FUNCTION COMPOSITION TOOL

FUNCTION COMPOSITION TOOL

By

ALBERT HENRY TYSON, B.SC., M.SC., B.ED.

A Thesis

Submitted to the School of Graduate Studies

in partial fulfillment of the requirements

for the degree of

Master of Engineering

McMaster University

MASTER OF ENGINEERING(1997) McMaster University
(Computer) Hamilton, Ontario

TITLE: Function Composition Tool

AUTHOR: Albert Henry Tyson,

B.Sc. (McMaster University)

M.Sc. (McMaster University)

B.Ed. (University of Toronto)

SUPERVISORS: Dr. David L. Parnas and Dr. Martin von Mohrenschildt

NUMBER OF PAGES: x, 107

Abstract

Experience has shown that the issue of software documentation cannot be ignored if safe reliable software is the goal. To be useful, software documentation should be easy to manipulate. Function tables are a natural way of documenting software through the use of mathematical tabular notation. The need has been expressed for a tool to automatically generate the mathematical composition of two function tables, which would document the sequential execution of two programs. The Function Composition Tool is a prototype toward this end. The tool is based on existing algorithms using normal function tables, and their extensions to vector function tables. This work involves the design and implementation of the software. Supporting software enables the execution of test suites on the tool.

Acknowledgements

I would like to express my appreciation for the efforts and advice of my supervisors, Dr. David L. Parnas and Dr. Martin von Mohrenschildt, and those of the defense committee, Dr. Emil Sekerinski and Dr. Jeff Zucker.

I would like to thank Dr. Martin Von Mohrenschildt for his help with the flex and yacc unix facilities used in the Maple to TTS translator.

I would like to thank Dennis Peters for his helpful comments and Ruth Abraham for integrating the code into the TTS.

To my family, a special thanks for: the infinite patience of my wife Selena, the support of my Father, and the courage and confidence to continue through the memory of my Mother.

This work was funded by the Telecommunications Research Institute of Ontario (TRIO), the Natural Sciences and Engineering Research Council (NSERC), and Bell Canada.

Contents

Abstract **iii**

Acknowledgements **iv**

1 Introduction **1**

2 Nomenclature **5**

2.1 Conditions and Simultaneous Substitutions 6

2.2 Composition with only Simultaneous Substitutions and Conditions . . 7

2.3 Tables 8

2.4 Composition with only Simultaneous Substitutions and Tables 10

3 Table Tool System (TTS) **13**

3.1 Table Holder 13

3.2 Info 14

3.3 General Table Semantics (GTS) modules 15

4 Design **17**

4.1 Assumptions 19

4.2 Introduction to the Modules 22
4.3 Module Guide 24
4.4 Informal Interface Specifications 26
4.5 Uses Hierarchy 30
4.6 Algorithms 34
4.6.1 Checking Expressions 34
4.6.2 Converting Normal to Vector Function Tables 35
4.6.3 Converting Vector to Normal Function Tables 37
4.7 Implementation Issues 38
4.7.1 Private Data Structure 38
4.7.2 Data Structure Guidelines 39
4.7.3 Status Checking Guidelines 39
4.7.4 Code Readability Guidelines 40
4.7.5 Symbol Naming Scheme 40

5 Testing Tools **42**

5.1 The MapToTts Translator 42
5.2 The ConSimSub Generator 43
5.3 The NorVecTab Generator 44
5.4 Function Composition Tool Test Harness 48

6 Examples **51**

6.1 Composition with only Simultaneous Substitutions and Conditions . . 51
6.2 Composition where Only One Expression is a Table 53

6.3 Composition of Two Normal or Vector Function Tables 55

7 Results and Conclusions **57**

7.1 Results 57

7.2 Limitations and Future Developments 58

7.3 Conclusions 60

A Conventional Expression Translator **62**

A.1 Lexical Analysis Rules 62

A.2 Syntax Analysis Rules 65

A.3 Translator Assumptions 66

A.4 Introduction to the Translator Modules 68

A.5 Translator Module Guide 70

A.6 Translator Informal Interface Specifications 72

A.7 Translator Uses Hierarchy 77

A.8 Translator Algorithms 83

A.9 Translator Implementation Issues 93

A.9.1 Data Structures 94

A.9.2 Data Structure Guidelines 94

A.9.3 Status Checking Guidelines 95

A.9.4 Code Readability Guidelines 95

B Module Interface Specifications **97**

List of Figures

4.1 Design Process 18

4.2 Organization of the Function Composition Tool 23

4.3 The Uses Hierarchy for CT_Mid_CreateVectorComp 32

4.4 The Uses Hierarchy for CT_Mid_CreateNormalFormComp 33

A.1 Organization of the Translator Utility 69

A.2 Legend for all Translator Uses Hierarchies. 78

A.3 Uses Hierarchy for MT_maptotts: MT_Init. 78

A.4 Uses Hierarchy for MT_maptotts: MT_MapleToTts. 78

A.5 Uses Hierarchy for MT_actions: ActDone. 78

A.6 Uses Hierarchy for MT_actions: ActBinFATag. 79

A.7 Uses Hierarchy for MT_actions: ActConstTag. 79

A.8 Uses Hierarchy for MT_actions: ActNatDot. 79

A.9 Uses Hierarchy for MT_actions: ActDotNat. 80

A.10 Uses Hierarchy for MT_actions: ActNatDotNat. 80

A.11 Uses Hierarchy for MT_actions: ActNegative. 81

A.12 Uses Hierarchy for MT_actions: ActName. 81

A.13 Uses Hierarchy for MT_actions: ActBinLETag. 81

A.14 Uses Hierarchy for MT_actions: ActUnaLETag. 81

A.15 Uses Hierarchy for MT_actions: ActBinPETag. 82

A.16 Uses Hierarchy for MT_actions: ActFun. 82

A.17 Uses Hierarchy for MT_actions: ActComma. 82

List of Tables

4.1 Expected EType of an Expression 20

4.2 Options for X and Y in $X \circ Y$ 21

4.3 The Uses Relation for the System 31

6.1 Composition of a simultaneous substitution and a condition 52

6.2 Composition of two simultaneous substitutions 52

6.3 Composition with One Vector Function Table on Left 53

6.4 Composition with One Normal Function Table on Right 54

6.5 Composition of Disjoint Tables 56

6.6 Composition of Nondisjoint Tables 56

A.1 Words and Names of Expected Symbols. 67

Chapter 1

Introduction

In the execution [1] of a terminating program, the content of certain memory locations, which are commonly referred to as program variables, is changed between the start and end of execution. One execution of a program can be thought of as an ordered pair, where the first component is the "before values" of all program variables, and the second component is the "after values" of the same. Thus a program can be represented by a set of ordered pairs. Since a mathematical relation is quite generally defined to be a set of ordered pairs, a relation can represent a program. The use of relations is an important concept in computer science, as evident from the references [2] and [3].

If A and B are programs, suppose F_A and F_B are the relations that represent them. Most computer languages provide facility for creating the new program, $A;B$, which is the program equivalent to program A, immediately followed by program B. Considering the collection of all program variables between A and B, each program may use some, but not necessarily all variables. Those variables not used are not changed, but may be included without loss of generality in the relation describing each program. The new program will use all the program variables and is represented by the relation $F_A \circ F_B$ that is the mathematical composition [4] of F_A and F_B, which is Definition 1.0.1.

Definition 1.0.1 (Mathematical Composition of Relations)
Suppose $F_A \subseteq S \times S$ and $F_B \subseteq S \times S$ are relations on S, the set of all tuples of the values of all program variables. The mathematical composition, $F_A \circ F_B$, is the set $\{(x,z)|\exists y \in S, (x,y) \in F_A \wedge (y,z) \in F_B\} \subseteq S \times S$.

It should be the case that a program is written to satisfy a *formal specification*. These specifications are also relations. A general Function Composition Tool can perform mathematical composition on a sequence of formal specifications. Suppose a large program has been separated into a semicolon sequence of smaller programs. If the original program and each member of the program sequence has a formal specification, then the original program specification can be compared for equivalence with the result of composing the sequence of formal specifications with the composition tool. This is a practical motivation for the development of the tool. Manual composition of formal specifications is tedious and thus error prone. An automated method would be particularly useful in safety-critical situations.

This work discusses a software tool that automates mathematical composition of functions represented by certain kinds of expressions found in the formal specification of software. These expressions can be grouped into two kinds; conventional expressions, and tabular expressions.

Expressions involving many conditions are common in the specification of software. If a conventional linear format is used, then reading the expressions becomes difficult and error prone. Experience has shown that tabular formats, with their multidimensionality, are natural and useful in reducing this problem.

Not all tabular arrangements are equally suited for a specific situation. The amount of repetition within parts of an expression, the degree of nesting of conditions, the number of conditions and results and the importance of various aspects of the expression to the user are some of the possible considerations. The readability and size of the resulting table are two of the key factors determining a table's usefulness. Since a variety of different forms of tables are used in practice, there must be a semantic or meaning associated with a table.

A collection of modules called the Table Holder, has been developed for representing tables which is robust and keeps the data structures private. The Table Holder is fully documented. Several tools, which use the Table Holder as a common element, have been developed for useful tasks, involving the creation and manipulation of tabular forms. The tool suite is collectively known as the Table Tool System (TTS). Development of the TTS is the focus of the McMaster

University Software Engineering Research Group (SERG), which promotes industry collaboration.

Most of the background information for this work can be found at McMaster University in the form of Communications Research Laboratory technical reports. In the early use of tables, the interpretation was intuitive. An understanding of some forms of tables, carefully defined, can be found in [5], which is based on the practical experience of using tables in real projects. Each table is treated separately, forming distinct classes of tables, each with their own meaning. It is possible to describe all tables with a single semantic model. The detailed development of a general semantics of tables appears in [6]. An introduction to the use of tabular forms in formal documentation occurs as [7]. In a different approach, which can be found in [8], tables are developed into an algebra. This same work contains the algorithms for the mathematical composition of normal function tables. The TTS Developer's Guide [9] describes the TTS system and how to use it.

The following list describes the organization of this document into chapters.

Nomenclature This chapter will introduce two specific tables, the Normal Function Table and the Vector Function Table. The background will also contain the algorithms for generating expressions which are the mathematical compositions of normal function tables.

Table Tool System (TTS) A brief discussion of terms used in the TTS is found in this chapter. The Table Holder, Info and General Table Semantics (GTS) module sets are included.

Design In this chapter, the necessary simplifying assumptions and special conventions required to define the scope of the problem are set down. Design contains the Module Guide, the Uses Hierarchy [10], and the informal Module Interface Specifications for the tool. Since the tool is to use both Normal and Vector Function Tables, conversion algorithms are described. The chapter finishes with some implementation issues that arise from the use of other modules, outside the tool.

Testing Tools This chapter includes the means to generate test cases in a timely fashion, and a test harness.

Examples This chapter includes some selected examples which are formatted results of the Function Composition Tool.

Results and Conclusions This chapter includes the results of the development of the tool, identifies limitations, implicates future work, and draws some

conclusions.

Appendix A This appendix contains documentation for a language translator used in test case generation.

Appendix B This appendix contains the Module Interface Specifications for modules directly accessible by the user of the Function Composition Tool.

Members of the intended audience might be interested in the mathematical composition of tables or the design and implementation of the tool itself. Specifically targeted are those involved in the Table Tool System.

Chapter 2

Nomenclature

This chapter defines the expressions that the tool uses to represent functions. It also describes the algorithms for composition.

The focus is on four kinds of expressions:

- conditions
- simultaneous substitutions
- proper normal function tables
- proper vector function tables.

The later two are collectively called *tables* and are defined in section 2.3. A condition is either true or false. In a simultaneous substitution, occurrences of specific variables are replaced with terms simultaneously. A simultaneous substitution can be thought of as a simultaneous assignment. A single assignment is <variable> ::= <expression>, in the familiar notation. A simultaneous assignment is <variable list> ::= <expression list>, where the single assignments occur at the same time. The definition of condition and simultaneous substitution is found in section 2.1.

The algorithms for composition between combinations of simultaneous substitutions and proper normal function tables, and between simultaneous substitutions and conditions are described.

This chapter makes use of ideas from mathematical logic. Good text references are [11] , [12], and [13]. All necessary definitions can be found in the primary source for the composition algorithms for proper normal function tables [8]. A brief explanation may be sufficient for these basic ideas.

sort Boolean, character and natural number as examples of sorts. Functions accept arguments of certain sorts and map them to some sort. A function with no arguments is a constant.

variable These may be of different sorts. The set of all variables is countable. The definition of variable is left as intuitive.

term A term is defined inductively as a constant, or a variable as the base case, and a function, applied to terms, as the induction step.

subterm A variable has only one subterm, namely itself. If $f(t_1, \ldots, t_n)$ is a function of terms, t_i, then the set of subterms contains $f(t_1, \ldots, t_n)$, and all the subterms of each t_i.

single substitution The notation $\text{subst}(\{x \mapsto \bar{t}\}, t)$ is a single substitution on the term t, where every free occurrence of the variable x in t is replaced by the term $\bar{t}$ (x must be free in the sense that it is not bound by a quantifier).

simultaneous substitution The idea is defined in section 2.1

2.1 Conditions and Simultaneous Substitutions

A *condition* is a term, c, of sort *boolean*. The expression $x + y < 0$ is a condition where x and y are variables, and $<$ is the usual function of sort boolean.

A *simultaneous substitution* is an expression of the form, $\{x_i \mapsto t_i | i = 1, \ldots, n\}$. The x_i are variables, no two of which are the same, and the t_i are terms. The sort of x_i must be the same as the sort of t_i. The $x_i \mapsto t_i$ notation is a single substitution, meaning occurrences of the variable x_i are to be substituted with the term t_i. If t is a general term, then the expression $\text{subst}(\{x_i \mapsto t_i | i = 1, \ldots, n\}, t)$ means the simultaneous substitution, $\{x_i \mapsto t_i | i = 1, \ldots, n\}$, has been applied to t. The result is the term produced from

t in which every occurrence of the variable, x_i, in t, has been substituted with the term, t_i, and where all these changes are done at the same time.

The following are some useful definitions about the content of terms and simultaneous substitutions. The set of all variables of the term t is written as $\text{var}(t)$. Extending var to simultaneous substitutions, F, and G,
$\text{var}(F) \stackrel{df}{=} \{y | y \in \text{var}(t_i) \wedge (x_i \mapsto t_i) \in F\}$, $\text{var}(F,G) \stackrel{df}{=} \text{var}(F) \cup \text{var}(G)$ and Lastly,
$\text{rangevar}(F) \stackrel{df}{=} \{x_i | (x_i \mapsto t_i) \in F\}$ and $\text{rangevar}(F,G) \stackrel{df}{=} \text{rangevar}(F) \cup \text{rangevar}(G)$.

Let $F = \{x \mapsto 1, y \mapsto a, z \mapsto f(1) + a\}$ be a simultaneous substitution. The above expression means every variable x will be substituted with the constant 1, every variable y will be substituted with the variable a, every variable z will be substituted with $f(1) + a$, and all these changes are done at the same time. Note that 1, a, and $f(1) + a$ are terms. The $\text{var}(F) = \{a\}$ and $\text{rangevar}(F) = \{x, y, z\}$.

2.2 Composition with only Simultaneous Substitutions and Conditions

The $\circ$ symbol notates the relational composition operator. [1] The composition, $F \circ c$, of a simultaneous substitution, $F = \{x_i \mapsto t_i | i = 1, \ldots, n\}$, and a condition, c, is the condition, $\text{subst}(\{x_i \mapsto t_i | i = 1, \ldots, n\}, c)$.

If $F = \{x \mapsto a, z \mapsto 1\}$, is a simultaneous substitution, and c is the condition $x + y < 5$, where x, y, z, a are variables, then the composition $F \circ c$ is $a + y < 5$.

For two simultaneous substitutions $F = \{x_i \mapsto t_i | i = 1, \ldots, n\}$ and $G = \{y_j \mapsto u_j | j = 1, \ldots, m\}$, the relational composition, $F \circ G$ is the simultaneous substitution,
$\{y_j \mapsto \text{subst}(F, u_j) | j = 1, \ldots, m\} \cup \{x_i \mapsto t_i | (x_i \mapsto t_i) \in F \wedge x_i \notin \text{rangevar}(G)\}$.

If $F = \{x \mapsto 3, y \mapsto a\}$ and $G = \{x \mapsto y + 2, z \mapsto xw\}$ where a, w, x, y, z are variables and F, G are simultaneous substitutions, then the composition, $F \circ G$, is $\{x \mapsto a + 2, y \mapsto a, z \mapsto 3w\}$.

The composition is associative. Note that all the range variables of F and all the range variables of G appear as range variables in the composition $F \circ G$.

[1] We write $f \circ g$ for $f; g$, first f then g and not the other way around.

Even if none of the range variables of F are used in G, the composition is not empty. In this case, $F \circ G$ will contain all the simple substitutions of both F and G.

2.3 Tables

A *proper normal function table* has the form $T = (H^1, H^2, \ldots, H^n, G)$. The H^i, called *guard headers*, are sets of conditions, indexed by a set I_i. In notation, $H^i = \{h^i_{\alpha_i}\}_{\alpha_i \in I_i}$. The index sets, I_i, taken as a cartesian product $I_1 \times I_2 \times, \ldots, \times I_n$, index the set G, called the *grid*, of simultaneous substitutions. In notation, $G = \{g_{(\alpha_1,\ldots,\alpha_n)}\}_{(\alpha_1,\ldots,\alpha_n) \in I_1 \times I_2 \times, \ldots, \times I_n}$.

The format of 2-dimensional normal function tables used in this document, is illustrated in the following illustration. Note the position of the headers and grid.

	H^1
H^2	G

Selecting a tuple of elements over the guard headers, determines an expression in the grid. For a normal table, the logical rule which relates the guard header elements, $h^i_{\alpha_i}$ to the grid element, $g_{(\alpha_1,\ldots,\alpha_n)}$ is $(h^i_{\alpha_1} \wedge \ldots \wedge h^n_{\alpha_n}) \Rightarrow g_{(\alpha_1,\ldots,\alpha_n)}$, which is an if...then... conditional expression. In more general tables, a broader means of interpreting the table is necessary and has been provided in [6].

The correspondence between header and grid cells is illustrated through the subscripts in the following illustration. For example, $(h^1_2 \wedge h^2_1) \Rightarrow g_{(2,1)}$, must hold.

	h^1_1	h^1_2
h^2_1	$g_{(1,1)}$	$g_{(2,1)}$
h^2_2	$g_{(1,2)}$	$g_{(2,2)}$
h^2_3	$g_{(1,3)}$	$g_{(2,3)}$

It is often the case that only one of the conditions in a guard header should be *true* in any instance. This expectation motivates the idea of properness, as found in [5]. A guard header is *proper* if and only if the disjunction of all its elements is *true* and the conjunction of any two elements is *false*. In notation, $\bigvee_{\alpha \in I_i} h^i_\alpha = \mathit{true}$ and $\forall \alpha_1, \alpha_2 \in I_i, h^i_{\alpha_1} \wedge h^i_{\alpha_2} = \mathit{false}$. All the guard headers in a proper normal function table are proper.

Consider the following examples of headers, where x is an integer variable. The following header is not proper since the disjunction omits the $x = 1$ case.

$x < 1$	$x > 1$

The next header is not proper since the overlap of the two conditions results in their conjunction not being always false.

$x < 2$	$x > 0$

The last header is proper.

$x < 1$	$x = 1$	$x > 1$

For a normal table, T, the set of all variables occurring in the table is, $\text{var}(T) \stackrel{df}{=} \text{var}(H_1, \ldots, H_n, G)$. The set of all the range variables of T is $\text{rangevar}(T)$ which is defined as $\text{rangevar}(G)$.

The following is an example of a proper normal function table.

	$x < 0$	$x = 0$	$x > 0$
$b = true$	$\{x \mapsto -x\}$	$\{y \mapsto x\}$	$\{x \mapsto x + 1,$ $y \mapsto f(x)\}$
$b = false$	$\{y \mapsto v\}$	$\{x \mapsto 1\}$	$\{x \mapsto xy,$ $y \mapsto x + y\}$

The preceding discussion of composition of tables was restricted to proper normal function tables. Another common form of table is the proper vector function table. Composition will not be reworked for this new table, since it will be clear subsequently that conversion to the normal table is straight forward. The definition of the vector table, however, is needed.

Another kind of header used in vector tables must be defined. The guard headers used previously contained only conditions. An H^v, is called a *vector header*, if it is a set of single variables, indexed by a set I_v.

A *proper vector function table* is similar to a proper normal function table with three modifications. Exactly one of the headers, H^v, is a vector header and the

rest are guard headers. The grid elements are terms rather than simultaneous substitutions. The sort of the variable, $h^v_{\alpha_v}$, must be the same as the range-sort of $g_{(\alpha_1,\ldots,\alpha_v,\ldots,\alpha_n)}$.

Selecting a tuple of elements over the guard headers determines a row of expressions in the grid. Each term, $g_{(\alpha_1,\ldots,\alpha_v,\ldots,\alpha_n)}$, in this row corresponds to one variable, $h^v_{\alpha_v}$, in the vector header. For a vector table, the logical rule which relates the header elements, $h^i_{\alpha_i}$, to the grid element, $g_{(\alpha_1,\ldots,\alpha_n)}$, is $(h^1_{\alpha_1} \wedge \ldots \wedge h^{v-1}_{\alpha_{v-1}} \wedge h^{v+1}_{\alpha_{v+1}} \wedge \ldots \wedge h^n_{\alpha_n}) \Rightarrow \forall \alpha_v \in I_v, (h^v_{\alpha_v} \mapsto g_{(\alpha_1,\ldots,\alpha_n)})$. Each guard header of a proper vector function table is proper.

The following is an example of a proper vector function table.

	$x < 0$	$x = 0$	$x > 0$
x	$-x$	1	$x+1$
y	v	$-y$	xy

2.4 Composition with only Simultaneous Substitutions and Tables

The relational composition of simultaneous substitutions can be extended to proper normal function tables. The extended composition is still associative as proven in [8], and can be described in four cases.

The relational composition of a simultaneous substitution, F, and a proper normal function table, $T = (H^1, \ldots, H^n, G)$, involves two cases. Performing $T \circ F$, generates a proper normal table, $\bar{T}$, from T, where the only change is the composition of each $g \in G$ with F. In notation, $\bar{T} = (H^1, \ldots, H^n, \bar{G})$, where $\bar{g}_{(\alpha_1,\ldots,\alpha_n)} = g_{(\alpha_1,\ldots,\alpha_n)} \circ F$. Performing $F \circ T$, generates the proper normal table $\bar{T}$, a modified T, where F has been composed with every element. In notation, $\bar{T} = (\bar{H}^1, \ldots, \bar{H}^n, \bar{G})$, where $\bar{h}^i_{\alpha_i} = F \circ h^i_{\alpha_i}$ and $\bar{g}_{(\alpha_1,\ldots,\alpha_n)} = F \circ g_{(\alpha_1,\ldots,\alpha_n)}$.

In this example, G is a simultaneous substitution. The following table, F,

C_1	C_2
S_1	S_2

is a normal function table, where C_1 and C_2 are conditions and S_1 and S_2 are

simultaneous substitutions. Then $F \circ G$ is the table,

C_1	C_2
$S_1 \circ G$	$S_2 \circ G$

and $G \circ F$ is the table

$G \circ C_1$	$G \circ C_2$
$G \circ S_1$	$G \circ S_2$

The relational composition, $T \circ \bar{T}$, of two proper normal function tables, $T = (H^1, \ldots, H^n, G)$ with $G = \{g_{(\alpha_1,\ldots,\alpha_n)}\}_{(\alpha_1,\ldots,\alpha_n)\in I_1\times\ldots\times I_n}$ and $\bar{T} = (\bar{H}^1, \ldots, \bar{H}^m, \bar{G})$ with $\bar{G} = \{\bar{g}_{(\beta_1,\ldots,\beta_m)}\}_{(\beta_1,\ldots,\beta_m)\in J_1\times\ldots\times J_m}$, results in a proper normal table. The composition is divided into two cases based on the occurrence of range variables of T, in the headers of $\bar{T}$. This relationship is summarized in the definition of the set $Disj = \text{var}(\bar{H}^1, \ldots, \bar{H}^m) \cap \text{rangevar}(T)$. If $Disj = \emptyset$ then the two tables are said to be *disjoint*, otherwise they are *nondisjoint*.

If $Disj = \emptyset$, then none of the substitutions in grid of T affect the headers of $\bar{T}$. Thus all the headers of the two tables are kept as is and the composition is only between elements of the two grids. Thus $\tilde{T} = T \circ \bar{T} = (H^1, \ldots, H^n, \bar{H}^1, \ldots, \bar{H}^m, \tilde{G})$, where $\tilde{G} = \{\tilde{g}_{(\alpha_1,\ldots,\alpha_n,\beta_1,\ldots,\beta_m)}\}_{(\alpha_1,\ldots,\alpha_n,\beta_1,\ldots,\beta_m)\in I_1\times\ldots\times I_n\times J_1\times\ldots\times J_m}$ with $\tilde{g}_{(\alpha_1,\ldots,\alpha_n,\beta_1,\ldots,\beta_m)} = g_{(\beta_1,\ldots,\beta_m)} \circ \bar{g}_{(\alpha_1,\ldots,\alpha_n)}$

If $Disj \neq \emptyset$, then the grid elements of T affect the header elements of $\bar{T}$. The grid of the new table, $\tilde{T}$, must contain the results, as grid element $\tilde{g}$, of the composition of one grid element, g, of T and one grid element, $\bar{g}$, of $\bar{T}$. The number of entries in the grid of $\tilde{T}$ would be the product of the number of grid elements in each of the two tables without duplicating any composition. The condition for $\tilde{g}$ would be the conjunction of the condition for g with a new condition formed by applying g to the condition of $\bar{g}$. Thus the table $\tilde{T}$ has the form $(\tilde{H}, \tilde{G})$, with $\tilde{G} = \{\tilde{g}_\gamma\}_{\gamma\in K}$. If $\tilde{g}_\gamma = g_{(\beta_1,\ldots,\beta_m)} \circ \bar{g}_{(\alpha_1,\ldots,\alpha_n)}$, then $\tilde{h}_\gamma = (h^1_{\alpha_1} \wedge \ldots \wedge h^n_{\alpha_n}) \wedge (g_{(\alpha_1,\ldots,\alpha_n)} \circ (\bar{h}^1_{\beta_1} \wedge \ldots \wedge h^m_{\beta_m}))$.

In this example, F is the normal table

$z < 0$	$z \geq 0$
S_1	S_2

and G is the normal table

C_1	C_2
$\{x \mapsto a, y \mapsto b\}$	$\{x \mapsto c\}$

where x, y, z are variables. Then the range variables of G, x and y, are completely disjoint from the variable in the conditions of F. Thus $G \circ F$ is

	C_1	C_2
$z < 0$	$\{x \mapsto a, y \mapsto b\} \circ S_1$	$\{x \mapsto a, y \mapsto b\} \circ S_2$
$z \geq 0$	$\{x \mapsto c\} \circ S_1$	$\{x \mapsto c\} \circ S_2$

In this example, F is the normal table

$x + y < 0$	$x + y \geq 0$
S_1	S_2

and G is the normal table

C_1	C_2
$\{x \mapsto a, y \mapsto b\}$	$\{x \mapsto c\}$

where x, y are variables. Then the range variables of G, x and y, are common with variables in the conditions of F. Thus $G \circ F$ is

$C_1 \wedge$ $(a + b < 0)$	$C_1 \wedge$ $(a + b \geq 0)$	$C_2 \wedge$ $(c + y < 0)$	$C_2 \wedge$ $(c + y \geq 0)$
$\left\{ \begin{array}{l} x \mapsto a \\ y \mapsto b \end{array} \right\} \circ S_1$	$\left\{ \begin{array}{l} x \mapsto a \\ y \mapsto b \end{array} \right\} \circ S_2$	$\{x \mapsto c\} \circ S_1$	$\{x \mapsto c\} \circ S_2$

Chapter 3

Table Tool System (TTS)

Tables can clearly represent multi-conditional expressions, but their manual manipulation is tedious. The creation of a system, known as the Table Tool System (TTS), to support the use of tables in documentation is the result of the Software Engineering Research Group (SERG) at McMaster University. The TTS is organized into a kernel with utilities that support a collection of applications or tools. The Function Composition Tool makes considerable use of the kernel and the kernel utilities. The TTS Developer's Guide [9] describes the TTS in reasonable detail. Since this document makes reference to TTS definitions, it is necessary to provide explanation here.

3.1 Table Holder

Table Holder is a set of modules which is part of the kernel of the TTS. The user has access to the Expn, Index, Path, Shape and TH_error modules. The following is a list of definitions of terms used with these modules that should be known.

TH_Token This is the token data type of the TH_error module. Tokens are used to indicate the success or failure status of a call to a module.

Expn This is the data type of the Expn module. It can be reasoned with by considering it like a prefix expression tree, where the nodes of the expression are symbols.

Id The Id data type is a means of identification. It applies to the node of an Expn, indicating what symbol belongs there.

EType The EType data type indicates if a node of an expression is a constant (ConstTag), a predicate constant (PConstTag), a variable (VarTag), a logical expression (LETag), a quantified logical expression (QLETag), a predicate expression (PETag), a function table (FTableTag) or a predicate table (PdTableTag).

Shape This data type of the Shape module, refers to the shape of a table, such as, the number of headers and their sizes.

Index This is the data type of the Index module. Only tables use an index. It is used to refer to a particular element of a header or the grid.

arity Refers to the number of arguments a function takes.

Path This is the data type of the Path module. It indicates a position in a arbitrary expression by acting like a route map from the tree root of the expression, using function argument position and table Index to indicate turns.

3.2 Info

Info is a set of modules which is the remaining part of the kernel of the TTS. Info is intended to be used with Table Holder. The user has access to the Information and In_error modules. The following is a list of definitions of terms used with these modules that should be known.

In_Token This is the token data type of the In_error module. Tokens are used to indicate the success or failure status of a call to a module.

SymTbl A data type of the Information module representing a symbol table. It contains information about the symbols used in expressions of the Table Holder. The Id data type is used to establish the association between a symbol and a node of the Expn data type.

class A category of information in the data type SymTbl. The entry for a class contains specific information about the symbol such as font or EType. There are many classes available to all symbols.

Name This class contains a character string which is intended to be the name of the symbol.

Tag This class contains information about the EType of a symbol. The information is not the EType data type itself.

Arity This class contains integer information which is the arity of a function or operator symbol.

3.3 General Table Semantics (GTS) modules

The GTS modules were developed by Ruth Abraham [14]. The Function Composition Tool uses GTS_SYNTAX_CHECKING, GTS_INFORMATION, GTS_TABLE_SEMANTICS, and GTS_STATUS_REPORTING modules to manage information about normal function tables and vector function tables.

The GTS makes use of the idea of information flow found in the general table semantics paper [6]. Information flow answers the question "In what order do I consider the headers and grid in order to read the table?". There is a characteristic flow between headers and grid for normal function tables and another for vector function tables which can be identified and named. In normal function tables, all guard headers are read before the grid. In vector function tables, all guard headers are read before the grid, and the vector header is read after the grid. This called the *cell connection graph*, since if all headers and the grid are considered vertices, then the order of reading defines the edges of a directed graph.

Recall the logical expression for relating the headers and the grid in normal function tables on page 8 and a similar rule for vector function tables on page 10. The antecedent of the logical expression combines the entries of various guard headers. This defines the *table predicate rule*. Similarly, the consequent of the logical expression defines the *table relation rule*.

The GTS requires the user to construct the table predicate rule and the table relation rule as an Expn in the table holder. Numbers are used to encode the headers or grid since headers and the grid are accessed in the table holder using numbers. A number is represented as a digit string in the Name class entry of the SymTbl being used. The GTS_Table_Semantics module requires this representation.

The following is a list of definitions of terms concerning the GTS modules that should be known.

GTS_Token This is the token data type of the GTS_STATUS_REPORTING module. Tokens are used to indicate the success or failure status of a call to a module.

GTS_Sem This is a data type of the GTS_TABLE_SEMANTICS module. It contains the information necessary for the interpretation of a table.

GTS_CCG This is a data type of the GTS_TABLE_SEMANTICS module. It refers to the cell connection graph by name. All normal function tables have a NORMAL cell connection graph, while vector function tables have a VECTOR cell connection graph.

Chapter 4

Design

The process followed in designing the Function Composition Tool is illustrated in Figure 4.1. This chapter discusses several topics related to the design of the tool. Section 4.1, containing assumptions about the composition problem that the tool is to solve, is part of the *requirements specification*. The *module guide*, the *usage hierarchy*, and informal *module interface specifications* are in this chapter. The additional algorithms of section 4.6 and implementation issues in section 4.7 are part of the *module internal designs*. This chapter uses ideas found in chapter 2 and chapter 3.

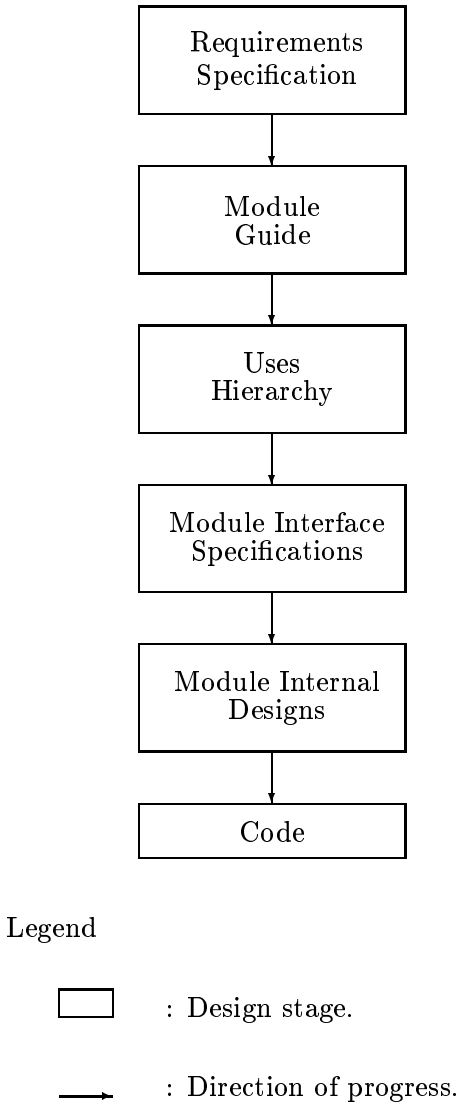


Figure 4.1: Design Process

4.1 Assumptions

This section contains the ideas used to make the algorithms for the composition of expressions more easily realizable within the TTS. They serve to make the problem more specific, cutting it down from the more general situation. Thus this section is part of the requirements specification for the Function Composition Tool.

The operation of the tool is that of a simple batch process, where the access function is called on the arguments of the composition. No input expressions are modified. The result of composition of tables can be either a normal or vector function table. This is determined by the access function call given by the user. No simplification is done to expressions either before or after composition. Issues surrounding simplification will be delegated to other efforts such as those of [15].

Rather than represent a simultaneous substitution as the set, $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, a notation using n-tuples is adopted. The representation can be thought of as $(x_1, \ldots, x_n) \mapsto (t_1, \ldots, t_n)$, or in prefix form, $\mapsto (\text{n-tuple}(x_1, \ldots, x_n), \text{n-tuple}(t_1, \ldots, t_n))$, where $\mapsto$ and n-tuple are thought of as operators.

In a normal table, each simultaneous substitution in the grid can have a different set of range variables. The range variables need not be the same. This means the grid expressions can appear very different in arity.

It is assumed there is no sort information. Every expression in the TTS has an EType. The Table 4.1 shows the expected ETypes for the expressions used by the Function Composition Tool. Since substitution on quantified expressions can bind variables that should be free, it is assumed that there are no quantifiers in the expressions. Thus the QLETag EType should not occur. Since only normal and vector function tables are allowed, the PdTableTag EType should not occur. The ETypes allowed for terms and the inductive definition of term in chapter 2 indicate that no subterm can have EType FTableTag. Thus no condition can have a subterm of FTableTag either. This means that tables are not nested. Further discussion on quantified expressions and nested tables occurs in section 7.2.

The tool handles specific forms of relational composition of expressions representing functions. Suppose X and Y are expressions. In order for the tool to perform $X \circ Y$, the possibilities for X and Y are found in Table 4.2.

All tables to be used with the Function Composition Tool are defined

Expression	EType
variable	VarTag
term	ConstTag PConstTag VarTag LETag PETag FATag
condition	PConstTag PETag LETag
simultaneous substitution	FATag
table	FTableTag

Table 4.1: Expected EType of an Expression

using the table semantics of the GTS Tool (General Table Semantics Tool). In particular, the cell connection graph, table predicate rule, and table relation rule must be set.

The Function Composition Tool uses two conventions concerning table semantics. There is an existing TTS convention that specifies the main grid of a table is the last grid, in the sense of the table holder. This is actually only for convenience since the information can be deduced from the GTS semantic information.

The position of the vector header, in the case of vector function tables, is provided from the GTS directly. When an output vector table is generated by the current Function Composition Tool, the vector header position is set as the second to last grid, and this fact is recorded in the GTS semantics.

The Function Composition Tool takes for the name of a symbol, the entry found in the Name class field of the symbol table record for that symbol. The tool assumes all the names found in the default symbol table under the class Name. The name of a symbol is used to identify it and any information associated with it. Thus the name of an operator should be unique. The tool locates the Id of a symbol by searching on the symbols name. Thus if more than one Id has that name, the resulting status will indicate an error. All expressions are considered to be associated with the same symbol table. This is equivalent to agreeing on the

	X	Y
Case 1	simultaneous substitution	condition
Case 2	simultaneous substitution	simultaneous substitution
Case 3	simultaneous substitution	normal or vector function table
Case 4	normal or vector function table	simultaneous substitution
Case 5	normal or vector function table	normal or vector function table

Table 4.2: Options for X and Y in $X \circ Y$

notation to be used in a mathematical discussion. It is assumed that the simultaneous substitution operator is already present in the symbol table. No table is assumed to have a particular name. Thus names and Ids for tables are discretionary, but are generally meaningful to the user.

The success or failure of the Function Composition Tool, is referred to as the status. Status is maintained between calls to the tool. The notion of status is used in all of the TTS. For each module, it is assumed there is only one success token. For modules external to the Function Composition Tool, only the knowledge of the success status token is assumed. All status tokens, except those indicating simply success, from any used modules, either external or internal, are assumed to indicate error. The design of the Function Composition Tool is such that if an error is detected, then progress should be avoided.

4.2 Introduction to the Modules

Most modern software systems require considerable time and effort in their construction. The work of many people over many months is usually necessary. The notion of the work assignment, a portion small enough for one person to manage in a reasonable time, can decompose a system into modules. In a paper [16] now considered classic, the idea that a module should hide the exact implementation of an abstract data type, making the rest of the system immune to changes in its implementation, is put forward. The module communicates data with the outside world through only its access programs which are designed to keep the data structure secret. Modularization can be thought of as the separation of design concerns, allowing attention to be directed toward one issue at a time.

Using the principles of modularization [16], the Function Composition Tool is decomposed into nine modules. The Figure 4.2 illustrates the modular organization. Modules are indicates by solid line boxes. Portions of the tool to which can be applied a label carrying an informal meaning are enclosed by a dashed line box. The label "user" refers to anything using the tool, including another tool. The portion labeled "core functionality" is that portion that deals with only the composition algorithms. The "middleware" section, between the "user" and "core functionality", extends functionality to make the tool more useful to the user. The solid lines segments indicate direct communication between two modules or between a module and the user. The communication relation is non-reflexive, symmetric, and non-transitive. The CT_Error module is omitted from the diagram. It is to be understood that all modules communicate with CT_Error.

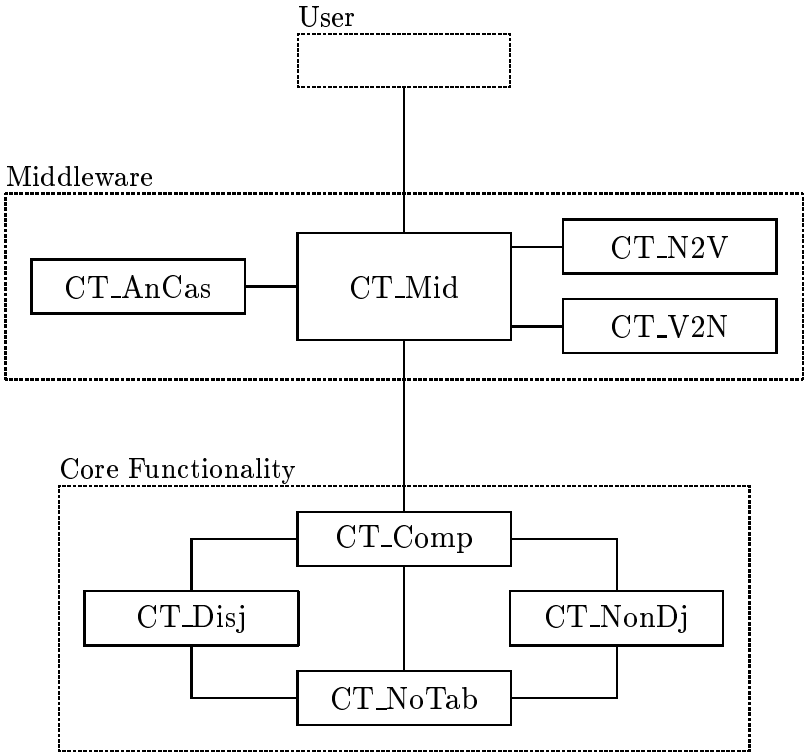


Figure 4.2: Organization of the Function Composition Tool

4.3 Module Guide

The module guide indicates what is encapsulated in each module. These are the secrets of the module or the information which it hides.

CT_Mid Module

The module hides the form of the composition problem that is acceptable to the Function Composition Tool. The secret of the module is the set of management decisions used to identify the necessary process steps.

CT_AnCas Module

The secret of the module is the algorithm for checking expressions. This module hides information regarding which individual expressions will not be acceptable to the Function Composition Tool. It does not contain tactics to describe how to proceed with composition.

CT_V2N Module

The secret of this module is the algorithm for converting vector function tables to normal function tables. The module hides the information relating the elements of the normal function table to the vector function table. The module does not contain validity information to check expressions.

CT_N2V Module

The secret of the module is the algorithm for converting normal function tables to vector function tables. The module hides the information relating the elements of the vector function table to the normal function table. The module does not contain validity information to check expressions.

CT_Comp Module

The secret of this module is the set of two algorithms for composition problems in which exactly one of the arguments is a normal function table. This module hides the management decisions which identify the several cases of composition that occur. The module does not contain information for checking validity.

CT_NonDj Module

The secret of this module is the algorithm for performing composition on nondisjoint normal function tables. The module does not contain information for checking validity.

CT_Disj Module

The secret of this module is the algorithm for performing composition on disjoint normal function tables. The module does not contain information for checking validity.

CT_NoTab Module

The secret of this module is the set to two algorithms for the composition of conventional expressions. The module hides the use of a private data structure to represent simultaneous substitutions. There is no information contained in this module for checking validity of expressions.

CT_Error Module

This module is responsible for providing the data type CT_Token for the user and other programs. The module hides the data structure of the CT_Token type. The CT_Token type preserves the status token of the Function Composition Tool from the last access program invocation, indicating success or the general reason for failure.

4.4 Informal Interface Specifications

The functions of each module are described in this section, without using formal language. A listing of the access programs for each module contains a brief explanation and the types of the arguments and returns.

The CT_Mid module is one of two modules intended to be accessed directly by the user, but the only module to which the user submits information. The other module is the CT_Error module. All other modules are intended to be local and not available to the user. For these two modules, refer also to appendix B.

CT_Mid Module Interface

This is the middleware. Since function composition of tables is defined for a restricted set of expressions, it is necessary to provide middleware to bridge the gap to expressions of the TTS in general. To illustrate this functionality, CT_Mid is responsible for accepting expressions of either condition, simultaneous substitution normal function table, or vector function table types. The key functionality of the module is to allow the user to decide the type of the resulting table, either normal or vector. There are three access programs.

CT_Mid_Init() Initializes the tool to a known initial state.

Expn CT_Mid_CreateNormalFormComp(SymTbl t**, Expn** l**, Expn** r **)** Returns the composition, $l \circ r$, using the symbol table t. A tabular result will be a normal table.

Expn CT_Mid_CreateVectorComp(SymTbl t**, Expn** l**, Expn** r **)** Returns the composition, $l \circ r$, using the symbol table t. A tabular result will be a vector table.

CT_AnCas Module Interface

This is the case analyzer. The module examines the expressions submitted for composition and determines if they satisfy the assumptions in section 4.1. It provides the tool with the ability to determine if it should proceed with the composition. No indication is given as to how to proceed. There is one access program.

bool CT_AnCas_AnalyseCase(SymTbl t**, Expn** e **)** It returns BOOL_TRUE if the expression e, supported by the symbol table t is found to satisfy all the described assumptions in section 4.1. Otherwise it returns BOOL_FALSE.

CT_V2N Module Interface

This is the vector to normal converter. The module converts a vector function table into a normal function table, using the assumptions in section 4.1. No checking of expressions is performed. The module contributes toward the functionality of allowing the user to submit vector tables. There is one access program.

Expn CT_V2N_VectorToNf(SymTbl t**, Expn** e **)** For a vector function table e supported by symbol table t, it returns a normal function table.

CT_N2V Module Interface

This is the normal to vector converter. The module converts a normal function table into a vector function table, using the assumptions in section 4.1. No checking of expressions is performed. The module contributes toward the functionality of allowing the user to request that tabular results are vector tables. There is one access program.

Expn CT_N2V_NfToVector(SymTbl t**, Expn** e **)** For a normal function table e supported by symbol table t, it returns a vector function table.

CT_Comp Module Interface

This is a manager for composition. The module handles a composition problem according to the assumptions in section 4.1, with the additional requirement that all tables are normal function tables. No checking of expressions is performed. The different cases of composition are detected by the module and appropriate action is taken. There is one access program.

Expn CT_Comp_GenNF(SymTbl t**, Expn** l**, Expn** r **)** Returns the composition, $l \circ r$, using the symbol table t. All tables are normal function

tables.

CT_NonDj Module Interface

This is the nondisjoint composition engine. The module handles a composition problem satisfying the assumptions in section 4.1. There is an additional requirement, that the problem is the composition of two normal function tables, and the result is also a normal function table. It is intended, but not necessary, that some of the range variables of the left-hand-side table must also be found in the headers of the right-hand-side table. The resulting table has only one header and a dimensionality of 1. No checking of expressions is performed. The module contributes part of the functionality of the composition of two tables. There is one access program.

Expn CT_NonDj_Comp(SymTbl t**, Expn** l**, Expn** r **)** Returns a normal function table, which is the composition of two normal function tables, $l \circ r$, using symbol table t and considered non-disjoint.

CT_Disj Module Interface

This is the disjoint composition engine. The module handles a composition problem satisfying the assumptions in section 4.1, with two additional requirements. Firstly, the problem is the composition of two normal function tables, and the result is also a normal function table. Secondly, none of the range variables of the left-hand-side table are found in the headers of the right-hand-side table. The resulting table has each of the headers of the original tables and a dimensionality which is the sum of that of the original tables. No checking of expressions is performed. The module contributes part of the functionality of the composition of two tables. There is one access program.

Expn CT_Disj_Comp(SymTbl t**, Expn** l**, Expn** r **)** Returns a normal function table, which is the composition of two normal function tables, $l \circ r$, using symbol table t, and considered disjoint.

CT_NoTab Module Interface

This is the composition engine. The module handles a composition problem satisfying the assumptions in section 4.1, with the additional requirement that no tables are involved. No checking of expressions is performed. The module provides all the functionality necessary for composition of conditions and simultaneous substitutions. There are two access programs.

Expn CT_NoTab_SubOSub(SymTbl t**, Expn** l**, Expn** r **)** Returns the simultaneous substitution that results from the composition of two simultaneous substitutions, $l \circ r$, using symbol table t.

Expn CT_NoTab_SubOComp(Expn l**, Expn** r **)** Returns the condition expression that results from the composition, $l \circ r$, of the condition expression, r, and the simultaneous substitution, l, using the symbol table t.

CT_Error Module Interface

This is the status module for the tool. The status token persists between calls to the module. The purpose of the module is to maintain the tool status from the last access program invocation. The tool status can be checked using the modules access programs to determine if an error has been detected or if progress can be made. The module also checks with other parts of the TTS that are not part of the tool to determine if their status is clear to proceed. The initial state of this module must be set when the tool is first used. There are thirteen tokens.

CT_Success No errors or potential problems detected. Clear to proceed.

CT_GTS_Error The GTS tool has not reported success.

CT_Info_Error The Info module set has not reported success.

CT_TH_Error The TH module set has not reported success.

CT_Multi_Ids More than one Id with the given name has been found in the symbol table.

CT_Unknown_EType The indicated EType is not known.

CT_Memory_Full Allocation of more memory has failed. The memory is full.

CT_Wrong_EType The EType encountered was not expected.

CT_No_Id The are no Ids with the given name in the symbol table.

CT_Bad_Boolean The value is not a boolean value.

CT_Bad_Grid_Num The value for a grid number has exceeded its bounds.

CT_Reject_Expr The expression is not admissible.

CT_Failure A non-specific problem occurred.

There are four access programs.

SetErrCT(CT_Token T **)** Sets the module to the CT_Token value of T.

CT_Token GetErrCT( ) Returns the CT_Token value currently maintained by the module.

char * GetStrErrCT(CT_Token T **)** Returns a pointer to a character string, which is the text message for the CT_Token value currently maintained by the module.

CT_Token CT_CheckStatus( ) Returns the CT_Token value maintained by the module after checking the status of the CT_Error module, the TH_Error module, the In_Error module, and the GTS_Error module.

4.5 Uses Hierarchy

Some programs belonging to a module call access programs. The Uses Hierarchies, herein, illustrate the uses relation between access programs. Table 4.3 provides the uses relation for the whole system. Figure 4.3 is the uses hierarchy for CT_Mid_CreateVectorComp. Figure 4.4 is the uses hierarchy for CT_Mid_CreateNormalFormComp. The arrows point from the calling program to the program being called and thus indicate uses. The vertical position of a program in the hierarchy is meaningful, in that programs which use no others must be at the bottom of the hierarchy, and programs which are at the same vertical position cannot use each other. If program A uses program B, then A must be above B in the hierarchy.

#	Program	Uses Programs #
1	CT_Mid_ Init	
2	CT_Mid_ Create VectorComp	4 5 6 7
3	CT_Mid_ CreateNormal FormComp	4 5 6 7
4	CT_AnCas_ AnalyseCase	
5	CT_Comp_ GenNF	8 9 10 11
6	CT_N2V_ NfToVector	
7	CT_V2N_ VectorToNf	
8	CT_Disj_ Comp	10
9	CT_NonDj_ Comp	10 11
10	CT_NoTab_ SubOSub	
11	CT_NoTab_ SubOComp	

Note: Every program can use any CT_Error module access program.

Table 4.3: The Uses Relation for the System

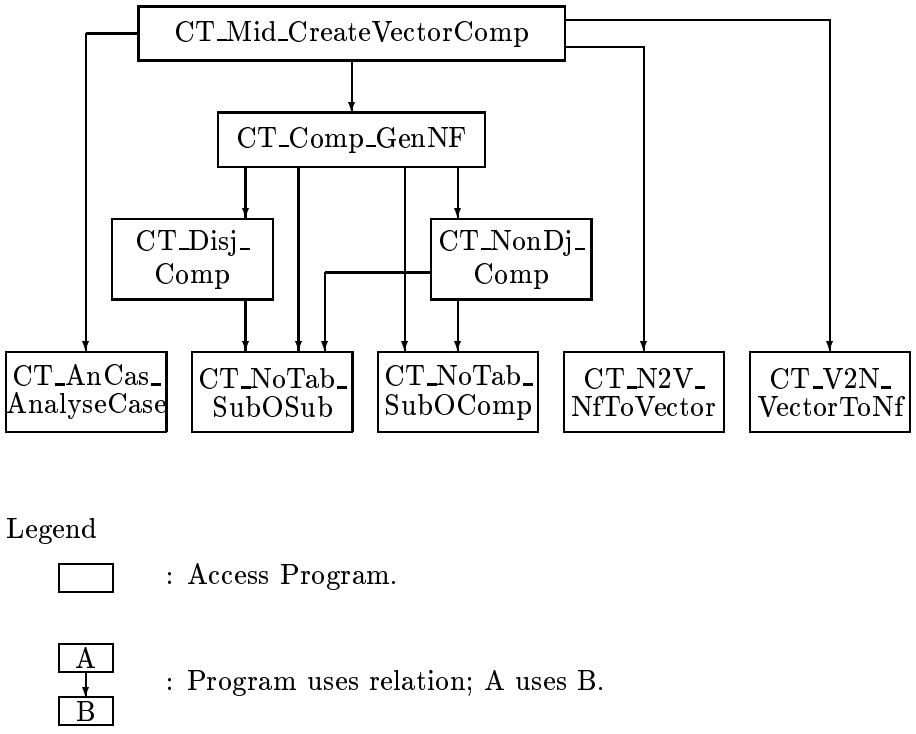


Figure 4.3: The Uses Hierarchy for CT_Mid_CreateVectorComp

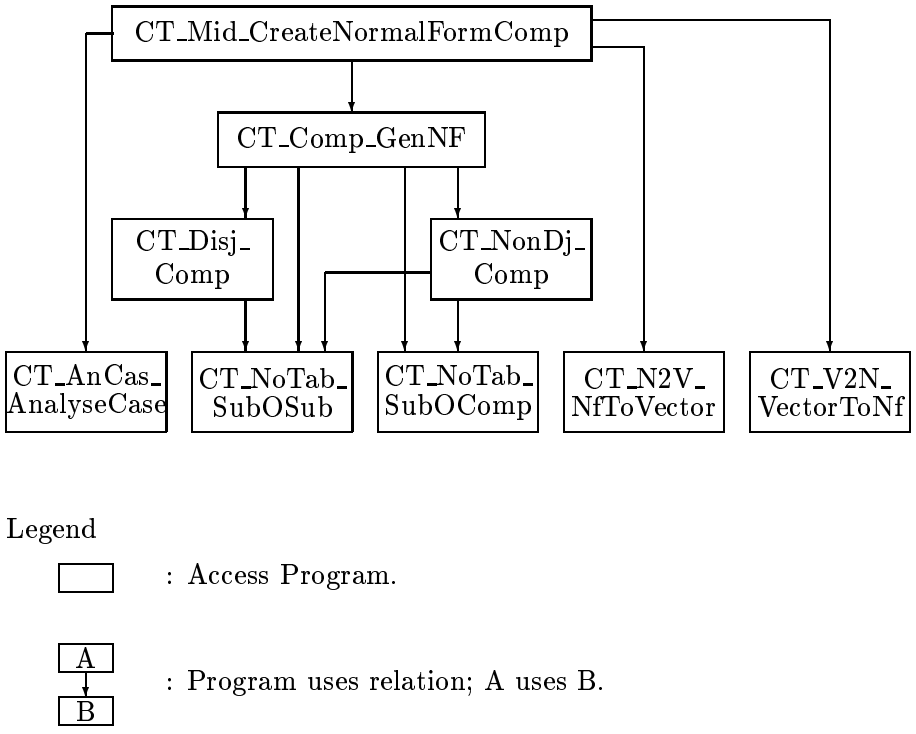


Figure 4.4: The Uses Hierarchy for CT_Mid_CreateNormalFormComp

4.6 Algorithms

This section is considered part of an *informal module internal design* document where the details enabling a module to meet its obligations are provided without using formal language. In addition to the algorithms, found in chapter 2, that define composition, additional algorithms are needed by the tool for higher level features, such as expression checking and table conversion. These algorithms are described herein.

4.6.1 Checking Expressions

An expression must be checked for acceptability, to the Function Composition Tool. The assumptions of section 4.1 require tabular expressions to be proper. The expression checking module does not verify properness.

The following is the algorithm for CT_AnCas_AnalyseCase. If the function does not reject the expression, then it is considered acceptable.

1. Count the instances of the tags QLETag, PdTableTag and FTableTag, in the expression. Reject the expression if there are any QLETag, or PdTableTag tags or if there is more than one FTableTag tag.
2. If the tag at the expression's root is not for a table, then, if the tag at the expression's root is not for a condition, then, if the expression is not a valid simultaneous substitution, then reject the expression.
3. Else the tag of the expression's root is for a table, then...
   (a) If GTS_checkShape applied to the expression returns BOOL_FALSE, indicating a problem between an expression's shape and semantics, then, reject the expression.
   (b) Else the shape of the expression agrees with the semantics, then...
      i. If the ccg of the expression's root is NORMAL, then, check the expression with CT_AnCas_CheckNormalForm. If the expression is not a valid normal function table, then reject the expression.
      ii. Else If the ccg of the expression's root is VECTOR, then, check the expression with CT_AnCas_CheckVector. If the expression is not a valid vector function table, then reject the expression.
      iii. Else the expression is not a useable table and reject it.

The following is the algorithm for CT_AnCas_CheckNormalForm. If the function does not reject the expression, then it is considered a valid normal function table.

1. For each header and the grid of the table...
   (a) If it is a guard header, (determined by GTS_semIsGuardGrid), then check, by verifying tags, that the expression in each cell is a condition, rejecting the table otherwise.
   (b) Else If it is the grid, (determined by GTS_semIsValueGrid), then check, by verifying tags, function names and arities, that the expression in each cell is a valid simultaneous substitution, rejecting the table otherwise.

The following is the algorithm for CT_AnCas_CheckVector. If the function does not reject the expression, then it is considered a valid vector function table.

1. Find the vector header of the table using GTS_semVectHeader.
2. If any expression in the vector header does not have tag VarTag, then reject the table.
3. For each header and the grid of the table...
   (a) If it is a guard header, (determined by GTS_semIsGuardGrid), then check, by verifying tags, that the expression in each cell is a condition, rejecting the table otherwise.

4.6.2 Converting Normal to Vector Function Tables

The functionality of the CT_Mid module requires that normal function tables be convertible to vector function tables. The CT_N2V_NfToVector function performs this task. This function does not check expressions.

The following is the algorithm for CT_N2V_NfToVector.

1. Create a set from the normal function table, which is the union, over every index into the grid, of the subsets rangevar(F), where F is the simultaneous substitution located by the index.

2. Create the Shape of the new vector function table. This Shape is that of the normal function table, but with a vector header added as the last header and with the vector header dimension and length added to the grid as the last dimension.

3. Construct the Name of the new vector function table. Use the new Shape to construct the name as described in the assumptions of section 4.1.

4. Produce the Id for the new vector function table. This is done by searching the symbol table for Ids with the constructed name.

   (a) If exactly one Id is found, then simply use that one.

   (b) Else If no Id is found, then create a new one with the constructed Name and appropriate table semantics.

   (c) Else an error condition has occurred. Set a status token indicating more than one Id has been found.

5. Create a new vector function table having tag FTableTag, the produced Id, and the created Shape.

6. Assign the variables from the created set of range variables to cells of the vector header, such that in the vector header, every variable is present and each cell contains exactly one variable and nothing else.

7. Copy the guard headers from the normal function table to corresponding guard headers of the vector function table.

8. Create and enter the grid expressions into the grid of the vector function table. These are created from the grid of the normal function table. The logical correspondence described in the definition of the vector function table, between the guard headers, vector header, and the grid must apply. The algorithm for CT_N2V_AssignMain accomplishes this.

9. Return the vector function table.

The following is the algorithm for CT_N2V_AssignMain. Some notation needs definition. The normal function table is $(H^1, \ldots, H^n, G)$, while the vector function table is $(H^1, \ldots, H^n, \bar{H}^{n+1}\bar{G})$, where $\bar{H}^{n+1} = \{\bar{h}_1^{n+1}, \ldots, \bar{h}_m^{n+1}\}$ is the vector header. This algorithm concerns the calculation of the elements of $\bar{G}$.

1. For each element $g_{\alpha_1,\ldots,\alpha_n}$ of the normal function table grid G...

(a) For each i from 1 to m along the vector header dimension of the vector function table grid...

i. If $\bar{h}_i^{n+1} = x_j$, for some j where x_j is from the simultaneous substitution, $(x_1, \ldots, x_m) \mapsto (e_1, \ldots, e_m)$, of the normal function table grid element, $g_{\alpha_1,\ldots,\alpha_n}$, then the vector function table grid element, $\bar{g}_{\alpha_1,\ldots,\alpha_n,i} = e_j$, the term of the simultaneous substitution.

ii. Else the vector function table grid element, $\bar{g}_{\alpha_1,\ldots,\alpha_n,i} = \bar{h}_i^{n+1}$, the variable of the vector header.

4.6.3 Converting Vector to Normal Function Tables

The functionality of the CT_Mid module requires that vector function tables be convertible to normal function tables. The CT_V2N_VectorToNf function performs this task. The function does not check expressions.

The following is the algorithm for CT_V2N_VectorToNf.

1. Find the vector header of the vector function table using GTS_semVecHeader.
2. Create the Shape of the new normal function table. This Shape is that of the vector function table, but without the vector header and without the vector header dimension in the grid.
3. Construct the Name of the new normal function table. Use the new Shape to construct the name as described in the assumptions of section 4.1.
4. Produce the Id for the new normal function table. This is done by searching the symbol table for Ids with the constructed name.
   (a) If exactly one Id is found, then simply use that one.
   (b) Else If no Id is found, then create a new one with the constructed Name and appropriate table semantics.
   (c) Else an error condition has occurred. Set a status token indicating more than one Id has been found.
5. Create a new normal function table having tag FTableTag, the produced Id, and the created Shape.
6. Copy the guard headers from the vector function table to corresponding guard headers of the normal function table.

7. Create and enter the grid expressions into the grid of the normal function table. These are created from the vector header and grid of the vector function table. The logical correspondence described in the definition of the normal function table, between the guard headers and the grid must apply. The algorithm for CT_V2N_AssignMain accomplishes this.

8. Return the normal function table.

The following is the algorithm for CT_V2N_AssignMain. Some notation needs definition. The vector function table is $(H^1, \ldots, H^v, \ldots, H^n, G)$, while the normal function table is $(H^1, \ldots, H^{v-1}, H^{v+1}, \ldots, H^n, \bar{G})$, where $H^v = \{h_1^v, \ldots, h_m^v\}$ is the vector header. This algorithm concerns the calculation of the elements of $\bar{G}$.

1. Create an Expn, a simultaneous substitution, e, where in infix form, the right hand side is the tuple $(t_1, \ldots, t_m)$ where the t_i are terms to be defined, and the left hand side is the tuple $(h_1^v, \ldots, h_m^v)$.

2. For every Index, $(\alpha_1, \ldots, \alpha_{v-1}, \alpha_{v+1}, \ldots, \alpha_n)$, into the grid, $\bar{G}$, of the normal function table...

   (a) For every $\alpha_v \in I^v$, the term of e, $t_{\alpha_v} = g_{(\alpha_1,\ldots,\alpha_{v-1},\alpha_v,\alpha_{v+1},\ldots,\alpha_n)}$, in the grid, G, of the vector function table.

   (b) The normal function table grid, $\bar{G}$, element, $\bar{g}_{(\alpha_1,\ldots,\alpha_{v-1},\alpha_{v+1},\ldots,\alpha_n)} = e$, the simultaneous substitution.

4.7 Implementation Issues

This section contains ideas used in the implementations of the modules of the Function Composition Tool. Thus it is part of the module internal designs documentation.

4.7.1 Private Data Structure

The implementation of the algorithm for the composition of simultaneous substitutions, in the CT_NoTab module, does not leave the expressions in the table

holder as expression trees, as it might, but rather translates the expressions into a private data structure first. After the composition is performed, the resulting expression is translated back to the table holder again. The new data structure is a list where each node contains the range variable and the term of the same sort that is to be assigned to the variable.

The reason a private data structure is used, is that the arity of expressions in the table holder cannot be extended. Instead a new expression must be created to replace the smaller one. A simultaneous substitution as a list, can be easily extended to include more range variables. Since the list must be searched for the presence of certain range variables, however, it is not the optimal data structure for the application.

4.7.2 Data Structure Guidelines

In general, if a data structure must be used, the implementation attempts to use the data structures of existing modules in preference to the creation of new private data structures. Modules constructed in this way, have as their secret the fact that they delegate their data structures to another module. Thus this paradigm avoids data structure code, removing a significant source of error. Only the CT_NoTab, and CT_Error modules violate the paradigm.

Modules are designed to make the use of traces unnecessary. The only canonical traces the paradigm allows is the empty trace and the trace composed of only the last access program call. Thus the implementation of a module can avoid the complexities of the interaction of access programs on the data structures. For example, a module not following the paradigm might have access programs to create and destroy an object and to add or remove information contained by it. Such a module can have complex traces with the possibility of many conflicts between access programs. Only the CT_Error module violates the paradigm.

4.7.3 Status Checking Guidelines

The CT_Error module maintains a token indicating the status of the tool. When the tool is initialized, the token is set to CT_Success. At various points in the tool's code, the token may be set to something other than CT_Success, indicating a change of status. The CT_Error module follows the paradigm that in general, a change in status is taken to be the detection of a potential error. In designing the

tool, tokens have been invented as needed to indicate the nature of a problem.

The TTS modules external to the tool also use tokens to indicate status. A check is done on the status of any external module used. Only the success tokens for these modules are known explicitly by the CT_Error module. On receiving any status token from an external module which is not success, the CT_Error module sets a token pointing out which module reported the problem.

4.7.4 Code Readability Guidelines

In coding a function for the tool, checking the status of called functions requires the addition of code, which is not part of the algorithm code. Typically the lines of code for status checking, significantly outnumber the lines of algorithm code. The readability of the code can be seriously eroded, even for the programmer, resulting in errors that might otherwise have been noticed. The paradigm of separating status checking code from algorithm code can preserve readability.

The term *armor* refers to performing all status checks at only the beginning and possibly at the end of each function. The algorithm code remains in an undisturbed form, without having the status checking code dispersed through it. The armor checks the status of relevant tokens at the beginning of a function. Only if all responses indicate success, will the algorithm code for the function be performed. The success of armor at catching errors depends on the algorithm code. The code of the tool has been implemented with armor, except for the top level access programs.

4.7.5 Symbol Naming Scheme

Symbols are searched for and identified by their name. The Function Composition Tool uses a naming scheme for simultaneous substitution operators, ordered n-tuple operators, and normal and vector function tables. The user can input tables which have names that do not follow the naming scheme, but the tool will not reuse those table Ids, since they will not be found. The maximum length of a name is set by a constant, $NAMELENGTH$. Name are constructed in the following way:

Simultaneous substitution name The constant string represented by $SIMULTANEOUS\_SUBSTITUTION$.

Ordered n-tuple name The concatenation of the constant string prefix, represented by $ORDERED\_TUPLE$, with a digit string for n. The maximum length of the digit string is represented by $ARITYSTRINGLENGTH$.

Normal function table name The concatenation of the constant string prefix, represented by $NFTABLENAMEPREFIX$, with a digit string for the number of guard headers in the table plus one. The maximum length of the digit string is represented by $NUMGRIDSTRLEN$.

Vector function table name The concatenation of the constant string prefix, represented by $VECTABLENAMEPREFIX$, with a digit string for the number of guard headers in the table plus two. The maximum length of the digit string is represented by $NUMGRIDSTRLEN$.

Chapter 5

Testing Tools

This chapter includes two test case generators for the Function Composition Tool. The TTS has two input tools, but neither were finished development at the time of this work. A Function Composition Tool test harness, is included that was used for the processing of sets of test cases. The Function Composition Tool has since been integrated into the TTS, which can allow interactive test case processing.

5.1 The MapToTts Translator

Prior to the designing of the Function Composition Tool, an expression translator that generates TTS expressions from text, was written to support TTS research. This translator is known in the TTS as *MapToTts* and is part of the kernel utilities library. The MapToTts Translator is a key part of the generation of test cases for the Function Composition Tool.

Expressions in the TTS are represented as tree structures rather than strings of text. A text string, however, is frequently a convenient means of representing mathematical expressions. Symbolic mathematics packages, such as the Maple [17] [18] system, generally accept lines of text as input, since it is a conventional way for the user to enter data. The purpose of the MapToTts Translator is to convert a text string having a simplified Maple syntax, into a TTS expression. Details concerning the MapToTts Translator comprise Appendix A.

To generate a test case for the Function Composition Tool, in which the

two expression arguments are simultaneous substitutions or conditions, two text string representations and a symbol table are used. The arguments of the MapToTts Translator are the symbol table and a text file. The text file contains a single text string, thus the MapToTts Translator must be used twice for this kind of test case. Translation results in the TTS representation of the expressions.

To generate a test case for the Function Composition Tool, in which one or more of the expression arguments is a table, requires more functionality than the MapToTts Translator provides. The MapToTts Translator does not generate tables, but since the tables used with the Function Composition Tool do not contain tables, the MapToTts Translator can be used.

5.2 The ConSimSub Generator

The Condition and Simultaneous Substitution (ConSimSub) Generator produces conditions and simultaneous substitutions, suitable as arguments to the Function Composition Tool. It uses the MapToTts Translator, to produce expressions from an appropriate text string. Since a symbol table is required, it is convenient to use a context. A *context* is a set of TTS expressions and a symbol table. Contexts are supported by the higher modules of the TTS. All expressions in the context share the symbol table. Thus the ConSimSub Generator, can produce many expressions, using one symbol table and the MapToTts Translator.

The generator assumes a human user, which allows for interactive use. The user is queried for names and actions. In general, a context is modified by the generator, and thus when the context is to be saved, a new name can be used. The original context file will then remain unchanged.

The following is the algorithm for the ConSimSub Generator utility.

1. Call MT_Init() of the MapToTts Translator and initialize any other required modules. It is recommended to use the Index, Shape, Path, Expn, Info, ErrUI, SymUtil, and CReg modules.
2. Open for reading, the user indicated context file, $file\_A$, containing the symbol table to use with the MapToTts Translator.
3. Load the context from $file\_A$.
4. Get the symbol table, t, from the context.

5. Determine if the user wishes to supply the text string to translate from a file, or from standard input.
6. If the user chooses to supply a file containing the text string, then ...
   (a) Open for reading, the user indicated file, assigning it to file pointer, $file$.
   (b) Call MapleToTts(t, $file$), assigning the returned expression to e.
7. Else
   (a) Get the text string, *input*, from standard input.
   (b) Open for writing, a temporary file, and write *input* into the file.
   (c) Reopen the same temporary file for reading, assigning it to file pointer $file$.
   (d) Call MapleToTts(t, $file$), assigning the returned expression to e.
8. Display the expression e using symbol table t.
9. Determine from the user the name, *name*, to use in referring to the expression e within the context.
10. Add the expression e with the name *name* into the 0th position of the context, thus modifying the context.
11. Open for writing, the user indicated context file, $file\_B$, and write the modified context into it.
12. If ErrHndlGet() of the ErrUI module, does not return TIF_Success, then indicate a TIF problem.
13. If CheckStatus() does not return MT_success, then indicate a translator problem.
14. Destroy e and close all files.

5.3 The NorVecTab Generator

The Normal and Vector Function Table (NorVecTab) Generator is used to generate tabular expressions suitable as input to the Function Composition Tool. Similarly to the ConSimSub Generator, a context is used to contain both the

symbol table on which expressions are based, and the potentially large number of expressions that are generated. In contrast, since the MapToTts Translator does not produce tables, the utility must provide the functionality of creating tables and their names and assigning expressions to the cells of a table. The MapToTts Translator generated table cell expressions can be produced from text strings either from a text file or from standard input. In addition, any expression, which is already present in the context, may be placed in a cell. Thus a cell of a table may be assigned a table which was created earlier by the utility.

The NorVecTab Generator assumes a human user. The user is queried for names and actions. In general, a context is modified by the NorVecTab Generator, and thus when the context is to be saved, a new name can be used. The original context file will then remain unchanged.

The following is the algorithm for the NorVecTab Generator. The utility assigns table semantics to Ids of tables in the symbol table. In particular, the cell connection graph, table predicate rule, and table relation rule class entries are filled. More information about table semantics can be found in [14] [7] [6].

1. Call MT_Init() of the MapToTts Translator and initialize any other required modules. It is recommended to use the Index, Shape, Path, Expn, Info, ErrUI, SymUtil, and CReg modules.
2. Open for reading, the user indicated context file, $file\_A$, containing the symbol table to use with the MapToTts Translator.
3. Load the context from $file\_A$.
4. Get the symbol table, t, from the context.
5. Get a name, *tabnam*, from the user for the new table within the context.
6. Query the user for the necessary data to define the table shape. The data would be the number of grids, and the number and length of dimensions for each grid.
7. Display the constructed shape, s, of the table.
8. Convert the number of grids specified in s into a string of digits, *strnum*.
9. Determine from the user if the table is to be a normal function table, or a vector function table.
10. If the table is a normal function table, then ...

(a) Concatenate the strings "nf" and *strnum*, assigning the result to *name*.

(b) Call getnfid(*t*, *name*, *s*), assigning the returned Id to *id*.

11. Else

(a) Concatenate the strings "vec" and *strnum*, assigning the result to *name*.

(b) Query the user as to which header of the table will be the vector header, assigning this value to *vecgrid*.

(c) Call getvecid(*t*, *name*, *s*, *vecgrid*), assigning the returned Id to *id*.

12. Create the table expression, *e*, with EType FTableTag, Id *id*, and Shape *s*.

13. For each header and the grid of *e*, do ...

(a) For each valid Index, *i*, do ...

i. Display *i* to the user.

ii. Determine if the expression, which will be located at *i*, is to be supplied as a text string in a text file, is in the current context already, or is to be supplied as a text string from standard input.

iii. If the expression is supplied by a text file, then ...

A. Query the user for the name of the text file and assign it to *input*.

B. Open the file for reading, with the name *input*, assigning it file pointer *file*.

C. Call MapleToTts(*t*, *file*), assigning the returned expression to *cell*.

D. Display *cell* to the user.

E. Assign *cell* to table *e* at *i*.

F. Destroy *cell*

iv. Else if the expression is in the context already, then ...

A. Query the user for the name of the expression in the context and assign the name to *input*.

B. Get the expression with name *input* from the context and assign it to *cell*.

C. Display *cell* to the user.

D. Assign *cell* to the table *e* at *i*.

v. Else ...

A. Query the user for the text string input and assign it to *input*.

B. Open a temporary file for writing and write *input* to the file.

C. Reopen the temporary file for reading and assign it to file pointer *file*.

D. Call MapleToTts(*t*, *file*), assigning the returned expression to *cell*.

E. Display *cell* to the user.

F. Assign *cell* to the table *e* at *i*.

G. Destroy *cell*.

14. Display *e* to the user.

15. Add *e* with name *tabnam* to the 0th position of the context.

16. Open for writing, the user indicated context file, *file_B*, and write the modified context into it.

17. If ErrHndlGet() of the ErrUI module, does not return TIF_Success, then indicate a TIF problem.

18. If CheckStatus() does not return MT_success, then indicate a translator problem.

19. Destroy *e* and close all files.

The algorithm above calls two functions which find the Id of a table. Namely, these functions are Id getnfid(SymTbl *t*, char *name*[31], Shape *s*) and Id getvecid(SymTbl *t*, char *name*[31], Shape *s*, int *vecgrid*). The names of the variables have been preserved from the calling algorithm for reference.

The following is the algorithm for the function Id getnfid(SymTbl *t*, char *name*[31], Shape *s*) .

1. Search for all symbols in *t* with Name class entry *name*.

2. If there is only one such symbol, then assign its Id to *id*.

3. Else if there are no such symbols, then ...

   (a) Create an Id, *id*, in *t*.

   (b) Assign *name* to the Name class entry of *id*.

   (c) Assign 7 to the Tag class entry of *id* indicating the EType FTableTag.

   (d) Assign default information to *id*.

(e) Assign font information to *id*.

(f) Assign cell connection graph, table predicate rule, and table relation rule information to *id*, given that the table is a TTS normal function table and has Shape *s*.

4. Else assign -1 to *id*.

5. Return *id*.

The following is the algorithm for the function Id getvecid(SymTbl *t*, char *name*[31], Shape *s*, int *vecgrid*).

1. Search for all symbols in *t* with Name class entry *name*.

2. If there is only one such symbol, then assign its Id to *id*.

3. Else if there are no such symbols, then ...

(a) Create an Id, *id*, in *t*.

(b) Assign *name* to the Name class entry of *id*.

(c) Assign 7 to the Tag class entry of *id* indicating the EType FTableTag.

(d) Assign default information to *id*.

(e) Assign font information to *id*.

(f) Assign cell connection graph, table predicate rule, and table relation rule information to *id*, given that the table is a TTS vector function table, has a Shape *s*, and header number *vecgrid* as the vector header.

4. Else assign -1 to *id*.

5. Return *id*.

5.4 Function Composition Tool Test Harness

The test harness utility is intended to be used for performing a set of tests on the Function Composition Tool. Each test is an example problem of composition. The set of tests is referred to as a *test suite*, which is in the form of a context. At input, each test in the suite contains two expressions for composition

and the expected result. If the test run completes successfully, the actual result for each test case will also be included in the context. The test harness automatically determines if the expected and actual results are the same.

The test harness assumes certain names for the expressions in the context. For the ith test in the suite, the name of the left hand expression in the composition is "left$i$". Similarly, the right hand expression is named "right$i$". The expected result and the actual result are named "result$i$" and "test$i$" respectively. Thus the user must be cognoscente of the names and number of test cases when creating a test suite with the ConSimSub Generator or the NorVecTab Generator.

The test harness assumes a human user, and allows interactive use. In general, a context is modified by the test harness, and thus when the context is to be saved, a new name can used. The original context file will then remain unchanged. The test harness does not determine the number of test cases in a suite directly, but simply prompts the user for that data. Messages of success or failure of specific cases are written into a log file. The name of the log is determined by the user. Since there are two forms of tabular output from the Function Composition Tool, the test harness queries the user to decide if tabular output should be normal or vector. This decision will be applied to all actual results of the test suite. If the form of the expected and actual results is not the same, the test harness will not consider them equal and will report a failure.

The following is the algorithm for the Function Composition Tool test harness.

1. Call CT_Init() and initialize any other required modules. It is recommended to use the Index, Shape, Path, Expn, Info, ErrUI, SymUtil, and CReg modules.
2. Open for reading, the user indicated context file, $file\_A$, which is the input test suite.
3. Load the context from $file\_A$.
4. Get the symbol table, t, from the context.
5. Query the user for the number of test cases in the test suite, assigning the value to $numtests$.
6. Open for reading, the user indicated log file, $file\_C$, to which success or failure statements can be written for each test case.
7. Query the user for the form, either normal or vector, of the actual results, in order to match the form of the expected results.

8. For i equals 1 to $numtests$, do ...
   (a) Convert i to a string of digits, $strdig$.
   (b) Get the expression for the context whose name is the concatenation of "left" with $strdig$, assigning it to $left$.
   (c) Display $left$ to the user.
   (d) Get the expression for the context whose name is the concatenation of "right" with $strdig$, assigning it to $right$.
   (e) Display $right$ to the user.
   (f) Get the expression for the context whose name is the concatenation of "result" with $strdig$, assigning it to $result$.
   (g) Display $result$ to the user.
   (h) If all actual results should be normal function tables, then call CT_Mid_CreateNormalFormComp(t, $left$, $right$), assigning the returned expression to $test$.
   (i) Else call CT_Mid_CreateVectorComp(t, $left$, $right$), assigning the returned expression to $test$.
   (j) Display $test$ to the user.
   (k) Add at the 0th position in the context, the expression $test$, using the name being the concatenation of "test" and $strdig$.
   (l) If comparing the expression $result$ and the expression $test$ shows they are exactly the same, then write a statement to the log file indicating the specific test produced the expected result.
   (m) Else write a statement to the log file indicating the specific test did not produce the expected result.
9. Open for writing, the user indicated context file, $file\_B$, and write the modified context into it.
10. If ErrHndlGet() of the ErrUI module, does not return TIF_Success, then indicate a TIF problem.
11. If CheckStatus() does not return MT_success, then indicate a translator problem.
12. Close all files.

Chapter 6

Examples

Any testing effort on the Function Composition Tool should recognize the several distinct kinds of composition problem. These kinds are based on the composition algorithms themselves. Since there are two access functions, CT_Mid_CreateNormalFormComp, in which tabular results are normal function tables and CT_Mid_CreateVectorComp, in which tabular results are vector function tables, test problems can be submitted using each. This section contains some examples of composition problems which the Function Composition Tool can perform. Since the tool is not designed with a graphical display in mind, the examples have been reformatted for readability.

6.1 Composition with only Simultaneous Substitutions and Conditions

In this situation there are at least three issues to consider.

1. A simultaneous substitution composed with a condition.
2. The composition of two simultaneous substitutions.
3. Both access programs should produce exactly the same expression, since the result is either a condition or a simultaneous substitution.

For the composition problem in Table 6.1, a, b, x, y, z are variables and the $\circ$ symbol is composition. The right hand side is a condition and the left hand side is

a simultaneous substitution.

For the composition problem in Table 6.2, F and G are simultaneous substitutions, with variables u, v, x, y, z and constants A and B.

$$\left\{ \begin{array}{l} x \mapsto a+1, \\ y \mapsto 2*b, \\ z \mapsto a/b \end{array} \right\} \circ ((x*y*z) < 2)$$

$$= (((a+1)*(2*b)*(a/b)) < 2)$$

Table 6.1: Composition of a simultaneous substitution and a condition

$$F = \left\{ \begin{array}{l} x \mapsto x+1, \\ y \mapsto v/2 \end{array} \right\}$$

$$G = \left\{ \begin{array}{l} x \mapsto -u, \\ y \mapsto x*y+B, \\ z \mapsto x-A \end{array} \right\}$$

$$G \circ F = \left\{ \begin{array}{l} z \mapsto x-A, \\ x \mapsto (-u)+1, \\ y \mapsto v/2 \end{array} \right\}$$

$$F \circ G = \left\{ \begin{array}{l} x \mapsto -u, \\ y \mapsto (x+1)*(v/2)+B, \\ z \mapsto (x+1)-A \end{array} \right\}$$

Table 6.2: Composition of two simultaneous substitutions

6.2 Composition where Only One Expression is a Table

In this situation there are at least three issues to consider.

1. A normal or vector function table composed with a simultaneous substitution
2. A simultaneous substitution composed with a normal or vector function table
3. The problem should be tested on both access functions, since the resulting expressions are not exactly the same

In Table 6.3 the composition of a vector function table and a simultaneous substitution results in a vector function table with one guard header by the use of CT_Mid_CreateVectorComp. In this example, F is a simultaneous substitution, G is a vector function table, v, x, y, z are variables and A, B are constants.

In Table 6.5 the composition of a simultaneous substitution and a normal function table results in a normal function table by the use of CT_Mid_CreateNormalFormComp. In this example, F is a normal function table, G is a simultaneous substitution, v, x, y, z are variables and A, B are constants.

$$F = \left\{ \begin{array}{l} x \mapsto Ay + B \\ z \mapsto xy \end{array} \right\}$$

$$G = \begin{array}{|c||c|c|} \hline & y < 0 & y \geq 0 \\ \hline\hline x & xyz & v - x \\ \hline y & By & z + x \\ \hline \end{array}$$

$$G \circ F = \begin{array}{|c||c|c|} \hline & y < 0 & y \geq 0 \\ \hline\hline y & By & z + x \\ \hline x & A(By) + B & A(z + x) + B \\ \hline z & (xyz)(By) & (v - x)(z + x) \\ \hline \end{array}$$

Table 6.3: Composition with One Vector Function Table on Left

$F =$

	$y < 0$	$y \geq 0$
$x + y = B$	$\{x \mapsto xyz\}$	$\{y \mapsto v - x\}$
$x + y \neq B$	$\{y \mapsto Bz\}$	$\{x \mapsto z + x\}$

$$G = \left\{ \begin{array}{l} x \mapsto Ay + B \\ z \mapsto xy \end{array} \right\}$$

$G \circ F =$

	$y < 0$	$y \geq 0$
$(Ay + B) + (xy) = B$	$\{z \mapsto xy,$ $x \mapsto (Ay + B)y(xy)\}$	$\{x \mapsto Ay + B,$ $z \mapsto xy,$ $y \mapsto v - x\}$
$(Ay + B) + (xy) \neq B$	$\{x \mapsto Ay + B,$ $z \mapsto xy,$ $y \mapsto B(xy)\}$	$\{z \mapsto xy,$ $x \mapsto (xy) + (Ay + B)\}$

Table 6.4: Composition with One Normal Function Table on Right

6.3 Composition of Two Normal or Vector Function Tables

In this situation there are at least three issues to consider.

1. Input tables can be either normal or vector independently.
2. Separate algorithms exist for disjoint and nondisjoint tables.
3. Both access functions should be tested, since the resulting expressions are not exactly the same.

For more than trivial examples, manual inspection of test results is impractical due to the occurrence of tables with three or more dimensions. The automatic checking of the expected and actual results done by a single function call to the Expn module function, ExpnEqualSub, makes the use of test suites practical.

In Table 6.5 the composition of disjoint normal and vector function tables results in a normal function table by the use of CT_Mid_CreateNormalFormComp. In this example, F is a vector function table, G is a normal function table, v, w, x, y, z are variables and A, B are constants.

In Table 6.6 the composition of nondisjoint vector and normal function tables results in vector function table with one guard header by the use of CT_Mid_CreateVectorComp. In this example, F is a normal function table, G is a vector function table, v, w, x, y, z are variables and A, B are constants.

$$F = \begin{array}{|c||c|c|}\hline & w < 0 & w \geq 0 \\ \hline\hline y & 2y & v + x \\ \hline z & x & y \\ \hline\end{array}$$

$$G = \begin{array}{|c|c|}\hline A = 1 & A \neq 1 \\ \hline \{x \mapsto y\} & \{x \mapsto B, \\ & y \mapsto x\} \\ \hline\end{array}$$

$$G \circ F = \begin{array}{|c||c|c|}\hline & A = 1 & A \neq 1 \\ \hline\hline w < 0 & \{x \mapsto y, \\ & y \mapsto 2 * y, \\ & z \mapsto y\} & \begin{array}{c}\{x \mapsto B, \\ y \mapsto 2 * x, \\ z \mapsto B\}\end{array} \\ \hline w \geq 0 & \{x \mapsto y, \\ & y \mapsto v + y, \\ & z \mapsto y\} & \begin{array}{c}\{x \mapsto B, \\ y \mapsto v + B, \\ z \mapsto x\}\end{array} \\ \hline\end{array}$$

Table 6.5: Composition of Disjoint Tables

$$F = \begin{array}{|c|c|}\hline w < x & w \geq x \\ \hline \{y \mapsto 2 * y, & \{y \mapsto v + x, \\ z \mapsto x\} & z \mapsto y\} \\ \hline\end{array}$$

$$G = \begin{array}{|c||c|c|}\hline & A = 1 & A \neq 1 \\ \hline\hline x & y & B \\ \hline y & y & x \\ \hline\end{array}$$

$$G \circ F = \begin{array}{|c||c|c|c|c|}\hline & A = 1 \wedge w < y & A = 1 \wedge w \geq y & A \neq 1 \wedge w < B & A \neq 1 \wedge w \geq B \\ \hline\hline x & y & y & B & B \\ \hline y & 2 * y & v + y & 2 * x & v + B \\ \hline z & y & y & B & x \\ \hline\end{array}$$

Table 6.6: Composition of Nondisjoint Tables

Chapter 7

Results and Conclusions

This chapter summarizes the results of designing and implementing the Function Composition Tool. In particular, limitations are noted and future developments suggested. Some conclusions are drawn from the overall effort.

7.1 Results

The Function Composition Tool is a significant contribution to the TTS. Initial considerations suggested that composition could best be accomplished with the use of a commercial symbolic engine. The Maple [18] [17] system was a clear option, which explains the use of Maple syntax in the MapToTts Translator. The later publication of the technical report [8], provided a way for composition to be performed within the tool. The restriction on tabular expressions to normal function tables was extended by additional algorithms to include vector function tables. At the time this document was written, the tool has been undergoing integration into the TTS and given a graphical interface.

This work has provided a second significant contribution to the TTS. The MapToTts Translator was originally conceived as part of the Function Composition Tool. It has shown that converting Maple syntax text strings into TTS expressions is useful, not only in generating expressions for composition, but also to at least one other TTS effort, namely, the SAST [15] tool. The MapToTts Translator has become part of the TTS kernel utilities module for general use. Its documentation is Appendix A.

Together with the translator, the ConSimSub Generator and the NorVecTab Generator have allowed the Function Composition Tool operation to be illustrated for several kinds of composition problem. These test cases compose a small set of test suites used in the development of the tool.

7.2 Limitations and Future Developments

The limitations of the Function Composition Tool are generally the result of the need to reduce the scope of the problem. The tool accepts only conditions, simultaneous substitutions (simultaneous assignments), proper normal function tables, and proper vector function tables. This places a burden on the user, which could be reduced by the development a utility for the conversion of other kinds of expressions to these. The tool cannot use predicate tables or quantified logical expressions. No tables may be nested within another expression. This is a barrier to the potential usefulness of the tool as described in the introduction and a clear target for future research.

The early decision to use Maple was abandon after some investigation. The strengths of Maple are not required for the simple composition problem. Implementing the composition engine directly requires less code than if Maple were used which results in a faster process. This difference is due partly to the absence of the interface between the tool and Maple. The difficult part is the translation from Maple to the TTS, since it would use the lex and yacc unix facilities. Lex and yacc are complex general systems for lexical and syntax analysis and tend to be sensitive to changes both in their rule systems and to the compiler that uses them. This sensitivity causes problems in the maintenance of such a system. Currently, Maple cannot help with the problem of nested tables and the difficulty of dealing with quantified expressions becomes greater within Maple.

To perform substitution on a quantified expression, two concerns must be met. Firstly, it must be possible to determine if a variable is free, (not bound by a quantifier), since substitution occurs for only free variables. Secondly, when substituting for a free variable in the scope of a quantifier, it must be certain that the term does not carry a variable of the same name as the variable which is bound. If it does, then a replacement must be found for the bound variable. The method of replacement must be systematic if the new expression is to be equivalent to the original. (Otherwise it is α-equivalent.) The replacement variable should not have already occurred in the expressions for composition since a used name is likely physically meaningful to the user. Adding to the problem is the need to make the

new variable name acceptable. There may be physical or aesthetic choices to be made, which might require user interaction or user predefined defaults. This puts much stronger requirements on the user interface. The difficulties inherent in dealing acceptably with quantified expressions lead to the practical decision not to support them with the tool in order limit the duration of this research.

A nested table is one which occurs within an expression. Composition of tables eventually reduces to a sequence of compositions of expressions that appear in the tables. With nested tables this reduction implies recursive calls to the Function Composition Tool. For the tool to handle nested tables, at least five specific issues should be dealt with.

Modifications are needed to the composition engine. In the composition of two simultaneous substitutions, if the right hand side expression has terms with nested tables, then substitution needs to be restricted. For example, if F is the expression $x \mapsto t$, which is a substitution, it is important that the composition engine acting on F, not substitute for the x that appears in the left hand side of F. Substitution must include a check for this pitfall. Since the nesting of tables can be recursive, there could be recursive calls to substitution.

In the composition of two tables, if the left hand side table contains a table as a grid element, then it must be possible to take the composition of a table and a condition. To perform this kind of a composition problem would require additional definitions or theory for a new algorithm. Previously, the composition of a proper normal function table and a simultaneous substitution was again a proper normal function table, and this was called closure. It is the author's opinion that the composition of a proper normal function table and a condition is some form of predicate table, not a normal function table. This problem requires investigation.

If nested tables occur in guard headers, it would be expected that in general these tables would be predicate tables. In the composition of two tables, if the left hand side table has predicate tables nested in the guard headers, then it must be possible to take the composition of a simultaneous substitution and a predicate table. This problem requires support from definitions and theory. It is the author's opinion that the result would be again a predicate table. A proof of the apparent closure would be necessary and then a new algorithm could be developed.

Presently the Function Composition Tool supports proper normal function tables and proper vector function tables. Accepting nested tables opens the possibility that the user may submit tables nested with both normal and vector tables. Thus the table conversion algorithms must be recursive to convert all tables to the normal table form the tool uses internally.

Currently, at most one table occurs in the results from the tool. Thus the user can easily choose the form of the table by selecting between two access programs; one for normal table output and another for vector table output. In the case of nested tables the user may not want all tables to be in the same format. It is necessary to allow user interaction or user predefined defaults to decide what the forms of the nested tables should be. This point becomes more important as more forms of tables are supported and puts much stronger requirements on the user interface.

Clearly the effort to support nested tables is significant. The difficulties inherent in dealing acceptably with nested tables lead to the practical decision not to support them with the tool in order limit the duration of this research. Partial support was rejected in favor of keeping the topic intact for future research.

The assumptions in section 4.1 of the Function Composition Tool and those of the MapToTts Translator in section A.3 indicate the importance of the names of symbols. The Id of a symbol is located primarily by its Name class entry in a symbol table. Tables are identified by a name, which is dependant on shape. As a result, the tool assumes that both input expressions are members of the same context. It is likely that the result of a physical variable having two different names will occur due to its appearance in two different contexts. The composition of expressions from different contexts suggests the need of a means of identifying names. In this way the tool would not duplicate the appearance of variables.

If this research were to be repeated, it is suggested that initial efforts focus on the development of a name management system. This would be useful in the problem of substitution into quantified expressions. Secondly, development of the Function Composition Tool should begin with a substitution engine, since composition uses substitution as a basic process. The substitution engine should be designed to handle quantified expressions and nested tables. In this way, it would be easier to extend composition.

7.3 Conclusions

The Function Composition Tool can perform automated composition within the TTS. It does not use a commercial external symbolic engine, eliminating licensing concerns, making source code readily available, and avoiding the intricacies of the interface to the external software. Familiar mathematical composition is realized using the standard definition of the composition of simultaneous

substitutions found in mathematical logic. The implementation in C language is simple and can be fast. Thus the tool is a good prototype for additional research on composition of TTS expressions where quantification and nested tables are involved.

In addition to contributing toward the TTS, this work is an example of the application of software documentation ideas. This experience has shown the importance of documentation to the individual developer, since it is difficult for the programmer to recall the exact behavior of a module written only a few months ago. Thus usefulness of software is dependant on the continued readability of its documentation to a wide variety of individuals over its developmental lifetime.

Appendix A

Conventional Expression Translator

The translator, known as *MapToTts*, is contained in the libraries of the TTS. It recognizes a subset of Maple [17] [18] syntax, enabling conventional expressions to be generated within the TTS from a text file supplied by the user. On this file, the translator performs lexical analysis, syntax analysis , and carries out actions within the TTS. The lexical analysis identifies the words from which the TTS expressions are to be built, while the syntax analysis recognizes patterns between the words. The actions carried out as a result, build expressions using the Table Holder and the Information Module of the TTS.

A.1 Lexical Analysis Rules

The following list describes the words, which are the sequences of characters, recognized by the translator. This information is part of the requirements specification for the translator.

,

A comma, used as a separator.

;

A semicolon, used to end the text string.

`/`

A forward slash, as division.

`.`

A period, as the decimal point.

`=`

The equality sign.

`>`

The greater-than sign.

`<`

The less-than sign.

`(`

A left parenthesis, as a left delimiter.

`-`

The minus sign.

`+`

The plus sign.

`^`

A circumflex, used to indicate an exponent.

`*`

An asterisk, used to represent multiplication.

`)`

A right parenthesis, as a right delimiter.

`@`

A at-sign, used to indicate mathematical composition.

`<=`

A sequence representing the less-than-or-equal-to sign.

`>=`

A sequence representing the greater-than-or-equal-to sign.

`<>`

A sequence representing the not-equal-to sign.

`:=`

A sequence representing the assignment operation.

`or`

The logical OR sign.

`and`

The logical AND sign.

`not`

The logical NOT sign.

`[0-9]+`

A sequence of one or more digits representing a *NATURAL* number.

`[A-Z_a-z][0-9A-Z_a-z]*`

A sequence of one or more digits, upper case letters, underscores, or lower case letters, beginning with an upper case letter, an underscore, or a lower case letter, representing a *STRING*.

`"'"[A-Z_a-z][0-9A-Z_a-z]*`

A sequence, beginning with a right quote. The rest of the sequence is as above, namely, one or more digits, upper case letters, underscores, or lower case letters, beginning with an upper case letter, an underscore, or a lower case letter. This sequence is again a *STRING*.

`[A-Z_a-z][0-9A-Z_a-z]*"'"`

A sequence, beginning with a subsequence of one or more digits, upper case letters, underscores, or lower case letters, where the first character is an upper case letter, an underscore, or a lower case letter. The remainder of the sequence is a right quote character. The sequence is again a *STRING*.

`"‘"[^‘]+"‘"`

A sequence beginning and ending with a left quote. No left quotes are allowed internally, but any other character is acceptable. This sequence is again a *STRING*.

A.2 Syntax Analysis Rules

The following list describes the patterns of words recognized by the translator. Note the rules are given in BNF grammar and are applied in the order they appear. This information is part of the requirements specification for the translator.

⟨ **start** ⟩ ::=
    STRING `:=` ⟨ exp ⟩ `;`
    | ⟨ exp ⟩ `;`

⟨ **exp** ⟩ ::=
    ⟨ exp ⟩ `,` ⟨ exp1 ⟩
    | ⟨ exp1 ⟩

⟨ **exp1** ⟩ ::=
    ⟨ exp1 ⟩ `or` ⟨ exp2 ⟩
    | ⟨ exp2 ⟩

⟨ **exp2** ⟩ ::=
    ⟨ exp2 ⟩ `and` ⟨ exp3 ⟩
    | ⟨ exp3 ⟩

⟨ **exp3** ⟩ ::=
    `not` ⟨ exp3 ⟩
    | ⟨ exp4 ⟩

⟨ **exp4** ⟩ ::=
    ⟨ exp5 ⟩ `<` ⟨ exp5 ⟩
    | ⟨ exp5 ⟩ `>` ⟨ exp5 ⟩
    | ⟨ exp5 ⟩ `=` ⟨ exp5 ⟩
    | ⟨ exp5 ⟩ `<=` ⟨ exp5 ⟩
    | ⟨ exp5 ⟩ `>=` ⟨ exp5 ⟩
    | ⟨ exp5 ⟩ `<>` ⟨ exp5 ⟩
    | ⟨ exp5 ⟩

⟨ **exp5** ⟩ ::=
    `+` ⟨ exp6 ⟩
    | `-` ⟨ exp6 ⟩
    | ⟨ exp4 ⟩ `+` ⟨ exp6 ⟩
    | ⟨ exp4 ⟩ `-` ⟨ exp6 ⟩
    | ⟨ exp6 ⟩

⟨ **exp6** ⟩ ::=
⟨ exp6 ⟩ `*` ⟨ exp7 ⟩
| ⟨ exp6 ⟩ `/` ⟨ exp7 ⟩
| ⟨ exp6 ⟩ `@` ⟨ exp7 ⟩
| ⟨ exp7 ⟩

⟨ **exp7** ⟩ ::=
⟨ exp8 ⟩ `^` ⟨ exp8 ⟩
| ⟨ exp8 ⟩

⟨ **exp8** ⟩ ::=
⟨ name ⟩
| `(` ⟨ exp ⟩ `)`
| ⟨ exp9 ⟩

⟨ **exp9** ⟩ ::=
NATURAL .
| . *NATURAL*
| *NATURAL* . *NATURAL*
| *NATURAL*

⟨ **name** ⟩ ::=
STRING
| *STRING* `(` ⟨ exp ⟩ `)`

A.3 Translator Assumptions

This section discusses ideas which are part of the requirements specification of the translator. Their purpose is to restrict the translation problem, making it more manageable, while keeping it useful to the TTS.

The translator generates one TTS expression from the contents of a text file. The last character of a text string representing a single expression must be a semicolon. Translation begins at the beginning of the text file and ends with the first semicolon encountered.

In the syntax analysis rule pattern, *STRING* `:=` ⟨ exp ⟩ `;`, it should not be assumed that, ⟨ exp ⟩, will be given the name, *STRING*. In fact, only the ⟨ exp ⟩ `;` portion of the pattern is used in producing an expression. The beginning portion is present simply as an artifact of the original Maple syntax and is useful in allowing Maple output expressions to be translated into the TTS.

In addition to the basic string composed of only numbers, letters, and underscores, there are three alternate forms of string allowed. A basic string, preceded by a right quote, is a string intended to indicate the before value of a variable. A right quote occurring after a basic string, is a string intended to indicate the after value of a variable. Any sequence of characters beginning and ending with a left quote is also a string. This string is allowed in order to comply with the idea of a quoted string which is useful in Maple.

In the process of translation, the symbol table id for the symbol associated with each word, is required in order to create the TTS expression. It is assumed that a symbol can be found within a symbol table through a search over the Name class entries of the symbols. If this assumption is to hold, no two symbols may have the same name. Thus it must be taken as an error if a search retrieves more than one symbol id for a given name.

For many symbols, such as the addition operator and the logical negation operator, which are commonly used, it is assumed that these symbols are already present in the symbol table. It is expected that a name search will return one id for these symbols or an error occurred. The words associated with these expected symbols and their Name class entries are specified in Table A.1.

Lexical Word	Name class entry
/	/
=	=
>	>
<	<
-	-
+	+
^	^
*	*
@	@
<=	lessequal
>=	greaterequal
<>	notequal
or	logicalor
and	logicaland
not	logicalnot

Table A.1: Words and Names of Expected Symbols.

Other symbols, which could be thought of as optional, are automatically

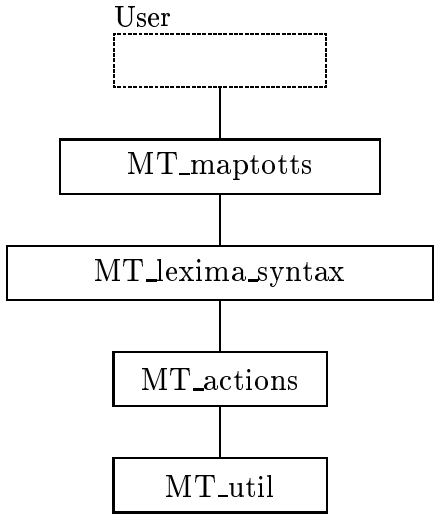
added to the symbol table as needed by the translator. Numeric constants, variables, and prefix functions are considered to have symbols that could be included as encountered. If the pattern, *STRING*, is found, then the *STRING* is the Name class entry of a variable. If the pattern, *STRING* (⟨ exp ⟩), is found, then the *STRING* is the Name class entry of a prefix function. Any sequence of digits matching the patterns identified as, ⟨ exp9 ⟩, in the syntax analysis rules, is the Name class entry of a numeric constant. If a Name class entry search for *STRING* finds one id, then that id can be used. If no id is found, it is assumed the symbol is not present and is then added as an id with *STRING* as the Name class entry. The length of *STRING* has a maximum, which is set by the symbolic constant *CHARCAPACITY*.

The translator assumes that all numeric constants are positive, with the exception of -1. The -1 constant is added to the symbol table if it is needed and not found. The expression, -1*⟨numeric constant⟩, represents a negative constant and allows fewer constants to be added to the symbol table. Therefore a negative numeric constant, such as -5, will not occur in the symbol table as a result of translation.

The success or failure of the translator, is referred to as the status. Status is maintained between calls to the utility. The notion of status is used in all of the TTS. For each module, it is assumed there is only one success token. For modules external to the translator, only the knowledge of the success status token is assumed. All status tokens, except those indicating simply success, from any used modules, either external or internal, are assumed to indicate error. The design of the translator is such that if an error is detected, then progress should be avoided.

A.4 Introduction to the Translator Modules

The translator is organized into five modules: MT_maptotts, MT_lexima_syntax, MT_actions, MT_util, and MT_error. The relationship between these modules is illustrated in Figure A.1. Modules are indicated by solid line boxes. Portions of the tool to which can be applied a label carrying an informal meaning are enclosed by a dashed line box. The label "user" refers to anything using the utility, including a tool. The solid line segments indicate direct communication between two modules or between a module and the user. The communication relation is non-reflexive, symmetric, and non-transitive. The MT_error module is omitted from the diagram. It is to be understood that all modules communicate with MT_error.



Legend

: Labeled portion of the system.

: Module.

: Communication relation.

Note: All modules communicate with the MT_error module.

Figure A.1: Organization of the Translator Utility

A.5 Translator Module Guide

The module guide indicates what is encapsulated in each module. These are the secrets of the module or the information which it hides.

MT_maptotts Module

The module hides the use of a variable, yyin, to access the input file and the use of the function, yyparse(), to begin the lexical and syntax analysis. The module is responsible for defining three global variables used by other modules, namely, a symbol table of type SymTbl, an output expression of type Expn, and a special comma symbol of type Id.

MT_lexima_syntax Module

The module hides the rules for lexical and syntax analysis. It encapsulates the use of the flex and yacc unix facilities. The access program is created by these facilities.

MT_actions Module

The secret of this module is the set of algorithms for the actions called by the MT_lexima_syntax module. The module uses the global symbol table, the global expression, and the global special comma symbol id, mentioned in the MT_maptotts module.

MT_util Module

The functions contained in this module are used by only the MT_actions module. The module was created on a basis of work assignment, since to include its contents into the MT_actions module would have made that module too large. Many of the functions are useful in a more general way, such as those that search for Ids in a symbol table. The module uses the global symbol table, and the global special comma symbol id.

MT_error Module

This module is responsible for providing the data type MT_Token for the user and other programs. The module hides the data structure of the MT_Token type. The MT_Token type preserves the status token of the translator from the last access program invocation, indicating success or the general reason for failure.

A.6 Translator Informal Interface Specifications

The functions of each module are described in this section, without using formal language. A listing of the access programs for each module contains a brief explanation and the types of the arguments and returns.

The MT_mapstotts module is one of two modules intended to be accessed directly by the user, but the only module to which the user submits information. The other module is the MT_Error Module. All other modules are intended to be local and not available to the user.

MT_maptotts Module Interface

This is the top level module. The module defines three global variables used by other modules, namely, a symbol table of type SymTbl, an output expression of type Expn, and a special comma symbol of type Id. The symbol table is accessed and modified by other modules as symbol Ids are required. The expression is the output expression of the utility, which is built gradually by other modules as translation continues. The special comma symbol is an artifact used in handling a comma separated sequence of expressions. The module sets the variable, FILE* yyin, defined by the yacc facility, to point to the input text file. Parsing is accomplished by calling the yyparse() function which is automatically generated by the yacc facility. There are two access programs.

MT_Init() Initializes the module to a known state.

Expn MapleToTts(SymTbl t, FILE * f) Reads the first text expression in the described syntax, from the file pointed to by f. Returns the equivalent TTS Expn; constructing it using the contents of t, and adding certain symbols to SymTbl as needed. The t is set as the global symbol table.

MT_lexima_syntax Module Interface

This is the lexical analyzer and syntax analyzer. The module has a global variable of type FILE* , with a standard name, yyin, used to access the text input file. There is one access program, called yyparse() which is automatically generated by the yacc unix facility.

int yyparse() Returns the integer value 0 if the text file pointed to by yyin is parsed successfully. Uses the lexical analysis rules and the syntax analysis rules to parse the file. Calls a sequence of actions to be taken as the result of a successful parsing.

MT_actions Module Interface

This is the actions module. It contains all the actions that need to be taken as a result of successfully parsing the text input file. Each particular type of action is an access program. The module uses the global symbol table, the global expression, and the global special symbol, mentioned in the MT_maptotts module. There are thirteen access programs.

ActDone(Expn e) Sets the completed expression, e, to the global expression variable.

Expn ActBinFATag(Expn e1, char c[CHARCAPACITY], Expn e2) This function generates a binary function application. Returns Expn of EType FATag. The root symbol has arity two and Name class entry c, the left child is e1, and the right child is e2.

Expn ActConstTag(char c[CHARCAPACITY]) This function generates a numeric constant decimal, whose value is encoded in its name. Returns Expn of EType ConstTag. The root symbol is an atom, which has a Name class entry c.

Expn ActNatDot(char c[CHARCAPACITY]) This function generates a numeric constant decimal, whose value is encoded in its name. Returns Expn of EType ConstTag. The root symbol is an atom, which has the Name class entry c concatenated with "." .

Expn ActDotNat(char c[CHARCAPACITY]) This function generates a numeric constant decimal, whose value is encoded in its name. Returns Expn of EType ConstTag. The root symbol is an atom, which has a Name class entry "." concatenated with c.

Expn ActNatDotNat(char c[CHARCAPACITY]) This function generates a numeric constant decimal, whose value is encoded as its name. Returns Expn of EType ConstTag. The root symbol is an atom, which has a Name class entry "." concatenated with c concatenated with "." .

Expn ActNegative(Expn e) This function generates an expression representing the negative of a numeric constant. Returns Expn of EType FATag. The root symbol has arity two and the Name class entry *. The left child is an atom of EType ConstTag and Name class entry -1. The right child is the expression e.

Expn ActName(char c[CHARCAPACITY]) This function generates a variable atom. Returns Expn of EType VarTag. The root symbol is an atom with Name class entry c.

Expn ActBinLETag(Expn e1, char c[CHARCAPACITY], Expn e2) This function generates a binary logical expression. Returns Expn of EType LETag. The root symbol has arity two and Name class entry c. The left child is e1 and the right child is e2.

Expn ActUnaLETag(char c[CHARCAPACITY], Expn e) This function generates a unary logical expression. Returns Expn of EType LETag. The root symbol has arity one and Name class entry c. The child is expression e.

Expn ActBinPETag(Expn e1, char c[CHARCAPACITY], Expn e2) This function generates a binary predicate expression. Returns Expn of EType PETag. The root symbol has arity two and Name class entry c. The left child is e1 and the right child is e2.

Expn ActFun(char c[CHARCAPACITY], Expn e) This function generates a function application expression, where there could be many arguments. Returns Expn of EType FATag. The expression e, generated by the parsing process, holds the arguments of a general arity function application. The arity of the root symbol is derived from expression e. The Name class entry of the root symbol is c. The children of the returned expression are extracted from the expression e.

Expn ActComma(Expn e1, Expn e2) This function is using in generating the expression representing a function's list of arguments. Returns Expn of EType FATag. The root symbol has arity two. The left child is e1 and the right child is e2.

MT_util Module Interface

This is a utility module. The functions contained in it are used by only the MT_actions module. The module was created on a basis of work assignment, since to include its contents into the MT_actions module would have made that

module too large. Many of the functions are useful in a more general way, such as those that search for Ids in a symbol table. There are eight access programs.

Id getid(char c[CHARCAPACITY]) This function is a simple search of the global symbol table. Returns the Id for a symbol with Name class entry c. The function is useful for finding the Id of an expected symbol.

Id getorputidwithtag(char c[CHARCAPACITY], EType t) This function searches the global symbol table and if not successful, it adds the symbol with the given name c, and EType t. Returns the Id for a symbol from the global symbol table with Name class entry c. The function is useful in finding the Id of a numerical constant.

Id getstrid(char c[CHARCAPACITY]) This function searches the global symbol table and if not successful, it adds the symbol with the given name c, and EType VarTag. Returns the Id for a symbol from the global symbol table with name class entry c. The function is useful in finding the Id of a variable.

EType converttag(int t) This function converts the integer Tag class entry, t, as found in the global symbol table, into its matching EType. Returns the EType which corresponds to integer t.

int tagconvert(EType t) This function converts the EType t into its integer encoding that would be the Tag class entry in the global symbol table. Returns the integer which corresponds to EType t.

Expn connectexpns(Expn e1, Expn e2) This function converts between two different representations of the arguments for an n-ary function. The expression e2 is a kind of binary tree representation of an n-tuple. The expression e1 is the n-ary function application to which the arguments are to be assigned. Returns an Expn which is the n-ary function application with arguments assigned in the correct order.

Id getfunid(char c[CHARCAPACITY], EType t, int a) This function searches the global symbol table for Name class entry c. If not successful, it searches the global symbol table for MapleName class entry c. If still not successful, it adds the symbol with the given name c, EType t, and arity a. Returns the Id for a symbol from the global symbol table with name class entry c. The function is useful in finding the Id of a function application. The function checks c as a MapleName class entry in case the reference is to a Maple defined function.

int getarity(Expn e) The expression e is a kind of binary tree representation of an n-tuple. The n-tuple represents the arguments of a n-ary function application. This function determines what the arity of the function application would need to be for it to use all of the arguments contained in expression e. Returns the arity of e and an integer.

MT_error Module Interface

This is the status module for the translator. The status token persists between calls to the module. The purpose of the module is to maintain the utility status from the last access program invocation. The utility status can be checked using the modules access programs to determine if an error has been detected or if progress can be made. The module also checks with other parts of the TTS that are not part of the utility to determine if their status is clear to proceed. The initial state of this module must be set when the utility is first used. There are twelve tokens.

MT_Success No errors or potential problems detected. Clear to proceed.

MT_NULL_File The pointer to the text file to parse is NULL.

MT_TH_Error The TH module set has not reported success.

MT_Info_Error The Info module has not reported success.

MT_Arity_Mismatch The arity of a function application does not match the number or arguments applied to it.

MT_Unknown_EType The EType encountered was not expected.

MT_Multi_Ids More than one Id with the given name has been found in the symbol table.

MT_String_Length A concatenated name has a length no longer less than *CHARCAPACITY*.

MT_yyparse_Fail A nonspecific failure occurred in the yyparse function of the MT_lexima_syntax module.

MT_THInfo_Error Either the TH module or the Info module has not reported success.

MT_No_Id The are no Ids with the given name in the symbol table.

MT_Failure A non-specific problem occurred.

There are four access programs.

SetErrMT(MT_Token T **)** Sets the module to the MT_Token value of T.

MT_Token GetErrMT( ) Returns the MT_Token value currently maintained by the module.

char * GetStrErrMT(MT_Token T **)** Returns a pointer to a character string, which is the text message for the MT_Token value currently maintained by the module.

MT_Token MT_CheckStatus( ) Returns the MT_Token value maintained by the module after checking the status of the MT_Error module, the TH_Error module, and the In_Error module.

A.7 Translator Uses Hierarchy

The following diagrams illustrate the uses hierarchy of the translator. In general the programs of the MT_error module are not shown, since any program may use the access programs of that module. Note that the legend appearing in Figure A.2 applies to all uses hierarchies of the translator. Since the yyparse program uses all access programs of the MT_actions module, the hierarchy has been broken apart to illustrate the MT_actions uses hierarchies separately.

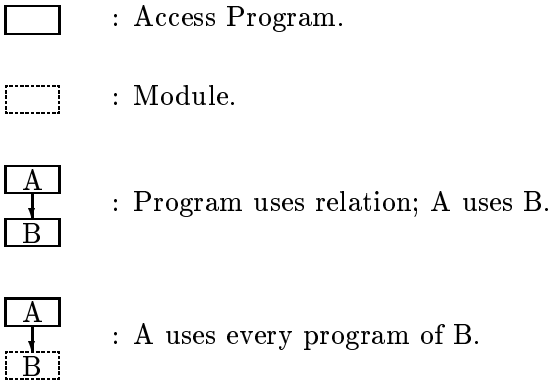


Figure A.2: Legend for all Translator Uses Hierarchies.

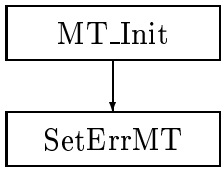


Figure A.3: Uses Hierarchy for MT_maptotts: MT_Init.

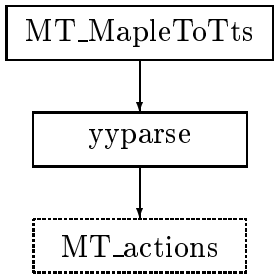


Figure A.4: Uses Hierarchy for MT_maptotts: MT_MapleToTts.

Figure A.5: Uses Hierarchy for MT_actions: ActDone.

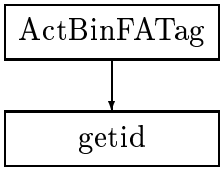


Figure A.6: Uses Hierarchy for MT_actions: ActBinFATag.

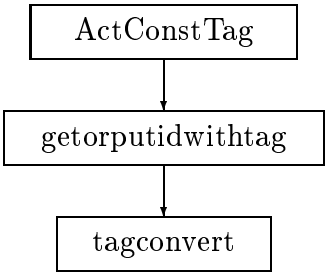


Figure A.7: Uses Hierarchy for MT_actions: ActConstTag.

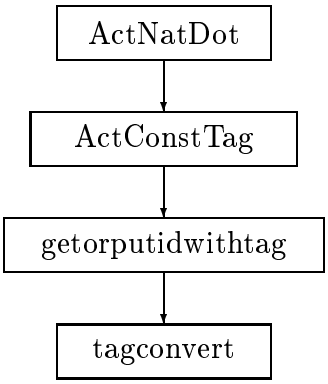


Figure A.8: Uses Hierarchy for MT_actions: ActNatDot.

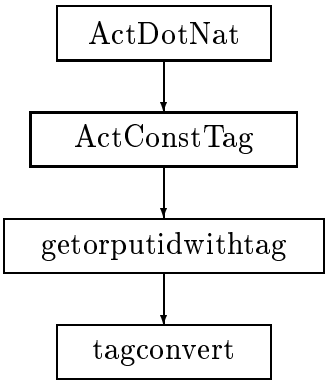


Figure A.9: Uses Hierarchy for MT_actions: ActDotNat.

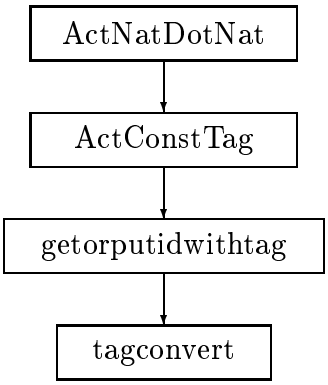


Figure A.10: Uses Hierarchy for MT_actions: ActNatDotNat.

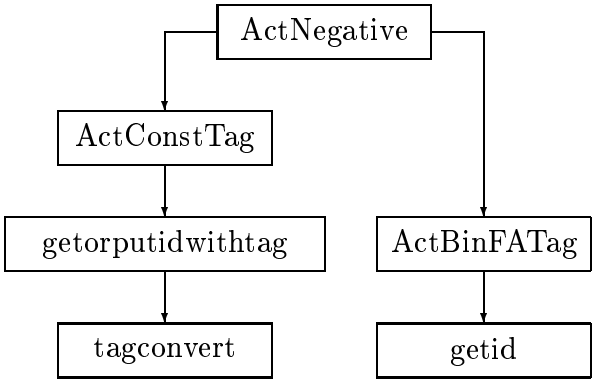


Figure A.11: Uses Hierarchy for MT_actions: ActNegative.

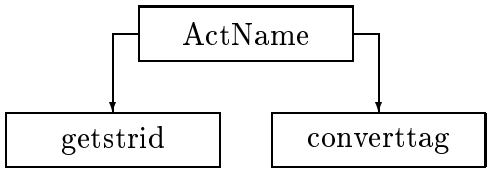


Figure A.12: Uses Hierarchy for MT_actions: ActName.

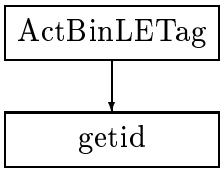


Figure A.13: Uses Hierarchy for MT_actions: ActBinLETag.

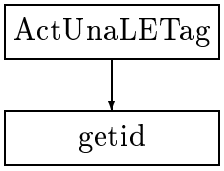


Figure A.14: Uses Hierarchy for MT_actions: ActUnaLETag.

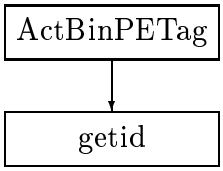


Figure A.15: Uses Hierarchy for MT_actions: ActBinPETag.

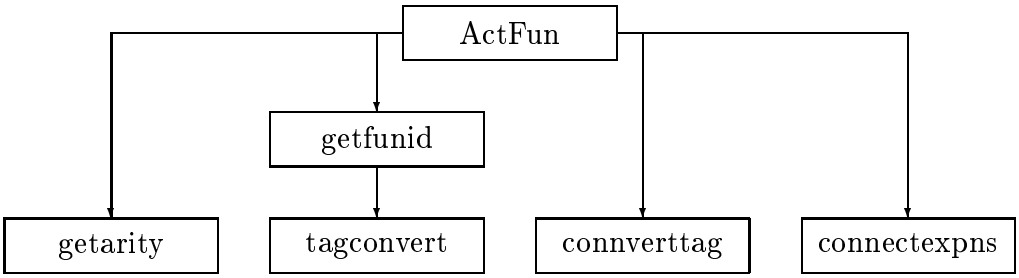


Figure A.16: Uses Hierarchy for MT_actions: ActFun.

Figure A.17: Uses Hierarchy for MT_actions: ActComma.

A.8 Translator Algorithms

This section is considered part of an *informal module internal design* document where the details enabling a module to meet its obligations are provided without using formal language. The algorithms for the functions in the informal module interfaces are contained herein.

MapleToTts

The following is the algorithm for function
Expn MapleToTts(SymTbl t, FILE * f).

1. If f is not NULL then...
   (a) To the global symbol table, assign t.
   (b) To the global file pointer, yyin, of the MT_lexima_syntax module assign f.
   (c) Create a global Id, *comma*, for the comma, recognized by the syntax analysis rules, in the global symbol table.
   (d) Call the yyparse() function of the MT_lexima_syntax module.
   (e) Destroy the global Id, *comma*, for the comma from the global symbol table.
   (f) If the yyparse() function reports the parse was unsuccessful, Then call SetErrMT(MT_yyparse_Fail).
2. Else if f is NULL, then call SetErrMT(MT_NULL_File).
3. Return the global output expression.

MT_Init

The algorithm for the function MT_Init(), is to call SetErrMT(MT_Success) and assign the undefined symbol table constant, *TABLEUNDEF*, to the global symbol table.

ActDone

The algorithm for the function ActDone(Expn e), is to copy e to the global output expression and then destroy e.

ActBinFATag

The following is the algorithm for the function,
Expn ActBinFATag(Expn $e1$, char *name*[CHARCAPACITY], Expn $e2$).

1. Call the function getid(*name*). Let *id* be the returned Id.
2. Create an expression, $e3$, with EType FATag, Id *id*, and arity 2.
3. Copy $e1$ as the first argument of $e3$ and copy $e2$ as the second argument of $e3$.
4. Destroy $e1$ and $e2$.
5. Return $e3$.

ActConstTag

The following is the algorithm for the function
Expn ActConstTag(char *name*[CHARCAPACITY]).

1. Call the function getorputidwithtag(*name*, ConstTag), with the indicated arguments. Let *id* be the returned Id.
2. Create an expression, e, with EType ConstTag and Id *id*.
3. Return e.

ActNatDot

The following is the algorithm for the function
Expn ActNatDot(char *name*[CHARCAPACITY]).

1. Let i be the length of the string *name*.
2. If $i + 1 <$ CHARCAPACITY, then let *nameD* be the concatenation of *name* and "." and call ActConstTag(*nameD*) assigning the result to e.
3. Else Call SetErrMT(MT_String_Length).
4. Return e.

ActDotNat

The following is the algorithm for the function
Expn ActDotNat(char *name*[CHARCAPACITY]).

1. Let i be the length of the string *name*.
2. If $i + 1 <$ CHARCAPACITY, then let *Dname* be the concatenation of "." and *name* and call ActConstTag(*Dname*) assigning the result to e.
3. Else Call SetErrMT(MT_String_Length).
4. Return e.

ActNatDotNat

The following is the algorithm for the function
Expn ActNatDotNat(char *name*1[CHARCAPACITY],
char *name*2[CHARCAPACITY]).

1. Let i be the length of the string *name*1.
2. Let j be the length of the string *name*2
3. If $i + j + 1 <$ CHARCAPACITY, then let *name*1*Dname*2 be the concatenation of *name*1, ".", and *name*2 and call ActConstTag(*name*1*Dname*2) assigning the result to e.
4. Else Call SetErrMT(MT_String_Length).
5. Return e.

ActNegative

The following is the algorithm for the function
Expn ActNegative(Expn e).

1. Call ActConstTag("-1") assigning the result to $e1$.
2. Call ActBinFATag($e1$, "*", e) assigning the result to $e2$.
3. Return $e2$.

ActName

The following is the algorithm for the function
Expn ActName(char *name*[CHARCAPACITY]).

1. Call the function getstrid(*name*), assigning the returned Id to *id*.
2. Get the Tag class information, *tag*, for *id* from the global symbol table.
3. Call converttag(*tag*) to get the EType value, assigning it to *type*.
4. Create an expression, e, with EType *type* and Id *id*.
5. Return e.

ActBinLETag

The following is the algorithm for the function
Expn ActBinLETag(Expn $e1$, char *name*[CHARCAPACITY], Expn $e2$).

1. Call the function getid(*name*), assigning the returned Id to *id*.
2. Create an expression, e, with EType LETag, Id *id*, and arity 2.
3. Copy $e1$ as the first argument of e and copy $e2$ as the second argument of e.
4. Destroy $e1$ and $e2$.
5. Return e.

ActUnaLETag

The following is the algorithm for the function
Expn ActUnaLETag(char *name*[CHARCAPACITY], Expn $e1$).

1. Call the function getid(*name*), assigning the returned Id to *id*.
2. Create an expression, e, with EType LETag, Id *id*, and arity 1.
3. Copy $e1$ as the argument of e.
4. Destroy $e1$.
5. Return e.

ActBinPETag

The following is the algorithm for the function
Expn ActBinPETag(Expn $e1$, char *name*[CHARCAPACITY], Expn $e2$).

1. Call the function getid(*name*), assigning the returned Id to *id*.
2. Create an expression, e, with EType PETag, Id *id*, and arity 2.
3. Copy $e1$ as the first argument of e and copy $e2$ as the second argument of e.
4. Destroy $e1$ and $e2$.
5. Return e.

ActFun

The following is the algorithm for the function
Expn ActFun(char *name*[CHARCAPACITY], Expn e). The function uses the global Id, *comma*, defined in MT_maptotts.

1. Call the function getarity(e), assigning the returned integer to *arity*.

2. Call the function getfunid(*name*, FATag, *arity*), assigning the returned Id to *id*.
3. Get the Tag class information, *tag*, for *id* from the global symbol table.
4. Call converttag(*tag*) to get the EType value, assigning it to *type*.
5. Get the Arity class information, *intdata*, for *id* from the global symbol table.
6. Create an expression, $e1$, with EType *type*, Id *id*, and arity *intdata*.
7. Get the Id, *id*1 for the root symbol of e.
8. If *arity* equals *intdata*, then ...
   (a) If *id*1 equals *comma*, then call connectexpns($e1$, e), assigning the returned expression to $e2$.
   (b) Else ...
      i. Copy e as the first argument of $e1$.
      ii. Copy $e1$ to $e2$.
      iii. Destroy e and $e1$.
9. Else Call SetErrMT(MT_Arity_Mismatch).
10. Return $e2$.

ActComma

The following is the algorithm for the function Expn ActComma(Expn $e1$, Expn $e2$). The function uses the global Id, *comma*, defined in MT_maptotts.

1. Create an expression, $e3$, with EType FATag, Id *comma*, and arity 2.
2. Copy $e1$ as the first argument of $e3$ and copy $e2$ as the second argument of $e3$.
3. Destroy $e1$ and $e2$.
4. Return $e3$.

getid

The following is the algorithm for the function
Id getid(char *name*[CHARCAPACITY]).

1. Get the Id of symbols in the global symbol table with Name class entry *name*.
2. If there is only one Id with the given Name class entry, then assign that Id to *id*.
3. Else if there is more than one Id with the given Name class entry, then call SetErrMT(MT_Multi_Ids).
4. Else call SetErrMT(MT_No_Id).
5. Return *id*.

getorputidwithtag

The following is the algorithm for the function
Id getorputidwithtag(char *name*[CHARCAPACITY], EType *tag*).

1. Get the Id of symbols in the global symbol table with Name class entry *name*.
2. If there is only one Id with the given Name class entry, then assign that Id to *id*.
3. Else if there is no Id with the given Name class entry, then ...
   (a) Create a new symbol in the global symbol table and assign its Id to *id*.
   (b) Assign *name* to the Name class entry of the symbol with Id *id*.
   (c) Call tagconvert(*tag*), assigning the returned value to *tagdata*.
   (d) Assign *tagdata* to the Tag class entry of the symbol with Id *id*.
   (e) Assign any other necessary additional class entry data to symbol of Id *id*.
4. Else call SetErrMT(MT_Multi_Ids).
5. Return *id*.

getstrid

The following is the algorithm for the function
Id getstrid(char *name*[CHARCAPACITY]).

1. Get the Id of symbols in the global symbol table with Name class entry *name*.
2. If there is only one Id with the given Name class entry, then assign that Id to *id*.
3. Else if there is no Id with the given Name class entry, then ...
   (a) Create a new symbol in the global symbol table and assign its Id to *id*.
   (b) Assign *name* to the Name class entry of the symbol with Id *id*.
   (c) Assign the integer 2 to the Tag class entry of the symbol with Id *id*.
   (d) Assign any other necessary additional class entry data to symbol of Id *id*.
4. Else call SetErrMT(MT_Multi_Ids).
5. Return *id*.

converttag

The following is the algorithm for the function
EType converttag(int *tag*).

1. If *tag* equals 0, then to EType variable, *type*, assign ConstTag.
2. Else if *tag* equals 1, then to EType variable, *type*, assign PConstTag.
3. Else if *tag* equals 2, then to EType variable, *type*, assign VarTag.
4. Else if *tag* equals 3, then to EType variable, *type*, assign FATag.
5. Else if *tag* equals 4, then to EType variable, *type*, assign PETag.
6. Else if *tag* equals 5, then to EType variable, *type*, assign LETag.
7. Else if *tag* equals 6, then to EType variable, *type*, assign QLETag.

8. Else if *tag* equals 7, then to EType variable, *type*, assign FTableTag.
9. Else if *tag* equals 8, then to EType variable, *type*, assign PdTableTag.
10. Else call SetErrMT(MT_Unknown_EType).
11. Return *type*.

tagconvert

The following is the algorithm for the function
int tagconvert(EType *tag*).

1. If *tag* equals ConstTag, then assign 0 to *tagdata*.
2. Else if *tag* equals PConstTag, then assign 1 to *tagdata*.
3. Else if *tag* equals VarTag, then assign 2 to *tagdata*.
4. Else if *tag* equals FATag, then assign 3 to *tagdata*.
5. Else if *tag* equals PETag, then assign 4 to *tagdata*.
6. Else if *tag* equals LETag, then assign 5 to *tagdata*.
7. Else if *tag* equals QLETag, then assign 6 to *tagdata*.
8. Else if *tag* equals FTableTag, then assign 7 to *tagdata*.
9. Else if *tag* equals PdTableTag, then assign 8 to *tagdata*.
10. Else call SetErrMT(MT_Unknown_EType).
11. Return *tagdata*.

connectexpns

The following is the algorithm for the function
Expn connectexpns(Expn $e1$, Expn $e2$).

1. Copy $e1$ as the expression, $e3$.

2. Assign the arity of the root of $e1$ to *arity*.
3. Insert the value 2 as the first element of an empty Path, p.
4. Copy the expression at p in $e2$, as the *arity*th argument of $e3$.
5. For integer i from $(arity - 1)$ down to 2, do the following ...
   (a) Insert the value 1 as the first element of p.
   (b) Copy the expression at p in $e2$, as the ith argument of $e3$.
6. Assign the value 1 to the last element of p.
7. Copy the expression at p in $e2$, as the first argument of $e3$.
8. Destroy $e1$, and $e2$.
9. Return $e3$.

getfunid

The following is the algorithm for the function
Id getfunid(char *name*[CHARCAPACITY], EType *tag*, int *arity*).

1. Get the Id of symbols in the global symbol table with Name class entry *name*.
2. If there is only one Id with the given Name class entry, then assign that Id to *id*.
3. Else if there is no Id with the given Name class entry, then ...
   (a) Get the Id of symbols in the global symbol table with MapleName class entry *name*.
   (b) If there is only one Id with the given MapleName class entry, then assign that Id to *id*.
   (c) Else if there is no Id with the given MapleName class entry, then ...
      i. Create a new symbol in the global symbol table and assign its Id to *id*.
      ii. Assign *name* to the Name class entry of the symbol with Id *id*.
      iii. Call tagconvert(*tag*), assigning the returned value to *tagdata*.

iv. Assign *tagdata* to the Tag class entry of the symbol with Id *id*.

v. Assign *arity* to the Arity class entry of the symbol with Id *id*.

vi. Assign any other necessary additional class entry data to symbol of Id *id*.

(d) Else call SetErrMT(MT_Multi_Ids).

4. Else call SetErrMT(MT_Multi_Ids).

5. Return *id*.

getarity

The following is the algorithm for the function
int getarity(Expn *e*). This function uses the global Id, *comma*.

1. Assign the value 1 to the integer *arity*.

2. Assign the Id of the root of *e* to *id*.

3. Let *p* be an empty Path.

4. While *id* equals *comma*, do ...

   (a) Increment *arity*.

   (b) Insert the value 1 as the first element of *p*.

   (c) Assign the Id of the symbol at *p* in *e* to *id*.

5. Return *arity*.

A.9 Translator Implementation Issues

This section contains ideas used in the implementation of the modules of the translator. Thus it is part of the module internal designs documentation.

A.9.1 Data Structures

The arity of expressions in the table holder cannot be extended. Thus the idea of an arbitrary comma separated list, which occurs in the syntax analysis rules, is provided in the translator by the special comma symbol. This symbol is considered as having arity 2. A *comma* expression has the comma symbol as root. The left child is either another comma expression or a member of the comma separated list. The right child is a member of the comma separated list.

In building a comma expression from a comma separated list, there is a base step and inductive step. In the base step, the first argument of the list is copied as the left child of the comma expression. The second argument of the list is copied as the right child of the comma expression. In the inductive step, adding the ith argument of the list, requires the comma expression for the previous $(i-1)$ arguments, which becomes the left child of a comma expression. The ith list argument becomes the right child. Thus the order of occurrence of arguments is preserved.

The MT_lexima_syntax module uses the well known facilities of *flex* and *yacc* in the Unix system. The flex utility performs lexical analysis with the rules found within the MT_lexima.l file. The yacc utility performs syntax analysis with the rules found within the MT_syntax.y file. The codes resulting from these two utilities communicate with each other and with the rest of the translator using global variables and assumed names, such as the file pointer *yyin* and the parsing function *yyparse()*. It is these utilities that motivate the use of the three global variables defined in the MT_maptotts module.

A.9.2 Data Structure Guidelines

In general, if a data structure must be used, the implementation attempts to use the data structures of existing modules in preference to the creation of new private data structures. Modules constructed in this way, have as their secret the fact that they delegate their data structures to another module. Thus this paradigm avoids data structure code, removing a significant source of error. Only the MT_lexima_syntax, and MT_error modules violate the paradigm.

Modules are designed to make the use of traces unnecessary. The only canonical traces the paradigm allows is the empty trace and the trace composed of only the last access program call. Thus the implementation of a module can avoid

the complexities of the interaction of access programs on the data structures. For example, a module not following the paradigm might have access programs to create and destroy an object and to add or remove information contained by it. Such a module can have complex traces with the possibility of many conflicts between access programs. Only the MT_error module violates the paradigm.

A.9.3 Status Checking Guidelines

The MT_error module maintains a token indicating the status of the translator. When MT_Init is called, the token is set to MT_Success. At various points in the translator code, the token may be set to something other than MT_Success, indicating a change of status. The MT_error module follows the paradigm that in general, a change in status is taken to be the detection of a potential error. In designing the translator, tokens have been invented as needed to indicate the nature of a problem.

The TTS modules external to the translator also use tokens to indicate status. A check is done on the status of any external module used. Only the success tokens are known explicitly by the MT_error module. On receiving any status token from an external module which is not success, the MT_error module sets a token pointing out the module the reported the problem. The term *finger-pointing* is applied to this non-specific error report which indicates only the problem's origin.

A.9.4 Code Readability Guidelines

In coding a function for the translator, checking the status of called functions requires the addition of code, which is not part of the algorithm code. Typically the lines of code for status checking, significantly outnumber the lines of algorithm code. The readability of the code can be seriously eroded, even for the programmer, resulting in errors that might otherwise have been noticed. The paradigm of separating status checking code from algorithm code can preserve readability.

The term *armor* refers to performing all status checks at only the beginning and possibly at the end of each function. The algorithm code remains in an undisturbed form, without having the status checking code dispersed through it. The armor checks the status of relevant tokens at the beginning of a function. Only if all responses indicate success, will the algorithm code for the function be

performed. The success of armor at catching errors depends on the algorithm code. The code of the translator has been implemented with armor, except for the top level access programs.

Appendix B

Module Interface Specifications

This appendix contains module interface specifications for the user accessible modules of the Function Composition Tool, namely, the CT_Mid and CT_Error modules.

CT_Error Module Interface Specification

(0) CHARACTERISTICS

Imported Types: None
Exported Types: CT_Token

(1) SYNTAX

Access Programs

Program Name	Value	Arg#1
SetErrCT		CT_Token
GetErrCT	CT_Token	
GetStrErrCT	char *	CT_Token
CT_CheckStatus	CT_Token	

(2) ABSTRACT REPRESENTATION

$token$ = The last error token reported.
$string[t]$ = The error string associated with the error token t

(3) BEHAVIOUR

SetErrCT(t):

Conditions	**New State**
t is a valid token	$token' = t$
t is not a valid token	$token' = 'token$

GetErrCT() $\rightarrow t$:

Conditions	**New State**
true	$t = 'token$

GetStrErrCT(t) $\rightarrow s$:

Conditions	**New State**
t is a valid token	$s = 'string[t]$
t is not a valid token	$s =$"Undefined Error Token"

CT_CheckStatus() $\rightarrow t$:

Conditions	**New State**
GetErrCT() $\neq$ CT_Success	t ='*token*
GetErrCT()=CT_Success $\wedge$ GetErrTH()$\neq$TH_Success	t =CT_TH_Error
GetErrCT()=CT_Success $\wedge$ GetErrTH()=TH_Success $\wedge$ GetErrInfo()$\neq$In_Success	t =CT_Info_Error
GetErrCT()=CT_Success $\wedge$ GetErrTH()=TH_Success $\wedge$ GetErrInfo()=In_Success $\wedge$ GetErrGTS()$\neq$GTS_Success	t =CT_GTS_Error
GetErrCT()=CT_Success $\wedge$ GetErrTH()=TH_Success $\wedge$ GetErrInfo()=In_Success $\wedge$ GetErrGTS()=GTS_Success	t ='*token*

CT_Mid Module Interface Specification

(0) CHARACTERISTICS

Imported Types: Expn, SymTbl
Internal Types: bool
Exported Types: None

(1) SYNTAX

Access Programs

Program Name	Value	Arg#1	Arg#2	Arg#3
CT_Mid_Init				
CT_Mid_ CreateNormalFormComp	Expn	SymTbl	Expn	Expn
CT_Mid_ CreateVectorComp	Expn	SymTbl	Expn	Expn

(2) ABSTRACT REPRESENTATION

status = The error status in CT_Error (= CT_Success if not given).

(3) BEHAVIOUR

CT_Mid_Init(): (SetErrCT(CT_Success))

CT_Mid_CreateNormalFormComp(t, l, r) $\rightarrow e$:

Conditions	**Output Values**	**New State**
$\neg$ IsValid(l, t) $\vee$ $\neg$ IsValid(r, t)	undefined	$status$' = CT_Reject_Expr
IsCond(r)	undefined	$status$ = CT_Reject_Expr
IsCond(l) $\wedge$ IsTable(r)	undefined	$status$' = CT_Reject_Expr
IsCond(l) $\wedge$ IsSimSub(r)	$e = l \circ r$	
IsSimSub(l) $\wedge$ IsSimSub(r)	$e = l \circ r$	
IsSimSub(l) $\wedge$ IsTable(r)	$e = l \circ \text{VecNor}(r, t)$	
IsTable(l) $\wedge$ IsSimSub(r)	$e = \text{VecNor}(l, t) \circ r$	
IsTable(l) $\wedge$ IsTable(r)	$e = \text{VecNor}(l, t) \circ \text{VecNor}(r, t)$	

CT_Mid_CreateVectorComp(t, l, r) $\rightarrow e$:

Conditions	**Output Values**	**New State**
$\neg$ IsValid(l, t) $\vee$ $\neg$ IsValid(r, t)	undefined	$status$' = CT_Reject_Expr
IsCond(r)	undefined	$status$ = CT_Reject_Expr
IsCond(l) $\wedge$ IsTable(r)	undefined	$status$' = CT_Reject_Expr
IsCond(l) $\wedge$ IsSimSub(r)	$e = l \circ r$	
IsSimSub(l) $\wedge$ IsSimSub(r)	$e = l \circ r$	
IsSimSub(l) $\wedge$ IsTable(r)	$e = \text{NorVec}(l \circ \text{VecNor}(r, t), t)$	
IsTable(l) $\wedge$ IsSimSub(r)	$e = \text{NorVec}(\text{VecNor}(l, t) \circ r, t)$	
IsTable(l) $\wedge$ IsTable(r)	$e = \text{NorVec}(\text{VecNor}(l, t) \circ \text{VecNor}(r, t), t)$	

AUXILIARY PREDICATES

IsCond: Expn $\rightarrow$ bool
IsCond(e) $\stackrel{df}{=}$
(ExpnGetSubTag(e, PathCreate()) = PConstTag $\vee$
ExpnGetSubTag(e, PathCreate()) = LETag $\vee$
ExpnGetSubTag(e, PathCreate()) = PETag)

IsTable: Expn $\rightarrow$ bool
IsTable(e) $\stackrel{df}{=}$ ExpnGetSubTag(e, PathCreate()) = PETag

IsSimSub: Expn $\rightarrow$ bool

IsSimSub(e) $\stackrel{df}{=}$ ExpnGetSubTag(e, PathCreate()) = FATag

IsValid: Expn $\times$ SymTbl $\rightarrow$ bool
IsValid(e, t) $\stackrel{df}{=}$

Conditions	**Value**
e with t satisfies the algorithm for *Checking Expressions* for Function Composition Tool	BOOL_TRUE
$\neg$ (e with t satisfies the algorithm for *Checking Expressions* for Function Composition Tool)	BOOL_FALSE

AUXILIARY FUNCTIONS

cgg: SymTbl $\times$ Expn $\rightarrow$ GTS_CCG
ccg(t, e) $\stackrel{df}{=}$
GTS_semGetCCG(GTS_infoGetTabSem(t, ExpnGetSubId(e, PathCreate())))

VecNor: Expn $\times$ SymTbl $\rightarrow$ Expn
VecNor(e, t) $\stackrel{df}{=}$

Conditions	**Value**
ccg(t, e) = NORMAL	e
ccg(t, e) = VECTOR	The table created by the algorithms for *Converting Vector to Normal Function Tables*, for the Function Composition Tool.

NorVec: Expn $\times$ SymTbl $\rightarrow$ Expn

NorVec$(e, t) \stackrel{df}{=}$

Conditions	**Value**
ccg$(t, e) =$ VECTOR	e
ccg$(t, e) =$ NORMAL	The table created by the algorithms for *Converting Normal to Vector Function Tables*, for the Function Composition Tool.

Bibliography

[1] D. L. Parnas, "Precise description and specification of software," in *Mathematics of Dependable Systems II* (V. Stavridou, ed.), pp. 1–14, Clarendon Press, 1997.

[2] C. Brink, W. Kahl, and G. Schmidt, eds., *Relational methods in computer science*. New York: Springer, 1997.

[3] J. G. Sanderson, *A relational theory of computing*. New York: Springer-Verlag, 1980.

[4] P. R. Halmos, *Naive Set Theory*. New York: Van Nostrand Rheinhold, 1960.

[5] D. L. Parnas, "Tabular representation of relations," CRL Report No. 260, Communications Research Laboratory (CRL), Telecommunications Research Institute of Ontario (TRIO), McMaster University, Hamilton, Ontario, Canada, L8S 4K1, Oct. 1992. 17 pgs.

[6] R. Janicki, "On a formal semantics of tabular expressions," CRL Report No. 355, Communications Research Laboratory (CRL), Telecommunications Research Institute of Ontario (TRIO), McMaster University, Hamilton, Ontario, Canada, L8S 4K1, Oct. 1997. 32 pgs.

[7] R. Janicki, D. L. Parnas, and J. Zucker, "Tabular representations in relational documents," CRL Report No. 313, Communications Research Laboratory (CRL), Telecommunications Research Institute of Ontario (TRIO), McMaster University, Hamilton, Ontario, Canada, L8S 4K1, May 1995. 15 pgs.

[8] M. von Mohrenschildt, "Algebra of normal function tables," CRL Report No. 350, Communications Research Laboratory (CRL), Telecommunications Research Institute of Ontario (TRIO), McMaster University, Hamilton, Ontario, Canada, L8S 4K1, May 1997. 14 pgs.

[9] McMaster University Software Engineering Research Group, "Table tool system developer's guide," CRL Report No. 339 & 340, Communications Research Laboratory (CRL), Telecommunications Research Institute of Ontario (TRIO), McMaster University, Hamilton, Ontario, Canada, L8S 4K1, Jan. 1997.

[10] D. L. Parnas, "On a 'buzzword': Hierarchical structure," in *Proceedings of IFIP World Congress 1974*, pp. 354–359, North Holland, 1974.

[11] E. Mendelson, *Introduction to Mathematical Logic*. Pacific Grove, CA: Wadsworth and Brooks, third ed., 1987.

[12] V. Sperschneider and G. Antoniou, *Logic: a foundation for computer science*. New York: Addison-Wesley, 1991.

[13] E. Sekerinski, "A calculus for predicative programming," in *Proceedings of the 2nd International Conference on the Mathematics of Program Construction* (R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, eds.), 1992.

[14] R. Abraham, "Evaluating generalized tabular expressions in software documentation," CRL Report No. 346, Communications Research Laboratory (CRL), Telecommunications Research Institute of Ontario (TRIO), McMaster University, Hamilton, Ontario, Canada, L8S 4K1, Feb. 1997.

[15] P. Rastogi, "Specialization: An approach to simplifying tables in software documentation," CRL Report No. 360, Communications Research Laboratory (CRL), Telecommunications Research Institute of Ontario (TRIO), McMaster University, Hamilton, Ontario, Canada, L8S 4K1, Mar. 1998.

[16] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, pp. 1053–1058, Dec. 1972.

[17] B. W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan, and S. M. Watt, *Maple V Language Reference Manual*. New York: Springer-Verlag, first ed., 1991.

[18] B. W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan, and S. M. Watt, *Maple V Library Reference Manual*. New York: Springer-Verlag, first ed., 1991.