input)

E: 059

Plan: IndexSum_default

D: Seq1 = IndexSum_default (Tab1)

IndexSum_default sums up all the cells over one of the indexes of the input table to the output sequence

T: SumClause(Tab1.index1)

S: Rowwise

S: Columnwise

E: 002, 047, 050

Plan: IndexSum_splitting

D: Seq1 = IndexSum_splitting (Tab1)

IndexSum_splitting sums up all the cells in all the parts of the input table over one of the indexes of the input table to the output sequence

T: SumClause(Tab1.index1)

S: Rowwise

S: Columnwise

E: 002, 046

Plan: CumulativeSum_default

D: Seq2 = CumulativeSum_default (Seq1)

CumulativeSum sums over all the cells of the input sequence Seq1 to the output sequence Seq2

T: + Seq1.c_n+Seq2.c_{n-1}

S: Rowwise: 047
S: Columnwise: 091
E: 047, 091

Plan: Duplication_default
D: Cham2 = Duplication_default (Cham1)
Duplication_default duplicates value of the input cell to the output cell
T: +Cham1
E: 024, 030, 033

Plan: IncompleteDuplication_default
D: Seq2 = IncompleteDuplication_default (Seq1)
IncompleteDuplication_default duplicates some but not all values of the input sequence to the output sequence
T: $Seq2.c_1$: $+Seq1.c_1$
:
$Seq2.c_{n-1}$: $+Seq1.c_{n-1}$
O: Usually all but the last value are duplicated.
E: 090

Plan: SubstractionByConstant_default
D: Cham2 = SubstractionByConstant_default (Cham1, Constant)
SubstractionByConstant_default repairs a value by subtracting the duplicated original by constant
R: Duplication
T: + Cham1-Constant1
O: in 090 cell: B32 template was -(Calculator)+cell reference
E: 090

Plan: SubstractionFromConstant_default
D: Cham2 = SubstractionFromConstant_default (Cham1, Constant)
SubstractionFromConstant_default repairs a value by subtracting a duplicate from the original constant
R: Duplication(Cham1)
T: +Constant1- Duplication(Cham1)
E: 090

Plan: MultiplyByConstant_default
D: Cham2 = MultiplyByConstant_default (Cham1, Constant)
MultiplyByConstant_default repairs a value by multiplying it with constant
T: + Cham1*Constant1
E: 016

Plan: UnitProportion_default
D: Output = UnitProportion_default (Input1,Input2)
UnitProportion_default gives a proportion of a measure by another measure for instance money divided by time marks/hour.
T: $+Input1.c_x/Input2.c_x$
E: 059, 060

Plan: Percentage_default
D: Output = Percentage_default (Input)
Percentage_default gives a percentage of a cell value from total of input structure.
T: $+ Input.c_x$/SUM(Input)*100
E: 065, 072, 062

Plan: DifferencePairwise_default
D: Seq3 = DifferencePairwise_default (Seq1, Seq2)
DifferencePairwise_default subtracts cells of Seq2 from cells of Seq1 pairwise
T: $+Seq1.c_n-Seq2.c_n$
E: 049, 090, 065

Plan: Negation_default
D: Output = Negation_default (Input)
Negation_default changes the values of the input structure to their negations to the output structure
T: $-Input.c_n$
C: Input and Output of equal sizes
E: xxx

Plan: PercentagePairwise_default
D: Seq3 = PercentagePairwise_default (Seq1, Seq2)
PercentagePairwise_default gives a percentage proportion between pairs of cells of input sequences Seq1 and Seq2.
T: $+Seq1.c_n/Seq2.c_n*100$
E: 048

Plan: PercentagePairwise_usingPercentageFormat
D: Seq3 = PercentagePairwise_usingPercentageFormat (Seq1, Seq2)
PercentagePairwise_ usingPercentageFormat gives a percentage proportion between pairs of cells of input sequences Seq1 and Seq2.
T: $+Seq1.c_n/Seq2.c_n$
O: Output values must be formatted with Percentage-format
E: 105

Plan: PercentageProportion_default
D: Seq2 = PercentageProportion _default (Seq1, Cham1)
PercentageProportion _default gives a percentage proportion between pairs of cells of input sequence Seq1 and chamber Cham1.
T: $+Seq1.c_n/Cham1*100$
E: 072

Plan: PercentageProportion_usingConstantReference
D: Seq2 = PercentageProportion _default (Seq1, Cham1)
PercentageProportion _default gives a percentage proportion between pairs of cells of input sequence Seq1 and chamber Cham1.
T: $+Seq1.c_n/Cham1*Constant1$
R: Constant1 = Duplication(100)
E: 070

Plan: IncreaseByPercentage_default
D: Seq2 = IncreaseByPercentage _default (Seq1, Cham1)
IncreaseByPercentage _default increases the values of cells of input sequence Seq1 with a percentage proportion constant in chamber Cham1.
T: $+Seq1.c_n*(1+Cham1/100)$
E: 046

Plan: StepwiseIncrement_default

D: Seq2 = StepwiseIncrement_default (Cham1)
StepwiseIncrement_default increases the values of cells of sequence Seq2 with a constant value in chamber Cham1. This looks like CumulativeSum.
T: +Cham1+Seq2.c_{n-1}
E: 047, 076

Plan: CumulativeDifference_default

D: Seq3 = CumulativeDifference_default (Seq1, Seq2)
CumulativeDifference_default sums cumulatively over all the cells of the input sequences Seq1 and Seq2 and then takes a pairwise difference of these cumulated sequences to the output sequence Seq3
T: + DifferencePairwise(Seq1, Seq2)
R: CumulativeSum(Seq1), CumulativeSum(Seq2)
E: 047

Plan: Duplicates_default

D: Output = Duplicates_default (Cham1)
Duplicates_default duplicates a value in the input chamber to the output structure
P:
T: +Output.c_1: +Cham1
+Output.c_n: +Output.c_{n-1}
E: 092

Plan: NumberSeries_default

D: Output = NumberSeries_default (Input)
NumberSeries_default cumulates a constant value of 1 to the original cells
T: +Output.c_n: +Output.c_{n-1}+1
E: 092

Plan: CorrectionByIndexClause_default

D: Output = CorrectionByIndexClause_default (Input, Cham1, Cham2)
CorrectionByIndexClause_default changes the values in input structure to the corrected values of output structure (eg. changes currency values according one base year) by dividing with base value Cham1 and multiplying by current year value Cham2 of the original cell
T: +Input.c_n*Cham1/Cham2
E: 033, 030, 049, 052÷

Plan: Average_default

D: Cham1 = Average_default (Input)
Average_default calculates an average value of the original cells using build in arithmetic average function
T: @AVE(Input)
E: 059

Plan: Average_ byHand

D: Cham1 = Average_ byHand (Input)
Average_ byHand calculates an average value of the original cells using TotalSum and division
T: +TotalSum(Input)/count of cells in Input

E: 054

Plan: AdditionByConstant_default

D: Seq2 = AdditionByConstant_default (Seq1, Constant)
AdditionByConstant_default adds a constant value to the cells of input sequence. This is not Calculator because it is used for forecasting purposes in planned budget application.
T: $+Seq1.c_n+Constant$
E: 076

Plan: FormatToThousands_default

D: Seq2 = FormatToThousands_default (Seq1)
FormatToThousands_default divides the cells of input sequence by 1000.
T: $+Seq1.c_n/1000$
E: 048, 049

Plan: ComputedTitles_default

D: Seq2 = ComputedTitles_default (Seq1)
ComputedTitles_default computes titles out of other numeric year titles. This plan strips the centuries out of year by substracting 1900 from original.
T: $+Seq1.c_n-1900$
E: 048

Plan: PartialTotalSum_default

D: Cham1 = PartialTotalSum_default (Input)
PartialTotalSum_default sums up all the cells in input structure to the output chamber Cham1 and then substracts one of the cells out of this total sum. This is usually done because one of the values is so dominating that further analysis are affected most of this value.
T: $+TotalSum (Input)-Input.c_x$
E: 065

Plan: SumPairwise_default

D: Seq3 = SumPairwise_default (Seq1, Seq2)
SumPairwise_default sums cells of Seq2 from cells of Seq1 pairwise. Seq1 and Seq2 of equal sizes.
T: $+Seq1.c_n+Seq2.c_n$
E: 065

Plan: DifferenceBetweenPairs_default

D: Seq2 = DifferenceBetweenPairs_default (Seq1)
DifferenceBetweenPairs_default takes a difference between two consecutive cells in Seq1 so that first cell in Seq2 yields a difference between $Seq1.c_1$ and $Seq1.c_2$ and so on
T: $Seq2.c_1$: $+Seq1.c_2-Seq1.c_1$
$Seq2.c_m$: $+Seq1.c_n-Seq1.c_{n-1}$
O: m = n-1 ie. Seq2 is one cell shorter than Seq1
E: 072

Plan: MultiplyPairwise_default

D: Output = MultiplyPairwise_default (Input1, Input2)
MultiplyPairwise_default multiplies values of cells of Input1 by values of cells of Input2 pairwise
T: $+ Input1.c_n * Input2.c_n$

E: 108

Plan: DividePairwise_default
D: Output = DividePairwise_default (Input1, Input2)
DividePairwise_default divides values of cells of Input1 by values of cells of Input2 pairwise. This is probably always UnitProportion.
T: + $Input1.c_n$/ $Input2.c_n$
E: 073

Plan: RepairFormatToThousands_default
D: Output = RepairFormatToThousands_default (Input)
RepairFormatToThousands_default repairs FormatToThousands's division by multiplying values by 1000.
Is this computational goal?
T: $Input.c_n$*1000
E: 075

Plan: ProportionByPercentage_default
D: Seq2 = ProportionByPercentage_default (Seq1, Cham1)
ProportionByPercentage_default gives a percentage proportion Cham1 from input sequence Seq1.
T: +$Seq1.c_n$*Cham1/100
E: 046, 062

Plan: ProportionByPercentage_usingConstantReference
D: Seq2 = ProportionByPercentage_default (Seq1, Cham1)
ProportionByPercentage_default gives a percentage proportion Cham1 from input sequence Seq1.
T: +$Seq1.c_n$*Cham1/Constant1
R: Constant1 = Duplication(100)
E: 087

Plan: PercentageGrowth_default
D: Seq3 = PercentageGrowth_default (Seq1, Seq2)
PercentageGrowth_default gives a percentage grow rate from input sequence Seq1 to input sequence Seq2.
T: +$Seq2.c_n$/$Seq1.c_n$*100-100
E: 057

Plan: PercentageGrowth_usingPercentageFormat
D: Seq3 = PercentageGrowth_usingPercentageFormat (Seq1, Seq2)
PercentageGrowth_usingPercentageFormat gives a percentage grow rate from input sequence Seq1 to input sequence Seq2.
T: +($Seq2.c_n$-$Seq1.c_n$)/$Seq1.c_n$
O: Output values must be formatted with Percentage-format
E: 060

Plan: DivisionByConstant_default
D: Seq2 = DivisionByConstant_default (Seq1, Constant1)
DivisionByConstant_default divides values in input sequence Seq1 by Constant1.
T: +$Seq1.c_n$/Constant1
E: 059

Plan: DecreaseByPercentage_default

D: Seq3 = DecreaseByPercentage_default (Seq1, Seq2)

DecreaseByPercentage_default decreases values in input sequence Seq1 by percentages in Seq2.

T: +Seq2.c_n*(1-Seq1.c_n/100)

E: 062

Plan: PercentageDecrease_default

D: Seq3 = PercentageDecrease_default (Seq1, Seq2)

PercentageDecrease_default gives a percentage decrease rate from input sequence Seq1 to input sequence Seq2.

T: +(Seq1.c_n-Seq2.c_n)/Seq1.c_n*100

E: 062

Plan: ProportionPartition_default

D: Seq2 = ProportionPartition_default (Cham1, Cham2, Seq1)

ProportionPartition_default gives a proportion parts of Cham1 by proportions in input sequence Seq1, eg. Cham1 is to be divede to parts which are proportions in Seq1.

R: Cham2 = TotalSum(Seq1)

T: +Cham1/Cham2*Seq1.c_n

E: 061

Plan: PartlySum_default

D: Seq1= PartlySum_default (Constant1, Seq2)

PartlySum_default sums a size Constant1 portions of sequence Seq2.

T: Seq1.c_1 = TotalSum(Seq2.c_1..Seq2.$c_{(Constant1)}$)

Seq1.c_n = TotalSum(Seq2.$c_{(n-Constant1)}$..Seq2.c_n)

0: The size of TotalSum argument area is Constant1 cells.

E: 106

Plan: FormatTo100Kilometers_default

D: Seq2 = FormatTo100Kilometers_default (Seq1)

FormatTo100Kilometers_default divides the cells of input sequence by 100.

O: Used to format vehicle's usage kilometers to hundreds.

T: +Seq1.c_n/100

E: 063

Plan: FormatToHours_default

D: Seq2 = FormatToHours_default (Seq1)

FormatToHours_default divides the cells of input sequence by 60.

O: Used to format minutes to hours.

T: +Seq1.c_n/60

E: 063

5.2 Formula Plans

Plan: SumClause_addition

D: Cham1 = SumClause_addition (Input)
SumClause_addition sums up all the values of cells in the input structure using addition operator
T: $+Input.c_1+...+Input.c_n$
E: 047, 49

Plan: SumClause_sumFuntion

D: Cham1 = SumClause_sumFuntion (Input)
SumClause_sumFuntion sums up all the values of cells in the input structure using sum function
T: : @SUM(Input)
E: 047, 090

Plan: Calculator_default

D: Output = Calculator_default(Input)
Calculator_default calculates value of a cell using constant operands and basic operators +,-,* and /.
T: No particular form
O: No other operators or functions.
E: 001, 090

5.3 Application Specific Plans

Plan: MeansEndsAnalysis_default

D: Seq3 = MeansEndsAnalysis_default (Seq1, Seq2, Cham1)
MeansEndsAnalysis calculates a productivity value by means/ends values of the input sequence Seq1 and corrects it by index values Seq2 and Cham1.
T: $+Seq1.c_n/Seq2.c_n*Cham1$
R: Input Seq1 contains previously calculated means/ends values.

Plan: TurnTime_default

D: Cham4 = TurnTime_default (Cham1, Cham2, Cham3)
TurnTime_default computes a turn time of store.
T: +2* Cham1/((2* Cham2)+ Cham3)
R: Cham1 = Yearly need of products
Cham2 = Safety resource of products
Cham3 = Amount to be acquired at time
E: 064

Plan: VacationPay_default

D: Seq3 = VacationPay_default (Seq1, Seq2, 50)
VacationPay_default calculates a vacation time extra payment by dividing the monthly payment in Seq1 with 50 and multiplying this amount by the number of vacation days in Seq2.

T: $+Seq2.c_n*Seq1.c_n/50$
E: 046

Plan: XeroxCopyPrice_default
D: Cham6 = XeroxCopyPrice_default (Cham1, Cham2, Cham3, Cham4, Cham5)
XeroxCopyPrice_default calculates an approximate xerox copy price for square metre taking account of employee's salary and price of materials.
T: +Cham5*Cham2/(Cham4*Cham2+Cham3*Cham1)+Cham2
R: Cham1 = price of paper copy material÷
Cham2 = price of plastic copy material
Cham3 = amount paper used in a year
Cham4 = amount plastic used in a year
Cham5 = employee's salary
E: 054

Plan: AverageSquareMetreAge_default
D: Cham1 = AverageSquareMetreAge_default (Seq1, Seq2)
AverageSquareMetreAge_default calculates an average age for a number of buildings weighted by the area of buildings. Seq1 holds the areas and Seq2 holds the ages.
T: +Cham2/Cham3
R: Cham2=TotalSum(Seq1*Seq2)
Cham3=TotalSum(Seq3)
Seq3=MultiplyPairwise(Seq1,Seq2)
E: 075

Plan: TemperatureIndex_default
D: Seq4 = TemperatureIndex (Seq1, Seq2, Seq3, Constant1, Constant2)
TemperatureIndex calculates a monthly normalized temperature index. Seq1 holds the normalized degree-day for current month, Seq2 holds a sum of energy consumption for last twelve months, Seq3 holds a sum of degree-days for last twelve months.
T: $+Seq1.c_n*100*Seq2.c_n/(Seq3.c_n*11890)$
R: Seq1 = normalized degree-day for current month
Seq2 = TotalSum(energy consumption last twelve months)
Seq3 = TotalSum(degree-days for last twelve months)
Constant1 = 100
Constant2 = 11890
E: 106

Plan: EnergyConsumption_default
D: Seq3 = EnergyConsumption_default (Seq1, Seq2, Constant1, Constant2, Constant3)
EnergyConsumption_default calculates a monthly normalized energy consumption. Seq1 holds a sum of energy consumption for last twelve months, Seq2 holds a sum of degree-days for last twelve months.
T: $+(Seq1.c_n-12*145)*5267/Seq2.c_n+12*145$
R: Seq1 = TotalSum(energy consumption last twelve months)
Seq2 = TotalSum(degree-days for last twelve months)
E: 106

Plan: ObtainableSales_default

D: Cham3 = ObtainableSales_default (Cham1, Cham2, Constant1, Constant2)
ObtainableSales_default calculates obtainable sales for a year. Cham1 holds a premis for sales for year, Cham2 holds a turnover for year.
T: +Cham1/360*Cham2*11
R: Cham1 = Premis for sales
Cham2 = Turnover
E: 069

Plan: DepreciatedValues_default

D: Seq4 = DepreciatedValues_default (Seq1, Seq2, Seq3)
DepreciatedValues_default calculates depreciated values for different things. Seq1 and Seq2 hold values of things, Seq3 holds depreciation value factors.
T: $+(Seq1.c_n+Seq2.c_n)*Seq3.c_n$
R: Seg3 = Depreciation value factors
E: 071

Plan: CirculationPeriodFactors_default

D: Seq2 = CirculationPeriodFactors_default (Seq1, Cham1, Cham2, Constant1, Constant2)
TurnoverTimeFactors_default calculates circulation period factors. Seq1 holds the values to be circulated. Cham1 holds a turnover for year, Cham2 holds the number of months in an accounting period.
T: $+Constant1*Seq1.c_n/Cham1*Constant2/Cham2$
R: Cham1 = Turnover
Cham2 = Number of months
Constant1 = Duplication(360)
Constant2 = Duplication(12)
E: 071

Plan: ComputedAreaValue_default

D: Seq2 = ComputedAreaValue_default (Seq1, Cham1, Cham2)
ComputedAreaValue_default calculates a value of build area for company's assets. Seq1 holds the areas to be valuated. Cham1 holds a total sum of the areas, Cham2 holds the weighted value of the area according to geographic location.
T: $+Seq1.c_n/Cham1*Cham2$
R: Cham1 = TotalSum(Seq1)
Cham2 = (Cham1/15000*32450)*40/1000
E: 087

Plan: YearlyCustomerSales_default

D: Seq5 = YearlyCustomerSales_default (Seq1, Seq2, Seq3, Seq4, Constant1)
YearlyCustomerSales_default calculates a yearly customer sales of animal food. Sales are calculated taking into account of percentage of customer sales (Seq1), yearly sales in kilograms (Seq2), price for 100 kilograms (Seq3) and profit percentage (seq4).
T: $+(Seq1.c_n/100*Seq2.c_n)*Seq3.c_n/(100-Seq4.c_n)$
E: 062

5.4 Merged Plans

MPlan:PercentagePairwise_&_SumClause_default
D: Seq2 = PercentagePairwise_withSumClause (Seq1, Tab1)
T: $Seq1.c_n$/SUM(Tab1.index1)*100
C: Seq1 and Tab1.index1 of equal sizes
E: 019

MPlan:UnitProportion_&_MultiplyByConstant _default
D: Output = UnitProportion_withMultiplyByConstant_default (Input1, Input2, Constant1)
UnitProportion_with MultiplyByConstant _default gives a proportion of a measure by another measure repaired with constant.
T: +$Input1.c_x$/$Input2.c_x$*Constant1
E: 016

MPlan:FormatToThousands_&_CorrectionByIndexClause_default
D: Seq2 = FormatToThousands_&_CorrectionByIndexClause_default (Seq1)
FormatToThousands_&_CorrectionByIndexClause_default gives index clause corrected sequenced measured in thousands.
T: +$Seq1.c_n$/1000*Cham1/Cham2
E: 049

MPlan:DifferencePairwise_&_IndexSum
D: Seq2 = DifferencePairwise (Seq1,_IndexSum(Tab1))
DifferencePairwise_&_IndexSum gives a difference pairwise of input sequence Seq1 and formula goal IndexSum output sequence.
T: +$Seq1.c_n$-IndexSum(Tab1).c_n
E: 073

MPlan:Duplicates_&_CorrectionByIndexClause_default
D: Seq1 = Duplicates_&_CorrectionByIndexClause_default (Cham1, Constant1.. Constantn)
Duplicates_&_CorrectionByIndexClause_default gives a serie of duplicates of Cham1 and corrects some of them with constants Constant1.. Constantn using formula goal.
T: +Cham1*Constantx
E: 075

MPlan:UnitProportion_&_ RepairFormatToThousands_default
D: Cham3= UnitProportion_&_ FormatToThousands_default (Cham1, Cham2, 1000)
UnitProportion_&_ FormatToThousands_default gives a proportion of Cham1 and Cham2 and multiplies this value by 1000 in order to correct previous FormatToThousands usage.
T: +Cham1/Cham2*1000
E: 075

MPlan:ProportionByPercentage_&_ FormatToThousands _default

D: Seq2= ProportionByPercentage_&_ FormatToThousands_default (Seq1, Cham1, 100000)

ProportionByPercentage_&_ FormatToThousands_default gives a percentage proportion in Seq1 of Cham1 and divides this value by 1000 in order to change value to thousands.

T: +Seq1.c_n/100*Cham1/1000

(in matter of fact written Seq1.c_n*Cham1/100000)

E: 059

MPlan:Input_&_CorrectionByIndexClause_default

D: Seq2= Input_&_CorrectionByIndexClause_default (Cham1, Constant1, Constant2)

Input_&_CorrectionByIndexClause_default transforms the original input in Constant2 by index correction Cham1 and Constant1. This is novice merged goal. Probable cause could be that novice does not want to see the original inputs and does not know how to hide cells so he puts the inputs to the formulas.

T: +Cham1/Constant1*Constant2

E: 048, 049

MPlan:TotalSum_&_FormatToThousands_default

D: Seq2= TotalSum_&_FormatToThousands_default (Seq1, 1000)

TotalSum_&_FormatToThousands_default calculates a total sum of Seq1 and formats it to thousands.

T: +TotalSum(Seq1)/1000

E: 033

MPlan:MultiplyPairwise_&_ DifferencePairwise_default

D: Seq4= MultiplyPairwise_&_ DifferencePairwise_default (Seq1, Seq2, Seq3)

MultiplyPairwise_&_ DifferencePairwise_default calculates a pairwise difference of Seq1 and Seq2 and multiplies result pairwise with Seq3.

T: +Seq3.c_n*(Seq1.c_n-Seq2.c_n)

E: 108

MPlan:MultiplyPairwise _&_FormatToThousands_default

D: Seq3=MultiplyPairwise_&_FormatToThousands_default (Seq1, Seq2, 1000)

MultiplyPairwise_&_FormatToThousands_default multiplies Seq1 and Seq2 pairwise and formats Result to thousands.

T: +Seq1.c_n*Seq2.c_n/1000

E: 115

References:

Bonar, J. G. (1985). "Understanding the Bugs of Novice Programmers." Ph.D. Thesis, University of Massachusetts.

Brooks, R. (1983). "Towards a Theory of the comprehension of computer programs." International Journal of Man-Machine Studies 18 : 543-554.

Detienne, F. (1990). Expert Programming Knowledge: A Schema-based Approach. Psychology of Programming Ed. J.-M. Hoc Green, T. R. G., Samurcay, R., Gilmore, D. J. London, Academic Press. 205-222.

Ehrlich, K., Soloway E. (1984). An Empirical Investigation of the Tacit Plan Knowledge in Programming. Human Factors in Computer Systems Ed. J. T. &. M. Scheider. Ablex Publ. Co. 113-133.

Rist, R. S. (1989). "Schema Creation in Programming." Cognitive Science 13 : 389-414.

Sajaniemi, J., Pekkanen, J. (1988). "An empirical Analysis of Spreadsheet Calculation." Software-Practice and Experience 18 (6): 583-596.

Soloway, E., Ehrlich, K. (1984). "Empirical Studies of Programming Knowledge." IEEE Trans. on Software Engineering 10 (5): 595-609.

Soloway, E., Ehrlich, K., Bonar, J., Greenspan, J. (1982). What Do Novices Know About Programming. Directions in Human/Computer Interaction Ed. A. Badre Shneiderman, B. Ablex Publ. Co. 27-54.

Spohrer, J. C., Soloway, E., Pope, E. (1985). "A Goal/Plan Analysis of Buggy Pascal Programs." Human-Computer Interaction 1 (2): 163-207.