ACCEPTED
FACULTY OF GRADUATE STUDIES
DEAN
DATE March 12, 1987

AN INTENSIONAL 3-D SPREADSHEET AND ITS IMPLEMENTATION

by

WEICHANG DU
B.Sc., Beijing Institute of Computer, 1983

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

Dr. William W. Wadge

Dr. Michael Levy

Dr. Gary G. Miller

Dr. Charles G. Morgan

© WEICHANG DU, 1986

University of Victoria

December 1986

*All rights reserved. This thesis may not be reproduced
in whole or in part, by mimeograph or other means,
without the permission of the author.*

Supervisor: Professor William W. Wadge

ABSTRACT

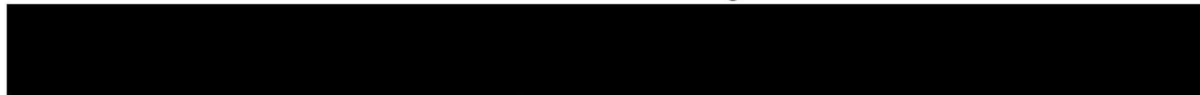
The spreadsheet design presented in this thesis is based on the following observations. Existing spreadsheets are intended mainly for business and financial applications and they have user friendly interfaces but relatively poor programming and calculating abilities. This restricts their usefulness for problem-solving. Most existing spreadsheet languages are not based on sound programming techniques, and this makes the structure of spreadsheet programs not very logical or clear.

The **intensional spreadsheet**, proposed in this thesis, is expected to be suitable for many applications, especially for scientific ones, and to be suitable for users at all levels including ones who already have programming experience. The intensional spreadsheet not only retains most of the features of existing spreadsheets but also adds new features.

The intensional spreadsheet introduces **intensional logic** into spreadsheet design. This allows the spreadsheet to be considered as an entity varying in three different dimensions instead of as a group of many individual cells. The intensional spreadsheet language is based on a high-level declarative language called **Plane Lucid**, which is a functional programming language with intensional operators. Intensional spreadsheet programs are evaluated using **eduction** - a tagged demand-driven computation model, which is shown to be a natural way to evaluate spreadsheet programs. On the application side, the availability of **three dimensions, global variables,** and **user-defined functions** provides users with powerful problem solving techniques.

Examiners:

Professor William W. Wadge



Professor Michael Levy

Professor Gary G. Miller

Professor Charles G. Morgan

Table Of Contents

Abstract ii

Table Of Contents iv

List Of Tables vi

List Of Figures viii

Acknowledgements ix

1. Introduction 1

2. Background 5

 2.1 Spreadsheets 5

 2.2 The Intensional Programming Language Plane Lucid 10

  2.2.1 Intensional Logic and Intensional Programming 10

  2.2.2 The Intensional Language Lucid 12

3. An Intensional 3-D Spreadsheet 18

 3.1 Overview 18

 3.2 The spreadsheet Variable and Spatial Intensional Operators 20

 3.3 Global Variables And User-defined Functions 25

 3.4 Time Operation 31

 3.5 Advanced Intensional Operators 36

4. An Implementation of the Intensional Spreadsheet 43

 4.1 Principles of the Spreadsheet Interpreter 44

4.1.1 Eductive Computation 44

4.1.2 An Eductive interpreter for the Intensional Spreadsheet 46

4.1.3 The Implementation of the Spreadsheet Interpreter 48

4.2 Entering A Definition 50

4.2.1 Compilation of a Definition 51

4.2.2 Storage of Object Definitions 54

4.3 Evaluation of the Spreadsheet 55

4.4 Deletion and Modification of a Definition and Re-evaluation 56

4.4.1 Deletion 56

4.4.2 Modification 57

4.4.3 Re-evaluation 58

4.5 About dependencies and storage Management 58

5. Application And Comparison 59

5.1 A Scientific Application example 60

5.2 Comparison with Conventional Spreadsheets 64

5.2.1 Iteration 65

5.2.2 Goal Seeking 70

5.2.3 Performance Comparison 72

6. Conclusions and Further Work 73

Bibliography 76

Appendix A 78

Appendix B 86

List of Tables

Table 1.1 1

Table 1.2 2

Table 2.1.1 6

Table 2.1.2 8

Table 2.2.1 11

Table 2.2.2 16

Table 3.1.1 18

Table 3.2.1 21

Table 3.2.2 22

Table 3.2.3 23

Table 3.2.4 24

Table 3.3.1 26

Table 3.3.2 26

Table 3.3.3 29

Table 3.3.4 30

Table 3.4.1 32

Table 3.4.2 34

Table 3.4.3 35

Table 3.5.1 40

Table 5.1.1 63

Table 5.2.1 66

Table 5.2.2 67

Table 5.2.3 67

Table 5.2.5 70

Table 5.2.6 71

List of Figures

Figure 4.1.1 48

Acknowledgement

I wish to thank Professor William W. Wadge for suggesting the topic of this thesis and for his valuable guidance of my research.

CHAPTER 1

INTRODUCTION

Electronic spreadsheets are well-known as powerful problem-solving tools for many applications. Originally devised by Daniel Bricklin as calculation aids with simultaneous update capabilities, spreadsheets have been developed into user-friendly programming tools. Bricklin's VisiCalc (VisiCorp)[1], the first spreadsheet available on the market, was born of the observation that many problems are commonly solved with a calculator, a pencil, and a sheet of paper. It offered users the ability to enter rows and columns of information and perform many mathematical operations on groups of numbers. Computing is one pillar on which this structure rests and simultaneous updating of the memory cells is the other. For example, Table 1.1 shows a VisiCalc spreadsheet for a home budget;

	A	B	C	D
1	Home Budget			
2		Jan	Feb	Mar
3	Food	300	330	363
4	Clothing	200	220	242
5	Transport	100	110	121
6	Rent	400	400	400
7				
8	Total	1000	1060	1126

Table 1.1 a spreadsheet for the home budget

In this spreadsheet, the cells in row 8 ("Total") (except the first one) are defined by a formula to be the sum of the values of cells containing the various costs for each month, respectively. As soon as the user enters the formula, the spreadsheet calculates it automatically and shows the calculation result in the cell having the formula. Whenever the user changes any entry, the spreadsheet recalculates the values of the cells whose formula depends on the changed cell. For instance, Table 1.2

shows the new values if the cell C3 ("Food" for Feb.) in the above example is changed to 320.

	A	B	C	D
1	Home Budget			
2		Jan	Feb	Mar
3	Food	300	320	363
4	Clothing	200	220	242
5	Transport	100	110	121
6	Rent	400	400	400
7				
8	Total	1000	1050	1126

Table 1.2 the recalculated spreadsheet for the home budget

Subsequent to VisiCalc came the "second generation" spreadsheets such as SuperCalc (Sorcim/IUS)[2] and Multiplan (Microsoft)[5], which featured greater calculating power and the ability to run on a wide range of microcomputers. The arrival of the "third generation" spreadsheets such as Lotus 1-2-3 (Lotus Development Corp.)[3] was heralded by the release of the first integrated software products for microcomputers. As before, the new products offered analysis functions not found in earlier generation spreadsheets. The "fourth generation" spreadsheets, such as Symphony (Lotus Development Corp.)[4] released in 1984, included highly integrated packages but did not significantly improve functionality. Generally speaking, the main contribution of the spreadsheets released after VisiCalc is in end user interface techniques. In other words, although the new developments did expand and improve spreadsheet problem-solving abilities, they were not revolutionary. If we know one of them well, learning the others is easy.

Those four generations of spreadsheets will be called "conventional" spreadsheets. It is obvious that almost all of them were designed for business applications and for use by computer science novices. So most of the conventional spreadsheets have user friendly interfaces (certainly one of the most important considerations in spreadsheet design), but they have relatively low calculating ability and poor programming power. They appear to be inadequate for wider uses, especially in science and engineering. This limits the usefulness of a potentially powerful programming paradigm.

In conventional spreadsheets, indeed, there are not too many operations which can be applied within the formulas of cells, and most of the built-in functions are designed for business or financial use. Also, formulas contained by cells in a spreadsheet are often very simple mathematical expressions, which further restricts the complexity of calculations. In addition, almost none of the conventional spreadsheets allow user-defined functions and global variables. We will show the usefulness of these concepts in the next chapter of the thesis. Conventional spreadsheets are two-dimensional, that is, they are static in "time". Values of spreadsheet cells only depend on other cells in different spatial positions of the spreadsheet and there is no notion of variation of time. For many scientific applications, there are quantities which vary with time as well as with space.

When we study the structure of a conventional spreadsheet, the integrity of the whole spreadsheet is not very clear, because relationships between cells are restricted to relative or absolute references in formula definitions. No relation seems to exist between conventional spreadsheet concepts and current high level programming languages as regard syntax, semantics or programming practice. In many cases, this makes programming spreadsheets difficult.

These deficiencies of conventional spreadsheets form the main motivation for designing a new spreadsheet. This new spreadsheet is expected to be suitable for wide-ranging applications, especially scientific ones, and to be useful to various people including some who have had conventional programming experience. This new spreadsheet not only keeps most of the features of the conventional spreadsheets, but also has new features which overcome the deficiencies mentioned above.

One of the most important features of the new spreadsheet is that it is based on a high level intensional language -- Plane Lucid, which is an extension of the Lucid programming language[12]. In the new spreadsheet, all definitions of cells (in the new spreadsheet a formula is called a definition) as well as of user-defined functions and global variables, can be thought of as constituting a Plane Lucid-like program. A demand-driven computation model is used to implement the spreadsheet interpreter. When programming the new spreadsheet, a user should think of the whole spreadsheet as an

entity and the cells as the points inside the entity. Cells can be defined globally or locally by using intensional operators, and their definitions can be complex expressions in which various data types and operations are allowed. The introduction of global variables and user-defined functions makes the spreadsheet easier to program. The most powerful feature is that the new spreadsheet is three-dimensional instead of two dimensional; users can observe the values of cells at any time and search the desired results while time varies. This makes the new spreadsheet a revolutionary improvement over conventional spreadsheets.

In summary, since the new spreadsheet is a combination of existing spreadsheet techniques and a high level programming language, it is expected to be more powerful than conventional ones. Because it is implemented by a demand-driven method, it is expected to be a significant experiment in data flow computation.

In the next chapter, we give some necessary background on spreadsheets and the Plane Lucid programming language, which is the basis of this intensional spreadsheet design. In Chapter 3, the principle of the intensional spreadsheet will be described. The chapter will be concerned mainly with programming the spreadsheet; a simple description of the user-interface of the current implementation can be found in the Spreadsheet Command List in Appendix B. In Chapter 4 the current implementation of the new spreadsheet will be described. A practical application example and some comparisons with conventional spreadsheets will be given in Chapter 5. The conclusions and further work needed are the contents of the last chapter.

CHAPTER 2

BACKGROUND

2.1. Spreadsheets

Briefly, a spreadsheet is a program that provides a large grid of cells into which numerical data, textual data and formulas can be entered. A cell that contains a data item causes the program to display that data; a cell that contains a formula causes the program to display the formula's value and to recalculate the formula automatically whenever any cell on which it depends is changed. A spreadsheet permits a decision maker to calculate with accounting and financial data, the management of a product line or the administration of an account. It also permits the analysis and prototyping of systems. Both at the decision making level and at the system and procedure level, people benefit from the capabilities embedded in a spreadsheet.

In a spreadsheet, the computer's screen becomes a display window. The user can scroll this window in all four directions to look at any part of the sheet. The spreadsheet is organized as a grid of rows labeled (say) as 1, 2, 3 ... and columns labeled (say) as 1, 2, 3 ... . At each intersection of a row and a column there is a cell with a coordinate identifier such as R1C1, R2C5, R3C10 and so on. The total number of rows and columns that a spreadsheet may have depends on the design of the spreadsheet and the amount of memory in the computer. The display width of a column can be one character wide or wider, at the choice of the user. The screen usually consists of two basic areas: the control panel and the window. The window shows the displayed cells. The control panel includes an input line to input commands and data, a prompt line to display various interactive messages and an editor subwindow to edit formulas. For example, Table 2.1.1 shows the scheme of the Multiplan[5] spreadsheet.

current cell	formula				
spreadsheet name					
	1	2	3	4	5
1					
2					
3					
4					
5					

Table 2.1.1 a Multiplan scheme

Through the user-interface of a spreadsheet, users can enter various commands to program and manipulate the spreadsheet. Although various spreadsheets have different programming styles and manipulation facilities, they commonly provide some similar basic commands which are briefly described below.

Using the *edit* command, a user can define cells in the spreadsheet. Into each cell the user can enter one of three types of data: numbers, textual strings (which are normally used to show the meanings or the titles of rows or columns in the spreadsheet) and formulas. A formula is the most complex data type, which is usually an arithmetic expression. The expression consists of constants, references to other cells and operators. The operators are usually arithmetic operators: addition, subtraction, multiplication, division and exponentiation, or they are built-in functions defined by arithmetic operations. The ability of a cell to contain a formula is the crux of the spreadsheet's ability. It is this ability that makes "what-if" analysis a simple-to-use and powerful tool. Implicit in the use of formulas is an understanding of the point of reference. The references to cells in a formula can be absolute or relative to the cell containing the formula. The reference mode is important if the formula is copied to other cells.

Using the *copy* command, a user can copy one or more cells from one region of the spreadsheet to another. After the copy operation, a cell in the destination region has the same or different content(a number, a string or a formula) of the corresponding cell in the source region, according to the reference mode and the copy strategy of the particular spreadsheet design. When a formula is

copied to another cell, an absolute reference still points to the same cell, but a relative reference probably points to a different cell. In other words, in the later case, the copy operation does not change the intended meaning of the formula. Usually, the user can change the type of addressing of a cell, and hence change the effect of the copy operation. If all references in a formula are addressed absolutely, the user may be requested to indicate the type of copy. An absolute copy makes the references in the formula of a destination cell point to the same cells as they did in the source cell, while a relative copy makes all references in the formula of the destination cell to point to the cells which have the same relative positions to the destination cell as to the source cell before being copied. For example, when the formula "R1C1 + 10" is copied from cell R2C2 to R5C6, with the absolute copy strategy the formula in cell R5C6 is still "R1C1 + 10" after the copy operation, but with the relative copy strategy the formula becomes "R4C5 + 10".

Using the *delete* command, a user can delete the definitions of any number of cells from the spreadsheet to make them become undefined. After the deletions, the values of the cells whose formulas depend directly or indirectly on one of the deleted cells become invalid and error messages are displayed in these cells.

Using the *move* command, a user can do two kinds of operations. One is to move the current cell (which most programming and manipulating operations work on) to another cell, and the other is to move the current display window from a part of the spreadsheet to another part which will appear on the screen.

Using the *input* and *output* commands, a user can print out the spreadsheet or store it in a file, and later he/she can recover the stored spreadsheet to the screen.

The calculation of the values of spreadsheet cells occurs every time one or more cells are defined. This usually involves the calculations of the values of the newly defined cells and other cells whose calculations failed before because some cells they depended on were undefined. The recalculation of a spreadsheet occurs every time the contents of some cells are changed. It usually involves not

only the calculations of the newly modified cells, but also the recalculations of some of the cells which were calculated before, because the modifications may make the value of those cells invalid. A naive way of dealing with recalculation is to recalculate all defined cells in the spreadsheet, no matter what cells they are, even if their values may not be affected by the modification. This is obviously inefficient, but it makes the implementation simple so that now many spreadsheets recalculate this way. The recalculation is performed by starting at the upper left corner of the sheet and moving downward and to the right until the lower right corner of the sheet is reached. Another way is to calculate only the cells whose values are made invalid by the modifications; other cells which are not affected still keep their previous values. It is clear that the second way is more efficient because no unnecessary calculations are involved. To use the second method it is necessary to determine which cells that are affected by the modification. This requires finding dependencies among the cells, and it is one of the most difficult problems faced in spreadsheet implementations.

The following is a simple Multiplan spreadsheet similar to the example VisiCalc spreadsheet in Table 1.1 and Table 1.2, which is a part of the home budget spreadsheet. Table 2.1.2(a) shows a part of the spreadsheet after the user entered the budgets for January.

R7C2	=R[-5]C+R[-4]C+R[-3]C+R[-2]C			
	Home Budget			
	1	2	3	4
1		Jan.		
2	Food	300		
3	Clothing	200		
4	Transport	100		
5	Rent	400		
6				
7	Total	1000		

Table 2.1.2(a) part of a Multiplan spreadsheet for a home budget.

In the formula of the current cell R7C2, R[i] refers to the row at offset i from the current row (i can be positive or negative), and C refers to the current column number. It is easy to see that relative references are used in the formula. When the user enters data items or formulas for the rest of the

budgets, he/she does not need to enter the formulas for R7C3(the total for February) and R7C4(the total for March) again. What he/she needs is only to copy the formula from cell R7C2 to cell R7C3 and R7C4. Since the copy operation in Multiplan is always relative[5], it causes no problems. Table 2.1.2(b) shows the whole spreadsheet program.

R7C4	=R[-5]C+R[-4]C+R[-3]C+R[-2]C			
	Home Budget			
	1	2	3	4
1		Jan.	Feb.	Mar.
2	Food	300	330	363
3	Clothing	200	2200	242
4	Transport	100	110	121
5	Rent	400	400	400
6				
7	Total	1000	1060	1126

Table 2.1.2(b) a Multiplan spreadsheet for a home budget.

Afterwards, when the user wants to modify the food budget for February from 330 to 320, instead of modifying cell R2C3 directly, he/she first deletes the definition of cell R2C3. In this case, cell R2C3 becomes undefined and the value of cell R7C3 becomes invalid. Table 2.1.2(c) shows this situation.

R2C3				
	Home Budget			
	1	2	3	4
1		Jan.	Feb.	Mar.
2	Food	300	#N/A	363
3	Clothing	200	220	242
4	Transport	100	110	121
5	Rent	400	400	400
6				
7	Total	1000	#REF!	1126

Table 2.1.2(c) the spreadsheet for the home budget after a deletion from (b)

Finally, the user enters the new data for cell R2C3, which is immediately shown on the spreadsheet, and the value of cell R7C3 is recalculated. Table 2.1.2(d) shows the spreadsheet after the modification and the recalculation.

R7C3	=R[-5]C+R[-4]C+R[-3]C+R[-2]C			
Home Budget				
	1	2	3	4
1		Jan.	Feb.	Mar.
2	Food	300	320	363
3	Clothing	200	220	242
4	Transport	100	110	121
5	Rent	400	400	400
6				
7	Total	1000	1050	1126

Table 2.1.2(d) the modified spreadsheet for home budget from (b).

2.2. The Intensional Programming Language Plane Lucid

The intensional programming language Plane Lucid is the basis for the intensional spreadsheet to be discussed in this paper. Before we describe the language, we introduce the concepts of intensional logic and intensional programming, which are the fundamentals of Plane Lucid as well as the intensional spreadsheet.

2.2.1. Intensional Logic and Intensional Programming

Intensional logic is concerned with assertions and other expressions whose meaning depends on an implicit context[7]. For example, the expression "five degrees less than yesterday's temperature" denotes a numerical value. This value clearly depends on a numerical quantity called temperature. It also depends on the time of utterance, and on the place even though there is no explicit reference to either of the two parameters. Although the value of the expression above depends on that of temperature, we cannot conclude anything about the value even if today's temperature is known, because the value of the expression on any given day depends on the value of temperature on the previous day. In fact, the quoted expression seems to correspond to a mathematical expression of the form y(t) - 5, where t is temperature and the function y corresponds to yesterday's. It is obvious, however, that there is no function y which makes the value of the mathematical expression corresponding to that of the cited phrase.

For many years, logicians considered examples such as the one above to be simply further evidence of the nonmathematical and illogical nature of natural languages. In relatively recent years logicians discovered a more direct and natural way to capture formally these "context sensitive" values. The solution to this long-standing puzzle is based on the distinction between what is called the **extension** and **intension** of expressions.[8] The **extension** of an expression is the value in a given context. A natural language expression can obviously have different extensions in different contexts. The **intension** gathers all these different extensions together and captures the way in which the extensions depend on their contexts. In other words, the intension is the function which assigns to each context the value of the expression in that context.

Consider the example cited earlier. The intension of "temperature" is essentially a table giving the temperature on each day in question, which may look like this:

Date	Month	Year	temperature
30	Dec.	85	23
31	Dec.	85	21
1	Jan.	86	25
2	Jan.	86	23
3	Jan.	86	19
---	---	---	---

Table 2.2.1(a)

The intension of "five degrees less than yesterday's temperature" is a similar table:

Date	Month	Year	temperature
31	Dec.	85	18
1	Jan.	86	16
2	Jan.	86	20
3	Jan.	86	18
4	Jan.	86	14
---	---	---	---

Table 2.2.1(b)

In fact the latter intension can be obtained from the former by first subtracting 5 from all the temperatures in the temperature column and then advancing all the dates in the time columns by one day. A mathematically respectable function y which accurately captures the meaning of the phrase "yesterday's" can be found. The function y mapping intensions to intensions simply increases all dates by one day.

Programmers very often think about programs intensionally in conventional languages. For example, when examing the body of a procedure declaration, we cannot know exactly what values parameters have because that depends on implicit parameters, the "call" of the procedure in question. It is possible to extend a conventional language by adding some intensional operators, but this extended language would not be based on intensional logic.

By "intensional programming" we mean programming in a language which is both a programming language and, at the same time, a formal system based on intensional semantics. Intensional language programmers must be encouraged to think intensionally and be provided with context switching operators which allow values from different contexts to be combined without explicit context manipulation.

2.2.2. The Intensional Language Plane Lucid

Plane Lucid is an extension of the Lucid programming Language, or Plane Lucid is a member of the Lucid family.

Lucid is a dataflow programming language invented by W.W.Wadge and E.A.Ashcroft[9-12]. The language is a purely declarative (nonprocedural) language in which iterative algorithms can be expressed easily and naturally; it is well suited for expressing algorithms based on a dataflow view of computation - one in which data flows through a network of asynchronously operating processing stations.

The Lucid language is based on intensional logic in which the values (extensions) of expressions and variables depend on an implicit natural number - a (time) parameter that is not manipulated explicitly. In this sense, the operators of Lucid are used as intensional operators. In Lucid, a programmer thinks intensionally. In a Lucid program, the expressions and variables have the values varying in time, i.e. the value of an expression or a variable depends on the time point at which the expression or variable is evaluated.

A natural extension of the language Lucid is formed by allowing intensions to vary in space as well as in time. We will show that this makes the language more problem-oriented and versatile. The idea is to change the semantics of Lucid by allowing the values of expressions or variables to be time sequences of elements that themselves vary in space. If Lucid is considered as an one(time)-dimensional language, the extended language may be considered as a multi-dimensional(a time dimension and one or more space dimensions) language. In general, the language allows an arbitrary number of space dimensions. As a particular case, there is a three dimensional (a time dimension and two space dimensions) version called **Plane Lucid**. Plane Lucid is the basis of the intensional spreadsheet discussed in the paper.

Syntactically a simple Lucid program corresponds to an expression(the output) qualified by a block of definitions. A simple definition consists of a left-hand side, which is the name of a variable defined by the expression on right-hand side of the equality symbol. The complete syntax of Plane Lucid can be found in Appendix A. For example, in the Plane Lucid program

x + y where x = 1 ; y = 2 ; end

the syntactic objects 1 and 2 correspond to two expressions which are constant in space as well as in time, i.e. they have the same value 1 or 2 at any spatial and time points, respectively. Thus the definitions x = 1 and y = 2 define the variables x and y to be constants (1 or 2) in space and in time. The result of the program is still a constant, 3, i.e. no matter where and when the expression x+y in the program is evaluated, the evaluation always produces the value 3.

For more complicated programs, Plane Lucid provides a number of operators that permit expressions or variables to vary in space as well as in time. Those operators are called the *intensional operators.* For all dimensions (two spatial dimensions and a time dimension), there are five kinds of intensional operators to do different context switching work. The intensional operators for the horizontal dimension in space are: **right**, **left**, **hsby**(for "horizontally succeeded by"), **hpby**(for "horizontally preceded by") and **side**. Similarly, the corresponding intensional operators for the vertical dimension in space are: **up**, **down**, **vsby**(for "vertically succeeded by"), **vpby**(for "vertically preceded by") and **ledge**. The corresponding intensional operators for the time dimension are: **next**, **before**, **fby**(for "followed by"), **pby**(for "preceded by") and **first**. All other operators in Plane Lucid are (space and time) pointwise, i.e. they operate on the value(s) of their operand(s) at every spatial and time point. In the following, we describe the operational meanings of the spatial intensional operators in the horizontal dimension. The operational meanings of others are not difficult to understand from these descriptions.

The unary operator **right** shifts the context one point right in space. The value of **right** x, where x is any expression, at a given spatial point and a given time is the value of x at the point immediately to the right of the given point in space, and at the same time.

Similarly, the unary operator **left** shifts the context one point left in space. The value of **left** x at a given spatial point and a given time is the value of x at the point immediately to the left of the given point in space, and at the same time.

The binary operator **hsby** combines two different spatial operations, the **left** operation and the normal pointwise operation. The value of x **hsby** y at a given spatial point and a given time is the value of y at the spatial point immediately to the left of the given one and at the same time (which is the same as a **left** operation), if the given point's horizontal coordinate is positive; otherwise, it is the value of x at the same spatial point and at the same time. For example, the definition

x where x = 0 **hsby** (x + 1) ; end ;

defines x as a variable whose value at any spatial point is equal to the horizontal coordinates of that point, no matter what the time, if the horizontal coordinate is not negative; otherwise the value of x is 0.

Similarly, the other binary operator **hpby** combines the **right** operation and the normal pointwise operation. The value of **x hpby y** at a given spatial point and a given time is the value of x at the spatial point immediately to the right of the given one and at the same time (which is the same as **right** operation), if the given point's horizontal coordinate is negative; otherwise, it is the value of y evaluated at the same spatial point and at the same time. For example, the definition

x where x = (x-1) **hpby** 0 ; end

defines x as a variable whose value at any spatial point is equal to the horizontal coordinates of that point, no matter what the time, if the horizontal coordinate is not positive; otherwise the value of x is 0. Therefore the combination of the above two definitions of x

x where x = (x-1) hpby (0 hsby (x+1)) ; end

defines x as the index of the whole horizontal dimension.

The negative space coordinates permit intensional programming using a full plane. The negative time coordinates are also expected to be useful for some kinds of applications.

The unary operator **side** evaluates the value of its operand at the horizontal coordinate 0. The value of **side x** at a given spatial point and a given time is the value of x at the spatial point which has the same vertical coordinate as the given point but has the horizontal coordinate 0, and at the same time.

Table 2.2.2 lists the informally operational meanings of all intensional operators in Plane Lucid. Their formal semantics can be found in Appendix A of the paper.

Operations	Evaluation Results
right x	x at the spatial point right to the evaluation point
up x	x at the spatial point above the evaluation point
next x	x at the time point next to the evaluation point
left x	x at the spatial point left to the evaluation point
down x	x at the spatial point below the evaluation point
before x	x at the time point before the evaluation point
x hsby y	left y if the current horizontal coordinate >0 else x
x vsby y	down y if the current vertical coordinate >0 else x
x fby y	before y if the current time >0 else x
x hpby y	right x if the current horizontal coordinate <0 else y
x vsby y	up x if the current vertical coordinate <0 else y
x pby y	next x if the current time <0 else y
side x	x at the spatial point with horizontal coordinate 0
ledge x	x at the spatial point with vertical coordinate 0
first x	x at the time 0

Table 2.2.2 the operational meanings of intensional operators

As an example, the following Plane Lucid program solves Laplace's equation over a two dimensional space.

```
s
  where
    s = if ELECTRODE then POTENTIAL else 0 fby avg(s) fi ;
    avg(M) = (left M + right M + up M + down M) / 4 ;
  end ;
```

Here, ELECTRODE is the characteristic predicate of a region of the plane in which an electrode is present, and at each of these points the electrical potential is fixed as the value of potential. The initial potential at all other points is 0. At each step of the iteration, their new value is the average of the potential of the four surrounding space points at the previous time instant.

In addition, there are also some advanced intensional operators in Plane Lucid, which we will discuss in Chapter 3. Apart from these operators, the other extremely important and useful feature of Plane Lucid is user-defined functions. The concept of the user-defined functions in Plane Lucid is the

same as in Lucid, which is described in the book on Lucid by Wadge and Ashcroft[12] in great detail.

In Plane Lucid, variables vary pointwise in time and two spatial dimensions. If all the values of a variable at a time point are considered to be a group, it is called a **field.** In Plane Lucid , **fields** have finite(two) dimensionality but infinite size in each dimension, in other words a field is a matrix with infinite rows and columns. Therefore, a variable can also be thought of as varying only in time, and at different time points the variable has different fields as its values. For example, the following Plane Lucid program enumerates the prime numbers using the "sieve" method of Eratosthenes. Here, the operator **hwherever** is an advanced intensional operator for the horizontal dimension of space. It scans the values of its arguments from horizontal coordinate 0 to left and right simultaneously and passes on only those extensions of its first argument for which the corresponding extensions of its second argument (a predicate) are true.

```
P
  where
    P = side D ;
    D = N2 fby ( D hwherever ( D mod P ne 0 )) ;
    N2 = 2 hsby N2 + 1 ;
  end ;
```

In the program, a field can be thought of as a vector (along the horizontal dimension). The variable N2 is constant in time, whose field is the vector < 2, 3, 4, 5, ... > if the negative coordinates are ignored. At time 0, the field of the variable D is the vector < 2, 3, 4, 5 ... > from N2, so **side D** at time 0 is 2 ; at time 1, the field of D is the vector < 3, 5, 7, ... > or all odd numbers by doing the **hwherever** operation on D with the predicate **D mod P ne 0** at the previous time point, so **side D** is 3 at this time; in the same way, the field of D at time 2 and 3 are vectors < 5, 7, 11, 13, 17, ... > and < 7, 11, 13, 17, ... > respectively, and the results of **side D** are 5 and 7. As time is increasing, eventually all prime numbers will be enumerated.

CHAPTER 3

AN INTENSIONAL 3-D SPREADSHEET

In the following chapters, we describe a new spreadsheet which is designed using intensional logic and is implemented by using an demand-driven evaluation technique. In this chapter the programming features of the new spreadsheet are described and the relation between the spreadsheet and the underlying programming language is discussed.

3.1. Overview

This new spreadsheet, like conventional spreadsheets, is a display-oriented calculation tool. The main part of the spreadsheet on the screen is a group of grids divided by rows and columns. Those grids are called **cells**. The Table 3.1.1 shows what the spreadsheet looks like on the screen.

*Name**	---	L2	L1	0	R1	R2	---

U2							
U1							
0							
D1							
D2							
---							

* user defined spreadsheet name

Table 3.1.1 a window of the intensional spreadsheet

Unlike other spreadsheets, which allow only the first quadrant or positive row numbers and column numbers to be used to define cells, the new spreadsheet allows cells to be defined on a full plane. For example, a cell at coordinate U2L2(row 2, column -2) lies in the second quadrant, a cell at coordinate U1R1 lies in the first quadrant and so on, where **U** means "Up", **D** means "Down", **L** means

"Left" and **R** means "Right" with respect to the coordinate 0.

Like conventional spreadsheets, the new spreadsheet provides users with several commands to program and manipulate the spreadsheet. Those commands includes **move, width, copy, delete, edit, evaluate** among others. The **move** command changes the current cell, as indicated by the user, and moves the current display window to include the new current cell in the screen. The **width** command changes the width of a column to fit those data items which are longer than the normal cell width. The **copy** command copies the definitions of a group of cells into any other groups of cells according to their relative positions in each group. The **delete** command deletes the definitions and/or values from one or more cells. The **edit** command enters or modifies the definition for a cell, a global variable or a user-defined function. Whenever a definition is modified, the values of the cells which are affected by this modification are re-evaluated and redisplayed on the screen. The **evaluate** command evaluates whatever cells the user requires and displays the result values of the cells on the screen in one step.

Since this chapter concentrates on explaining the principle of the **intensional** spreadsheet, and on programming the spreadsheet intensionally, the user interface will not be discussed further. It is described in the Command List in Appendix B of this paper.

As was mentioned in the previous chapter, the design of this spreadsheet is based on a high-level intensional language called Plane Lucid, which is an extension of the Lucid programming language. Most of the intensional operators in Plane Lucid were explained in the previous chapter, and a brief description of the language can be found in Appendix A, in which the syntax of Plane Lucid is given and the semantics for most of the operators are explained.

The syntax of definitions for cells, global variables and user-defined functions in spreadsheet programs is the same as that in Plane Lucid programs. That is, the format of a definition in a spreadsheet program is exactly the same as the format of a definition of a variable or a function in a Plane Lucid program, which allows **where** clauses for local definitions when necessary. For example,

a cell may be defined as

1 + 2 ;

or

a + b where a = 1; b = 2 ; end ;

and a user-defined function may be defined as

f(x,y) = x + 10 * y ;

or

f(x,y) = x + a where a = 10 * y ; end ;

3.2. The Spreadsheet Variable and Spatial Intensional Operators

Unlike conventional spreadsheets which cannot be referred to as a whole from within because the cells in them are dealt with individually, this new spreadsheet, consisting of infinite rows and columns of cells, is considered as an entity which is called **the spreadsheet variable.** The spreadsheet variable varies in a two-dimensional space, called the **spreadsheet plane,** as well as in time. Therefore a cell in the spreadsheet corresponds to values of the spreadsheet variable at a particular point in space, or the value of the cell is the value of the spreadsheet variable at this spatial point at a given time.

Apart from ordinary operators, intensional operators are used to reference other cells intensionally in the definitions of cells. Basically these operators are **left** , **right** , **up** and **down.** If the spreadsheet variable is named S and a cell in it is defined as **left S** , the result of evaluating the cell is the value of the cell which is on the immediate left of the cell being evaluated. This is because according to the semantics of the operator **left** the result of **left S** should be the value of S at the point to the left of the current point, or generally speaking, the value of **left E,** where E is an expression in a cell, is the value E would have in the cell immediately to the left. For example, Table 3.2.1 shows a spreadsheet program, using the intensional operator **left,** to calculate the home budget exam-

ple that we showed in Table 1.1. In the program we assume that the entries for food, clothing and transport would be increased by 10% every month from January but that the rent would not be changed. The function csumn(n), which will be described in a later section of the chapter, calculates the sum of n cells above the current cell.

S	0	R1	R2	R3
0	'Home Budget'			
D1		'Jan.'	'Feb.'	'Mar.'
D2	'Food'	300	left S*1.1	left S*1.1
D3	'Clothing'	200	left S*1.1	left S*1.1
D4	'Transport'	100	left S*1.1	left S*1.1
D5	'Rent'	400	left S	left S
D6	'Total'	csumn(4)	csumn(4)	csumn(4)

Table 3.2.1 (a) the spreadsheet program for a home budget

S	0	R1	R2	R3
0	Home Budget			
D1		Jan.	Feb.	Mar.
D2	Food	300	330	363
D3	Clothing	200	220	240
D4	Transport	100	110	121
D5	Rent	400	400	400
D6	Total	1000	1060	1126

Table 3.2.1 (b) the result of the spreadsheet program in (a)

We can interpret the operators **right, up** and **down** in a similar way. If a cell is defined as **right E,** the value of the cell is the value of E in the cell on its immediate right; if a cell is defined as **up E (down E)** the value of the cell is the value of E in the cell immediately above (below);

For example, in numerical analysis we often need to calculate the value of a point in a plane from the values of its adjacent points. Table 3.2.2, as an example, shows how to calculate the value of a point(cell) in the spreadsheet from that of its eight adjacent points(cells) by using the intensional operators introduced above. In the example, the cell 0 is defined as

(up S + up right S + right S + down right S +
down S + down left S + left S + up left S) / 8

i.e. the value of cell 0 is the average of its neighbors.

S	L1	0	R1
U1	1	2	3
0	8	*	4
D1	7	6	5

* the definition of cell 0

Table 3.2.2(a) the spreadsheet program obtaining the average value

S	L1	0	R1
U1	1	2	3
0	8	4.5	4
D1	7	6	5

Table 3.2.2(b) the result of (a)

Apart from those primitive intensional operators, some corresponding built-in intensional functions are also provided. They include **right**(x,p), **left**(x,p), **up**(x,p) and **down**(x,p), where x is a variable and p is an integer variable or constant. For instance, the operation "right(x,2)" is equivalent to "right right x " and the operation "right(down x,2)" is equivalent to "right right down x" and so on. When the actual parameter x is the spreadsheet variable shorter forms of those operations are defined. For example, we can define built-in global variables (global variables will be introduced in the next section) **Ri, Li, Ui** and **Di** as

Ri = right(S,i)
Li = left(S,i)
Ui = up(S,i)
Di = down(S,i)

and more complex variables **UiRj, UiLj, DiRj** and **DiLj** as

UiRj = up(right(S,j),i) = right(up(S,i),j)
UiLj = up(left(S,j),i) = left(up(S,i),j)
DiRj = down(right(S,j),i) = right(down(S,i),j)
DiRj = down(left(S,j),i) = left(down(S,i),j).

where i,j are some non-negative integers. Table 3.2.3 is an example showing the use of those variables. The spreadsheet program in the example defines the value of each non-diagonal cell of the

symmetrical (5×5) matrix as the sum of the values of the diagonal cells which are in the same row or column.

S	R1	R2	R3	R4	R5
D1	1	L1+D1	L2+D2	L3+D3	L4+D4
D2	U1R1	2	L1+D1	L2+D2	L3+D3
D3	U2R2	U1R1	3	L1+D1	L2+D2
D4	U3R3	U2R2	U1R1	4	L1+D1
D5	U4R4	U3R3	U2R2	U1R1	5

Table 3.2.3(a) a spreadsheet program for a symmetric 5×5 matrix

S	R1	R2	R3	R4	R5
D1	1	3	4	5	6
D2	3	2	5	6	7
D3	4	5	3	7	8
D4	5	6	7	4	9
D5	6	7	8	9	5

Table 3.2.3(b) the result of (a)

Later, we will introduce other forms of those intensional operations; their formal definitions can be found in Appendix A.

The relative references to other cells in conventional spreadsheets seem similar to what the intensional operators do. For example, the operation **right S** is similar to the reference **RC[+1]** in Multiplan[5] and the operation **down right(S,2)** is similar to **R[-1]C[+2]**, because they both get the value of the cells on the right or two cells away (diagonally). But there are some significant differences between these approaches. Firstly, the former is the result of the intensional *operator* being *applied* to the spreadsheet variable, while the latter is a *reference* to an *individual cell.* The intensional operator, say **left**, can be applied to a more complicated expression which may include other intensional operators, while the Multiplan form is used only to access the value of a particular cell. Secondly, the spatial index parameters in the built-in intensional functions, such as p in right(S,p) are considered as variables, while the relative references can only be constants. The advantages of using the intensional operators will become clearer, when more operators, global variables

and user-defined functions are introduced.

In the intensional spreadsheet, users are more or less forced to think intensionally about space and time, but sometimes it seems that a "global" non-intensional view of the spreadsheet is also appropriate. For example, when the evaluation of a cell needs to access a value at an absolute space point instead of a point relative to the current cell, no intensional operators we introduced up to now can do it. Therefore, two spatial operators, **side** and **ledge**, are introduced. The operation **side** E always returns the value of the expression E evaluated at the spatial point with the horizontal coordinate 0 and the same vertical coordinate. Similarly the operation **ledge E** always returns the value of E evaluated at the spatial point with the vertical coordinate 0 and the same horizontal coordinate. An auxiliary operator, **home**, can be defined using these two operators. The operation **home E** is equivalent to the operation **side ledge E**. The result of the operation is the value of E at the origin (coordinate 0). For example, Table 3.2.4 is a spreadsheet program to calculate the instantaneous speed of an accelerating object at every second. In the program, the cell D3R1 is defined as the acceleration(25m/s^2) which other cells in the row D1 (for speed) and D2 (for distance) refer to. It is easy to see that when the user changes the acceleration, he/she need not change the definitions of other cells to get the new results.

S	0	R1	R2	R3	R4
0	'Time'	1	2	3	4
D1	'Speed'	home(D3R1)*U1	home(D3R1)*U1	home(D3R1)*U1	home(D3R1)*U1
D2	'Distance'	0.5*U1*U2	0.5*U1*U2	0.5*U1*U2	0.5*U1*U2
D3	'Accel'	25			

Table 3.2.4(a) a spreadsheet program showing speeds and distances

S	0	R1	R2	R3	R4	R5
0	Time	1	2	3	4	5
D1	Speed	25	50	75	100	125
D2	Distance	12.5	50	112.5	200	312.5
D3	Accel	25				

Table 3.2.4(b) the evaluation result of (a)

An interesting observation is that the expression **home(DiRj)**, such as **home(D3R1)** in this program, just accesses the value of cell **DiRj**, because the home operator moves the evaluation point back to the origin before the expression D3R1. This, in a sense, makes the notation for accessing particular cells similar to that of conventional spreadsheets (such as RiCj in Multiplan), but they have totally different operational meanings.

3.3. Global Variables and User-defined Functions

While considering the spreadsheet variable as a two-dimensional array, we may think that a spreadsheet program is similar to a program in an ordinary high-level programming language where only one array variable (the spreadsheet variable) is used. All computations in a spreadsheet program are done by operating on the cells of the spreadsheet variable , which is in some sense the way conventional spreadsheets work. It is obvious that in most cases using only one variable in a program, even if it is an array variable, is inadequate. Therefore, it is necessary to introduce global variables into spreadsheet programs.

A global variable is similar to the spreadsheet variable in that it also varies in the spreadsheet plane as well as in time. The main difference between a global variable and the spreadsheet variable is that a global variable is defined globally with a single expression, while the spreadsheet variable is defined pointwise by defining its cells. For instance, if a global variable, say g, is defined as **g = 5**, it means that g is the variable with the value 5 at every point of the plane at any time; if g is defined as **g = 0 hsby g+1**, where **hsby** is an intensional operator defined in Plane Lucid, it means that g is the variable that varies along the horizontal dimension, increasing by one with each column. In programming, more than one global variable can be used in a spreadsheet program, and any other variables, including the spreadsheet variable and functions, can appear in a global variable's definition.

In the following, we give two examples showing the use of global variables. The first example is a spreadsheet program to calculate the approximate values of the base *e* of natural logarithms using

the Maclaurin series

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \ldots + \frac{1}{n!} \ldots \quad (n \to \infty).$$

In the program, the global variable **n** is used as the index of the series. The larger **n** is, the closer the result is to the exact value of *e*.

```
n = 0 hsby n + 1 ;
```

S	0	R1	R2	R3	R4	R5
0	'index'	n	n	n	n	n
D1	'factorial'	1	L1*U1	L1*U1	L1*U1	L1*U1
D2	'*e*'	2	L1+1/U1	L1+1/U1	L1+1/U1	L1+1/U1

Table 3.3.1(a) a spreadsheet program showing the Maclaurin series of *e*

S	0	R1	R2	R3	R4	R5
0	index	1	2	3	4	5
D1	factorial	1	2	6	24	120
D2	*e*	2.0000	2.5000	2.6666	2.7087	2.7166

Table 3.3.1(b) the evaluation result of (a)

In the second example, we give a spreadsheet program to construct the symmetric matrix which we constructed in Table 3.2.3 now using global variables. The program shows that when global variables are introduced, the spreadsheet becomes more powerful and the spreadsheet programming becomes more convenient. Table 3.3.2 shows the new spreadsheet program.

```
x = 0 hsby 1 + x ;
y = y + 1 vpby 0 ;
A = L(x-y) + D(x-y) ;
```

S	0	R1	R2	R3	R4
0	1	A	A	A	A
D1	A	2	A	A	A
D2	A	A	3	A	A
D3	A	A	A	4	A
D4	A	A	A	A	5

Table 3.3.2 another spreadsheet program constructing the 5×5 matrix

In the program, the global variable **x** is defined to be the index of the horizontal dimension; the global variable **y** is defined to be the index of the vertical dimension. Instead of defining the cells not in

the main diagonal individually as in Table 3.2.3, we define a global variable **A**, used to define all the cells on the right and left of the main diagonal. In the definition of **A**, instead of using a constant, we use the expression **x-y** as the actual parameter of the spatial intensional function calls, L(x-y) and D(x-y), where **L(p)** and **D(p)** are the short forms of the function calls left(S,p) and down(S,p). Using global variables in definitions of cells makes the program clearer and the programming simpler. Designing a spreadsheet program like this example using conventional spreadsheets will be difficult.

The example above also shows how the spreadsheet variable can be used in the definition of a global variable. Conversely, global variables can also be used in the definitions of cells which represent the spatial points of the spreadsheet variable. The question arises: are those two types of variables, i.e. global variables and the spreadsheet variable, interchangeable? If so, we need not have a special spreadsheet variable; instead, any global variable may be assigned as that variable whose values can be shown on the screen, and this can be substituted by another global variable later on. If so, it will undoubtedly enhance the capability of the spreadsheet, but to do so is not very easy. First, we need to find a method to deal with the null cells of the spreadsheet variable, (the cells which are not defined yet), when it is turned into a global variable. Fortunately, the spreadsheet is evaluated by a demand-driven technique; so if during the evaluation, the values of a global variable (the former spreadsheet variable) at spatial points corresponding to the null cells are not needed, the evaluation may be successful; but when the value of a null point is demanded, an evaluation error is caused. If that happens, the null point must be defined in some way. On the other hand, it seems that we need to modify the syntax of the basis language to allow global variables to be defined not only globally but also individually, that is, a spatial point of a global variable should be allowed to be defined individually. In fact, using the if-then-else-fi operator and some auxiliary variables we may define a global variable individually without changing the current grammar. For example, if we want to define the global variable g having the constant value 1 at all spatial points except at spatial point 0, at which it is defined as 0, it can be done by the definition

```
if x eq 0 and y eq 0 then 0 else 1 fi
  where
    x = 0 hsby x + 1 ;
    y = 0 vsby y + 1 ;
  end ;
```

where only the first quadrant is considered. It is obvious that when many points need to be defined individually, this method is very tedious. We should find some kind of representation like the "case" statement in conventional languages to make things simple. In addition, when a global variable becomes the spreadsheet variable, since it has definitions at all spatial points of the spreadsheet plane, it means that there are an infinite number of cells in the spreadsheet which have been defined. It is surely impossible to store those definitions for all cells, so another problem is to determine which cell's definition should be stored individually and which cell's should be shared with others.

In our spreadsheet programs inputs are dealt with in such a way that when an undefined global variable appears in the definition of a cell , it is considered as an input variable. An input variable, like other global variables, is also a variable varying in space and time. Instead of requiring all the values of an input variable sequentially in space and time, the values of an input variable in a spreadsheet program are always required individually, i.e. only when an individual value of the input variable is demanded during the evaluation, is the user asked to input a value for it at some particular spatial and temporal point. For example, in the spreadsheet program for instant speeds in Table 3.2.4(a), if we do not define the acceleration to be a constant (it was 25 m/s) and instead let it be an input variable, then the program can show the instant speeds and distances at various accelerations. Table 3.3.3(a) shows this modified spreadsheet program and (b)-(d) show the result when the input values of the acceleration are 10, 20 and 30, respectively.

S	0	R1	R2	R3	R4	R5
0	'Time'	1	2	3	4	5
D1	'Speed'	side D3U1	side D3*U1	side D3*U1	side D3*U1	side D3*U1
D2	'Distance'	0.5*U1*U2	0.5*U1*U2	0.5*U1*U2	0.5*U1*U2	0.5*U1*U2
D3	Acceleration					

Table 3.3.3(a) a modified spreadsheet program from Table 3.2.4

Acceleration input value = 10

S	0	R1	R2	R3	R4	R5
0	Time	1	2	3	4	5
D1	Speed	10	20	30	40	50
D2	Distance	5	20	45	80	125
D3	Acceleration					

Table 3.3.3(b) the result of (a) when input is 10

Acceleration input value = 20

S	0	R1	R2	R3	R4	R5
0	Time	1	2	3	4	5
D1	Speed	20	40	60	80	100
D2	Distance	10	40	90	160	250
D3	Acceleration					

Table 3.3.3(c) the result of (a) when input is 20

Acceleration input value = 30

S	0	R1	R2	R3	R4	R5
0	Time	1	2	3	4	5
D1	Speed	30	60	90	120	150
D2	Distance	15	60	135	240	375
D3	Acceleration					

Table 3.3.3(d) the result of (a) when input is 30

In conventional spreadsheets, in which global variables are not allowed (hence running time input is not allowed either), users have to simulate input by modifying the constant values of some cells as inputs which other cells refer to.

Besides global variables, user-defined functions have proved very useful, even necessary in a sense, in programming. But in most of the conventional spreadsheets, the functions are all built-in; users cannot define their own functions. Although some conventional spreadsheets such us Open

Access (Software Products International)[6] allow users to define functions when they enter formulas, those functions are usually very simple ones in which only basic arithmetic operations can be used. In the intensional spreadsheet, users can define various functions themselves as in high level language programs. All operators and data objects used in the definitions of global variables can also be used in the definition of user-defined functions. On the other hand, user-defined functions can be used in the definitions of global variables as well as in the definitions of cells of the spreadsheet variable.

In most cases, the facilities for user-defined functions are provided to spreadsheet developers instead of end-users. The developers can define various functions to meet special demands from various users. For example, the following spreadsheet program shows how to define a sum function to sum the values of n cells in a row or a column.

```
rsumn(n) = if n <= 0 then 0 else L(n) + rsum(n-1) fi;
csumn(n) = if n <= 0 then 0 else U(n) + csum(n-1) fi;
```

S	0	R1	R2	R3	R4	R5
0	0	0.1	0.2	0.3	0.4	rsumn(4)
D1	1	1.1	1.2	1.3	1.4	rsumn(4)
D2	2	2.1	2.2	2.3	2.4	rsumn(4)
D3	3	3.1	3.2	3.3	3.4	rsumn(4)
D4	csumn(4)	csumn(4)	csumn(4)	csumn(4)	csumn(4)	csumn(4)+rsumn(4)

Table 3.3.4(a) a spreadsheet program with sum functions

S	0	R1	R2	R3	R4	R5
0	0	0.1	0.2	0.3	0.4	1.0
D1	1	1.1	1.2	1.3	1.4	2.0
D2	2	2.1	2.2	2.3	2.4	3.0
D3	3	3.1	3.2	3.3	3.4	4.0
D4	6	6.4	6.8	7.2	7.6	43.2

Table 3.3.4(b) the evaluation result of (a)

The function rsumn(n) (csumn(n)) returns the sum of the values of the spreadsheet variable at the n cells to the left of (above) the current cell. If the sum functions are always used to sum the values of the cells in a row (a column) from the cell at column 0 (row 0) to the left (above) cell of the cell using the sum function, the functions rsum and csum are much simpler. They may be defined as

rsum = 0 hsby rsum + S ;

csum = csum + S vpby 0 ;

where the functions work in the fourth quadrant.

3.4. Time Operation

The spreadsheet we have described so far improves conventional spreadsheets by using spatial intensional operators and by providing more programming facilities. The intensional time operations of the new spreadsheet mark a significant change from the conventional ones. In this case the spreadsheet becomes three-dimensional instead of two-dimensional. The addition of the time dimension is especially useful when we consider an object varying not only in space but also in time. In other words, if conventional spreadsheets can do what a user can do on a piece of paper, then the intensional spreadsheet can do whatever the user can do on many, even infinite, pieces of paper which have some describable (time) relations among them.

In principle, the spreadsheet represented by the spreadsheet variable consists of an infinite number of planes which themselves are also infinite. At any given time the screen can show only a window of a particular plane of the spreadsheet variable. In other words, a plane is actually a **field**, which is a matrix including all values of the spreadsheet variable at all spatial points and a particular time point. At each time only one of the fields of the spreadsheet variable can be the "current" field, a part of which can be shown on the screen, depending on the "current" time decided by the user. Therefore, on one hand, the points on a plane of the spreadsheet may have different values depending on their spatial positions; on the other hand, the planes of the spreadsheet may have different fields depending on the times they are associated with. An analogy is the world's weather, which may be different in different places and changes every day, where the spatial points are the places and the plane is the world.

The time changing facility of the spreadsheet updates the screen when the current time point changes, so that the current screen always displays the values of the cells at the current time point. The spreadsheet provides users with the time changing command. With it, a user can set any time point he/she wants and see what is on the spreadsheet at that time point. This is like reading a book where a page of the book corresponds to a plane at a given time point. The following is an example to show an ordinary use of the time operation, in which the spreadsheet is used to record the item-price list for a store. In that store, the prices of all items are increased by 5% every month. To solve this problem, we do not need to change the prices every month; instead we just need to define every cell representing the price of an item as **initial price fby S * 1.05**. Then for every month the spreadsheet can automatically show the new prices (**fby** is the intensional operator for time dimension whose operational meaning was described in the previous chapter).

S	0	R1	R2	R3
0		'orange'	'banana'	'apple'
D1	'retail'	0.6 fby S*1.05	0.7 fby S*1.05	0.8 fby S*1.05
D2	'wholesale'	up S*0.9	up S*0.9	up S*0.9

Table 3.4.1(a) a spreadsheet for updating prices

January (time=0)

S	0	R1	R2	R3
0		orange	banana	apple
D1	retail	0.6	0.7	0.8
D2	wholesale	0.54	0.63	0.72

Table 3.4.1(b) the evaluation result of (a) at time 0

July (time=6)

S	0	R1	R2	R3
0		orange	banana	apple
D1	retail	0.80	0.94	1.07
D2	wholesale	0.72	0.84	0.96

Table 3.4.1(c) the evaluation result of (a) at time 6

December (time=11)

S	0	R1	R2	R3
0		orange	banana	apple
D1	retail	1.02	1.20	1.37
D2	wholesale	0.92	1.08	1.39

Table 3.4.1(d) the evaluation result of (a) at time 11

The time feature is expected to help solve some scientific problems, as many quantities in the natural world vary in space as well as time, and many mathematical problems abstracted from physics and other scientific areas are solved using iterative techniques. In the following we give an example to show how the spreadsheet is used to study the general mathematical problem of extracting square roots, which is solved by using an iterative technique. We will give another practical scientific application of the spreadsheet in in Chapter 5.

In Table 3.4.2, the spreadsheet shows the square roots of the numbers from 2 to 10 in each iteration step. Every cell in the spreadsheet is defined in the same way: at time 0, its value is the number for which we want to get its square root; at time 1, its value is the initial value for the iterations; then from time on, its value is the value produced by successive iterations. The spreadsheet program can be used to study the different convergent rates for different numbers when using the same iteration method. Alternatively, when we use different iteration approaches for the same number, we can compare the rates of convergence of the approaches by defining a spreadsheet program similar to the example.

```
n = (S + 3) vpby (2 hsby n + 1) ;
initial = S / 2 ;
approx(z,x) = (x + z/x) / 2 ;
root = n fby initial fby approx(first S, next S) ;
```

S	0	R1	R2
0	root	root	root
D1	root	root	root
D2	root	root	root

Table 3.4.2(a) a spreadsheet for square roots

original numbers

S	0	R1	R2
0	2	3	4
D1	5	6	7
D2	8	9	10

Table 3.4.2(b) the evaluation result of (a) at time 0

initial values

S	0	R1	R2
0	1	1.5	2
D1	2.5	3	3.5
D2	4	4.5	5

Table 3.4.2(c) the evaluation result of (a) at time 1

the first iteration step

S	0	R1	R2
0	1.5	1.75	2
D1	2.25	2.5	2.75
D2	3	3.25	3.5

Table 3.4.2(d) the evaluation result of (a) at time 2

the fourth iteration step

S	0	R1	R2
0	1.41	1.73	2
D1	2.23	2.45	2.65
D2	2.83	3	3.16

Table 3.4.2(e) the evaluation result of (a) at time 5

The user-defined function **approx(z,x)** calculates the new approximations at each iteration step and returns them to the cells through the global variable **root**.

Finally, we give one more example to show the 3-dimensional feature of the intensional spreadsheet. We use the intensional spreadsheet to play the game of life. The game of life, invented by John Horton Conway of Cambridge University may briefly be described as follows. Each point in

a grid may contain an organism. Every gridpoint is adjacent to eight other gridpoints. We use occ(k) to represent the number of the adjacent gridpoints which are occupied by the organisms. By using two simple rules we can get a new organism configuration from the previous one:

(1) if $2 \leq occ(k) \leq 3$, then the organism in the grid k can survive, otherwise it will die.

(2) if occ(k)=3, then a new organism will be born at any point k which is currently empty.

Programming the intensional spreadsheet to solve this problem is very simple, because many features of our spreadsheet are well suited to the problem. Table 3.4.3(a) shows a spreadsheet program for the game of life which is played in a 5×5 array and initially(time 0) has seven organisms. Table 3.4.3(b)-(d) show the organism configurations at time 0,1 and 5.

```
t(x) = if x eq '*' then 1 else 0 fi;
occ = t(L1) + t(R1) + t(U1) + t(D1) +
      t(U1R1) + t(U1L1) + t(D1R1) + t(D1L1) ;
survive = if occ eq 2 or occ eq 3 then S else '' fi ;
birth = if occ eq 3 then '*' else S fi ;
life = if S eq '*' then survive else birth fi ;
```

S	0	R1	R2	R3	R4	R5	R6
0	''	''	''	''	''	''	''
D1	''	'' fby life	'*' fby life	'' fby life	'*' fby life	'' fby life	''
D2	''	'' fby life	'*' fby life	'' fby life	'*' fby life	'' fby life	''
D3	''	'' fby life	'*' fby life	'*' fby life	'*' fby life	'' fby life	''
D4	''	'' fby life	'' fby life	'' fby life	'' fby life	'' fby life	''
D5	''	'' fby life	'' fby life	'' fby life	'' fby life	'' fby life	''
D6	''	''	''	''	''	''	''

Table 3.4.3(a) the spreadsheet program for the game of life

S	0	R1	R2	R3	R4	R5R6	
0							
D1			*		*		
D2			*		*		
D3			*	*	*		
D4							
D5							
D6							

Table 3.4.3(b) the configuration of organisms at time 0 (initial)

S	0	R1	R2	R3	R4	R5R6	
0							
D1							
D2		*	*		*	*	
D3			*		*		
D4				*			
D5							
D6							

Table 3.4.3(c) the configuration of organisms at time 1

S	0	R1	R2	R3	R4	R5	R6
0							
D1							
D2		*	*		*	*	
D3		*				*	
D4			*	*	*		
D5			*	*	*		
D6							

Table 3.4.3(d) the configuration of organisms at time 5

3.5. Advanced Intensional Operators

There are some other intensional operators, called *advanced* intensional operators, which can also be used in the intensional spreadsheet programs. Just like the primitive intensional operators we described before, each of these advanced operators works in one of three dimensions and has two analogs which work in the other two dimensions. The advanced intensional operators for the

horizontal dimension in space are **haca**(for "horizontally as close as"), **hbaca**(for "horizontally backward as close as"), **hwherever**(for "horizontally wherever") and **hupon**(for "horizontally upon"). The advanced intensional operators for vertical dimension in space are **vaca**(for "vertically as close as"), **vbaca**(for "vertically as close as"), **vwherever**(for "vertically wherever") and **vupon**(for "vertically upon"). The advanced intensional operators in time are **asa**(for "as soon as"), **basa**(for "backward as soon as"), **whenever** and **upon**, which are the original operators of the Lucid language[12].

These operators are called *advanced* operators because they can perform more complicated operations than other primitive operators. In fact, all the advanced operators can be defined as functions in terms of the primitive operators; these definitions can be found in Appendix A. Through these definitions, the formal semantics of the advanced operators can be derived from the formal semantics of the primitive operators. In the following, we will informally describe the operational meanings of some of these operators. The operational meanings of others are easy to understand from the descriptions.

The binary operator **haca**, which has a data argument and a boolean argument, takes only one desired value from the data argument according to the values (predicates) of the boolean argument at the points in each horizontal dimension. Therefore, the operation **D haca P** always produces a new variable whose values in each horizontal dimension are constant. When evaluated at a given spatial point (x,y), where x,y are its horizontal and vertical coordinates respectively, the expression defined by the above operation returns the value of D at the spatial point (1) whose vertical coordinate is y and (2) at which P has a true value and it is the first true value in the non-negative horizontal dimension from 0 and forward. In other words, the operator **haca** scans the current row from coordinate 0 forward until a true value of P is obtained at some point; then the value of D at this point is returned. For example, the expression

```
d haca p
  where
    d = 0 hsby d+1;
    p = false hsby false hsby true;
```

```
end
```

has the constant value 2, because at horizontal coordinate 2, p has the first true value where the corresponding value of d is 2. The behavior of the operator **hbaca** is almost the same as that of **haca**. The only difference between them is that **hbcac** scans P backwards from the horizontal coordinate 0.

The binary operator **hwherever**, which also has a data argument and a boolean argument, removes those values from the horizontal dimension (i.e. a row) of its data argument for which the corresponding values of its boolean argument are false. Therefore, the operation **D hwherever P** produces a new variable whose values in each horizontal dimension (row) are a subset of the values of D in that dimension. When evaluated at a given spatial point (x,y), the above expression returns the value of D at the spatial point (1) whose vertical coordinate is y and (2) at which the xth true value of P is found from horizontal coordinate 0 and forward, if $x \geq 0$; or the $|x|^{th}$ true value of P is found from horizontal coordinate -1 and backward, if $x<0$. For example, the expression

```
d hwherever p
  where
    d = 0 hsby d+1;
    p = true hsby not p;
  end
```

has the value equal to two times of the horizontal coordinate at which the expression is evaluated, because p has false values at every odd horizontal coordinate which results in all odd non-negative values of d being filtered out.

The binary operator **hupon**, which also has a data argument and a boolean argument, duplicates some values in the data argument for each horizontal dimension. That is, if at a spatial point the value (predicate) of the boolean argument is false, then the value of the result variable at this point is some duplicate value of the data argument. The operation **D hupon P** produces a new variable whose values in each horizontal dimension are a "superset" (including duplicate elements) of the set of values of D in that dimension. When being evaluated at a given spatial point (x,y), the above

operation returns the value of D at either point (x,y) if x is 0 or the point (1) whose vertical coordinate is y and (2) which is the first point scanned from x-1 backward if x>0 or from x+1 forward if x<0 at which P has a true value unless the point (0,y) is reached. For example, the expression

```
d hupon p
  where
    d = 0 hsby d+1;
    p = false hsby not p;
  end
```

has the value equal to the horizontal coordinates divided by 2, because p has false values at every even horizontal coordinate which results in the value of d at every point being duplicated to the right point, where only non-negative coordinates are considered.

One of the most interesting features of the advanced operators **h(v)wherever** and **h(v)upon** is that these operators can be used to "shrink" or "extend" the spreadsheet itself, which is usually done by some user-interface commands and never by the spreadsheet program in conventional spreadsheets. The following examples show how the spreadsheet is shrunk and extended by using the operators **hwherever, vwherever, hupon** and **vupon.** The spreadsheet program in Table 3.5.1 constructs various sub-matrices or extended sub-matrices of the matrix we constructed in Table 3.2.3 and Table 3.3.2.

```
x = 0 hsby 1 + x ;
y = y + 1 vpby 0 ;
Ph = true hsby not Ph ;
Pv = not Pv vpby true
SM = a fby b fby c fby d
   where
     a = S hwherever Ph ;
     b = S vwherever Pv ;
     c = S hupon right Ph ;
     d = S vupon up Pv ;
   end ;
A = L(x-y) + D(x-y) fby SM ;
```

S	0	R1	R2	R3	R4
0	1	A	A	A	A
D1	A	2	A	A	A
D2	A	A	3	A	A
D3	A	A	B	4	A
D4	A	A	A	A	5

Table 3.5.1(a) a spreadsheet program using **wherever** and **upon**

the original matrix

S	0	R1	R2	R3	R4
0	1	3	4	5	6
D1	3	2	5	6	7
D2	4	5	3	7	8
D3	5	6	7	4	9
D4	6	7	8	9	5

Table 3.5.1(b) the evaluation result of (a) at the initial step

sub-matrix 1

S	0	R1	R2	R3	R4
0	1	4	6		
D1	3	5	7		
D2	4	3	8		
D3	5	7	9		
D4	6	8	5		

Table 3.5.1(c) the evaluation result of (a) at step 1

sub-matrix 2

S	0	R1	R2	R3	R4
0	1	4	6		
D1	4	3	8		
D2	6	8	5		
D3					
D4					

Table 3.5.1(d) the evaluation result of (a) at step 2

extended sub-matrix 3

S	0	R1	R2	R3	R4	R6
0	1	1	4	4	6	6
D1	4	4	3	3	8	8
D2	6	6	8	8	5	5
D3						
D4						

Table 3.5.1(e) the evaluation result of (a) at step 3

extended sub-matrix 4

S	0	R1	R2	R3	R4	R6
0	1	1	4	4	6	6
D1	1	1	4	4	6	6
D2	4	4	3	3	8	8
D3	4	4	3	3	8	8
D4	6	6	8	8	5	5
D5	6	6	8	8	5	5

Table 3.5.1(f) the evaluation result of (a) at step 4

In the above spreadsheet program, the variable **Ph** varies along the horizontal dimension, and it has value true at the points with even horizontal coordinates and has value false at the other points. Similarly, the variable **Pv** varies along the vertical dimension; it has value true at the points with even vertical coordinates and has value false at the other points; the variable **SM** describes the steps of shrinking and extending the spreadsheet to make the sub-matrices.

During evaluation, at the first step (sub-matrix 1), the columns with the odd numbers in the original matrix are removed by the operation **S hwherever Ph** because the points with odd horizontal

coordinates in **Ph** are false; thus the spreadsheet is shrunk in the horizontal dimension. Similarly at the second step (sub-matrix 2), the rows with the odd numbers in the sub-matrix 1 are removed by the operation **S vwherever Pv** because the points with odd vertical coordinates in **Pv** are false; thus the spreadsheet is shrunk in the vertical dimension. At step 3 (extended sub-matrix 3), new columns are added between every two columns in the sub-matrix 2, and the cells in the new columns have the same values as their neighbors to the immediate left. This is done by the operation **S hupon right Ph** because **right Ph** has value true at the points with odd horizontal coordinates and value false at the other points; thus the spreadsheet is extended in the horizontal dimension. In a similar way, at the last step (extended sub-matrix 4), new rows are added between every two rows of the extended sub-matrix 3, and the cells in the new rows have the same values as the cell immediately below them. This is done by the operation **S vupon up Pv** because **up Pv** has value true at the points with odd vertical coordinates and value false at the other points; thus the spreadsheet is extended in the vertical dimension.

Given different patterns to **Ph** and **Pv**, we can shrink or extend the spreadsheet in various ways. But in order to implement these kinds of behaviors in the spreadsheet, we have to modify the spreadsheet a little, that is, we must allow undefined cells to be operated upon. To solve this problem, we can consider that any cells which are not defined before the evaluation have a **null** value; thus they can appear in operations without causing an evaluation error. But the other side of the problem is that sometimes the spreadsheet must, instead of keeping silent, remind the user that there are undefined cells if they are referred to by other cells. A possible tradeoff is that when the operation on an undefined cell is only copying its value, then the value can be **null**; otherwise it must be reported to the user.

Sometimes the intensional operators discussed in this section may cause problems during evaluation. One of the problems is the operator **asa** when it is used in spreadsheet programs[14]. This is because the computation of **asa** , under some cases, terminates its evaluation at different time

points for the different cells, or in other words the termination condition of the operation **S asa P** is "distributed". For example, if we write a spreadsheet program which is a solution of Laplace's equation using the relaxation method, the definitions for the non-electrode cells should be defined as follows:

```
0 fby avg(S) asa P
where
  P = abs(next S -S) <= 0.0001  ;
end ;
```

and the function avg(x) may be defined as:

```
avg(x) = (left x + right x + up x + down x) / 4 ;
```

By evaluating this program, theoretically, we should obtain the desired iteration results at all non-electrode points immediately after the initialization; but things are not so simple. If the relaxation settles down quickly at one point in the spreadsheet variable S, there need be no further calculation for that point, even though calculation can be continuing for other points. The problem occurs when P is defined as a purely local condition. From a local view, it may appear that a particular point has settled down when a more global view might show that a disturbance in potential, from a distant electrode, is approaching the point. P must postpone making its local decision at any point until disturbances from all the electrodes have begun to be felt at that point. This requires some initial global analysis to determine what is a suitable "waiting time" at each point. This is a question that does not seem to have been addressed at all in the work on relaxation methods. If it is impossible to find a suitable local termination condition, P may have to be defined globally such as

```
P = abs(sumavg(next S) - sumavg(S)) <= 0.0001
```

where the function sumavg(x) returns the average of all the non-electrode cells at any given time. Now P is a globally defined variable, because in order to calculate its value at a time point, all the values of the defined cells in the program at the previous time must be known first.

CHAPTER 4

AN IMPLEMENTATION OF THE INTENSIONAL SPREADSHEET

The intensional spreadsheet described in the previous chapter has been implemented by the author. The implementation consists of two relatively independent parts: the user interface and the spreadsheet interpreter. In this chapter we will concentrate on the spreadsheet interpreter.

The interface between the user interface and the spreadsheet interpreter is made as simple as possible to keep them independent. The user interface passes the following information to the interpreter: (1) requests to enter or delete a definition of a cell of the spreadsheet variable, a global variable or a user-defined function, and to evaluate a cell of the spreadsheet variable; (2) values input when they are needed. The spreadsheet interpreter passes the following messages to the user interface: (1) returned values after evaluation or re-evaluation; (2) requests for input; (3) error messages.

4.1. Principles of the Spreadsheet Interpreter

4.1.1. Eductive Computation

Eductive computation or "eduction" is tagged demand-driven computation[15].

In demand-driven computation, an operation is performed only if all its operands, which are data items, are available and there is a demand for the result of the operation. If there is a demand for an operation which depends upon an operand whose value is not available, then a demand for that value is made to the operation that produces it. When, as a result of such demands, there is a value available for each of the operands, the operation is performed (consuming the operands), and its result is sent out to the operands which need this result. In this model, in fact, there is a two-way traffic in the computation: data flows in one direction from the producers to the consumers and demands are

sent from the consumers to the producers in the other direction.

In the tagged-demand driven computation model, the data and demand buffers are sets instead of queues. However, some discipline must be imposed on the order in which an operation takes the data items from its operands or accepts demands for response. This is achieved by associating the data items and demands with tags so that the operation looks for data items and responds to demands with appropriate tags.

The main advantages of the tagged demand-driven model are the potential for eliminating a vast amount of computation by evaluating only what is necessary for computing a desired result, and the better handling of infinite data structures. In this model, it is much easier to deal with non-strict operations (a strict operation is one that requires values for all its arguments in order to produce a result). For example, with the demand driven evaluation, the typical non-strict operation "if-then-else" is quite efficient, because only the operands that are needed are evaluated, or in other words only the branch that is needed is demanded.

One of the most surprising properties of eduction is that it does not require any static memory. The program is never changed during computation. As a result, values that have been computed but not saved can be recomputed from scratch if they are needed again. Unfortunately, a memoryless educer is hopelessly inefficient. Endless recomputations can be avoided in implementations by storing computed values in a large warehouse. Every time a new demand is generated, the implementation first checks to see if the value is already available in the warehouse. Values in the warehouse are labeled by the name of the variable whose value is stored and the tag identifying the particular value in question. Data items in the warehouse are accessed by their labels.

4.1.2. An Eductive Interpreter for the Intensional Spreadsheet

The spreadsheet interpreter, or the spreadsheet "educer", which is written in C, is eductive, that is, it is based on the tagged demand-driven scheme: a demand for the value of a cell in the spreadsheet, or the value of the spreadsheet variable at a given spatial and time point, generates demands for values of other cells and/or variables, internal to the spreadsheet program, at the same or different spatial and time points. The demands, and the values returned in response, are tagged by a time coordinate, two spatial coordinates, and a "place" coordinate corresponding to some particular function call history (place coordinates are discussed later). Once computed, values are stored in an associative memory, called the value-warehouse, in case they are demanded again.

The principle behind the eductive approach of the spreadsheet interpreter may be briefly described as follows. The entire computation is driven by the demand of the user who wants to evaluate a cell in the spreadsheet after he/she has entered the spreadsheet program. In computing the value of the spreadsheet variable at some spatial and time point, the educer is led to compute various variables at possibly different spatial points and times. When the value of a variable **V** at horizontal coordinate $\mathbf{S_h}$, vertical coordinate $\mathbf{S_v}$ and time **t** is required, the educer consults the relevant definition of the variable. It uses the formal semantics of the operations appearing in the definition and computes the required value from the values of the operands at possible other coordinates.

The spreadsheet interpreter avoids recomputations by storing computed values in a large **value-warehouse**, which is an associative memory. Each time a new demand is generated, the interpreter first checks the value-warehouse to see if the value is already available. Values in the value-warehouse are labeled by the variable whose value is stored and the tag identifying the particular value in question. These values are efficiently accessed by their labels using hashing.

Apart from the value-warehouse, the interpreter also contains a **definition-warehouse**, which is another associative memory, to store the definitions of variables and user-defined functions. Each time a new demand for the value of a variable is generated, if the educer cannot find the needed

value of the variable in the value-warehouse, it fetches the definition of the variable and evaluates the variable by interpreting its definition; then, the evaluated value is stored into the value-warehouse for later use. Definitions in the definition-warehouse are labeled differently depending on whether they belong to the spreadsheet variable, global variables or user-defined functions. The definitions of global variables and user-defined functions are labeled only by their names, while the definitions of the spreadsheet variable, since it is defined individually, are labeled by the name of the spreadsheet variable and the tags identifying the particular spatial point(cell). In addition, the definitions of actual parameters in user-defined function calls are labeled by the names of the corresponding formal parameters as well as the place number.

Eduction is a very simple technique when the "tag" involved is composed of time and space coordinates, i.e. essentially small integers for indicating a particular spatial and time point. But complications arise when the method is extended to handle some basic features of the intensional spreadsheet such as user-defined functions. For example, the value of a variable like **M** in

f(x) = M where M = x hsby M + right x; end

depends on more than just time and space; it depends on the particular "call" or "invocation" of the function **f**. Operationally, the definition of **f** is simply a template, and a use of **f** is a separate copy with its own separate internal storage. As a result, at any given time t and spatial position (s_v, s_h), there will be many different individual values of the variable **M**. The spreadsheet interpreter incorporates a simple solution to the problem. It assigns unique "coordinates" to the calls of user-defined functions in any particular spreadsheet program. A place parameter is set for each user-defined function. When a user-defined function call is found in the definition of a variable or another function during compilation, the place parameter corresponding to the called function is inserted into the proper object codes for that function call, and the place parameter is increased by 1. As a result, during evaluation, every function call has a unique place number for the called function. In case a function is defined nestedly or recursively, all the nested or recursive calls to the function share one place

number, but the values evaluated in every call are given different place numbers when they are stored in the value-warehouse. Therefore, the time, space and place parameters together are enough to determine individual values of variables like **M**.

4.1.3. The Implementation of the Spreadsheet Interpreter

The spreadsheet interpreter is built on a software tool called the "popshop"[16]. The popshop is an attempt to use sound software engineering principles to facilitate the production of prototype implementations for experimental nonprocedural languages. The popshop is intended solely for languages which are based on the data objects, operations and syntax of pop-2 (a lisp-like AI language[17]). The data types of pop-2 are numbers(integers and reals), strings(of ASCII characters), words(identifiers and sequence of "signs") and lists. The data types of pop-2 are also the data types used in intensional spreadsheet programs. At present the actual programming of the popshop is done in C. Figure 4.1.1 is the scheme of the spreadsheet interpreter. In the following paragraphs we will describe each of the main parts of the interpreter briefly.

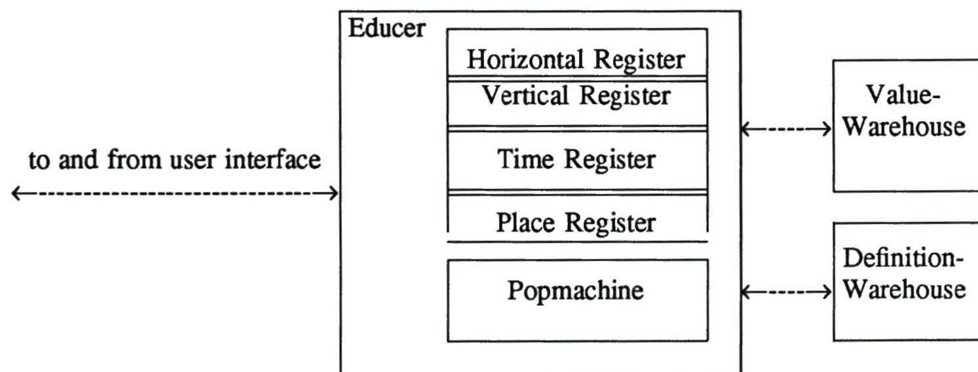


Figure 4.1.1 the structure of the spreadsheet interpreter

The basis of the implementation of the spreadsheet interpreter is the "popmachine"[16], which works like the "cpu" in conventional computers. The popmachine is centered around the "value stack" which is a stack of pop-2 data items. The basic operations are performed on the top elements of the stack. The operations include ones to read from and write onto the stack, to pop the stack, to swap

the top elements, to cycle the top elements and so on. The basic popmachine has no storage other than its stack. There are therefore no fetch or store operations in the core instruction set. The machine has all the data operations of the pop-2 algebra. The data operations act on the appropriate number of items on the top of the stack. The assembly/machine language of the popmachine is called "popcode". A popcode program is a pop-2 data object. It is a list of items each of which is either an operand to be placed on the stack, or an operation to be executed. Items which are lists, strings or numbers are taken to be operands; items which are words are taken to be operations. The popmachine is a RISC (Reduced Instruction Set Computer) design, i.e. all the popmachine operations take their arguments from the stack, and there are no address fields in popcode. For example, for a simple expression **x + y**, the compiled popcode is ["x eval "y eval +], where the operation "x and "y push the identifiers, x and y, onto the stack, respectively; the operation *eval* evaluates the variable whose name is at the top of the stack and leaves the result on the top of the stack; and the operation + always replaces the two data items on the top of the stack by their sum.

The spreadsheet interpreter is an expansion of the popmachine, which is designed to run pre-processed spreadsheet programs or extended popcode programs compiled from their source programs. The interpreter is constructed by adding some "boards" to the popmachine. These boards provide the time register, the horizontal space register, the vertical space register and the place register as well as the value- and definition-warehouses.

The **time register** holds a single time coordinate which is a natural number. It enables commands to increase and decrease the time parameter, to set it to any number and to save and restore its contents. During evaluation, the value of the time register is always equal to the "current" time.

The **horizontal and vertical space registers** hold single horizontal and vertical space coordinates, respectively. They enable commands to increase and decrease the horizontal and vertical dimension parameters, to set them to any integers and to save or restore their contents. During evaluation, the contents of the horizontal and vertical space registers are always equal to the "current"

horizontal and vertical coordinates, respectively.

The **place register** holds a single place coordinate which is a small natural number in most cases, or a finite sequence of numbers when function nesting takes place. The register enables various commands to set, save and restore it. The content of the place register is always equal to the place number of the current function call appearing in the spreadsheet program.

The **value-warehouse** stores arbitrary data items tagged by the names of the variables they belong to, by spatial coordinates, and by time and place. It enables commands for storing and fetching items in it. During accessing, the spatial coordinates, time and place parameters which constitute the tag are the current contents of the spatial, time and place registers. Those coordinates and parameters are sent to the value-warehouse directly rather than through the value stack.

The **definition-warehouse** permits definitions of variables to be added, modified and deleted during "run" time. It associates object definitions in popcode with variables and functions. The popcode associated with a variable or function has the effect, when executed, of placing on the stack the value of the variable or function at the spatial coordinates, time and place currently in the appropriate registers. The execution of the popcode may involve changing the contents of the registers, which should be restored upon completion.

4.2. Entering a Definition

A user may enter one of three sorts of definitions: the formula of a cell (the definition of the spreadsheet variable at a particular spatial point), the definition of a global variable or the definition of a user-defined function. Regardless of the kinds of definitions they are, they have the same syntax as the definitions in the Plane Lucid grammar. The user interface passes the definition as a string of characters to the spreadsheet interpreter associated with the name of the defined variable or function and the spatial position if the defined variable is the spreadsheet variable. After the interpreter receives the definition, it takes two steps to complete the task: compiling the source definition and

storing the compiled object definition(s).

4.2.1. Compilation of a Definition

Compiling a definition has three phases: generating a parse tree, renaming intermediate code and generating popcode. The spreadsheet interpreter, at any one time, can compile only one definition and cannot do global analysis of the whole program, because the definitions are entered separately. This sometimes causes difficulties, especially when user-defined function calls are involved in the source definitions.

The first phase, **generating a parse tree**, is common to most compilers. In this phase, the parser produces a parse tree from the source definition. The parse tree is represented as a list whose members may include sublists representing subtrees. For example, if a variable, say root, is defined as

initial fby approx(n,S) where n = 2 hsby n+1; end

the corresponding parse tree has the form

```
[where
  [fby [var initial] [call approx [var n] [var S]]]
  [= n [hsby [" 2] [+ [" 1] [var n]]]
].
```

The second phase, **renaming intermediate code**, does the following work on the parse tree produced by the previous phase.

(1) If the entered definition is a user-defined function or involves user-defined function calls, it gives the user-defined function and their formal parameters new names to avoid name conflicts, because eventually their object definitions will be stored in the definition-warehouse in the same way as other variables.

(2) If the entered definition includes a *where* clause, it flattens the where clause, i.e. it converts local variables of any nested where clause to globals which have unique names in the

spreadsheet program.

The rules to rename a user-defined function, say **F**, and its formal parameters: are

(1) adding an under score '_' before the original function name to form a new unique name **_F**,

(2) changing the name of each formal parameter of the function to **_F_i** where **i** is the position of the parameter in the argument list of **F**.

The rules to rename a local variable, say **x**, in a definition are:

(1) if the definition is for a global variable, say G, then the new name of the local variable is **G_x**;

(2) if the definition is for the spreadsheet variable (**S**) at a particular spatial point such as U1R1, then the new name of the local variable is **S_U1R1_x**;

(3) if the definition is for a user-defined function, say **F**, then the new name of the local variable is **_F_x.**

It is easy to see that all new names produced by the renaming rules in this phase are unique in the spreadsheet program. For example, after the renaming phase the definition for global variable root now becomes

```
[where
  [fby [var initial] [call _approx [var root_n] [var S]]]
  [= root_n [hsby [" 2] [+ [" 1] [var root_n]]]
]
```

where function *approx* is renamed as *_approx* and the local variable *n* is renamed as *root_n.*

The last phase, **generating popcode**, is more complicated. It not only translates the source definition from the renamed parse tree into the appropriate object definition(s) in popcode, which is relatively easy because for most of operators in the source definition there are similar instructions in popcode, but also performs some other tasks for the storage step and later evaluations.

One of these tasks is that if the source definition involves a *where* clause, apart from producing an object definition for the subject of the where clause and making it the object definition of the variable or function the source definition belongs to, it has to produce an object definition for each local variable that has become a "global" variable after the renaming phase.

The other task of this phase is to produce a proper place number for a particular function call. It keeps a function-call count (beginning at 1) for every user-defined function during the compilation of the spreadsheet program. Whenever a user-defined function call appears in the definition, the current function-call count for that user-defined function is added into the code of the object definition as the referenced place number and then the count is increased by 1. Eventually, for every particular call to a user-defined function, a unique place number is assigned. Meanwhile, the object definition for every actual parameter in a user-defined function call is produced , and is stored into the definition-warehouse with the same name as its corresponding formal parameter's and the place number corresponding to that particular function call.

To sum up, a source definition for a variable or a user-defined function may be translated into one or more object definitions. These object definitions may include the object definition for the original variable or function which the source definition belongs to, the object definitions for the local variables in the where clause and the object definitions for actual parameters of the function calls. For example, the source definition of global variable root is eventually compiled into four object definitions - one for the variable itself, another for the renamed local variable root_n and two for the actual parameters of renamed user-defined function _approx. The following are the object definitions in popcode.

The object definition for the global variable root is

```
[[" initial eval] [saveplace 1 pushplace " _approx eval restoreplace] fby]
```

where the place number for the function call approx(n,S) is 1.

The object definition for renamed local variable root_n is

```
[[2] [1 " root_n eval +] hsby]
```

The object definition for the first actual parameter of function _approx is

```
[saveplace popplace " root_n eval restoreplace]
```

and the object definition for the second actual parameter of function _approx is

```
[saveplace popplace " S eval restoreplace]
```

4.2.2. Storage of Object Definitions

In this step, the object definition(s) produced in the previous step are stored in the definition-warehouse with proper labels. A label for an object definition includes the name of the object it belongs to, which may have one of the following types: a global variable, the spreadsheet variable, a renamed local variable, a renamed user-defined function or a renamed formal parameter. The label also consists of a tag which may include the spatial coordinates to indicate an individual definition for the spreadsheet variable and the place number to define the formal parameter with the definition of its actual parameter in a particular function call.

The strategy for storing the object definitions of various variables and functions can be described as follows:

(1) the object definition of a cell in the spreadsheet is stored into the definition-warehouse with the label consisting of the name of the spreadsheet variable and the cell's horizontal and vertical coordinates;

(2) the object definition of a global variable is stored into the definition-warehouse with the label consisting of only the name of the variable;

(3) the object definition of a user-defined function or a local variable is stored into the definition-warehouse with the label consisting of its new name;

(4) the object definition of an actual parameter is stored into the definition-warehouse with the label consisting of the name of the corresponding renamed formal parameter and the place number of the particular function call the actual parameter belongs to.

For example, the object definitions shown in the previous example will be stored into the definition-warehouse separately. That is, the object definition for the global variable root is stored with the label containing the variable's name *root*; the object definition for the local variable n is stored with the label containing the renamed variable's name *root_n*; and the object definitions for the actual parameters of the function call approx(n,S) are stored separately with the labels containing the renamed formal parameter's name and the place number 1.

4.3. Evaluation of the Spreadsheet

The evaluation of the spreadsheet variable is pointwise, i.e. every time only one value of the spreadsheet variable at a given spatial and time point is evaluated. It is impossible to evaluate the spreadsheet variable at all spatial and time points because the spreadsheet variable spans an infinite plane and is infinite in time. The interpreter can accept an evaluation demand from the user at a particular spatial point(cell) and a particular time. The evaluation is eductive.

The evaluation begins when the educer injects a demand for the spreadsheet variable with the tag consisting of the given vertical and horizontal spatial coordinates as well as time. During the evaluation, whenever a value of a variable is demanded, the educer first checks if the demanded value has been produced by accessing the value-warehouse with the label consisting of the variable's name, the current spatial coordinates in the spatial registers, the current time in the time register and the current place in the place register. If the value was produced, then it is returned. Otherwise, the definition of the variable is fetched from the definition-warehouse according to the variable's name, the current spatial coordinates (if it is the spreadsheet variable) and the current place if it is a function parameter. The fetched definition is then executed; the result is returned and also stored in the

value-warehouse, tagged with the variable name as well as the current spatial coordinates, time and place.

If the definition-warehouse does not contain the definition of the demanded variable, it means that the demanded variable is either an input variable or an undefined cell or function. For the former, the interpreter sends a request to the user interface asking the user to input the value for that variable. The input value is stored in the value-warehouse tagged with the name of the input variable as well as the spatial coordinates, time and place, which are the same as when the input variable is demanded. After the needed input value is entered, the evaluation is continued. If the failure of the definition fetch results from some undefined cell or function, an error message is returned to the user interface to prompt the user to define that cell or function before re-evaluating.

4.4. Deletion and Modification of a Definition and Re-evaluation

A user can delete or modify the definition of a cell of the spreadsheet variable, a global variable or a user-defined function at any time during programming the spreadsheet.

4.4.1. Deletion

The deletion of a source definition may result in (1) deleting more than one object definition in the definition-warehouse if the source definition contains a *where* clause and/or user-defined function call(s); (2) removing those stored values from the value-warehouse whose owner depends, directly or indirectly, on the deleted definitions, because those values are no longer valid.

Therefore, we need two corresponding steps to delete a definition. To avoid dealing with complicated dependencies among cells, global variables and functions, in the current spreadsheet interpreter the step (2) is performed in a naive way, i.e. we remove all values from the value-warehouse when deleting the definition of a cell, a global variable or a user-defined function, because even though the values which are still valid after the deletion are removed they can be recomputed later.

The algorithm for the deletion is as follows:

Assume X is the cell or global variable or user-defined function whose definition the user wants to delete.

Step 1:

(1) for each renamed local variable defined in X's source definition, remove its object definition from the definition-warehouse;

(2) for each user-defined function call appearing in X's source definition, find its place number; remove its formal parameters' object definition with the place number obtained from the definition-warehouse;

(3) delete X's object definition from the definition-warehouse.

Step 2:

remove all values from the value-warehouse.

After these two steps, there is no trace of the deleted definition. That is, after the deletion, if the deleted definition is for a cell of the spreadsheet variable or a user-defined function, it becomes an undefined cell or function, while if the deleted definition is for a global variable, it becomes an input variable.

4.4.2. Modification

The modification of the definition of a cell, a global variable or a user-defined function can be thought of as two separate steps: deleting the existing definition and entering a new definition. Apart from the problems caused by the deletion, the modification of a definition may result in re-evaluation of the cells in the spreadsheet which depend on the modified cell, global variable or user-defined function.

4.4.3. Re-evaluation

If the definition of a cell, a global variable or a user-defined function is modified, re-evaluation for the cells which directly or indirectly depend on the modified definition has to be performed, because their values displayed on the screen are now invalid and have to be updated.

In the current spreadsheet interpreter, since all values in the value-warehouse are removed during modification, the re-evaluation necessitates re-evaluating all the cells which were defined.

4.5. About Dependencies and Storage Management

It is obvious that if we do not set the dependencies explicitly among the cells of the spreadsheet variable, global variables and user-defined functions in a spreadsheet program, after the user deletes or modifies even a single definition, the evaluation of the spreadsheet program becomes very inefficient. Since it is unknown which cells are affected by the modification, all defined cells have to be re-evaluated. The dependency problem, and the spreadsheet size problem, are the two main efficiency considerations in spreadsheet design[18].

Due to the presence of intensional operators as well as user-defined functions and global variables, the dependency problem in intensional spreadsheets is even more complicated than in conventional spreadsheets. The multi-level and nested applications of the intensional operators can make such complicated references to variables at particular spatial and time points that it is very difficult to determine the dependencies textually. In addition, since a global variable or a user-defined function can be used in any definition, the dependencies among them, therefore, are often hidden, which makes things even more difficult.

Although the author has designed a preliminary algorithm to determine the dependencies in an intensional spreadsheet program, it is incomplete and needs to be implemented and refined, and has been left as a task for future work.

The size of the intensional spreadsheet depends mainly on the size of the value-warehouse because the number of values it stores increases rapidly during evaluation. Some method may be used to restrict the value-warehouse to a reasonable size, because not all values computed by the educer will be used again and any value can be recomputed even if it was thrown away prematurely.

If every value ever computed was stored in the value-warehouse and nothing was ever removed, it would be very time-efficient because nothing is ever recomputed, but it would be incredibly wasteful of space. In the spreadsheet case, of course, the situation is not so bad, because some values will be removed when the user modifies or deletes definitions, but once the spreadsheet program is settled down the same problem occurs. Therefore, it is necessary to remove some values which will not be further used from the value-warehouse before it becomes too large. The problem is how to determine whether a value in the value-warehouse is useful or not.

The author has designed a heuristic to remove some values which have a high probability of not being of further use. But the algorithm needs to be experimented on with the spreadsheet interpreter, and is left as another task for future work.

CHAPTER 5

APPLICATION AND COMPARISON

In this chapter we will first present one more scientific application to show that the intensional spreadsheet is more powerful and of wider applicability than conventional spreadsheets. Then we will compare the intensional spreadsheet with some conventional spreadsheets to show that most features in the conventional spreadsheets can also be implemented in the intensional spreadsheet. In other words, we want to show that the intensional spreadsheet can do not only what the conventional spreadsheets can do but also some things they cannot do.

5.1. A Scientific Application Example

Consider a classical problem in engineering: heat transfer in a solid. Suppose that we have a large thin metal plate which is initially cool (temperature 0). On the plate there is a point that touches a heat source(temperature 100). The heat will gradually diffuse through the plate, with parts nearer the heat source at first warming more quickly than those further away. The problem is to determine the temperature in various points of the plate.

The solution to the problem can be derived from the classical Two-Dimensional Heat-Flow Equation[19]:

$$\frac{\partial T}{\partial t} = a^2\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right)$$

which is a partial differential equation. In the equation, T is the temperature at the spatial point with the horizontal coordinate x and vertical coordinate y and at time t; and a is a constant related to a property of the solid in question. We can solve the partial differential equation by using a simple discrete difference approximation.

The problem involves an intension (temperature) which varies in space(2-dimensional in this case) and in time. Can we arrive at a solution which reflects the intensional point of view in the intensional spreadsheet? To illustrate the problem oriented nature of the intensional spreadsheet let us first derive a spreadsheet program from the above equation. The heat flow equation given earlier can be integrated in time to obtain

$$T = C + \int(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2})\, dt$$

where C is a constant related to a^2.

We can use intensional operators and user-defined functions to help solve the problem. First we need a function which gives us an approximation to the second space derivative. This is not very difficult, if we think intensionally. Suppose we have a quantity **H** varying in the horizontal dimension of the space, $\mathbf{h}_1$ is the value at a particular spatial point, and $\mathbf{h}_0$ and $\mathbf{h}_2$ are the values of the points on its left and right side, respectively. The ratio

$$\frac{h_1 - h_0}{dx}$$

is the derivative one-half step to the left, and

$$\frac{h_2 - h_1}{dx}$$

is the derivative one-half step to the right. It follows easily that the following is approximately the second derivative at $\mathbf{h}_1$:

$$\frac{\frac{h_2 - h_1}{dx} - \frac{h_1 - h_0}{dx}}{dx}$$

This can be reduced to

$$\frac{h_2 - 2h_1 + h_0}{dx^2}$$

From this we are led to the definition of the user-defined function

```
DX2(h) = (left h - 2*h + right h) / dx**2 ;
```

Similarly, if a quantity **V** is varying in vertical dimension, and $\mathbf{v}_1$ is the value at a particular spatial point, and $\mathbf{v}_0$ and $\mathbf{v}_2$ are the values of the points above and below it, respectively, the following approximates the second derivative in vertical dimension

$$\frac{v_2 - 2v_1 + v_0}{dy^2}$$

From this we are led to the definition of the user-defined function

DY2(v) = (down v - 2*v + up v) / dy**2 ;

When programming the spreadsheet, we may think that the whole spreadsheet is the plate, and that the cells in the spreadsheet are the points on the plate which have distance **dx** to their immediate neighbors on their right and left and **dy** to the ones above and below them. In order to observe the temperature changes in time at different points on the plate after adding the heat source, first we define the point(cell) having the value of the temperature of the heat source(100), and then we define other points(cells) on the plate(spreadsheet) as

0 fby ((DX2(S) + DY2(S)) * dt + S) ;

where S is the spreadsheet variable. The above definition means that all the points except for the one with the heat source have temperature 0 initially, and that after every **dt** time slice their temperatures are increased by **DX2(S)+DY2(S)**. The definitions of the cells in the spreadsheet, therefore, reflect the pattern of temperature at the corresponding points on the plate. During evaluation, the user can observe the temperature of any points (cells) on the plate (spreadsheet) at any time. Table 5.1.1 gives some observation results from a special instance of the spreadsheet program described above. In this spreadsheet program, we assume 4dt = a^{-2}dxdy where a is the constant in the Heat-Flow Equation, and a = dx = dy = 1. In this particular case, in fact, the definitions of the non-source cells can be simply defined as

0 fby (left S + right S + down S + up S) / 4 ;

t = 0 (real time = 0)

S	L3	L2	L1	0	R1	R2	R3
U3	0	0	0	0	0	0	0
U2	0	0	0	0	0	0	0
U1	0	0	0	0	0	0	0
0	0	0	0	100	0	0	0
D1	0	0	0	0	0	0	0
U2	0	0	0	0	0	0	0
D3	0	0	0	0	0	0	0

Table 5.1.1(a) temperature of the plate at time 0

t = 1 (real time = 0.25)

S	L3	L2	L1	0	R1	R2	R3
U3	0	0	0	0	0	0	0
U2	0	0	0	0	0	0	0
U1	0	0	0	25.00	0	0	0
0	0	0	25.00	100	25.00	0	0
D1	0	0	0	25.00	0	0	0
U2	0	0	0	0	0	0	0
D3	0	0	0	0	0	0	0

Table 5.1.1(b) temperature of the plate at time 0.25

t = 2 (real time = 0.5)

S	L3	L2	L1	0	R1	R2	R3
U3	0	0	0	0	0	0	0
U2	0	0	0	6.25	0	0	0
U1	0	0	12.50	25.00	12.50	0	0
0	0	6.25	25.00	100	25.00	6.25	0
D1	0	0	12.50	25.00	12.50	0	0
U2	0	0	0	6.25	0	0	0
D3	0	0	0	0	0	0	0

Table 5.1.1(c) temperature of the plate at time 0.5

t = 3 (real time = 0.75)

S	L3	L2	L1	0	R1	R2	R3
U3	0	0	0	1.56	0	0	0
U2	0	0	4.68	6.25	4.68	0	0
U1	0	4.68	12.50	32.81	12.50	4.68	0
0	1.56	6.25	32.81	100	32.81	6.25	1.56
D1	0	4.68	12.50	32.81	12.50	4.68	0
U2	0	0	4.68	6.25	4.68	0	0
D3	0	0	0	1.56	0	0	0

Table 5.1.1(d) temperature of the plate at time 0.75

t = 4 (real time = 1)

S	L3	L2	L1	0	R1	R2	R3
U3	0	0	1.56	1.56	1.56	0	0
U2	0	2.34	4.68	10.93	4.68	2.34	0
U1	1.56	4.68	18.75	32.81	18.75	4.68	1.56
0	1.56	10.93	32.81	100	32.81	10.93	1.56
D1	1.56	4.68	18.75	32.81	18.75	4.68	1.56
U2	0	2.34	4.68	10.93	4.68	2.34	0
D3	0	0	1.56	1.56	1.56	0	0

Table 5.1.1(e) temperature of the plate at time 1

In the example given above, one can see that the intensional spreadsheet is extremely suitable for managing values which vary in three dimensions, or two spatial dimensions and one time dimension or three spatial dimensions. That is its most significant improvement over conventional spreadsheets.

5.2. Comparison With Conventional Spreadsheets

Compared to the conventional spreadsheets, the intensional spreadsheet has further advantages. The policy behind the design of this intensional spreadsheet is to improve conventional spreadsheets in order to provide users with more problem-solving capability, while keeping most of the features of conventional spreadsheets. In this section, we will compare the intensional spreadsheet with some conventional ones to see if the former can do what the latter can.

In conventional spreadsheets a formula of a cell is a simple mathematical or logical expression and no user-defined functions are allowed in the expression. It is obvious that a definition (formula) of a cell in the intensional spreadsheet can be similar or even more complicated. Also, in Chapter 3, we have shown that some built-in functions in conventional spreadsheets can be implemented by defining the proper user-defined functions. Among the features of the conventional spreadsheets, there are two main features: **Iteration** and **Goal Seeking**, which are called "advanced features". We will show how the intensional spreadsheet implements those features, and discuss the differences between them and some intensional features of the intensional spreadsheet. In the following, we refer to Multiplan[5] to compare iteration and Open Access[6] to compare goal seeking. Features of conventional spreadsheets that belong to the user interface, such as spreadsheet consolidation, multi-models and so on, will not be considered further.

5.2.1. Iteration

Two dimensions in conventional spreadsheets such as Multiplan are not enough for iteration because we can think of every iteration step as a variation of the spreadsheet with time. But this variation with time is very restricted and has a different interpretation from that of intensional spreadsheet programs.

When working with Multiplan, a user can define formulas in cells to refer to each other in a circular manner (called *circular references*), and in this way the user can use iteration. When Multiplan iterates, it calculates the spreadsheet over and over until some condition is satisfied. For each calculation, Multiplan uses the results of the previous calculation. The previous results do not solve the problem exactly, but each iteration yields results closer to the solution. Before every iteration, Multiplan performs a completion test. If the completion test is FALSE, it increments the iteration count by one, and begins an iteration. When the completion test is TRUE, it stops the iteration process. The Multiplan model (program) in Table 5.2.1(a) shows how to use iteration to solve a simple

circular reference[5]. This model is designed to calculate a bonus that is ten percent of net profit.

	1	2
1	Gross profit	1000.00
2	Bonus	=Net_profit - Bonus
3	Net_profit	=Gross_profit - Bonus

Table 5.2.1(a) a Multiplan model with circular references.

Here, cell R2C2 (Bonus) and cell R3C2 (Net_profit) circularly reference each other. The completion test uses the built-in function DELTA which is the difference between the values calculated previously and currently. If DELTA ≤ 0.01 then the test is TRUE; otherwise it is FALSE. The Table 5.2.1(b)-(d) are the results of the iterations at steps: 0, 2 and 4.

initial iteration step

	1	2
1	Gross profit	1000.00
2	Bonus	90.00
3	Net profit	900.00

Table 5.2.1(b) the result of (a) at initial iteration step

iteration step 2

	1	2
1	Gross profit	1000.00
2	Bonus	90.00
3	Net profit	909.00

Table 5.2.1(c) the result of (a) at iteration step 2

iteration step 4

	1	2
1	Gross profit	1000.00
2	Bonus	90.91
3	Net profit	909.09

Table 5.2.1(d) the result of (a) at iteration step 4

After the fourth iteration, the model is solved because 90.91(Bonus) is 10% of 909.09(Net_profit) and 909.09 is equal to 1000.00(Gross_profit) - 90.91.

The above example can be solved with the intensional spreadsheet by using intensional operators to define the formulas of cells. The intensional spreadsheet program in Table 5.2.2 gives the same results as the Multiplan model, where every iteration step corresponds to an increase by one of the time parameter.

S	R1	R2
D1	'Gross Profit'	1000.00
D2	'Bonus'	90 fby D1/10
D3	'Net Profit'	900 fby next(U2 - U1)

Table 5.2.2 an intensional program simulating iteration

One difference between the two spreadsheet programs is that in the intensional program the initial values are defined by the user while in the Multiplan model this is done by the spreadsheet itself. If the user wants to see the final result immediately after starting the evaluation as the automatic iteration in Multiplan, the above intensional spreadsheet program can be modified as is shown in Table 5.2.3.

DELTA = abs(next S - S) ;

S	R1	R2
D1	'Gross Profit'	1000.00
D2	'Bonus'	(90 fby D1/10) asa DELTA<0.01
D3	'Net Profit'	(900 fby next(U2 - U1)) asa DELTA<0.01

Table 5.2.3 an intensional program simulating iteration

During iteration, Multiplan calculates the values of the cells in a certain order. The Table 5.2.4 illustrates the order of the calculation during iteration.

	1	2	3	4	5
1	|	|	|	|	|
2	|	|	|	|	|
3	|	|	|	|	|
4	|	|	|	|	|
5	↓	↓	↓	↓	↓

Table 5.2.4 the evaluation order of Multiplan

The designers of Multiplan suggest putting all circular references in a single column so as to make it

easier to keep track of the calculation order. But in the intensional spreadsheet the evaluation order of the cells does not affect their values because the underlying programming language is declarative and the whole spreadsheet program is evaluated by eduction. To make the iteration in the intensional spreadsheet exactly the same as in Multiplan, we have to simulate the evaluation order in the intensional spreadsheet. To accomplish this, during evaluation, if a cell's formula refers to another cell's value which will be evaluated before it according to the evaluation order of Multiplan, it should refer to the cell's value at the current time instead of at a previous time. Therefore we need to add a **next** operator before every cell referenced in a cell's definition in the intensional spreadsheet program if the referenced cell is one of the predecessors in the evaluation order. This explains the intensional program in the example(Table 5.2.2) where the cell D3R2 is defined as **900 fby next(ledge S - up S)** because both cells D2R2(U1) and D1R2(U2), are before it in the evaluation order.

In Multiplan, a user can set up three different kinds of completion tests: (1) a completion test using the ITERCNT function which defines the maximum number of iterations; (2) a completion test using the DELTA function defines the maximum changes between answers from one iteration and the next iteration; and (3) a completion test using any logical expression which defines a condition that stops the iteration when the expression is TRUE. In the intensional spreadsheet, during iterations, a user only needs to set up one kind of completion test by using a logical expression and the intensional operator **asa**. For (1) above, we can define a global variable **ITERCNT** which is the iteration count as **ITERCNT = 0 fby ITERCNT + 1**, and define the logical expression as **ITERCNT** >*maximum iteration number*. For (2) above, we can define a function to simulate the **DELTA** function as **DELTA(S) = abs(next S - S)** or **DELTA(S) = max(abs(next S - S))**, where the former gives the difference between two successive iterations individually, (i.e. the difference may be different for individual cells), while the latter gives the difference globally, (i.e. the difference is the maximum one among the cells). The function **max (X)** returns the maximum value of X at all defined spatial points and the current time.

To sum up, using the intensional spreadsheet to perform iteration in the same way as in conventional spreadsheets needs two steps:

(1) Define the cells which are circularly referenced as formulas :

"initial value" **fby** "the circularly referenced expression"

where, whenever a cell is circularly referenced in the expression which will be evaluated before the current cell in the calculation order, the intensional operator **next** is added before it;

(2) Define a logical expression by using global variables or user-defined functions, and use them with the iteration operator **asa** to do the completion test.

The Multiplan iteration function can also be used to solve the classical heat transfer problem discussed earlier. That is, except for the heat source point, we can define cells in Multiplan spreadsheet as the values of their four neighbors divided by 4. During iteration the values (temperatures) of the cells change at every iteration step. But a significant difference between the solutions of the intensional spreadsheet and Multiplan is that in the former every cell's value is computed exactly from its neighbor's values at the *previous* time when the time parameter is increased, but in the latter a cell's value at every iteration step is computed *partially* from some neighbor's values at the *previous* iteration step if they are behind it in the evaluation order and *partially* from other neighbor's values at the *current* iteration step if they are before it in the evaluation order. In other words, the solution of the heat transfer problem by the intensional spreadsheet program exactly reflects the Heat-Flow Equation, but the solution by the conventional iteration spreadsheet does not. Generally speaking, this in fact reflects one of the most important differences between the time dimension in the intensional spreadsheet and the iteration in the conventional spreadsheets.

5.2.2. Goal Seeking

The "Goal Seeking" in Open Access[6] is an advanced option that enables users to specify target values or "goals" for one or more variables (cells) and then obtain values for the spreadsheet cells which achieve these goals. A goal is set for the dependent variable (cell), whose value is a function of a specific independent variable. Goal Seeking computes the value of the independent variable which will achieve the user's specified goal. For example, the Open Access model in Table 5.2.5 is designed to project a company's gross profits over next five years.

	A	B	C	D
1	Year	Revenue	Expenses	Profit
2	1986	1000.00	800.00	Revenue - Expenses
3	1987	B2 * Growth_rate	C2 * Growth_rate	Revenue - Expenses
4	1988	B3 * Growth_rate	C3 * Growth_rate	Revenue - Expenses
5	1989	B4 * Growth_rate	C4 * Growth_rate	Revenue - Expenses
6	1990	B5 * Growth_rate	C5 * Growth_rate	Revenue - Expenses
7	Growth Rate	1.05		

Table 5.2.5(a) an Open Access Model about profit

After evaluation, the profit model has the result like Table 5.2.4(b).

	A	B	C	D
1	Year	Revenue	Expenses	Profit
2	1986	1000.00	800.00	200.00
3	1986	1050.00	840.00	210.00
4	1986	1102.00	882.00	220.00
5	1986	1157.63	926.10	231.00
6	1990	1215.51	972.41	243.10
7	Growth Rate	1.05		

Table 5.2.5(b) the result of the profit model

If the user denotes the cell D6 or the profit in 1990 as the dependent variable with target value 300 and B7 or the growth rate as the independent variable, the required value of the growth rate (B7) after the Goal Seeking is done is 1.1066. That is, if the growth rate is 1.1066% and the revenue and expenses in 1986 are $1,000 and $800, respectively, after 5 years the company can obtain a profit of $300.

Goal Seeking in Open Access uses a mathematical model similar to that used by the iteration in Multiplan, but here circular references are not necessary. The Goal Seeking program uses Newton's algorithm to obtain solutions. The method may be briefly described as follows. The goal is to obtain a desired target value for a dependent variable (cell). The program is initialized by a value for the independent variable given by the program user. The algorithm uses this value to compute the value of the dependent variable. If the result is within 0.001 of the target value, the target is considered achieved and the procedure ends. If the result differs by more than 0.001, the procedure is repeated until the goal is reached.

In the intensional spreadsheet, if we have a built-in function or define a user-defined function to calculate the approximation for the independent variable at each iteration step, we can also do goal seeking. For example, for the profit model in Open Access, we can define a corresponding intensional spreadsheet program in Table 5.2.6(a) and the result of goal seeking in table 5.2.6(b).

```
approx(dep,targ,init1,init2) = ab2
  where
    ab2 = (a + b) / 2;
    D = dep - targ;
    a = init1 fby if D < 0 then ab2 else a fi;
    b = init2 fby if D > 0 then ab2 else b fi;
  end
```

S	0	R1	R2	R3
0	'Year'	'Revenue'	'Expenses'	'Profit'
D1	1986	1000.00	800.00	L2 - L1
D2	U1 + 1	U1*home D6R1	U1* home D6R1	L2 - L1
D3	U1 + 1	U1*home D6R1	U1* home D6R1	L2 - L1
D4	U1 + 1	U1*home D6R1	U1* home D6R1	L2 - L1
D5	U1 + 1	U1*home D6R1	U1* home D6R1	L2 - L1
D6	'Growth Rate'	approx(U1R2,300,1.05,2)		

Table 5.2.6(a) an intensional goal seeking spreadsheet program

S	0	R1	R2	R3
0	Year	Revenue	Expenses	Profit
D1	1986	1000.000	800.000	200.000
D2	1986	1106.682	885.364	221.336
D3	1986	1224.746	979.797	244.949
D4	1986	1355.406	1084.324	271.081
D5	1990	1500.004	1200.003	300.000
D6	Growth Rate	1.106*		

* the iteration finished at the 19th step with the difference <0.001.
Table 5.2.6(b) the result of the goal seeking in (a)

For simplicity we use a binary algorithm, instead of the Newton's algorithm, to calculate the desired value of the independent variable (Growth Rate). The function approx(dep,targ,init1,init2) is a function which calculates the approximate value for the cell D6R1 (Growth Rate) every time the time parameter is increased. The parameter *dep* is the dependent variable (cell) for the goal seeking; the parameter *targ* is the target value of the dependent variable; and the parameters *init*1 and *init*2 are the initial values of the independent variable to start iteration according to the algorithm. As when simulating Multiplan iteration, if the user wants to see the result of goal seeking immediately after giving initial values, we can use the intensional operator asa with a predicate to stop the iteration, where the predicate should have the form **next x - x < 0.001.** For example, in the above program we can define cell D6R1(Growth Rate) as

approx(S,U1R2,300,1.05,2) asa next U1R2 - U1R2< 0.001

From the above, we see that it is possible to simulate Goal Seeking by using corresponding intensional spreadsheet programs.

5.2.3. Performance Comparison

It is very difficult to compare precisely the performance of our experimental implementation of the intensional spreadsheet with conventional spreadsheets. The intensional spreadsheet is suitable for more applications than conventional ones, or in other words the former has more functions than the latter, so the performance of many functions in the intensional spreadsheet cannot be compared with

the conventional ones. Even for the common functions, their performance is hard to compare because the current experimental intensional spreadsheet is implemented in a Unix-Vax system, whereas almost all conventional spreadsheets are implemented on microcomputers.

Although we cannot do a precise comparison, we must admit that in many cases the current experimental intensional spreadsheet may not perform any better than existing conventional ones because (1) the intensional spreadsheet is built on a stack-based abstract machine in which the stack operations may slow down speed; (2) the C code of the current spreadsheet interpreter is not optimized; (3) the current non-dependency analysis algorithm makes the re-evaluations inefficient when modifications occur, while the existing spreadsheets are well-optimized before becoming commercial. However, for the examples given in this paper and some other spreadsheet programs, the performance of our current spreadsheet is reasonable.

CHAPTER 6

CONCLUSIONS AND FURTHER WORK

We have described the principles of an intensional spreadsheet and its implementation. The intensional spreadsheet is based on intensional logic and is programmed in a high level declarative language. The key features of the spreadsheet are a third dimension for time, user-defined functions and global variables.

Using intensional logic to program a spreadsheet brings a new concept to spreadsheet programming. Instead of thinking about cells in a spreadsheet as individually unrelated variables, the whole spreadsheet can be regarded as a variable varying in space and time an can be programmed intensionally using intensional operators. The advantages of intensionally programming the spreadsheet are that it makes the structure of a spreadsheet program logical and clear, and it makes the designs of spreadsheet programs more problem-oriented because intensional logic itself is abstracted from the real world.

Combining a spreadsheet design principle with an existing high level declarative language not only shows another application of the declarative language but also gives the spreadsheet greater vitality. It shows that the declarative language (here Plane Lucid) is highly suitable for spreadsheet programming. By being based on a high level language spreadsheets can acquire programming and problem-solving ability. In addition, further development of programming languages such us Plane Lucid may lead to further improvements in spreadsheet techniques. Conversely, a deep study of spreadsheets may help further development of the language.

It was seen that demand-driven computation is extremely suitable for spreadsheet techniques because the evaluation of cells in the spreadsheet is governed by the data dependencies among the cells. In particular, the unnecessary computations are avoided and only the minimum evaluations for

the cells and other variables are performed. The storage strategy for evaluated values makes the evaluation more efficient when the spreadsheet variable varies in space and time. If we did not use eduction it would be very difficult, if not impossible, to solve problems such as the heat transfer using the spreadsheet.

On the application side, the most important improvements of the intensional spreadsheet are allowing three dimensional programming, allowing user-defined functions and having global variables. This makes the intensional spreadsheet suitable for more applications than conventional spreadsheets and enables more users to solve their problems, whether they be novices or computer scientists. The intensional spreadsheet is especially useful for study of problems involving changes in three dimensions. In short, if we can not say that the described intensional spreadsheet is a revolution over the conventional spreadsheets, we can at least say that the former is a significant attempt to improve the latter.

Apart from the tasks of improving our current spreadsheet in efficiency by using the dependency analysis and value-warehouse management, which were mentioned in Chapter 5, we currently need to do at least four more things to make the intensional spreadsheet really practical and useful. The current implementation of the spreadsheet interpreter must be optimized, even redesigned, because the abstract machine on which it is based is not specially designed for the spreadsheet and therefore is quite inefficient. The current user-interface of the spreadsheet must be improved because a user friendly interface for a spreadsheet is just as important as its programming and calculating power. We must prepare to transplant the intensional spreadsheet onto microcomputers because most users want to use spreadsheets in their own office or home computers. And finally we should combine our spreadsheet with other integrated software, such as data base and graphics, because only with such combinations can the spreadsheet show its ability to solve more complicated problems, and give the users results directly perceived through the visual senses.

BIBLIOGRAPHY

[1] D.M. Castlewitz, "VisiCalc – Program Made Easy," Osborne/McGraw-Hill, Berkley, California, 1983.

[2] D. Smithy-Willis, J. Willis and M.K. Miller, "How To Use SuperCalc," Tab Books Inc., Blue Ridge Summit, PA, 1983.

[3] Weber System Inc. Staff, "Lotus 1-2-3 User's Handbook," Ballatine Book, New York, 1984.

[4] Ewing, D. Paul and G.T. LeBlond, "Using Symphony," Que Corporation, Indianapolis, Indiana, 1984.

[5] Microsoft Corporation, "Microsoft Multiplan – Electronic Worksheet User Manual," 1984.

[6] Software Products International Inc(SPI), "Open Access – An Integrated System User Manual," 1983.

[7] A.A. Faustini and W.W. Wadge, "Intensional Programming," Technical Report DCS-55-IR, Department of Computer Science, University of Victoria, 1986.

[8] R. Thomason, editor, "Formal Philosophy, Selected Papers of R. Montague," Yale University Press, new Haven, Conn, 1974.

[9] E.A. Ashcroft and W.W. Wadge, "Lucid – A Formal System for Writing and Proving Programs," SIAM J. Computing, Vol. 5, No. 3, Sept. 1976, pp. 336-354.

[10] E.A. Ashcroft and W.W. Wadge, "Lucid, a Nonprocedural Programming Language," Comm.Acm, Vol.20,No. 7, July 1977, pp. 519-526.

[11] E.A. Ashcroft and W.W. Wadge, "Structured Lucid," Technical Report CS-79-21, University of Waterloo, 1979.

[12] W.W. Wadge and E.A. Ashcroft , "Lucid, the Dataflow Programming Language," Academic Press, London, England, 1985.

[13] A.A. Faustini and W.W. Wadge, "An Eductive Interpreter for pLucid," Technical Report TR-004-86, Department of Computer Science, Arizona State University, 1986.

[14] E.A. Ashcroft, "Ferds – Massive Parallelism in Lucid," Proceedings of the Phoenix Conference on Computers and Communications, IEEE, March 1985, pp. 16-21.

[15] E.A. Ashcroft, A.A. Faustini and B. Huey, "Eduction – A Model of Parallel Computation and The Programming Language Lucid," Proceedings of the Phoenix Conference on Computers and Communications, IEEE, March 1985, pp. 9-15.

[16] W.W. Wadge, "The Popshop Philosophy," Department of Computer science, University of Victoria, 1985.

[17] R.M. Burstall, J.S. Collins and R.J. Popplestone, "Programming in POP-2," Edinburgh University Press, Edinburgh, 1971.

[18] J. Amsterdam, "Build a Spreadsheet Program," BYTE, June 1986, pp. 97-108.

[19] K.L. Nielsen, "Methods in Numerical Analysis," The Macmillan Company, New York, 1956, pp. 246-258.

APPENDIX A

PLANE LUCID PROGRAMMING LANGUAGE

(1) Syntax

```
accept::
        expr
|       expr ";"
;

constant::
        NUMC
|       WORDC
|       STRINGC
|       LISTC
;

|       "[%" "%]"
;

listbody::
        expr
|       expr "," listbody
;

expr::
        expr "where" wherebody "end"
|       IDENT
|       constant
|       list
|       "(" expr ")"
|       "not" expr
|       "next" expr
|       "next" "(" expr "," expr ")"
|       "previous" expr
|       "previous" "(" expr "," expr ")"
|       "up" expr
|       "up" "(" expr "," expr ")"
|       "down" expr
|       "down" "(" expr "," expr ")"
|       "right" expr
|       "right" "(" expr "," expr ")"
|       "left" expr
|       "left" "(" expr "," expr ")"
```

```
|       "first" expr
|       "first" "(" expr "," expr ")"
|       "side" expr
|       "side" "(" expr "," expr ")"
|       "ledge" expr
|       "ledge" "(" expr "," expr ")"
|       "home" expr
|       expr "atpos" "(" expr "," expr ")"
|       expr "fby" expr
|       expr "pby" expr
|       expr "vsby" expr
|       expr "vpby" expr
|       expr "hsby" expr
|       expr "hpby" expr
|       expr "asa" expr
|       expr "haca" expr
|       expr "hbaca" expr
|       expr "vaca" expr
|       expr "vbaca" expr
|       expr "wherever" expr
|       expr "hwherever" expr
|       expr "vwherever" expr
|       expr "upon" expr
|       expr "hupon" expr
|       expr "vupon" expr
|       IDENT "("   exprlist ")"
|       "substr" "(" expr "," expr "," expr ")"
|       expr "^" expr
|       expr "::" expr
|       expr "<>" expr
|       expr "+" expr
|       expr "-" expr
|       "-" expr %prec "*"
|       "+" expr %prec "*"
|       expr "*" expr
|       expr "**" expr
|       expr "/" expr
|       expr "mod" expr
|       expr ">" expr
|       expr "<" expr
|       expr "<=" expr
|       expr ">=" expr
|       expr "eq" expr
|       expr "ne" expr
|       expr "and" expr
|       expr "or" expr
|       ifexpression
|       casexpr
;

ifexpression::
```

```
        "if" expr "then" expr ending
    ;

    ending::
        "elseif" expr "then" expr ending
    |   "else" expr "fi"
    ;

    casexpr::
      "case"  expr  "of" casebody "end"
    ;

    casebody::
        expr ":" expr ";" casebody
    |   "default" ":" expr ";"
    ;

    wherebody::
        assertion
    |   assertion wherebody
    ;

    identlist::
        IDENT
    |   IDENT "," identlist
    ;

    assertion::
        IDENT "=" expr ";"
    |   IDENT "(" identlist ")" "=" expr ";"
    ;

    exprlist::
        expr
    |   expr "," exprlist
    ;
```

(2) The Semantics of the Primitive Intensional Operators

* when the operation is being applied,

h is the current horizontal coordinate in space;

v is the current vertical coordinate in space;

t is the current time.

$(\textbf{ next } x\ )_t^{vh} = \mathbf{x}_{t+1}^{vh}$

$(\textbf{ next(x,y) })_t^{vh} = \mathbf{x}_{t+y}^{vh}$

$(\textbf{ previous } x\ )_t^{vh} = \mathbf{x}_{t-1}^{vh}$

$(\textbf{ previous(x,y) })_t^{vh} = \mathbf{x}_{t-y}^{vh}$

$(\textbf{ right } x\ )_t^{vh} = \mathbf{x}_t^{v(h+1)}$

$(\textbf{ right(x,y) })_t^{vh} = \mathbf{x}_t^{v(h+y)}$

$(\textbf{ left } x\ )_t^{vh} = \mathbf{x}_t^{v(h-1)}$

$(\textbf{ left(x,y) })_t^{vh} = \mathbf{x}_t^{v(h-y)}$

$(\textbf{ up } x\ )_t^{vh} = \mathbf{x}_t^{(v+1)h}$

$(\textbf{ up(x,y) })_t^{vh} = \mathbf{x}_t^{(v+y)h}$

$(\textbf{ down } x\ )_t^{vh} = \mathbf{x}_t^{(v-1)h}$

$(\textbf{ down(x,y) })_t^{vh} = \mathbf{x}_t^{(v-y)h}$

$(\textbf{ first } x\ )_t^{vh} = \mathbf{x}_0^{vh}$

$(\textbf{ first(x,y) })_t^{vh} = \mathbf{x}_y^{vh}$

$(\textbf{ side } x\ )_t^{vh} = \mathbf{x}_t^{v0}$

$(\textbf{ side(x,y) })_t^{vh} = \mathbf{x}_t^{vy}$

$(\textbf{ ledge } x\ )_t^{vh} = \mathbf{x}_t^{0h}$

$(\textbf{ ledge(x,y) })_t^{vh} = \mathbf{x}_t^{yh}$

$(\textbf{home}\ x)_t^{vh} = \mathbf{x}_t^{00}$

$(z\ \textbf{atpos}\ (y,x))_t^{vh} = \mathbf{z}_t^{yz}$

$$(x\ \textbf{fby}\ y)_t^{vh} = \begin{cases} \mathbf{x}_t^{vh} & t \le 0 \\ \mathbf{y}_{t-1}^{vh} & t > 0 \end{cases}$$

$$(x\ \textbf{pby}\ y)_t^{vh} = \begin{cases} \mathbf{x}_{t+1}^{vh} & t < 0 \\ \mathbf{y}_t^{vh} & t \ge 0 \end{cases}$$

$$(x\ \textbf{hsby}\ y)_t^{vh} = \begin{cases} \mathbf{x}_t^{vh} & h \le 0 \\ \mathbf{y}_t^{v(h-1)} & h > 0 \end{cases}$$

$$(x\ \textbf{hpby}\ y)_t^{vh} = \begin{cases} \mathbf{x}_t^{v(h+1)} & h < 0 \\ \mathbf{y}_t^{vh} & h \ge 0 \end{cases}$$

$$(x\ \textbf{vsby}\ y)_t^{vh} = \begin{cases} \mathbf{x}_t^{vh} & v \le 0 \\ \mathbf{y}_t^{(v-1)h} & v > 0 \end{cases}$$

$$(x\ \textbf{vpby}\ y)_t^{vh} = \begin{cases} \mathbf{x}_t^{(v+1)h} & v < 0 \\ \mathbf{y}_t^{vh} & v \ge 0 \end{cases}$$

(3) The definitions of The Advanced Intensional Operators

```
whenever(x,y) = if first y then x fby z else z fi
          where
            z = whenever(next x, next y);
          end
```

```
hwherever(x,y) = left bhwherever(x,y) hpby fhwherever(x,y)
```

```
          where
            fhwherever(x,y) = if side y then x hsby z else z fi
                              where
                                z = fhwherever(right x, right y);
                              end;
            bhwherever(x,y) = if side y then x hpby z else z fi
                              where
                                z = bhwherever(left x, left y);
                              end;
          end


vwherever(x,y) = down bvwherever(x,y) vpby fvwherever(x,y)
          where
            fvwherever(x,y) = if ledge y then x vsby z else z fi
                              where
                                z = fvwherever(up x, up y);
                              end;
            bvwherever(x,y) = if ledge y then z vpby x else z fi
                              where
                                z = bvwherever(down x, down y);
                              end;
          end


upon(x,y) = x fby
        if first y
          then upon(next x, next y)
          else upon(x, next y)
        fi;


hupon(x,y) = left bhupon(x,y) hpby fhupon(x,y)
        where
          fhupon(x,y) = x hsby
                    if side y
                      then fhupon(right x, right y)
                      else fhupon(x, right y)
                    fi;

          bhupon(x,y) = (if side y
                          then bhupon(left x, left y)
                          else bhupon(x, left y)
                        fi) hpby x;
        end


vupon(x,y) = down bvupon(x,y) vpby fvupon(x,y)
```

```
     where
       fvupon(x,y) = x vsby
                  if ledge y
                    then fvupon(up x, up y)
                    else fvupon(x, up y)
                  fi;

       bvupon(x,y) = (if ledge y
                        then bvupon(down x, down y)
                        else bvupon(x, down y)
                      fi) vpby x;
     end


asa(x,y) = first whenever(x,y) ;


haca(x,y) = side hwherever(x,y) ;


hbaca(x,y) = side bhwherever(x,y)
     where
       bhwherever(x,y) = if side y then x hpby z else z fi
                      where
                        z = bhwherever(left x, left y);
                      end;
     end


vaca(x,y) = ledge vwherever(x,y) ;


vbaca(x,y) = ledge bvwherever(x,y)
     where
       bvwherever(x,y) = if ledge y then x vpby z else z fi
                      where
                        z = bvwherever(down x, down y);
                      end;
     end
```

(4) The Short Forms of the Intensional Spatial Function Calls

* In the following, S is the spreadsheet variable, p,p1,p2 are variables and i,j are non-negative numbers.

R(p) = right(S,p)
L(p) = left(S,p)
U(p) = up(S,p)
D(p) = down(S,p)

UR(p1,p2) = up(right(S,p2),p1)
UL(p1,p2) = up(left(S,p2),p1)
DR(p1,p2) = down(right(S,p2),p1)
DL(p1,p2) = down(left(S,p2),p1)

Ri = right(S,i)
Li = left(S,i)
Ui = up(S,i)
Di = down(S,i)

UiRj = up(right(S,j),i)
UL(i,j) = up(left(S,j),i)
DR(i,j) = down(right(S,j),i)
DL(i,j) = down(left(S,j),i)

APPENDIX B

THE COMMANDS OF THE INTENSIONAL SPREADSHEET

(1) Edit: **v** or **e**

A user can use either of the edit commands to enter or modify a definition for a cell, a global variable or a user-defined function. Using **v** command enter a "vi"-like edit version and using **e** command enter a line edit version.

Command Format: **v[cgf]**
e[cgf]

Explanation:

vc or ec --- edit the current cell.

vg or ec --- edit a global variable named by the user.

vf or ec --- edit a function named by the user.

(2) Calculation: **c**

A user can use the calculation command to calculate any regions of cells on the spreadsheet at the current time indicated by the user. The undefined cells in the regions are not calculated.

Command Format: **c**

Explanation:

When the command is entered, the user will be required to enter the calculation regions. The format of the regions are

A1[:A2];B1[:B2];....

where A1 or B1 is the left top cell of the region and A2 or B2 is the right bottom cell of the region.

(3) Time: **t**

A user can use the time command to change the current time to any time. The spreadsheet is recalculated automatically whenever the current time is changed.

Command Format: **t**

Explanation:

When the command is entered, the user is required to enter a data item to set new time. The format of the entered data item is

f --- the new time is the current time plus 1 ;
b --- the new time is the current time minus 1 ;
+n --- the new time is the current time plus n ;
-n --- the new time is the current time minus n ;
n --- the new time is n .

(4) Delete: **d**

A user can use the delete command to delete any global variables and user-defined functions as well as to delete the definitions of any regions of cells.

Command Format: **d[cgf]**

Explanation:

dc delete some regions of cells, the format of the regions which the user enters is as the same as showed (2).

dg delete a global variable indicated by the user.

df delete a user-defined function indicated by the user.

(5) Copy: **y**

A user can use the copy command to copy the definition of a cell to any regions of cells or to copy the definitions of a region of cells to any other regions which have the same size.

Command Format: **y**

Explanation:

When the command is entered, the user be required to enter the source and destination.

a. If the source is a cell, the destination can be any regions formated as the same as shown in (2); after the copy operation all the cells in the regions have the definitions as the same as the source cell.

b. If the source is a region of cells, the destination must be list of cells separated by semi-column, each of which is the left top cell of a region in which, after the copy operation, every cell has the same definition as the corresponding cell in the source region.

(6) Display: **s**

A user can use the display command to display the definition of a cell, a global variable or a user-defined function as well as the value of a cell in the display window.

Command Format: **s[c[dv]gf]**

Explanation:

scd display the definition of a cell indicated by the user.

scv display the current value of a cell indicated by the user.

sg display the definition of a global variable indicated by the user.

sf display the definition of a user-defined function indicated by the user.

(7) Clear: **l**

A user can use the clear command to clear displayed values in any regions of cells, but not to delete the definitions of those cells.

Command Format: **l**

Explanation:

The clear regions which the user enters later as the same as showed in (2).

(8) Width: **w**

A user can use the width command to change the width of a column of the spreadsheet.

Command Format: **w**

Explanation:

a. The command always changes the column in witch the current cell stays.

b. The column width entered by the user cannot be less than the minimum column width, otherwise the standard width is set.

(9) Move: **m** or *arrows*

A user can use the move command to indicate the current cell and to select a region of cells to be displayed on the screen.

Command Format: **m**
←
→
↑
↓

Explanation:

m Move the new indication of the current cell to the cell indicated by the user. If that cell is not in the current screen window, change the current screen window to make the new current cell in the left top of the window.

← Move the new indication of the current cell to the left cell. If that cell is not in the current screen window, move the window left one column.

→ Move the new indication of the current cell to the right cell. If that cell is not in the current screen window, move the window one column.

↑ Move the new indication of the current cell to the cell above. If that cell is not in the current screen window, move the window up one column.

↓ Move the new indication of the current cell to the cell below. If that cell is not in the current screen window, move the window down one column.

(10) Help: **h**

A user can use the help command to find all the user-interface commands with the formats and the brief explanations.

Command Format: **h**

(11) Quit: **q**

A user can use the quit command to quit from the spreadsheet.

Command Format: **q**

VITA

Surname: Du Given Names: Weichang

Place of Birth: Tianjin, China Date of Birth: March 16, 1953

Educational Institutions Attended, with Dates of Entering and Leaving:

Beijing Institute of Computer, Beijing China 1979 to 1983

University of Victoria, B.C. Canada 1984 to 1986

Degrees, Diplomas, Etc., Awarded, with Dates and Names of Institutions:

B.Sc. Beijing Institute of Computer Beijing China

Honors and Awards:

University of Victoria Special Fellowship 1984/1986

Publications:

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis (the title of which is shown below) to users of the University of Victoria Library, and to make *single copies only* for such users or in response to a request from the library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

An Intensional 3-D Spreadsheet and Its Implementation

Author Weichang Du

Weichang Du

Dec 23, 1986

Date