Delft University of Technology
Parallel and Distributed Systems Report Series

Template Manager Reference Manual

C. van Reeuwijk

report number PDS-2000-003

ISSN 1387-2109

Published and produced by:
Parallel and Distributed Systems Section
Faculty of Information Technology and Systems Department of Technical Mathematics and Informatics
Delft University of Technology
Zuidplantsoen 4
2628 BZ Delft
The Netherlands

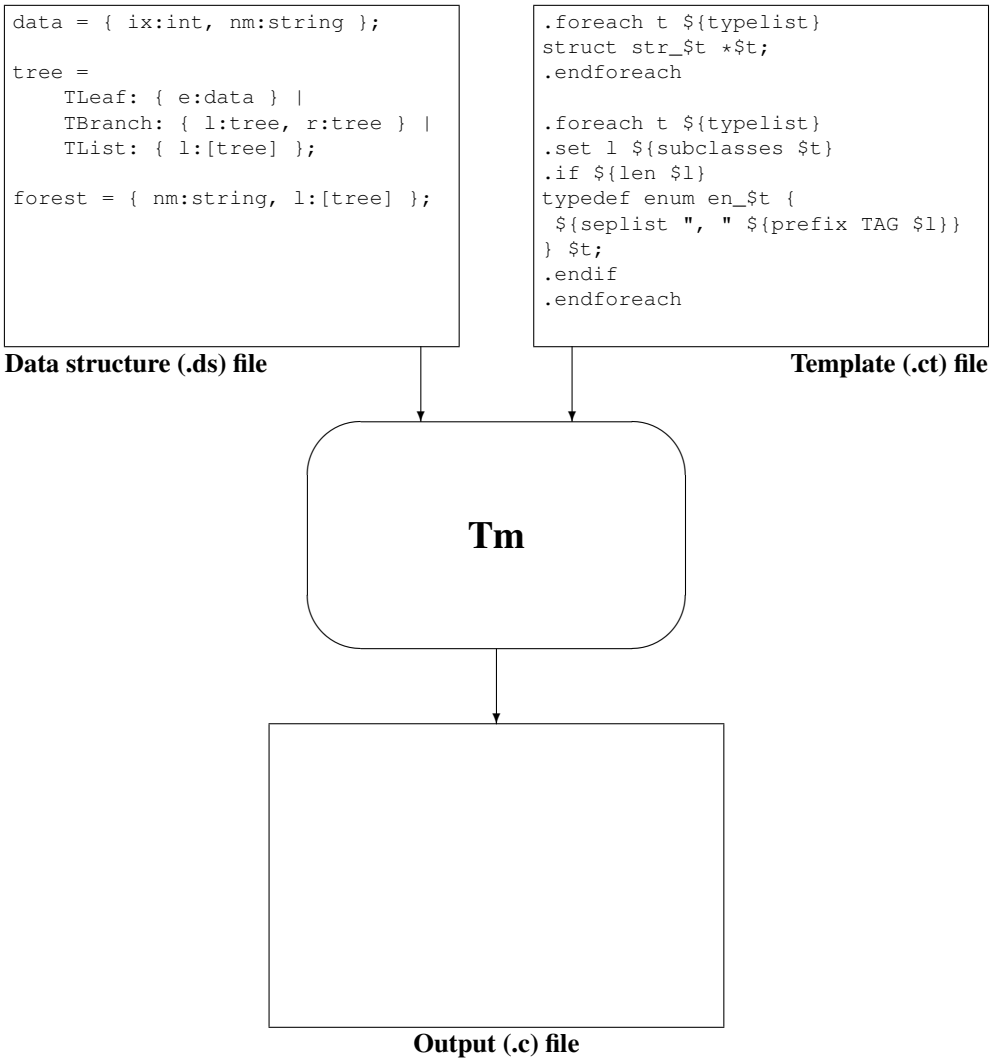
Information about Parallel and Distributed Systems Report Series:
reports@pds.twi.tudelft.nl

Information about Parallel and Distributed Systems Section:
http://pds.twi.tudelft.nl/

Template Manager Reference Manual

edition 7

by C. van Reeuwijk, Delft University of Technology



Contents

Chapter 1

Preface

1.1 Preface to the 7th edition

This edition documents Tm kernel package version 2.0.

The new version has undergone the largest change thus far; so much has changed, in fact, that this deserves an increment in the major version number.

Highlights of the changes:

- A new metatype, *class*, has been added. This required changes in many other places, and made it all the more necessary to assist code generation with well-chosen template functions.
- A new set of data structure access functions has been defined. The old functions are for the moment still supported, but are likely to be removed in a future version.
- A number of new service functions have been added to simplify code requirement analysis and other administration.
- This manual has been updated to reflect the changes in the language, and to present the information more clearly.
- The old C templates have been declared obsolete, and are not documented any more. Appendix C lists a few templates and scripts that can assist in converting code to use the new C template.

Delft, 20 August 1997.

1.2 Preface to the 6th edition

This edition documents Tm version 34.

There have been lots of changes:

- The data structure descriptions allow the definition of inheritance relations. At the moment none of the standard templates supports this, but this may change in future versions.
- The print optimiser now uses an explicit print state, that must be passed to the `print_<type>()` functions.
- Some of the types in the C support library have been renamed to prevent clashes with other libraries. This usually means that the names have been prefixed with `tm` or `tm_`.
- In the template language, there now is a new statement `.redirect` that redirects a range of text to a file.
- In the template language, functions `tmhost` and `tmdate` have been deleted.
- A number of obsolete functions in the C template `calu`, such as `ins_<type>_list()`, have been deleted.
- The C templates now require ANSI C.
- MS-DOS 16-bits code is no longer supported. All traces of 'long' and 'huge' have been removed.

Delft, 13 February 1996.

1.3 Preface to the 5th edition

This edition documents Tm version 32. This will probably be the last edition for some time. The new features of Tm are:

- Support for nested lists. The parser for data structures has supported this for some time, but now the text substitution language and the C templates also support this.
- The C templates no longer require the specification of `otherlists`, and will deduce dependencies for all list types, even for single types that are not defined in the data structure definition file.
- The C templates can generate list reversal functions; they are called `rev_<type>_list()` and `reverse_<type>_list()`.

Support of nested lists required the introduction of a number of new Tm text substitution language functions: `mklist`, `stemname`, `ttypellev` and `ctypellev`.

Delft, 21 August 1992.

1.4 Preface to the 4th edition

This edition documents Tm version 31 and a greatly enhanced version of the C templates and support library. New features of Tm are:

- Macros.
- File inclusion in data structure files.
- Search paths for data structure and template files. If a file with a given name can not be opened, Tm will prefix it with directory names from its search path, and try to open the file with that name.
- Enlargement of data structures. Once a data structure type is defined, subsequent definitions of the same type will enlarge the original definition with new constructors, new tuple fields or new constructor fields.

New features of the Tm C templates and support library:

- A facility to record the origin of Tm block allocations and releases. This way, the origin of pending blocks can be shown.
- The allocation and freeing routines that are used in Tm templates are now available to the user. These functions have two advantages over the standard `malloc()` and friends: allocation errors are handled, and as much as possible, compiler and `lint` warnings that occur with the use of the normal `malloc()` are suppressed.
- Functions for a new primitive type, `symbol` are provided.

Delft, 2 March 1992.

1.5 Preface to the 3rd edition

This edition of the Tm manual documents the various additions to the C templates (including a new template) and the Tm text substitution language. Also, a large number of small errors and ambiguities in the manual have been corrected, and the description of the C templates has been reorganised.

Delft, 22 May 1991.

Chapter 2

Introduction

2.1 Motivation

The transfer of structured data between programs is often carried out using binary or ad-hoc textual formats. However, this can result in ambiguous and non-portable file formats. For example, the Pascal type declaration

```
record foo x,y: integer; c: char; end;
```

is 'equivalent' to the C type definition

```
typedef struct { int x, y; char c; } foo;
```

This does not imply that it is easy to transfer data in these records and structures from one language to the other. Facilities that are provided for this purpose, like `get` and `put` in Pascal and `fread` and `fwrite` in C are useless, and may even cause problems if data is transferred between different implementations of the same language.

An effective way to solve this problem is to introduce a textual representation of the data. The binary read and write routines can then be replaced by text printing and parsing routines. It is now necessary, however, to define a suitable language for this representation; if this is not done properly, it may lead to inconsistency or system dependency.

The program *Tm* (for *template manager*) allows such textual representations to be defined in a special data structure definition language. Tm is able to generate data structure definitions for a number of programming languages from Tm data structure definitions. It can also generate code to read the textual representation into internal data structures, and code to write these internal data structures to the textual representation.

At the moment, Tm can generate code for C, Lisp, Miranda and Pascal. Code generation for any other language is easily added, because the generation is based on *templates*: source texts for the target programming language interspersed with text substitution and repetition commands for Tm.

2.2 About this manual

This document is a reference manual for template manager itself, and the support for C. Templates for other languages are described in separate documents. Since the templates for C are the most extensive, and the most frequently used, they have been included in this manual.

This manual is not intended as a tutorial, but appendix B shows the construction of two small programs that use the Tm templates. Studying these will probably clarify a lot of the things that are discussed there. It is therefore a very good idea to examine this example before trying to understand the rest of the manual. The example programs are available for downloading; compiling and perhaps modifying the code is also very useful.

Fig. 2.1 shows the flow of data through Tm. The syntax and meaning of the data structure definition file is described in section 4. The syntax and meaning of the template file is described in section 5. Section 6 describes the C template and support library.

2.3 Terminology

A few notes on the used terminology: Tm is given a data structure file and a file —called the *template file*— containing source code and text substitution commands. From these files a source file for the target language is generated. This is called *translation*. Target languages are supported by providing *standard templates* for them.

2.4 Used abbreviations

In this manual the following shorthand notations are used:

`<type>`	A type.
`<fieldtype>`	The type of a field.
`<fieldname>`	The name of a field.
`<something>`	The actual contents of this fragment are not important.
`<basename>`	Fill in the value of Tm variable `basename`.

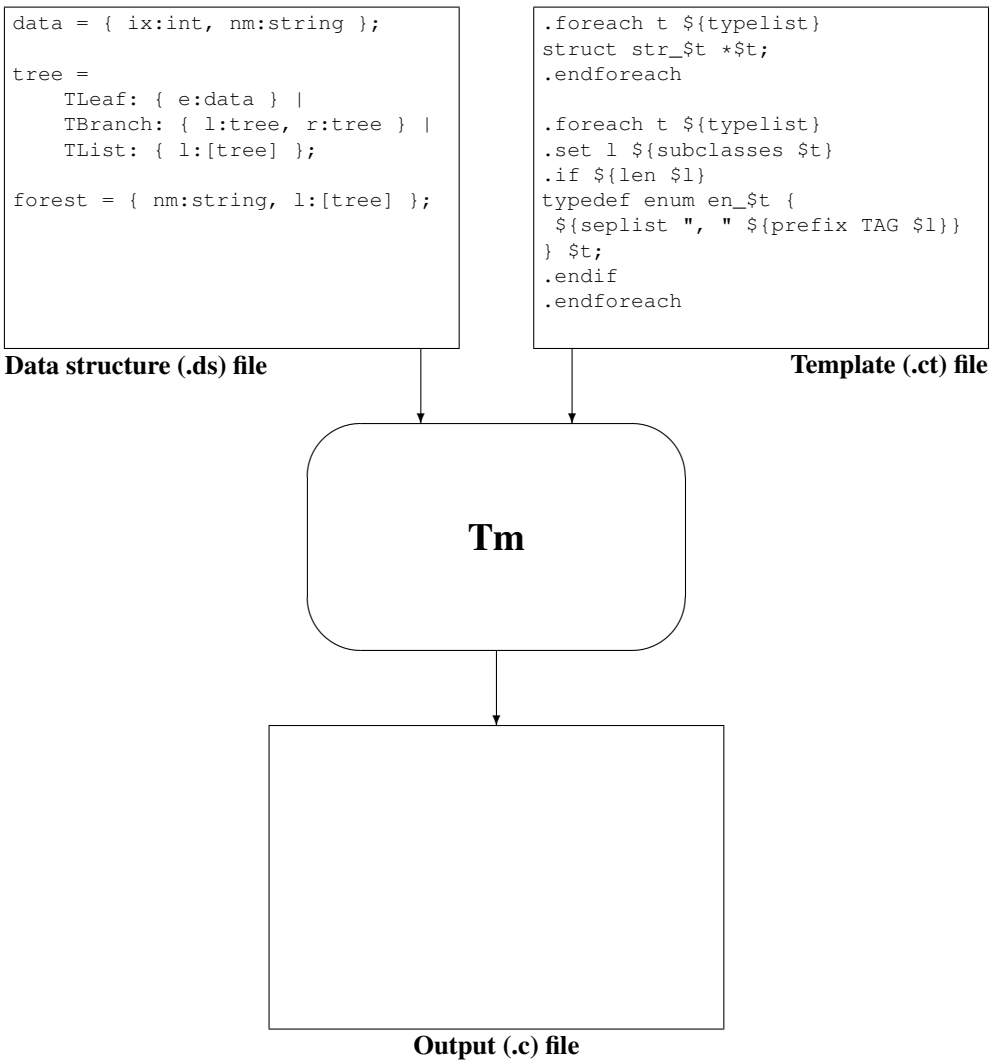


Figure 2.1: The flow of data through Tm. Given a data structure description file (a `.ds` file) and a template file (in this figure a `.ct` file), Tm produces an output file (in this figure a `.c` file).

Chapter 3

Invoking Tm

In its simplest form, Tm is invoked as follows:

```
tm <ds-file> <template-file>
```

Where <ds-file> is a data structure definition file, and <template-file> is a template file. If no <template-file> is given, input is read from standard input. If neither file is given, an empty <ds-file> is assumed, and input is read from standard input.

Also, the following flags can be given:

-d<debugflags>	Set given debug flags.
-e<file>	Redirect errors to file <file>.
-h	Show a help text.
-I<path>	Append <path> to the search path.
-o<file>	Redirect output to file <file>.
-s<var>	Set variable <var> to the empty string.
-s<var>=<val>	Set variable <var> to string <val>.
-v	equivalent to -sverbose.

For the debugging flags, see the online help text given for

```
tm -h
```

Chapter 4

Tm data structures

4.1 Definition of the data structures

A Tm data structure definition file consists of a series of definitions of Tm *types*. A Tm type belongs to one of the following four meta-types: *class*, *tuple*, *constructor* or *constructor base*[1].

The exact syntax of the type definitions is given in appendix A, in the following sections an informal overview of each type is given.

4.1.1 Fields

Each type contains a number of *fields*. A field contains a type name surrounded by an arbitrary number of square bracket pairs ('`[`' and '`]`'). A type without brackets denotes a single element of that type; a type surrounded with one pair of brackets denotes a list of arbitrary length of elements of that type; a type surrounded with two pairs of brackets denotes a list of lists, and so on. Each field must be given a name—called the *field name*—by prefixing the type specification with `<name>:`. This name is used to refer to the field. It is not allowed to use the same field name more than once in the same type, but the same field name can be used in different types.

For example, the following are all valid fields:

```
line:int
file:tmstring
points:[point]
word:[[char]]
```

4.1.2 The class meta-type

In its simplest form, a class type consists of a list of fields separated by commas, and surrounded by curly braces. Like all type definitions, it must be terminated by a semicolon ('`;`'). For example:

[1] In previous versions of Tm constructors were not considered types, only the constructor base type. For reasons of regularity this has been changed.

```
position = { file:tmstring, line:int };
```

The list of fields can be split into multiple *components* separated by + signs. Thus, the following definition would also be allowed:

```
position = { file:tmstring } + { line:int };
```

A class can also inherit from other types. For example:

```
ifStatement = statement + { cond:expr, then:block, else:block };
```

means that the `ifStatement` class inherits from the `statement` class.

A class can be defined to be *virtual* by using the '~=' operator instead of the '=' operator. For example:

```
statement ~= { pos:position };
```

Labelling a class 'virtual', indicates that the class itself will never be created, only subclasses of this class.

To allow compact and clear specification of a class with many subclasses, these subclasses can be specified in the class itself. For example:

```
statement = { pos:position } +
    ifStatement: { cond:expr, then:block, else:block } |
    whileStatement: { cond:expr, body:block } |
    forStatement: { var:string, bound:expr, body:block } |
    assignStatement: { lhs:expr, rhs:expr }
    ;
```

Every labelled component is called an *alternative*; every alternative defines a subclass with the name of its label. A class containing alternatives is flagged as virtual. Thus, the definition above is equivalent with:

```
statement ~= { pos:position };
ifStatement = statement + { cond:expr, then:block, else:block };
whileStatement = statement + { cond:expr, body:block };
forStatement = statement + { var:string, bound:expr, body:block };
assignStatement = statement + { lhs:expr, rhs:expr };
```

A list of components can be grouped by surrounding it with parentheses. For example:

```
expr =
    plusExpr: ( { a:expr } + { b:expr } ) |
    constExpr: { n:int }
    ;
```

4.1.3 The tuple meta-type

A *tuple* consists of a list of fields separated by commas and surrounded by parentheses. Like all type definitions, it must be terminated by a semicolon (';'). For example:

```
position == ( file:tmstring, line:int );
```

The '==' operator indicates that the following is a tuple type.

A tuple can inherit from other types. For example:

```
ifStatement = statement + ( cond:expr, then:block, else:block );
```

A tuple statement cannot contain alternatives or multiple lists of fields.

Clearly, every tuple type can be converted to an equivalent class type. Tuple types are mainly provided for backward compatibility with older versions of TM.

4.1.4 The constructor and constructor-base meta-types

A *constructor* group consists of a number of *constructors* and a *constructor base* type. Every constructor consists of a *name*, followed by a list of fields separated by whitespace. Like all type definitions, it must be terminated by a semicolon (‘`;`’). For example:

```
expr ::=
    constExpr n:int |
    plusExpr a:expr b:expr
    ;
```

Types `constExpr` and `plusExpr` are *constructor* types; type `expr` is a *constructor base* type. The ‘`::=`’ operator indicates that the following is a group of constructor type.

A constructor type can inherit. For example:

```
operatorExpression ::= expression +
    addExpression a:expr b:expr |
    multiplyExpression a:expr b:expr |
    negateExpression x:expr
    ;
```

All constructors types are considered to be subclasses of their constructor base type. All constructor base types are virtual.

If the same constructor base type is defined repeatedly, Tm will merge the definitions. Thus, a definition like

```
|| representation of plot data
plot ::= XYPlot xycolor:int points:[xypoint];
plot ::= YPlot ycolor:int xstart:num xend:num points:[num];
```

is allowed, and defines a constructor base type `plot` with two constructor types.

Clearly, every constructor type can be converted to a similar class type. Constructor types are mainly provided for backward compatibility with older versions of Tm.

4.1.5 Restrictions

A few restrictions are enforced on the type definitions:

- Except for from constructor base types, a type can not have the same name as a previously defined type.
- A type can not, directly or indirectly, inherit from itself.
- A type can not, directly or indirectly, inherit from the same type twice.
- A type can not have two fields with the same name, or inherit a field with the same name as one of its own fields.

Also, some templates (for example the standard C template) allow inheritance from only one type.

4.1.6 Comment

All text starting with the characters '`||`' up to the end of the line is comment that is ignored by Tm. For example:

```
|| The representation of a position in a file
position = { file:string, line:num };
```

4.1.7 File inclusion

Inbetween type definitions, file inclusion commands may be put. They are of the form

```
%include "subfile.ds";
```

The file name (in this example `subfile.ds`) must be surrounded by double quotes. An included data structure file may only contain complete data structure definitions. No check is done on recursive file inclusion.

4.1.8 Definition style

Tm is designed to handle data structures in a large range of languages. It is therefore a bad idea to define data structures that use specific features of a certain language. For example, in Miranda strings are defined as lists of characters (`[char]`). Since this is probably very inefficient in other languages (where a general list manipulation mechanism must be used) it is not a good idea to use `[char]` in the data structure file. A better solution is to introduce a new type — e.g. `string` — and to specify the exact representation of a `string` for each language separately.

4.1.9 A larger example

The following file shows the use of all meta-types in a more-or-less useful definition file. The mixing of classes and constructor types in one file is not realistic, though. In practice either constructors or classes will be used consistently.

This file will be used in the examples of chapter 5 to demonstrate the use of the various template language functions.

```
|| representation of plot data
plot = { xscale:scale, yscale:scale, curves:[curve] };

curve ::=
    XYCurve color:int points:[xypoint] |
    YCurve color:int xstart:num xend:num points:[num] ;

xypoint == ( x:num, y:num );

scale = { title:tmstring } +
    LinScale: { from:num, to:num } |
    LogScale: { from:num, to:num }
    ;

num -> int;
```

4.2 Textual representation of the data

To allow the exchange of data between various programs, a standard text representation is provided. This standard text representation is as follows:

- Each constructor and class is represented by the constructor or class name followed by the representation of the fields of that constructor or class. The entire representation may be surrounded by any number of '`(`' '`)`' pairs to avoid ambiguity.
- Lists are surrounded by '`[`' and '`]`', and the members of the list are separated by commas.
- Tuples are surrounded by exactly *one* pair of parentheses, and all fields of the tuple are separated by commas ('`,`').
- The representation of primitives cannot be prescribed here; it is entirely dependent on the primitive type involved.
- Between constructor, tuple and class fields, constructor and class names and punctuation any number of spaces, tabs, carriage returns, newlines or form-feeds may be inserted. Also, any number of comment lines may be inserted. Comment starts with the characters '`||`' and ends at the end of the line.

For example, a valid text representation of an instance of type `curve` as defined in section 4.1.9 is:

```
XYCurve 1
[
    || The points.
    (1.0, 1.0),
    (2.0, 2.0),
    ((-1e9), 1e50)
]
```

Others are:

```
|| A larger curve
YCurve 1 (-1.0) 1.0 [1,2,3,4,5,6,7,8,9]
(XYCurve 1 [])
```

Instances of classes, such as class

```
expr = exprPlus: { a:expr, b:expr } | exprConst: { n:int };
```

look like this:

```
exprConst 3
exprPlus exprConst 2 exprPlus exprConst 4 exprConst 42
```

Chapter 5

The text substitution language

The standard modules for the various languages are implemented with the text substitution language described in this section.

One should only use the text substitution language directly if the available modules do not meet the requirements, since writing in this language is not exactly a pleasure. The casual user of Tm is expected to use only the standard modules that are included in the Tm package. For this purpose it is sufficient to know the `.include` command (to invoke the modules), the `.insert` command (to insert configuration files), the `.set` command (to set some parameters for the modules), and the `..` command (to add comments to the Tm input).

5.1 Conventions

Boolean functions return the string '`0`' for *false* or the string '`1`' for *true*. At places where a boolean expression is expected, the string '`0`' is *false*, all other strings are *true*.

A *number* is a string of digits with an optional sign before it.

A *word* is either a string of non-blank characters, or a string of arbitrary characters surrounded by double quotes. A quoted string must end before the end of the line, and it may not contain double quote characters. Thus, the following lines each contain one word:

```
a
word
"this is quoted and therefore treated as one word"
```

5.2 Line commands

Each time a dot ('`.`') is encountered as the first character of a line, that line is interpreted as a Tm line command. If the second character of the line is also a dot, the line is comment, and is completely ignored. Otherwise the first word after the dot (possibly

preceded by some spaces and tabs) is interpreted as the line command, and the remainder of the line is interpreted as a list of parameters. The list of parameters is evaluated each time the command is executed. Evaluation of the parameters does *not* occur for termination commands (`.endif`, `.endforeach`, `.endwhile`, etc.).

5.2.1 Assignment

Assignment statements modify the value of a variable. In most of these statements, the new value is stored in the current *context*. A context is created upon invocation of a macro, or the execution of a `.include` command. Upon completion of the macro, or termination of the `.include`, the context is destroyed, and the old value of a variable (if it has a value in the old context) is visible again. This means that it usually is not necessary to worry about side-effects of macros or included files.

Since in some cases it is desirable to allow side-effects, the `.globalappend` and `.globalset` commands are provided. These commands remove all current values of a variable from all contexts, and store the new value in the outermost context. As a result the new value is visible in all contexts.

`.set <var> <val>..<val>`
Set variable `<var>` to the given list of values. The new value of the variable is visible in the current context.

`.append <var> <val>..<val>`
Append the given list of values after the current values of variable `<var>`. The new value of the variable is visible in the current context. For example:

```
.set x 1 2
$x
.append x 3 4
$x
```

will produce:

```
1 2
1 2 3 4
```

`.globalappend <var> <val> <val> <val>`
Append the given list of values after the current value of variable `<var>`. The new value of the variable visible in *all* variable contexts.

`.globalset <expr>`
Set variable `<var>` to the given list of values. The new value of the variable visible to *all* variable contexts.

5.2.2 Datastructure modification

These commands directly modify the data structures that have been read in from the data structure file. There is no way to revert the changes.

`.rename <old> <new>`

Rename all occurrences of type <old> to type <new> at all places where types occur: in type definitions, inheritances, and fields.

It is not allowed to rename a defined type to another defined type, nor is it allowed to introduce inheritance cycles.

`.deletetype <t>..<t>`
Delete the definitions of all given types. Deletion requests for non-existing types is silently ignored.

For example, these commands can be used to remove alias types from data structures. Given the following data structure definition file

```
Expr = { file:String, line:Number } +
    ConstExpr: { n:Number } |
    AddExpr: { left:Expr, right:Expr }
    ;

String -> tmstring;
Number -> int;
```

The following template

```
.macro do_unaliases
.foreach t ${aliases}
.set nw ${alias $t}
.deletetype $t
.rename $t $(nw)
.endforeach
.endmacro
..
.macro dump_types
.foreach t ${typelist}
${metatype $t} $t:
.foreach e ${fields $t}
    field '$e': ${type $t $e}
.endforeach
.endforeach
.endmacro
..
Before:
.call dump_types
.call do_unaliases
After:
.call dump_types
```

Will produce the following output.

```
Before:
class ConstExpr:
    field 'n': Number
class AddExpr:
    field 'left': Expr
    field 'right': Expr
class Expr:
    field 'file': String
    field 'line': Number
alias String:
alias Number:
After:
class ConstExpr:
    field 'n': int
```

```
class AddExpr:
    field 'left': Expr
    field 'right': Expr
class Expr:
    field 'file': tmstring
    field 'line': int
```

5.2.3 Flow control

`.exit <n>`
Stop Tm immediately with exit code `<n>`. This is usually used to halt upon fatal errors in a template.

`.foreach <var> <val>..<val>`
Set variable `<var>` to each of the values `<val>` and do a translation of all following lines up to the next unbalanced `.endforeach` for each of these values of `<var>`. The new value of the variable is only visible in the current context. See section 5.2.1 for a discussion of contexts.

For example:

```
.foreach e 1 2 3 4
$e
.endforeach
```

will produce:

```
1
2
3
4
```

`.if <expr>`

`.for <var> <start> <end> [<stride>]`
Set variable `<var>` to the value `<start>`, and as long as `<var>` is smaller than `<end>`, repeatedly translate all following lines up to the next unbalanced `.endfor`. After each iteration, increment the value of `<var>` with the value `<stride>`, or with 1 if no stride is specified. The new value of the variable is only visible in the current context. See section 5.2.1 for a discussion of contexts.

The values `<start>`, `<end>`, and `<stride>` (if specified), must be numerical. Stride must be 1 or greater.

For example:

```
.for e 0 5 2
$e
.endfor
```

will produce:

```
0
2
4
```

`.if <expr>`

Evaluate <expr>. If <expr> is *true*, translate the following lines up to the next unbalanced .else or .endif and if relevant skip all lines between .else and .endif.

If <expr> is *false*, only translate all lines between the following unbalanced .else and .endif. If no .else is encountered, no translation is done.

For example:

```
.foreach v 0 1 42
.if $v
Yes
.else
No
.endif
.endforeach
```

will produce:

```
No
Yes
Yes
```

.switch <expr>
.case <expr> .. <expr>
.default
.endswitch

Evaluate the parameter of the .switch expression; the result should be a single word. Evaluate all .case lines from top to bottom, and if the list of words of a .case line contains the word that is switched on, execute the lines of that case. Execution stops at the next .case or .default line, or at the .endswitch line.

If the list of cases is exhausted, but no case has been executed, execute the statement in the .default section, if it exists.

The list of words of a case statement may contain any number of words, including no words at all. There can be at most one .default section. Note that more than one case can be executed for a switch value.

For example:

```
.foreach v one two three four five
v: $v
.switch $v
.case one
Case A
.case two three
Case B
.case two five
Case C
.default
Default
.endswitch
.endforeach
```

will produce:

```
v: one
Case A
v: two
Case B
Case C
```

```
v: three
Case B
v: four
Default
v: five
Case C
```

`.while <expr>`
Evaluate `<expr>` and translate the subsequent lines up to the next unbalanced `.endwhile` if `<expr>` is *true*. Repeat this until `<expr>` is *false*.

5.2.4 File inclusion and redirection

`.error <line>`
Write `<line>` to the error output stream. This is useful for debugging or error messages.

`.include <filename>`
Use the Tm commands in `<filename>` as if they were inserted at this position. It is not allowed to leave commands unterminated in `<filename>`. Nested `.include` or `.insert` commands can occur to an arbitrary depth. No checking is done on recursion! All changes to variables that are done in this module are invisible after the `.include` has terminated.

Before a `.include` file is executed, a new variable context is created. This variable context is destroyed upon completion of execution of the file. This behaviour is useful for the invocation of standard translation modules.

If the file cannot be opened, Tm searches a list of directories for it. By default this list contains the directory of the variable `libpath`, but the user may supply additional directories with the `-I` command line option.

`.insert <filename>`
Use the Tm commands in `<filename>` as if they were inserted at this position. It is not allowed to leave commands unterminated in `<filename>`. Nested `.include` or `.insert` commands can occur to an arbitrary depth. No checking is done on recursion!

The variable settings that are done in an `.insert` file remain visible outside the file. This is useful for configuration files or other files that must make global changes to variables.

If the file cannot be opened, Tm searches a list of directories for it. By default this list contains the directory of the variable `libpath`, but the user may supply additional directories with the `-I` command line option.

`.redirect <filename>`
Redirect the output up to the balancing `.endredirect` to the given file. The parameter list of the `.redirect` command must evaluate to exactly one filename. The old contents of the file is overwritten without warning.

`.appendfile <filename>`
Append the output up to the balancing `.endappendfile` to the given file. The parameter list of the `.appendfile` command must evaluate to exactly one filename. If the file does not exist it is created.

5.2.5 Macros

`.call <macro> <parm>..<parm>`

Invoke macro `<macro>` with the given list of parameters. Before execution of the macro, the formal parameters of the macro are set to the values in the parameter list. The number of parameters must match the number of formal parameters, or else an error message is given. To pass a list of words to a formal parameter, surround it with double quotes. Before a macro is executed, a new execution context is created. This context is destroyed upon completion of the macro.

During execution of the macro, `.return` commands may be given to record a return value for the macro, but in the `.call` command this value is ignored. If you want to use it, use the `${call <macro> <parm>..<parm>}` form instead.

For example:

```
.. Given a enum name 'nm' and a list of names 'l', generate
.. a C enum type with globally unique identifiers.
.. The macro uses a global variable 'tagval' to keep track of the
.. used tag values.
.set tagval 0
.macro gen_tags nm l
/* Enum type '$(nm)'. */
typedef enum en_$(nm) {
.set last ${first ${rev $l}}
.foreach e $l
.if ${eq $e $(last)}
    TAG$e = $(tagval)
.else
    TAG$e = $(tagval),
.endif
.globalset tagval $[$(tagval)+1]
.endforeach
} $(nm);

.endmacro
.call gen_tags expr "ExprConst ExprPlus ExprMinus ExprVar"
```

will produce:

```
/* Enum type 'expr'. */
typedef enum en_expr {
    TAGExprConst = 0,
    TAGExprPlus = 1,
    TAGExprMinus = 2,
    TAGExprVar = 3
} expr;
```

`.macro <name> <parm>..<parm>`

Define a new macro `<name>` with the given list of formal parameters. Any previous macro of this name is replaced by the new definition. The body of the macro consists of the lines after the `.macro` up to the next unbalanced `.endmacro`. A macro may contain any Tm command, including other `.macro` definitions. Macros may return a list of words (see the `.return` command), and may be invoked with the line command `.call` or an expression of the form `${call <macro> parm..parm}`. See the description of these commands for further details.

For example (a `$[]` expression evaluates a numerical expression):

```
.. Example macro: calculate !n
.macro fac n
.if $[$n<2]
.return 1
.else
.return $[$n*${call fac $[$n-1]}]
.endif
.endmacro
.. Now test it:
.set n -12
.foreach i 1 2 4 5 7
$i! = ${call fac $i}
.endforeach
n: $n
```

will produce:

```
1! = 1
2! = 2
4! = 24
5! = 120
7! = 5040
n: -12
```

Note that the variable `n` retains the value it had before the invocations of the macro, because all changes to `n` that are done in the macro are removed upon completion of the macro. Only by using the `.globalset` or `.globalappend` command the change can be made permanent.

`.return <val>..<val>`

Record the given values as return value of the current macro. Any previous return value is replaced by this one. Contrary to the `return` command in C, this `.return` command does *not* terminate execution of the macro.

See the example shown for the `.macro` command for an example of the use of the `.return` command.

5.3 Variables

Variables are referenced with the expression `$(<varname>)` or optionally `$<letter>` for single character variables. Each time the characters '`$(`' are encountered, the evaluated string up to the next unbalanced '`)`' is interpreted as the name of a variable. If '`$<letter>`' or '`$<digit>`' is encountered it is interpreted as a reference to a variable with name '`<letter>`' or `<digit>` respectively.

For example:

```
.set b +
.set bla b
$b, $(b), $bla, $(bla), $($(bla))
```

will produce:

```
+, +, +la, b, +
```

Note that the result for '`$bla`' is '`+la`', since the variable '`b`' is referenced, not the variable '`bla`'.

If the characters '`${`' are encountered the evaluated string up to the next unbalanced '`}`' is interpreted as a function invocation, and the result of that function replaces the invocation string, see section 5.4. If the characters '`$[`' are encountered the evaluated string up to the next unbalanced '`]`' is interpreted as an integer expression, and the result of the evaluation of that expression replaces the expression, see section 5.5. If any other character is encountered after the '`$`', it is copied literally.

The following variables are predefined:

`libpath`	The absolute directory path of the library of standard modules. This may differ between installations, so use this variable to locate the library, or let Tm locate it using its search path. By default Tm uses a built-in path, but you can change this by setting the environment variable TMLIBPATH.
`pathsep`	The character used as separation character between elements in a directory path. This is operating system dependent.
`kernel-version`	The version number of the Tm kernel, of which Tm itself is a part.
`tmvers`	The version number of Tm itself. Not that this version number is different from the version number of the kernel, of which Tm is a part. `tmvers` is an integer, and is used in templates to determine whether the version of Tm that is executing the template is recent enough. `tmkernel-version` is a string, and is used to label the generated code.
`verbose`	The verbose flag. It is *true* if the flag `-v` has been given at the command line, else *false*.

The variables `listpre` and `listsuff` are used by the functions `mklist`, `stemname` and `deptype` to construct list type names from elements names or vice versa.

5.4 Functions

Functions are invoked with the expression

```
${<fnname> <par>..<par>}
```

Each time the characters '`${`' are encountered, the string up to the next unbalanced '`}`' is evaluated, and the resulting string is interpreted as a function invocation consisting of a function with the name `<fnname>` possibly followed by a number of parameter words.

5.4.1 Comparison functions

Most comparison functions only work on numbers—and will complain if they get anything else than a number—but `eq` and `neq` work on strings. Do *not* compare numbers with these functions since this is not safe (to these functions '`0`' is different from '`00`').

`eq sa sb`	Return *true* if the string `sa` = `sb`. Strings are equal if they are of the same length and are have the same character at the same position in the string. To compare numbers use ==.
`neq sa sb`	Return *true* if the string `sa` ≠ `sb`. To compare numbers use !=.
`== na nb`	Return *true* if `na` = `nb`. To compare strings use `eq`.
`!= na nb`	Return *true* if `na` ≠ `nb`. To compare strings use `neq`.
`< na nb`	Return *true* if `na` < `nb`.
`<= na nb`	Return *true* if `na` ≤ `nb`.
`> na nb`	Return *true* if `na` > `nb`.
`>= na nb`	Return *true* if `na` ≥ `nb`.

For example:

```
${eq bla bla}, ${eq 00 0}, ${== 00 0}, ${< 1 2}, ${< 2 1}
```

will produce:

```
1, 0, 1, 1, 0
```

5.4.2 Arithmetic functions

All arithmetic functions only work on numbers.

`+ n1..nn`	Return `n1` + ··· + `nn`. The sum of an empty list is 0.
`- na nb`	Return `na` − `nb`.
`* n1..nn`	Return `n1` × ··· × `nn`. The product of an empty list is 1.
`/ na nb`	Return `na`/`nb`.
`% na nb`	Return the remainder of the division of `na` by `nb`.
`max n1..nn`	Return the maximum of `n1` ··· `nn`. The list must contain at least 1 element.
`min n1..nn`	Return the minimum of `n1` ··· `nn`. The list must contain at least 1 element.

For example:

```
+: ${+ 1 2 3 4}
-: ${- 42 36}
*: ${* 1 2 3 4}
/: ${/ 42 9}
%: ${% 42 9}
max: ${max 1 2 3 4}
min: ${min 1 2 3 4}
```

will produce:

```
+: 10
-: 6
*: 24
/: 4
%: 6
max: 4
min: 1
```

5.4.3 Boolean functions

`and b1..bn`	Return *true* if all of `b1 ... bn` are *true*, or else *false*. The `and` of an empty list is *true*.
`not b`	Return *true* if `b` is *false*, or else *true*.
`or b1..bn`	Return *true* if one of `b1 ... bn` is *true*, or else *false*. The `or` of an empty list is *false*.
`if cond a b`	Evaluate the expression `cond`, and if it is *true*, return `a`, else return `b`. If `a` or `b` are absent, an empty string is assumed.

For example:

```
.foreach a 0 1
.foreach b 0 1
.foreach c 0 1
$a $b $c | ${and $a $b $c} | ${or $a $b $c} | ${not $c} | ${if $c Y N}
.endforeach
.endforeach
.endforeach
```

will produce:

```
0 0 0 | 0 | 0 | 1 | N
0 0 1 | 0 | 1 | 0 | Y
0 1 0 | 0 | 1 | 1 | N
0 1 1 | 0 | 1 | 0 | Y
1 0 0 | 0 | 1 | 1 | N
1 0 1 | 0 | 1 | 0 | Y
1 1 0 | 0 | 1 | 1 | N
1 1 1 | 1 | 1 | 0 | Y
```

5.4.4 String functions

`capitalize e..e`	Return the given words with the first character of each word converted to upper case.
`strpad w len pw`	Return word `w` made to length `len` either by truncation or padding at the end with copies of `pw`.
`strlen w`	Return the number of characters in word `w`.
`strindex c w`	Return the index of the first occurrence of character `c` in word `w`, counting from 1, or 0 if `c` does not occur in `w`.
`tolower e..e`	Return the given words with all upper case characters converted to lower case.
`toupper e..e`	Return the given words `e` with all lower case characters converted to upper case.
`tr o n e..e`	Given a string of old characters `o`, a list of new characters `n`, and a list of words `e`, return all `e`, with all characters occuring in `o` replaced with the corresponding character in `n`. The strings `o` and `n` must be of the same length.

For example:

```
.set s a list of WORDS
.set w bicycle
${len $s}, ${tolower $s}, ${toupper $s}, ${capitalize $s}
${strindex c $w}, ${strpad $w 15 -=}, ${strpad $w 3 -}
${tr ae ea creation and reality are no friends}
```

will produce:

```
4, a list of words, A LIST OF WORDS, A List Of WORDS
3, bicycle-=-=-=-=, bic
craetion end raelity era no friands
```

5.4.5 List manipulation functions

`index w e..e`	Search for the first occurrence of word `w` in the list of elements. Return the index of that position counting from 1 upwards, or 0 if `w` does not occur in the list.
`member w e..e`	Return *true* if word `w` occurs in the list of elements, or *false* if it does not.
`shift e..e`	Return the given list with the first element deleted. The `shift` of an empty list is empty.
`first e..e`	Return the first element of the given list. The first element of an empty list is empty.
`seplist s e..e`	Given a separator `s` and a list of elements, create a new string that consists of all the elements in the list separated by a copy of `s`. *Note*: in contrast to other functions that return a list, the elements in the new list are *only* separated by the given separator; no additional spaces are added.
`prefix pf e..e`	Given a prefix `pf` and a list of elements, create a new list that consists of all the elements in the list prefixed by a copy of `pf`.
`suffix sf e..e`	Given a suffix `sf` and a list of elements, create a new list that consists of all the elements in the list suffixed by a copy of `sf`.
`len e..e`	Return the number of elements in the given list.
`sort e..e`	Return a copy of the given list that is lexicographically sorted.
`sizesort e..e`	Return a copy of the given list that is sorted from smallest to largest word. If two words are equally long, they are sorted lexicographically.
`rev e..e`	Reverse the elements in the list.

For example:

```
.set l a long list of nice and different words
index: ${index of $l}, ${index bla $l}
member: ${member of $l}, ${member bla $l}
shift: ${shift $l}
first: ${first $l}
seplist: ${seplist - $l}
prefix: ${prefix z $l}
suffix: ${suffix z $l}
len: ${len $l}
sort: ${sort $l}
sizesort: ${sizesort $l}
rev: ${rev $l}
```

will produce:

```
index: 4, 0
member: 1, 0
shift: long list of nice and different words
first: a
seplist: a-long-list-of-nice-and-different-words
prefix: za zlong zlist zof znice zand zdifferent zwords
suffix: az longz listz ofz nicez andz differentz wordsz
len: 8
sort: a and different list long nice of words
sizesort: a of and list long nice words different
rev: words different and nice of list long a
```

5.4.6 Set operations

These functions implement common operations on sets, although they do not require sets as parameters. However, if the input lists are not sets, some of the operations do not result in a set.

Note that `comm` and `excl` require a separator between two groups of parameters, that cannot be used as a list element. For this the empty string (‘`""`’) is chosen, since it is unlikely that this will occur in a set. However, this means that the empty string cannot occur in a set.

`comm a "" b`	Both `a` and `b` are lists of elements. Return a copy of all elements in `a` that also occur in `b`. This can be used as a ‘set intersection’ operation.
`excl a "" b`	Both `a` and `b` are lists of elements. Return a copy of all elements in `a` that do not occur in `b`. This can be used as a ‘set difference’ operation.
`uniq e1..en`	Return a copy of the given list that is lexicographically sorted, and where all duplicate elements are deleted. This function can be used as a ‘set union’ operation, and to convert an arbitrary list to a set.

For example:

```
.set a a b c d
.set b c d e f
{${comm $a "" $b}}, {${excl $a "" $b}}, {${uniq $a $b}}
```

will produce:

```
{c d}, {a b}, {a b c d e f}
```

5.4.7 Regular expressions

The regular expressions in Tm are based on the regular expressions used in **Unix** shells. A pattern must match the entire string. The following meta-characters are recognized:

`?`	Matches any character.
`\`	Matches the character following it. It is used as an escape character for all meta-characters, including itself. When used in a set (see below), it is treated as an ordinary character.
`[set]`	Matches one of the characters in the set. If the first character in the set is '`^`', it matches a character *not* in the set. A shorthand like `a-z` is used to specify a set of characters `a` up to `z`, inclusive. The special characters '`]`' and '`-`' have no special meaning if they appear as the first characters in the set. Examples: `[a-z]` Any lower case alpha. `[^]-]` Any char except '`]`' and '`-`'. `[^A-Z]` Any char except upper case alpha. `[a-zA-Z]` Any alpha.
`*`	Matches the longest possible span of zero or more arbitrary characters.
`(form)`	Matches what `form` matches, and assigns the matching pattern to one of the numbered patterns. The patterns are numbered from left to right by their opening bracket.

When patterns are substituted the substitution string can also have some meta-characters:

`&`	Is replaced by the entire matched pattern.
`\d`	Where `d` is a digit, is replaced by numbered pattern number '`d`' as matched by '`(form)`'. If the numbered pattern was not assigned in the original string, it is empty. `\0` is replaced by the entire matched pattern.
`\&`	Is replaced by '`&`'.
`\\`	Is replaced by '`\`'.

`filt ps pr e..e`	Given a search pattern `ps` and a replacement pattern `pr`, try to match all elements `e` with `ps`, and for each matching element return a copy of `pr` with the proper substitutions for any `&` and `\d`.
`rmlist pat e..e`	Remove all elements matching `pat` from the list.
`subs ps pr e..e`	Given a search pattern `ps` and a replacement pattern `pr`, copy all elements `e`, and replace all elements matching `ps` with `pr` with the proper substitutions for any `&` and `\d`. All elements that do not match `ps` are copied without change.

For example:

```
.set l bla blup burp zwoing
filt: [${filt b(*)p &(\1) $l}]
rmlist: [${rmlist b*p $l}]
subs: [${subs b(*)p &(\1) $l}], [${subs b(*)p \&(\1) $l}]
```

will produce:

```
filt: [blup(lu) burp(ur)]
rmlist: [bla zwoing]
subs: [bla blup(lu) burp(ur) zwoing], [bla &(lu) &(ur) zwoing]
```

5.4.8 Environment access functions

`getenv nm`	Return the value of environment variable `nm`. If the variable does not exist an empty string is returned.
`isinenv nm`	Determine whether environment variable `nm` exists. Return *true* if it exists, or *false* if it does not.
`dsfilename`	Return the name of the file that describes the data structures.
`searchfile fn..fn`	Given a list of file names, return a list of full names files that have been located using the search path, see the function `searchpath`. If a file can not be found in the search path, the string `?` (a single question mark) is returned for that file name.
`searchpath`	Return the search path for `.include` and `.insert` files. By default, the search path contains the directory the variable `libpath` is set to. If additional directories are specified with the `-I` option, they are appended to the list.
`tplfilename`	Return the name of the current template file.
`tpllineno`	Return the current line number in the current template file.
`matchvar pat`	Given a regular expression as defined in section 5.4.7, return a list of all variables that match this regular expression. In particular, `${matchvar *}` will return a list of all variables. The order of the returned list is arbitrary.
`matchmacro pat`	Given a regular expression as defined in section 5.4.7, return a list of all macros that match this regular expression. In particular, `${matchmacro *}` will return a list of all macros. The order of the returned list is arbitrary.
`defined v`	Return *true* if variable `v` is defined, or *false* otherwise.
`definedmacro m`	Return *true* if macro `m` is defined, or *false* otherwise.

For example:

```
${getenv SHELL}
```

may produce (depending on the value of the `SHELL` environment variable):

```
/bin/csh
```

5.4.9 Time functions

`now` Return the current time as an integer number of seconds from the epoch (system-dependent, but usually 1 jan 1970 00:00:00), similar to the function `time()` in the standard C library.

`formattime time fmt` Given a time in an integer number of seconds from the epoch, (as returned by the function `now`), return a formatted string for this time. The format string has the same syntax as for the function `strftime()` in the C library. That is, all characters are copied to the output string, except for the following character sequences, which are replaced by the listed string:

`%a`	abbreviated weekday name
`%A`	full weekday name
`%b`	abbreviated month name
`%B`	full month name
`%c`	local date and time representation
`%d`	day of the month
`%H`	hour (24-hour clock) (00-23)
`%I`	hour (12-hour clock) (01-12)
`%j`	day of the year (001-366)
`%m`	month (01-12)
`%M`	minute (00-59)
`%p`	local equivalent of AM or PM
`%S`	second (00-61)
`%U`	week number of the year (Sunday as 1st day of week) (00-53)
`%w`	weekday (0-6, Sunday is 0)
`%W`	week number of the year (Monday as 1st day of week) (00-53)
`%x`	local date representation
`%X`	local time representation
`%y`	year without century (00-99)
`%Y`	year with century
`%Z`	time zone name, if any
`%%`	`%`

By the default the format string `%a %b %Y %H:%M:%S` is used.

`processortime` Return the current amount of processor time that has been consumed by the program as an integer number of milliseconds. Note that this is *not* the elapsed time.
Internally, the function `clock()` from the standard C library is used. Although the returned time has a *resolution* of milliseconds, it may have a much worse *accuracy*. The accuracy is system-dependent, but typically there is an error of about 10ms.

For example, the input file:

```
now: ${formattime ${now} "%c %Z"}
```

```
.. Do something that takes some time: build a large list and sort it.
before: ${processortime}
.set n 1000
.set l
.while $[$n>0]
.append l $n
.set n $[$n-1]
.endwhile
.set l ${sort $l}
after: ${processortime}
```

could produce (depending on time and processor speed) the following output:

```
now: Thu Sep 16 14:12:01 1999 GMT
before: 10
after: 120
```

5.4.10 Data structure access functions

`classlist` Return the list of *class* types defined in the data structure file.

`conslist t..t` Given a list of types `t`, return the list of constructors of each type. If a type is not a constructor base type, and empty list is returned for that type.

`ctypelist` Return the list of *constructor* types defined in the data structure file.

`fields t..t` Given a list of types `t`, return the list of field names of each type. Fields in superclasses of this type are *not* listed.

`allfields t..t` Given a list of types `t`, return the list of field names of each type, including inherited fields. The fields are ordered 'height first', so that the fields of superclasses are listed before the fields of their inheritors.

`isvirtual t` Given a type `t`, return `1` if the type is virtual, or `0` if it is not. A type is virtual if it is a constructor base type, if it is a class type containing a labeled component, or if it is a class type defined as virtual (with the `~=` operator).

`metatype t..t` Given a list of types `t`, return the metatype of each type. If `t` is defined in the datastructure definition file, return one of the strings `class`, `tuple`, `constructor`, `constructorbase`, or `alias`. If `t` has the prefix `listpre` and the suffix `listsuff`, return the string `list`; in all other cases return the string `atom`.

`tuplelist` Return the list of *tuple* types defined in the data structure file.

`type t e` Given a type `t` and a field name `e`, construct the proper name of this type from the type name and list level of this field. The list type name is constructed by prefixing the type name with the correct number of copies of `listpre`, and suffixing the type name with the correct number of copies of `listsuff`. This function is equivalent with the expression `${mklist ${typelevel $t $e} ${typename $t $e}}`.
Field `e` may belong to the given type or one of the superclasses of the type.

`types t..t` Given a list of types, return the proper type names of the fields of these types. The types of inherited fields are *not* returned. The proper type names are constructed as described for the function `type`. The types are returned in such an order that they match up with the field names in the order returned by `fields`.

`alltypes t..t` Given a list of types, return the proper type names of the fields of these types, including those of inherited fields. The proper type names are constructed as described for the function `type`. The types are returned in such an order that they match up with the field names in the order returned by `allfields`.

`typelevel t e` Given a type `t` and a field name `e`, return return the level of list bracketing of the field. Thus, a field of type `t` has list level 0, a field of type `[t]` has list level 1, and so on.
Field `e` may belong to the given type or one of the superclasses of the type.

`typelist` Return the list of types defined in the data structure file.

`typename t e` Given a type `t` and a field name `e`, return the type of that field. Field `e` may belong to the given type, or to one of the superclasses of the type.

For example, given the data structure definitions:

```
|| representation of plot data
plot = { xscale:scale, yscale:scale, curves:[curve] };

curve ::=
    XYCurve color:int points:[xypoint] |
    YCurve color:int xstart:num xend:num points:[num] ;

xypoint == ( x:num, y:num );

scale = { title:tmstring } +
    LinScale: { from:num, to:num } |
    LogScale: { from:num, to:num }
    ;

num -> int;
```

and the template:

```
.set listpre [
.set listsuff ]
classlist: ${classlist}
ctypelist: ${ctypelist}
tuplelist: ${tuplelist}
.foreach t ${typelist}
metatype $t: ${metatype $t}
isvirtual $t: ${isvirtual $t}
allfields $t: ${fields $t}
alltypes $t: ${alltypes $t}
.foreach e ${fields $t}
  type $t $e: ${type $t $e}
  typename $t $e: ${typename $t $e}
  typelevel $t $e: ${typelevel $t $e}
.endforeach
.endforeach
```

the following output is produced:

```
classlist: plot LinScale LogScale scale
ctypelist: curve
tuplelist: xypoint
metatype plot: class
isvirtual plot: 0
allfields plot: xscale yscale curves
alltypes plot: scale scale [curve]
  type plot xscale: scale
  typename plot xscale: scale
  typelevel plot xscale: 0
  type plot yscale: scale
  typename plot yscale: scale
  typelevel plot yscale: 0
  type plot curves: [curve]
  typename plot curves: curve
  typelevel plot curves: 1
metatype curve: constructorbase
isvirtual curve: 1
allfields curve:
alltypes curve:
metatype XYCurve: constructor
isvirtual XYCurve: 0
allfields XYCurve: color points
alltypes XYCurve: int [xypoint]
  type XYCurve color: int
```

```
  typename XYCurve color: int
  typelevel XYCurve color: 0
  type XYCurve points: [xypoint]
  typename XYCurve points: xypoint
  typelevel XYCurve points: 1
metatype YCurve: constructor
isvirtual YCurve: 0
allfields YCurve: color xstart xend points
alltypes YCurve: int num num [num]
  type YCurve color: int
  typename YCurve color: int
  typelevel YCurve color: 0
  type YCurve xstart: num
  typename YCurve xstart: num
  typelevel YCurve xstart: 0
  type YCurve xend: num
  typename YCurve xend: num
  typelevel YCurve xend: 0
  type YCurve points: [num]
  typename YCurve points: num
  typelevel YCurve points: 1
metatype xypoint: tuple
isvirtual xypoint: 0
allfields xypoint: x y
alltypes xypoint: num num
  type xypoint x: num
  typename xypoint x: num
  typelevel xypoint x: 0
  type xypoint y: num
  typename xypoint y: num
  typelevel xypoint y: 0
metatype LinScale: class
isvirtual LinScale: 0
allfields LinScale: from to
alltypes LinScale: tmstring num num
  type LinScale from: num
  typename LinScale from: num
  typelevel LinScale from: 0
  type LinScale to: num
  typename LinScale to: num
  typelevel LinScale to: 0
metatype LogScale: class
isvirtual LogScale: 0
allfields LogScale: from to
alltypes LogScale: tmstring num num
  type LogScale from: num
  typename LogScale from: num
  typelevel LogScale from: 0
  type LogScale to: num
  typename LogScale to: num
  typelevel LogScale to: 0
metatype scale: class
isvirtual scale: 1
allfields scale: title
alltypes scale: tmstring
  type scale title: tmstring
  typename scale title: tmstring
  typelevel scale title: 0
metatype num: alias
isvirtual num: 0
allfields num:
alltypes num:
```

5.4.11 Alias functions

These functions are used to access alias types.

`aliases`	Return the list of alias types defined in the data structure file.
`alias t..t`	Apply the alias definitions to the given types. For each type, if no alias type is defined, return the type itself, else return the alias of the type.

For example, given the data structure definitions shown on page 13 and the following input file:

```
aliases: [${aliases}]
alias use: [${alias num number num_list number_list}]
```

the following output is produced:

```
aliases: [num]
alias use: [int number num_list number_list]
```

5.4.12 Class inquiry functions

The functions listed below all access the inheritance information of the defined types.

`inherits t..t`	Given a list of types, return the list of types this type inherits directly. If a type is inherited by several types, it will be listed several times.
`inheritors t..t`	Given a list of types, return the list of types that inherit directly from this type.
`subclasses t..t`	Given a list of types, return the list of types that inherit directly or indirectly from these types.
`superclasses t..t`	Given a list of types, return the list of types this type inherits directly or indirectly. If a type is inherited by several types, it will be listed several times.

For example:

```
.foreach t ${typelist}
inherits $t: ${inherits $t}
inheritors $t: ${inheritors $t}
subclasses $t: ${subclasses $t}
superclasses $t: ${superclasses $t}
.endforeach
```

will produce (assuming the data structure definitions of page 13):

```
inherits plot:
inheritors plot: XYPlot YPlot
subclasses plot: XYPlot YPlot
superclasses plot:
inherits XYPlot: plot
```

```
inheritors XYPlot:
subclasses XYPlot:
superclasses XYPlot: plot
inherits YPlot: plot
inheritors YPlot:
subclasses YPlot:
superclasses YPlot: plot
inherits xypoint:
inheritors xypoint:
subclasses xypoint:
superclasses xypoint:
inherits LinScale: scale
inheritors LinScale:
subclasses LinScale:
superclasses LinScale: scale
inherits LogScale: scale
inheritors LogScale:
subclasses LogScale:
superclasses LogScale: scale
inherits scale:
inheritors scale: LinScale LogScale
subclasses scale: LinScale LogScale
superclasses scale:
```

5.4.13 Code generation service functions

These functions implement some of the complicated operations that are necessary during code generation.

One definition must be introduced: A type t *depends* on a type s if s is used in at least one of the elements of t, or if t has a type in one of its elements that depends on s. Also, a list of a type depends on that type. Primitive types depend on nothing.

For example: in the data structure on page 13 type `plot` depends on the single types `xypoint`, `int`, and `num`, and on the list types `[xypoint]` and `[num]`.

Several functions in this group refer to the Tm variables `listpre` and `listsuff` for a prefix and a suffix respectively to construct a list name from a single type name. If one of these variables is not defined, an empty string is assumed.

`depsort t..t` Given a list of types, rearrange them so that all types that contain a certain type are placed after the type itself. Thus, if a type `ta` uses a type `tb`, `tb` will be placed before `ta`; this is in effect a topological sort. The function will complain about circularities.
List types are considered to contain their element type.
The function does not consider inheritance relations.

`inheritsort t..t` Given a list of types, rearrange them so that all types that all types are placed after the types they inherit from.

`delisttypes t..t` Given a list of types, return a list of the element types of the list types by removing the list prefix `listpre` the list suffix `listsuff`. For types that don't have this prefix and suffix nothing is returned.

`listtypes t..t` Given a list of types, return a list of all types that are prefixed by `listpre` and suffixed by `listsuff`.

`mklist n e..e` Given a list level `n` and a list of type names `e`, construct for each type a list name for the given list level by prefixing each type name with `n` copies of `listpre` and suffixing each type name with `n` copies of `listsuff`.

`singletypes t..t` Given a list of types, return a list of all types that are not prefixed by `listpre` or are not suffixed by `listsuff`.

`stemname t..t` Given a list of list type names `t`, return a list of stem names. All pairs of list prefixes and suffices are stripped from all the type names. The list prefix is taken from the variable `listpre`, the list suffix is taken from the variable `listsuff`. If both are empty the list is returned unchanged, and an error is emitted.

`virtual t..t` Given a list of types `t`, return all types that are virtual.

`nonvirtual t..t` Given a list of types `t`, return all types that are not virtual.

`reach t..t "" b..b` Given a list of types `t`, and a list of blocking types `b`, return all types that can be reached from the types `t` without visiting any of the types `b`. The list of starting types and the list of blocking types is separated by an empty string (`""`). If there is no empty string, there are no blocking types.
The *direct reach* of a class, tuple, constructor or constructor-base type are the types of its fields and inheritors; the direct reach of a list type is its element type, the direct reach of an alias type is the target type of the alias, and the direct reach of an atomic type is empty.
The *reach* of a type is the union of the type itself, the direct reach of the type, and the reach of the types in the direct reach of the type. In other words, the reach is the transitive closure of the direct reach.

For example:

```
inheritsort ${typelist}:
  ${inheritsort ${typelist}}
depsort ${typelist}:
  ${depsort ${typelist}}
```

```
.set listpre <
.set listsuff >
.set tl a <b> <<c>> <<<d>>> e> <f
delisttypes: [${delisttypes $(tl)}]
listtypes: [${listtypes $(tl)}]
mklist: [${mklist 3 a z}]
singletypes: [${singletypes $(tl)}]
stemname: [${stemname $(tl)}]
.foreach t int ${typelist}
reach $t: ${reach $t}
.endforeach
```

will produce (assuming the data structure definitions of page 13):

```
inheritsort plot curve XYCurve YCurve xypoint LinScale LogScale scale num:
  plot curve XYCurve YCurve xypoint scale LinScale LogScale num
depsort plot curve XYCurve YCurve xypoint LinScale LogScale scale num:
  curve XYCurve scale plot num YCurve xypoint LinScale LogScale
delisttypes: [b <c> <<d>>]
listtypes: [<b> <<c>> <<<d>>>]
mklist: [<<<a>>> <<<z>>>]
singletypes: [a e> <f]
stemname: [a b c d e> <f]
reach int: int
reach plot: tmstring int num LinScale LogScale scale <curve> <xypoint> xypoint XYCurve <num>
reach curve: int <xypoint> num xypoint XYCurve <num> YCurve curve
reach XYCurve: int <xypoint> num xypoint XYCurve
reach YCurve: int num <num> YCurve
reach xypoint: int num xypoint
reach LinScale: tmstring int num LinScale
reach LogScale: tmstring int num LogScale
reach scale: tmstring int num LinScale LogScale scale
reach num: int num
```

5.4.14 Obsolete data structure access functions

These functions have been used in previous version of Tm to retrieve information about the data structures. They have been replaced by the functions in the previous sections, and should not be used in new templates.

`ttypeclass t e` — Given a type `t` and a field name `e`, return the type class of that field. Possible type classes are `single` and `list` for a single element and a list of elements respectively.
Field `e` may belong to the given type or one of the superclasses of the type.
This function is obsolete; use `typelevel` instead.

`ttypelist` — Equivalent to `tuplelist`.

`telmlist t` — Equivalent to `field`.

`ttypeclass t e` — Equivalent to `typeclass`.

`ttypellev t e` — Equivalent to `typelevel`.

`ttypename t e` — Equivalent to `typename`.

`celmlist t c [inherited]` — Given a constructor type `t`, and a constructor name `c`, return the list of element names of that constructor. If `t` is a tuple type, an error message is given.
If a third parameter is given, and this parameter is the string `inherited`, the fields of the superclasses of this constructor are also listed. The fields are ordered 'height first', so that the fields of superclasses are listed before the fields of the types that inherit these superclasses.
This function is obsolete; use `${fields c}` instead.

`ctypeclass t c e` — Given a constructor type `t`, a constructor `c` and a constructor element name `e`, return the type class of that element. Possible type classes are `single` and `list` for a single element and a list of elements respectively. If `t` is a tuple type, an error message is given.
This function is obsolete; use `${typeclass c}` instead.

`ctypellev t c e` — Given a type `t`, a constructor `c` and a constructor element name `e`, return return the level of list bracketing of the element. Thus, an element of type `t` has list level 0, an element of type `[t]` has list level 1, and so on. If `t` is a tuple type, an error message is given.
This function is obsolete; use `${typelevel c}` instead.

`ctypename t c e` — Given a type `t`, a constructor `c` and a constructor element name `e`, return the type of that element. If `t` is not a constructor base type, an error message is given.
This function is obsolete; use `${typename c}` instead.

5.4.15 Deferred evaluation

`call m p1..pn` Given a macro name `m` and a list of parameters `p1..pn`, invoke macro with the given list of parameters. Before execution, the formal parameters of the macro are set to the values in the parameter list. To pass a list of words to a formal parameter, it must be surrounded by double quotes. The number of parameters must match the number of formal parameters, or else an error message is given. Note that each parameter word corresponds to one formal parameter. Macros 'see' the variable values and macros that are in effect at the moment of execution of the macro. All changes that are made to variables or macros within a macro are invisible outside the macro. During execution of the macro at least one `.return` command must be given to record a return value for the macro. The macro is not allowed to generate output, and therefore may only contain line commands. If you want to generate output, use the `.call <macro> <parm>..<parm>` form instead.

`eval e1..en` Given a list of expressions `e1..en`, evaluate all expressions, and return them in a list.

For example:

```
.set x "$${+ 12 45}"
x=$x, e = ${eval "$x"}
.macro square l
.set res
.foreach e $l
.append res ${* $e $e}
.endforeach
.return $(res)
.endmacro
square: ${call square "1 2 3 4 5"}
```

will produce:

```
x=${+ 12 45}, e = 57
square: 1 4 9 16 25
```

5.5 Arithmetic expressions

Arithmetic expressions have the form

```
$[<expr>]
```

Each time the characters '`$[`' are encountered, the string up to the next unbalanced '`]`' is evaluated, and the resulting string is interpreted as an integer expression. The following operators are available, where the number in the first column indicates priority; the lower the number, the stronger an operator binds:

priority	symbol	function
0	`()`	Priority brackets.
1	−	Unary minus.
1	`!`	Boolean not.
1	+	Unary plus.
2	*	Multiplication.
2	/	Division.
2	`%`	Modulus.
3	+	Addition.
3	−	Subtraction.
4	`!=`	Not equal to.
4	==	Equal to.
4	<=	Less or equal.
4	<	Less.
4	>=	Greater or equal.
4	>	Greater.
5	`&`	Boolean and.
5	`|`	Boolean or.

The integer '0' represents 'false', all other integers represent 'true'. Boolean operators always return 'true' as '1'. The binary operators +, −, *, /, and `%` are left binding. That is:

$$a \text{ op } b \text{ op } c = (a \text{ op } b) \text{ op } c$$

The remaining binary operators are right binding. That is:

$$a \text{ op } b \text{ op } c = a \text{ op } (b \text{ op } c)$$

For example:

```
.set n 42
$[$n*$n], $[$n&1], $[1>2], $[00==0], $[3-2-1], $[16/4/2]
```

will produce:

```
1764, 1, 0, 1, 2, 8
```

Chapter 6

C support

In this chapter a detailed description of the C templates is given. In appendix B an example project is shown that demonstrates the use of these templates.

The C support consists of two template files: `tmc.ht` is used to generate a header file, and `tmc.ct` is used to generate a code file. Also, a library is provided that contains service functions needed by the templates, and the equivalent of the generated routines for a number of C types.

The functions are described using ANSI C style type specifications, and require an ANSI C compiler.

6.1 Restrictions

These templates impose the following restrictions:

- Tuple types cannot inherit or have inheritors.
- Only inheritance from a single type is allowed (no multiple inheritance).
- Virtual types must have at least one inheritor.

6.2 Type representation

For all Tm meta-types a representation in C must be chosen. This is shown in the sections below. As illustration the type representation of the following types will be shown:

```
tree == ( v:int, t:[tree] );

btree ::= BTree l:btree r:btree | BLeaf v:int;

ctree =
```

```
treeBranch: { v:int, l:ctree, r:ctree } |
treeNull: {}
;
```

6.2.1 Tuple representation

For tuple types, the representation of the type is simply a pointer to a structure containing all the fields of the tuple. For example:

```
typedef str_tree *tree;

struct str_tree {
    int v;
    tree_list t;
};
```

6.2.2 Class representation

Class types are pointers to a structure containing a tag and all the fields of the type. The tag is used to distinguish between the class and the possible subclasses. To access the fields of a subclass, the pointer must be cast to the appropriate subclass type. For example, constructor type `ctree` is represented as:

```
typedef str_ctree *ctree;
typedef str_treeBranch *treeBranch;
typedef str_treeNull *treeNull;

enum en_ctree {
    TAGtreeBranch, TAGtreeNull
};

struct str_ctree {
    tags_ctree tag;
};

struct str_treeBranch {
    tags_ctree tag;
    int v;
    ctree l;
    ctree r;
};

struct str_treeNull {
    tags_ctree tag;
};
```

The possible tags for a type are enumerated in the `enum` type `tags_<type>`. Note that no tag is generated for `ctree`, because this type is virtual.

If a C++ compiler is detected (the preprocessor variable `__cplusplus` is defined), a slightly different representation is used:

```
typedef str_ctree *ctree;
typedef str_treeBranch *treeBranch;
typedef str_treeNull *treeNull;

enum en_ctree {
    TAGtreeBranch, TAGtreeNull
```

```
};

class str_ctree {
public:
    tags_ctree tag;
};

class str_treeBranch : str_ctree {
public:
    int v;
    ctree l;
    ctree r;
};

class str_treeNull : str_ctree {
public:
};
```

This representation avoids some of the casts that are needed in the C version for `rdup_<type>` and `new_<type>`.

6.2.3 Constructor and constructor base representation

Constructor types are represented in a similar way to class types: they are pointers to a structure containing a tag and all the fields of the type. The tag is used to distinguish between the class and the possible subclasses. To access the fields of a subclass, the pointer must be cast to the appropriate subclass type. For example, constructor type `btree` is represented as:

```
typedef str_btree *btree;
typedef str_BTree *BTree;
typedef str_BLeaf *BLeaf;

typedef enum en_btree {
    TAGBTree, TAGBLeaf
} tags_btree;

struct str_BTree {
    tags_btree tag;
    btree l;
    btree r;
};

struct str_BLeaf {
    tags_btree tag;
    int v;
};

struct str_btree {
    tags_btree tag;
};
```

The constructor base type is represented by a separate structure containing a `tag` field to indicate the actual constructor. The possible tags for a type are enumerated in the `enum` type `tags_<type>`.

As for classes, a slightly different representation is used if the preprocessor variable `__cplusplus` is used:

```
typedef str_btree *btree;
typedef str_BTree *BTree;
typedef str_BLeaf *BLeaf;

typedef enum en_btree {
    TAGBTree, TAGBLeaf
} tags_btree;

struct str_btree {
    tags_btree tag;
};

struct str_BTree : str_btree {
    btree l;
    btree r;
};

struct str_BLeaf : str_btree {
    int v;
};
```

6.2.4 List representation

List types are defined as pointers to a list description structure. For example:

```
typedef struct str_tree_list *tree_list;

struct str_tree_list {
    unsigned int sz;     /* current number of elements */
    unsigned int room;   /* maximum number of elements */
    tree *arr;           /* ptr to array of 'room' elements */
};
```

Note that in the C representation `[tree]` is called `tree_list`, and `[[tree]]` is called `tree_list_list`. The list description structure contains a pointer to an array, the current length of the list, and the maximum length of the list before re-allocation is necessary.

6.2.5 Casting macros

To access subclasses of a class or constructor base, a cast must be used. For example:

```
void clear_leaf( btree tree )
{
    if( tree->tag == TAGBLeaf ){
        /* This is not recommended. */
        ((BLeaf) tree)->v = 0;
    }
}
```

It is not recommended to access elements this way. Instead, one should use the `to_<type>()` and `to_const_<type>()` macros that are provided for that purpose:

```
void clear_leaf( btree tree )
{
    if( tree->tag == TAGBLeaf ){
        to_BLeaf( tree )->v = 0;
    }
}
```

These macros have as advantage that they can also be used in older Tm templates (the latest version of these templates provide them), that future templates are likely to use them, and that the casting macro can contain a check on the appropriateness of the cast (not yet implemented).

Since in the current template every subclass is an independent type, it is also possible to have variables of these types. This allows constructs such as:

```
void clear_leaf( btree tree )
{
    if( tree->tag == TAGBLeaf ){
        BLeaf leaf = to_BLeaf( tree );

        leaf->v = 0;
    }
}
```

6.2.6 Null pointers

For each type and type list a suitable `NIL` pointer is defined. For example:

```
#define treeNIL (tree)0
#define btreeNIL (btree)0
#define tree_listNIL (tree_list)0
```

The following functions show how all the type representations (in this case the types `tree`, `btree`, `ctree` and the list type `[tree]`) are used in a program: Each function returns the sum of the leave or node values in a given tree.

```
/* Return the sum of all the node values in tree 't'. */
int sum_tree( tree t )
{
    unsigned int ix;
    tree_list tl;
    int s;

    s = 0;
    tl = t->t;
    for( ix=0; ix<tl->sz; ix++ ){
       s += sum_tree( tl->arr[ix] );
    }
    return t->v + s;
}

/* Return the sum of all the leave values in ctree 't'. */
int sum_ctree( ctree t )
{
    int s;

    switch( t->tag )
    {
        case TAGtreeBranch:
            s = sum_ctree( to_treeBranch(t)->l ) +
                sum_ctree( to_treeBranch(t)->r ) +
                to_treeBranch(t)->v;
            break;

        case TAGtreeNull:
            s = 0;
            break;
```

```
        default:
            tm_badtag( __FILE__, __LINE__, (int) t->tag );
            break;
    }
    return s;
}

/* Return the sum of all the leave values in btree 't'. */
int sum_btree( btree t )
{
    int s;

    switch( t->tag )
    {
        case TAGBTree:
            s = sum_btree( to_BTree(t)->l ) + sum_btree( to_BTree(t)->r );
            break;

        case TAGBLeaf:
            s = to_BLeaf(t)->v;
            break;

        default:
            tm_badtag( __FILE__, __LINE__, (int) t->tag );
            break;
    }
    return s;
}
```

6.2.7 Class hierarchy inquiry functions

Since Tm allows chains of inheritance relation of arbitrary length, it is sometimes useful to determine whether an element belongs to a given subclass, without having to enumerate all members of the subclass. For this purpose the `is_<type>()` macros are provided.

`bool is_<type>( <supertype> e );`
`bool is_<type>_list( <type>_list e );`

Given an element return `TRUE` if it is a member of type `<type>` or one of its subclasses, or `FALSE` if it has the type of another subclass of the root class of `<type>`. For classes that do not have superclasses, the macro degenerates to the constant `TRUE`.

For elements that are member of another type tree, the result is unspecified.

These macros are currently only available in the `tmc` template.

6.3 Dynamic memory allocation

One of the important features of C is that it has dynamic memory allocation available through the functions `malloc()` and `free()`. These functions are, however, sensitive to two forms of abuse. First, blocks that are requested from `malloc()` may be released with `free()`, but used after that or even passed to `free()` for a second time. This may lead to serious problems that may be hard to find, since the symptoms are often not related to the cause, differ between machines and sometimes even

differ between runs. Another form of abuse of `malloc()` is that requested blocks are never released. This is not inherently dangerous, and is even a sensible practice in small programs. For large programs, however, it is undesirable, since a large amount of memory may be consumed in these blocks, leading to performance degradation or program failure.

Therefore, in the ideal situation each call to `malloc()` is balanced with *exactly* one call to `free()`.

Tm offers two facilities to ensure that the `malloc()` and `free()` calls are in balance. First, Tm can generate code to count the number of calls to `new_<type>` and `fre_<type>`. The code will be generated if you request the generation of the function `stat_<basename>` or `get_balance_<basename>`. For each file of generated C code you may request a function `stat_<basename>` to print the allocation and freeing statistics of the functions in that file. You can call this function at a place where you expect the statistics to be in balance (usually at the end of the program) to print the actual statistics. To facilitate checking this, the function `get_balance_<basename>()` can be generated. It returns the current balance state (there are less, more, or the same number of allocations as deallocations). This can be used to routinely check the balance at the end of the program, and issue a warning if necessary.

It is highly recommended to ensure that a program always terminates with balanced allocation statistics.

For the types `tmstring` and `tmtext` in the `tmc` library the functions `stat_tmstring()`, `get_balance_tmstring()`, and `stat_tmtext()` and `get_balance_tmtext()` are already provided.

If the statistics reveal that more blocks have been freed than have been allocated, there is a serious problem in the software that deserves immediate attention. If the statistics reveal that less blocks have been freed than have been allocated, the program will function properly, but you should repair this eventually, since it may conceal an imbalance in the opposite direction.

To find which blocks are never freed, Tm provides another facility: if the code is compiled with the C preprocessor variable `LOGNEW` defined, the source file and line of all Tm block allocations are recorded in a special list. The entries for all blocks that are freed are removed from the list, so that the list always contains a description of the currently allocated blocks. A report on this list can be written to a file with the function `report_lognew()`. Usually the cause of the imbalance is easily determined from this information.

The `LOGNEW` facility is a powerful debugging aid, but it has an important disadvantage: *all* code must be compiled with this option.

To find the place where a second `fre_<type>` of a block is done is more difficult; general `malloc()` debugging packages may be useful for this.

6.4 Creation and destruction

It is necessary to create and destroy instances of Tm types explicitly. The functions in this section handle this. All dynamic memory allocation in Tm templates and routines is done through the functions `tm_malloc()` and `tm_realloc()`.

Dynamic memory allocation is very sensitive to errors, and therefore Tm provides extensive support to detect and repair such bugs, see section 6.3.

`<type>_list new_<type>_list();`
Create a new, empty, list. No array is allocated to store the elements, since this is not necessary when there are no elements. When necessary an array will be allocated through `setroom_<type>_list()`.

`<type> new_<type>( <elmtype> f1, .. <elmtype> fn );`
Create a constructor, class or tuple, and set the elements to the value of the given parameters. For constructor and class types, the `tag` field is set to the appropriate value.

If the type is a subclass, the fields of the superclasses must also be given, in the order described for the Tm template function `fields`.

For example, if `expr` is defined as:

```
expr = { line:int } + exprPlus: { a:expr, b:expr } | exprConst: { n:int };
```

The following functions can be generated:

```
exprPlus new_exprPlus( int line, expr a, expr b );
exprConst new_exprConst( int line, int n );
```

`<type>_list setroom_<type>_list( <type>_list l, unsigned int rm );`
Specify that list `l` must have room for at least `rm` elements. In the templates that use an array to represent the list, this function will ensure that the array has sufficient room to store `rm` elements; in the linked list template this function is a dummy. The room in a list is never reduced.

Functions that add elements to a list (such as `append_<type>_list()`, `concat_<type>_list()` and `insert_<type>_list()`) use this function implicitly, the user only needs these functions for efficiency reasons (to prevent repeated enlargement of the array) or to build new list functions.

`void fre_<type>( <type> e );`
`void fre_<type>_list( <type>_list l );`
Destroy an instance of `<type>` or `<type>_list`.

`void rfre_<type>( <type> e );`
`void rfre_<type>_list( <type>_list e );`
Recursively destroy all elements of `e` and `e` itself.

`void stat_<basename>( FILE *f );`
Write statistics of allocation and freeing for each type and type list associated with `<basename>` to file `f`. The code that counts allocations and deallocations is only generated if `stat_<basename>` or `get_balance_<basename>` is requested.

`int get_balance_<basename>( void );`
Return `-1` if any of the types managed in `<basename>` has been freed more often than allocated. Return `1` if any of the types managed in `<basename>` has been freed less often than allocated. Return `0` if the allocation and freeing are in balance. The code that counts allocations and deallocations is only generated if `stat_<basename>` or `get_balance_<basename>` is requested.

6.4.1 List manipulation

`<type>_list append_<type>_list( <type>_list l, <type> e );`
Given a list `l` and an element `e`, put element `e` after the elements of `l`. Enlarge the array if necessary. Element `e` is now 'owned' by list `l`.

`<type>_list concat_<type>_list( <type>_list la, <type>_list lb );`
Given two lists `la` and `lb`, append the elements of `lb` after the elements of `la`. Enlarge the array of `la` if necessary. The elements of `lb` are now 'owned' by `la`, and `lb` itself is destroyed.

`<type>_list insert_<type>_list( <type>_list l, unsigned p, <type> e );`
Insert element `e` at position `p` in list `l`. All elements at and after position `p` are moved up in the list (they have their index in the list incremented). If `p` is greater than or equal to the length of `l`, the element is appended after `l`. Element `e` is now 'owned' by list `l`.

```
<type>_list insertlist_<type>_list(
 <type>_list la,
 unsigned p,
 <type>_list lb
);
```
Insert list `lb` at position `p` in list `la`. All elements at and after position `p` are moved up in the list (they have their index in the list incremented). If `p` is greater than or equal to the length of `l`, the elements are appended after `la`. All elements in `lb` are now 'owned' by list `la`, and the list container of `lb` is destroyed.

If `la` is `<type>_listNIL`, an error is generated. If `lb` is `<type>_listNIL`, `la` is returned unchanged.

`<type>_list delete_<type>_list( <type>_list l, unsigned int pos );`
Delete the element at position `pos` in list `l`. The element is freed using `rfre_<type>()`. All elements after position `pos` are moved down (have their index in the list decremented). If `pos` is greater than or equal to the length of `l`, nothing happens.

```
<type>_list deletelist_<type>_list(
 <type>_list l,
 unsigned int from,
 unsigned int to
);
```
Delete the element at position `from` up to, but not including, `to` in list `l`. The elements are freed using `rfre_<type>()`. All elements at position `to` and up are moved down (have their index in the list decremented). If `from` is greater than or equal to `to`, or if `from` is greater than or equal to the length of `l`, nothing happens.

```
<type>_list extract_<type>_list(
 <type>_list l,
 unsigned int pos,
 <type> *e,
 int *valid
);
```
Remove the element at position `pos` from list `l`, and assign it to `*e`. All elements after position `pos` are moved down (have their index in the list decremented). If `pos` is greater than or equal to the length of `l`, nothing happens. If there was a valid element at position `pos`, it is assigned to `*e`, and `*valid` is set to `1`. Otherwise `*valid` is set to `0`, and `*e` is left unchanged.

This function is similar to `delete_<type>_list()`, but instead of deleting an element, it puts it in possession of the caller.

```
<type>_list extractlist_<type>_list(
 <type>_list l,
 unsigned int from,
 unsigned int to,
 <type>_list *e
);
```
Remove the elements in the range `from` up to, but not including `to` from list `l`, put them in a new list, and assign this list to `*e`. All elements after position `to` are moved down (have their index in the list decremented) to fill the gap. If a range extends beyond the size of `l`, the range is silently truncated.

This function is similar to `extract_<type>_list()`, but it extracts a range of elements instead of a single one.

```
<type>_list reverse_<type>_list( <type>_list l );
```
Reverse the elements in the given list.

6.5 Recursive duplication

These functions handle recursive duplication of instances of types and lists of types. Duplication means that new instances are created for all elements in the data-structures.

```
<type> rdup_<type>( const_<type> e );
<type>_list rdup_<type>_list( const_<type>_list e );
```
Recursively duplicate instance `e` of type `<type>` or `<type>_list`.

```
<type>_list slice_<type>_list(
 const_<type>_list e,
 unsigned int b,
 unsigned int e
);
```
Create a new list that contains recursive duplicates of the elements of list `l` with indices `b` up to but *not* including `e`. If `b>e` or `b` points beyond the list, an empty list is returned. If `e` points beyond `l`, it is taken to point to the last element in `l`.

6.6 Data structure input and output

These functions handle reading of a textual description of the data structures and allocation of new instances of the type and list structures for the read data. The `print_<type>` and `print_<type>_list` functions rely on the print optimiser in the C support library to handle the actual printing to a file, see section 6.11.

As an extension to the standard Tm internal representation, the `print_<type>` and `fprint_<type>` functions print null pointers as the symbol '`@`'. The `fscan_<type>` functions for the array list code recognises this symbol.

```
int fscan_<type>( FILE *f, <type> *p );
int fscan_<type>_list( FILE *f, <type>_list *p );
```
Read an instance of data structure `<type>` or `<type>_list` from file `f`, allocate new room to hold the data that is read, and set pointer `p` to point to the new data. If no error occurs, a value `0` is returned. If an error *does* occur, a value `1` is returned, and an error message is put in the array `tm_errmsg` provided by the C support library. Elements are created all the data that has been read before the error elements, and they must be freed to keep the allocation statistics in balance. The `fscan_<type>()` routines will ensure that in all cases the data-structures can be destroyed again using `rfre_<type>`.

```
void print_<type>( TMPRINTSTATE *st, const_<type> t );
void print_<type>_list( TMPRINTSTATE *st, const_<type>_list l );
```
Given a print state `st`, print an instance of data structure `<type>` or `<type>_list` using the print optimiser, see section 6.11.

```
void fprint_<type>( FILE *f, const_<type> t );
void fprint_<type>_list( FILE *f, const_<type>_list t );
```
Print an instance of data structure `<type>` or `<type>_list` to file `f`.

6.7 Comparison

These functions handle recursive comparison of structures. Given two data structures `a` and `b`, an int `n` < 0 is returned if `a` $<$ `b`, 0 if `a` $=$ `b`, and an int `n` > 0 if `a` $>$ `b`.

For comparison the following rules are applied:

- Tuples are equal if all their elements are equal, otherwise the first differing element (in the order in which they occur in the tuple definition) determines the comparison.
- Constructors are ordered according to their order of definition in the data structure file, where the first defined constructor is the smallest. Constructors are equal if all their elements are equal, otherwise the first differing element (in the order in which they occur in the tuple definition) determines the comparison.
- Lists are equal if all their elements are equal, else the first differing element determines the comparison, or else the shortest list is the smallest.

Resemblance to `strcmp()` is intentional. In fact it is possible to do:

```
#define cmp_tmstring strcmp
```

also note that it is possible to do

```
#define cmp_int (a-b)
```

However, it is not wise to do this for `float` or `double`, since rounding errors may lead to unexpected behaviour. There are a `cmp_float()` and `cmp_double()` in the support library, see section 6.11.

These functions are mainly intended for equality comparison, and to impose 'some' repeatable ordering on the data structures. The ordering may be different from the desired ordering. If another comparison function is required for a data type, the generation of the standard function can be suppressed by using `notwantdefs`, see section 6.12.

```
int cmp_<type>( const_<type> a, const_<type> b );
int cmp_<type>_list( const_<type>_list a, const_<type>_list b );
```

Compare data structures `a` and `b`, and return a code according to the table listed previously.

6.8 Equality test

These functions implement an equality test on structures. Given two data structures `a` and `b`, `TMTRUE` is returned if all the elements of the structure are equal, else `TMFALSE` is returned.

For the equality test the following rules are applied:

- Tuples are equal if and only if all their elements are equal.
- Constructors are equal if and only if all their elements are equal. Different constructor types are always differ from eachother.
- Lists are equal if and only if they are of the same length, and all their elements are equal.

These functions are mainly intended for equality comparison, but cannot be used to impose an ordering on the data structures. For this the comparison functions in the previous section can be used. If another equality test function is required for a data type, the generation of the standard function can be suppressed by using `notwantdefs`, see section 6.12.

```
tmbool isequal_<type>( const_<type> a, const_<type> b );
tmbool isequal_<type>_list( const_<type>_list a, const_<type>_list b );
```

Compare data structures `a` and `b`, and return `TMTRUE` if and only if all the elements of the structure are equal. Two constructors can only be equal if they are of the same type.

6.9 The tree walker template

In many programs it is necessary to traverse ("walk") a tree of data structures, and so that information can be collected, or checking or modification can be done. In many cases only a few types of nodes have to be visited, but nodes of many other types have to be traversed to reach these nodes. Also, it is very simple to overlook a path to a node of the given type. For that reason the `tmc` template[1] is supplemented with a template that generates code to traverse trees of Tm data structures.

Given a list of types to start from, and a list of types to visit, the template generates code to traverse the tree of Tm data structures from the given starting points, and visit all the specified types. For every instance of the target types the action function of the type is invoked.

To make the template sufficiently flexible, the user must specify the signature and invocation of the walker functions. This must be done by defining a number of macros that generate these parts of the walker functions. The user must also specify the types to start from and the types to visit, and provide action functions for each of the target types.

The use of the tree walker template is demonstrated in appendix B.

6.9.1 Global structure

A tree walker template will normally contain the components listed below. They should be placed in the order shown here:

1. Definitions for the walker macros.
2. Insertion of the walker template. This will define some additional macros.
3. Definition of the list of target types.
4. Generation of forward declarations of the walker functions.
5. Definition of the action functions.
6. Generation of the walker functions.
7. Definition of interface functions to the outside world.

6.9.2 Macros supplied by the user

These macros must be defined before the template itself is included (see below).

To illustrate the description of the macros, a simple walker is constructed that returns the number of `int` fields in a tree.

```
.macro generate_walker_signature var t
```

[1] The tree walker template that is described here is specific for the `tmc` template. However, it would be fairly simple to write a similar template for other standard templates.

Given the name `var` of the variable that holds the data structure we're walking on, and the type `t` of that variable, generate a signature for the walker function of the given type.

The signature should contain "`$t $(var)`" somewhere in the list of parameters. Apart from that, the function can have any signature.

It is recommended to declare the functions as `static`, so that unused functions can be reported. When a walker function is reported as being unused, this usually means that the action function does not invoke its walker function, as it should.

For example:

```
.macro generate_walker_signature var t
static int count_$t_walker( const $t $(var ) )
.endmacro
```

`.macro generate_walker_declaration var t`

Given the name `var` of the variable that holds the datastructure we are walking on, and the type `t` of that variable, generate a forward declaration for the walker function of the given type.

The signature should contain "`$t $(var)`" somewhere in the list of parameters. Apart from that, the function can have any signature.

This macro is usually identical to `generate_walker_signature`, except that the declaration is ended with a semicolon.

For example:

```
.macro generate_walker_declaration var t
static int count_$t_walker( const $t $(var) );
.endmacro
```

`.macro generate_descent_call indent var type nowtype`

Given an indent `indent`, a variable `var` to walk into, the type of the variable `type`, and the current type of the variable `nowtype`, generate an invocation to an action or walker.

Assuming that the Tm variable `actors` contains the list of types that have actions associated with them, it is recommended to define this macro as follows:

```
.macro generate_descent_call indent var type nowtype
.if ${member $(type) $(actors)}
.call generate_action_call "$(indent)" "$(var)" "$(type)" "$(nowtype)"
.else
.call generate_walker_call "$(indent)" "$(var)" "$(type)" "$(nowtype)"
.endif
.endmacro
```

This assumes macros `generate_action_call` and `generate_walker_call`, that have the same parameters as this macro, and generate a call to an action function and a walker function respectively. Note that since action functions are expected to invoke their walker function, they usually have the same signature as the walker function, so the two macros

will look very similar. In fact, it is recommended to invoke the walker function from an action function with a call to the macro `generate_walker_call`.

For example, assuming we have defined `generate_descent_call` as shown above, we could define `enerate_action_call` and `generate_walker_call` as follows:

```
.macro generate_action_call indent var type nowtype
.if ${eq $(type) $(nowtype)}
$(indent)n += count_$t_action( $(var) );
.else
$(indent)n += count_$t_action( to_$(type)( $(var) ) );
.endif
.endmacro
..
.macro generate_walker_call indent var type nowtype
.if ${eq $(type) $(nowtype)}
$(indent)n += count_$t_walker( $(var) );
.else
$(indent)n += count_$t_walker( to_$(type)( $(var) ) );
.endif
.endmacro
```

`.macro generate_walker_return indent var t`

Given an indent `indent`, the name `var` of the variable that holds the datastructure we are walking on, and a type `t`, of that variable, generate a return statement.

If the tree walker function is defined as a `void` function, the macro should contain the line

```
$(indent)return;
```

Otherwise a return statement should be generated that returns the appropriate value.

For example:

```
.macro generate_walker_return indent var t
$(indent)return n;
.endmacro
```

`.macro generate_walker_locals indent var t`

Given an indent `ident`, the name `var` of the variable that holds the datastructure we are walking on, and a type `t` of that variable, generate a list of local declarations for the walker function of that type.

For example:

```
.macro generate_walker_locals indent var t
$(indent)int n = 0;
.endmacro
```

`.macro generate_empty_walker_body indent var t`

Given an indent `indent`, the name `var` of the variable that holds the data structure we are walking on, and a type `t`, generate a body for an empty function.

Usually, this macro contains code to prevent compiler warnings about unused variables and parameters. For example, this macro will usually contain the following line:

```
$(indent)(void) $(var);
```

This will generate a properly indented source code line that casts the variable holding the data structure to void. Most compilers consider this a use of the variable.

For example:

```
.macro generate_empty_walker_body indent var t
$(indent)(void) $(var);
.endmacro
```

6.9.3 Invoking the tree walker template

After the macros describe in the previous section have been defined, the tree walker template can be invoked. This is done as follows:

```
.insert tmcwalk.t
```

The template will check whether the macros described above have been defined, and it will define the following macros:

`.macro calc_treewalk starts targets`

Given the list of types to start from `starts`, and the list of types to visit `targets`, return the list of types that must be visited.

Note that both parameters of this macro could potentially consist of a list of types. It is therefore recommended that the expressions for these parameters are surrounded by quotes, so that the list is not broken up.

`.macro generate_walker_forwards list`

Given a list `list` of types the tree walker should visit, generate forward declarations for the walker functions for these types.

To ensure that the list that is passed to the macro is evaluated as a single parameter, it should be surrounded with quotes.

`.macro generate_walker list`

Given a list `list` of types the tree walker should visit, generate walker functions for these types.

To ensure that the list that is passed to the macro is evaluated as a single parameter, it should be surrounded with quotes.

For example:

```
.set actors int
.set startnodes expr
..
.insert tmcwalk.t
..
.set visit_types ${call calc_treewalk "$(startnodes)" "$(actors)"}
```

6.9.4 Generating forward declarations

To generate forward declarations for all walker functions, the `generate_walker_forwards` macro should be invoked. Since action functions should invoke their corresponding walker function, these forward declarations should be placed before the definition of the action functions.

For example:

```
.call generate_walker_forwards "$(visit_types)"
```

6.9.5 Action functions

The macro `generate_descent_call` will usually generate a call to an 'action function' for every function that specified as a target type to `calc_treewalk`. These functions must be supplied by the user. Usually these functions have the same signature as the tree walker functions, and invoke the walker function somewhere in their body, so that sub-trees are also traversed. It is recommended to generate the call to the walker function with a call to the macro `generate_walker_call`.

For example:

```
static int count_int_action( int n )
{
    int n = 1;

.call generate_walker_call "    " n int int
    return n;
}
```

The walker function for `int` will be empty, but it is good practice to call it anyway, if only to prevent a complaint about unused static functions.

6.9.6 Generating of the tree walker

After the action functions have been defined, the walker functions can be generated with a call to the macro `generate_walker`.

For example:

```
.call generate_walker "$(visit_types)"
```

6.9.7 Interfacing to the outside world

Walker functions often have additional parameters that represent the state during the traversal of the tree. It is therefore often useful to wrap the top-level walker function in an interface function that creates the initial state before traversal, destroys the state after traversal, etc.

The most convenient way to invoke the top-level walker function is to use a call to the macro `generate_descent_call` with the appropriate parameters.

For example:

```
int count_ints( expr x )
{
    int n = 0;

.call generate_descent_call "    " x expr expr
    return n;
}
```

6.10 The analyser template

Although the tree walker template described in Section 6.9 is very useful, there is a particular tree walking task that is better served by providing a specialized template, namely tree analysis.

It is often necessary to traverse ("walk") a tree of datastructures, to collect information about the leaves of the tree, and to combine partial results with a reduction operation. Examples of such operations are: calculating the code size of a statement list, determining wether an expression is constant, and constructing a list of variables that are used in an expression. Analysis operations like this are easily generated by the analyser template.

In contrast to the tree walker template of Section 6.9, actions on leave nodes can usually be generated by the template, although there is an 'escape' to a user-defined function.

6.10.1 Global structure

An analyser template will normally contain the components listed below. They should be placed in the order shown here:

1. Definition of the reduction operation and neutral element.
2. Optionally, definition of a termination test.
3. Definitions for the analyser macros.
4. Insertion of the analyser template. This will define some additional macros.
5. Definition of the list of target types.

6. Definition of the reduction operations for the target types.
7. Generation of forward declarations of the analyser functions.
8. Definition of the action functions for 'escaped' nodes.
9. Generation of the analyser functions.
10. Definition of interface functions to the outside world.

6.10.2 Variables supplied by the user

The analyser template requires that the user defines a number of variables. These variables must be defined before the template itself is inserted (see below).

To illustrate the description of the macros, we construct a simple analyser that returns the sum of all `int` fields in a tree.

`.set reduction_type <type>`
The type of the reduction value we work on.

For example:

```
.set reduction_type int
```

`.set neutral_element <value>`
The value of the neutral element of the reduction operation. This must be a value of the type that is set for `reduction_type`.

For example:

```
.set neutral_element (0)
```

`.set analysis_name <name>`
The name of the analysis to perform. This name is used as prefix for the tree traversal functions that are generated.

For example:

```
.set analysis_name add
```

```
.set analyze_action_<type> reduction [<value>]
.set analyze_action_<type> function <function_name>
.set analyze_action_<type> constant <value>
.set analyze_action_<type> ignore
```
For each target type, there should be a variable of the form `analyze_action_<type>` to specify the kind of action to perform for this type. The supported kinds of action are:

`reduction` A reduction on all fields of the class, constructor type or tuple, or all elements of the list is done. If an optional value is given, this value is also incorporated in the reduction.

`function` The node is passed to a function that should be provided by the user. The name of the function to invoke must be provided as parameter. This function must have a formal parameter list as generated by the `generate_formal_parameters` macro (see below).

`constant` The node has constant value `value`. Recursion stops at this node, any fields or list elements are *not* visited.

`ignore` The node is ignored. Recursion stops at this node, any fields or list elements are *not* visited.

For example:

```
.set analyze_action_exprConst reduction
.set analyze_action_exprPlus reduction 1
.set analyze_action_exprSubtract constant 3
.set analyze_action_foldexpr ignore
.set analyze_action_int function add_int_action
```

6.10.3 Macros supplied by the user

The analyser template requires that the user defines a number of macros. These macros must be defined before the template itself is included (see below).

To illustrate the description of the macros, we continue the construction of a simple analyser that returns the sum of all `int` fields in a tree.

`.macro generate_formal_parameters nm t`
Given the name of the formal parameter `nm` and its type `t`, construct and return the formal parameter list of the analysis functions.

The signature should contain "`$t $(nm)`" somewhere in the list of parameters. Apart from that, the function can have any signature.

For example:

```
.macro generate_formal_parameters nm t
.return "( $t $(nm) )"
.endmacro
```

`.macro generate_actual_parameters nm should_type is_type`
Given the name `nm` of the variable that holds the datastructure we are walking on, the type `should_type` of the formal parameter, and the type `is_type` of the variable `nm`, construct and return the the actual parameter list of the analyser invocation.

For example:

```
.macro generate_actual_parameters nm should_type is_type
.if ${eq $(should_type) $(is_type)}
.return "( $(nm) )"
.else
.return "( ($(should_type)) $(nm) )"
.endif
.endmacro
```

`.macro generate_reduction_operation a b`

Given two expressions `a` and `b`, construct and return an expression to evaluate the reduction of these expressions.

For example:

```
.macro generate_reduction_operation a b
.return "($a)+($b)"
.endmacro
```

`.macro generate_termination_test val`

Given an expression representing the calculated value thus far, return an expression that evaluates to true if and only if there is no point in continuing the evaluation. If this macro is not defined, the analysis tree walk is never terminated prematurely.

For example, if in the running example we are not interested in the result if one of the intermediate results is negative, we could define `generate_termination_test` as follows:

```
.macro generate_termination_test val
.return "(($(val))<0)"
.endmacro
```

The termination test is only intended to speed up analysis; it is not guaranteed to be used at any point.

The termination test is particularly useful when a boolean reduction operation is done, since once an intermediate result is `true` for a boolean 'or' reduction, or `false` for a boolean 'and' reduction, we know further reduction is useless.

`.macro generate_empty_walker_body indent var t`

Given an indent `indent`, the name `var` of the variable that holds the data structure we are walking on, and a type `t`, generate a body for an empty function.

Usually, this macro contains code to prevent compiler warnings about unused variables and parameters. For example, this macro will usually contain the following line:

```
$(indent)(void) $(var);
```

This will generate a properly indented source code line that casts the variable holding the data structure to void. Most compilers consider this a use of the variable.

For example:

```
.macro generate_empty_analyzer_body indent var t
$(indent)(void) $(var);
.endmacro
```

6.10.4 Invoking the analyser template

After the macros describe in the previous section have been defined, the analyser template can be invoked. This is done as follows:

```
.insert tmcanalyze.t
```

The template will check whether the variables and macros described above have been defined, and it will define the following macros:

`.macro calc_analyzer starts targets`
: Given the list of types to start from `starts`, and the list of types to visit `targets`, return the list of types that must be visited.

  Note that both parameters of this macro could potentially consist of a list of types. Therefore expressions for these parameters should be surrounded by quotes, so that the list is not broken up.

`.macro generate_analyzer_forwards list`
: Given a list `list` of types the analyser should visit, generate forward declarations for the walker functions for these types.

  To ensure that the list that is passed to the macro is evaluated as a single parameter, it should be surrounded with quotes.

`.macro generate_analyzer list`
: Given a list `list` of types the analyser should visit, generate walker functions for these types.

  To ensure that the list that is passed to the macro is evaluated as a single parameter, it should be surrounded with quotes.

For example:

```
.set
.set startnodes expr
.set reduction_type int
.set neutral_element (0)
.set analysis_name add
..
.insert tmcanalyze.t
..
.set visit_types ${call calc_analyzer "$(startnodes)" "$(actors)"}
```

6.10.5 Generating forward declarations

To generate forward declarations for all analyser functions, the `generate_analyzer_forwards` macro should be invoked. Since action functions may invoke their corresponding generated analyser function, these forward declarations should be placed before the definition of the action functions.

For example:

```
.call generate_analyzer_forwards "$(visit_types)"
```

6.10.6 Action functions

For nodes for which a `function` action is specified, the user must specify the function that implements the analysis on this node, and provide an implementation of this function. The function must have a formal parameter list that is compatible with the one generated by the `generate_formal_parameters` macro, and must return the type that was specified in the variable `reduction_type`.

For example, for an analyser that calculates the sum of all `int` expressions in a tree, we can simply define an action

```
.set analyze_action_int function add_int_action
```

and provide the following function to implement it:

```
static int add_int_action( int n )
{
    return n;
}
```

6.10.7 Generating of the analyser

After the action functions have been defined, the analyser functions can be generated with a call to the macro `generate_analyzer`.

For example:

```
.call generate_analyzer "$(visit_types)"
```

6.10.8 Interfacing to the outside world

Analyzer functions often have additional parameters that represent the context during the traversal of the tree. It is therefore often useful to wrap the top-level analyser function in an interface function that creates the context before traversal.

The most convenient way to invoke the top-level analyser function is to use the variables and macros that already have been defined for the analyser.

For example:

```
int count_ints( expr x )
{
    int n = $(analysis_name)${call generate_formal_parameters x expr expr};
    return n;
}
```

6.11 C support library

The C support library provides definitions in the same class as the generated definitions for a number of commonly used primitive types. It also provides definitions that are necessary for various templates (e.g. the `print` functions).

For the library the following files are provided:

`libtmc.a`	The library itself.
`tmc.h`	The header file with declarations for the library functions.

6.11.1 Print optimizer functions

The print optimizer will attempt as much as possible to write a constructor, tuple or list on one line. If this is not possible it will write each item of it on a separate line.

```
TMPRINSTATE *tm_setprint(
 FILE *f,
 const int i,
 const int w,
 const int tw,
 const unsigned int flags
);
```

Initialize the print optimizer handler. The output of the `print_<type>()` functions will be written to file `f` with an indent `i` for each list, tuple or constructor. The output has a maximum of `w` characters on each line of output. If `tw` is not equal to zero, it is interpreted as the width of the tab character, and all indents will be implemented with as many tab characters as possible. Usually `tw` should be 8.

The `flags` can be used to modify the style of the output. At the moment no style options are implemented, and the parameter should be 0.

The print state descriptor returned by `tm_setprint()` stores the formatting state. It must be passed to all functions that are used to print the data structures. After all formatting and printing has been done, the print state must be destroyed with the function `tm_endprint()` described below.

Typically the function is used as follows:

```
TMPRINTSTATE *st = tm_setprint( f, 1, LINEWIDTH, TABSIZE, INDENT );
print_Block( st, blk );
int level = tm_endprint( st );
if( level>0 ){
    fprintf( stderr, "Internal botch: bracket level is %d\n", level );
}
```

```
int tm_endprint( TMPRINTSTATE *st );
```

Destroy the given print state, and return the listing level of the print state. The listing level is the number of calls to `tm_openlist()`, `tm_opencons()`, and `tm_opentuple()` that have not been balanced with the corresponding `tm_closelist()`, `tm_closecons()`, resp. `tm_closetuple()`. Normally the listing level returned by `tm_endprint()` should be zero.

```
void tm_openlist( TMPRINTSTATE *st );
void tm_closelist( TMPRINSTATE *st );
```
Start or stop a list.

```
void tm_opencons( TMPRINTSTATE *st );
void tm_closecons( TMPRINTSTATE *st );
```
Start or stop a constructor. Since classes are represented in the same way as constructors, these functions are also used for classes.

```
void tm_opentuple( TMPRINTSTATE *st );
void tm_closetuple( TMPRINTSTATE *st );
```
Start or stop a tuple.

```
void tm_printword( TMPRINTSTATE *st, const char *w );
```
Add a word `w` to the current list, tuple, or constructor.

6.11.2 File scan support functions

These functions are provided to support the `fscan_<type>` template functions, and similar functions for primitive types.

```
int tm_fscanspace( FILE *f );
int tm_lineno;
```
Skip all spaces, tabs, form feeds and comment in the input up to the next non-white character. Increment `tm_lineno` for each newline character that is encountered. A comment is started with `||` and is terminated by a newline. It is not allowed to have a single `|` followed by another character than another `|` in the input: this will put an error message in `tm_errmsg` and a return value `1`. In all other cases `0` is returned.

```
int tm_fneedc( FILE *f, int c );
```
After skipping spaces, tabs, form feeds and comment, try to read character `c` from file `f`. Return a value `1` and put an error message in `tm_errmsg` if this is not possible, else return `0`.

```
int tm_fscanopenbrac( FILE *f );
int tm_fscanclosebrac( FILE *f, int n );
```
Handle brackets around constructors. `tm_fscanopenbrac()` counts and returns the number of open brackets (‘`(`’) it encounters up to the first character that is not white space, comment or an open bracket. `tm_fscanclosebrac()` tries to read `n` close brackets (‘`)`’). It will return `1` and put an error message in `tm_errmsg` if this is not possible. It will return `0` if it is successful.

```
int tm_fscancons( FILE *f, char *buf, int sz );
```
Read a constructor name. `tm_fscancons()` tries to read a constructor name consisting of up to $sz - 1$ upper- or lower-case characters or digits. The constructor name will be copied into `buf`, and terminated by a '`\0`'. It will return `1` and put an error message in `tm_errmsg` if the constructor name has length 0 or is longer than $sz - 1$, else it will return `0`.

```
const char *tm_escapestring( const unsigned int code );
```
Given an unsigned int `code` representing the ASCII code of a character, return a pointer to a string representing this character in an escape sequence (or an ordinary character, if that is possible). The returned pointer points to a static buffer that is overwritten upon the next invocation of the function.

```
int tm_fscanescapedchar( FILE *f, int *code );
```
Try to read a (possibly escaped) character from the file `f`. It will return `1` and put an error message in `tm_errmsg` if no escaped character can be read, else it will return `0`.

6.11.3 Template functions for primitive types

```
void fprint_tmbool( FILE *f, const tmbool b );
void fre_tmbool( tmbool b );
int fscan_tmbool( FILE *f, tmbool *b );
void print_tmbool( TMPRINSTATE *st, const tmbool b );
void rfre_tmbool( tmbool b );
#define tmboolNIL <something>
#define TMTRUESTR "True"
#define TMFALSESTR "False"
typedef tmbool <something>
```
Module functions for type 'tmbool'. A `tmbool` is the boolean representation used by Tm. It may have the values `TMTRUE` and `TMFALSE`. This name has been chosen instead of the more obvious 'bool', since this type name is often defined at another place. For example, the `curses` software and MS-Windows both define `bool`. In `fscan_tmbool()`, `fprint_tmbool()` and `print_tmbool()` use `TMTRUESTR` and `TMFALSESTR` as representation strings.

```
typedef signed char tmschar;
int cmp_tmschar( const tmschar a, const tmschar b );
tmbool isequal_tmschar( const tmschar a, const tmschar b );
void fprint_tmschar( FILE *f, const tmschar c );
void fre_tmschar( tmschar c );
int fscan_tmschar( FILE *f, tmschar *cp );
#define tmscharNIL <something>
void print_tmschar( TMPRINSTATE *st, const tmschar c );
tmschar rdup_tmschar( const tmschar c );
void rfre_tmschar( tmschar c );
```
Module functions for type '`signed char`'.

```
typedef unsigned char tmuchar;
int cmp_tmuchar( const tmuchar a, const tmuchar b );
tmbool isequal_tmuchar( const tmuchar a, const tmuchar b );
void fprint_tmuchar( FILE *f, const tmuchar c );
void fre_tmuchar( tmuchar c );
int fscan_tmuchar( FILE *f, tmuchar *cp );
#define tmucharNIL <something>
void print_tmuchar( TMPRINSTATE *st, const tmuchar c );
tmuchar rdup_tmuchar( const tmuchar c );
void rfre_tmuchar( tmuchar c );
```

Module functions for type ‘`unsigned char`’.

```
int cmp_int( const int a, const int b );
tmbool isequal_int( const int a, const int b );
void fprint_int( FILE *f, const int i );
void fre_int( int i );
int fscan_int( FILE *f, int *i );
#define intNIL <something>
void print_int( TMPRINSTATE *st, const int i );
int rdup_int( int i );
void rfre_int( int i );
```

Module functions for type ‘`int`’.

```
typedef unsigned int tmuint;
int cmp_tmuint( const tmuint a, const tmuint b );
tmbool isequal_tmuint( const tmuint a, const tmuint b );
void fprint_tmuint( FILE *f, const tmuint i );
void fre_tmuint( tmuint u );
int fscan_tmuint( FILE *f, tmuint *p );
void print_tmuint( TMPRINSTATE *st, const tmuint i );
tmuint rdup_tmuint( const tmuint u );
void rfre_tmuint( tmuint u );
#define tmuintNIL <something>
```

Module functions for type ‘`unsigned int`’.

```
int cmp_long( const long a, const long b );
tmbool isequal_long( const long a, const long b );
void fprint_long( FILE *f, const long i );
void fre_long( long u );
int fscan_long( FILE *f, long *p );
void print_long( TMPRINSTATE *st, const long i );
long rdup_long( const long u );
void rfre_long( long u );
#define longNIL <something>
```

Module functions for type ‘`long int`’.

```
typedef unsigned long int tmulong;
int cmp_tmulong( const tmulong a, const tmulong b );
tmbool isequal_tmulong( const tmulong a, const tmulong b );
```

```
void fprint_tmulong( FILE *f, const tmulong i );
void fre_tmulong( tmulong u );
int fscan_tmulong( FILE *f, tmulong *p );
void print_tmulong( TMPRINSTATE *st, const tmulong i );
tmulong rdup_tmulong( const tmulong u );
void rfre_tmulong( tmulong u );
#define tmulongNIL <something>
```

Module functions for type ‘`unsigned long int`’.

```
typedef short int tmshort;
int cmp_tmshort( const tmshort a, const tmshort b );
tmbool isequal_tmshort( const tmshort a, const tmshort b );
void fprint_tmshort( FILE *f, const tmshort i );
void fre_tmshort( tmshort u );
int fscan_tmshort( FILE *f, tmshort *p );
void print_tmshort( TMPRINSTATE *st, const tmshort i );
tmshort rdup_tmshort( const tmshort u );
void rfre_tmshort( tmshort u );
#define tmshortNIL <something>
```

Module functions for type ‘`short int`’.

```
typedef unsigned short int tmushort;
int cmp_tmushort( const tmushort a, const tmushort b );
tmbool isequal_tmushort( const tmushort a, const tmushort b );
void fprint_tmushort( FILE *f, const tmushort i );
void fre_tmushort( tmushort u );
int fscan_tmushort( FILE *f, tmushort *p );
void print_tmushort( TMPRINSTATE *st, const tmushort i );
tmushort rdup_tmushort( const tmushort u );
void rfre_tmushort( tmushort u );
#define tmushortNIL <something>
```

Module functions for type ‘`unsigned short int`’.

```
int cmp_double( const double a, const double b );
tmbool isequal_double( const double a, const double b );
#define doubleNIL <something>
void fprint_double( FILE *f, const double d );
void fre_double( double d );
int fscan_double( FILE *f, double *d );
void print_double( TMPRINTSTATE *st, double d );
double rdup_double( const double d );
void rfre_double( double d );
```

Module functions for type ‘`double`’.

```
int cmp_float( float a, float b );
tmbool isequal_float( float a, float b );
#define floatNIL <something>
void fprint_float( FILE *f, float d );
void fre_float( float d );
```

```
int fscan_float( FILE *f, float *d );
void print_float( TMPRINSTATE *st, const float d );
float rdup_float( const float d );
void rfre_float( float d );
```
Module functions for type 'float'.

```
void fprint_tmsymbol( FILE *f, const tmsymbol s );
void fre_tmsymbol( tmsymbol s );
int fscan_tmsymbol( FILE *f,  tmsymbol *s );
void print_tmsymbol( TMPRINTSTATE *st, const tmsymbol s );
tmsymbol rdup_tmsymbol( const tmsymbol s );
void rfre_tmsymbol( tmsymbol s );
#define tmsymbolNIL <something>
```
Module functions for type 'tmsymbol'. See section 6.11.8 for details about type `tmsymbol`.

```
typedef char *tmstring;
typedef const char *const_tmstring;

void fprint_tmstring( FILE *f, const tmstring s );
void fre_tmstring( tmstring s );
int fscan_tmstring( FILE *f,  tmstring *s );
tmstring new_tmstring( const char *s );
void print_tmstring( TMPRINTSTATE *st, const tmstring s );
tmstring rdup_tmstring( const tmstring s );
void rfre_tmstring( tmstring s );
void stat_tmstring( FILE *f );
int get_balance_tmstring( void );
#define tmstringNIL <something>
```
Module functions for type 'tmstring'. A 'tmstring' is an array of characters terminated by a '\0' character. Note that `fscan_tmstring()`, `rdup_tmstring()`, and `new_tmstring()` allocate memory for the scanned string, you must use `fre_tmstring()` to return that memory. `Stat_tmstring` prints to file `f` a line specifying the number of allocated and freed strings. There is also a function `realloc_tmstring()` to enlarge the string array, see section 6.11.6.

```
tmstring create_tmstring( size_t sz );
```
Given a desired string size `sz`, return a new, empty, `tmstring` containing room for `sz` characters.

```
typedef char *tmword;
typedef const char *const_tmword;

void fprint_tmword( FILE *f, const_tmword s );
void fre_tmword( tmword s );
int fscan_tmword( FILE *f,  tmword *s );
tmword new_tmword( const char *s );
void print_tmword( TMPRINTSTATE *st, const_tmword s );
tmword rdup_tmword( const tmword s );
void rfre_tmword( tmword s );
#define tmwordNIL <something>
```

Module functions for type 'tmword'. A 'tmword' is the same as a 'tmstring', but the scanning and printing routines are different. `fscan_tmword()` reads a single word consisting of one or more characters. All characters are accepted, except whitespace, and the characters '`(),@[]`'. Like other `fscan_<type>` functions, `fscan_tmword()` allows any number of parenthesis pairs around the word, and skips whitespace before the word. Since a `tmword` is represented as a `tmstring`, you must use `fre_tmstring`, or the alias `fre_tmword()`, to deallocate that memory.

```
typedef struct <something> *tmtext;
typedef const struct <something> *const_tmtext;

void fprint_tmtext( FILE *f, const tmtext s );
int cmp_tmtext( const tmtext ta, const tmtext tb );
tmbool isequal_tmtext( const tmtext ta, const tmtext tb );
void fre_tmtext( tmtext s );
int fscan_tmtext( FILE *f,  tmtext *s );
tmtext new_tmtext( const char *s );
void print_tmtext( TMPRINTSTATE *st, const tmtext s );
tmtext rdup_tmtext( const tmtext s );
void rfre_tmtext( tmtext s );
void stat_tmtext( FILE *f );
int get_balance_tmtext( void );
#define tmtextNIL <something>
```

Module functions for type 'tmtext'. A 'tmtext' is a list of characters, similar to the lists that can be generated for arbitrary types. See section 6.11.7 for more details.

6.11.4 Error functions

```
char tm_errmsg[TM_ERRLEN];
```

A buffer for error messages from Tm functions, in particular the `fscan_<type>()` functions. The value `TM_ERRLEN` is defined in the include file `tmc.h`.

```
void tm_fatal( char *file, int line, char *s );
```

Fatal error handler for Tm functions: given the source file name `file`, the source line `line` and the error message `s`, print the error message `s`. If `line` is not equal to 0, and `file` is not equal to `""` (the empty string) the origin (source file and line) of the error is printed. After this, stop the program.

All error messages, except those about bad tags and dynamic memory problems, are passed through `tm_fatal()`. The user can supply another error handler by replacing this function.

```
void tm_badtag( char *file, int line, int tag );
```

Fatal error handler for bad tags: print the position given by source file name `file` and source line `line`, and print the tag value as a decimal and hexadecimal value. After this stop the program. This message is *not* printed through `tm_fatal()`, since the occurrence of a bad tag is a symptom of a serious organization problem.

```
void tm_noroom( void );
```
Handler for allocation errors of `tm_malloc()`, `tm_calloc()` and `tm_realloc()`. By default it prints 'no room' to `stderr` and does an `exit(1)`, but it may be replaced by another function to incorporate panic shutdown code (to save, for example, the file being edited). If this function returns (the default one does not), the Tm allocation routines will try again to allocate the requested block. This way, garbage collection may be implemented.

6.11.5 `LOGNEW` functions

For a detailed explanation of the purpose of `LOGNEW` functions, see section 6.3.

```
void tm_lognew(
 const tm_neutralp p,
 const char *file,
 const int line
);
```
Given a pointer `p`, record it as a new entry in the lognew table. The current values of `nwl_f` and `nwl_l` are stored to record the source-code position of the block request.

```
void tm_logfre( const tm_neutralp p );
```
Given a pointer `p`, delete its entry from the lognew table.

```
long int tm_new_logid( const char *file, const int line );
```
Request a new number to identify a Tm allocation block.

When called, `file` and `line` are stored to record the source-code position of the block request. The Tm code templates store the identification number in the allocated block. When the block is freed, the entry is removed from the list using the identification number, with a call to `tm_fre_logid()`. This method is more efficient than that of `tm_lognew()` and `tm_logfre()`, but requires that the identification number is stored in the allocated block. It is therefore unsuitable to log `tmstring` allocations.

```
void tm_fre_logid( const long int i );
```
Remove the block with id `i` from the list of pending blocks. See the description of `tm_new_logid()`.

```
void report_lognew( FILE *f );
```
Print the pending entries in the lognew table to file `f`.

```
void flush_lognew();
```
Clear the table of lognew entries, and free any memory that has been allocated for it.

6.11.6 Dynamic memory allocation

Type handling for `malloc()` and friends is a *minefield*. Different compilers have different ideas about neutral pointers, whether the result of a malloc should be cast, etc. The functions and defines in the Tm C library try to meet the demands for the various compilers by introducing a number of compiler-dependent definitions.

The allocation functions invoke a function `tm_noroom()` to print the actual error message. A default function is provided that prints the message to `stderr` and does an `exit(1)`. To implement emergency handling or even recovery, the user can provide a different `tm_noroom()` function.

Dynamic memory allocation is very sensitive to errors. Therefore, Tm provides extensive support to detect and repair such bugs, see section 6.3.

```
typedef <something> tm_neutralp;
```
The pointer type that is the 'neutral' pointer type for this compiler.

```
tm_neutralp tm_malloc( size_t sz );
tm_neutralp tm_calloc( size_t n, size_t sz );
tm_neutralp tm_realloc( tm_neutralp p, size_t sz );
```
The standard `malloc()` routine and friends, but with important differences:

1. They return a pointer of type `tm_neutralp`, which may not be the same type as that of the original routine.
2. They handle out of memory situations by invoking `tm_noroom()`. If `tm_noroom()` returns, (the default one does not), they try again to allocate the requested memory.
3. They are guaranteed to accept `sz = 0` and `n = 0`.

You cannot use these functions directly in portable code, since some compilers will require casts of the pointer parameters and return values, while other compilers require that you do *not*. Therefore, a set of compiler dependent macros is provided to solve this problem.

```
#define TM_MALLOC(t,sz) <something>
#define TM_CALLOC(t,n,sz) <something>
#define TM_REALLOC(t,p,sz) <something>
#define TM_FREE(p) <something>
```
System-independent versions of the `tm_malloc()` functions and friends. They have the same parameters as the functions, but have an additional parameter `t` that indicates the type the returned pointer should have.

```
tmstring realloc_tmstring( tmstring s, const size_t sz );
```
Given a tmstring `s` and a string size `sz`, invoke `tm_realloc()` to enlarge `s` to have at least `sz` bytes. You should not use `tm_realloc()` on strings directly, since this confuses the `LOGNEW` functions.

6.11.7 Text handling

```
typedef tmtextptr <something>;

typedef struct str_tmtext {
    tmtextptr arr;
    long curpos;        /* Current read or write pointer. */
    long sz;
    long room;
    long int lognew_id;
} *tmtext;
```

A `tmtext` is a list of `tmuchar`, similar to other lists, with some additional functionality. The type `tmtextptr` is a pointer to `tmuchar`. The main reason this deserves a typedef is that this way it can be declared `huge`, as is required for MS-DOS and some versions of MS-Windows. Next to the standard support functions (see 6.11.3), a number of additional functions are provided.

`extern void stat_tmtext( FILE *f );`
This function is equivalent to the standard `stat_<basename>()` function: given a file handle `f`, print the allocation and freeing statistics to that file.

`extern int get_balance_tmtext( void );`
This function is equivalent to the standard `get_balance_<basename>()` function: return the current balance state of `tmtext`. The function returns -1 if more instances have been freed than have been allocated, 1 if more instances have been allocated than freed, and 0 if the same number of instances have been allocated and freed.

`extern tmtext setroom_tmtext( tmtext t, long rm );`
This function is similar to the standard `setroom_<type>()` function. Specify that text `t` must have room for at least `rm` characters. This function ensures that the array has sufficient room to store `rm` elements. The room in a list is never reduced.

Functions that add elements to a `tmtext` use this function implicitly, the user only needs these functions for efficiency reasons (to prevent repeated enlargement of the array) or to build new list functions.

`extern tmtext slice_tmtext( const tmtext t, long from, long to );`
This function is similar to the standard `slice_<type>()` function. Given a text `t`, a start position `from` and an end position `to`, create a new text containing the character from `t` starting from `from` up to (but not including) `to`.

If the specified range extends beyond the real text, the range is limited to the real text.

`extern tmtext delblock_tmtext( tmtext t, long from, long to );`
Given a text `t`, a starting position `from` and an end position `to`, delete the characters from `from` up to, but not including `to`. Characters above this range are moved down to close the gap. If all or part of the specified range to delete falls beyond the text, that part of the range is ignored.

```
extern tmtext replace_tmtext(
 tmtext t,
 const long from,
 const long to,
 const tmtext nw
);
```
Given a text `t`, a starting position `from`, an ending position `to`, and a text `nw`, replace the characters in that range with the characters in `nw`. The characters beyond the range, from `to` upwards, are moved up or down to close the gap. Return the new text.

```
extern tmtext insert_tmtext( tmtext t, const long pos, const tmtext nw );
```
Given a text `t`, a position `pos`, and a text `nw`, insert `nw` in `t` at position `pos`. If `pos` is negative, `nw` is inserted at the start of the text, if `pos` is greater than the size of `t`, `nw` is appended at the end of the text.

The new text is returned.

```
extern int cmp_string_tmtext( const char *s, const tmtext t );
```
Given a string `s` and a text `t`, return -1 if `t` is smaller than `s`, return 1 if `t` is greater than `s`, and return 0 if they are equal.

```
extern tmstring tmtext_to_tmstring( const tmtext t );
```
Given a text `t`, return a `tmstring` that contains these characters. It is assumed that `t` does not contain `'\0'` characters.

```
extern tmtext string_to_tmtext( const char *s );
```
Given a string `s`, return a new text consisting of this string.

```
extern void puts_tmtext( const char *s, tmtext t );
extern void putc_tmtext( tmuchar c, tmtext t );
```
Given a text `t` and a character `c` or a string `s`, write this string or character to the text at the position `t->curpos`. If necessary the text is enlarged. `t->curpos` is incremented to point past the written characters.

6.11.8 Symbol handling

These functions maintain a table of strings. Each time a string is added to the table, a value of type `symbol` is returned. It is guaranteed that for equal strings the same value is returned, and it is possible to compare these values with a simple compare (==). There is a special 'nil' value, `symbolNIL`.

To store the information for a symbol, the following data structure is used:

```
/* Storage for a symbol string */
struct _tmc_sym {
   struct _tmc_sym *next;       /* next in list */
   tmstring name;               /* pointer to the string */
   tm_neutralp data;            /* any info for it. */
```

```
};

typedef struct _tmc_sym *tmsymbol;
typedef const struct _tmc_sym *const_tmsymbol;
```

The tmstring of this symbol is stored in field `name`. The name of a tmsymbol `s` can be printed with

```
printf( "%s", s->name );
```

The field `data` is never touched by the symbol handling routines, it may be used to store user data associated with the symbol.

Next to the standard template functions, a number of other functions are defined:

`tmsymbol add_tmsymbol( const char *name );`
: Add string `name` to the symbol table. If the string already occurs in the table, the value of the old entry is returned, else a new entry is created, and a new value is returned.

  This routine ensures that for all symbols with the same name, the same value is returned.

`tmsymbol find_tmsymbol( char *name );`
: Try to locate string `name` in the symbol table. If it occurs in the table, its value is returned, else `symbolNIL` is returned.

`void flush_tmsymbol();`
: Empty the symbol table, and free all memory allocated to it. It is allowed to start using `add_tmsymbol()` and `gen_tmsymbol()` again, but new symbol values will be returned.

`tmsymbol gen_tmsymbol( const char *prefix );`
: Given a string `prefix`, generate a new symbol value with a new string. It is not allowed to use `add_tmsymbol()` after you have used `gen_tmsymbol()`. Otherwise it cannot be guaranteed that the generated symbols remain unique. This restriction is enforced by the routines.

6.12 Tm and C configuration variables

The templates use a few Tm variables to modify the contents of the generated code. Unless stated otherwise, it is not necessary to set them; a reasonable default will be chosen.

`basename`	This variable is used to generate the names of definitions that are generated only once (e.g. initialization and statistics functions). This variable *must* be set.
`wantdefs`	If set, this variable contains the names of the type and function definitions that must be generated. If these functions require the definition of other functions, the necessary functions will be generated, but will be declared `static`. If `alldefs` is set all possible definitions are generated.
`notwantdefs`	If set, this variable contains the names of the type and function definitions for which under *all* conditions *no* code must be generated.
`alldefs`	If set, this variable indicates that all possible type and function definitions, and all functions for first level lists, must be generated for all defined types.

At least one of the variables `wantdefs` and `alldefs` must be set.

In the generated C code a number of preprocessor variables are used:

`FATAL(msg)`	`#define` this if you want to supply a fatal error handler to print `msg`. By default `tm_fatal()` is used, see section 6.11.
`FIRSTROOM`	The initial room of an array when created with `new_<type>_list()`. Default it is 0.
`LOGNEW`	`#define` this if you want code to record the origin of all `new_<type>()` requests. If this code is enabled, you can print with `report_lognew()` the source file and line of all pending allocated blocks. See section 6.3 for a detailed description. *Important:* `LOGNEW` only works if all code is compiled with `LOGNEW`.

Appendix A

Syntax of Tm data-structure descriptions

The data structure description must have the following syntax:

dsList:
    ϵ
    ds dsList

ds:
    name `:=` *inherits constructorList* `;`
    name `==` *inherits* `(` *tuple* `)` `;`
    name `=` *classcomponentList* `;`
    name `~=` *classcomponentList* `;`
    name `->` *type* `;`
    `%include` *string* `;`

inherits:
    ϵ
    name `+` *inherits*

tuple:
    field
    field `,` *tuple*

constructorList:
    constructor
    constructor `|` *constructorList*

constructor:
    name fieldList

fieldList:
    ϵ
    field fieldList

classcomponentList:
    ϵ
    classcomponent
    classcomponent + *classcomponentList*

classcomponent:
    { *tuple* }
    name
    subclassList
    (*classcomponentList*)

subclassList:
    subclass
    subclass | *subclassList*

subclass:
    name : *classcomponent*

field:
    name : *type*

type:
    name
    [*type*]

Characters in typewriter font `like this` are tokens, ϵ stands for an empty token list, *name* stands for a token with the regular expression `[a-zA-Z][a-zA-Z0-9]*`, and *string* stands for a string of arbitrary characters surrounded by double quotes `"`.

If a constructor base type is defined repeatedly, the definitions are merged. Redefinition of all other types is not allowed.

Appendix B

An example project: expression optimisation

In this chapter we will present three small but complete programs. One parses a simple language, and prints it out in the standard Tm representation. A second program reads in this representation, replaces constant expressions with a single constant, and writes it out again. A third one also reads this representation, replaces references to known variables with their value, and writes it out again. The third program not only uses external functions generated by a standard Tm template, but is for a significant part generated from a tree walker template.

The parser reads a series of assignments, such as:

```
a = 1 + 2 * 3;
b = 1 - 2;
c = - 1 + 2;
d = -(1+2);
e = c + 3 + 4;
c = c * 3;
f = c + (3 + 4);
z = x + z + c;
```

and writes out the parse tree of the input file in Tm format. The other programs reads the parse tree in Tm format, transform it, and write out the transformed parse tree.

The files shown in this appendix are available as a separate package, called `tmdemo`. See the Tm FTP site.

B.1 The parser

The assignment commands are represented with the following data structures:

```
|| Representation of a stream of variable assignments.
command = { lhs:tmstring, rhs:expr };

expr ~=
    ExprPlus: { a:expr, b:expr } |
    ExprTimes: { a:expr, b:expr } |
```

```
    ExprNegate: { x:expr } |
    ExprConst: { n:int } |
    ExprSymbol: { s:tmstring }
    ;
```

To read in the commands, lexical analysis and parsing must be done. Lexical analysis is done with a lexical analyser generated by `lex`. It is specified as follows:

```
%{
#include <stdlib.h>
#include <tmc.h>

#include "calc.h"
#include "tokens.h"

#ifdef LINUX
#define yywrap() (1)
#endif

%}
%option noyywrap
%option input
%option nounput

white_space     [ \t\f\n]
letter          [A-Za-z_]
identifier      {letter}({letter}|[0-9])*
number          [0-9]+

%%

{white_space}+  {}
{number}        {
                    yylval._literal = atoi( yytext );
                    return LITERAL;
                }
{identifier}    {
                    yylval._identifier = new_tmstring( yytext );
                    return IDENTIFIER;
                }
.               { return *yytext; }

%%
```

Parsing is done in a `yacc` file

```
%{
#include <tmc.h>

#include "calc.h"

static command_list result;

static void yyerror( const char *s )
{
    fprintf( stderr, "%s\n", s );
    exit( 1 );
}

%}

%union {
    expr                        _expr;
    command                     _command;
```

```
    command_list                _commandList;
    tmstring                    _identifier;
    int                         _literal;
}

%start program

%token <_identifier>            IDENTIFIER
%token <_literal>               LITERAL

%type   <_expr>                 expr
%type   <_command>              command
%type   <_commandList>  commandList

%left '+' '-'
%left '*'

%%

program:
    commandList
        { result = $1; }
;

commandList:
    /* empty */
        { $$ = new_command_list(); }
|
    commandList command
        { $$ = append_command_list( $1, $2 ); }
;

command:
    IDENTIFIER '=' expr ';'
        { $$ = new_command( $1, $3 ); }
;

expr:
    expr '+' expr
        { $$ = (expr) new_ExprPlus( $1, $3 ); }
|
    expr '-' expr
        { $$ = (expr) new_ExprPlus( $1, (expr) new_ExprNegate( $3 ) ); }
|
    expr '*' expr
        { $$ = (expr) new_ExprTimes( $1, $3 ); }
|
    '-' expr
        { $$ = (expr) new_ExprNegate( $2 ); }
|
    LITERAL
        { $$ = (expr) new_ExprConst( $1 ); }
|
    IDENTIFIER
        { $$ = (expr) new_ExprSymbol( $1 ); }
|
    '(' expr ')'
        { $$ = $2; }
;

%%
```

```
int main( void )
{
    TMPRINTSTATE *st;
    int level;

    if( yyparse() ){
        exit( 1 );
    }
    st = tm_setprint( stdout, 1, 75, 8, 0 );
    print_command_list( st, result );
    level = tm_endprint( st );
    if( level != 0 ){
        fprintf( stderr, "Internal error: bracket level = %d\n", level );
        exit( 1 );
    }
    rfre_command_list( result );
    if( get_balance_calc() != 0 || get_balance_tmstring() != 0 ){
        fprintf( stderr, "Tm object allocation not balanced\n" );
        stat_calc( stderr );
        stat_tmstring( stderr );
        report_lognew( stderr );
    }
    exit( 0 );
    return 0;
}
```

This file also contains the `main()` function, which invokes the parser and prints the result to a Tm textual representation using the function `print_command_list()`. The actions of the `yacc` grammar also contain functions generated by Tm to construct intermediate parts of the parse tree, for example the `new_command()` function.

Finally, the file contains the functions `rfre_command_list()`, `get_balance_calc()` and `stat_calc()`. The first recursively deallocates all elements in the constructed command list, and the latter two are used to report any imbalance in the number of allocations and deallocations. They are not strictly necessary, but in practice have proven to be invaluable for reliable dynamic memory management.

Given an input file:

```
a = 1 + 2 * 3;
b = 1 - 2;
c = - 1 + 2;
d = -(1+2);
e = c + 3 + 4;
c = c * 3;
f = c + (3 + 4);
z = x + z + c;
```

the parser will produce the following output:

```
[
 (
  command
  "a"
  (ExprPlus (ExprConst 1) (ExprTimes (ExprConst 2) (ExprConst 3)))
 ),
 (command "b" (ExprPlus (ExprConst 1) (ExprNegate (ExprConst 2)))),
 (command "c" (ExprPlus (ExprNegate (ExprConst 1)) (ExprConst 2))),
 (command "d" (ExprNegate (ExprPlus (ExprConst 1) (ExprConst 2)))),
 (
  command
```

```
  "e"
  (ExprPlus (ExprPlus (ExprSymbol "c") (ExprConst 3)) (ExprConst 4))
 ),
 (command "c" (ExprTimes (ExprSymbol "c") (ExprConst 3))),
 (
  command
  "f"
  (ExprPlus (ExprSymbol "c") (ExprPlus (ExprConst 3) (ExprConst 4)))
 ),
 (
  command
  "z"
  (ExprPlus (ExprPlus (ExprSymbol "x") (ExprSymbol "z")) (ExprSymbol "c"))
 )
]
```

B.2 The optimiser

The optimiser is a very simple program that reads a command list in Tm format, evaluates all constant expressions, and writes back the simplified command list in Tm format.

```
#include <stdio.h>
#include <tmc.h>

#include "calc.h"

/* Small utility function to get the value of an ExprConst. */
static int get_const( expr x )
{
    if( x->tag != TAGExprConst ){
        fprintf( stderr, "Internal error: not a const\n" );
        exit( 1 );
    }
    return to_ExprConst(x)->n;
}

static expr optimize_expr( expr x )
{
    switch( x->tag ){
        case TAGExprPlus:
        {
            ExprPlus px = to_ExprPlus( x );

            px->a = optimize_expr( px->a );
            px->b = optimize_expr( px->b );
            if( px->a->tag == TAGExprConst && px->b->tag == TAGExprConst ){
                int sum = get_const( px->a )+get_const( px->b );

                rfre_expr( x );
                x = (expr) new_ExprConst( sum );
            }
            break;
        }

        case TAGExprTimes:
        {
            ExprTimes px = to_ExprTimes( x );
```

```
            px->a = optimize_expr( px->a );
            px->b = optimize_expr( px->b );
            if( px->a->tag == TAGExprConst && px->b->tag == TAGExprConst ){
                int prod = get_const( px->a )*get_const( px->b );

                rfre_expr( x );
                x = (expr) new_ExprConst( prod );
            }
            break;
        }

        case TAGExprNegate:
        {
            ExprNegate px = to_ExprNegate( x );

            px->x = optimize_expr( px->x );
            if( px->x->tag == TAGExprConst ){
                int nx = -get_const( px->x );

                rfre_expr( x );
                x = (expr) new_ExprConst( nx );
            }
            break;
        }

        case TAGExprConst:
        case TAGExprSymbol:
            break;
    }
    return x;
}

static command optimize_command( command s )
{
    s->rhs = optimize_expr( s->rhs );
    return s;
}

static command_list optimize_command_list( command_list sl )
{
    unsigned int ix;

    for( ix=0; ix<sl->sz; ix++ ){
        sl->arr[ix] = optimize_command( sl->arr[ix] );
    }
    return sl;
}

int main( void )
{
    TMPRINTSTATE *st;
    int level;
    command_list sl;

    tm_lineno = 1;
    if( fscan_command_list( stdin, &sl ) ){
        fprintf( stderr, "calcopt: %d: %s\n", tm_lineno, tm_errmsg );
        exit( 1 );
    }
    sl = optimize_command_list( sl );
    st = tm_setprint( stdout, 1, 75, 8, 0 );
    print_command_list( st, sl );
```

```
    level = tm_endprint( st );
    if( level != 0 ){
        fprintf( stderr, "Internal error: bracket level = %d\n", level );
        exit( 1 );
    }
    rfre_command_list( sl );
    if( get_balance_calc() != 0 || get_balance_tmstring() != 0 ){
        fprintf( stderr, "Tm object allocation not balanced\n" );
        stat_calc( stderr );
        stat_tmstring( stderr );
        report_lognew( stderr );
    }
    exit( 0 );
    return 0;
}
```

Reading and writing is done with the functions `fscan_command_list()` and `print_command_list()` respectively. Inbetween, a simple recursive tree walker visits all nodes of all expression trees, and replaces all nodes containing constant expressions with the result of evaluation.

Given the output from the parser shown above, the optimiser will produce the following output:

```
[
 (command "a" (ExprConst 7)),
 (command "b" (ExprConst (-1))),
 (command "c" (ExprConst 1)),
 (command "d" (ExprConst (-3))),
 (
  command
  "e"
  (ExprPlus (ExprPlus (ExprSymbol "c") (ExprConst 3)) (ExprConst 4))
 ),
 (command "c" (ExprTimes (ExprSymbol "c") (ExprConst 3))),
 (command "f" (ExprPlus (ExprSymbol "c") (ExprConst 7))),
 (
  command
  "z"
  (ExprPlus (ExprPlus (ExprSymbol "x") (ExprSymbol "z")) (ExprSymbol "c"))
 )
]
```

Note that the expression $c + 3 + 4$ is not simplified, because the parser interprets this as $(c + 3) + 4$, which does not contain a constant expression. A real-life optimiser will probably have to do better than this.

B.3 The substitution engine

The substitution engine is a small program that reads a command list in Tm format, and substitutes all references to known variables with their value. It is implemented using the tree walker template, see Section 6.9.

```
.. File: rewrite.ct
..
.. Tree walker for rewriting operations.
..
.. Return 'walk' or 'action', depending on the contents of 'actors'
.macro walkername t
```

```
.if ${member $t $(actors)}
.return action
.else
.return walker
.endif
.endmacro
..
.macro generate_empty_walker_body indent var t
$(indent)(void) lets;
.endmacro
..
.macro generate_walker_return indent var t
$(indent)return $(var);
.endmacro
..
.macro generate_walker_signature var t
static $t subst_$t_walker( $t $(var), command_list *lets )
.endmacro
..
.macro generate_walker_declaration var t
static $t subst_$t_walker( $t $(var), command_list *lets );
.endmacro
..
.macro generate_action_call indent var type nowtype
.if ${eq $(type) $(nowtype)}
$(indent)$(var) = ($(nowtype)) subst_$(type)_action( $(var), lets );
.else
$(indent)$(var) = ($(nowtype)) subst_$(type)_action( to_$(type)( $(var) ), lets );
.endif
.endmacro
..
.macro generate_walker_call indent var type nowtype
.if ${eq $(type) $(nowtype)}
$(indent)$(var) = ($(nowtype)) subst_$(type)_walker( $(var), lets );
.else
$(indent)$(var) = ($(nowtype)) subst_$(type)_walker( to_$(type)( $(var) ), lets );
.endif
.endmacro
..
.macro generate_descent_call indent var type nowtype
.if ${member $(type) $(actors)}
.call generate_action_call "$(indent)" "$(var)" "$(type)" "$(nowtype)"
.else
.call generate_walker_call "$(indent)" "$(var)" "$(type)" "$(nowtype)"
.endif
.endmacro
..
.. For which types are there actions defines?
.set actors command ExprSymbol
..
.insert tmcwalk.t
..
.set visit_types ${call calc_treewalk "command_list" "$(actors)"}
..
/* File: subst.c
 *
 * Substitute known variables in expressions.
 */

#include <assert.h>
#include <tmc.h>
```

```
#include "calc.h"

.call generate_walker_forwards "$(visit_types)"

static tmbool find_command( tmconststring nm, command_list l, unsigned int *pos )
{
    unsigned int ix;

    for( ix=0; ix<l->sz; ix++ ){
        if( strcmp( l->arr[ix]->lhs, nm ) == 0 ){
            *pos = ix;
            return TMTRUE;
        }
    }
    return TMFALSE;
}

static expr subst_ExprSymbol_action( ExprSymbol s, command_list *lets )
{
    unsigned int pos;
    expr res;

.call generate_walker_call "    " s ExprSymbol ExprSymbol
    if( find_command( s->s, *lets, &pos ) ){
        rfre_expr( (expr) s );
        res = rdup_expr( (*lets)->arr[pos]->rhs );
    }
    else {
        res = (expr) s;
    }
    return res;
}

/* First apply substitutions in the rhs of the command, then
 * add or replace the current assignment to the state.
 */
static command subst_command_action( command c, command_list *lets )
{
    unsigned int pos;

.call generate_walker_call "    " c command command
    if( find_command( c->lhs, *lets, &pos ) ){
        *lets = delete_command_list( *lets, pos );
    }
    *lets = append_command_list( *lets, rdup_command( c ) );
    return c;
}

.call generate_walker "$(visit_types)"

int main( void )
{
    TMPRINTSTATE *st;
    int level;
    command_list sl;
    command_list context;
    command_list *lets = &context;

    tm_lineno = 1;
    if( fscan_command_list( stdin, &sl ) ){
        fprintf( stderr, "calcopt: %d: %s\n", tm_lineno, tm_errmsg );
        exit( 1 );
```

```
    }
    context = new_command_list();
.call generate_descent_call "    " sl command_list command_list
    rfre_command_list( context );
    st = tm_setprint( stdout, 1, 75, 8, 0 );
    print_command_list( st, sl );
    level = tm_endprint( st );
    if( level != 0 ){
        fprintf( stderr, "Internal error: bracket level = %d\n", level );
        exit( 1 );
    }
    rfre_command_list( sl );
    if( get_balance_calc() != 0 || get_balance_tmstring() != 0 ){
        fprintf( stderr, "Tm object allocation not balanced\n" );
        stat_calc( stderr );
        stat_tmstring( stderr );
        report_lognew( stderr );
    }
    exit( 0 );
    return 0;
}
```

The input and output is done in the same way as in the optimiser described in the previous section. The substitution is implemented as a tree walker generated by the `tmcwalk.t` template. It has two actions: function `subst_ExprSymbol_action` searches the list of known definitions for the symbol of the expression, and replaces it with the value of that symbol. Function `subst_command_action` updates the list of known symbols.

If we feed the output from the parser shown above to the substitution engine, and apply the optimiser engine above to the output, the following output is produced:

```
[
 (command "a" (ExprConst 7)),
 (command "b" (ExprConst (-1))),
 (command "c" (ExprConst 1)),
 (command "d" (ExprConst (-3))),
 (command "e" (ExprConst 8)),
 (command "c" (ExprConst 3)),
 (command "f" (ExprConst 10)),
 (
  command
  "z"
  (ExprPlus (ExprPlus (ExprSymbol "x") (ExprSymbol "z")) (ExprConst 3))
 )
]
```

For people that are curious about the generated tree walker code, the result of

```
tm calc.ds subst.ct
```

is the following file:

```
/* File: subst.c
 *
 * Substitute known variables in expressions.
 */

#include <assert.h>
#include <tmc.h>
```

```
#include "calc.h"

/* ----------- Generated forward declarations start here ----------- */

/* Forward declarations. */
static command_list subst_command_list_walker( command_list e, command_list *lets );
static ExprPlus subst_ExprPlus_walker( ExprPlus e, command_list *lets );
static ExprTimes subst_ExprTimes_walker( ExprTimes e, command_list *lets );
static ExprNegate subst_ExprNegate_walker( ExprNegate e, command_list *lets );
static ExprSymbol subst_ExprSymbol_walker( ExprSymbol e, command_list *lets );
static expr subst_expr_walker( expr e, command_list *lets );
static command subst_command_walker( command e, command_list *lets );

/* ----------- Generated forward declarations end here ----------- */

static tmbool find_command( tmconststring nm, command_list l, unsigned int *pos )
{
    unsigned int ix;

    for( ix=0; ix<l->sz; ix++ ){
        if( strcmp( l->arr[ix]->lhs, nm ) == 0 ){
            *pos = ix;
            return TMTRUE;
        }
    }
    return TMFALSE;
}

static expr subst_ExprSymbol_action( ExprSymbol s, command_list *lets )
{
    unsigned int pos;
    expr res;

    s = (ExprSymbol) subst_ExprSymbol_walker( s, lets );
    if( find_command( s->s, *lets, &pos ) ){
        rfre_expr( (expr) s );
        res = rdup_expr( (*lets)->arr[pos]->rhs );
    }
    else {
        res = (expr) s;
    }
    return res;
}

/* First apply substitutions in the rhs of the command, then
 * add or replace the current assignment to the state.
 */
static command subst_command_action( command c, command_list *lets )
{
    unsigned int pos;

    c = (command) subst_command_walker( c, lets );
    if( find_command( c->lhs, *lets, &pos ) ){
        *lets = delete_command_list( *lets, pos );
    }
    *lets = append_command_list( *lets, rdup_command( c ) );
    return c;
}

/* ----------- Generated code starts here ----------- */

/* Given a list command_list, rewrite it. */
```

```
static command_list subst_command_list_walker( command_list e, command_list *lets )
{
    unsigned int ix;

    for( ix=0; ix<e->sz; ix++ ){
        e->arr[ix] = (command) subst_command_action( e->arr[ix], lets );
    }
    return e;
}

/* Given a class ExprPlus, rewrite it. */
static ExprPlus subst_ExprPlus_walker( ExprPlus e, command_list *lets )
{
    e->a = (expr) subst_expr_walker( e->a, lets );
    e->b = (expr) subst_expr_walker( e->b, lets );
    return e;
}

/* Given a class ExprTimes, rewrite it. */
static ExprTimes subst_ExprTimes_walker( ExprTimes e, command_list *lets )
{
    e->a = (expr) subst_expr_walker( e->a, lets );
    e->b = (expr) subst_expr_walker( e->b, lets );
    return e;
}

/* Given a class ExprNegate, rewrite it. */
static ExprNegate subst_ExprNegate_walker( ExprNegate e, command_list *lets )
{
    e->x = (expr) subst_expr_walker( e->x, lets );
    return e;
}

/* Given a class ExprSymbol, rewrite it. */
static ExprSymbol subst_ExprSymbol_walker( ExprSymbol e, command_list *lets )
{
    (void) lets;
    return e;
}

/* Given a class expr, rewrite it. */
static expr subst_expr_walker( expr e, command_list *lets )
{
    switch( e->tag ){
        case TAGExprPlus:
            e = (expr) subst_ExprPlus_walker( to_ExprPlus( e ), lets );
            break;

        case TAGExprTimes:
            e = (expr) subst_ExprTimes_walker( to_ExprTimes( e ), lets );
            break;

        case TAGExprNegate:
            e = (expr) subst_ExprNegate_walker( to_ExprNegate( e ), lets );
            break;

        case TAGExprSymbol:
            e = (expr) subst_ExprSymbol_action( to_ExprSymbol( e ), lets );
            break;

        default:
            break;
```

```
    }
    return e;
}

/* Given a class command, rewrite it. */
static command subst_command_walker( command e, command_list *lets )
{
    e->rhs = (expr) subst_expr_walker( e->rhs, lets );
    return e;
}


/* ----------- Generated code ends here ----------- */

int main( void )
{
    TMPRINTSTATE *st;
    int level;
    command_list sl;
    command_list context;
    command_list *lets = &context;

    tm_lineno = 1;
    if( fscan_command_list( stdin, &sl ) ){
        fprintf( stderr, "calcopt: %d: %s\n", tm_lineno, tm_errmsg );
        exit( 1 );
    }
    context = new_command_list();
    sl = (command_list) subst_command_list_walker( sl, lets );
    rfre_command_list( context );
    st = tm_setprint( stdout, 1, 75, 8, 0 );
    print_command_list( st, sl );
    level = tm_endprint( st );
    if( level != 0 ){
        fprintf( stderr, "Internal error: bracket level = %d\n", level );
        exit( 1 );
    }
    rfre_command_list( sl );
    if( get_balance_calc() != 0 || get_balance_tmstring() != 0 ){
        fprintf( stderr, "Tm object allocation not balanced\n" );
        stat_calc( stderr );
        stat_tmstring( stderr );
        report_lognew( stderr );
    }
    exit( 0 );
    return 0;
}
```

B.4 Generated functions

To use Tm to generate code, it is necessary to provide a number of files. The data structure definitions were already shown above. Apart from that, the functions that must be generated are listed in a separate file:

```
.. File: calcconf.t
.set basename calc
.set wantdefs
.append wantdefs new_command_list
```

```
.append wantdefs new_command
.append wantdefs rdup_command
.append wantdefs rfre_expr
.append wantdefs rdup_expr
.append wantdefs append_command_list
.append wantdefs delete_command_list
.append wantdefs print_command_list
.append wantdefs fscan_command_list
.append wantdefs rfre_command_list
.append wantdefs ${prefix new_ ${subclasses expr}}
.append wantdefs stat_$(basename)
.append wantdefs get_balance_$(basename)
```

Since the same source file is used for both the parser and the optimiser, this file lists all functions that are required by either. Notice the trick to generate all `new_<t>` functions for subclasses of `expr`.

Next, two small files must be provided that include the configuration file shown above, and invoke a standard template. These files will be expanded into a `.c` and a `.h` file. The template for the `.c` file must also contain a `#include` for the header of the standard Tm C library (called `tmc.h`), and the generated `.h` file. This is shown in the template files listed here:

For the example the two template files are:

```
..File: calc.ht

.insert calcconf.t
.include tmc.ht
```

and

```
..File: calc.ct
#include <tmc.h>
#include "calc.h"

.insert calcconf.t
.include tmc.ct
```

In general, the header files may also contain other stuff, such as functions for primitive types.

Finally, you can add rules to your makefile to generate the code:

```
calc.c: calc.ds calc.ct calcconf.t
        tm calc.ds calc.ct > calc.c

calc.h: calc.ds calc.ht calcconf.t
        tm calc.ds calc.ht > calc.h
```

This will result in a header file `calc.h`:

```
/* Requirement analysis took 100 milliseconds. */
/*** WARNING: THIS IS GENERATED CODE. ***/

/* ---- start of /usr/local/lib/tmc.ht ---- */
/* External definitions (Version for array list).

   template file:      /usr/local/lib/tmc.ht
   datastructure file: calc.ds
   tm version:         36
   tm kernel version:  2.0-beta19
 */
```

```
/* data structures */

/* forward reference typedefs for all types.
 * C does not like the use of undefined types, but does not
 * mind the use of pointers to (yet) undefined types.
 */
typedef struct str_command_list *command_list;
typedef struct str_expr *expr;
typedef struct str_ExprConst *ExprConst;
typedef struct str_ExprNegate *ExprNegate;
typedef struct str_ExprPlus *ExprPlus;
typedef struct str_ExprSymbol *ExprSymbol;
typedef struct str_ExprTimes *ExprTimes;
typedef struct str_command *command;

#define command_listNIL (command_list)0
#define ExprConstNIL (ExprConst)0
#define ExprNegateNIL (ExprNegate)0
#define ExprPlusNIL (ExprPlus)0
#define ExprSymbolNIL (ExprSymbol)0
#define ExprTimesNIL (ExprTimes)0
#define commandNIL (command)0
#define exprNIL (expr)0

typedef enum en_tags_command {
    TAGcommand
} tags_command;

typedef enum en_tags_expr {
    TAGExprPlus, TAGExprTimes, TAGExprNegate, TAGExprConst, TAGExprSymbol
} tags_expr;


#ifdef __cplusplus
/* Structure for class 'command'. */
class str_command {
public:
#ifdef LOGNEW
    long int lognew_id;
#endif
    tags_command tag;
    tmstring lhs;
    expr rhs;
};

/* Structure for class 'expr'. */
class str_expr {
public:
#ifdef LOGNEW
    long int lognew_id;
#endif
    tags_expr tag;
};

/* Structure for class 'ExprConst'. */
class str_ExprConst: public str_expr {
public:
    int n;
};

/* Structure for class 'ExprNegate'. */
```

```
class str_ExprNegate: public str_expr {
public:
    expr x;
};

/* Structure for class 'ExprPlus'. */
class str_ExprPlus: public str_expr {
public:
    expr a;
    expr b;
};

/* Structure for class 'ExprSymbol'. */
class str_ExprSymbol: public str_expr {
public:
    tmstring s;
};

/* Structure for class 'ExprTimes'. */
class str_ExprTimes: public str_expr {
public:
    expr a;
    expr b;
};

#else
/* Structure for class 'expr'. */
struct str_expr {
#ifdef LOGNEW
    long int lognew_id;
#endif
    tags_expr tag;
};

/* Structure for class 'ExprConst'. */
struct str_ExprConst {
#ifdef LOGNEW
    long int lognew_id;
#endif
    tags_expr tag;
    int n;
};

/* Structure for class 'ExprNegate'. */
struct str_ExprNegate {
#ifdef LOGNEW
    long int lognew_id;
#endif
    tags_expr tag;
    expr x;
};

/* Structure for class 'ExprPlus'. */
struct str_ExprPlus {
#ifdef LOGNEW
    long int lognew_id;
#endif
    tags_expr tag;
    expr a;
    expr b;
};
```

```
/* Structure for class 'ExprSymbol'. */
struct str_ExprSymbol {
#ifdef LOGNEW
    long int lognew_id;
#endif
    tags_expr tag;
    tmstring s;
};

/* Structure for class 'ExprTimes'. */
struct str_ExprTimes {
#ifdef LOGNEW
    long int lognew_id;
#endif
    tags_expr tag;
    expr a;
    expr b;
};

/* Structure for class 'command'. */
struct str_command {
#ifdef LOGNEW
    long int lognew_id;
#endif
    tags_command tag;
    tmstring lhs;
    expr rhs;
};

#endif

struct str_command_list {
    unsigned int sz;
    unsigned int room;
    command *arr;
#ifdef LOGNEW
    long int lognew_id;
#endif
};

/* Type casting macros. */
#define to_ExprConst(e) ((ExprConst)e)
#define to_ExprNegate(e) ((ExprNegate)e)
#define to_ExprPlus(e) ((ExprPlus)e)
#define to_ExprSymbol(e) ((ExprSymbol)e)
#define to_ExprTimes(e) ((ExprTimes)e)
#define to_command(e) ((command)e)
#define to_expr(e) ((expr)e)

/* new_<type> routines */
#ifdef LOGNEW
#define new_ExprConst(n) real_new_ExprConst(n,__FILE__,__LINE__)
#define new_ExprNegate(x) real_new_ExprNegate(x,__FILE__,__LINE__)
#define new_ExprPlus(a,b) real_new_ExprPlus(a,b,__FILE__,__LINE__)
#define new_ExprSymbol(s) real_new_ExprSymbol(s,__FILE__,__LINE__)
#define new_ExprTimes(a,b) real_new_ExprTimes(a,b,__FILE__,__LINE__)
#define new_command(lhs,rhs) real_new_command(lhs,rhs,__FILE__,__LINE__)
#define new_command_list() real_new_command_list(__FILE__,__LINE__)
#define rdup_command(e) real_rdup_command(e,__FILE__,__LINE__)
#define rdup_expr(e) real_rdup_expr(e,__FILE__,__LINE__)
#define fscan_command_list(f,l) real_fscan_command_list(f,l,__FILE__,__LINE__)
#endif
```

```
#ifdef LOGNEW
extern ExprConst real_new_ExprConst( int, const char *, const int );
extern ExprNegate real_new_ExprNegate( expr, const char *, const int );
extern ExprPlus real_new_ExprPlus( expr, expr, const char *, const int );
extern ExprSymbol real_new_ExprSymbol( tmstring, const char *, const int );
extern ExprTimes real_new_ExprTimes( expr, expr, const char *, const int );
extern command real_new_command( tmstring, expr, const char *, const int );
extern command_list real_new_command_list( const char *file, const int line );
#else
extern ExprConst new_ExprConst( int );
extern ExprNegate new_ExprNegate( expr );
extern ExprPlus new_ExprPlus( expr, expr );
extern ExprSymbol new_ExprSymbol( tmstring );
extern ExprTimes new_ExprTimes( expr, expr );
extern command new_command( tmstring, expr );
extern command_list new_command_list( void );
#endif
extern command_list append_command_list( command_list, command );
extern command_list delete_command_list( command_list, const unsigned int );
extern void rfre_command_list( command_list );
extern void rfre_expr( expr );
extern void print_command_list( TMPRINTSTATE *, const command_list );
#ifdef LOGNEW
extern command real_rdup_command( const command, const char *_f, const int _l );
extern expr real_rdup_expr( const expr, const char *_f, const int _l );
#else
extern command rdup_command( const command );
extern expr rdup_expr( const expr );
#endif
#ifdef LOGNEW
extern int real_fscan_command_list( FILE *, command_list *, const char *, const int );
#else
extern int fscan_command_list( FILE *, command_list * );
#endif
extern void stat_calc( FILE * );
extern int get_balance_calc( void );
/* ---- end of /usr/local/lib/tmc.ht ---- */
/* Code generation required 130 milliseconds. */
```

and a C source file `calc.c`:

```
#include <tmc.h>
#include "calc.h"

/* Requirement analysis took 110 milliseconds. */
/*** WARNING: THIS IS GENERATED CODE. ***/

/* ---- start of /usr/local/lib/tmc.ct ---- */

/* Routines for 'calc'.

   template file:      /usr/local/lib/tmc.ct
   datastructure file: calc.ds
   tm version:         36
   tm kernel version:  2.0-beta19
 */

#ifndef FIRSTROOM
#define FIRSTROOM 0
#endif

/* Counters for allocation and freeing. */
```

```
static long newcnt_command_list = 0, frecnt_command_list = 0;
static long newcnt_ExprConst = 0, frecnt_ExprConst = 0;
static long newcnt_ExprNegate = 0, frecnt_ExprNegate = 0;
static long newcnt_ExprPlus = 0, frecnt_ExprPlus = 0;
static long newcnt_ExprSymbol = 0, frecnt_ExprSymbol = 0;
static long newcnt_ExprTimes = 0, frecnt_ExprTimes = 0;
static long newcnt_command = 0, frecnt_command = 0;


static char tm_srcfile[] = __FILE__;

#ifndef FATAL
#define FATAL(msg) tm_fatal(tm_srcfile,__LINE__,msg)
#endif

/* Error strings. */
static char tm_nilptr[] = "NIL pointer";
static char tm_badcons[] = "bad constructor for '%s': '%s'";
static char tm_badeof[] = "unexpected end of file";

#ifndef FATALTAG
#define FATALTAG(tag) tm_badtag(tm_srcfile,__LINE__,(int) tag)
#endif

/*************************************************
 *    set array room routines                    *
 *************************************************/

/* Announce that you will need room for 'rm' elements in
 *  command_list 'l'.
 */
static command_list setroom_command_list( command_list l, const unsigned int rm )
{
    if( l->room>=rm ){
        return l;
    }
    if( l->room==0 ){
        l->arr = TM_MALLOC( command *, rm * sizeof(*(l->arr)) );
    }
    else {
        l->arr = TM_REALLOC( command *, l->arr, rm * sizeof(*(l->arr)) );
    }
    l->room = rm;
    return l;
}

/*************************************************
 *    Allocation routines                        *
 *************************************************/

#ifdef LOGNEW
#undef new_ExprConst
#define new_ExprConst(n) real_new_ExprConst(n,_f,_l)
#undef new_ExprNegate
#define new_ExprNegate(x) real_new_ExprNegate(x,_f,_l)
#undef new_ExprPlus
#define new_ExprPlus(a,b) real_new_ExprPlus(a,b,_f,_l)
#undef new_ExprSymbol
#define new_ExprSymbol(s) real_new_ExprSymbol(s,_f,_l)
#undef new_ExprTimes
#define new_ExprTimes(a,b) real_new_ExprTimes(a,b,_f,_l)
#undef new_command
```

```
#define new_command(lhs,rhs) real_new_command(lhs,rhs,_f,_l)
#undef new_command_list
#define new_command_list() real_new_command_list(_f,_l)
#endif

#ifdef LOGNEW
command_list real_new_command_list( const char *_f, const int _l )
#else
command_list new_command_list( void )
#endif
{
    command_list nw;

    nw = TM_MALLOC( command_list, sizeof(*nw) );
    nw->sz = 0;
#if FIRSTROOM==0
    nw->arr = (command *) 0;
    nw->room = 0;
#else
    nw->arr = TM_MALLOC( command *, FIRSTROOM*sizeof(command) );
    nw->room = FIRSTROOM;
#endif
    newcnt_command_list++;
#ifdef LOGNEW
    nw->lognew_id = tm_new_logid( _f, _l );
#endif
    return nw;
}

/* Allocate a new instance of class 'ExprConst'. */
#ifdef LOGNEW
ExprConst real_new_ExprConst( int p_n, const char *_f, const int _l )
#else
ExprConst new_ExprConst( int p_n )
#endif
{
    ExprConst nw;

    nw = TM_MALLOC( ExprConst, sizeof(*nw) );
    nw->tag = TAGExprConst;
    nw->n = p_n;
    newcnt_ExprConst++;
#ifdef LOGNEW
    nw->lognew_id = tm_new_logid( _f, _l );
#endif
    return nw;
}

/* Allocate a new instance of class 'ExprNegate'. */
#ifdef LOGNEW
ExprNegate real_new_ExprNegate( expr p_x, const char *_f, const int _l )
#else
ExprNegate new_ExprNegate( expr p_x )
#endif
{
    ExprNegate nw;

    nw = TM_MALLOC( ExprNegate, sizeof(*nw) );
    nw->tag = TAGExprNegate;
    nw->x = p_x;
    newcnt_ExprNegate++;
#ifdef LOGNEW
```

```
    nw->lognew_id = tm_new_logid( _f, _l );
#endif
    return nw;
}

/* Allocate a new instance of class 'ExprPlus'. */
#ifdef LOGNEW
ExprPlus real_new_ExprPlus( expr p_a, expr p_b, const char *_f, const int _l )
#else
ExprPlus new_ExprPlus( expr p_a, expr p_b )
#endif
{
    ExprPlus nw;

    nw = TM_MALLOC( ExprPlus, sizeof(*nw) );
    nw->tag = TAGExprPlus;
    nw->a = p_a;
    nw->b = p_b;
    newcnt_ExprPlus++;
#ifdef LOGNEW
    nw->lognew_id = tm_new_logid( _f, _l );
#endif
    return nw;
}

/* Allocate a new instance of class 'ExprSymbol'. */
#ifdef LOGNEW
ExprSymbol real_new_ExprSymbol( tmstring p_s, const char *_f, const int _l )
#else
ExprSymbol new_ExprSymbol( tmstring p_s )
#endif
{
    ExprSymbol nw;

    nw = TM_MALLOC( ExprSymbol, sizeof(*nw) );
    nw->tag = TAGExprSymbol;
    nw->s = p_s;
    newcnt_ExprSymbol++;
#ifdef LOGNEW
    nw->lognew_id = tm_new_logid( _f, _l );
#endif
    return nw;
}

/* Allocate a new instance of class 'ExprTimes'. */
#ifdef LOGNEW
ExprTimes real_new_ExprTimes( expr p_a, expr p_b, const char *_f, const int _l )
#else
ExprTimes new_ExprTimes( expr p_a, expr p_b )
#endif
{
    ExprTimes nw;

    nw = TM_MALLOC( ExprTimes, sizeof(*nw) );
    nw->tag = TAGExprTimes;
    nw->a = p_a;
    nw->b = p_b;
    newcnt_ExprTimes++;
#ifdef LOGNEW
    nw->lognew_id = tm_new_logid( _f, _l );
#endif
    return nw;
```

```
}

/* Allocate a new instance of class 'command'. */
#ifdef LOGNEW
command real_new_command( tmstring p_lhs, expr p_rhs, const char *_f, const int _l )
#else
command new_command( tmstring p_lhs, expr p_rhs )
#endif
{
    command nw;

    nw = TM_MALLOC( command, sizeof(*nw) );
    nw->tag = TAGcommand;
    nw->lhs = p_lhs;
    nw->rhs = p_rhs;
    newcnt_command++;
#ifdef LOGNEW
    nw->lognew_id = tm_new_logid( _f, _l );
#endif
    return nw;
}

/***********************************************
 *    Freeing routines                          *
 ***********************************************/

static void fre_command_list( command_list );
static void fre_command( command );
static void fre_ExprConst( ExprConst );
static void fre_ExprNegate( ExprNegate );
static void fre_ExprPlus( ExprPlus );
static void fre_ExprSymbol( ExprSymbol );
static void fre_ExprTimes( ExprTimes );
/* Free an element 'e' of class type 'command'. */
static void fre_command( command e )
{
    if( e == commandNIL ){
        return;
    }
    switch( e->tag ){
        case TAGcommand:
            frecnt_command++;
#ifdef LOGNEW
            tm_fre_logid( e->lognew_id );
#endif
            TM_FREE( e );
            break;

        default:
            FATALTAG( e->tag );
    }
}

/* Free an element 'e' of class type 'ExprConst'. */
static void fre_ExprConst( ExprConst e )
{
    if( e == ExprConstNIL ){
        return;
    }
    switch( e->tag ){
        case TAGExprConst:
            frecnt_ExprConst++;
```

```
#ifdef LOGNEW
            tm_fre_logid( e->lognew_id );
#endif
            TM_FREE( e );
            break;

        default:
            FATALTAG( e->tag );
    }
}

/* Free an element 'e' of class type 'ExprNegate'. */
static void fre_ExprNegate( ExprNegate e )
{
    if( e == ExprNegateNIL ){
        return;
    }
    switch( e->tag ){
        case TAGExprNegate:
            frecnt_ExprNegate++;
#ifdef LOGNEW
            tm_fre_logid( e->lognew_id );
#endif
            TM_FREE( e );
            break;

        default:
            FATALTAG( e->tag );
    }
}

/* Free an element 'e' of class type 'ExprPlus'. */
static void fre_ExprPlus( ExprPlus e )
{
    if( e == ExprPlusNIL ){
        return;
    }
    switch( e->tag ){
        case TAGExprPlus:
            frecnt_ExprPlus++;
#ifdef LOGNEW
            tm_fre_logid( e->lognew_id );
#endif
            TM_FREE( e );
            break;

        default:
            FATALTAG( e->tag );
    }
}

/* Free an element 'e' of class type 'ExprSymbol'. */
static void fre_ExprSymbol( ExprSymbol e )
{
    if( e == ExprSymbolNIL ){
        return;
    }
    switch( e->tag ){
        case TAGExprSymbol:
            frecnt_ExprSymbol++;
#ifdef LOGNEW
            tm_fre_logid( e->lognew_id );
```

```
#endif
            TM_FREE( e );
            break;

        default:
            FATALTAG( e->tag );
    }
}

/* Free an element 'e' of class type 'ExprTimes'. */
static void fre_ExprTimes( ExprTimes e )
{
    if( e == ExprTimesNIL ){
        return;
    }
    switch( e->tag ){
        case TAGExprTimes:
            frecnt_ExprTimes++;
#ifdef LOGNEW
            tm_fre_logid( e->lognew_id );
#endif
            TM_FREE( e );
            break;

        default:
            FATALTAG( e->tag );
    }
}

/* Free a list of command elements 'l'. */
static void fre_command_list( command_list l )
{
    if( l == command_listNIL ){
        return;
    }
#ifdef LOGNEW
    tm_fre_logid( l->lognew_id );
#endif
    frecnt_command_list++;
    if( l->room!=0 ){
        TM_FREE( l->arr );
    }
    TM_FREE( l );
}

/***********************************************
 *    Append routines                          *
 ***********************************************/

/* Append a command element 'e' to list 'l', and return the new list. */
command_list append_command_list( command_list l, command e )
{
    if( l->sz >= l->room ){
        l = setroom_command_list( l, 1+(l->sz)+(l->sz) );
    }
    l->arr[l->sz] = e;
    l->sz++;
    return l;
}

/***********************************************
 *    Recursive freeing routines               *
```

```
 ***********************************************/

static void rfre_command( command );
static void rfre_ExprConst( ExprConst );
static void rfre_ExprNegate( ExprNegate );
static void rfre_ExprPlus( ExprPlus );
static void rfre_ExprSymbol( ExprSymbol );
static void rfre_ExprTimes( ExprTimes );
/* Recursively free an element 'e' of class type 'expr'
 * and all elements in it.
 */
void rfre_expr( expr e )
{
    if( e == exprNIL ){
        return;
    }
    switch( e->tag ){
        case TAGExprPlus:
            rfre_ExprPlus( (ExprPlus) e );
            break;

        case TAGExprTimes:
            rfre_ExprTimes( (ExprTimes) e );
            break;

        case TAGExprNegate:
            rfre_ExprNegate( (ExprNegate) e );
            break;

        case TAGExprConst:
            rfre_ExprConst( (ExprConst) e );
            break;

        case TAGExprSymbol:
            rfre_ExprSymbol( (ExprSymbol) e );
            break;

        default:
            FATALTAG( (int) e->tag );
    }
}

/* Recursively free an element 'e' of class type 'command'
 * and all elements in it.
 */
static void rfre_command( command e )
{
    if( e == commandNIL ){
        return;
    }
    switch( e->tag ){
        case TAGcommand:
            rfre_tmstring( e->lhs );
            rfre_expr( e->rhs );
            fre_command( e );
            break;

        default:
            FATALTAG( (int) e->tag );
    }
}
```

```
/* Recursively free an element 'e' of class type 'ExprConst'
 * and all elements in it.
 */
static void rfre_ExprConst( ExprConst e )
{
    if( e == ExprConstNIL ){
        return;
    }
    switch( e->tag ){
        case TAGExprConst:
            rfre_int( e->n );
            fre_ExprConst( e );
            break;

        default:
            FATALTAG( (int) e->tag );
    }
}

/* Recursively free an element 'e' of class type 'ExprNegate'
 * and all elements in it.
 */
static void rfre_ExprNegate( ExprNegate e )
{
    if( e == ExprNegateNIL ){
        return;
    }
    switch( e->tag ){
        case TAGExprNegate:
            rfre_expr( e->x );
            fre_ExprNegate( e );
            break;

        default:
            FATALTAG( (int) e->tag );
    }
}

/* Recursively free an element 'e' of class type 'ExprPlus'
 * and all elements in it.
 */
static void rfre_ExprPlus( ExprPlus e )
{
    if( e == ExprPlusNIL ){
        return;
    }
    switch( e->tag ){
        case TAGExprPlus:
            rfre_expr( e->a );
            rfre_expr( e->b );
            fre_ExprPlus( e );
            break;

        default:
            FATALTAG( (int) e->tag );
    }
}

/* Recursively free an element 'e' of class type 'ExprSymbol'
 * and all elements in it.
 */
static void rfre_ExprSymbol( ExprSymbol e )
```

```
{
    if( e == ExprSymbolNIL ){
        return;
    }
    switch( e->tag ){
        case TAGExprSymbol:
            rfre_tmstring( e->s );
            fre_ExprSymbol( e );
            break;

        default:
            FATALTAG( (int) e->tag );
    }
}

/* Recursively free an element 'e' of class type 'ExprTimes'
 * and all elements in it.
 */
static void rfre_ExprTimes( ExprTimes e )
{
    if( e == ExprTimesNIL ){
        return;
    }
    switch( e->tag ){
        case TAGExprTimes:
            rfre_expr( e->a );
            rfre_expr( e->b );
            fre_ExprTimes( e );
            break;

        default:
            FATALTAG( (int) e->tag );
    }
}

/* Recursively free a list of elements 'e' of type command. */
void rfre_command_list( command_list e )
{
    unsigned int ix;

    if( e == command_listNIL ){
        return;
    }
    for( ix=0; ix<e->sz; ix++ ){
        rfre_command( e->arr[ix] );
    }
    fre_command_list( e );
}

/***********************************************
 *    print_<type> and print_<type>_list routines *
 ***********************************************/

static void print_command( TMPRINTSTATE *st, const command );
static void print_expr( TMPRINTSTATE *st, const expr );
static void print_ExprConst( TMPRINTSTATE *st, const ExprConst );
static void print_ExprNegate( TMPRINTSTATE *st, const ExprNegate );
static void print_ExprPlus( TMPRINTSTATE *st, const ExprPlus );
static void print_ExprSymbol( TMPRINTSTATE *st, const ExprSymbol );
static void print_ExprTimes( TMPRINTSTATE *st, const ExprTimes );
/* Print an element 't' of class type 'command'
 * using print optimizer.
```

```
 */
static void print_command( TMPRINTSTATE *st, const command t )
{
    if( t == commandNIL ){
        tm_printword( st, "@" );
        return;
    }
    switch( t->tag ){
        case TAGcommand:
            tm_opencons( st );
            tm_printword( st, "command" );
            print_tmstring( st, t->lhs );
            print_expr( st, t->rhs );
            tm_closecons( st );
            break;

        default:
            FATALTAG( t->tag );
    }
}

/* Print an element 't' of class type 'expr'
 * using print optimizer.
 */
static void print_expr( TMPRINTSTATE *st, const expr t )
{
    if( t == exprNIL ){
        tm_printword( st, "@" );
        return;
    }
    switch( t->tag ){
        case TAGExprPlus:
            print_ExprPlus( st, (ExprPlus) t );
            break;

        case TAGExprTimes:
            print_ExprTimes( st, (ExprTimes) t );
            break;

        case TAGExprNegate:
            print_ExprNegate( st, (ExprNegate) t );
            break;

        case TAGExprConst:
            print_ExprConst( st, (ExprConst) t );
            break;

        case TAGExprSymbol:
            print_ExprSymbol( st, (ExprSymbol) t );
            break;

        default:
            FATALTAG( t->tag );
    }
}

/* Print an element 't' of class type 'ExprConst'
 * using print optimizer.
 */
static void print_ExprConst( TMPRINTSTATE *st, const ExprConst t )
{
    if( t == ExprConstNIL ){
```

```
        tm_printword( st, "@" );
        return;
    }
    switch( t->tag ){
        case TAGExprConst:
            tm_opencons( st );
            tm_printword( st, "ExprConst" );
            print_int( st, t->n );
            tm_closecons( st );
            break;

        default:
            FATALTAG( t->tag );
    }
}

/* Print an element 't' of class type 'ExprNegate'
 * using print optimizer.
 */
static void print_ExprNegate( TMPRINTSTATE *st, const ExprNegate t )
{
    if( t == ExprNegateNIL ){
        tm_printword( st, "@" );
        return;
    }
    switch( t->tag ){
        case TAGExprNegate:
            tm_opencons( st );
            tm_printword( st, "ExprNegate" );
            print_expr( st, t->x );
            tm_closecons( st );
            break;

        default:
            FATALTAG( t->tag );
    }
}

/* Print an element 't' of class type 'ExprPlus'
 * using print optimizer.
 */
static void print_ExprPlus( TMPRINTSTATE *st, const ExprPlus t )
{
    if( t == ExprPlusNIL ){
        tm_printword( st, "@" );
        return;
    }
    switch( t->tag ){
        case TAGExprPlus:
            tm_opencons( st );
            tm_printword( st, "ExprPlus" );
            print_expr( st, t->a );
            print_expr( st, t->b );
            tm_closecons( st );
            break;

        default:
            FATALTAG( t->tag );
    }
}

/* Print an element 't' of class type 'ExprSymbol'
```

```
 * using print optimizer.
 */
static void print_ExprSymbol( TMPRINTSTATE *st, const ExprSymbol t )
{
    if( t == ExprSymbolNIL ){
        tm_printword( st, "@" );
        return;
    }
    switch( t->tag ){
        case TAGExprSymbol:
            tm_opencons( st );
            tm_printword( st, "ExprSymbol" );
            print_tmstring( st, t->s );
            tm_closecons( st );
            break;

        default:
            FATALTAG( t->tag );
    }
}

/* Print an element 't' of class type 'ExprTimes'
 * using print optimizer.
 */
static void print_ExprTimes( TMPRINTSTATE *st, const ExprTimes t )
{
    if( t == ExprTimesNIL ){
        tm_printword( st, "@" );
        return;
    }
    switch( t->tag ){
        case TAGExprTimes:
            tm_opencons( st );
            tm_printword( st, "ExprTimes" );
            print_expr( st, t->a );
            print_expr( st, t->b );
            tm_closecons( st );
            break;

        default:
            FATALTAG( t->tag );
    }
}

/* Print a list of elements 'l' of type 'command'
   using print optimizer.
 */
void print_command_list( TMPRINTSTATE *st, const command_list l )
{
    unsigned int ix;

    if( l == command_listNIL ){
        tm_printword( st, "@" );
        return;
    }
    tm_openlist( st );
    for( ix=0; ix<l->sz; ix++ ){
        print_command( st, l->arr[ix] );
    }
    tm_closelist( st );
}
```

```
/*********************************************
 *    Duplication routines                    *
 *********************************************/

#ifdef LOGNEW
#undef rdup_command
#define rdup_command(e) real_rdup_command(e,_f,_l)
#undef rdup_expr
#define rdup_expr(e) real_rdup_expr(e,_f,_l)
#define rdup_ExprConst(e) real_rdup_ExprConst(e,_f,_l)
#define rdup_ExprNegate(e) real_rdup_ExprNegate(e,_f,_l)
#define rdup_ExprPlus(e) real_rdup_ExprPlus(e,_f,_l)
#define rdup_ExprSymbol(e) real_rdup_ExprSymbol(e,_f,_l)
#define rdup_ExprTimes(e) real_rdup_ExprTimes(e,_f,_l)
static ExprConst real_rdup_ExprConst( const ExprConst, const char *, const int );
static ExprNegate real_rdup_ExprNegate( const ExprNegate, const char *, const int );
static ExprPlus real_rdup_ExprPlus( const ExprPlus, const char *, const int );
static ExprSymbol real_rdup_ExprSymbol( const ExprSymbol, const char *, const int );
static ExprTimes real_rdup_ExprTimes( const ExprTimes, const char *, const int );
#else
static ExprConst rdup_ExprConst( const ExprConst );
static ExprNegate rdup_ExprNegate( const ExprNegate );
static ExprPlus rdup_ExprPlus( const ExprPlus );
static ExprSymbol rdup_ExprSymbol( const ExprSymbol );
static ExprTimes rdup_ExprTimes( const ExprTimes );
#endif
/* Recursively duplicate a class command element 'e'. */
#ifdef LOGNEW
command real_rdup_command( const command e, const char *_f, const int _l )
#else
command rdup_command( const command e )
#endif
{
    tmstring i_lhs;
    expr i_rhs;

    if( e == commandNIL ){
        return commandNIL;
    }
    i_lhs = rdup_tmstring( e->lhs );
    i_rhs = rdup_expr( e->rhs );
    return new_command( i_lhs, i_rhs );
}

/* Recursively duplicate a class expr element 'e'. */
#ifdef LOGNEW
expr real_rdup_expr( const expr e, const char *_f, const int _l )
#else
expr rdup_expr( const expr e )
#endif
{
    if( e == exprNIL ){
        return exprNIL;
    }
    switch( e->tag ){
        case TAGExprPlus:
            return (expr) rdup_ExprPlus( (ExprPlus) e );

        case TAGExprTimes:
            return (expr) rdup_ExprTimes( (ExprTimes) e );

        case TAGExprNegate:
```

```
            return (expr) rdup_ExprNegate( (ExprNegate) e );

        case TAGExprConst:
            return (expr) rdup_ExprConst( (ExprConst) e );

        case TAGExprSymbol:
            return (expr) rdup_ExprSymbol( (ExprSymbol) e );

        default:
            FATALTAG( e->tag );
    }
    return exprNIL;
}

/* Recursively duplicate a class ExprConst element 'e'. */
#ifdef LOGNEW
static ExprConst real_rdup_ExprConst( const ExprConst e, const char *_f, const int _l )
#else
static ExprConst rdup_ExprConst( const ExprConst e )
#endif
{
    int i_n;

    if( e == ExprConstNIL ){
        return ExprConstNIL;
    }
    i_n = rdup_int( e->n );
    return new_ExprConst( i_n );
}

/* Recursively duplicate a class ExprNegate element 'e'. */
#ifdef LOGNEW
static ExprNegate real_rdup_ExprNegate( const ExprNegate e, const char *_f, const int _l )
#else
static ExprNegate rdup_ExprNegate( const ExprNegate e )
#endif
{
    expr i_x;

    if( e == ExprNegateNIL ){
        return ExprNegateNIL;
    }
    i_x = rdup_expr( e->x );
    return new_ExprNegate( i_x );
}

/* Recursively duplicate a class ExprPlus element 'e'. */
#ifdef LOGNEW
static ExprPlus real_rdup_ExprPlus( const ExprPlus e, const char *_f, const int _l )
#else
static ExprPlus rdup_ExprPlus( const ExprPlus e )
#endif
{
    expr i_a;
    expr i_b;

    if( e == ExprPlusNIL ){
        return ExprPlusNIL;
    }
    i_a = rdup_expr( e->a );
    i_b = rdup_expr( e->b );
    return new_ExprPlus( i_a, i_b );
```

```
}

/* Recursively duplicate a class ExprSymbol element 'e'. */
#ifdef LOGNEW
static ExprSymbol real_rdup_ExprSymbol( const ExprSymbol e, const char *_f, const int _l )
#else
static ExprSymbol rdup_ExprSymbol( const ExprSymbol e )
#endif
{
    tmstring i_s;

    if( e == ExprSymbolNIL ){
        return ExprSymbolNIL;
    }
    i_s = rdup_tmstring( e->s );
    return new_ExprSymbol( i_s );
}

/* Recursively duplicate a class ExprTimes element 'e'. */
#ifdef LOGNEW
static ExprTimes real_rdup_ExprTimes( const ExprTimes e, const char *_f, const int _l )
#else
static ExprTimes rdup_ExprTimes( const ExprTimes e )
#endif
{
    expr i_a;
    expr i_b;

    if( e == ExprTimesNIL ){
        return ExprTimesNIL;
    }
    i_a = rdup_expr( e->a );
    i_b = rdup_expr( e->b );
    return new_ExprTimes( i_a, i_b );
}

/***********************************************
 *    Scan routines.                           *
 ***********************************************/

#ifdef LOGNEW
#define fscan_command(f,ep) real_fscan_command(f,ep,_f,_l)
#define fscan_expr(f,ep) real_fscan_expr(f,ep,_f,_l)
#undef fscan_command_list
#define fscan_command_list(f,lp) real_fscan_command_list(f,lp,_f,_l)
static int real_fscan_command( FILE *, command *, const char *_f, const int _l );
static int real_fscan_expr( FILE *, expr *, const char *_f, const int _l );
#else
static int fscan_command( FILE *, command * );
static int fscan_expr( FILE *, expr * );
#endif

/* Read a class of type command
   from file 'f' and allocate space for it.
   Set the pointer 'p' to point to that structure.
 */
#ifdef LOGNEW
static int real_fscan_command( FILE *f, command *p, const char *_f, const int _l )
#else
static int fscan_command( FILE *f, command *p )
#endif
{
```

```
    int n;
    int c;
    char tm_word[11];   /* Largest constructor should fit in it.. */
    int err;

    *p = commandNIL;
    n = tm_fscanopenbrac( f );
    err = tm_fscanspace( f );
    if( err ){
        return 1;
    }
    c = getc( f );
    if( c == '@' ){
        return tm_fscanclosebrac( f, n );
    }
    ungetc( c, f );
    if( tm_fscancons( f, tm_word, 11 ) ){
        return 1;
    }
    if( strcmp( tm_word, "command" ) == 0 ){
        tmstring l_lhs;
        expr l_rhs;

        l_lhs = tmstringNIL;
        if( !err ){
            err = fscan_tmstring( f, &l_lhs );
        }
        l_rhs = exprNIL;
        if( !err ){
            err = fscan_expr( f, &l_rhs );
        }
        *p = new_command( l_lhs, l_rhs );
    }
    else {
        (void) sprintf( tm_errmsg, tm_badcons, "command", tm_word );
        return 1;
    }
    if( err ){
        return 1;
    }
    return tm_fscanclosebrac( f, n );
}

/* Read a class of type expr
   from file 'f' and allocate space for it.
   Set the pointer 'p' to point to that structure.
 */
#ifdef LOGNEW
static int real_fscan_expr( FILE *f, expr *p, const char *_f, const int _l )
#else
static int fscan_expr( FILE *f, expr *p )
#endif
{
    int n;
    int c;
    char tm_word[14];   /* Largest constructor should fit in it.. */
    int err;

    *p = exprNIL;
    n = tm_fscanopenbrac( f );
    err = tm_fscanspace( f );
    if( err ){
```

```
    return 1;
}
c = getc( f );
if( c == '@' ){
    return tm_fscanclosebrac( f, n );
}
ungetc( c, f );
if( tm_fscancons( f, tm_word, 14 ) ){
    return 1;
}
if( strcmp( tm_word, "ExprPlus" ) == 0 ){
    expr l_a;
    expr l_b;

    l_a = exprNIL;
    if( !err ){
        err = fscan_expr( f, &l_a );
    }
    l_b = exprNIL;
    if( !err ){
        err = fscan_expr( f, &l_b );
    }
    *p =  (expr) new_ExprPlus( l_a, l_b );
}
else if( strcmp( tm_word, "ExprTimes" ) == 0 ){
    expr l_a;
    expr l_b;

    l_a = exprNIL;
    if( !err ){
        err = fscan_expr( f, &l_a );
    }
    l_b = exprNIL;
    if( !err ){
        err = fscan_expr( f, &l_b );
    }
    *p =  (expr) new_ExprTimes( l_a, l_b );
}
else if( strcmp( tm_word, "ExprNegate" ) == 0 ){
    expr l_x;

    l_x = exprNIL;
    if( !err ){
        err = fscan_expr( f, &l_x );
    }
    *p =  (expr) new_ExprNegate( l_x );
}
else if( strcmp( tm_word, "ExprConst" ) == 0 ){
    int l_n;

    l_n = intNIL;
    if( !err ){
        err = fscan_int( f, &l_n );
    }
    *p =  (expr) new_ExprConst( l_n );
}
else if( strcmp( tm_word, "ExprSymbol" ) == 0 ){
    tmstring l_s;

    l_s = tmstringNIL;
    if( !err ){
        err = fscan_tmstring( f, &l_s );
```

```
        }
        *p =  (expr) new_ExprSymbol( l_s );
    }
    else {
        (void) sprintf( tm_errmsg, tm_badcons, "expr", tm_word );
        return 1;
    }
    if( err ){
        return 1;
    }
    return tm_fscanclosebrac( f, n );
}

/* Read an instance of a list of datastructure of type command
   from file 'f' and allocate space for it. Set the pointer 'p' to
   point to that structure.
 */
#ifdef LOGNEW
int real_fscan_command_list( FILE *f, command_list *p, const char *_f, const int _l )
#else
int fscan_command_list( FILE *f, command_list *p )
#endif
{
    int c;
    int n;
    command nw;
    int err = 0;

    *p = command_listNIL;
    n = tm_fscanopenbrac( f );
    if( tm_fscanspace( f ) ){
        return 1;
    }
    c = getc( f );
    if( c == '@' ){
        return tm_fscanclosebrac( f, n );
    }
    ungetc( c, f );
    if( tm_fneedc( f, '[' ) ){
        return 1;
    }
    *p = new_command_list();
    if( tm_fscanspace( f ) ){
        return 1;
    }
    c = getc( f );
    if( c == ']' ){
        return 0;
    }
    if( c == EOF ){
        (void) strcpy( tm_errmsg, tm_badeof );
        return 1;
    }
    ungetc( c, f );
    for(;;){
        if( !err ){
            err = fscan_command( f, &nw );
        }
        *p = append_command_list( *p, nw );
        if( err || tm_fscanspace( f ) ){
            return 1;
        }
```

```
        c = getc( f );
        if( c == EOF ){
            (void) strcpy( tm_errmsg, tm_badeof );
            return 1;
        }
        if( c != ',' ){
            ungetc( c, f );
            break;
        }
    }
    if( tm_fneedc( f, ']' ) ){
        return 1;
    }
    return tm_fscanclosebrac( f, n );
}

/***********************************************
 *    delete_<type>_list routines               *
 ***********************************************/

/* Delete 'command' element at position 'pos' in list 'l'. */
command_list delete_command_list( command_list l, const unsigned int pos )
{
    unsigned int ix;

    if( l == command_listNIL ){
        FATAL( tm_nilptr );
    }
    if( pos >= l->sz ){
        return l;
    }
    rfre_command( l->arr[pos] );
    l->sz--;
    for( ix=pos; ix<l->sz; ix++ ){
        l->arr[ix] = l->arr[ix+1];
    }
    return l;
}

/***********************************************
 *    Miscellaneous routines                    *
 ***********************************************/
/* Print allocation and freeing statistics to file 'f'. */
void stat_calc( FILE *f )
{
    const char tm_allocfreed[] = "%-20s: %6ld allocated, %6ld freed.%s\n";

    fprintf(
        f,
        tm_allocfreed,
        "ExprConst",
        newcnt_ExprConst,
        frecnt_ExprConst,
        ((newcnt_ExprConst==frecnt_ExprConst)? "": "<-")
    );
    fprintf(
        f,
        tm_allocfreed,
        "ExprNegate",
        newcnt_ExprNegate,
        frecnt_ExprNegate,
        ((newcnt_ExprNegate==frecnt_ExprNegate)? "": "<-")
```

```
    );
    fprintf(
        f,
        tm_allocfreed,
        "ExprPlus",
        newcnt_ExprPlus,
        frecnt_ExprPlus,
        ((newcnt_ExprPlus==frecnt_ExprPlus)? "": "<-")
    );
    fprintf(
        f,
        tm_allocfreed,
        "ExprSymbol",
        newcnt_ExprSymbol,
        frecnt_ExprSymbol,
        ((newcnt_ExprSymbol==frecnt_ExprSymbol)? "": "<-")
    );
    fprintf(
        f,
        tm_allocfreed,
        "ExprTimes",
        newcnt_ExprTimes,
        frecnt_ExprTimes,
        ((newcnt_ExprTimes==frecnt_ExprTimes)? "": "<-")
    );
    fprintf(
        f,
        tm_allocfreed,
        "command",
        newcnt_command,
        frecnt_command,
        ((newcnt_command==frecnt_command)? "": "<-")
    );
    fprintf(
        f,
        tm_allocfreed, "command_list",
        newcnt_command_list,
        frecnt_command_list,
        ((newcnt_command_list==frecnt_command_list)? "": "<-")
    );
}

/* Return -1 if there is a structure that has freed more than allocated, or
 * else return 1 if there is a structure that has been freed less than
 * allocated, or else return 0.
 */
int get_balance_calc( void )
{
    /* Check for too many fre()s. */
    if( newcnt_command_list<frecnt_command_list ){
        return -1;
    }
    if( newcnt_ExprConst<frecnt_ExprConst ){
        return -1;
    }
    if( newcnt_ExprNegate<frecnt_ExprNegate ){
        return -1;
    }
    if( newcnt_ExprPlus<frecnt_ExprPlus ){
        return -1;
    }
    if( newcnt_ExprSymbol<frecnt_ExprSymbol ){
```

```
        return -1;
    }
    if( newcnt_ExprTimes<frecnt_ExprTimes ){
        return -1;
    }
    if( newcnt_command<frecnt_command ){
        return -1;
    }
    /* Check for too few free()s. */
    if( newcnt_command_list>frecnt_command_list ){
        return 1;
    }
    if( newcnt_ExprConst>frecnt_ExprConst ){
        return 1;
    }
    if( newcnt_ExprNegate>frecnt_ExprNegate ){
        return 1;
    }
    if( newcnt_ExprPlus>frecnt_ExprPlus ){
        return 1;
    }
    if( newcnt_ExprSymbol>frecnt_ExprSymbol ){
        return 1;
    }
    if( newcnt_ExprTimes>frecnt_ExprTimes ){
        return 1;
    }
    if( newcnt_command>frecnt_command ){
        return 1;
    }
    return 0;
}

/* ---- end of /usr/local/lib/tmc.ct ---- */
/* Code generation required 180 milliseconds. */
```

Appendix C

Upgrading code using the old C templates

The most recent version of the Tm templates provide `to_<type>()` macros to convert to a subclass. This is also used to access constructors from a constructor base type. In previous versions of the Tm templates this was done with another macro; for example:

```
void clear_leaf( btree tree )
{
    if( tree->tag == TAGBLeaf ){
        tree->BLeaf.v = 0;
    }
}
```

The following Tm template will produce a `sed` file that will replace all old-style access expressions with the new ones.

```
.foreach c ${conslist ${typelist}}
s/\([a-zA-Z_][a-zA-Z0-9_]*\) *-> *$c\./to_$c(\1)->/g
.endforeach
```

The conversion is not perfect: it will probably require some manual modifications, but nevertheless it will simplify the modification considerably.

Another useful tool is the following script:

```
#!/bin/csh -f
# apply sed(1) with commands in file $1 to the files $2 $3 ..
set sedfile = $1
set tmpfile = ch$$
set me = $0
shift
foreach i ($*)
    sed -f $sedfile < $i > $tmpfile
    if $status then
        echo "${me}: file '$i' left unchanged."
        rm $tmpfile
    else
        cmp -s $tmpfile $i
        if $status then
            mv $tmpfile $i
```

```
        endif
    endif
end
rm -f $tmpfile
```

Given an `sed` script and a list of files, it will apply the script to all files. Only the files that are actually changed are touched, so that the required recompilation is reduced to a minimum.

Index