GENERATING INDEXED FORMAL SOFTWARE DOCUMENTS

Generating Indexed Formal Software Documents

By
Li Wang, MLIS, B.Sc.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree

Master of Science

MASTER OF SCIENCE (1999) McMaster University
(Computer Science) Hamilton, Ontario, Canada

TITLE: Generating Indexed Formal Software Documents

AUTHOR: Li Wang
MLIS (University of Western Ontario, London, Ontario)
B.Sc (Central-China Normal University, Hubei, China)

SUPERVISOR: Dr. David L. Parnas

NUMBER OF PAGES: x, 81

ABSTRACT

Precise software documentation can be written as a set of mathematical expressions, but in practice, people require a mixture of informal (English) material and formal mathematical expressions. Furthermore, most documents for real software comprise hundreds of tables. Appropriate indices are required to efficiently search for information. In this work we consider the problem of automatic preparation of indices for such software documents. The user is able to prepare text and diagrams using a system such as LaTeX, while using the TTS system [21] to produce the tables. The TTS system will then produce LaTeX representations of the tables to be inserted in the document using a tool called TLT (Table LaTeX Tool). Finally, the DIT (Document Indexing Tool) will provide a set of indices indicating the table, row, column and page number where each symbol appears.

This work describes the design and development of TLT and DIT for generating indexed software documents. The indices generated by DIT can also be used for efficiently editing tabular expressions, and checking tabular documentation for correctness.

ACKNOWLEDGEMENTS

I would first and foremost like to express my sincere thanks and deep appreciation to my supervisor, Dr. Dave L. Parnas, for his guidance, advice, and enthusiasm throughout my thesis work. Without his constant encouragement and support, it would have been impossible for me to finish this work.

I am grateful to Dr. Sanzheng Qiao and Dr. Martin von Mohrenschildt, for their valuable suggestions and comments. I would like to thank Dr. Ridha Khedri for his kind assistance in LaTeX. I would also like to thank all the members of the Software Engineering Research Group, especially Dennis Peters, Ruth Abraham and Jianwei Zhou, for their helpful discussions and valuable suggestions.

Special thanks to my husband, Xiaoming and my family, for their love, encouragement and support.

Finally, I would like to thank the Communications and Information Technology Ontario (CITO), Natural Sciences and Engineering Research Council (NSERC) and Bell Canada for the financial support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter provides a brief introduction to the thesis including the background, motivation, and purpose.

1.1 Background

With the wide-spread use of computers, software has become an essential part of society. Software is used in many areas of our lives, including health care, communications, banking, transportation, manufacturing, household appliances, etc. With the increasing dependence of society on software, the correct behaviour of software is becoming more and more crucial. However, most software on the market is not error-free. As a matter of fact, many large software systems are often released with thousands of known, but uncorrected, bugs for such reasons as cost, schedule, etc. [4] Consequently, there is an urgent need to improve the quality of software to ensure its trustworthiness and reliability. In addition, with the demand for increasingly sophisticated computer applications, the cost of software, particularly maintenance cost, is becoming higher and higher [23, 24, 19].

As a fundamental approach, documentation can be used to improve the quality of software and reduce its cost in various ways: guiding programmers, aiding maintainers, facilitating communication among designers, programmers and users, reducing cost when software needs to be extended or modified, etc. [7, 12] The increasing need of software reliability and the high cost of software maintenance have

led to a growing acceptance of the importance of software documentation [22].

To be really useful, software documentation has to be correct, complete, precise, and easy to read [15, 20, 2]. Since software documentation using natural language (e.g. English) are usually imprecise and inadequate, the Software Engineering Research Group (SERG) at McMaster University advocates using mathematic functions and relations to describe the intended behaviour of computer systems [7, 13]. With mathematical precision, automatic checking and testing can be conducted to improve the reliability of software [17, 3]. Unfortunately, although conventional (linear) mathematical notation works well for short mathematical formulas, it is not well-suited to long ones. Furthermore, in practice, using traditional mathematical notation to document real software products often results in long, complex expressions [1, 7, 13]. If software documentation is too complex, quite often, it has very little value, no matter how precise and accurate it might be. Notations that are easy to read and review are therefore necessary for the wide acceptance of precise documentation, particularly mathematical documentation.

To address the above needs, Parnas and his colleagues [13, 6] developed tabular notations (tables) for describing the functions and relations in computer systems. Expressions represented as tables are equivalent to expressions written in more traditional notation. However, tabular notation makes mathematical expressions more understandable and easier to review, as illustrated by the comparison between Figure 1.1 and Table 1.1. The advantages of tabular expressions over traditional expressions are as follows: [7, 1, 18]

- Easier to write and read. There are usually many cases to be identified when designing software. Table structure allows a person to write and read each case separately.
- Easier to inspect. Since each expression in a cell only applies to a subset of the function's domain, it is easier to check if all the cases have been covered using table format.
- Less complexity. Table structure makes it possible to eliminate many unnecessary repetitions of common sub-expressions. With the reduced duplication, expressions are clearer and more concise.

	$y > 5$	$y = 5$	$y < 5$
$x > 3$	y^2	0	y
$x = 3$	27	$-x$	$y + 27$
$x < 3$	$x + y$	$-y$	$x - y$

Table 1.1: The Function f(x,y) Represented as a Table

$$f(x,y) = \begin{cases} y^2 & \text{if } x > 3 \wedge y > 5 \\ 0 & \text{if } x > 3 \wedge y = 5 \\ y & \text{if } x > 3 \wedge y < 5 \\ 27 & \text{if } x = 3 \wedge y > 5 \\ -x & \text{if } x = 3 \wedge y = 5 \\ y + 27 & \text{if } x = 3 \wedge y < 5 \\ x + y & \text{if } x < 3 \wedge y > 5 \\ -y & \text{if } x < 3 \wedge y = 5 \\ x - y & \text{if } x < 3 \wedge y < 5 \end{cases}$$

Figure 1.1: The Function f(x, y) Written in Traditional Notation

1.2 Thesis motivation

Tabular documentation provides mathematical precision and readability, which are very valuable for developing and maintaining reliable computer systems. However, existing documentation tools are not well-suited for producing such documents – creating and editing tables is a time-consuming and error-prone process. To overcome this problem, a set of tools, known as Table Tool System (TTS), is being developed to facilitate the manipulation of tabular expressions such as creating, editing, analyzing, etc. [21]

In practice, it is very likely that people require both informal (English) material and formal mathematical expressions in a single document. For example, some narrative description can be placed around each table to help users (who may have less technical background) to understand it. It is necessary to provide TTS with the

capability of producing software documents that contain both informal and formal material. In addition, most documents for real software comprise hundreds of tables, which makes it frustrating and time-consuming to find the desired information, such as where the definition of a specific symbol is and which condition will change the value of this symbol. It is important to provide TTS with the ability to generate a set of indices for such documents.

There are many typesetting and word processing systems available in the market. It is thus not necessary to provide TTS with the capability of both text editing and table manipulation, if we can have a table output format that can be easily inserted into documents prepared by the existing systems. A table output tool (called Table LaTeX Tool) has been provided for TTS. To efficiently search information contained in tables, an automatic indexing tool (called Document Indexing Tool) has also been built for TTS to indicate in which table a particular symbol appears and the row, column and page number. With these two tools, TTS is able to generate software documents that contain both formal and informal material, as well as a set of indices for quick reference.

As a document preparation system, LaTeX was chosen for TTS to generate indexed formal software documents due to the following reasons:

- LaTeX output is portable. Tables represented in LaTeX can be easily imported into any LaTeX document. In addition, LaTeX representations of tables can be converted into postscript format, which can be imported into documents prepared by other systems such as FrameMaker.

- The cross-references facility of LaTeX can be used to generate indices for tables, indicating in which table and page a particular symbol has been defined and used.

- Mathematical formulas and tables can be easily formatted and typeset in LaTeX.

- LaTeX is extendable. If a new feature is required, we can look for an "add-on" or write it ourselves. For example, with LaTeX2HTML package of LaTeX, documents produced by TTS can be part of the World Wide Web and the cross-references of LaTeX documents can be automatically translated into hyper-links.

1.3 Thesis purpose

The purpose of this thesis is to develop two tools, namely Table LaTeX Tool (TLT) and Document Indexing Tool (DIT), for generating indexed formal software documents. TLT produces LaTeX representation of expressions, and DIT automatically generates a set of indices for expressions, indicating the table, row, column and page number where each symbol appears. To produce indexed formal software documents, indices generated by DIT also need to be converted to LaTeX. In addition, this thesis also provides an opportunity for applying software engineering principles to software development practice in a large, integrated and ongoing project environment. The following goals are to be achieved:

- Producing table outputs that can be easily combined with informal material.
- Generating adequate and well-organized indices for efficient search and ease of access.
- Providing different layout options for users to customize the format of tables.
- Handling both scalar expressions and tabular expressions.
- Handling complicated tables such as nested tables, long tables and wide tables.
- Producing software that are easy to extend and modify.

1.4 Thesis outline

Chapter 2 describes the support systems of TLT and DIT, namely Table Tool System and LaTeX. Since TTS is based on tabular expressions, a brief introduction of tabular expressions is also provided.

Chapter 3 describes TLT and DIT in general, including the overview, features and user's guide.

Chapter 4 describes the design of TLT, which includes the requirements, module decomposition, module guide, module interface specifications and module internal design.

Chapter 5 describes the design of DIT, which includes the requirements, module decomposition, module guide, module interface specifications and module internal design.

Chapter 6 summaries the results of the design and implementation of TLT and DIT, discusses the limitations and future work, and draws the conclusions.

Chapter 2

Support Systems

This chapter describes the support systems of TLT and DIT, namely Table Tool System and LaTeX. Since TTS is based on tabular expressions, a brief description of tabular expressions is also presented.

2.1 Tabular Expressions

Tabular expressions (tables) are multi-dimensional expressions, which are often easier to read and interpret than conventional (linear) expressions. Tabular expressions are constructed recursively from scalar expressions and grids [13]. A *scalar expression* is a term or predicate expression as defined in [14]. A *Grid* is an indexed set of cells that contain either scalar expressions or previously constructed grids. A tabular expression consists a main grid G and header grids H_1, H_2, ..., $H_{dim(G)}$, where dim(G) is the dimensionality (the number of dimensions) of the grid G [13]. Table A.2 is an example of two-dimensional table, which has a main grid G and two header grids H_1 and H_2. For more details on the syntax of tables and grids refer to [13].

Table 2.1: An Example of Two-dimensional Table

$count(x, A, lower, upper) defined$		H_2
	$A[lower] = x$	$\neg(A[lower] = x)$
$lower < upper$	$1 + count(x, A, lower + 1, upper)$	$count(x, A, lower + 1, upper)$
$lower = upper$	1	0
H_1		G

2.2 Table Tool System

Tabular expressions provide mathematical precision for software documentation. However, manipulation of tables is tedious and error-prone due to the lack of suitable tools. To facilitate the use of tabular expressions in software documentation, a set of tools, known as the Table Tool System (TTS), is being developed by the Software Engineering Research Group (SERG) at McMaster University. The capabilities of TTS currently include expression creation, formatting, transformation, simplification, composition as well as specification checking and test oracle code generation. TTS consists of three parts: TTS Kernel, TTS Infrastructure, and TTS Applications. Figure 2.1 illustrates the TTS structure and the positions of TLT and DIT in TTS. Detailed description on the design of TTS is provided in the TTS Developer's Guide [21].

2.2.1 TTS Kernel

The TTS Kernel hides the representation of expressions and the algorithms for their manipulation. It includes two parts: Table Holder and Symbol Information.

Table Holder (TH)

The Table Holder is a set of modules to represent and manipulate expressions. Expressions are stored in the Table Holder in a tree structure. Each node of a tree is a *symbol*, which connects to the root node by a unique path. Each node and its

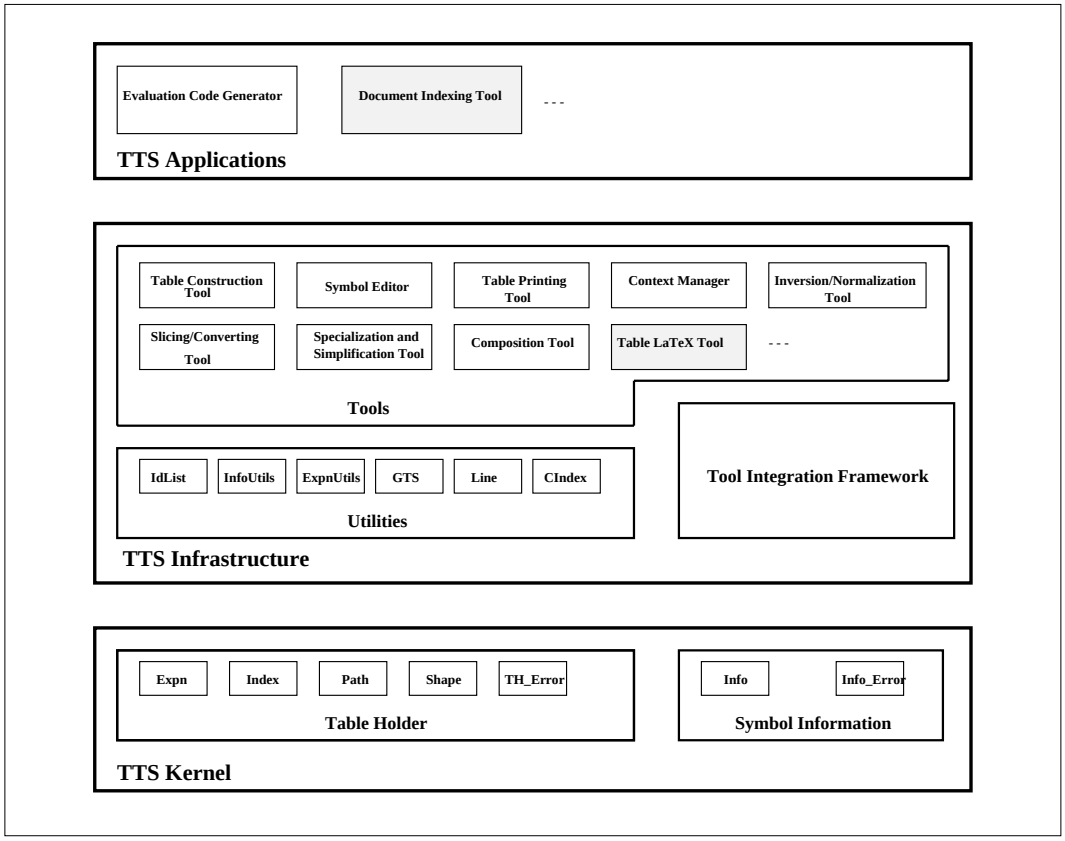


Figure 2.1: TTS Structure

children represent a *sub-expression*. The descendents of a table are its cells and the descendents of a function are its arguments. Constants and variables are the leaves of an expression tree. For example, the tree structure of Table A.2 is illustrated by Figure 2.2.

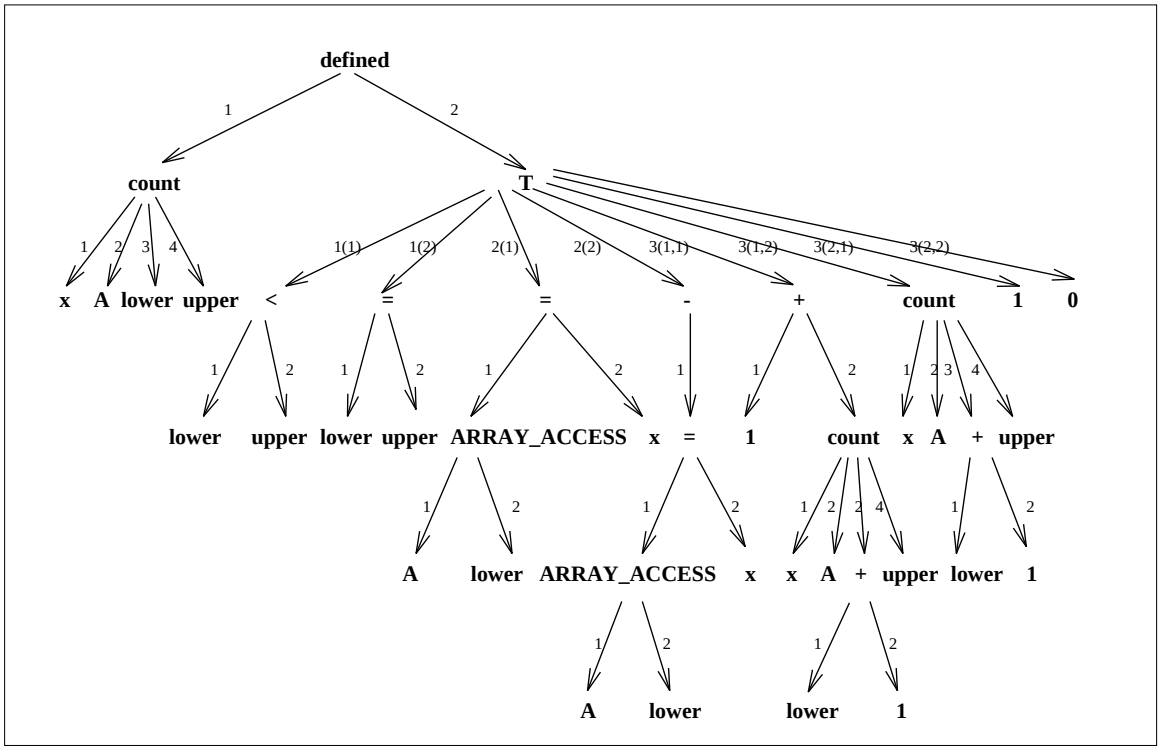


Figure 2.2: The Tree Structure of Table A.2

In Figure 2.2, symbol "defined" is the root node of the expression. The first sub-tree of the root represents the sub-expression "count(x, A, lower, upper)", which is the first argument of the symbol "defined". The symbol "x" in the first row and the first column of main grid G (see Table A.2) is connected to the root node by the path (2, 3(1, 1), 2, 1), which is a sequence of four elements separated by ".". The first element of the path 2 means that "x" is in the second sub-tree (table "T") of the root; the second element of the path 3(1, 1) means the position of "x" in the table "T" is the first row and the first column of grid 3; the third element of the path 2

means that "x" is in the second sub-tree of its grandparent "+"; the last element of the path 1 means that "x" is in the first sub-tree of its parent "count".

The Table Holder contains five modules: Expn, Index, Path, Shape, and TH_Error. The following are some of the definitions used in the Table Holder.

Expn is the data type implemented by the Expn module. An Expn object represents the tree structure of an expression. Expressions can be grouped into three different categories: atoms (constants or variables), applications (functions with one or more arguments), and tables.

Index is the data type implemented by the Index module. An Index object refers to a particular element of a table by indicating in which cell it is located. Note: Index is a TTS data type. It is NOT the set of indices generated by DIT, which is used for software documentations indicating in which page and expression a particular symbol appears as well as the row and column number if applicable (see Section 3.2.2).

Path is the data type implemented by the Path module. A Path object can be interpreted as the direction (starting at the "root") for finding a particular sub-expression.

Shape is the data type implemented by the Shape module. A Shape object describes the shape of a table in terms of the number of grids, the number of dimensions of each grid, and the length of each dimension.

Etype is the data type used to indicate the intended interpretation of an expression such as constant, variable, application, and table.

Id is the data type used to identify the symbols of an expression. Every node of an expression tree is a symbol. Each symbol is assigned an Id as its identifier.

arity refers to the number of arguments of a function.

Symbol Information

Symbol Information refers to a set of modules that associates each symbol Id with the information about that symbol such as name, arity, etc. Two modules are included: Info and Info_Error. The following are some of important definitions.

SymTbl is a data type of the Info module representing a symbol table. A symbol table contains information about the symbols used in the expressions such as name, arity, etc.

Default symbols The most commonly used symbols such as "+", "-", "∩", etc. are considered as default symbols in TTS. A default symbol table (default.sym) contains those symbols with their corresponding information.

2.2.2 TTS Infrastructure

The TTS Infrastructure is a collection of modules that provide general purpose programs for constructing TTS applications (see Section 2.2.3). It includes three parts: Tools, Utilities, and Tool Integration Framework. The following are some of the definitions used in the TTS Infrastructure.

CHandle is a data type representing a context. A context is a collection of expressions that the user wishes to treat as a group.

Context file is a file for storing a context. It contains a set of expressions together with a symbol table. The extension of a context file name is "tts".

Line is the data type implemented by the Line utility module. A Line object represents a sequence of arbitrary length of text. The operations of the ADT Line [1] include inserting a text at the beginning of a Line, appending a text at the end of a Line, joining two Lines, obtaining the text contained in a Line, etc.

IdList is the data type implemented by the IdList utility module. An IdList object represents a sequence of symbol Ids. The operations of ADT IdList include adding an Id, deleting an Id, retrieving the Ids in an IdList, etc.

CIndex is the data type implemented by the CIndex utility module. A CIndex object contains the indexing information about a context (e.g. how many times a particular variable has been used in the context).

[1] Abstract Data Type

Cellholder is the data type implemented by the TLT_cellholder module (see Section 4.2.4). A Cellholder object associates the generated code (e.g. LaTeX code) with each cell of a table.

ExpnInfo is the data type implemented by the TLT_expninfo module (see Section 4.2.4). An ExpnInfo object associates each expression with the information (e.g. symbol table) needed for converting the expression to a format language (e.g. LaTeX).

Layout is the data type implemented by the TLT_layout module (see Section 4.2.4). A Layout object represents the parameters that specify the layout of a table such as table width, column width, header position, etc.

Indexing mode refers to the boolean variable that indicates if indices are required for an expression.

Representation mode refers to the boolean variable that indicates if an expression is required to be represented as a single document.

Split mode refers to the boolean variable that indicates if a table need to be split into two or more sub-tables.

Context Manager is a tool for manipulating contexts such as creating, destroying, saving, etc.

Table Construction Tool is a tool for constructing and editing tabular expressions.

Slicing Tool is a tool for extracting slices of tabular expressions to reduce the number of dimensions of the tables. For example, taking a slice of a three-dimensional table results in a two-dimensional table.

2.2.3 TTS Applications

The TTS Applications operate at the documentation level to allow the user to edit, analyze or interpret documentations. For example, DIT is a TTS application for generating a set of indices for a software documentation.

2.3 LaTeX

LaTeX is a document preparation system that enables users to typeset and print their work with high typographical quality, using a predefined professional layout. LaTeX is built on top of TeX which is a powerful text formatter designed for high-quality typesetting. Both LaTeX and TeX are macro-based format systems [2]. TeX uses about 600 macro definitions as commands to allow the user to specify the format of the text. LaTeX uses TeX macros to form a set of new macros that are more convenient to use. As a special version of TeX, LaTeX is particularly suitable for long articles and books due to its facilities of the automatic numbering of chapters, sections, equations, etc., and also cross-referencing. This section provides a brief introduction to LaTeX. For detailed information on it refer to [5, 8]. The following are some of the LaTeX terms used in this thesis:

Cross-reference is a mechanism for referring the reader to a particular part of a document such as tables, figures, etc. LaTeX provides the following commands for cross-referencing: \label{key}, \ref{key}, \pageref{key}. The key is an identifier assigned by the user. LaTeX replaces \ref{key} command by the number of the table, figure, or section where the corresponding \label{key} command was issued. \pageref{key} command prints the page number of the corresponding \label{key} command.

Environment is a type of LaTeX construction. It is defined for typesetting special text such as quotations, mathematical formulas, etc. Environments start with \begin{*name*} and end with \end{*name*}, where *name* is the name of the environment. The existing LaTeX environments such as equation, table, etc. are not suitable for typesetting the expressions [3] used in software documents. This is because typesetting expressions requires automatic numbering of both scalar expressions and tabular expressions using an independent counting system. Therefore, a new environment called "expression" has been defined for TLT. With the "expression" environment, DIT can refer the reader to a specific expression in the document using cross-references.

[2] A *macro* is a new definition of TeX command in terms of existing ones.
[3] Expressions consists of scalar expressions and tabular expressions.

Packages are the standard enhancements of LaTeX for additional features (e.g. LaTeX document to HTML conversion). Packages are activated with the command \usepackage. For example, *longtable* is a package used by TLT to handle tables that cross more than one page.

Modes TeX is the engine of LaTeX. As TeX processes the input text, it is always in one of three modes: a paragraph mode, LR mode (left-to-right mode), or math mode. The paragraph mode is the normal mode of TeX. It is for the ordinary text. The LR mode is similar to the paragraph mode, except no line breaking is allowed in it. The command for starting the LR mode is \mbox{}. The math mode is for mathematical formulas. The commands for starting the math mode are $, \(, etc.

2.3.1 Structure of LaTeX Documents

LaTeX documents usually consist of two main parts, the preamble and the body. The preamble consists of commands that influence the style of the whole document, or packages that add new features to the LaTeX system. The body contains the actual content (text) of the document and text formatting commands. Every LaTeX document begins with an "\documentclass" command and end with an "\end{document}" command. The preamble goes between the "\documentclass" and the "\begin{document}" commands. The body is sandwiched between "\begin{document}" and "\end{document}" commands. The structure of LaTeX documents (version LaTeX2ε) is shown in Figure 2.3.

2.3.2 Process of Creating LaTeX Documents

Typically, creating LaTeX documents involves the following four steps. Figure 2.4 illustrates the process of creating a LaTeX document (sample.tex).

1. Prepare an ASCII document (.tex file) as the input for LaTeX. The input file contains the text of the document as well as the commands which tell LaTeX how to typeset the text.

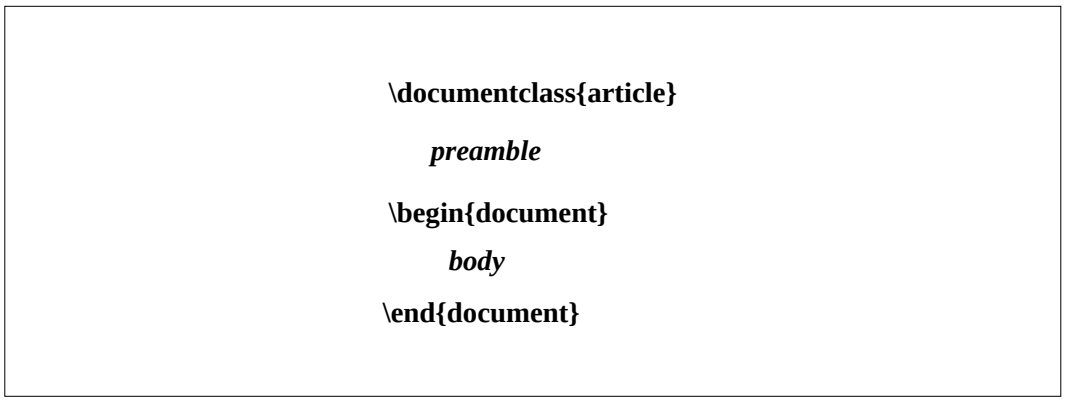


Figure 2.3: Structure of LaTeX Documents

2. Compile the input file (.tex file) using *latex* to produce a "dvi" (device-independent) file that can be displayed with XWindows previewer *xdvi*.

3. Convert the .dvi file to a postscript file (.ps file) using *dvips*. Postscript files can also be displayed on the screen using *ghostview*.

4. Print the .ps file with *lpr* command to get a printed document.

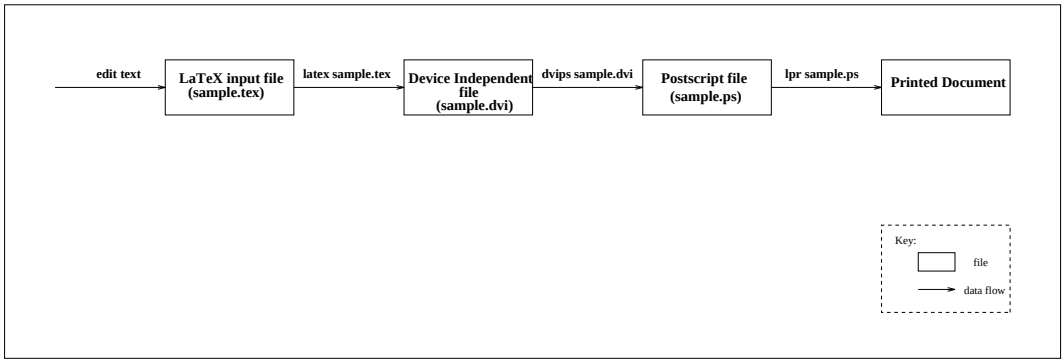


Figure 2.4: The Process of Creating a LaTeX Document

Chapter 3

Table LaTeX Tool and Document Indexing Tool

This chapter describes TLT and DIT in general, including the overview, features and user's guide.

3.1 Overview of TLT and DIT

TLT and DIT have been developed for generating indexed formal software documents. To produce such a document, the expressions have to be converted into LaTeX using TLT, and a set of indices are then created using DIT. To ensure the correctness of the indices, DIT must be used along with TLT and the outputs of TLT and DIT have to be included in a same document. Since TLT and DIT are two closely related tools, it is necessary to provide an overview of them together. Detailed descriptions on the design of TLT and DIT are presented in Chapter 4 and Chapter 5 respectively.

From a user's perspective, the process of creating an indexed formal software document involves the following four steps: 1) prepare text and diagrams using LaTeX, while using TTS (e.g. Table Construction Tool) to produce tables; 2) produce LaTeX representations of tables that can be inserted in the document using TLT; 3) create a set of indices indicating the table, row, column and page number where each symbol appears using DIT; 4) put the text, diagrams and the outputs of TLT and DIT together to generate the indexed document. Examples of the output of TLT and DIT

are provided in Appendix A and Appendix B.

From a designer's perspective, the process of generating an indexed formal software document using TLT and DIT includes the following three steps: 1) tabular expressions are created using the Table Construction Tool and stored in the Table Holder; 2) TLT takes a single expression from the Table Holder, and produces the LaTeX representation for it. DIT accepts a context file (see Section 2.2.2) which contains a set of expressions together with a symbol table, and generates a set of indices for it; 3) the outputs of TLT and DIT are then put together to produce the indexed document. This process is illustrated in Figure 3.1.

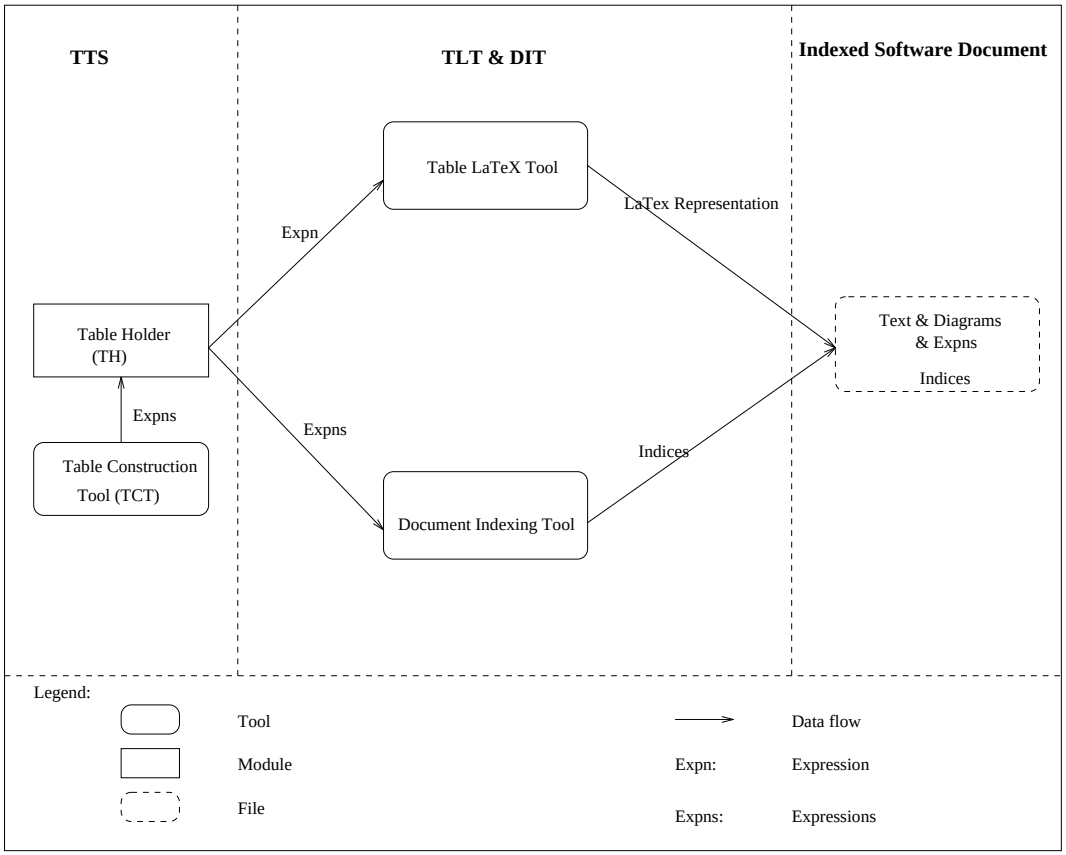


Figure 3.1: The Process of Generating an Indexed Formal Software Document Using TLT and DIT

3.2 Features

3.2.1 Features of TLT

The features of TLT are listed as follows:

1. TLT can handle both tabular expressions and scalar expressions as defined in Section 2.1.

2. TLT can handle both one- and two-dimensional tables. For a table of higher dimension, we can split it into a set of two-dimensional tables (slices) using the Slicing Tool of TTS (see Section 2.2.2) and then use TLT to convert those two-dimensional tables (slices) into LaTeX.

3. Tabular documentations often contain complicated tables such as nested tables (tables inside table), wide tables (total column widths greater than the declared table width), long tables (multipage tables). TLT provides a very simple way to assist users with these kinds of tables. The algorithm of TLT is general and sufficient enough for handling nested tables and long tables. For a wide table, TLT automatically splits it into a set of sub-tables and produces the LaTeX representation for each sub-table.

4. The output of TLT can be viewed/printed as a single document or inserted into a document. With a separate output file for each expression, users are able to place the expression where they want in the document.

5. The output of TLT can also be easily converted to a postscript file, which can be included in the documents prepared by other systems such as FrameMaker.

6. TLT helps users prepare nice-looking tables by allowing them to customize the layout of the tables.

3.2.2 Features of DIT

The features of DIT are listed as follows:

1. Two different types of indices can be produced by DIT, namely a definition-index and a use-index. A *definition-index* indicates in which expression and page a particular symbol or sub-expression (e.g. count(x, A, lower, upper)) is defined. Refer to Section 2.2.1 for the definition of sub-expression. A *use-index* indicates in which expression a specific symbol (e.g. lower) appears and the row, column, page number. DIT produces an index entry for each occurrence of a symbol in the expressions, even if the occurrences are identical in terms of the same expression, row, column and page number.

2. DIT is able to generate both selected indices and complete indices. A *selected index* is an index generated for a specific type of symbols in order to reduce the size of the index. The selected index can be one of: constant index, variable index, application index. For example, if users are only interested in the variables of a documentation, they can generate a variable index. A *complete index* is an index generated for all types of symbols including variable, constant and application symbols.

3. With a set of indices generated by DIT, users can search for information easily and efficiently and thus be able to understand the document better. DIT significantly improves the readability and usability of tabular documentations.

3.3 User's Guide

This section provides the user's guide for TLT and DIT. To ensure the correctness of the indices, DIT must be used together with TLT.

3.3.1 TLT User's Guide

TLT is a tool for generating LaTeX representations of expressions. It takes a single expression as input and produces LaTeX representation of the expression as output. The following two support systems, as described in Chapter 2, are required for operating TLT:

- LaTeX as a document preparation system (version LaTeX2ε).
- TTS Kernel and TTS Infrastructure for creating, storing, manipulating expressions.

User Interface

The user interface of TLT consists of the following four parts: Command Line Arguments, Expression Selection Menu, Expression Option Menu, and Table Layout Menu.

1. Command Line Arguments

TLT provides a command line interface based on the following syntax:
tlt *contextFile outFile*

Where:

contextFile	the context file name with "tts" as extension.
outFile	the LaTeX output file name with "tex" as extension.

2. Expression Selection Menu

The Expression Selection Menu is provided for selecting an expression to be converted. It lists all the expressions in the context file by their names. The following is the Expression Selection Menu.

```
********** Expression Selection Menu **********
1. The name of Expn1
2. The name of Expn2
3. The name of Expn3
.
.
.
N. The name of ExpnN

To select an expression, enter 1-N.
```

3. Expression Option Menu

The Expression Option Menu is provided for setting up the options associated with each expression. Two options are provided for indicating: 1) if indices are required for the expression; 2) if the expression needs to be presented as a single document for viewing or printing. The expected answer is either "Y" or "N". The following is the Expression Option Menu together with the explanation on how to use it.

```
********** Expression Option Menu **********
1. Create indices: Y
2. Represented as a single document: Y
3. Done.



To change, enter 1-2.
To exit, enter 3.
```

Options:

1 Set the indexing mode (see Section 2.2.2) for the expression. If indices are required for the expression, the value is "Y". Otherwise, it is "N". The default value is "Y".

2 Set the representation mode (see Section 2.2.2) for the expression. If the expression needs to be represented as a single document, the value is "Y", Otherwise, it's "N". The default is "Y".

3 Exit from the menu.

Note:

- Once the value of an option is updated, the Expression Option Menu will be refreshed with the updated information.

4. Table Layout Menu

The Table Layout Menu is provided for setting up layout parameters for a table, including the table width, column widths, header positions, and caption. The following is the Table Layout Menu together with the explanation on how to use it.

********** Table Layout Menu **********
1. Table width (mm): 120 2. Column widths (mm): 30, 30, 30, 30 3. Header positions: L, U 4. Caption: 5. Done   To update the layout parameters, enter 1-4 To exit, press 5

Options:

1 Set the total table width. The maximum table width is 150mm and the default is 120mm.

2 Set the column widths. Different columns can have different column widths. For example, "30, 40, 50" indicates that the table has 3 columns and the width of the first, second and third column is 30mm, 40mm, 50mm respectively. TLT automatically determines how many columns are in the table and calculates the default column width using formula $columnwidth = tablewidth \div numberofcolumns$. For example, if a table has 4 columns, its default column widths are "30, 30, 30, 30", since the default table width is 120. If the sum of the column widths is greater than the declared table width, TLT prints the following statement on the screen: "Table is too wide. Type (Y/N) to indicate if the table needs to be split." If the answer is "Y", the table will be split into several sub-tables and the LaTeX representation of each sub-table will be generated. Otherwise, the table will not be split and the LaTeX representation of the table will be generated, regardless how wide it is.

3 Set the header positions. Header 1 can be either left ("L") or right ("R"). Header 2 can be either up ("U") or down ("D"). The default position for Header 1 and Header 2 is "L" and "U" respectively.

4 Set the caption of the table. The default is an empty string.

5 Exit the menu.

Notes:

- Once the value of a parameter is updated, the Table Layout Menu will be refreshed with the updated information.
- If the expression contains more than one table, the Table Layout Menu will be displayed more than once to enable the user to set up the layout for each table.

3.3.2 DIT User's Guide

DIT produces a set of indices for expressions. The input of DIT is a context file that contains a set of expressions and a symbol table. The output of DIT is a set of indices indicating the expression, row, column and page number where each symbol appears. DIT requires the same support systems as TLT.

User Interface

The user interface of DIT includes the following two parts: Command Line Arguments and Index Option Menu.

1. Command Line Arguments

DIT provides a command line interface based on the following syntax:
dit *contextFile outFile*

Where:

contextFile	the context file name with "tts" as extension.
outFile	the index file name with "tex" as extension.

2. Index Option Menu

The Index Option Menu is provided for choosing the type of index that needs to be created and the type of symbols that needs to be indexed. The following is the Index Option Menu together with the explanation on how to use it.

```
********** Index Option Menu **********
1. Index type: U
2. Symbol type: T
3. Done



To change, enter 1-2.
To exit, enter 3.
```

Options:

1 Set the index type for the expression(s). The index type indicates the type of index that needs to be created. It can be either U (use-index) or D (definition-index), as described in Section 3.2.2. The default is U.

2 Set the symbol type. The symbol type indicates the type of symbols that need to be indexed. It can be one of C, V, A or T (constant, variable, application, total), as described in Section 3.2.2. The default is T.

3 Exit the menu.

Chapter 4

Table LaTeX Tool Design

This chapter discusses several topics related to the design of TLT including requirements, module decomposition, module guide, module interface specifications, and module internal design.

4.1 Requirements

The Table LaTeX Tool (TLT) produces LaTeX representations of expressions that can be viewed/printed as a single document or inserted as part of a document. It accepts a single expression from the Table Holder as input, and generates a LaTeX file as output.

4.1.1 Anticipated Changes

It is anticipated that the following items are likely to change during the life time of TLT:

- LaTeX as a format language
- Layout parameters used for formatting tabular expressions
- User interface

4.2 Module Decomposition

To make the design easy to understand and modify, TLT is decomposed into a set of modules using the "information hiding" principle [9].

4.2.1 Module-Uses Hierarchy

The module-uses relation is used to illustrate the system design of TLT. Module A is said to use Module B if at least one access program in Module A relies on the correct behavior of some programs in Module B to accomplish its task [10]. The top-level module-uses relation of TLT is shown in Figure 4.1.

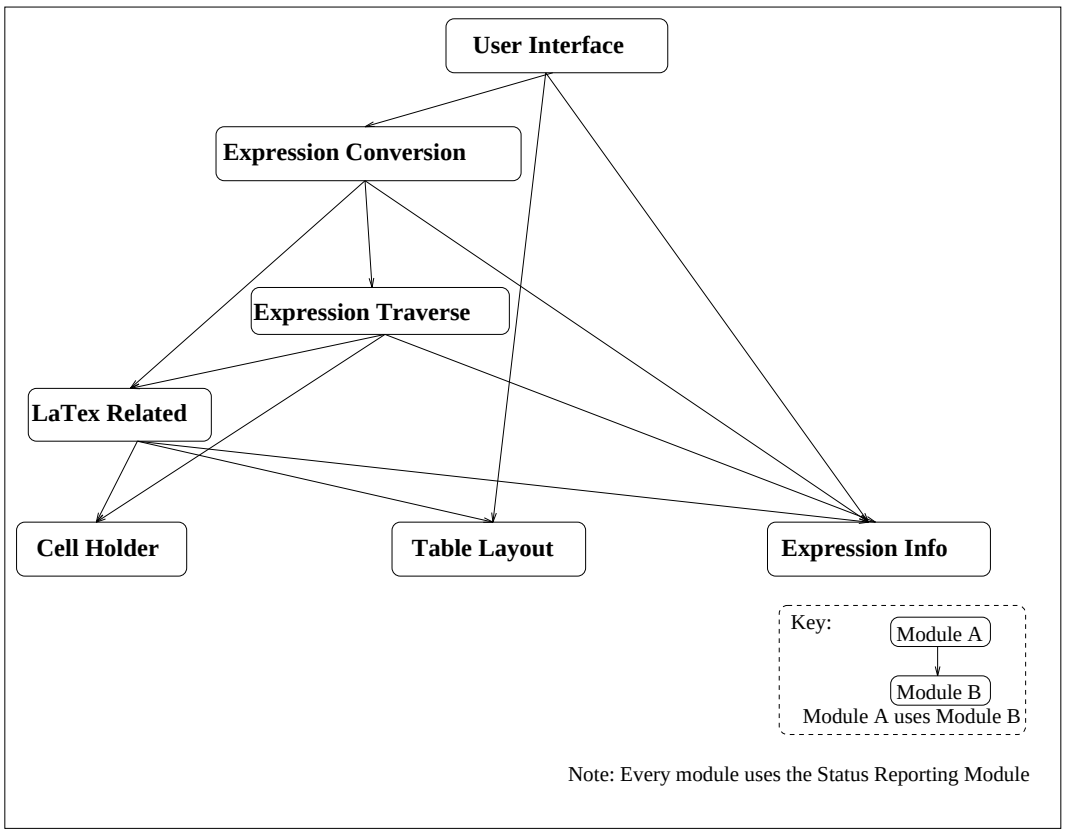
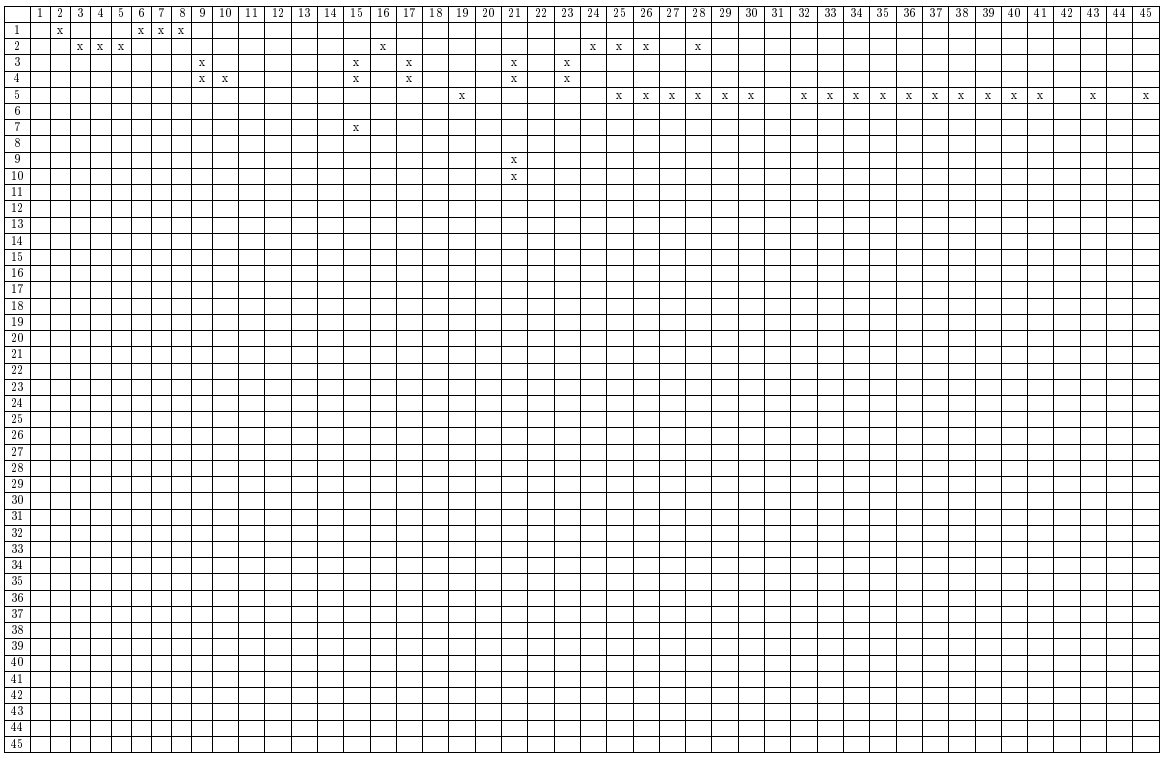


Figure 4.1: TLT Top Level Module-Uses Relation

4.2.2 Program-Uses Hierarchy

The program-uses relation is used to illustrate the detailed design of TLT in terms of the uses relation between the access programs. Program A uses Program B if there exist situations in which the correct functioning of Program A depends on the correct behavior of Program B [11].

The program-uses relation of TLT is shown as a matrix in Figure 4.2. Each program of TLT is identified by an integer (i.e. the program id). Table 4.1 provides the mapping between the program ids and the program names. "x" in the matrix indicates that the program located in the corresponding row uses the program located in the corresponding column.



Note: Every program can use any access program of TLT_error module.

Figure 4.2: Program-Uses Relation of TLT

Table 4.1: Access Programs of TLT

Id	Name	Id	Name
1	TLT_convertExpn	2	TLT_expn
3	TLT_codeAtom	4	TLT_codeApplic
5	TLT_codeTable	6	TLT_codeMathText
7	TLT_codeExpnLabel	8	TLT_codeDocument
9	TLT_infoGetName	10	TLT_infoGetForm
11	ExpnInfo_init	12	ExpnInfo_create
13	ExpnInfo_destroy	14	ExpnInfo_setExpn
15	ExpnInfo_getExpn	16	ExpnInfo_setPath
17	ExpnInfo_getPath	18	ExpnInfo_setLayout
19	ExpnInfo_getLayout	20	ExpnInfo_setSymTbl
21	ExpnInfo_getSymTbl	22	ExpnInfo_setIndexMode
23	ExpnInfo_getIndexMode	24	Cellholder_init
25	Cellholder_create	26	Cellholder_destroy
27	Cellholder_getNumGrids	28	Cellholder_addCell
29	Cellholder_getCell	30	Cellholder_isGridEmpty
31	Layout_init	32	Layout_create
33	Layout_destroy	34	Layout_setTableWidth
35	Layout_getTableWidth	36	Layout_setNumColumns
37	Layout_getNumColumns	38	Layout_setColumnWidth
39	Layout_getColumnWidth	40	Layout_setHeaderPos
41	Layout_getHeaderPos	42	Layout_setCaption
43	Layout_getCaption	44	Layout_setSplitMode
45	Layout_getSplitMode		

4.2.3 Module Guide

The module Guide is an informal document that states the secret encapsulated in each module [16].

Expression Conversion Module

The secret of this module is the parallel processsing used in expression conversion. It hides how many activities are ongoing at once.

Expression Traverse Module

The secret of this module is the algorithm to traverse the expressions stored in the Table Holder.

LaTeX Related Module

The secret of this module is LaTeX. It can be divided into the following two submodules:

LaTeX Code Module
The secret of this module is what LaTeX text will be produced for each expression.

LaTeX Info Module
The secret of this module is how the info module is used to store the LaTeX name for each atom/application symbol and the LaTeX form for each application symbol.

Table Layout Module

The secret of this module is the representation of the parameters that specify the layout of a table such as table width, column width, header position, caption, etc.

Cell Holder Module

The secret of this module is the data structure used to associate the generated code (e.g. LaTeX code) with each cell of a table.

Expression Info Module

The secret of this module is the data structure used to store information associated with each expression such as symbol table, indexing mode, etc.

Status Reporting Module

The secret of this module is the representation of the operating statuses such as TLT_Success, TLT_THErr, etc., of the access programs. Refer to Table 4.10 for the interpretation of each TLT status token.

4.2.4 Module Interface Specifications (Informal)

A module interface specification treats the module as a black-box, identifying the access programs of the module and describing the externally visible effects of using them [16]. Each access program of TLT is described informally by indicating the types of arguments and returns, and a brief explanation of how to use it.

Expression Conversion Module (TLT_convert.c)

The Expression Conversion Module converts an expression into a format language (e.g. LaTeX) and outputs the code into a file. The access program of this module is described in Table 4.2.

Table 4.2: Expression Conversion Module Access Program

Access Program	Return	Arguments	Description
TLT_convertExpn	void	Expn e Path p bool m ExpnInfo i FILE* f	Converts the expression *e* located at the path *p* to the chosen format language (e.g. LaTeX) and outputs the code into the file *f*. Representation mode *m* is BOOL_TRUE when the expression need to be represented as a single document for viewing/printing. Otherwise it is BOOL_FALSE. ExpnInfo *i* is used to store/retrieve the information associated with the expression such as indexing mode.

Expression Traverse Module (TLT_expn.c)

The Expression Traverse Module traverses the expressions stored in the Table Holder. The access program of this module is described in Table 4.3.

Table 4.3: Expression Traverse Module Access Program

Access Program	Return	Arguments	Description
TLT_expn	Line	Expn e Path p ExpnInfo i Line buf	Traverses the sub-expression *e* located at path *p* and appends the generated code to the Line object *buf*. Returns a Line object containing the code text of the expression. ExpnInfo *i* is used to store/retrieve information associated with the expression such as indexing mode.

Expression Info Module (TLT_expninfo.c)

The Expression Info Module contains information associated with an expression such as symbol table, indexing mode, layout, etc. Table 4.4 is the description of its access programs.

Table 4.4: Expression Info Module Access Programs

Access Program	Return	Arguments	Description
ExpnInfo_init			Initializes the module to an empty state.
ExpnInfo_create	ExpnInfo		Creates a new ExpnInfo object.
ExpnInfo_destroy		ExpnInfo i	Destroys the ExpnInfo object *i*.

Table 4.4: Expression Info Module Access Programs

Access Program	Return	Arguments	Description
ExpnInfo_setExpn		ExpnInfo i Expn e	Sets the Expn object of the ExpnInfo i to be e.
ExpnInfo_getExpn	Expn	ExpnInfo i	Returns the Expn object of the ExpnInfo i.
ExpnInfo_setPath		ExpnInfo i Path p	Sets the Path object of the ExpnInfo i to be p.
ExpnInfo_getPath	Path	ExpnInfo i	Returns the Path object of the ExpnInfo i.
ExpnInfo_setLayout		ExpnInfo i Layout l	Sets the Layout object of the ExpnInfo i to be l.
ExpnInfo_getLayout	Layout	ExpnInfo i	Returns the Layout object of the ExpnInfo i.
ExpnInfo_setSymTbl		ExpnInfo i SymTbl t	Sets the SymTbl object (symbol table) of the ExpnInfo i to be t.
ExpnInfo_getSymtbl	SymTbl	ExpnInfo i	Returns the SymTbl object (symbol table) of the ExpnInfo i.
ExpnInfo_setIndexMode		ExpnInfo i bool m	Sets the indexing mode of the ExpnInfo i to be m.
ExpnInfo_getIndexMode	bool	ExpnInfo i	Returns the indexing mode of the ExpnInfo i.

LaTeX Code Module (TLT_code.c)

The LaTeX Code Module is responsible for generating LaTeX code for an expression. Table 4.5 is the description of its access programs.

Table 4.5: LaTeX Code Module Access Programs

Access Program	Return	Arguments	Description
TLT_codeAtom	char*	Id id ExpnInfo i	Returns code for the atom *id*. The ExpnInfo *i* is used to retrieve information associated with the atom expression such as indexing mode, symbol table, etc.
TLT_codeApplic	Line	Id id int arity Line args[] ExpnInfo i	Constructs code for the application *id* with *arity* number of arguments stored in the array of Line object *args*. Returns a Line object containing the code for the application expression. The ExpnInfo *i* is used to retrieve information associated with the application expression such as indexing mode, symbol table, etc.
TLT_codeTable	Line	Cellholder ch ExpnInfo i	Constructs code for the tabular expression where code for each cell of the table is stored in the Cellholder *ch*. Returns a Line object containing the code for the tabular expression. The ExpnInfo *i* is used to retrieve information associated with the tabular expression such as layout.

Table 4.5: LaTeX Code Module Access Programs

Access Program	Return	Arguments	Description
TLT_codeMathText	Line	Line buf	Returns code for adding math text (e.g. "$" in LaTeX) to the expression that is represented by the code contained in the Line object *buf*
TLT_codeExpnLabel	Line	Line buf ExpnInfo i	Returns code for adding a label (e.g. \label{key} in LaTeX) to the expression that is represented by the code contained in the Line object *buf*. The ExpnInfo *i* is used to retrieve information associated with the expression such as Expn object.
TLT_codeDocument	Line	Line buf	Returns code for a document that includes the text contained in the Line object *buf*.

LaTeX Info Module (TLT_info.c)

The LaTeX Info Module retrieves LaTeX name for atom/application symbols and LaTeX form for application symbols from the info module. The LaTeX form of an application is an array of arity + 1 strings used together with the code of its argument expressions to construct the code for the application expression. Table 4.6 describes the access programs of this module.

Table 4.6: LaTeX Info Module Access Programs

Access Program	Return	Arguments	Description
TLT_infoGetName	char*	SymTbl tbl Id id	Returns the name of the symbol *id* in a specific format language (e.g. LaTeX) from the symbol table *tbl*.
TLT_infoGetForm	char*	SymTbl tbl Id id int num char* form	Returns the num^{th} form of the symbol *id* in a specific format language (e.g. LaTeX) from the symbol table *tbl*.

Cell Holder Module (TLT_cellholder.c)

The Cell Holder Module associates the generated code with each cell of a table. Table 4.7 is the description of its access programs.

Table 4.7: Cell Holder Module Access Programs

Access Program	Return	Arguments	Description
Cellholder_init			Initializes the module to an empty state.
Cellholder_create	Cellholder	int num_grids	Creates an new Cellholder object for a table with the number of grids being *num_grids*.

Table 4.7: Cell Holder Module Access Programs

Access Program	Return	Arguments	Description
Cellholder_destroy		Cellholder ch	Destroys the Cellholder object *ch*.
Cellholder_getNumGrids	int	Cellholder ch	Returns the number of the grids of the Cellholder *ch*.
Cellholder_addCell		Cellholder ch int g Line buf	Adds the Line *buf* to the grid *g* of the Cellholder *ch*. Line *buf* contains the code text generated for a cell of the table.
Cellholder_getCell	Line	Cellholder ch int g	Returns a Line object from the grid *g* of the Cellholder *ch*.
Cellholder_isGridEmpty	bool	Cellholder ch int g	Returns BOOL_TRUE if the grid *g* of the Cellholder *ch* is empty. Otherwise, returns BOOL_FALSE.

Table Layout Module (TLT_layout.c)

The Table Layout Module stores the layout parameters for a tabular expression such as table width, column width, caption, etc. Table 4.8 is the description of the access programs.

Table 4.8: Layout Module Access Programs

Access Program	Return	Arguments	Description
Layout_init			Initializes the module to an empty state.
Layout_create	Layout		Creates a new Layout object.
Layout_destroy		Layout l	Destroys a Layout object *l*.
Layout_setTableWidth		Layout l int w	Sets the table width of the Layout *l* to be *w*. The unit is "mm".
Layout_getTableWidth	int	Layout l	Returns the table width of the Layout *l*.
Layout_setNumColumns		Layout l int c	Sets the number of columns of the Layout *l* to be *c*.
Layout_getNumColumns	int	Layout l	Returns the number of columns of the Layout *l*.
Layout_setColumnWidth		Layout l int num int w	Sets the width of the num^{th} column of the Layout *l* to be *w*. The unit is "mm".
Layout_getColumnWidth	int	Layout l int num	Returns the width of the num^{th} column of the Layout *l*.
Layout_setHeaderPos		Layout l int h int pos	Sets the position of the header *h* of the Layout *l* to be *pos*. *h* is 1 for H2 to be up, 2 for H2 to be down, 3 for H1 to be left, 4 for H1 to be right.

Table 4.8: Layout Module Access Programs

Access Program	Return	Arguments	Description
Layout_getHeaderPos	int	Layout l int h	Returns the position of the header *h* of the Layout *l*.
Layout_setCaption		Layout l char* c	Sets the caption of the Layout *l* to be *c*.
Layout_getCaption	char*	Layout l	Returns the caption of the Layout *l*.
Layout_setSplitMode		Layout l bool m	Sets the split mode of the Layout *l* to be *m*. m is BOOL_TRUE if the table need to be split into sub-tables. Otherwise it's BOOL_FALSE.
Layout_getSplitMode	bool	Layout l	Returns the split mode of the Layout *l*.

Status Reporting Module (TLT_error.c)

The Status Reporting Module provides a means for the access programs of the other modules to indicate the operating statuses to their callers. If a failure occurs, the general reason for it will be provided. Table 4.9 is the description of the access programs of this module. The interpretation of each TLT status token is presented in Table 4.10.

Table 4.9: Status Reporting Module Access Programs

Access Program	Return	Arguments	Description
SetErrTLT		TLT_Token t	Sets the status token to *t*.
GetErrTLT	TLT_Token		Returns the current status token.
GetStrErrTLT	char*	TLT_Token t	Returns a string describing the status token *t*.

Table 4.10: TLT Status Tokens

Status Token	Interpretation
TLT_Success	No error.
TLT_ModNotInit	Module has not been initialized yet.
TLT_ModAlreadyInit	Module has been initialized.
TLT_AllocFail	Memory allocation failed.
TLT_BadHandle	Current object is invalid.
TLT_Excess	No more objects can be created.
TLT_BadTableWidth	Table width is invalid.
TLT_BadNumColumns	The number of columns is invalid.
TLT_BadColumnWidth	Invalid column width.
TLT_BadColumnNum	Invalid column number.
TLT_BadHeader	Invalid header.
TLT_BadHeaderPos	Invalid header position.
TLT_BadCaption	Invalid caption.
TLT_BadNumGrids	The number of grids is invalid.

Table 4.10: TLT Status Tokens

Status Token	Interpretation
TLT_BadGridNum	Invalid grid number.
TLT_EmptyGrid	No more code is stored for the grid.
TLT_EmptyLayoutList	No more Layout object is stored.
TLT_BadExpn	Invalid Expn object.
TLT_BadPath	Invalid Path object.
TLT_BadLayout	Invalid Layout object.
TLT_BadSymTbl	Invalid symbol table.
TLT_BadLine	Invalid Line object.
TLT_InfoErr	Error in Info module.
TLT_THErr	Error in TH module.
TLT_LineErr	Error in Line module.
TLT_InvalidPos	Invalid position.
TLT_InvalidShape	Invalid table shape.
TLT_TableExcess	Current table has been split into too many sub-tables.

4.3 Algorithm

The following is the algorithm used by the TLT to produce LaTeX representation of an expression:

step 1: Load a context file and extract the expression from it.

step 2: Traverse the expression depth-first.

step 3: While traversing the sub-expressions, different procedures are taken to generate LaTeX code for different types of sub-expressions.

- Atom

1. Generate LaTeX code for the Atom.

- Application
  1. Traverse the application expression depth-first.
  2. Go to step 3 to generate LaTeX code for each argument of the application symbol.
  3. Generate LaTeX code for the application symbol if code for all the arguments has been generated.
  4. Construct the application expression according to the form of the application symbol.
- Table
  1. Traverse the table depth-first.
  2. Go back to step 3 to generate LaTeX code for each cell of the table.
  3. Constructs the table based on the shape of the table and the layout specified by the user, if code for all the cells of the table has been generated.

step 4: Write LaTeX code into the output file.

Chapter 5

Document Indexing Tool Design

This chapter discusses several topics related to the design of DIT including the requirements, module decomposition, module guide, module interface specifications, and module internal design.

5.1 Requirements

The Document Indexing Tool (DIT) creates a set of indices for a formal software document. The input to this tool is a context file, which contains a set of expressions together with a symbol table. The output is a set of indices, each of which is either a definition-index or a use-index. A definition-index indicates in which expression and page a specific symbol or sub-expression is defined. A use-index indicates in which expression and page a specific symbol is used and the row, column numbers.

5.1.1 Anticipated Changes

The following items are anticipated to be changed during the life time of DIT:

- Representation and appearance of the indices
- User interface

5.2 Module Decomposition

5.2.1 Module-Uses Hierarchy

The module-uses relation, as described in 4.2.1, is used to illustrate the module design of DIT. Figure 5.1 shows the top level module-uses relation of DIT. The CIndex module is anticipated to be used by other TTS tools (e.g. Table Construction Tool). Therefore, it is now part of the TTS utility module for general use. (see Section 2.2.2).

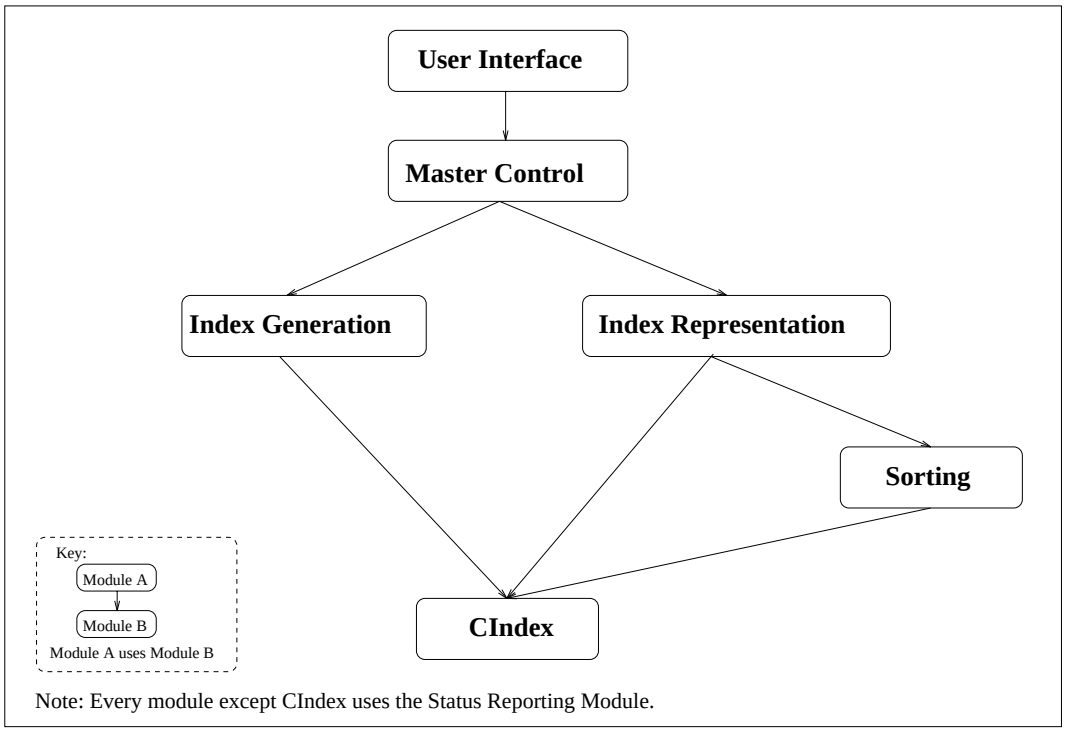


Figure 5.1: Top Level Module-Uses Relation of DIT

5.2.2 Program-Uses Hierarchy

The program-uses relation, as described in 4.2.2, is used to illustrate the detailed design of DIT in terms of the uses relation between the access programs. The program-uses relation of DIT is shown as a matrix in Figure 5.2.2. Each program of DIT is identified by an integer (i.e. the program id). Table 5.1 provides the mapping

between the program ids and the program names. "x" in the matrix indicates that the program located in the corresponding row uses the program located in the corresponding column.

Table 5.1: Access Programs of DIT

Id	Name	Id	Name
1	DIT_main	2	DIT_useIndex
3	DIT_defIndex	4	DIT_useRepresent
5	DIT_defRepresent	6	DIT_sort
7	CIndexInit	8	CIndexCreate
9	CIndexDestroy	10	CIndexAddExpn
11	CIndexGetExpnPos	12	CIndexGetExpn
13	CIndexDeleteExpn	14	CIndexAddLoc
15	CIndexGetLocPos	16	CIndexGetLoc
17	CIndexDeleteLoc	18	CIndexGetIdList
19	CIndexGetNumExpn	20	CIndexGetNumLoc

5.2.3 Module Guide

Master Control Module

The secret of this module is the control sequence of the other modules of DIT.

Index Generation Module

The secret of this module is the algorithm used to generate an index with information that can be obtained from the Table Holder. For instance, in which expression (Expn) the symbol appears and what are the locations (paths) of the symbol in the expression.

Index Representation Module

The secret of this module is the representation of an index in terms of the format

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1		x	x	x	x															
2								x						x						
3								x						x						
4						x			x			x				x			x	x
5						x			x			x				x			x	x
6												x						x	x	x
7																				
8																				
9													x							
10											x									
11																				
12																				
13																	x			x
14										x	x				x					
15											x									
16											x									
17											x									
18																				
19																				
20											x									

Note: Every program can use any access program of DIT_error module.

Figure 5.2: Program-Uses Relation of DIT

language that is chosen and the format that is used. For example, we can choose different format languages such as LaTeX, TeX, HTML, etc, to format an index. With the chosen format language, we can also use different formats to represent the index (e.g. array or table in LaTeX). This module also hides the appearance of an index.

Sorting Module

The secret of this module is the sorting algorithm and criterion. Since they usually come as a package, there is no need to separate them.

CIndex Module

The secret of this module is the data structure used to store the index information such as IdList, Expn, Path, etc, for a context file.

Status Reporting Module

The secret of this module is the representation of the operating statuses such as DIT_Success, DIT_InfoErr, etc. of the access programs. The interpretation of each DIT status token is provided in Table 5.8.

5.2.4 Module Interface Specifications (Informal)

Each access program of DIT is described informally by indicating the types of arguments and returns, and a brief explanation of how to use it.

Master Control Module (DIT_main.c)

The Master Control Module generates an index for a formal software document and outputs the generated code into a file. The access program of this module is described in Table 5.2.

Table 5.2: Master Control Module Access Program

<table>
<tr><th>Access Program</th><th>Return</th><th>Arguments</th><th>Description</th></tr>
<tr><td>DIT_main</td><td></td><td>CHandle ch
int i
int s
FILE* f</td><td>Generates an index of type i for the symbols with type s in the context ch and outputs the generated index into file f. The value of index type i is described as:
<table>
<tr><td>Value</td><td>Description</td></tr>
<tr><td>1</td><td>use-index</td></tr>
<tr><td>2</td><td>definition-index</td></tr>
</table>
The value of symbol type s is described as:
<table>
<tr><td>Value</td><td>Description</td></tr>
<tr><td>1</td><td>total</td></tr>
<tr><td>2</td><td>constant</td></tr>
<tr><td>3</td><td>variable</td></tr>
<tr><td>4</td><td>application</td></tr>
</table>
</td></tr>
</table>

Index Generation Module (DIT_index.c)

The Index Generation Module generates a basic index that contains information depending on the Table Holder (e.g. Path), and stores the information in a CIndex object. Table 5.3 is the description of its access programs.

Table 5.3: Index Generation Module Access Programs

Access Program	Return	Arguments	Description
DIT_useIndex	CIndex	CHandle ch SymTbl t int s	Returns a CIndex object showing where the symbols of type *s* appear in the context *ch*. Symbol table *t* is used to generate a list of ids for indexing.
DIT_defIndex	CIndex	CHandle ch int s	Returns a CIndex object showing where the symbols of type *s* are defined in the context *ch*.

Index Representation Module (DIT_represent.c)

The Index Representation Module represents an index in a specific format language (e.g. LaTeX). The access programs of this module is described in Table 5.4.

Table 5.4: Index Representation Module Access Programs

Access Program	Return	Arguments	Description
DIT_useRepresent	Line	CIndex ci SymTbl t int s	Extracts the index information such as Expn and Path from the CIndex *ci* and represents the use index using a chosen format language (e.g. LaTeX). Returns a Line object that contains the formatted index. Symbol table *t* is used to retrieve information about the symbols (e.g. LaTeX name). Symbol type *s* is used to indicate the type of the indexed symbols.
DIT_defRepresent	Line	CIndex ci SymTbl t int s	Extracts the index information (e.g. Expn) from the CIndex *ci* and represents the definition index using a specific format language (e.g. LaTeX). Returns a Line object that contains the formatted index. Symbol table *t* is used to retrieve information about the symbols (e.g. LaTeX name). Symbol type *s* is used to indicate the type of the indexed symbols.

Sorting Module (DIT_sort.c)

The Sorting Module sorts the index entries. Table 5.5 is the description of its access program.

Table 5.5: Sorting Module Access Program

Access Program	Return	Arguments	Description
DIT_sort	IdList	CIndex ci SymTbl t	Sorts the index entries stored in the CIndex *ci* and returns an IdList which lists the symbols in the sorted order. Symbol table *t* is used to get symbol information (e.g. name).

CIndex Module (CIndex.c)

The CIndex Module stores the index information such as IdList, Expn, etc. of a context file. The access programs of this module is described in Table 5.6.

Table 5.6: CIndex Module Access Programs

Access Program	Return	Arguments	Description
CIndexInit			Initializes the module to an empty state.
CIndexCreate	CIndex	IdList l int num	Returns a new CIndex object to store the index information of the IdList *l* in a context file which has *num* of expressions.

Table 5.6: CIndex Module Access Programs

Access Program	Return	Arguments	Description
CIndexDestroy		CIndex ci	Destroys the CIndex object *ci*.
CIndexAddExpn		CIndex ci Expn e	Adds the expression *e* to the CIndex *ci*.
CIndexGetExpnPos	int	CIndex ci Expn e	Returns the position of the Expression *e* in the CIndex *ci*.
CIndexGetExpn	Expn	CIndex ci int pos	Returns the expression located in the position *pos* of the CIndex *ci*.
CIndexDeleteExpn		CIndex ci int pos	Deletes the expression located in the position *pos* of the CIndex *ci*.
CIndexAddLoc		CIndex ci Id id Expn e Path p	Adds the Path *p* of the symbol *id* in the expression *e* to the CIndex *ci*.
CIndexGetLocPos	int	CIndex ci Id id Expn e Path p	Returns the position of Path *p* that the symbol *id* appears in the expression *e* from the CIndex *ci*.
CIndexGetLoc	Path	CIndex ci Id id Expn e int num	Returns the num^{th} Path of the symbol *id* in the expression *e* from the CIndex *ci*.

Table 5.6: CIndex Module Access Programs

Access Program	Return	Arguments	Description
CIndexDeleteLoc		CIndex ci Id id Expn e int num	Deletes the num^{th} Path of the symbol *id* in the expression *e* from the CIndex *ci*.
CIndexGetIdList	IdList	CIndex ci	Returns the IdList of the CIndex *ci*.
CIndexGetNumExpn	int	CIndex ci	Returns the number of expressions in the CIndex *ci*.
CIndexGetNumLoc	int	CIndex ci Id id Expn e	Returns the number of Path that the symbol *id* appears in the expression *e* from the CIndex *ci*.

Status Reporting Module (DIT_error.c)

The Status Reporting Module provides a means for the access programs of the other DIT modules to indicate the operating statuses to their callers. If a failure occurs, the general reason for it will be provided. Table 5.7 is the description of the access programs of this module. The interpretation of each DIT status token is presented in Table 5.8.

Table 5.7: Status Reporting Module Access Programs

Access Program	Return	Arguments	Description
SetErrDIT		DIT_Token t	Sets the status token to *t*.
GetErrDIT	DIT_Token		Returns the current status token.
GetStrErrDIT	char*	DIT_Token t	Returns a string describing the status token *t*.

Table 5.8: DIT Status Tokens

Status Token	Interpretation
DIT_Success	No error.
DIT_AllocFail	Memory allocation failed.
DIT_BadContext	Current CHandle object is invalid.
DIT_BadCIndex	Current CIndex object is invalid.
DIT_BadSymTbl	Current SymTbl object is invalid.
DIT_IdxEntryExcel	Too many index entries.
DIT_THErr	Error in TH module.
DIT_InfoErr	Error in Info module.
DIT_TLTErr	Error in TLT module.
DIT_CIErr	Error in CIndex module.
DIT_InvalidIdxType	Invalid index type.
DIT_InvalidSymType	Invalid symbol type.
DIT_InvalidShape	Invalid table shape.

5.3 Algorithm

The following is the algorithm used by DIT to produce a set of indices:

step 1: Load the context file.

step 2: Generate an index with information depending on the Table Holder. For example, in which expression (Expn) the symbol appears and what are the locations (Paths) of the symbol in the expression. Different procedures are taken for generating different types of index.

- Use-index:
  1. Generate an IdList that contains the symbols that need to be indexed.
  2. Extract the expressions from the context file.
  3. For each expression, traverse it to get the locations (Paths) of each symbol in the expression.
- Definition-index:
  1. Extract the expressions from the context file.
  2. For each expression, check if symbol "defined" is the root symbol of the expression. If true, check if the root symbol of its first sub-tree needs to be indexed.
  3. Generate an IdList that contains all the symbols that need to be indexed.

step 4: Create a CIndex object to store the generated index information.

step 5: Sort the index entries alphabetically.

step 6: Extract the index information from the CIndex in the sorted order and represent the index in a chosen format language (e.g. LaTeX).

step 7: Write the generated code into the output file.

Chapter 6

Results and Conclusions

This chapter summaries the results of the design and implementation of TLT and DIT, discusses the limitations and future work, and draws the conclusions.

6.1 Results

TLT and DIT have been successfully developed for generating indexed software documents. Using TLT, mathematical expressions can be viewed/printed as a 2 single document or as part of the documents prepared using LaTeX or other systems (see Appendix A for examples). Users can make nice-looking tables without any knowledge of the input format and the interpretation of tabular expressions. In terms of the capabilities, TLT can handle all kinds of expressions, which can be either scalar expressions or tabular expressions. For a tabular expression, it can be one- or two-dimension. Furthermore, TLT can handle complicated tables, particularly nested tables, wide tables and long tables, which are often used in practice but few tools of TTS can handle them.

The indices generated by DIT are based on the anticipated queries that users may have, and they are sufficient to satisfy the different needs of users (see Appendix B for examples). Furthermore, the generated indices are sorted and well-formatted to allow users quick and easy access to particular information. In addition to the original indexing purpose, DIT has been found useful in the following two other areas:

1. Editing tabular expressions. The indices that show the occurrences of a specific

symbol in expressions can be used by the Table Construction Tool for editing expressions. For example, a user can easily find all the occurrences of a specific symbol in an expression and substitute all of them or a particular one with another symbol.

2. Checking the correctness of software documentation. The set of indices generated by DIT can be used to detect if a symbol has been defined more than once, or a symbol has been used but not defined, or defined but never used.

6.2 Limitations and Future Work

As a limitation of the current implementation, the user is not able to produce a table that is wider than 150mm. This is because the maximum table width is defined as 150mm in TLT. If wider tables are required, we have to redefine the maximum table width and recompile the source code to generate a new executable file for TLT.

At present, LaTeX information about the default symbols are input to the symbol table by TLT. To be consistent with TTS conventions, two new classes (LaTexName and LaTexForm) need to be added to the symbol table as default classes, and LaTeX information about the default symbols needs to be moved from TLT to the default symbol table (default.sym). LaTexName is a class to store LaTeX name for atom/application symbols. For example, the LaTeX name for default symbol "***true***" (predicate true) is "\underline{\bf\emph{true}}". LaTexForm is a class to store LaTeX form for application symbols. For example, the LaTeX form for application symbol "$\int$" is an array of four strings: "\int_{", "}^{", "}", " ".

Another area of future work is to develop graphical user interfaces (GUIs) for TLT and DIT. In particular, the interface of TLT should be designed friendly so that the user does not have to know how to produce LaTeX documents such as compiling LaTeX code, converting dvi files to postscript files, etc.

6.3 Conclusions

Generating indices for software documents is a significant approach to improve the usability of formal documentation in general and tabular documentation in particular.

We have successfully developed TLT and DIT for such purpose. TLT makes it possible to produce software documents that include both informal (English) material and formal mathematical expressions, which is one of the most important and fundamental capabilities that should be provided by TTS. DIT increases the readability of such documents by providing a set of indices. The indices generated by DIT are particularly useful for searching and analyzing information about or contained in tables, which is not provided by any other existing indexing tools or systems.

In addition to the contribution to TTS, developing these two tools has showed the importance of applying software engineering principles to software development practice. Designing the module structures using information hiding principle made these tools as general as possible to meet future extensions and changes. Following the rational design process [15] kept this work on the right track and on schedule.

Appendix A

Examples of LaTeX Representations Generated by TLT

This appendix provides examples of the LaTeX representations produced by TLT, which cover both scalar expressions and tabular expressions.

A.1 Scalar Expressions

This example shows the LaTeX representation produced by TLT for a scalar expression. Figure A.1 shows the scalar expression that needs to be indexed and to be represented as part of a document. Figure A.2 illustrates the corresponding LaTeX representation generated by TLT. The input is as follows:

Expression:	see Figure A.1
Creates indices:	Yes
Represented as a single document:	No

$$x = y \vee x = wildcard \vee y = wildcard$$

Figure A.1: A Scalar Expression to be Converted to LaTeX

```
\begin{expression}
$x\label{2002-<< 1 >,< 1 >>}=\label{2002-<< 1 >>}y\label {2002-<< 1 >,< 2 >>} \vee
x\ label{2002-<< 2 >,< 1 >,< 1 >>}=\label{2002-<< 2 >,< 1 >>}wildcard \label{2002-<< 2 >,< 1 >,< 2 >>} \vee
y\label{2002-<< 2 >,< 2 >,< 1 >>}=\label{2002-<< 2 >,< 2 >>}wildcard\label{2002-<< 2 >,< 2 >,< 2 >>} $
\label{2002}
\end{expression}
```

Figure A.2: The LaTeX Representation of the Scalar Expression in Figure A.1 Using TLT

A.2 Tabular Expressions

This example shows the LaTeX representation produced by TLT for a tabular expression. Table A.2 is the tabular expression that needs to be represented as a single document and no indices are required for it. Figure A.3 illustrates the corresponding LaTeX code generated by TLT. The input is as follows:

Expression:	see Table A.2
Creates indices:	No
Represented as a single document:	Yes
Table width:	150
Column widths:	27, 68, 55
Header positions:	L, U
Caption:	A Tabular Expression to be Converted to LaTeX

$count(x, A, lower, upper)\ defined$

	$A[lower] = x$	$\neg(A[lower] = x)$
$lower < upper$	$(1 + count(x, A, (lower + 1), upper))$	$count(x, A, (lower + 1), upper)$
$lower = upper$	1	0

Table A.2: A Tabular Expression to be Converted to LaTeX

```
\documentclass[12pt]{article}
\usepackage{longtable}
\usepackage{array}
\newcounter{expncounter}
\newenvironment{expression}{\trivlist\incrementexpn{}}{\endtrivlist}
\newcommand{\incrementexpn}{\refstepcounter{expncounter}
\item[\bf{[\arabic{expncounter }]}]}
\begin{document}
$count(x,A,lower,upper)\ defined $
\begin{longtable}{|
>{$}p{27mm}<{$}||
>{$}p{68mm}<{$}|
>{$}p{55mm}<{$}|
}\hline
&A[lower]=x &\neg{(A[lower]=x)} \\ \hline \hline
\endhead
lower<upper &(1+count(x,A,(lower+1),upper)) &count(x,A,(lower+1),upper) \\ \hline
lower=upper &1 &0 \\ \hline
\caption{A Tabular Expression to be Converted to LaTeX}
\end{longtable}
$ $
\end{document}
```

Figure A.3: The LaTeX Reprensentation of Table A.2 Using TLT

Appendix B

Examples of Indices Generated by DIT

This appendix shows the indices generated by DIT for a document that includes the five expressions presented in Section B.1. These five expressions cover different kinds of expressions including scalar expressions, nested tables, wide tables, long tables, one-dimensional tables, two-dimensional tables, etc. They have been converted into LaTeX using TLT and inserted into this document. Table B.1 provides a brief description each of them.

Table B.1: A Brief Description on the Expressions to be Indexed

Expression Number	**Description**
1	nested table & wide table
2	scalar expression
3	one-dimensional table with H1 on the right
4	one-dimensional table with H1 on the left
5	two-dimensional table & long table

B.1 Expressions to be Indexed

[1]

	$f\_irr\_band = high\_irrational$	$f\_irr\_band = high\_irrational$
$signal\_value(m\_signal)$	a	b
$s\_irr\_band = high\_irrational$: h $s\_irr\_band = low\_irrational$: l $s\_irr\_band = rational$: n	$dash$	h
		next column below ⇓

	$f\_irr\_band = rational$	$f\_irr\_band = rational$
$signal\_value(m\_signal)$	b	b
$s\_irr\_band = high\_irrational$: h $s\_irr\_band = low\_irrational$: l $s\_irr\_band = rational$: n	l	n
		next column below ⇓

	$f\_irr\_band = rational$	$f\_irr\_band = rational$
$signal\_value(m\_signal)$	c	d
$s\_irr\_band = high\_irrational$: h $s\_irr\_band = low\_irrational$: l $s\_irr\_band = rational$: n	$dash$	h
		next column below ⇓

	$f\_irr\_band = low\_irrational$	$f\_irr\_band = rational$
$signal\_value(m\_signal)$	d	d
$s\_irr\_band = high\_irrational$: h $s\_irr\_band = low\_irrational$: l $s\_irr\_band = rational$: n	l	n
		next column below $\Downarrow$

	$f\_irr\_band = low\_irrational$
$signal\_value(m\_signal)$	e
$s\_irr\_band = high\_irrational$: h $s\_irr\_band = low\_irrational$: l $s\_irr\_band = rational$: n	$dash$

[**2**] $x = y \vee x = wildcard \vee y = wildcard$

[**3**]

a	$(4900 \leq m\_signal)$
b	$(4900 - db) < m\_signal \wedge m\_signal < 4900$
c	$((100 + db) \leq m\_signal) \wedge (m\_signal \leq (4900 - db))$
d	$100 < m\_signal \wedge m\_signal < (100 + db)$
e	$(m\_signal \leq 100)$

[**4**]

$s\_irr\_band = high\_irrational$	h

$s\_irr\_band = low\_irrational$	l
$s\_irr\_band = rational$	n

[5]

	$f\_irr\_band = high\_irrational$	$f\_irr\_band = high\_irrational$
$signal\_value(m\_signal)$	a	b
$band\_value(s\_irr\_band)$	$dash$	h

	$f\_irr\_band = rational$	$f\_irr\_band = rational$
$signal\_value(m\_signal)$	b	b
$band\_value(s\_irr\_band)$	l	n

	$f\_irr\_band = rational$	$f\_irr\_band = rational$
$signal\_value(m\_signal)$	c	d
$band\_value(s\_irr\_band)$	$dash$	h

	$f\_irr\_band = low\_irrational$	$f\_irr\_band = rational$
$signal\_value(m\_signal)$	d	d
$band\_value(s\_irr\_band)$	l	n

	$f\_irr\_band = low\_irrational$
$signal\_value(m\_signal)$	e

	$f\_irr\_band = low\_irrational$
$band\_value(s\_irr\_band)$	$dash$

B.2 Generated Indices

Complete Use-Index

Name	Expn	Row/Column	Page
100	3	(3,0)	66
		(4,0)	66
		(4,0)	66
		(5,0)	66
4900	3	(1,0)	66
		(2,0)	66
		(2,0)	66
		(3,0)	66
a	1	(1,1)	65
	3	(1,1)	66
	5	(1,1)	67
b	1	(1,2)	65
		(1,3)	65
		(1,4)	65
	3	(1,2)	66
	5	(1,2)	67
		(1,3)	67
		(1,4)	67
band_value	5	(2,0)	68
c	1	(1,5)	65
	3	(1,3)	66
	5	(1,5)	67
d	1	(1,6)	65
		(1,7)	66
		(1,8)	66
	3	(1,4)	66
	5	(1,6)	67
		(1,7)	67

Complete Use-Index

Complete Use-Index

Complete Use-Index

Complete Use-Index

Constant Use-Index

Constant Use-Index

Constant Use-Index

Variable Use-Index

Variable Use-Index

Application Use-Index

Bibliography

[1] Abraham, Ruth, "Evaluating Generalized Tabular Expressions in Software Documentation," *CRL Report 346*, February 1997.

[2] Bauer, Brian, "Documenting Complicated Programs," *CRL Report 316*, December 1995.

[3] Chechik, Marsha, "SC(R)^3: Towards Usability of Formal Methods," *Proceedings of CASCON'98*, pp. 177–189, November 1998.

[4] Department of Software Engineering, *Software Engineering at McMaster University*. Hamilton, Ontario: McMaster University, 1998.

[5] Goossens, Michel, et.al., *The LaTeX Companion*. Massachusetts: Addison-Wesley Company, Inc., 1994.

[6] Janicki, R, Parnas, D.L., Zucker, J.I., "Tabular Representations in Relational Documents," *Relational Methods in Computer Science*, September 1996.

[7] Janicki, Ryszard, "Towards a Formal Semantics of Tables," *CRL Report 264*, September 1993.

[8] Lamport, Leslie, *LaTeX: a Document Preparation System (2nd ed.)*. Massachusetts: Addison Wesley Longman, Inc., 1994.

[9] Parnas, D.L., "On the Criteria to be Used in Decomposing Systems into Modules," *Commun ACM*, vol. 15, pp. 1053–1058, December 1972.

[10] Parnas, D.L., "On a 'Buzzword': Hierarchical Structure," *Proceedings of the IFIP Congress 1974*, pp. 336–339, 1974.

[11] Parnas, D.L., "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 2, pp. 128–138, March 1979.

[12] Parnas, D.L., "Software Engineering Principles," *IEEE Transactions on Software Engineering*, vol. 22, no. 2, pp. 305–316, November 1984.

[13] Parnas, D.L., "Tabular Representation of Relations," *CRL Report 260*, October 1992.

[14] Parnas, D.L., "Predicate Logic for Software Engineering," *IEEE Transactions on Software Engineering*, vol. 19, no. 9, pp. 856–862, September 1993.

[15] Parnas, D.L., Clements, Paul C., "A Rational Design Process: How and Why to Fake It," *IEEE Transactions on Software Engineering*, vol. 19, no. 2, pp. 251–257, February 1993.

[16] Parnas, D.L., Madey J., "Functional Documentation for Computer Systems," *Science of Computer Programming*, vol. 25, no. 1, pp. 41–61, October 1995.

[17] Peters, Dennis K., "Using Test Oracles Generated from Program Documentation," *IEEE Transactions on Software Engineering*, vol. 24, no. 3, pp. 161–173, March 1998.

[18] Rastogi, Preeti, "Specialization: an Approach to Simplifying Tables in Software Documentation," *CRL Report 360*, March 1997.

[19] Serbanati, Luca Dan, *Integrating Tools for Software Development.* Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1993.

[20] Shen, Hong, "Implementation of Table Inversion Algorithms," *CRL Report 315*, December 1995.

[21] Software Engineering Research Group, "Table Tool System Developer's Guide," *CRL Report 229*, January 1997.

[22] Tilley, Scott R., "Documenting-in-the-large vs. Documenting-in-the-small," *Proceedings of CASCON'93*, pp. 1083–1090, October 1993.

[23] Valdis Berzins, Luqi, *Software Engineering with Abstraction.* Massachusetts: Addison-Wesley Publishing Company, Inc, 1990.

[24] Wheeler, David, et.al., *Software Inspection: an Industry Best Practice.* California: IEEE Computer Society Press, 1996.