# *LTX2X:* A LaTeX to X Auto-tagger

Peter R. Wilson
Catholic University of America*
Email: `pwilson@cme.nist.gov`

January 1997

**Abstract**

*LTX2X* is a table-driven program that will replace LaTeX commands by user defined text. This report describes the beta version of the system. *LTX2X* supports both a declaritive command style and an interpreted procedural language tentatively called EXPRESS-A. Details are given of the program functionality including examples. System installation instructions are provided.

## Contents



*This work was performed while a Guest Researcher at the National Institute of Standards and Technology

# List of Tables

# 1 Introduction

LaTeX [1], which is built on top of TeX [2], is a document tagging system that is very popular in the academic and scientific publishing communities because of the high quality typeset material that the system outputs for normal text and especially for mathematics.

In particular, many of the documents forming the International Standard ISO 10303, commonly referred to as STEP [3], have been written using LaTeX as the document tagging language. Lately there have been moves towards converting the STEP documents to embody SGML [4] rather than LaTeX markup. This has led to an interest in the automatic conversion from LaTeX to SGML documents. The *LTX2X* system is an initial attempt to provide a generic capability for converting LaTeX tags into other kinds of tags.

The *LTX2X* system described below is in a beta release state. That is, there is probably some more work to be done on it but experience from use is needed to determine desirable additional functionality. However, the code has been stable for some time. Bug reports or suggested enhancements (especially if the suggestions are accompanied by working code) are encouraged, as are constructive comments about this document.

Essentially, *LTX2X* reads a file containing LaTeX markup, replaces the LaTeX commands by user-defined text, and writes the result out to another file. The program operates from a command table that specifies the replacement text. In general, no programming knowledge or skills are required to write a command table, which *LTX2X* will then interpret. Some knowledge of LaTeX is required, but no more than is necessary for authoring a LaTeX document.

*LTX2X* has proved capable of performing such functions as:

- Conversion of documents marked up according to a specific LaTeX documentclass to documents tagged according to a specific SGML DTD.
- Removal of LaTeX commands to produce deTeXed source.
- Conversion of simple LaTeX documents to HTML [5] tagged documents for publication on the World Wide Web.

The remainder of this introduction gives an overview of the *LTX2X* program. The command table is described in more detail in section 2 and information on running the *LTX2X* program is provided in section 3. Section 4 gives an overview of the EXPRESS-A language.[1] Although the functionality available through the command table facility is suitable for many tasks, especially since an interpreter for the EXPRESS-A general programming language is included within *LTX2X*, section 5 gives details on how the system can be extended for cases where this proves to be inadequate.

The report ends with several appendices. An example command table for deTeXing a document is reproduced in A and some of the issues in converting from LaTeX to HTML are discussed in B. The known limitations of *LTX2X* are listed in C and a summary of the

[1] The overview is necessarily rather brief as I am shortly moving to a new place of employment and EXPRESS-A is the latest addition to the system.

command table facilities are given in D. Appendix E provides instructions on installing the *LTX2X* program, together with copyright and warranty information. Finally, F and Gprovide grammars for the command tableand EXPRESS-A, respectively.

### 1.1 Overview

The intent of Leslie Lamport, the author of LaTeX, was to provide a document tagging system that enabled the capture of the logical structure of a document. This system uses Donald Knuth's TeX system as its typesetting engine [2], and thus has an inherent capability for high quality typesetting.

All LaTeX commands are distinguished by starting with a backslash (\). Generally speaking, the name of a command is a string of alphabetic characters (e.g. `\acommand`). Commands may take arguments. Required arguments are enclosed in curly braces (i.e. `{` and `}`). Optional arguments are enclosed in square brackets (i.e. `[` and `]`). The general syntax for a command is the command name (preceded by a backslash) followed by the argument list with a maximum[2] of nine arguments.

The *LTX2X* program reads a LaTeX document file and outputs a transformation of this file. By default it outputs the normal text while for each LaTeX command and argument performs some user-specified actions; typically these actions involve the output of specific text corresponding to the particular command. The actions are specified in a *command table* file, written by the user, which is read into the *LTX2X* system before document processing is begun. A command table consists of a listing of the LaTeX commands of interest together with the desired actiond for each of these commands and their arguments. Different effects may be easily obtained by changing the command table file. For example, a simple command table file may be written that will delete all the LaTeX commands from a document, resulting in a plain ASCII file with no embedded markup.[3] A more complex command table may be written that will replace LaTeX tags with appropriate SGML tags.

In some circles it is traditional to introduce a programming language by providing an example program that prints 'Hello world'. In contrast, the following command table file called `bye.ct`, when used in conjunction with a typical vanilla LaTeX file, will transform the LaTeX file to a file that consists only of the words 'Goodbye document'.

```
C=        bye.ct   "Goodbye document" for ltx2x

TYPE= COMMAND
NAME= \documentclass
  START_TAG= "Goodbye document"
  PC_AT_END= NO_PRINT
END_TYPE

C= just in case a LaTeX v2.09 document
TYPE= COMMAND
```

[2]Under very unusual circumstances this limit may be exceeded.
[3]To afficionados, this process is known as de-TeXing.

```
NAME= \documentstyle
  START_TAG= "Goodbye document"
  PC_AT_END= NO_PRINT
END_TYPE

C= just in case there is no \documentclass/style command
TYPE= BEGIN_DOCUMENT
  START_TAG= "Goodbye document"
  PC_AT_END= NO_PRINT
END_TYPE

TYPE= OTHER_COMMAND
  PRINT_CONTROL= NO_PRINT
END_TYPE

TYPE= OTHER_BEGIN
  PRINT_CONTROL= NO_PRINT
END_TYPE

TYPE= OTHER_END
  PRINT_CONTROL= NO_PRINT
END_TYPE

END_CTFILE=  end of bye.ct
```

Essentially the command table instructs *LTX2X* what to print for each LaTeX command. A command table file consists of a series of commands, one per line and introduced by a keyword such as `TYPE=`. Keywords are case insensitive but by convention are written in upper case. Comments in a command table are introduced by the keyword `C=`.

The main body of a command table consists of the specification of LaTeX commands of interest and the actions to be taken for these. Each specification commences with the keyword `TYPE=` and is completed by the keyword `END_TYPE`, the relevant actions being listed between these two keywords.

*LTX2X* treats some LaTeX commands specially; among these are `\begin{document}` and `\end{document}`. In a command table these are specified by the types `TYPE= BEGIN_DOCUMENT` and `TYPE= END_DOCUMENT`. The actions at `\begin{document}` are firstly to print the string 'Goodbye document' (specified in the line `START_TAG= "Goodbye document"`) and secondly to stop printing any output (specified in the line `PC_AT_END= NO_PRINT`).

By not specifying the `END_DOCUMENT` entry, the default action is used for the `\end{document}` command.

The command table entries for the commands `\documentclass` and `\documentstyle` specify that, if either of these is in the source document, then it is to be replaced by the text string `"Goodbye document"`, and then all further printing is to be switched off.

The other three entries in the command table specify the actions for any other kind of LaTeX command. The keyword `OTHER_BEGIN` signifies a LaTeX command of the form

\begin{name} and OTHER_END signifies a command of the form \end{name}. The keyword OTHER_COMMAND signifies any other kind of LaTeX command (e.g., \acommand ... ). The actions declared for these are all PRINT_CONTROL= NO_PRINT which shuts off any printing of the command or its arguments. In the command table bye.ct these are only included to prevent printing before the \begin{document}.

To run *LTX2X* with the above command table, type the following (where > is assumed to be the system prompt):

```
> ltx2x -f bye.ct input.tex output.tex
```

where bye.ct is the name of the command table, and input.tex and output.tex are the names of the input LaTeX file and the resulting processed file respectively.

As an example of a more useful command table file, the following one called decomm.ct will remove all LaTeX comments from a typical LaTeX source file.

```
C=  decomm.ct  Command table file for ltx2x to de-comment LaTeX source

C= ------------------------------------ set newline characters
ESCAPE_CHAR= ?
NEWLINE_CHAR= N

C=   ---------------------------------- built in commands
TYPE= BEGIN_DOCUMENT
  START_TAG= "\begin{document}"
END_TYPE

TYPE= END_DOCUMENT
  START_TAG= "\end{document}"
END_TYPE

TYPE= BEGIN_VERB
  START_TAG= "\verb|"
END_TYPE

TYPE= END_VERB
  START_TAG= "|"
END_TYPE

TYPE= BEGIN_VERBATIM
  START_TAG= "\begin{verbatim}"
END_TYPE

TYPE= END_VERBATIM
  START_TAG= "\end{verbatim}"
END_TYPE

TYPE= LBRACE
  START_TAG= "{"
```

```
END_TYPE

TYPE= RBRACE
  START_TAG= "}"
END_TYPE

TYPE= PARAGRAPH
  START_TAG= "?N?N    "
END_TYPE

C= ------------------- define '\item' tags within lists

TYPE= BEGIN_LIST_ENV
NAME= itemize
  START_TAG= "\begin{itemize}"
  START_ITEM= "\item "
END_TYPE

TYPE= BEGIN_LIST_ENV
NAME= enumerate
  START_TAG= "\begin{enumerate}"
  START_ITEM= "\item "
END_TYPE

TYPE= BEGIN_LIST_ENV
NAME= description
  START_TAG= "\begin{description}"
  START_ITEM= "\item"
  START_ITEM_PARAM= "["
  END_ITEM_PARAM= "] "
END_TYPE

TYPE= END_LIST_ENV
  NAME= itemize
END_TYPE

TYPE= END_LIST_ENV
  NAME= enumerate
END_TYPE

TYPE= END_LIST_ENV
  NAME= description
END_TYPE

C=    --------------------- pass through all other LaTeX commands

TYPE= OTHER_COMMAND
END_TYPE
```

```
TYPE= OTHER_BEGIN
END_TYPE

TYPE= OTHER_END
END_TYPE

END_CTFILE= end of file decomm.ct
```

In the above command table file, the first pair of commands (`ESCAPE_CHAR=` and `NEWLINE_CHAR=`) define the character pair that are to be used to signify a 'newline' within a tag. An example of their use is later in the file in the `PARAGRAPH` command type.

As indicated above, *LTX2X* treats some LaTeX commands specially. These are listed next in the command table. The special LaTeX commands are the begin and end of the `document` and `verbatim` environments, together with the `\verb` command, left and right braces, the `\␣` command, and the *LTX2X* `PARAGRAPH` specification. There are default actions for these, but apart from the `\␣` command the defaults are not appropriate in this case. Above, the actions are to replace the LaTeX command by the string forming the LaTeX command. The exception is that paragraphs (the `PARAGRAPH` specification) should start with at least one blank line and be indented some spaces.

The LaTeX `\item` command is used within lists. *LTX2X* has to be told how to treat the `\item` command within each kind of list. This has been done above for the `itemize`, `enumerate` and `description` environments.

The final instructions in the command table file tell *LTX2X* to pass through the text of all other commands and their arguments. The end of the command table file is either the physical end of the file or the command `END_CTFILE=`, whichever comes first. The `END_CTFILE=` command acts like the `C=` command in that arbitrary text can be put after the command.

To use the `decomm.ct` command table to de-comment a LaTeX file, type the following (where `>` is assumed to be the system prompt):

```
> ltx2x -f decomm.ct input.tex output.tex
```

where `input.tex` and `output.tex` are the names of the input LaTeX file for de-commenting and the resulting de-commented version respectively.

## 2 The command table file

By default, *LTX2X* does not output any LaTeX comments. Otherwise, whenever it comes across a LaTeX command it looks at the data in the command table file to determine what actions it should take. The two most typical actions are either to print out the command as read in, or to replace the command by some (possibly empty) text.

Each line in a command table file is either blank or starts with a keyword followed by one or more blanks. For example, a comment in the file is a line that starts with `C=` ; the remainder of the line is any comment text. Comments may be placed anywhere in the file.

## 2.1 Special print characters in tags

*LTX2X* is written in C [6]. The C language enables certain non-printing characters to be defined. These are typically written in the form `\c` where `\` is the C escape character and `c` is a particular character. *LTX2X* understands some of these special printing characters and the command table enables these to be given non-default values.

The default escape character (\) may be redefined via the `ESCAPE_CHAR=` command. For example,

```
ESCAPE_CHAR= ?
```

will make the question mark character the escape character. Typically, the escape character is changed in most command tables to avoid clashing with the LaTeX \ character. The following commands can be used to redefine the C special characters. Each of these commands takes a single character as its value. If a relevant command is not given, then the default value is used.

`NEWLINE_CHAR=` a new line (default is `n`)

`HORIZONTAL_TAB_CHAR=` horizontal tab (default is `t`)

`VERTICAL_TAB_CHAR=` vertical tab (default is `v`)

`BACKSPACE_CHAR=` backspace (default is `b`)

`CARRIAGE_RETURN_CHAR=` carriage return (default is `r`)

`FORMFEED_CHAR=` formfeed (default is `f`)

`AUDIBLE_ALLERT_CHAR=` beep the terminal (default is `a`)

`HEX_CHAR=` following characters form the hexadecimal number of the character to be printed (default is `x`) (e.g. `?xA3`)

These command lines are all optional within a command table and their ordering is immaterial. However, if any are present then they must be at the beginning of the command table.

The above special characters are useful when specifying the replacement text for LaTeX commands.

## 2.2 LaTeX command types

The commands for controlling the actions performed on LaTeX commands are enclosed between the command lines `TYPE=` and `END_TYPE`, as below.

```
TYPE= CommandType
  C= a possibly empty set of commands
END_TYPE
```

where `CommandType` is an *LTX2X* keyword signifying the kind of LaTeX command being specified.

### 2.2.1 Built in command types

Some LaTeX commands are pre-defined within *LTX2X*. Default actions are provided for these but it is recommended that type specifications for each of these commands be put in the command table anyway. The keywords for these commands are:

`BEGIN_DOCUMENT` Corresponds to the LaTeX command `\begin{document}`.

`END_DOCUMENT` Corresponds to the LaTeX command `\end{document}`.

`BEGIN_VERBATIM` Corresponds to the LaTeX commands `\begin{verbatim}` and `\begin{verbatim*}`.

`END_VERBATIM` Corresponds to the LaTeX commands `\end{verbatim}` and `\end{verbatim*}`.

`BEGIN_VERB` Corresponds to the LaTeX commands `\verb` and `\verb*`, together with the succeeding character.

`END_VERB` Corresponds to the appearance of the character that completes the LaTeX commands `\verb` and `\verb*`.

`LBRACE` Corresponds to the LaTeX left brace character `{`.

`RBRACE` Corresponds to the LaTeX right brace character `}`.

`BEGIN_DOLLAR` Corresponds to the LaTeX `$` symbol signalling the start of an in-text math formula.

`END_DOLLAR` Corresponds to the LaTeX `$` symbol signalling the end of an in-text math formula.

`PARAGRAPH` Corresponds to the LaTeX protocol of a blank line signalling the start/end of a paragraph.

`SLASH_SPACE` Corresponds to the LaTeX `\␣` command.

`OTHER_COMMAND` Corresponds to any LaTeX command of the form `\command` not specified elsewhere within the command table.

`OTHER_BEGIN` Corresponds to any LaTeX command of the form `\begin{environment}` not specified elsewhere within the command table.

`OTHER_END` Corresponds to any LaTeX command of the form `\end{environment}` not specified elsewhere within the command table.

The ordering of these built in type specifications is immaterial. If any of the above are not specified within the command table then *LTX2X* will use their default action. With the exception of the `SLASH_SPACE` command type, the default action is to do nothing (i.e., produce no output). The default action for the `SLASH_SPACE` command type is to output a space.

### 2.2.2 Optional command types

For the purposes of *LTX2X*, LaTeX commands are divided into various classes. The keywords for these clases, and the class descriptions, are listed below.

`TEX_CHAR` Corresponding to LaTeX's special characters (with the exception of the $, { and } characters).

`CHAR_COMMAND` Corresponding to LaTeX commands of the type `\c` where `c` is a single non-alphabetic character.

`COMMAND` Corresponding to LaTeX commands of the type `\command`, where `command` is the name of the command (except for `\begin`, `\end` and `\item`).

`BEGIN_ENV` Corresponding to LaTeX commands of the type `\begin{environment}` where `environment` is the name of the environment, except for those list environments whose bodies consist of `\item` commands.

`END_ENV` Corresponding to LaTeX commands of the type `\end{environment}`, with the same restrictions as for `BEGIN_ENV`.

`BEGIN_LIST_ENV` Corresponding to LaTeX commands of the type `\begin{environment}` where `environment` is the name of an environment whose body consists of `\item` commands.

`END_LIST_ENV` Corresponding to LaTeX commands of the type `\end{environment}` to match `BEGIN_LIST_ENV`.

`VCOMMAND` Corresponding to a LaTeX `\verb`-like command.

`BEGIN_VENV` Corresponding to the start of a verbatim-like environment.

`END_VENV` Corresponding to the end of a verbatim-like environment.

`SECTIONING` Corresponding to LaTeX commands of the type `\command`, where `command` is a document sectioning command such as `chapter` or `subsection`.

SPECIAL Reserved for possible future use.

SPECIAL_COMMAND Corresponding to the COMMAND keyword, except that some special output processing is to be defined.

SPECIAL_BEGIN_ENV Corresponding to the BEGIN_ENV keyword, except that some special output processing is to be defined.

SPECIAL_END_ENV Corresponding to the END_ENV keyword, except that some special output processing is to be defined.

SPECIAL_BEGIN_LIST Corresponding to the BEGIN_LIST_ENV keyword, except that some special output processing is to be defined.

SPECIAL_END_LIST Corresponding to the END_LIST_ENV keyword, except that some special output processing is to be defined.

SPECIAL_SECTIONING Corresponding to the SECTIONING keyword, except that some special output processing is to be defined.

_PICTURE_ Corresponding to some of the LaTeX picture drawing commands.

COMMAND_... Corresponding to some of the LaTeX commands whose arrangements of required and optional arguments are untypical.

The ordering of these types within a command table is immaterial.

Each of the above type specifications requires a NAME= command, whose value is the name of the relevant command or environment being specified. For example, the following is a (partial) specification of the figure environment and the caption command.

```
TYPE= BEGIN_ENV
NAME= figure
END_TYPE

TYPE= END_ENV
NAME= figure
END_TYPE

TYPE= COMMAND
NAME= \caption
END_TYPE
```

### 2.3 Command action tags

When *LTX2X* reads a LaTeX command it performs the following actions:

1. Looks up the name of the command or environment in the command table. If it is not found, then the appropriate default type is used.

2. Sets the printing mode according to the `PC_AT_START=` command.
3. Performs the actions specified in the command table by the `START_TAG=` command.
4. Processes any specified arguments to the command.
5. Performs the actions specified in the command table by the `END_TAG=` command.
6. Sets the printing mode according to the `PC_AT_END=` command.

**NOTES** :

1. Except for the default processing of `OTHER_` types, it does not output the command itself.
2. If a tag action is not specified, then the default action is null (e.g., nothing will appear in the output).

Within a command table all text strings for output are enclosed within double quotes. For example:

```
START_TAG=     "Some "text" string\n another line of text."
```

Assuming that `\n` means a newline, when this string action is performed by *LTX2X* it will appear in the output file as:

```
Some "text" string
another line of text.
```

A text string starts with the first double quote and ends with the last double quote on the command line. A text string has to be written on a single line within the command table. C language special print characters can be embedded within the text string (e.g. the `\n` for a newline in the above example). Remember that the first section of the command table is used for specifying the particular command table version of these.

If a text string is too long to fit comfortably on a single line in the command table, it may be continued via the `STRING:` command. As many of these can be used in succession as required (subject to internal limitations within *LTX2X*).

For instance,

```
START_TAG=     "Some "text" string\n"
  STRING: "another line of text."
```

has the same effect as the previous example.

The following specification is designed to write out the contents of the `\caption` command[4], preceded by the word 'CAPTION' and followed by at least one blank line (assuming that the escape character has been set to `?`).

[4]Strictly speaking, the specification does not do this exactly, but this simplified illustration will be corrected in the next sections.

```
TYPE= COMMAND
NAME= \caption
  START_TAG= "?n      CAPTION "
  END_TAG= "?n?n"
END_TYPE
```

Assuming that somewhere in a LaTeX file there is the command

```
stuff
\caption{This is a caption.}
more stuff
```

then the expected effect (see footnote) is

```
stuff

    CAPTION This is a caption.

more stuff
```

### 2.4 Argument actions

LaTeX commands can take arguments. The text for a required argument is enclosed in curly braces, while the text for an optional argument is enclosed in square brackets. *LTX2X* can be directed to perform actions at the start and end of each argument.

The number of required arguments is specified by the command line `REQPARAMS=` where the value of the command is a digit between 1 and 9 inclusive.

*LTX2X* assumes that a command can have only one optional argument, and that this is either first or last in the argument list. The potential presence of an optional argument is indicated by the command line `OPT_PARAM=`, where the value is either the keyword `FIRST` (for first in the list) or `LAST` (for last in the list).

The actions to be performed at the start and end of each required argument are specified via the commands `START_TAG_1=` and `END_TAG_1=` for the first required argument, through `START_TAG_9=` and `END_TAG_9=` for the ninth argument. The actions to be performed at the start and end of the optional argument are specified by the command lines `START_OPT=` and `END_OPT=`.

The argument delimiters (the braces or brackets) are not printed.

In the simplest case, the action is to print a specified text string (enclosed in double quotes, and continued with `STRING:` commands if necessary). Other kinds of actions are also possible. An unspecified tag defaults to doing no action.

#### 2.4.1 Print options

**2.4.1.1 Argument processing** By default, *LTX2X* processes (i.e. outputs as appropriate) the text of a argument. Printing of the argument text may be disabled, if required. The command line that controls argument printing is of the form `PRINT_P1=` through `PRINT_P9=`

for required arguments and PRINT_OPT= for the optional argument. The value of these commands is one from several keywords, the most common being NO_PRINT; this switches off printing of the text of the indicated argument. Default printing is resumed after the indicated argument.

Continuing the caption example from earlier, we can now complete it. The full syntax of the LaTeX command is:

```
\caption[optional table of contents entry]{Caption in the text}
```

That is, it has one required argument, which prints the caption text both in the body of the document and in the table of contents, unless the first optional argument is present, in which case its value gets printed in the table of contents instead.

Assume that an instance of the caption command in a document is:

```
Some stuff
\caption[Short caption]{Long caption for the body of the text.}
More stuff
```

Recall the previous command table caption specification. The actual output from processing this would be

```
Some stuff

    CAPTION [Short caption]{Long caption for the body of the text.}

More stuff
```

because, unless *LTX2X* is told that there are command arguments and how they should be treated, it will just print them out together with their surrounding delimiters.

The following command table entry will give more acceptable results.

```
TYPE= COMMAND
NAME= \caption
  START_TAG= "?n      CAPTION "
  END_TAG= "?n?n"
  OPT_PARAM= FIRST
  PRINT_OPT= NO_PRINT
  REQPARAMS= 1
END_TYPE
```

For the above captioning instance, the output will now be:

```
Some stuff

    CAPTION Long caption for the body of the text.

More stuff
```

The default print mode is to print text to the output file.

The keywords that can be used to control argument printing are:

NO_PRINT Do not print anything.

TO_SYSBUF Print to the *LTX2X* system buffer.

TO_BUFFER num Print to the *LTX2X* buffer number num.

TO_FILE name Print to the file called name.

NO_OP Skip all processing of the argument.

Note that even if the print mode is set to NO_PRINT, the argument text will still be processed. Only the NO_OP specification temporarily turns off the processing.

#### 2.4.1.2 General printing

Just as the printing mode can be set for each argument, it can also be set at the start and end of processing a LaTeX command or environment.

The specifications PC_AT_START= and PC_AT_END= can be used to set the printing mode at the start of processing a command and at the end, respectively. The keywords that can be used in these specifications are:

NO_PRINT Do not print anything.

TO_SYSBUF Print to the *LTX2X* system buffer.

TO_BUFFER num Print to the *LTX2X* buffer number num.

TO_FILE name Print to the file called name.

RESET Reset the print mode back to what it was.

Unlike the argument printing controls, the print mode is not automatically reset. This has to be explicitly specified.

As an example, assume that it is required to remove all figure environments from a LaTeX source and put them into a file on their own. The following command table code could be used to accomplish this.

```
TYPE= BEGIN_ENV
NAME= figure
  PC_AT_START= TO_FILE allfigs.tex
  START_TAG= "?n\begin{figure}"
END_TYPE

TYPE= END_ENV
NAME= figure
  START_TAG= "\end{figure}"
  PC_AT_END= RESET
END_TYPE
```

When a LaTeX figure environment is started, printing is switched to go to the file called allfigs.tex. At the end of the figure environment, the print mode is reset back to what it was before the environment began. If at the first figure environment the allfigs.tex file did not exist, then *LTX2X* would create it automatically.

**2.4.1.3 Read actions** As noted above, one of the actions that can be specified for a LaTeX comand's argument is to set the print mode for printing to a buffer or a file. Similarly there are actions which will read from a buffer or a file and print the contents. Within an argument tag these kinds of actions are specified via the keyword `SOURCE:`. This can take one of several values:

`SYSBUF` Print the contents of the *LTX2X* system buffer.

`BUFFER num` Print the contents of the *LTX2X* buffer number `num`.

`FILE name` Print the contents of the file called `name`.

In a previous example, the LaTeX `figure` environments were all written to the file `allfig.tex`. This file could be read in again just before the end of the document so that all figures will be typeset after everything else.

```
TYPE= END_DOCUMENT
  END_TAG= "?n %  figures collected here by LTX2X ?n"
    SOURCE: FILE allfigs.tex
    STRING: "?n\end{document}?n"
END_TYPE
```

As another example of the use of the print actions consider the LaTeX `\maketitle` command. This typesets the arguments of the `\title`, `\author` and `\date` commands, which must have been previously specified but not necessarily in this ordering. Here is one way this can be simulated using *LTX2X*.

```
TYPE= COMMAND
NAME= \title
  START_TAG=
    RESET_BUFFER: 1
  REQPARAMS= 1
  PRINT_P1= TO_BUFFER 1
END_TYPE

TYPE= COMMAND
NAME= \author
  START_TAG=
    RESET_BUFFER: 2
  REQPARAMS= 1
  PRINT_P1= TO_BUFFER 2
END_TYPE

TYPE= COMMAND
NAME= \date
  START_TAG=
    RESET_BUFFER: 3
  REQPARAMS= 1
  PRINT_P1= TO_BUFFER 3
```

```
END_TYPE

TYPE= COMMAND
NAME= \maketitle
  START_TAG= "?n"
    SOURCE: BUFFER 1
    STRING: "?n?n"
    SOURCE: BUFFER 2
    STRING: "?n?n"
    SOURCE: BUFFER 3
    STRING: "?n?n"
  END_TAG=
    RESET_BUFFER: 1
    RESET_BUFFER: 2
    RESET_BUFFER: 3
END_TYPE
```

For the `\title` command, the print mode for its argument is set for printing to the buffer number 1. The single action at the start of the command is to make sure that buffer 1 is empty (the line `RESET_BUFFER: 1`). The actions for the `\author` and `\date` commands are similar, except that they print their argument texts to buffers 2 and 3 respectively.

The `\maketitle` command takes no arguments, so all actions must be placed under `START_TAG=` and/or `END_TAG=`. There are a set of actions specified for `START_TAG=`. Firstly a newline is printed and this is followed by the contents of buffer 1 (i.e., the text of the argument of the `\title` command). Then two new lines are printed, followed by the contents of buffer 2 (the author). Finally another two newlines are printed, the contents of buffer 3 (the date), and another two newlines. The actions for `END_TAG=` are to clear the contents of the three buffers.

Just to extend the example, here is a specification for the LaTeX `\thanks` command. *LTX2X* is not designed to do footnoting (as it does not do page breaking) so instead the thanks text will be placed inside parentheses on a new line.

```
TYPE= COMMAND
NAME= \thanks
  START_TAG= "?n ("
  REQPARAMS= 1
  END_TAG= ") "
END_TYPE
```

Given these command table specification and the following portion of a LaTeX document

```
\date{29 February 2000}
\title{The Calculation of Leap Days\thanks{Originally published in JIR}}
\author{A. N. Other}
...
\maketitle
```

then output from *LTX2X* will be:

```
The Calculation of Leap Days
 (Originally published in JIR)

A. N. Other

29 February 2000
```

Note that as the `\thanks` command appears within the argument of the `\title` command, it is written to the same place as the text of the argument of `\title`. Thus, it also gets written to the output file when `\maketitle` is processed.

#### 2.4.2 Print switching

There are individual actions that enable the printing destination to be changed at will within the action set for any particular tag.

`SWITCH_TO_BUFFER: num` Direct any following printing to the *LTX2X* buffer number `num`.

`SWITCH_TO_FILE: name` Direct any following printing to the file called `name`.

`SWITCH_TO_SYSBUF` Direct any following printing to the *LTX2X* system buffer.

`SWITCH_BACK:` Undo the effect of the last `SWITCH_TO...` action.

As an example of the utility of this type of action, consider again the LaTeX `\maketitle` command. When LaTeX processes this command, it typesets the date as specified by the `\date` command, or if this has not been specified then it prints the current date instead. We can arrange for *LTX2X* to do something similar by adding the following to the command table shown earlier for the `\date` and `\maketitle` commands.

```
TYPE= COMMAND
NAME= \documentclass
  OPT_PARAM= FIRST
  REQPARAMS= 1
  PRINT_OPT= NO_PRINT
  PRINT_P1= NO_PRINT
  START_TAG=
      c= Initialise buffer 3 to ‘Today’
    RESET_BUFFER: 3
    SWITCH_TO_BUFFER: 3
    STRING: "Today"
    SWITCH_BACK:
END_TYPE
```

At the start of the document, the above actions put the string `Today` into `BUFFER 3`, having first ensured that it is empty. If the LaTeX source includes a `\date` command, then the contents of the buffer will be overwritten, otherwise it will be as initialised. In any event, when the `\maketitle` command is processed, the value output for the date will be either `Today` or whatever the argument was of the `\date` command.

### 2.4.3 Notes on the use of buffers and files

Resetting a buffer or a file always has the effect of emptying it of an prior contents.

When printing from a buffer or a file, the entire contents are printed. There is no limit to the number of times that a buffer or a file can be used as a printing source.

When printing to a buffer, the new strings are appended at the end of the current contents of the buffer, at least until it overflows. Unlike the behaviour of files, this is independant of any intervening prints from the buffer.

When printing to a file, the new strings are appended at the end of the current contents of the file. However, if a file is printed to after it has been printed from, the prior contents of the file are lost, and the new string is added at the start of the file. In general, it is safest to treat files as either read-only or write-only.

### 2.4.4 User specified modes

Consider the LaTeX command `\\`. In normal text this signifies that a line break must occur. In a `tabular` environment, though, it signifies the end of a row in a table. Suppose that in the *LTX2X* procesing of a tabular environment it is required to start and end each row with a vertical bar and to seperate each column also with a vertical bar. However, in normal text a `\\` command should just translate into a newline. Just to complicate matters further, assume that in an `eqnarray` environment, the `&` column seperator is to be translated to some spaces, and that the string '(X)' is to be put at the end of every row.

In other words, we need to process some commands differently according to where they appear in the LaTeX source. An *LTX2X* command table provides this capability through mode setting and mode-dependent actions. Here is an example of using this facility to solve the requirements outlined above.

```
TYPE= BEGIN_ENV
NAME= tabular
  C= starting actions, etc., here
  END_TAG=
    SET_MODE: tabular
END_TYPE

TYPE= END_ENV
NAME= tabular
  START_TAG=
    RESET_MODE:
END_TYPE

TYPE= BEGIN_ENV
NAME= eqnarry
  C= starting actions, etc., here
  END_TAG=
    SET_MODE: eqn
END_TYPE
```

```
TYPE= END_ENV
NAME= eqnarray
  START_TAG= "    (X)?n"
    RESET_MODE:
END_TYPE

TYPE= TEX_CHAR
NAME= &
  START_TAG= "  |  "
IN_MODE= eqn
  START_TAG= "  "
END_MODE
END_TYPE

TYPE= CHAR_COMMAND
NAME= \\
  START_TAG= "?n"
IN_MODE= tabular
  START_TAG= " |?n"
    STRING: "     |  "
END_MODE
IN_MODE= eqn
  START_TAG= "    (X)?n"
END_MODE
END_TYPE
```

Let us look at the specification for the `\tabular` environment first. The `END_TAG=` action is specified by the single command line `SET_MODE: tabular`, where `tabular` is any convenient name for identifying a mode. Thus, this will set the mode to be `tabular`. The action at the end of the environment is to reset the mode (`RESET_MODE:`) to whatever its previous value was. It is assumed that the last row in any `tabular` environment is finished by `\\`. Similar actions are performed for the `eqnarray` environment, except that the mode is called `eqn` instead of `tabular`. The other difference is that it is assumed that the last row is not ended by `\\`, so the end of the `eqnarray` environment has to also act like the `\\`.

Turning now to the specification for the `&` command, the first part of the specification identifies the type and name of the LaTeX command. This is then followed by the mode-independent set of actions, which in this case consists of printing a vertical bar with some spaces on either side of it. Following these are any mode-dependent actions, bracketed between `IN_MODE=` and `END_MODE`. The value for `IN_MODE=` is the name of the relevent mode. In this case the only mode-dependent action occurs when `MODE eqn` is in effect and it is to print some spaces instead of the default spaces and vertical bar.

The specification for the `\\` command has its set of mode-independent default actions, namely just to print a newline, and two sets of mode-dependent actions. When the `tabular` mode is in effect, it prints some spaces, a vertical bar, a newline, more spaces, a vertical bar, and finally some more spaces. On the other hand, when the `eqn` mode is in effect, it prints

Table 1: A simple table typeset by LaTeX.

| Stock | 1994 | | 1995 | |
|---|---|---|---|---|
| | low | high | low | high |
| ABC | 27 | 36 | 23 | 45 |
| DEF | 53 | 72 | 19 | 54 |
| GHI | 28 | 49 | 17 | 79 |

some spaces, the string '(X)' and a newline. If a mode is in effect that is not defined within the specification (e.g., mode `anon`) it performs the default mode-independent actions.

As a perhaps more practical example, the following command table code will convert simple LaTeX `tabular` environments to appropriate mark-up for HTML tables. It is assumed that the `tabular` environment is always within a `table` environment.

To set the perspective a little, here is the code for a simple table in LaTeX:

```
\begin{table}[tbp]
\centering
\caption{A simple table typeset by \LaTeX.} \label{tab:lxtab}
\begin{tabular}{|l||r|r||r|r|} \hline
Stock & \multicolumn{2}{c||}{1994} & \multicolumn{2}{c|}{1995} \\ \cline{2-5}
      &  low    &  high  &   low  & high  \\ \hline
ABC   &  27     &  36    &   23   & 45     \\
DEF   &  53     &  72    &   19   & 54     \\
GHI   &  28     &  49    &   17   & 79     \\ \hline
\end{tabular}
\end{table}
```

This will be typeset as shown in table 1.

The corresponding HTML code for the table after translation is:

```
<p><center><table border>
<caption>A simple table typeset by LaTeX.</caption> <a name="tab:lxtab"></a>

<tr><td> Stock </td><td colspan=2> 1994 </td><td colspan=2> 1995 </tr>
<tr><td >
       </td><td > low </td><td > high </td><td > low </td><td > high </tr>
<tr><td > ABC </td><td > 27 </td><td > 36 </td><td > 23 </td><td > 45 </tr>
<tr><td >
DEF </td><td > 53 </td><td > 72 </td><td > 19 </td><td > 54 </tr>
<tr><td >
GHI </td><td > 28 </td><td > 49 </td><td > 17 </td><td > 79 </tr>
<tr><td >
</table></center>
```

In the HTML browser that I use this is displayed approximately as shown for table 2.

In HTML a table is enclosed between `<table>` and `</table>` tags. Each row of the table is enclosed between `<tr>` and `</tr>` tags, and each element in a row is enclosed between

Table 2: A simple table typeset after translation to HTML.

| Stock | 1994 | | 1995 | |
|---|---|---|---|---|
| | low | high | low | high |
| ABC | 27 | 36 | 23 | 45 |
| DEF | 53 | 72 | 19 | 54 |
| GHI | 28 | 49 | 17 | 79 |

<td> and </td> tags. Under certain circumstances the closing tags (i.e., those like </...>) can be inferred by the HTML procesors and need not be explicitly put into the source text. The equivalent HTML tags to a LaTeX \multicolumn{num}{col}{text} command are <td colspan=num> text </td>.

The general actions that *LTX2X* has to perform in doing the LaTeX to HTML translation are:

- The start and end of a table environment has to translate to the HTML start and end table tags <table> and </table>. (Actually this also needs to handle the centering of the HTML table and drawing a border round it as well.)

- The start and end of the tabular environment has to translate into a row start <tr><td> and end </tr> in the HTML table (and set the mode for the LaTeX \\ command).

- The \\ command must end one row of the HTML table and start the next row (</tr>?n<tr><td>). It need not end a data element as this is automatically closed by the end of the row.

- The & column delimeter must end one HTML data element and start another one (i.e., </td><td>).

- The difficulty is in handling the \multicolumn command. In the easy case the LaTeX to HTML translation is:
  ... & text & ... maps to
  ... </td><td> text </td><td> ....
  However, when a multicolumn is involved the translation is
  ... & \multicolumn{N}{P}{text} & ... maps to
  ... </td><td colspan=N> text </td><td> ....
  As there is no look-ahead in *LTX2X* we have to be careful about starting a data element after a & because at that point *LTX2X* cannot know whether or not a \multicolumn command comes next, or just an ordinary data element.

We solve this last problem partly by using buffers (numbers 8 and 9 in the specification below) as temporary storage, and partly by a subtle specification for the \multicolumn command.

```
C=   start of a table
TYPE= BEGIN_ENV
NAME= table
  START_TAG= "<center><table border>"
  OPT_PARAM= FIRST
  C=  ignore the optional positioning argument
  PRINT_OPT= NO_PRINT
END_TYPE

C=  end a table
TYPE= END_ENV
NAME= table
  START_TAG= "</table></center>"
END_TYPE

C=  start a tabular
TYPE= BEGIN_ENV
NAME= tabular
  START_TAG= "?n<tr><td"
    RESET_BUFFER: 8
    RESET_BUFFER: 9
  OPT_PARAM= FIRST
  C=  ignore the optional positioning argument
  PRINT_OPT= NO_PRINT
  REQPARAMS= 1
  C=  ignore the column specification
  PRINT_P1= NO_OP
  END_TAG=
    SET_MODE: tabular
  PC_AT_END= TO_BUFFER 9
END_TYPE

C=  end a tabular
TYPE= END_ENV
NAME= tabular
  PC_AT_START= RESET
  START_TAG= ">"
    RESET_BUFFER: 8
    RESET_BUFFER: 9
    RESET_MODE:
END_TYPE

C=  we can do some processing of the \mutlicolumn command
TYPE= COMMAND
NAME= \multicolumn
  PC_AT_START= TO_BUFFER 8
  REQPARAMS= 2
  START_TAG_1= " colspan="
```

```
  PRINT_P2= NO_PRINT
  PC_AT_END= RESET
END_TYPE

C=  now for the end/start of a row
TYPE= CHAR_COMMAND
NAME= \\
  START_TAG= "<br>"
IN_MODE= tabular
  PC_AT_START= RESET
  START_TAG=
    SOURCE: BUFFER 8
    STRING: "> "
    RESET_BUFFER: 8
    SOURCE: BUFFER 9
  END_TAG= "</tr>?n<tr><td "
    RESET_BUFFER: 9
  PC_AT_END= TO_BUFFER 9
END_MODE
END_TYPE

C= and the column seperator
TYPE= TEX_CHAR
NAME= &
  PC_AT_START= RESET
  START_TAG=
    SOURCE: BUFFER 8
    STRING: "> "
    RESET_BUFFER: 8
    SOURCE: BUFFER 9
  END_TAG= " </td><td "
    RESET_BUFFER: 9
  PC_AT_END= TO_BUFFER 9
END_TYPE
```

Regarding the `\multicolumn` specification, we state that as far as *LTX2X* is concerned, it only has two required parameters, and that the action for the second one is `NO_PRINT`. The first argument is written to buffer `8` after 'colspan=' has first been put into it. *LTX2X* will treat the actual third argument to the `\multicolumn` as ordinary text, just as if there was no `\multicolumn` in the LaTeX source. We use buffer `9` for storing the text of a data element. When *LTX2X* processes a `&` column delimeter it first outputs the contents of buffer `8` (the number of columns specification) and then appropriate HTML characters. It then outputs the contents of buffer `9` (the element text), finishes off the element and partially starts the next element. Similar actions are performed at the start of the `tabular` environment and at the end of each row in the table.

### 2.5 Sectioning command types

*LTX2X* does some particular processing for sectioning command types. Although LaTeX can determine where any section of a document ends, other tagging systems cannot always do this. They require both a 'begin section' and an 'end section' tag. *LTX2X* can take account of the nesting depth of document sections and, given appropriate specifications, can supply both 'begin section' and 'end section' tags appropriately. This requires a little bit more in the way of specifications than we have met so far.

For a `SECTIONING` command type, the command line `SECTIONING_LEVEL=` must be included within the specification. The value of this command is a keyword from the following list.

`PART` For a sectioning command equivalent to the LaTeX `\part` command.

`CHAPTER` For a sectioning command equivalent to the LaTeX `\chapter` command.

`SECT` For a sectioning command equivalent to the LaTeX `\section` command.

`SUBSECT` For a sectioning command equivalent to the LaTeX `\subsection` command.

`SUBSUBSECT` For a sectioning command equivalent to the LaTeX `\subsubsection` command.

`PARA` For a sectioning command equivalent to the LaTeX `\paragraph` command.

`SUBPARA` For a sectioning command equivalent to the LaTeX `\subparagraph` command.

When a sectioning command is read from the LaTeX source, *LTX2X* firstly performs the `END_TAG=` actions for any 'lower level' sections that this one is closing off. It then performs the `START_TAG=` actions for the current command, and stores its own `END_TAG=` actions for later use. It then goes on and process any arguments as usual. The `END_DOCUMENT` command automatically closes off any opened sections.

As an example, assume that some kind soul has supplied a LaTeX style file that makes the commands `\clause` synonymous with `\subsection`, and `\sclause` synonymous with `\subsubsection`, etc. Also assume that it is required to output start and end tags of the form `<div.1>` and `</div.1>` for sections, `<div.2>` for clauses, etc., and surround the headings with tags `<heading>` and `</heading>`. Further, the first optional argument is of no interest as the output is going to be used by a processing system unable to automatically handle tables of contents. Part of an appropriate command table for doing this is:

```
TYPE= SECTIONING
NAME= \section
  SECTIONING_LEVEL= SECT
  START_TAG= "?n?n<div.1>?n"
  END_TAG= "?n</div.1>"
  OPT_PARAM= FIRST
  PRINT_OPT= NO_PRINT
  REQPARAMS= 1
  START_TAG_1= "<heading>"
```

```
  END_TAG_1= "</heading>?n"
END_TYPE

TYPE= SECTIONING
NAME= \clause
  SECTIONING_LEVEL= SUBSECT
  START_TAG= "?n?n<div.2>?n"
  END_TAG= "?n</div.2>"
  OPT_PARAM= FIRST
  PRINT_OPT= NO_PRINT
  REQPARAMS= 1
  START_TAG_1= "<heading>"
  END_TAG_1= "</heading>?n"
END_TYPE
```

An example output resulting from this command table (if it had been applied to this document) is:

```
...
</div.2>
</div.1>

<div.1>
<heading>The command table file</heading>

    By default, ...
```

### 2.6 List environment types

In LaTeX the use of the `\item` command is restricted to within a list environment. The typeset appearance of an `\item` typically depends on the particular environment in which it is used. *LTX2X* has a limited capability of modifying its `\item` tagging output. It can also provide an 'end item' tag for those tagging systems that require such a thing.

For such list environments, identified by the command type keyword `BEGIN_LIST_ENV`, the following command lines should be included within the type specification.

`START_ITEM=` Actions to be performed at the start of each `\item` command in the list.

`END_ITEM=` Actions to be performed after processing all the `\item`'s text.

`START_ITEM_PARAM=` Actions to be performed at the start of an `\item`'s optional argument text.

`END_ITEM_PARAM=` Actions to be performed at the end of an `\item`'s optional argument text.

As usual, an unspecified tag defaults to no actions.

For example, assume that we are not interested in tagging the end of an item, but we do want to mark each item in an `itemize` environment with the lowercase letter 'o', each `enumerate` item with '(N)' and put a colon after the optional argument in a description environment. Also, each item should have some indentation from the left hand margin.

```
TYPE= BEGIN_LIST_ENV
NAME= itemize
  START_ITEM= "?n    o "
END_TYPE

TYPE= END_LIST_ENV
NAME= itemize
END_TYPE

TYPE= BEGIN_LIST_ENV
NAME= enumerate
  START_ITEM= "?n    (N) "
END_TYPE

TYPE= END_LIST_ENV
NAME= enumerate
END_TYPE

TYPE= BEGIN_LIST_ENV
NAME= description
  START_ITEM= "?n  "
  END_ITEM_PARAM= " : "
END_TYPE

TYPE= END_LIST_ENV
NAME= description
END_TYPE
```

With the above commands, this LaTeX text:

```
\begin{description}
\item[An example]
  \begin{itemize}
  \item the first item;
  \item the second item.
  \end{itemize}
\end{description}
```

will be transformed into:

```
  An example :
    o the first item;
    o the second item.
```

## 2.7 Character types

LaTeX treats some characters specially. These special characters are: #, $, %, &, ~, _, ^, \, {, }, and, under some circumstances, also the character @. *LTX2X* recognizes these special characters and, if directed, will perform specified actions; otherwise it treats them as it treats any alphanumeric character, which is just to print it.

It has already been stated that commands for the left and right braces (i.e. { and }) must be given within the command table as command types LBRACE, RBRACE respectively. The dollar symbol ($) must also be specified via the two command types BEGIN_DOLLAR and END_DOLLAR. Here is an example of replacing the dollar signs by tags intended to indicate the start and end of a mathematical phrase.

```
TYPE= BEGIN_DOLLAR
  START_TAG= "<math>"
END_TYPE

TYPE= END_DOLLAR
  START_TAG= "</math>"
END_TYPE
```

Commands for the other special LaTeX characters are specified with the TEX_CHAR command type keyword.

The characters _ (underscore) and ^ (caret) are used in LaTeX math mode to indicate subscripting and superscripting respectively. The following will replace ^ by <sup>, print the superscript text (which must be enclosed in braces[5]) and at the end close with </sup>.

```
TYPE= TEX_CHAR
NAME= ^
  START_TAG= "<sup>"
  REQPARAMS= 1
  END_TAG= "</sup>"
END_TYPE
```

Given the above specifications, then `$(2^{15} - 1)$` will be transformed into
<math>(2<sup>15</sup> - 1)</math>.

## 2.8 Verbatim like types

The command type VCOMMAND is for the procesing of LaTeX \verb-like commands where the argument of the command is to be typeset as-is. For example, there might be a command called \url which takes one argument which is meant to be an Internet URL. If the application was the conversion of a LaTeX document to HTML, then the following specification could be useful.

```
TYPE= VCOMMAND
NAME= \url
```

[5]It is good practice to always enclose superscript and subscript text in braces, even though TeX does not always require this.

```
  REQPARAMS= 1
  PRINT_P1= TO_BUFFER 7
  START_TAG=
    RESET_BUFFER: 7
  END_TAG= "<a href=""
    SOURCE: BUFFER 7
    STRING: "">"
    SOURCE: BUFFER 7
    STRING: "</a>"
    RESET_BUFFER: 7
END_TYPE
```

If the LaTeX source included:

```
... obtainable from
\url{http://www.cdrom.com/pub/tex}
```

then the resulting *LTX2X* output would be:

```
... obtainable from
<a href="http://www.cdrom.com/pub/tex">http://www.cdrom.com/pub/tex</a>
```

which, if this was then read via an appropriate browser, a link to the URL would be automatically established.

Similarly `verbatim`-like environments can also be specified with the types `BEGIN_VENV` and `END_VENV`. For example, the `html.sty` package defines three LaTeX environments for documents that might be converted from LaTeX tagging to HTML tagging. One of these, `latexonly` is for LaTeX code that is not to occur in the HTMLed document and another is `htmlonly` which contains HTML code that is required for an HTML version of the document but which is not to appear in the LaTeXed document. The third one is `rawhtml` which is for HTML code to be output verbatim to the HTML document source. These could be simulated by:

```
TYPE= BEGIN_VENV
NAME= latexonly
  PC_AT_START= NO_PRINT
END_TYPE

TYPE= END_VENV
NAME= latexonly
  PC_AT_END= RESET
END_TYPE

TYPE= BEGIN_ENV
NAME= htmlonly
END_TYPE

TYPE= END_ENV
NAME= htmlonly
END_TYPE
```

```
TYPE= BEGIN_VENV
NAME= rawhtml
END_TYPE

TYPE= END_VENV
NAME= rawhtml
END_TYPE
```

## 2.9 Odd command types

The majority of commands in LaTeX that take optional arguments have only a single optional argument that is either immediately after the command or after all the required arguments. There are, however, some commands that do not fit this pattern. This set of command types enables at least some of these 'odd' commands to be handled.

The command type keyword is of the form `COMMAND_code`, where `code` indicates the type and ordering of the arguments. The `code` is composed from combinations of the letters `O` (for an optional argument) and `P` (for a required parameter (i.e., argument)). The ordering of these letters in the `code` specifies the type and ordering of the command's arguments.

The 'odd' command types are:

`COMMAND_OOP` Corresponding to a LaTeX command of the form `\com[OptParam][OptParam]{ReqParam}`. For example, the `\makebox` command falls into this category.

`COMMAND_OOOPP` Corresponding to a LaTeX command of the form `\com[OptParam][OptParam][OptParam]{ReqParam}{ReqParam}`. For example, the `\parbox` command falls into this category.

`COMMAND_OPO` Corresponding to a LaTeX command of the form `\com[OptParam]{ReqParam}[OptParam]`. For example, the `\RequirePackage` and `\LoadClass` commands fall into this category.

`COMMAND_POOOP` Corresponding to a LaTeX command of the form `\com{ReqParam}[OptParam][OptParam][OptParam]{ReqParam}`.

`COMMAND_POOP` Corresponding to a LaTeX command of the form `\com{ReqParam}[OptParam][OptParam]{ReqParam}`. For example, the `\newcommand` and its companion commands fall into this category.

`COMMAND_POOPP` Corresponding to a LaTeX command of the form `\com{ReqParam}[OptParam][OptParam]{ReqParam}{ReqParam}`. For example, the `\newenvironment` and its companion command fall into this category.

As usual, the command name is required, as are any actions. However, it is not necessary to specify the number of required arguments (i.e. `REQPARAMS=`) nor the position of the optional argument (i.e. `OPT_PARAM=`), as *LTX2X* already has this information. The tag actions are according to the argument ordering given in the `code` and are specified by the required argument tags (e.g. `START_TAG_n=` and `END_TAG_n=`). Do not use any of the command lines for optional arguments. Argument actions are controlled in the usual manner.

A typical example of the use of these commands is to supress any processing of the LaTeX `\newcommand` and its ilk. For example:

```
TYPE= COMMAND_POOP
NAME= \providecommand
  PRINT_P1= NO_OP
  PRINT_P2= NO_OP
  PRINT_P3= NO_OP
  PRINT_P4= NO_OP
END_TYPE

TYPE= COMMAND_POOPP
NAME= \renewenvironment
  PRINT_P1= NO_OP
  PRINT_P2= NO_OP
  PRINT_P3= NO_OP
  PRINT_P4= NO_OP
  PRINT_P5= NO_OP
END_TYPE
```

### 2.10 Other command types

The `OTHER_` command types (`OTHER_COMMAND`, `OTHER_BEGIN` and `OTHER_END`) are very limited in what can be affected. Basically, these provide for default printing actions if the corresponding LaTeX command has not been identified elsewhere in the command table.

If there are no commands within the specification, the name of the command and all its arguments will be printed verbatim.

The command lines `START_TAG=` and `END_TAG=` cause the corresponding actions to be performed before and after the name of the command is printed. Any arguments are printed verbatim.

The command line `PRINT_CONTROL=` with a value of `NO_PRINT` causes the command name not to be printed, nor any arguments that *LTX2X* may find associated with the command.

### 2.11 Picture types

The `_PICTURE_` command types differ from all the other types in *LTX2X*, just as they do in LaTeX. In LaTeX some of the picture drawing commands take arguments of the form **`(number, number)`**, representing a coordinate pair, as well as the usual required arguments enclosed in curly braces and possibly an optional argument enclosed in square brackets. Within *LTX2X*, commands that take coordinate arguments are treated specially in the command table.

Generally speaking, the *LTX2X* command types are of the form `PICTURE_code`, where `code` indicates the type and ordering of the arguments. The `code` is composed from combinations of the letters `C` (for a coordinate argument), `O` (for an optional argument) and `P` (for a required argument). For example, `PICTURE_PCOP` indicates a picture command that has a required argument, followed by a coordinate argument, followed by an optional argument and finally another required argument.

The provided picture types are:

`BEGIN_PICTURE_CC` Corresponding to a LaTeX command of the form
`\begin{PictureEnv}(coords)(coords)`, where the final coordinate argument is optional.

`PICTURE_CCPP` Corresponding to a LaTeX command of the form
`\com(coords)(coords){ReqParam}{ReqParam}`. For example, the `\multiput` command falls into this category.

`PICTURE_CO` Corresponding to a LaTeX command of the form
`\com(coords)[OptParam]`. For example, the standard LaTeX `\oval` command falls into this category.

`PICTURE_COP` Corresponding to a LaTeX command of the form
`\com(coords)[OptParam]{ReqParam}`. For example, the `\makebox` and `\framebox` commands fall into this category.

`PICTURE_CP` Corresponding to a LaTeX command of the form
`\com(coords){ReqParam}`. For example, the `\put`, `\line` and `\vector` commands fall into this category.

`PICTURE_OCC` Corresponding to a LaTeX command of the form
`\com[OptParam](coords)(coords)`. For example, the `\graphpaper` command from the `graphpap` package falls into this category.

`PICTURE_OCCC` Corresponding to a LaTeX command of the form
`\com[OptParam](coords)(coords)(coords)`. For example, the `\qbezier` command falls into this category.

`PICTURE_OCO` Corresponding to a LaTeX command of the form
`\com[OptParam](coords)[OptParam]`. For example, the `\oval` command from the `pict2e` package falls into this category.

`PICTURE_PCOP` Corresponding to a LaTeX command of the form
`\com{ReqParam}(coords)[OptParam]{ReqParam}`. For example, the `\dashbox` and `\savebox` commands fall into this category.

`END_PICTURE` Corresponding to a LaTeX command of the form
`\end{PictureEnv}`.

As usual, the command name is required, as are any actions. However, it is not necessary to specify the number of required arguments (i.e. `REQPARAMS=`) nor the position of the optional argument (i.e. `OPT_PARAM=`), as *LTX2X* already has this information. The tag actions are according to the argument ordering given in the `code` and are specified by the required argument tags (e.g. `START_TAG_n=` and `END_TAG_n=`). Do not use any of the command lines for optional arguments. Argument actions controlled in the usual manner.

As an example, the following specifications within a command table should be sufficient to ensure that any picture commands in a source file are not passed through to the output file.

```
TYPE= BEGIN_PICTURE_CC
NAME= picture
PRINT_P1= NO_PRINT
PRINT_P2 = NO_PRINT
END_TYPE

TYPE= PICTURE_CP
NAME= \put
PRINT_P1= NO_PRINT
PRINT_P2= NO_OP
END_TYPE

TYPE= PICTURE_CCPP
NAME= \multiput
PRINT_P1= NO_PRINT
PRINT_P2= NO_PRINT
PRINT_P3= NO_OP
PRINT_P4= NO_OP
END_TYPE

TYPE= PICTURE_PCOP
NAME= \savebox
PRINT_P1= NO_OP
PRINT_P2= NO_PRINT
PRINT_P3= NO_OP
PRINT_P4= NO_OP
END_TYPE

TYPE= PICTURE_OCC
NAME= \graphpaper
PRINT_P1= NO_OP
PRINT_P2= NO_PRINT
PRINT_P3= NO_PRINT
END_TYPE

TYPE= PICTURE_OCCC
NAME= \qbezier
```

```
PRINT_P1= NO_OP
PRINT_P2= NO_PRINT
PRINT_P3= NO_PRINT
PRINT_P4= NO_PRINT
END_TYPE

TYPE= END_PICTURE
NAME= picture
END_TYPE
```

**NOTE 1:** The action NO_OP cannot be applied to an argument that is a coordinate pair.

**NOTE 2:** As *LTX2X* is essentially limited to printing actions, and cannot actually process any LaTeX picture drawing commands, the suppression of picture printing is probably the most usual use of the picture commands.

### 2.12 Special command types

The SPECIAL_ commands, namely SPECIAL_COMMAND, SPECIAL_BEGIN_ENV, SPECIAL_BEGIN_LIST, SPECIAL_END_ENV, SPECIAL_END_LIST and SPECIAL_SECTIONING, are provided for cases where some special kind of output processing is required that is not built into *LTX2X*. In order to implement any commands of these types, it is necessary to modify the internals of *LTX2X* and recompile the source code. This is not recommended.

### 2.13 File inclusion

A command table file can include other command table files. In turn an included file can recursively include other command table files. The file inclusion command line is

```
INCLUDE= FileName
```

where FileName is the name of the command table file to be included. The effect is that the above line is replaced by the contents of FileName.

For example, assume that there are three command table files called respectively **detex.ct**, **detex.l2x** and **detex.fl**. The contents of these files are:

```
C=  ----------file detex.ct
...
INCLUDE= detex.l2x
...
C= ----------end of detex.ct
END_CTFILE=        end of detex.ct
```

and for **detex.l2x** as:

```
C= ---------- file detex.l2x

TYPE= COMMAND
```

```
NAME= \lx
  START_TAG= "LTX2X"
END_TYPE

INCLUDE= detex.fl

TYPE= COMMAND
NAME= \ctab
  START_TAG= "command table"
END_TYPE

C= ---------- end of file detex.l2x
END_CTFILE=         end of file detex.l2x
```

and lastly `detex.fl` is:

```
C= ---------- file detex.fl

TYPE= COMMAND
NAME= \fl
  START_TAG= "FLaTTeN"
END_TYPE

C= ---------- end file detex.fl
END_CTFILE=         end file detex.fl
```

Then, as far as *LTX2X* is concerned, the original `detex.ct` file is treated as though it had been written as:

```
C=  ----------file detex.ct
...
C= ---------- file detex.l2x

TYPE= COMMAND
NAME= \lx
  START_TAG= "LTX2X"
END_TYPE

C= ---------- file detex.fl

TYPE= COMMAND
NAME= \fl
  START_TAG= "FLaTTeN"
END_TYPE

C= ---------- end file detex.fl

TYPE= COMMAND
NAME= \ctab
```

```
  START_TAG= "command table"
END_TYPE

C= ---------- end of file detex.l2x
...
C= ----------end of detex.ct
END_CTFILE=        end of detex.ct
```

Note that nasty things will happen if you have a cycle of inclusions. That is, you must not have anything similar to file `A` including file `B` which includes file `C` which in turn includes either file `A` or `B`.

## 2.14 Interpreter commands

*LTX2X* includes an interpreter for a procedural programming language that is based on the ISO international standard Express information modeling language [7, 8]. At the moment the programming language within *LTX2X* is anonymous, but for ease of reference I will call it Express-A(Express -Almost? -Approximate? -Anonymous?). The Express-A language is described later in section 4, but for now it is sufficient to know the commands that signify the start and end of this code.

The command `CODE_SETUP=` indicates the commencement of code to be run before any document processing occurs. The `END_CODE` command signifies the end of this code block. This block should be placed in the command table before any other commands except for the `ESCAPE...` commands, if any. This block can contain variable declarations, function and procedure declarations, and statements.

Code consisting purely of statements can be placed anywhere that a tagging action may be specified. These statements are enclosed between a `CODE:` and `END_CODE` pair of commands.

The Express-A language is described in detail in 4, but to give a flavour of it here is a simple possible application. It has been noted that *LTX2X* will find difficulty in processing the contents of the LaTeX `picture` environment. The following portions of a command table write the contents of a `figure` environment to an external file and uses the programming language to keep a count of the number of figures so processed.

```
c=  declare and initialise a variable
CODE_SETUP=
  LOCAL
    fignum : INTEGER;
  END_LOCAL;
  fignum := 0;
END_CODE

c= write figure contents to an external file
TYPE= BEGIN_ENV
NAME= figure
```

```
  OPT_PARAM= FIRST
  PRINT_OPT= NO_PRINT
  PC_AT_START= TO_FILE figs.tmp
  START_TAG=
    CODE:
      fignum := fignum + 1;           -- increment figure counter
      println;                        -- print a blank line
      println('%%% FIGURE ', fignum); -- write counter as a LaTeX comment
    END_CODE
    STRING: "\begin{figure}"
END_TYPE

c= close figure environment, back to normal output, and output
c= text indicating that a figure should be here
TYPE= END_ENV
NAME= figure
  PC_AT_START= RESET
  START_TAG=
    SWITCH_TO_FILE: figs.tmp
    STRING: "\end{figure}?n?n"
    SWITCH_BACK:
    CODE:
      println;
      println('PLACE FOR FIGURE ', fignum);
      println;
   END_CODE
END_TYPE
```

# 3 The LTX2X program

*LTX2X* is written using flex and bison. The resulting C code should compile on any system. More details are given later, but for the end-user the next section describes how to run the program, assuming that is available on your system.

## 3.1 Running LTX2X

The syntax for running the compiled version of *LTX2X* is:

```
ltx2x [-c] [-f table-file] [-p number] [-w] [-D dir_cat_char]
      [-P path_seperators] [-S]
      [-i number] [-l number] [-t] [-y number] [-C] [-E]
      input-file output-file
```

where elements in square brackets are options. The options fall into two groups, one for the casual user and the other for those who may be interested in the internals of *LTX2X*. The first group of options includes:

-c By default, *LTX2X* ignores all LaTeX comments in the input file. This option causes *LTX2X* to write the comments to the output file.

-f By default, *LTX2X* reads the command table from a file called `ltx2x.ct`. If the required command table is in a file with another name this option is used to change from the default file. For example,

```
> ltx2x in.tex out.l2x
```

reads a command table from `ltx2x.ct`, while

```
> ltx2x -f detex.ct in.tex out.l2x
```

reads a command table from file `detex.ct`.

-p This option causes *LTX2X* to 'pretty print' the output file (as far as it is able to). The `number` is required and it indicates the desired maximum number of characters per output line. If this is considered to be too small, then *LTX2X* chooses a value. Note that pretty printing is only applied to the source file — not to any replacement tags. That is, it only tries to format the running text from the source file.

-w By default, *LTX2X* outputs source white space just at it reads it. This option causes *LTX2X* to collapse any amount of contiguous white space to a single space. The `-p` option includes the `-w` option.

-D The value of this option is the character that the operating system uses to catenate directory names to form a path (see 3.1.1). The default value is a slash (i.e. `/`). The default could be changed to a backslash, for example, by `-D \`.

-P The environment variable (see 3.1.1) contains a list of directories (also known as path names). In the operating system that I use, these are separated by the colon (`:`) character which, together with the semi-colon and space characters, form the *LTX2X* default separators. The path separator characters can be changed with this option. For example, `-P :` will make the separators be a colon or a space (space is automatically included in the separator list).

-S This option enables the source level debugger (see 4.6) for any embedded EXPRESS-A code.

The second group of options are principally for those who might be extending the *LTX2X* system.

-i This produces information that may be useful for debugging the EXPRESS-A interpreter. `number` is an integer between 1 and 9 inclusive. The greater the number, the more diagnostics are generated.

`-l` This produces information that may be useful for debugging the *LTX2X* program. `number` is an integer between 1 and 9 inclusive. The greater the number, the more diagnostics are generated.

`-t` This generates diagnostics related to the processing of the command table file.

`-y` Like the `l` option, but produces diagnostic information from the parser (this is actually a null option, but may be useful in the future).

`-C` Disable any interpreter debugging information during the code generation pass. This is not necessary unless the `-i` option is used.

`-E` Disable any interpreter debugging information during the code execution processing.This is not necessary unless the `-i` option is used.

*LTX2X* first reads the specified command table file, together with any included files, looking first in the current directory, then in the directories specified by the environment variable (if it exists). It then reads the `input-file` from the current directory, performs the actions specified in the command table and outputs the results to the `output-file`.

Three other files are also generated.

- `ltx2xct.lis` — This contains a human-readable form of the internal representation of the command table. It may be useful if there are any errors in the original command table.
- `ltx2x.err` — This contains any error or warning messages generated by *LTX2X*. It also contains any diagnostic information that may have been requested via a command line option.
- `interp.csg` — This contains a human readable listing of the internal byte code generated by the EXPRESS-A interpreter.

When *LTX2X* is running normally it prints out a counter to the terminal indicating how many hundreds of input source file lines it has processed. Lack of such output is an indication that the program may be in a loop and chewing up CPU cycles to no avail. In this case, stop the program and examine the output for indications of where the trouble is occurring.

A limited number of errors are allowed when processing the command table and the input LaTeX file before *LTX2X* gives up and quits. In particular, if it is reading a command table file that includes another file, say one called `zilch`, that it cannot read, it prints the following message to the user's terminal.

```
Can't open file zilch
Enter new file name, or I to ignore, or Q to quit
:
```

A `Q` (or `q`) response stops *LTX2X* from any further processing. An `I` (or `i`) response causes *LTX2X* to stop looking for the included file and continue processing the current file. Any other response is taken to be the name of an included file, which *LTX2X* then tries to read. If it fails, then the above message is repeated. The user is given a limited number of opportunities to identify a readable file before *LTX2X* quits altogether with this message:

```
Last attempt. Can't open file zilch. I'm giving up.
```

Regarding performance, the time taken by *LTX2X* to process a document does not appear to be significantly different from the time to LaTeX the same document.

### 3.1.1 Directory searching

The program employs a search algorithm to find files that are not in the current directory. It first looks in the current directory and if a file of the given name is found, then that is used. If the file is not found, then it searches for it among directories that are specified in a system environment variable. This variable specifies a list of pathnames, where the directories forming the path are combined using a catenation character. For example, `dir1/dir2/dir3` could be a pathname, where the slash (`/`) is the catenation character. If it is looking for file `afile.txt` it will catenate the file name to the path name (e.g. `dir1/dir2/dir3/afile.txt`) and look for that. The pathnames in the list are separated by another character (in fact it can be one from a list of characters). For example here is a list of two pathnames; `dir1/dir2;dir1/dir4`, where the semi-colon (`;`) is the pathname separator.

By default, the program uses a slash (`/`) as the directory catenation character and the pathname separators can be a space, or a colon or a semi-colon (i.e., any of `␣:;`). All these characters can be altered via the program command line options, and should be set to match the conventions of your operating system.

The environment variable used by the program is LTX2XTABLES. On the operating system that I use, I set this in my login file like:

```
setenv LTX2XTABLES .:/dir1/dir2:/dir3/dir4
```

Your system may have different conventions. Note that if the environment variable is not set, only files in the current directory are considered.

## 3.2 System components

The system consists of five main components — a lexer, a parser, a library of support functions and command table parsing code, a user-defined library of functions, and an interpreter for the EXPRESS-A language.

### 3.2.1 The lexer

The lexer is generated by flex [9] (a more functional version of lex [10]). The source for the lexer is in file `l2x.l`. Its principal function is to read a LaTeX source file and recognize LaTeX commands. In general, it passes off the relevant command tokens to the parser for performing appropriate actions.

However, the lexer does do some processing of the source itself.

- It recognizes LaTeX comments (i.e. any text starting with a percent sign and ending with a newline). It silently ignores these comments, unless run with the `-c` option.
- It recognizes the 'standard' LaTeX verbatims. It will write out the appropriate tags at the start and end of the verbatim text, while writing the verbatim text directly to the output file, bypassing the parser. It does not distinguish between the unstarred and starred versions of the verbatims.
- Newlines and whitespace are directly written to the output file, again bypassing the parser. The `-p` and `-w` options control the final appearance of whitespace, newlines and paragraphing.
- It recognizes the command `\␣` i.e. a backslash followed by a space, and writes the tags defined in the `SLASH_SPACE` specification directly to the output file.

The lexer is designed to recognize four kinds of LaTeX commands.

- Commands of the form `\begin{environment}`
- Commands of the form `\end{environment}`
- Any other command of the form `\command`
- As special cases, commands that are 'verbatim-like' or 'verb-like'.

When it finds a command, it looks up the command or environment name in the command table and sends the appropriate token and its command table location to the parser. As a special case the contents of verbatim-like environments and the argument of verb-like coommands are processesd within the lexer and not sent to the parser.

### 3.2.2 The parser

The parser is generated by bison [9] (a more powerful version of YACC [11]). The source for the parser is in file `l2x.y`. Essentially it defines a very simple grammar for a LaTeX document. That is, the grammar is limited to generic kinds of commands and command arguments. It does not understand the 'meaning' of any of the commands or arguments.

When the parser receives a token from the lexer it tries to match it with one of the grammar rules, performing the actions specified by the command table. Here is an extract from the parser grammar file, `l2x.y`, for a LaTeX command that has two required arguments followed by an optional argument.

```
l2xComm2Opt: COMMAND_2_OPT
        {
          start_with_req($1);
        }
        ReqParam
        {
          action_p_p1($1,1);
        }
        ReqParam
        {
          action_p_opt($1,2);
        }
        OptParam
        {
          action_last_opt($1);
        }
        ;
```

The actions are enclosed in braces, and are interspersed with the elements of the grammar.

The token `COMMAND_2_OPT` indicates that the lexer has found a command that takes two required arguments followed by an optional argument. The parser then performs some actions. The `start_with_req` function is the standard *LTX2X* function for the first action in a command production where the final argument is optional. The `$1` refers to the location of the particular command in the command table, and its value is passed to the parser by the lexer.

The parser then expects a required argument (i.e. `{`, token `LBRACE`) as the start of the required argument, followed by the text of the argument and finished off by a right brace (i.e. `}`, token `RBRACE`); the grammar for all of this is specified in the production called `ReqParam`). If it finds these it performs some further actions, otherwise it reports an error. In this case the action is defined by the function `action_p_p1`, which is the standard action performed between two required arguments (the second argument in the function call specifies the Pth argument that has been recognized). Another required argument is then expected. In this case the action is defined by the function `action_p_opt`, which is the standard action performed between the end of the Pth required argument and the start of an optional argument. It then looks for an optional argument, the grammar for which is specified in the production called `OptParam`. The final action is specified by the standard function `action_last_opt` for finishing off a command that ends with an optional argument.

The grammar for a command that that has two required arguments, and possibly an initial optional argument is similar:

```
l2xComm2: COMMAND_2
        {
          start_with_opt($1);
```

```
}
OptParam
{
  action_opt_first($1);
}
ReqParam
{
  action_p_p1($1,1);
}
ReqParam
{
  action_last_p($1,2);
}
;
```

### 3.2.3 The support libraries

Source code for the C main program and support functions is in file `l2xlib.c`. The main program is responsible for reading in the command table and calling the lexer and parser to do the appropriate processing. The file also contains a variety of support functions that are, or could be, used in the lexer, parser, action library, or user-defined library.

The standard actions for the grammar are contained in file `l2xacts.c`.

### 3.2.4 The user-defined library

The intent of this library is that masochistic users can define their own functions for use within *LTX2X* when processing their `SPECIAL_` commands, without having to modify the *LTX2X* support or action libraries.

Source code for the user-defined library should be maintained in a file called `l2xusrlb.c` and a corresponding header file called `l2xusrlb.h`.

### 3.2.5 The EXPRESS-A interpreter

The EXPRESS-A interpreter is based on algorithms originally developed by Ronald Mak [19] for interpreting Pascal. His original algorithms have been modified and extended to cater for EXPRESS-A. The interpreter module has a minimal interface with the rest of the *LTX2X* system, and could easily be modified to be a stand-alone program (in fact it started that way in the first place). The interface between *LTX2X* and the interpreter is confined to the small `l2xistup.c` file.

# 4 The EXPRESS-A programming language

EXPRESS is a language for information modeling and includes both declarative and procedural aspects [7]. There are also two other companion languages called respectively EXPRESS-G and EXPRESS-I. The former of these is a graphical form of the declaritive aspects of

EXPRESS, and the later is an instiation and test case specification language. These languages are either ISO international standards [8] or on the way to becoming so [12].

Certain of the procedural aspects of EXPRESS and EXPRESS-I are relevent to the *LTX2X* concepts and so, together with some other reasons, it seemed appropriate to provide an interpreter for a similar language for use within *LTX2X*. EXPRESS-A provides a major subset of the EXPRESS procedural language, together with some Pascal-like additions for input and output. Of particular note, strings are a built-in type in EXPRESS-A. The language also supports three-valued logic and the concept of an 'indeterminate' value of any type.

Earlier I gave an example command table to replace the text of a LaTeX document with the words 'Goodbye document'. Here is an EXPRESS-A program that outputs 'Goodbye document'.

```
println('Goodbye document');
END_CODE
```

The following gives a brief overview of EXPRESS-A. For more details consult Schenck & Wilson [7].

### 4.1 Basic elements

EXPRESS-A is a case-insensitive language and uses the ASCII character set. Two kinds of comments are supported — an end of line comment, which starts with a `--` pair and continues until the end of the current line — and an extended comment. An extended comment starts with a `(*` pair and is ended by a matching `*)` pair; extended comments may be nested.

The language contains many reserved words, some of which are only applicable to the EXPRESS and EXPRESS-I languages.

Identifiers are composed of an initial letter, possibly followed by any number of letters, digits, and the underscore character.

Literals are self defining constant values. An integer literal consists of one or more digits, the first of which shall not be zero. Real numbers start with one or more digits, followed by a decimal point. Further digits may occur after the point, and finaly there may be an exponent in the 'e' notation format (e.g., `123.456e-78`).

A string literal is any sequence of characters enclosed by single quote marks. If a single quote mark is meant to form part of the string, two quote marks must be used at that point.

Logical literals consists of one of these keywords: `FALSE`, `UNKNOWN` or `TRUE`.

EXPRESS-A also includes some other constants. `PI` stands for the value of the mathematical constant $\pi$ (3.1415...), and `CONST_E` stands for the value of the mathematical constant $e$ (2.7182...), the base of natural logarithms. The special token `?` stands for an indeterminate value of any type. The three constants `THE_DAY`, `THE_MONTH` and `THE_YEAR` are integer values for the current date holding the day of the month (1–31), the month of the year (1–12) and the year (four digits), respectively.

## 4.2 Data types

EXPRESS-A is a typed language. The simple data types are: INTEGER, REAL, STRING and LOGICAL.

The aggregation data types are ARRAY, BAG, LIST, and SET. The array data type is of a fixed size and must have declared lower and upper bounds (index range), such as ARRAY [-7:10] OF. The other aggregate data types are dynamic in size, but may have lower and upper bounds specified for the number of elements, such as SET [2:5] OF, meaning a set that should have between two and five members. For the dynamic aggregates the upper bound may be given as ?, which means an unlimited upper bound, such as LIST [2:?] OF. If a bound specification is absent, then the dynamic aggregate can hold from zero to any number of elements.[6]

Aggregates are one dimensional, but can be chained together for multi-dimensional aggregates, like

```
ARRAY [1:4] OF LIST OF INTEGER;
```

The enumeration data type is a parenthesised comma seperated list of identifiers. These identifiers represent the values of the enumerated type; for instance

```
ENUMERATION OF (red, green, blue)
```

A defined data type is one declared and named by the user using the TYPE and END_TYPE construct. For example

```
TYPE length = REAL; END_TYPE;
TYPE crowd_size = INTEGER; END_TYPE;
TYPE signal_colour = ENUMERATION OF (red, amber, green); END_TYPE;
```

An entity data type consists of a list of attributes and their types, enclosed in a ENTITY and END_ENTITY pair. An entity type is named.

```
ENTITY an_ent;
  auditorium_width : length;
  audience         : crowd_size;
  title            : STRING;
  profit           : REAL;
END_ENTITY;
```

EXPRESS-A provides for algorithms in the form of functions and procedures.

A FUNCTION is an algorithm that operates on parameters and returns a single resultant value of a specified data type. An invocation of a function in an expression evaluates to the resultant value at the point of invocation. For example:

```
FUNCTION func (par1 : INTEGER; par2 : STRING) : STRING;
  LOCAL
    str : STRING;
    -- other variable declarations
```

[6] The dynamic aggregates may not be fully implemented due to lack of time.

```
  END_LOCAL;
  -- the algorithm statements are here
  RETURN(str);
END_FUNCTION;
```

Note that the parameters are typed.

A PROCEDURE is an algorithm that receives parameters from the point of invocation and operates on them in some manner. Changes to the parameters within the procedure are only reflected to the point of invocation when the formal parameter is preceded by the keyword VAR. For example:

```
PROCEDURE proc (par1 : INTEGER; VAR par2 : STRING);
  -- local declarations and the algorithm statements
END_PROCEDURE;
```

Note that the parameters are typed. In this case the value of par2 may be changed.

Variables are declared in a local block, enclosed by the keywords LOCAL and END_LOCAL. A variable declaration consists of an identifer and its type, such as:

```
LOCAL
  str    : STRING;
  e1, e2 : an_ent;     -- e1 and e2 are both of type an_ent
  e3     : an_ent;     -- so is e3
  num    : INTEGER;
  col    : signal_colour;
  matrix : ARRAY [1:15] OF ARRAY [1:15] OF REAL;
END_LOCAL;
```

The above declarations must be in the following order:

1. ENTITY and/or TYPE declarations
2. FUNCTION and/or PROCEDURE declarations
3. a LOCAL declaration block

After the above can come any number of statements.

## 4.3 Statements

Express-A supports the following statements:

- Null statement
- Assignment statement
- Call statement
- BEGIN ... END compound statement

- `CASE ... END_CASE` statement
- `IF ... THEN ... ELSE ... END_IF` statement
- `REPEAT ... WHILE ... UNTIL ... END_REPEAT` statement. This also includes the `ESCAPE` and `SKIP` statements
- `RETURN` statement

All the above statements are completed by a ; (semicolon). The *null* statement just consists of a semicolon.

The *assignment* statement is used to assign an instance to a local variable or parameter. The data types must be compatible.

```
LOCAL
  a, b, c : REAL;
END_LOCAL;
...
  a := 2.3E-6;
  b := a;
  a := -27.0;
  c := 33.3*b;
```

The *call* statement invokes a procedure or a function. The actual parameters provided with the call must agree in number, order and type with the formal parameters specified in the procedure or function declaration. The supplied parameter values must be assignment compatible with the formal parameters. This is an example of calling the EXPRESS-A defined `INSERT` procedure which takes three parameters:

```
INSERT(my_list, list_element, 0);
```

The *compound* statement consists of one or more statements enclosed between a `BEGIN` and `END` pair. The enclosed statements are treated as a single statement.

```
...
  BEGIN
    a := 2.3e-7;
    b := a;
    c := b*33.3;
  END;
```

The *case* statement is a means of selectively executing statements based on the value of an expresion.

```
LOCAL
  a : INTEGER;
  x, y : REAL;
END_LOCAL;
...
  a := 2;
```

```
  x := 21.9;
  CASE 2*a OF
    1         : x := SIN{x};
    2         : x := SQRT(x);
    3         : x := LOG(x);
    4         : x := COS(x);  -- this is executed
    5, 6      : y := y**x;
    OTHERWISE : x := 0.0;
  END_CASE;
```

The integer expression following the `CASE` keyword is evaluated. The result is compared to the values of the case labels and the statement following the first matching label is executed. Execution then continues at the statement following the `END_CASE;`. If no label matches, then no statements within the case block are executed, except if an `OTHERWISE` label is included, which will match anything. All other labels are examined before looking for the `OTHERWISE`.

The *if ... then ... else* statement allows the conditional execution of statements depending on the value of a `LOGICAL` expression. When the expression evaluates to `TRUE` the statement(s) following the `THEN` are executed, after which control passes to the statement following the closing `END_IF`. When the logical expression evaluates to `FALSE` or `UNKNOWN` the `THEN` statements are jumped over and execution starts at the statement(s) following the `ELSE` keyword if present, or at the statement following the `END_IF` keyword.

```
IF a > 20 THEN
  b := a + 2;
  c := c - 1;
ELSE
  IF a > 10 THEN
    b := a + 1;
  ELSE
    c := c + 1;
  END_IF;
END_IF;
```

The *repeat* statement is used to control the conditional repetition of a series of statements. The control conditions are:

- finite iteration until an integer expression reaches a specified value;
- `WHILE` a logical condition is `TRUE`;
- `UNTIL` a logical condition is `TRUE`.

```
REPEAT i := 100 TO 0 BY -7 WHILE r >= 0.0 UNTIL err < 1.0e-8;
  ...
  r := ...;
  err := ...;
END_REPEAT;
```

At entry to the `REPEAT` statement the iteration variable is initialized to the first bound. If the variable less than or equal to the `TO` bound and the increment is positive, or the variable is less than the `TO` bound and the increment is negative, processing jumps to after the `END_REPEAT`, otherwise processing continues. The `WHILE` condition is checked and if `TRUE` then the statements in the body are executed. After these have been executed the `UNTIL` condition is checked. If this is not `TRUE` then processing continues by incrementing the iteration variable by either unity or by the `BY` value if present. The whole process then starts again with the checking of the iteration variable against the `TO` bound.

All three types of controls are optional. If none are given then the `REPEAT` statement will loop for ever. The *escape* statement causes an immediate transfer out of the `REPEAT` statement in which it occurs. The *skip* statement causes a jump to the end of the `REPEAT` statement in which it occurs (i.e., to the point where the `UNTIL` expression is tested).

```
REPEAT UNTIL (a = 1);
  ...
  IF a = 0 THEN
    ESCAPE;
  END_IF;
  ...
  IF a > 10 THEN
    SKIP;
  END_IF;
  ...
  ...
-- SKIP transfers control to here
END_REPEAT;
 -- ESCAPE transfers control to here
```

The *return* statement terminates the execution of a `FUNCTION` or `PROCEDURE`. The `RETURN` statement within a function must specify an expression, the value of which is the value returned by the function. A `RETURN` in a procedure must not specify an expression.

```
RETURN(a <> b);  -- example for within a function
RETURN;          -- example for within a procedure
```

## 4.4 Expressions

Expressions are combinations of operators, operands and function calls which are evaluated to produce a value. The simplest expression is either a literal value or the name of a variable.

### 4.4.1 Arithmetic operators

The arithmetic operators act on number values and produce a number result. If any operand is indeterminate (i.e., `?`) then the result is also indeterminate. The operators are:

**Unary** The operators `+` and `-`, the latter of which negates its following operand.

**Binary** Addition (`+`), subtraction (`-`), multiplication (`*`), real division (`/`), exponentiation (`**`), integer division (`DIV`), and modulo (`MOD`).

### 4.4.2 Relational operators

The result of a relational expression is a `LOGICAL` value. If either operand is indeterminate, the expression evaluates to `UNKNOWN`.

**Value comparison** Equal (`=`), not equal (`<>`), greater than (`>`), less than (`<`), greater than or equal (`>=`), and less than or equal (`<=`).

**Membership** The `IN` operator tests an item for membership in a dynamic aggregate (e.g., `IF fred IN mylist THEN ...`).

**Matching** The `LIKE` operator compares a string against a pattern, evaluating to `TRUE` if they match. The pattern characters are:

- `@` Matches any letter.
- `^` Matches any upper-case letter.
- `?` Matches any character.
- `&` Matches remainder of string.
- `#` Matches any digit.
- `$` Matches a substring terminated by a space character or end-of-string.
- `*` Matches any number of characters.
- `\` Begins a pattern escape sequence.
- `!` Negation character (used with the other characters).
- Any other character matches itself.

Some examples:

- `'The quick red fox' LIKE '$$$$'` is `TRUE`.
- `'Page 231' LIKE '$ ###'` is `TRUE`.
- `'Page 27' LIKE 'Page ###'` is `FALSE`.
- `'\aaaa' LIKE '\\aaaa'` is `TRUE`.
- `'\aaaa' LIKE '\aaaa'` is `FALSE`.
- `'aaaa' LIKE 'a@@a'` is `TRUE`.

### 4.4.3 Logical operators

The logical operators produce a logical result. Except for the `NOT` operator which takes one logical operand (e.g., `NOT op`), they take two logical operands (e.g., `op1 XOR op2`).

The evaluation of the `NOT` operator is given in table 3.

The evaluation of the `AND`, `OR` and `XOR` operators is given in table 4.

Table 3: The NOT logical operator

| Operand value | Result value |
|---|---|
| TRUE | FALSE |
| UNKNOWN | UNKNOWN |
| FALSE | TRUE |

Table 4: The AND, OR and XOR logical operators

| Op1 | Op2 | Op1 AND Op2 | Op1 OR Op2 | Op1 XOR Op2 |
|---|---|---|---|---|
| TRUE | TRUE | TRUE | TRUE | FALSE |
| TRUE | UNKNOWN | UNKNOWN | TRUE | UNKNOWN |
| TRUE | FALSE | FALSE | TRUE | TRUE |
| UNKNOWN | TRUE | UNKNOWN | TRUE | UNKNOWN |
| UNKNOWN | UNKNOWN | UNKNOWN | UNKNOWN | UNKNOWN |
| UNKNOWN | FALSE | FALSE | UNKNOWN | UNKNOWN |
| FALSE | TRUE | FALSE | TRUE | TRUE |
| FALSE | UNKNOWN | FALSE | UNKNOWN | UNKNOWN |
| FALSE | FALSE | FALSE | FALSE | FALSE |

#### 4.4.4 Miscellaneous

**Function call** A function may be called without the result necessarily being assigned to a variable. If `fun` is a function with two arguments (for simplicitly integer arguments) and returning a logical value, then

```
log := fun(i1, i2);
fun(i3, 24*i4);
```

are both legitimate calls.

**Dot operator** The dot operator is used to access an attribute from an entity. If `ent` is an `ENTITY` type with an attribute `att`, then `ent.attr` evaluates to the value of the `attr` attribute within the `ent`.

**String operators**
The `+` operator takes two strings as its operands and evaluates to the string that is the concatenation of its operands. For example:

```
str1 := 'string1';
str2 := 'string2';
str1 := str1 + str2;
-- str1 = 'string1string2'   is TRUE
```

The substring operator `[i1:i2]` is a postfix operator that when applied to a string, evalutes to the string whose characters are composed of the `i1`'th through the `i2`'th characters, inclusively, of its operand. Note that `i2` must be greater than or equal to `i1`, and both must be within the limits of the number of characters in the string. For example:

```
str1 := 'string';
str2 := str1[2:4];
str1 := str1 + str2;
-- str1 = 'tristring'   is TRUE
```

**Aggregate operators**

The index operator `[i]` is a postfix operator that can be applied to an aggregate operand; the expression evaluates to the value of the aggregate at the index position. For example, if `lagg` is a list of integers:

```
insert(lagg, 20, 0);
insert(lagg, 40, 0);
insert(lagg, 60, 0);
insert(lagg, 80, 0);
-- lagg[2] = 60    is TRUE
```

**Interval expression**

An *interval expression* is a `LOGICAL` expression consisting of three operands and two operators. It has the form:

```
{ low op1 test op2 high }
```

where `op1` and `op2` are either of the two relational operators `<` or `<=`, and `low`, `test` and `high` are expressions of the same type. The interval expression is equivalent to:

```
((low op1 test) AND (test op2 high))
```

The value of the interval expression is given by

1. If any operand is indeterminate, then it evauates to `UNKNOWN`.
2. If either of the logical relationships evaluates to `FALSE`, then it evauates to `FALSE`.
3. If both logical relationships evalute to `TRUE`, then it evauates to `TRUE`.
4. Otherwise it evaluates to `UNKNOWN`.

For example:

```
i := 10;
{1 <= i < 20}  -- is TRUE
{1 <= i < 10}  -- is FALSE
i := ?;
{1 <= i < 10}  -- is UNKNOWN
```

## 4.5 Built in procedures and functions

### 4.5.1 Procedures

The following procedures are an integral part of EXPRESS-A. They are shown as signatures to inidicate the data types of the formal parameters. For convenience, `GENERIC` is used to indicate any type.

- `INSERT (VAR L:LIST OF GENERIC; E:GENERIC; P:INTEGER)`
  `INSERT` inserts the element `E` into a list `L` at position `P`. The insertion follows the existing element at `P`, so that if `P=0`, `E` will become the first element.

- `REMOVE (VAR L:LIST OF GENERIC; P:INTEGER)`
  `REMOVE` modifies the list `L` by deleting the element at position `P`.

- `SYSTEM (V:STRING)`
  `SYSTEM` passes the string `V` to the operating system. This is typically used to get the operating system to perform some action.

- `READ(VAR V1, V2,...:GENERIC)`, `READLN(VAR V1, V2,...:GENERIC)`
  These two procedures are similar to the Pascal procedures of the same name and put data from standard input into the variable(s) `V1`, etc.

  The argument is a comma-seperated list of variables. The variables may be of different types, but the types are limited to `INTEGER`, `REAL`, `LOGICAL`, and `STRING`. The procedure gets the next value of the variable's type from standard input and assigns it to the variable. An integer is recognised as a set of digits, optinally preceeded by a sign. A real is in either decimal or scientific notation (e.g., `12.34` or `1.234e1`). A logical is `TRUE`, `FALSE` or `UNKNOWN` (case independent, so `TRUE` could also be `tRuE`). A string is any non-empty set of characters ended by white space (e.g., `string␣` is one string but `ball␣of␣str8␣string␣` is four strings). The difference between `READ` and `READLN` is that the former performs the actions described above, while the latter will discard any remaining characters in the input line after processing its arguments.

- `WRITE(format)`, `WRITELN(format)`
  These two procedures are similar to the Pascal procedures of the same name. They write data to standard output.

  The `format` consists of a comma-seperated list of variables with optional spacing specifications. The variable types may be `INTEGER`, `REAL`, `LOGICAL`, or `STRING`. The `LOGICAL` and `STRING` types take no spacing declarations. An `INTEGER` variable can take one optional space specification which is an integer number specifying the minimum field width for printing the value (e.g., `int:6` to specify a minimum field width of 6 characters). A `REAL` variable can take two optional space specifications. The first is the field width and the second is the number of digits to be printed (e.g., `r:10:5` for printing with a field width of 10 characters and to a pecision of 5 digits). For example:

```
BEGIN_LOCAL
  int : INTEGER;
  r   : REAL;
  log : LOGICAL;
  str : STRING;
END_LOCAL;
  int := 23;
  r := 23.0;
  log := true;
  str := 'This is a string.';
WRITE('Example', int, r:10:5, ' ', log, ' ', str);
```

will produce:

```
Example      23     23.000 TRUE This is a string.
```

The difference between `WRITE` and `WRITELN` is that the latter will end the output line after it has output the values of its arguments. (`WRITELN` need take no arguments, in which case it justs ends the current output line).

- `PRINT(format)`, `PRINTLN(format)`
  These `PRINT` procedures are the same as the `WRITE` procedures, except that they send the data to the current *LTX2X* output destination.

### 4.5.2 Functions

The following functions are supplied as part of EXPRESS-A. They are exhibited as signatures to show the formal parameters. For convenience, `NUMBER` is being used to denote either an `INTEGER` or a `REAL` number.

- `ABS (V:NUMBER) : NUMBER;`
  `ABS` returns the absolute value of its argument.

- `COS (V:NUMBER) : REAL;`
  Returns the cosine of an an angle specified in radians.

- `EOF () :  LOGICAL;`
  Returns `TRUE` if the next character from standard input is 'end-of-file', otherwise it returns `FALSE`.

- `EOLN () :  LOGICAL;`
  Returns `TRUE` if the next character from standard input is 'end-of-line', otherwise it returns `FALSE`.

- `EXISTS (V:GENERIC) : LOGICAL;`
  The function `EXISTS` returns `FALSE` if its argument is indeterminate or does not exist, otherwise it returns `TRUE`.

- `EXP (V:NUMBER) : REAL;`
  Returns $e$ (the base of natural logarithms (`CONST_E`)) raised to the power of `V`.

- `HIBOUND (V:AGGREGATE OF GENERIC) : INTEGER;`
  `HIBOUND` returns the declared upper index of an `ARRAY` or the declared upper bound of a `BAG`, `LIST` or `SET`.

- `HIINDEX (V:AGGREGATE OF GENERIC) : INTEGER;`
  `HIINDEX` returns the declared upper index of an `ARRAY` or the number of elements in a `BAG`, `LIST` or `SET`.

- `LENGTH (V:STRING) : INTEGER;`
  Returns the number of characters in its argument.

- `LOBOUND (V:AGGREGATE OF GENERIC) : INTEGER;`
  `LOBOUND` returns the declared lower index of an `ARRAY` or the declared lower bound of a `BAG`, `LIST` or `SET`.

- `LOG (V:NUMBER) : REAL;`
  Returns the natural logarithm of its argument.

- `LOG2 (V:NUMBER) : REAL;`
  Returns the base 2 logarithm of its argument.

- `LOG10 (V:NUMBER) : REAL;`
  Returns the base 10 logarithm of its argument.

- `LOINDEX (V:AGGREGATE OF GENERIC) : INTEGER;`
  `LOINDEX` returns the declared lower index of an `ARRAY` or the value 1 for a `BAG`, `LIST` or `SET`.

  The `..INDEX` functions are useful for iterating over aggregates. For example, if `lagg` is a list of integer, then all the elements can be printed out as a comma-seperated list enclosed in parentheses by:

  ```
  writeln;
  write('lagg = (');
  REPEAT i := LOINDEX(lagg) TO HIINDEX(lagg);
    IF (i = HIINDEX(lagg)) THEN write(lagg[i]:1);
    ELSE write(lagg[i]:1, ', ');
    END_IF;
  END_REPEAT;
  writeln(')');
  ```

- `NVL (V:GENERIC; SUBS:GENERIC) : GENERIC;`
  If the argument `V` exists then it is returned, otherwise the argument `SUBS` is returned. Both arguments must be of the same type.

- `ODD (V:INTEGER) : LOGICAL;`
  Returns `TRUE` or `FALSE` depending on whether or not its argument is odd or even.

- `REXPR (V:STRING; E:STRING) : LOGICAL;`
  This function tests whether the `V` string parameter matches a regular expression `E`. `REXPR` returns `TRUE` if there is a match, `FALSE` if there is not a match, or `UNKNOWN` if the regular expression is ill-formed.

  In the regular expression, most characters stand for themselves, but `\` can be used to escape any of the meta-characters.

  - The meta-characters `(` and `)` are used for grouping sub-expressions.
  - `|` between expressions means one or the other.
  - `+` following an expression means match one or more times.
  - `*` following an expression means match zero or more times.
  - `?` following an expression means match zero or one times.
  - `[...]` is an expression indicating that any of the enclosed characters are acceptable.
  - `[^...]` is an expression indicating that any characters except those enclosed are acceptable.
  - Within a bracket expression a range of characters can be specified by providing the first and last with a seperating hyphen. For instance, `[a-zA-Z]` will match any alphabetic character.

  Some examples:

  - `[a-zA-Z]+` match one or more letters.
  - `[0-9]+.[0-9]+([eE][\-\+]?[0-9]+)?` match a floating point number (e.g., `1.23e-27` or `0.987`)
  - `[a-zA-Z][0-9a-zA-Z_]*` match an Express-A variable.
  - `[^0-9a-zA-Z]` match anything except letters or digits.
  - `(I|i)(F|f)` case insensitive match for the word `IF`.

- `ROUND (V:NUMBER) : INTEGER;`
  Returns the nearest integer to its argument value.

- `SIN (V:NUMBER) : REAL;`
  Returns the sine of an an angle specified in radians.

- `SIZEOF (V:AGGREGATE OF GENERIC) : INTEGER;`
  `SIZEOF` returns the number of elements in its argument. When `V` is an `ARRAY` this is the declared number of elements. When `V` is a `BAG`, `LIST` or `SET` this is the actual number of elements.

- `SQRT (V:NUMBER) : REAL;`
  Returns the square root of its argument.
- `TAN (V:NUMBER) : REAL;`
  Returns the tangent of an an angle specified in radians.
- `TRUNC (V:NUMBER) : INTEGER;`
  Chops off any decimal part of its argument, returning the corresponding integer value.

### 4.6 Source level debugger

The EXPRESS-A interpreter includes a source level debugger for use when your code appears to be misbehaving. When in operation the debugger will prompt for a command to be entered. It understands the following commands.

- `<return>` Continue processing.
- `break <number>` Place a breakpoint at the statement on line `<number>`.
- `break` Print the line numbers of all the breakpoints.
- `unbreak <number>` Remove the breakpoint from line `<number>`.
- `unbreak` Remove all breakpoints.
- `trace` Turn on statement tracing.
- `untrace` Turn off statement tracing.
- `entry` Turn on tracing of entry to procedures and functions.
- `unentry` Turn off entry tracing.
- `exit` Turn on tracing of exits from procedures and functions.
- `unexit` Turn off exit tracing.
- `traceall` Turn on all tracing.
- `untraceall` Turn off all tracing.
- `stack` Turn on display of the runtime stack accesses.
- `unstack` Turn off stack display.
- `step` Turn on single-stepping.
- `unstep` Turn off single stepping.
- `fetch <variable>` Print data fetches for `<variable>`.

- store <variable> Print data stores for <variable>.
- watch <variable> Print both data fetches and stores for <variable>.
- watch Print the names of all variables being watched.
- unwatch <variable> Remove the watch from <variable>.
- unwatch Remove all watches.
- show <expression> Print the value of the Express-A expresion <expression>. The variables in the expression must have been declared in the Express-A code. For example:

  ```
  show (23.0 + LOG(num))/(PI*r**2)
  ```

- assign <variable := expression> Assign the value of <expression> to the Express-A variable <variable>. For example:

  ```
  assign num := SIN(theta/300.0)
  ```

- where Print the current line number and the text of the next statement to be executed.
- kill Terminate the execution of the *LTX2X* program.

### 4.7 Example Express-A code

The following demonstrates most of the functionality of Express-A. Most of this is not particularly interesting, except possibly for the algorithms for calculating the date of Easter and for generating magic squares.

```
    c=        fun.ct  Test of CODE ltx2x

CODE_SETUP=
  ENTITY ent;
    attr1, attr3 : INTEGER;
    attr2 : STRING;
  END_ENTITY;

  TYPE joe = INTEGER;
  END_TYPE;

  TYPE colour = ENUMERATION OF (red, blue, green);
  END_TYPE;


PROCEDURE easter;
(* calculates the date of Easter for the present year
```

```
   The algorithm can be applied to any year between
   1900 and 2099 inclusive, but if so, then the year
   should be checked to ensure that it is within this range. *)
  LOCAL
    n, a, b, m, q, w : INTEGER;
    day : INTEGER;
    month : STRING;
  END_LOCAL;

  n := THE_YEAR - 1900;
  a := n MOD 19;
  b := (7*a + 1) DIV 19;
  m := (11*a + 4 - b) MOD 29;
  q := n DIV 4;
  w := (n + q + 31 - m) MOD 7;
  day := 25 - m - w;
  month := 'April';
  IF (day < 1) THEN
    month := 'March';
    day := day + 31;
  END_IF;
  writeln('In ', THE_YEAR:1, ' Easter is on ', month,  day:3);
END_PROCEDURE;


FUNCTION magic_square(order:INTEGER): LOGICAL;
(* calculates magic squares from order 1 through 15.
   The order must be an odd number. *)
  LOCAL
  row, col, num : INTEGER;
  sqr_order : INTEGER;
  magic : ARRAY[1:15] OF ARRAY[1:15] OF INTEGER;
  END_LOCAL;

  IF (order > 15) THEN  -- only squares up to order 15
    RETURN(FALSE);
  ELSE
    IF (order < 1) THEN -- squares have at least one entry
      RETURN(FALSE);
    ELSE
      IF (NOT ODD(order)) THEN -- squares are odd
        RETURN(FALSE);
      END_IF;
    END_IF;
  END_IF;

  sqr_order := order**2;
  row := 1;
```

```
  col := (order + 1) DIV 2;
  REPEAT num := 1 TO sqr_order;
    magic[row][col] := num;
    IF ((num MOD order) <> 0) THEN
      IF (row = 1) THEN row := order; ELSE row := row - 1; END_IF;
      IF (col = order) THEN col := 1; ELSE col := col + 1; END_IF;
    ELSE
      IF (num <> sqr_order) THEN row := row + 1; END_IF;
    END_IF;
  END_REPEAT;

  writeln(Magic square of order ',order:2);
  REPEAT row := 1 TO order;
    REPEAT col := 1 TO order;
      write(magic[row][col]:4);
    END_REPEAT;
    writeln;
  END_REPEAT;
  writeln;

  RETURN(TRUE);
END_FUNCTION;

FUNCTION month(mnum:INTEGER) : STRING;
(* Given an integer representing the month in a year,
   returns the name of the month. *)
LOCAL
  str : STRING;
END_LOCAL;

  CASE mnum OF
    1 : str := 'January';
    2 : str := 'February';
    3 : str := 'March';
    4 : str := 'April';
    5 : str := 'May';
    6 : str := 'June';
    7 : str := 'July';
    8 : str := 'August';
    9 : str := 'September';
    10 : str := 'October';
    11 : str := 'November';
    12 : str := 'December';
    OTHERWISE : str := '';
  END_CASE;
RETURN(str);
END_FUNCTION;
```

```
LOCAL
 a : array[1:3] of integer;
 lagg : list [0:5] of integer;
 a23 : array[1:2] of array[1:3] of integer;
 i, n : integer;
 s1, s2 : string;
 b : logical;
 r1, r2 : real;
 nega : array[-3:-1] of integer;
 posa : array[3:5] of integer;
 j : joe;
 ex : ent;
END_LOCAL;

   -- start with a massive compound statement
BEGIN

 writeln;
 println;

 (* write today's date *)
 writeln('Today is ', THE_DAY:1, ' ', month(THE_MONTH), ' ', THE_YEAR:1);
 writeln;

 (* The user might be interested in Easter *)
 easter;
 writeln;

 (* Call some math functions *)
 r1 := PI/4;
 writeln('r1 = PI/4 (0.78539...)', r1);
 writeln('cos(r1) (0.70710...)', cos(r1));
 writeln('sin(r1) (0.70710...)', sin(r1));
 writeln('tan(r1) (1.0)', tan(r1));

 r1 := CONST_E;
 writeln('r1 = CONST_E (2.7182...)', r1);
 writeln('log(4.5) (1.50407...)', log(4.5));
 writeln('log2(8) (3.0)', log2(8));
 writeln('log10(10) (1.0)', log10(10));

 r2 := exp(10);
 writeln('exp(10) (2.203...e4)', r2);

 r2 := sqrt(121);
 writeln('sqrt(121) (11.0)', r2);

 (* populate and print some arrays *)
```

```
writeln;
posa[3] := 10;
posa[4] := 20;
posa[5] := 30;
REPEAT i := LOINDEX(posa) TO HIINDEX(posa);
  writeln('posa[', i:1, '] = ', posa[i]);
END_REPEAT;

writeln;
nega[-3] := 1;
nega[-2] := 2;
nega[-1] := 3;
REPEAT i := LOINDEX(nega) TO HIINDEX(nega);
  writeln('nega[', i:1, '] = ', nega[i]);
END_REPEAT;

(* Do some things with a list *)

    -- check the initial size (should be empty)
i := SIZEOF(lagg);
writeln('no. of els in lagg = ', i);

    -- insert elements at the front
INSERT(lagg, 10, 0);
i := SIZEOF(lagg);
writeln('no. of els in lagg = ', i);
INSERT(lagg, 20, 0);
writeln('no. of els in lagg = ', SIZEOF(lagg));

   -- print some of the elements
i := lagg[1];
writeln('first in lagg = ', i);
writeln('lagg[2] = ', lagg[2]);

    -- check if a value in in the list
b := 10 IN lagg;
writeln(b);           -- should be TRUE
b := 30 IN lagg;
writeln(b);           -- should be FALSE

    -- write all the elements
REPEAT i := LOINDEX(lagg) TO HIINDEX(lagg);
  writeln('lagg[', i:1, '] = ', lagg[i]);
  println('lagg[', i:1, '] = ', lagg[i]);
END_REPEAT;

(* see what happens with an indeterminate value *)
b := FALSE;
```

```
  b := ?;
  writeln(b);
  println(b);
  (* Some more attempts with indeterminate *)
  i := 2;
  n := 3*i;
  writeln(i, n);    -- should be 2 6
  n := 3*?;
  writeln(i, n);    -- should be 2 ?
  i := ?;
  n := 3*i;
  writeln(i, n);    -- should be ? ?

END;   -- end of compound statement
       -- but we can have individual statements


  (* Try to provide some excitement by making a magic square *)
  writeln;
  write('Enter an odd number between 1 and 15: ');
  readln(n);
  IF NOT magic_square(n) THEN
    writeln('I did not like your number which was ', n:1);
    writeln('If you get it right next time, something magic will happen.');
    write('Enter an odd number between 1 and 15: ');
    readln(n);
    magic_square(n);
  END_IF;


  (* Try a couple of REPEAT statements *)
  writeln('Test REPEAT (should print -2)');
  i := -2;
  REPEAT UNTIL i = 0;
    writeln(i);
    println(i);
    ESCAPE;
    i := i + 1;
  END_REPEAT;

  writeln('Test REPEAT (should print 3, 2, 1)');
  REPEAT i := 3 TO 1 BY -1;
    writeln(i);
  END_REPEAT;


  (* Try the LIKE operator *)
  writeln('Test LIKE');
```

```
writeln(('A' LIKE 'A'));             -- should be TRUE
writeln(('A' LIKE 'b'));             -- should be FALSE
writeln(('Page 407' LIKE '$###'));   -- should be TRUE
writeln(('Page 23' LIKE '$###'));    -- should be FALSE


(* Try the REXPR function *)
writeln('Test rexpr');
writeln(rexpr('A', 'A'));            -- should be TRUE
writeln(rexpr('A', 'b'));            -- should be FALSE
writeln(rexpr('Page 407', '[a-zA-Z]+\ [0-9]+')); -- should be TRUE
writeln(rexpr('Page 23', '[a-zA-Z]+\ [0-9]'));   -- should be FALSE


(* Try an ARRAY OF ARRAY *)
a23[1][1] := 11;
a23[1][2] := 12;
a23[1][3] := 13;
a23[2][1] := 21;
a23[2][2] := 22;
a23[2][3] := 23;

writeln('Test REPEAT (should be 1 1 11, 1 2 12, 1 3 13, 2 1 21, 2 2 22 etc)');
REPEAT n := 1 TO 2;
  REPEAT i := 1 TO 3;
      writeln(n, i, a23[n][i]);
  END_REPEAT;
END_REPEAT;


(* do some simple string operations *)
s1 := 'string';
writeln(s1);        -- should be string
s2 := s1[2:4];
writeln(s2);        -- should be tri
b := s1 <> s2;
writeln(b);         -- should be TRUE
writeln(s2 + s1);   -- should be tristring


(* Assign and print to a user-defined type *)
j := 33;
writeln(j*3);     -- should be 99

(* Do something with a variable of type ENTITY */
ex.attr1 := 33;
ex.attr2 := 'The attribute named attr2';
ex.attr3 := ex.attr1/3;
```

```
  writeln('ex.attr1 should be 33 and is: ', ex.attr1);
  writeln('ex.attr2 is: ', ex.attr2);
  writeln('ex.attr3 should be 11 and is: ', ex.attr3);

END_CODE
```

## 5 Specifying a SPECIAL_ command

This section gives some hints on how to specify a LaTeX command that requires some special processing. The faint-hearted should skip this. It is assumed that the implementor will have knowledge of LaTeX, C programming, and lex and YACC style lexer and parser generator systems.

There are two ways of defining `SPECIAL_` kinds of commands though neither is particularly simple. The easiest is by what is termed the *coding* method. This involves modifying the standard actions. The more complicated means is by the *grammar* method, which involves extending the production grammar and, typically, also coding new kinds of actions.

The process of specifying one of the `SPECIAL_` kinds of command actions is:

- Seriously question the need for the special command. One is only required if the standard actions cannot be coerced into serving the needs of the command processing and/or the command grammar is not supported by the *LTX2X* system.
- Design the required entry for the command table.
- Decide whether the coding or the grammar method is to be used for extending *LTX2X*.

  **Coding method** Modify the actions in `l2xusrlb.c`, and possibly add new functions in `l2xusrlb.c` and `l2xusrlb.h`.

  **Grammar method** Extend the grammar in `l2x.y`. Typically it be necessary to add new functions to the user-defined library `l2xusrlb` as well.
- Compile the modified *LTX2X* system. A `make` file for this is given in Appendix E.
- Test the extensions and debug the program.

The `l2xlib` has many functions that may be of use in this process. Some of these are indicated below.

- `char *strsave(char s[])` saves a string somewhere
- `void myprint(char s[])` writes a string to the output medium. Its particular action is controlled by the `set_print` and `reset_print` functions, as well as the `-p` option.
- `void verbatim_print(char s[])` like `myprint`, writes a string to the output medium. Its actions are controlled by the current print mode, and newlines are obeyed (i.e., it ignores any pretty-printing option).

- `void yyerror(char s[])` used by the lexer and parser to print an error message string.
- `void warning(char s[])` writes a warning message string.
- `void do_newline()` used within the lexer to set internal variables whenever a newline is encountered in the input.
- `void initialise_sysbuf()` initialises the system supplied string buffer.
- `void print_sysbuf()` writes the content of the system string buffer to the output file.
- `void copy_sysbuf(char s[])` copies the contents of the system string buffer to the user-supplied string. It is the user's responsibility to ensure that the string is big enough.
- `void set_print(PSTRWC pswitch)` controls the action of `myprint`, `print_newline` and `verbatim_print`. If the input argument is `p_default_print` then the print functions should write to the output file; this is the default behavior. If the argument is `p_no_print`, then no writing occurs. If the argument is `p_print_to_sysbuf`, then the print functions write to the system string buffer.
- `void reset_print()` resets the behavior of the print functions. This function should always be used after a call to `set_print`.
- `int lookup_entry(char s[], int kind)` returns the location within the command table of the command name given in `s` of the command type given in `kind`. If `kind` is `DONT_CARE` then the position of the first occurrence of `s` is returned.
- `void get_env_name(char s[])` extracts the name of a LaTeX environment from `s`, which is assumed to have the form `\something { environment }`. The name of the environment is put into the global string `env_name`.
- `PSENTRY get_mode_sym(int loc)` returns a pointer to the symbol table entry at position `loc` in the command table for the current mode.
- `int command_type(int loc)` returns the system defined kind of command at location `loc` in the command table.
- `int get_user_type(int loc)` returns the user input (`TYPE=`) kind of command at location `loc` in the command table.
- `PSTRWC get_t(PSENTRY loc)` returns a pointer to the `START_TAG=` specification for symbol entry `loc`. There are similar functions for other tagging specifications.

- `PSTRWC get_tag_t(PSENTRY loc, int n)` returns a pointer to the `START_TAG_n` specification for the `n`'th argument for symbol entry `loc`. There are similar functions for other argument tagging specifications.

- `PSTRWC get_param_print(PSENTRY loc, int n)` returns a pointer to the print control specification for the `n`'th required argument for symbol entry `loc`. There are similar functions for other print controls.

- `int get_level(PSENTRY loc)` returns the `SECTIONING_LEVEL=` value for symbol entry `loc`.

The process of specifying a `SPECIAL_` is best described via an example.

## 5.1 Example

Assume that there is a 'non-standard' LaTeX command which has one required argument. When this command is processed by LaTeX its effect is to start a new section in a document entitled *Normative References*. Some boilerplate text is then typeset (specified within the definition of the command). This boilerplate includes two instances of the text from the argument of the command. Finally, a `description` list environment is started.

In LaTeX terms, this command could have been defined as:

```
\newcommand{\XXspecial}[1]{\section{Normative References}
          Some boilerplate text with #1
          in the middle. Now there is
          some more boilerplate with #1
          in the middle of it.
          \begin{description} }
```

For the purposes of the example, it is desired to replace the occurrence of the `\XXspecial` command by the 'normal' section heading for the output tagged style, and also print out the boilerplate text including the argument text in the right places. The start of the list environment has also to be taken into account. The `\item` optional argument text is to be enclosed in parentheses, with a dash before the main text. These requirements are not something that can be currently accomplished with the standard *LTX2X* system.

To make the requirements more concrete, if the input LaTeX source includes:

```
....
\XXspecial{REQ PARAM TEXT}

\item[Ref 1] Text 1.
\item[Ref 2] Text 2.
\end{description}
....
```

then the desired output is to look like:

```
....

</div.1>

<div.1>
<heading>Normative References</heading>

Some boilerplate text with REQ PARAM TEXT
in the middle. Now there is
some more boilerplate with REQ PARAM TEXT
in the middle of it.

    (Ref 1) -- Text 1.
    (Ref 2) -- Text 2.
....
```

Now, let's write a specification for the command table, which we will do in pieces, starting with the sectioning tags. In this tagging style, end tags for sections take the form `</div.1>`, and start tags the form `<div.1>`. The titles of sections are enclosed between `<heading>` and `</heading>` tags. We also want some newlines in the output to set things off. If `?` is used as the escape character, then we can specify for the sectioning tagging:

```
SECTIONING_LEVEL= SECT
START_TAG= "?n?n<div.1>?n"
  STRING: "<heading>Normative References</heading>?n"
END_TAG= "?n</div.1>"
```

There is one required argument and no optional arguments, so we need:

```
REQPARAMS= 1
```

The LaTeX command also starts off a `description` environment, so we have to set the tags for the `\item` commands that will follow. This is done by:

```
START_ITEM= "?n"
START_ITEM_PARAM= "    ("
END_ITEM_PARAM= ") -- "
```

Most of the work is now completed though we still have to give the command name, decide what sort of `SPECIAL_` it will be and set the `SPECIAL_TOKEN` value. None of the provided `SPECIAL_` types exactly fit this entry as it is a mixture of sectioning and list environment, so we will just call it a `SPECIAL_COMMAND` type. To summarize, the effective state of the command table entry is:

```
TYPE= SPECIAL_COMMAND
C=      NAME= to be specified
C=      SPECIAL_TOKEN= to be specified
  SECTIONING_LEVEL= SECT
  START_TAG= "?n?n<div.1>?n"
    STRING: "<heading>Normative References</heading>?n"
```

```
  END_TAG= "?n</div.1>"
  REQPARAMS= 1
  START_ITEM= "?n"
  START_ITEM_PARAM= "    ("
  END_ITEM_PARAM= ") -- "
END_TYPE
```

For pedagogical purposes, this special will be implemented using both the grammar and code methods, and the command names used will be `\GRAMMspecial` and `\CODEspecial` respectively.

### 5.1.1 Grammar method implementation

The command name for this implementation will be `\GRAMMspecial`.

The grammar method requires changes to the grammar specified in `l2x.y`.

1. A new token has to be defined, call it `GRAMMSPECIAL`, in the first part of the file. There is a slot for this under the comment `/* specials */`. An integer number, greater than or equal to 10,000 (ten thousand) and less than 32,768 ($2^{15}$), has to be associated with this token.[7] Further, this number must not be the same as any other number associated with any other token. Let us use the maximum number 32,767. The relevant portion of `l2x.y` will look like

```
                                        /* specials */
%token <pos> /* other specials here */
%token <pos> GRAMMSPECIAL 32767
                                        /* precedences */
```

   This number is used for communication within the *LTX2X* system, and is the number set as the value of the `SPECIAL_TOKEN` in the command table. We can now finalize the command table entry as:

```
TYPE= SPECIAL_COMMAND
NAME= \GRAMMspecial
SPECIAL_TOKEN= 32767
  SECTIONING_LEVEL= SECT
  START_TAG= "?n?n<div.1>?n"
    STRING: "<heading>Normative References</heading>?n"
  END_TAG= "?n</div.1>"
  REQPARAMS= 1
  START_ITEM= "?n"
  START_ITEM_PARAM= "    ("
  END_ITEM_PARAM= ") -- "
END_TYPE
```

[7]The upper limit of $(2^{15} - 1)$ is set by the bison processor.

2. A new production, or productions, has to be added to the grammar. There is a place for this at the end of the rules section in the file, under the predefined production `l2xSpecials`. Let us call our new production `GrammSpecial`, and add it as:

   ```
   l2xSpecials: ASpecial
        | AnotherSpecial
        | GrammSpecial
        ;
   ```

   where the `ASpecial` and `AnotherSpecial` are pre-existing specials.

3. The production now has to be defined, specifying the expected syntax and required actions. This looks like:

   ```
   GrammSpecial: GRAMMSPECIAL
       {
          start_section($1);
          myprint(get_t($1));
          myprint(get_tag_t($1,1));
          initialise_sysbuf();
          set_print(p_print_to_sysbuf);
       }
       ReqParam
       {
         initialise_string(grammbuf);
         copy_sysbuf(grammbuf);
         reset_print();
         prwboiler1();
         print_sysbuf();
         prwboiler2();
         myprint(grammbuf);
         prwboiler3();
         start_list($1);
       }
       ;
   ```

   The actions are enclosed in braces and are defined in terms of C code.

   Once the parser has been given the `GRAMMSPECIAL` token from the lexer, it will attempt to perform the actions within the first set of braces. The first of these, `start_section($1)`, is the *LTX2X* action for starting a sectioning command. Basically, this deals with any closing of prior sections of the document and remembering the closing tag for this section. The next two actions print the start tags for the command and its required argument, taking the strings from the command table. `initialise_sysbuf()` initializes the system string buffer ready for new input. Then the print control is set so that any output will be directed into the system string buffer rather than the output file. This finishes the first set of actions.

The production grammar for the required argument comes next. If this is incorrect, the parser automatically gives an (uninformative) error message. Otherwise, the last set of actions are done. At this point, the text of the required argument will be contained in the system string buffer. This is then copied to a temporary buffer `grammbuf`, that we have yet to define, by calling `copy_sysbuf(grammbuf)` having first made sure that this buffer has been cleared of any previous contents (the `initialise_string(grammbuf)` action). The printing control must now be reset (`reset_print()`), or things might get corrupted later. A function `prwboiler1()` is called to print the first part of the boilerplate text, followed by printing the contents of the system buffer by the action `print_sysbuf()` (remember that this should contain the text of the required argument). The second part of the boilerplate is written by the function `prwboiler2()`. Just for pedagogical purposes, the required argument text is written out using the text stored in the temporary buffer (`myprint(grammbuf)`) rather than from the system buffer. The penultimate action is the printing of the last piece of boilerplate.

The final action — `start_list($1)` — is the standard *LTX2X* action at the start of a list environment. This remembers the various tags for the list items to follow.

A character buffer, `grammbuf`, is now defined in the initial section of the `l2x.y` file, as:

```
char grammbuf[80];
```

which is intended to be large enough to hold the text of the required argument of the command.

This completes the changes to the grammar file.

4. The three functions called out in the above actions for printing the boilerplate are coded and placed in the user library file `l2xusrlb.c` and are also added to `l2xusrlb.h`. Here is the relevant code as it would appear in `l2xusrlb.c`.

```
             /* demonstration string definition */

STRING boiler_string_3 = "\nin the middle of it.\n\n";

                   /* demonstration functions */

/* PRWBOILER1 print some demonstration boilerplate */
void prwboiler1()
{
  myprint("\nSome boilerplate text with ");
}                                 /* end PRWBOILER1 */

/* PRWBOILER2 print some demonstration boilerplate */
void prwboiler2()
```

```
{
  myprint("\nin the middle. Now there is\n");
  myprint("some more boilerplate with ");
}                                  /* end PRWBOILER2 */

/* PRWBOILER3 yet more demonstration boilerplate */
void prwboiler3()
{
  myprint(boiler_string_3);
}                                  /* end PRWBOILER1 */
```

5. The system is recompiled, using `make`, and tested on some example LaTeX files.

### 5.1.2 Code method implementation

This method 'merely' requires extending the standard actions to account for the new requirements. First, however, we must complete the definition of the command table entry. We will call the new command `\CODEspecial`. Also a unique value has to be assigned to the `SPECIAL_TOKEN`. This must have a value greater than or equal to 50,000 (fifty thousand). We will use a value 59,999. Later this value is used within the action code to identify the special. The command table entry is thus:

```
TYPE= SPECIAL_COMMAND
NAME= \CODEspecial
SPECIAL_TOKEN= 59999
  SECTIONING_LEVEL= SECT
  START_TAG= "?n?n<div.1>?n"
    STRING: "<heading>Normative References</heading>?n"
  END_TAG= "?n</div.1>"
  REQPARAMS= 1
  START_ITEM= "?n"
  START_ITEM_PARAM= "    ("
  END_ITEM_PARAM= ") -- "
END_TYPE
```

which only differs from that for the grammar implemented special in the `SPECIAL_TOKEN=` and the `NAME=` values.

Before proceeding further, some explanation of the internals of the *LTX2X* system is in order.

**Command table entry** Internally, an array of C `struct`s is used for storing the data corresponding to the command table. The `struct` is fully defined in file `l2xcom.h`, and type `PSENTRY` is a pointer to an instance of the `struct`. There is an entry in the internal command table for each command. Where a command specification is mode-dependent, then the entries for this are stored as a list (the command table array is actually an array of lists of command specifications, one list per command).

For the purposes at hand, only a few of the elements are of concern; these are `kind`, `parse_kind` and `special_token`. The element `kind` contains an identifier of the `TYPE=` value; that is, the type of the command as specified by the user. The element `special_token` contains the `SPECIAL_TOKEN=` value. The `parse_kind` element contains an identifier of the type of command as assigned internally by *LTX2X*. This last identifier is generated by the table processing code in `l2xlib.c` and corresponds to one of the token values acceptable to the parser.

**The lexer** The lexer reads the source LaTeX file, looking for LaTeX commands (essentially anything starting with a backslash). Each time it finds a command it looks it up in the command table array, and sends its parser token value (the `parse_kind` value) and the command table array position to the parser.

**The parser** Given a token from the lexer, the parser finds the appropriate grammar production and performs the specified actions. It is able to access command information through having the command table position.

**The grammar** The grammar used for the LaTeX general commands (and environments) is actually very simple — the complexity is reserved for the lexer and the actions. At the grammar level, no distinction is made between a command and an environment. There are a total of 19 different command types (tokens), which fall into a smaller number of groups.

1. A command with no arguments.
2. A command with just a single optional argument.
3. Commands with a final optional argument and between 1 and 8 required arguments.
4. Commands with between 1 and 8 required arguments. In this case it is always assumed that there might be an initial optional argument.
5. A command with 9 required arguments.

Note that this partitioning is based solely on the number of required arguments and the position (if any is declared) of an optional argument.

For example, if a command/environment is specified in the command table as having one required argument and no optional arguments, then this will be treated as a command with possibly an initial optional argument and one required argument. The grammar for this is:

```
l2xComm1: COMMAND_1
        {
          start_with_opt($1);
        }
        OptParam
        {
```

```
    action_opt_first($1);
  }
  ReqParam
  {
    action_last_p($1,1);
  }
  ;
```

where the words in all upper case are grammar tokens, and words in mixed case are other grammar productions. The actions are enclosed between braces. The `$1` is the position of the command in the command table array.

**Actions** The standard actions are contained in file `l2xacts.c`. Within the code for each standard action, provision is made for calling action code for specials. All the functions have the same general structure. Here, for instance, is the code for the standard action that is called between the start of a command and a first optional argument.

```
/* START_WITH_OPT start action for command with optional param */
void start_with_opt(pos)
int pos;                        /* position of command in table */
{
  int user_kind;                /* user-specified command type */

  user_kind = get_user_type(pos);

  switch(user_kind) {
  case TEX_CHAR:                /* the general, non-specials */
  case CHAR_COMMAND:
  case COMMAND:
  case BEGIN_ENV:
  case END_ENV:
  case BEGIN_LIST_ENV:
  case END_LIST_ENV:
  case SECTIONING:
    start_it(pos);                /* command start action */
    default_start_with_opt(pos);  /* start optional param */
    break;
  case SPECIAL:                   /* the specials */
  case SPECIAL_BEGIN_ENV:
  case SPECIAL_END_ENV:
  case SPECIAL_BEGIN_LIST:
  case SPECIAL_END_LIST:
  case SPECIAL_COMMAND:
  case SPECIAL_SECTIONING:
    special_start_with_opt(pos);
    break;
  default:                        /* should not be here! */
    warning("(start_with_opt) Unrecognized command type");
```

```
    break;
  } /* end switch on user_kind */
}                                 /* end START_WITH_OPT */
```

The special actions code is in file `l2xusrlib.c`. The code for these all follow the same general pattern. For example, here is the code implementing the special action between the start of a command and a first optional argument.

```
/* SPECIAL_START_WITH_OPT special start for command with opt param */
void special_start_with_opt(pos)
int pos;                          /* command position in table */
{
  int special_kind;               /* user-specified special token */

  special_kind = get_special_token(pos);

  switch(special_kind) {
    /* additional cases for specials added here */

    /* end of cases for specials */
  default:                       /* should not be here! */
    warning("(special_start_with_opt) Unrecognized SPECIAL");
    tdebug_str_int("SPECIAL_TOKEN =",special_kind);
    break;
  }  /* end of switch on user_kind */
}                           /* end SPECIAL_START_WITH_OPT */
```

Note that the code as provided just issues a warning message.

With this background, we will now go on with the example.

1. Decide on how the *LTX2X* system will translate your command/environment description into its internal grammar command type. In this case it will translate into a command with possibly an initial optional argument and one required argument.

2. Examine the grammar for the command and hence determine which standard actions will be called. In this case there are three of these, namely `start_with_opt`, `action_opt_first` and `action_last_p`. These are the actions that might require modification.

3. Determine what actions are required for your special. Conceptually replace the standard actions in the grammar by your actions. Then determine how these should be incorporated into the special action code in `l2xusrlb`. In plain language, the grammar and conceptual actions for the example are:

```
l2xComm1: COMMAND_1
        {
```

```
    start of as a sectioning command
    ignore the optional argument as there isn't one
  }
  OptParam
  {
    finish processing the non-existent optional
    get ready to store the argument text in a buffer
  }
  ReqParam
  {
    print the boilerplate and argument text
    start the description list
  }
  ;
```

Note that this is essentially the same as we did for the grammar implementation for `\GRAMMspecial`, except that there is the additional optional argument to be dealt with.

4. Modify the requisite special action code.

   In the example, three standard actions have to be modified. Here is the modification to `special_start_with_opt`:

```
/* SPECIAL_START_WITH_OPT special start for command with opt param */
void special_start_with_opt(pos)
int pos;                            /* command position in table */
{
  int special_kind;                 /* user-specified special token */

  special_kind = get_special_token(pos);

  switch(special_kind) {
    /* additional cases for specials added here */
  case 59999:                     /* example coded special */
    codespecial_start(pos);
    default_start_with_opt(pos);
    break;

    /* end of cases for specials */
  default:                         /* should not be here! */
    warning("(special_start_with_opt) Unrecognized SPECIAL");
    tdebug_str_int("SPECIAL_TOKEN =",special_kind);
    break;
  }  /* end of switch on user_kind */
}                             /* end SPECIAL_START_WITH_OPT */
```

   The addition is done by adding a new `case 59999:` together with appropriate code. The number 59999 is that corresponding to the value for `SPECIAL_TOKEN=` in the command table specification of \CODEspecial. The function codespecial_start is to be written, while default_start_with_opt is an *LTX2X* defined function which initiates processing of an initial optional argument.

Similarly, here is the modification to special_action_opt_first:

```
     /* additional cases for specials added here */
      case 59999:             /* example coded special */
        default_end_start_opt(pos);
        codespecial_p1(pos);
        break;

     /* end of cases for specials */
```

where codespecial_p1 is to be written and default_end_start_opt is the standard *LTX2X* action at the end of an initial optional argument.

Finally, here is the modification to special_action_last_p:

```
/* SPECIAL_ACTION_LAST_P action after last req argument */
void special_action_last_p(pos,p)
int pos;                      /* position of command in table */
int p;                        /* number of last argument */
{
  int special_kind;                /* user-specified special token */

  special_kind = get_special_token(pos);

  switch(special_kind) {
     /* additional cases for specials added here */
      case 59999:             /* example coded special */
        if (p == 1) {         /* has only one req param */
          codespecial_end(pos);
        }
        break;

     /* end of cases for specials */
    /* stuff deleted to save space */
```

5. Code the functions for the new actions.

   Code for these functions should be put into file l2xusrlb.c and file l2xusrlb.h modified accordingly.

   Here is the code for the three codespecial_ functions.

```
char codebuf[80];                /* a string buffer */

/* CODESPECIAL_START actions for start of CODEspecial command */
```

```
void codespecial_start(pos)
int pos;                        /* command table position */
{
  start_section(pos);           /* do start of sectioning */
  myprint(get_t(pos));          /* print start tag */
}                               /* end CODESPECIAL_START */

/* CODESPECIAL_P1 actions at start of CODEspecial param 1 */
void codespecial_p1(pos)
int pos;                        /* command table position */
{
  myprint(get_tag_t(pos,1));    /* print 1st param start tag */
  initialise_sysbuf();          /* clear system string buffer */
  set_print(p_print_to_sysbuf); /* put arg text into sys buffer */
}                               /* end CODESPECIAL_P1 */

/* CODESPECIAL_END actions at end of CODEspecial command */
void codespecial_end(pos)
int pos;                        /* command table position */
{
  initialise_string(codebuf);   /* clear this string buffer */
  copy_sysbuf(codebuf);         /* copy sys buffer into codebuf */
  reset_print();                /* normal printing */
  prwboiler1();                 /* print some boilerplate */
  print_sysbuf();               /* print system buffer */
  prwboiler2();                 /* print more boilerplate */
  myprint(codebuf);             /* print codebuff */
  prwboiler3();                 /* print yet more boilerplate */
  start_list(pos);              /* start a list environment */
}                               /* end CODESPECIAL_END */
```

Note that these actions are almost identical to those that were used within the grammar when implementing the `\GRAMMspecial` command.

6. Compile the modified system and test it on example LaTeX files.

## 5.2 Notes

1. Installation of a `SPECIAL_` can be limited to making changes to the parser (file `l2x.y`) and/or the user library (files `l2xusrlb.c` and `l2xusrlb.h`). It should not be necessary to touch any other part of the system.

2. Changes to `l2x.y` will necessitate executing the parser generator on this file and system compilation. Changes to the other files will only necessitate compilation.

3. Always use the `myprint`, `verbatim_print` or `print_sysbuf` functions for printing because they incorporate the print control capability.

4. The printing in the above example is trivial. However, it is good practice to define printed output separately from the parser. It makes for easier maintenance. If, for example the boilerplate above was several thousand characters, it might have been an idea to store the text in a file, or files, and then have the boilerplate printing functions read from the file(s). If the text is in a state of flux this could be a good design decision in any case, as changing the text would only involve modifying the text file(s) and avoid recompilation of *LTX2X*.

### 5.2.1 An updated method

The above descriptions of installing a `SPECIAL_` command were written for the original release of the *LTX2X* system, which did not have the input and output specification facilities currently available within a command table. Below is given a possible command table entry using these facilities.

```
TYPE= SPECIAL_COMMAND
NAME= \THIRDspecial
  C= SPECIAL_TOKEN=  set the appropriate number
  SECTIONING_LEVEL= SECT
  START_TAG= "?n?n<div.1>?n"
    STRING: "<heading>Normative References</heading>?n"
    RESET_SYSBUF:
  END_TAG= "?n</div.1>"
  REQPARAMS= 1
  PRINT_P1= TO_SYSBUF
  END_TAG_1=
    STRING: "?nSome boilerplate text with "
    SOURCE: SYSBUF
    STRING: "?nin the middle. Now there is?n"
    STRING: "some more boilerplate with "
    SOURCE: SYSBUF
    STRING: "?nin the middle of it.?n"
  START_ITEM= "?n"
  START_ITEM_PARAM= "    ("
  END_ITEM_PARAM= ") -- "
END_TYPE
```

The actual implementation of this as either a grammar special or a code special is left as an exercise for the reader. Basically it involves the deletion of the specific print action and buffer code because this is now handled automatically via the command table specification.

# A Example command table file for de-TeXing

This appendix provides the skeleton of a command table file that could be used for de-TeXing a LaTeX document.

```
C=  detex.ct command table file for ltx2x to deTeX source

C=   -----------------------------------escape sequences

C= don't use default here as it may clash with command name output
ESCAPE_CHAR= ?
C=       keep tye default vaues for the rest

C=   ----------------------------------- the built in commands
TYPE= BEGIN_DOCUMENT
END_TYPE

TYPE= END_DOCUMENT
END_TYPE

TYPE= BEGIN_VERB
END_TYPE

TYPE= END_VERB
END_TYPE

TYPE= BEGIN_VERBATIM
  START_TAG= "?n"
END_TYPE

TYPE= END_VERBATIM
  START_TAG= "?n"
END_TYPE

TYPE= BEGIN_DOLLAR
END_TYPE

TYPE= END_DOLLAR
END_TYPE

TYPE= SLASH_SPACE
  START_TAG= " "
END_TYPE

TYPE= OTHER_COMMAND
  PRINT_CONTROL= NO_PRINT
END_TYPE
```

```
TYPE= OTHER_BEGIN
  PRINT_CONTROL= NO_PRINT
END_TYPE

TYPE= OTHER_END
  PRINT_CONTROL= NO_PRINT
END_TYPE

C=       throw away naked braces
TYPE= LBRACE
END_TYPE

TYPE= RBRACE
END_TYPE

C=  Pretty printing will probably be applied. Indent start of paragraphs
TYPE= PARAGRAPH
  START_TAG= "?n?n    "
END_TYPE

C= -------------------------------------(La)TeX special characters

C= hash (for use in \def s )
TYPE= TEX_CHAR
NAME= #
END_TYPE

C= ampersand (tabular column delimiter, replace by some spaces)
TYPE= TEX_CHAR
NAME= &
  START_TAG= "   "
END_TYPE

C= twiddle (unbreakable space)
TYPE= TEX_CHAR
NAME= ~
  START_TAG= " "
END_TYPE

C= underscore (math subscript)
TYPE= TEX_CHAR
NAME= _
  START_TAG= "_"
END_TYPE

C= caret (math superscript)
TYPE= TEX_CHAR
NAME= ^
```

```
  START_TAG= "^"
END_TYPE

C= at
TYPE= TEX_CHAR
NAME= @
  START_TAG= "@"
END_TYPE

C= ------------------------- default single character commands
C=         (replace by appropriate character)

C= LaTeX start a new line
TYPE= CHAR_COMMAND
NAME= \\
  START_TAG= "?n"
END_TYPE

C= small space
TYPE= CHAR_COMMAND
  NAME= \,
START_TAG= " "
END_TYPE

C= end of sentence space
TYPE= CHAR_COMMAND
NAME= \@
  START_TAG= " "
END_TYPE

C= hash
TYPE= CHAR_COMMAND
NAME= \#
  START_TAG= "#"
END_TYPE

C= dollar
TYPE= CHAR_COMMAND
NAME= \$
  START_TAG= "$"
END_TYPE

C= ampersand
TYPE= CHAR_COMMAND
NAME= \&
  START_TAG= "&"
END_TYPE
```

```
C= underscore
TYPE= CHAR_COMMAND
NAME= \_
  START_TAG= "_"
END_TYPE

C= percent
TYPE= CHAR_COMMAND
NAME= \%
  START_TAG= "%"
END_TYPE

C= left brace
TYPE= CHAR_COMMAND
NAME= \{
  START_TAG= "{"
END_TYPE

C= right brace
TYPE= CHAR_COMMAND
NAME= \}
  START_TAG= "}"
END_TYPE

C= optional hyphenation
TYPE= CHAR_COMMAND
NAME= \-
  START_TAG= ""
END_TYPE

C= ----------------------------- General LaTeX

TYPE= COMMAND
NAME= \caption
  START_TAG= "?n    CAPTION: "
  OPT_PARAM= FIRST
  PRINT_OPT= NO_PRINT
  REQPARAMS= 1
END_TYPE

TYPE= BEGIN_LIST_ENV
NAME= itemize
  START_TAG= "?n"
  START_ITEM= "?n   o "
END_TYPE

TYPE= END_LIST_ENV
NAME= itemize
```

```
  START_TAG= "?n"
END_TYPE

TYPE= BEGIN_LIST_ENV
NAME= enumerate
  START_TAG= "?n"
  START_ITEM= "?n   -- "
END_TYPE

TYPE= END_LIST_ENV
NAME= enumerate
  START_TAG= "?n"
END_TYPE

TYPE= BEGIN_LIST_ENV
NAME= description
  START_TAG= "?n"
  START_ITEM= "?n    "
  END_ITEM_PARAM= " : "
END_TYPE

TYPE= END_LIST_ENV
  NAME= description
START_TAG= "?n"
END_TYPE

C=       replace \footnote with parenthesized text
TYPE= COMMAND
NAME= \footnote
  START_TAG= " ("
  END_TAG= ") "
  OPT_PARAM= FIRST
  PRINT_OPT= NO_PRINT
  REQPARAMS= 1
END_TYPE

C=          ----------------------- sectioning (keep headers only)

C=           repeat for all the other sectioning commands
TYPE= SECTIONING
NAME= \section
  SECTIONING_LEVEL= SECT
  START_TAG= "?n?n"
  OPT_PARAM= FIRST
  PRINT_OPT= NO_PRINT
  REQPARAMS= 1
  END_TAG_1= "?n?n"
END_TYPE
```

```
C=        repeat for all the other starred sectioning commands
TYPE= SECTIONING
NAME= \section*
  SECTIONING_LEVEL= SECT
  START_TAG= "?n?n"
  REQPARAMS= 1
  END_TAG_1= "?n?n"
END_TYPE

C=        and whatever else is interesting
END_CTFILE=
```

# B LaTeX to HTML translation

The command table file `l2h.ct` contains a set of commands that enable simple LaTeX documents to be translated into HTML tagged documents for display using a World Wide Web browser. At a minumum this command table can be used for conversion of the LaTeX source of this manual. It can also handle some very simple mathematics but not pictures.[8] The specification for the HTML tags was taken from Musciano and Kennedy [5].

Generally speaking and subject to the above limitations, a LaTeX document can be translated to HTML without the document having been planned for this purpose, with one exception. The exception is that a new LaTeX command should be used in the document preamble. I have called this `\mltitle` and its purpose to to define the contents of the header for the HTML text. The definition of this command is:

```
\newcommand{\mltitle}[1]{}
```

That is, as far as LaTeX is concerned, the argument to the command is thrown away and is a non-event. As far as the `l2h.ct` command table is concerned the argument is the header title. As an example, this manual starts with:

```
...
\mltitle{LaTeX to X translator}
\begin{document}
\title{\lx: A \LaTeX{} to X Auto-tagger}
...
```

which gets converted into:

```
<html>
<head>
<title>LaTeX to X translator</title>
```

[8]HTML itself cannot handle pictures directly (i.e., there is no equivalent to the LaTeX `picture` environment), and can only handle simple mathematics.

```
</head>
<body>

<h1 align=center>
 LTX2X: A LaTeX to X Auto-tagger
</h1>
...
```

If the `\mltitle` command is not used, then the effect is to have an empty `<title>` in the `<head>` of the HTML document.

Several aspects of the design of `l2h.ct` in the context of the conversion of typical LaTeX documents have been discussed as examples in the body of the manual. However, there are some aspects specific to the translation of this document should be mentioned. These stem from the fact that HTML has no tags corresponding to to the LaTeX `\verb` command or `verbatim` environment which switch off the meanings of special characters.

HTML treats the characters `<`, `>`, & and # specially. Within a `<pre>...</pre>` the browser honours the line breaks but does not switch off the meanings of the special characters. In LaTeX, the `\verb` command switches off all special characters but prohibits any line breaking. The `verbatim` environment both honours line breaks and switches off all special characters. The difficulty with this particular document is that I want to show author-formatted HTML source, and that is not easily possible, unlike using the LaTeX `verbatim` environment for showing user-formatted LaTeX source.

The problem was solved through the use of two LaTeX environments. The first of these is `latexonly` which is used for input that is to be processed normally by LaTeX but which is to be totally ignored by *LTX2X*. The other environment is `htmlverbatim` which is used for input that is to be totally ignored by LaTeX but which is to be processed by *LTX2X* into an HTML `<pre>` environment.

A package file has been written which provides some addtional commands and environments.

```
% ltx2html.sty  --- Some useful commands and environments when using
%                   ltx2x to convert from LaTeX to HTML tagging.
%
% Author: Peter Wilson    August 1996
%
\ProvidesPackage{ltx2html}[1996/08/29 ltx2x HTMLing]
\RequirePackage{html}  % the package file for the Perl program
                       % latex2html

% The document title for the WWW browser.
% If used, must be placed in the preamble.
\newcommand{\mltitle}[1]{}

% argument is for processing by LaTeX only
\providecommand{\latex}[1]{#1}
```

```
% argument is for HTML processing only
\providecommand{\html}[1]{}

% print argument as an SGML/HTML start tag
\newcommand{\ST}[1]{\texttt{<#1>}}

% print argument as an SGML/HTML end tag
\newcommand{\ET}[1]{\texttt{</#1>}}

% print HTML special characters
\newcommand{\Amp}{\&}
\newcommand{\GT}{\texttt{>}}
\newcommand{\LT}{\texttt{<}}
\newcommand{\HASH}{\#}

% treat contents as a LaTeX comment but
% translate contents into an HTML "verbatim" environment
% Use as: \begin{htmlverbatim} ... \end{htmlverbatim}
\excludecomment{htmlverbatim}

\endinput
```

The command table entries for some of these are:

```
TYPE= COMMAND
NAME= \Amp
  START_TAG= "&amp;"
END_TYPE

TYPE= COMMAND
NAME= \GT
  START_TAG= "&gt;"
END_TYPE

TYPE= COMMAND
NAME= \LT
  START_TAG= "&lt;"
END_TYPE

TYPE= COMMAND
NAME= \HASH
  START_TAG= "&#035;"
END_TYPE

TYPE= COMMAND
NAME= \ST
  START_TAG= "&lt;"
  END_TAG= "&gt;"
```

```
  REQPARAMS= 1
END_TYPE

TYPE= COMMAND
NAME= \ET
  START_TAG= "&lt;/"
  END_TAG= "&gt;"
  REQPARAMS= 1
END_TYPE
```

Finally, as an example, this is how some of the prior example text could be written in the source of this document.

```
\begin{latexonly}
\begin{verbatim}
<html>
<head>
<title>LaTeX to X translator</title>
</head>
<body>

<h1 align=center>
 LTX2X: A LaTeX to X Auto-tagger
</h1>
...

\end{verbatim}
\end{latexonly}


\begin{htmlverbatim}
<html>
<head>
<title>LaTeX to X translator\ET{title}
\ET{head}
<body>

<h1 align=center>
 LTX2X: A LaTeX to X Auto-tagger
\ET{h1}
...
\end{htmlverbatim}
```

Reading the LaTeX source of this document will reveal some other details. Admittedly the problem was compounded by the fact that this document contains demonstrations of both LaTeX and HTML commands which will be processed through both LaTeX and HTML browsers, thus a modicum of care is required to appropriately process both sets of special characters.

## C Known limitations

*LTX2X* does not do everything that it might (and probably never will). The following are some of the things that it does not do.

- It does not understand the LaTeX `\input` or `\include` commands — it just reads the source file as given. It may be useful to pre-process the source file through a program that will automatically incorporate included files into a LaTeX root file [13].

- The `\newcommand` and friends do not readily fit into the command patterns that *LTX2X* can deal with. In particular, if it comes across a `\newcommand` specification for a command that is specified in the command table, interesting results might occur (for example, all the following output could be thrown away if the command takes any arguments).

  For instance, if the document and the command table contain:

  ```
  \newcommand{\lx}{LTX2X}
  ....
  The \lx\ program ...

  TYPE= COMMAND
  NAME= \lx
    START_TAG= "LTX2X"
  END_TYPE
  ```

  then there is usually no problem. On the other hand, if the document and the command table contain:

  ```
  \newcommand{\fd}[1]{\texttt{#1}}
  ....
  where \fd{InputFile} is the name ...

  TYPE= COMMAND
  NAME= \fd
    REQPARAMS= 1
  END_TYPE
  ```

  then there may be a problem, which might be as 'minor' as *LTX2X* reporting a parse error when it has reached `\newcommand{\fd}` in the input file and then carrying on, or it may be more serious.

- There is a slight problem with optional arguments. *LTX2X* always takes the first close bracket (`]`) after the opening bracket as signalling the end of the argument text. This occurs even if the close bracket is enclosed in braces (i.e. `{]}`). Opening brackets within optional argument text are handled correctly.

- It cannot sensibly handle LaTeX constructs of the form `{\em emph text}`. That is, except for command arguments, it does not recognize `{...}` as a grouping construct, so cannot successfully tag the end of the `emph text` in the example. On the other hand, if constructs like `\emph{emph text}` or `\begin{em}emph text\end{em}` are used instead, start and end tags can be generated, given appropriate specifications in the command table.

- It assumes that all commands that take arguments are written so that each argument is enclosed in braces. For example, the superscripting command should be written as `^{2}` and not as `^2`. Similarly, accent commands should be written as `\={o}` rather than `\=o`, and so on.

- There has not been time to test all aspects of the EXPRESS-A interpreter. It is possible that this may not perform quite as advertised. In particular dynamic aggregates have not been fully implemented. For example:

  ```
  LIST OF INTEGER;
  ```

  appears to be handled correctly. More complicated constructs involving dynamic aggregates, such as

  ```
  ARRAY [1:7] OF LIST OF ARRAY [-21:21] OF INTEGER;
  ```

  have not been tested. It is improbable that `BAG` will work; the status of `SET` is similar and additionally the uniqueness test for set membership has not been implemented.

- No doubt other limitations will come to light as *LTX2X* gets more use. On the other hand, *LTX2X* has been able to handle a broader range of cases than it was designed to address.

# D Command table summary

This section summarizes the commands and specifications available for defining a command table.

## D.1 Special print characters

The combination of an escape character and another character can be used to specify certain non-visible characters within a tag string. The commands are given in Table 5.

Table 5: Special print character commands.

| Command | Default |
|---|---|
| AUDIBLE_ALERT_CHAR= | a |
| BACKSPACE_CHAR= | b |
| CARRIAGE_RETURN_CHAR= | r |
| ESCAPE_CHAR= | \ |
| FORMFEED_CHAR= | f |
| HEX_CHAR= | x |
| HORIZONTAL_TAB_CHAR= | t |
| NEWLINE_CHAR= | n |
| VERTICAL_TAB_CHAR= | v |

These commands take one character as their value. If any commands are not specified, then the default value is used. These commands, if used, must be at the beginning of the command table before any `TYPE=` commands, although their ordering is not significant among themselves.

## D.2 Express-A code initialization

The keyword `CODE_SETUP=` indicates that the following part of the command table, up until the `END_CODE` keyword, contains Express-A code declarations and/or statements. If used, this block must come before any of the `TYPE=` commands.

## D.3 Comments and file inclusion

A comment within a command table file is any line starting with `C=` .

A file can be included within another command table file with the command line

```
INCLUDE= FileName
```

where `FileName` is the name of the file to be included. The `INCLUDE=` command cannot appear between the command pair `TYPE=` and its following `END_TYPE`.

The end of a command table file is either the physical end of the file or the command `END_CTFILE=`, whichever occurs first.

## D.4 Command types

All command type specifications have the general form:

```
TYPE= CommandType
NAME= CommandName
   C= a possibly empty list of mode-independent commands
   C= possibly sets of mode-dependent commands
END_TYPE
```

where `CommandType` is a keyword identifying the kind of command being specified and `CommandName` is the identifier of a LaTeX command or environment. The potential set of commands that can be used between the `TYPE=` and `END_TYPE` commands depends on the kind of command being specified, but the special print character commands, Table 5, and the `INCLUDE=` command cannot appear within a type specification. All command specifications, except for the built in command types (see Table 6), must include at least a `NAME=` command. The ordering of commands within a type specification is not significant. The ordering of type specifications within a command table file is not significant.

The `NAME=` command takes as its value the name of a LaTeX command or environment. The name must be written exactly as it would appear in a LaTeX source file. That is, `\command` for any command except `\begin{}` or `\end{}`, and as `env` for an environment begun as `\begin{env}` or ended by `\end{env}`.

### D.4.1 Built in command types

Table 6 lists the keywords for the built in command types.

Table 6: Built in command type keywords.

| Keyword | LaTeX command |
|---|---|
| BEGIN_DOCUMENT | `\begin{document}` |
| BEGIN_DOLLAR | `$` at start of in-text math |
| BEGIN_VERB | `\verb` or `\verb*` and its following character |
| BEGIN_VERBATIM | `\begin{verbatim}` or `\begin{verbatim*}` |
| END_DOCUMENT | `\end{document}` |
| END_DOLLAR | `$` at end of in-text math |
| END_VERB | the ending character for `\verb` or `\verb*` |
| END_VERBATIM | `\end{verbatim}` or `\end{verbatim*}` |
| LBRACE | `{` |
| OTHER_BEGIN | of the form `\begin{env}` not specified elsewhere |
| OTHER_COMMAND | of the form `\comm` not specified elsewhere |
| OTHER_END | of the form `\end{env}` not specified elsewhere |
| PARAGRAPH | blank source line |
| RBRACE | `}` |
| SLASH_SPACE | `\␣` |

The built in command type specifications can only sensibly use two kinds of actions — those specified at the start of the command (e.g., `PC_AT_START=` and `START_TAG=`) and/or actions at the end of the command (e.g., `PC_AT_END` and `END_TAG=`). The `NAME=` command must not be used.

The `OTHER_` types are an exception to the above, in that they can include the command line `PRINT_CONTROL= NO_PRINT`.

*LTX2X* checks the command table for the presence of these required types. If one or more have not been specified, then they are automatically added to the command table with default values (e.g. empty strings) for the tags, and a warning message is printed giving the default value(s).

### D.4.2 Optional command types

For discussion purposes, the optional command types have been tabulated in different categories. The basic distinction between these categories is the sets of commands that are permissible within the command specification.

At a minimum, all the specifications must include a `NAME=` command and must not contain any `PRINT_CONTROL=` or `INCLUDE=` commands or the special print character commands listed in Table 5.

The keywords for the general command types are given Table 7.

Table 7: General command type keywords.

| Keyword | LaTeX command form |
|---|---|
| TEX_CHAR | LaTeX special characters (except { } $) |
| CHAR_COMMAND | \c, where c is non-alphabetic |
| COMMAND | \command except for sectioning or picture commands |
| BEGIN_ENV | \begin{env} except for \item lists |
| END_ENV | \end{env} except for \item lists |
| VCOMMAND | a \verb-like command |
| BEGIN_VENV | start of a verbatim-like environment |
| END_VENV | end of a verbatim-like environment |

A general command type specification can include any of the tagging and print option commands. They cannot contain a `SECTION_LEVEL=` command, nor can they contain any of the `_ITEM_` commands.

The keywords for the specific command types are given in Table 8.

A `BEGIN_LIST_ENV` specification should include at least a `START_ITEM=` command. The other `_ITEM_` commands are optional. Other commands follow the rules for the general command types.

The potential commands for the `_PICTURE_` commands are the same as for the general commands, with the exception that commands related to optional argument processing are not available for use.

A `SECTIONING` command specification must include a `SECTIONING_LEVEL=` command. Other commands follow the rules for the general command types.

The keywords for the special command types are given in Table 9.

Apart from the general restrictions on the allowed commands within a specification, there are no restrictions on the commands that can be included within the specification of a `SPECIAL_` command. It is up to the creator of the special to decide what is appropriate. However, each `SPECIAL_` specification must include the command

```
SPECIAL_TOKEN= N
```

Table 8: Specific command type keywords.

| Keyword | LaTeX command form |
|---|---|
| BEGIN_LIST_ENV | \begin{env} for \item lists |
| BEGIN_PICTURE_CC | \begin{pic}()() |
| END_LIST_ENV | \end{env} for \item lists |
| END_PICTURE | \end{pic} |
| PICTURE_CCPP | \pic()(){}{} |
| PICTURE_CO | \pic()[] |
| PICTURE_COP | \pic()[]{} |
| PICTURE_CP | \pic(){} |
| PICTURE_OCC | \pic[]()() |
| PICTURE_OCCC | \pic[]()()() |
| PICTURE_OCO | \pic[]()[] |
| PICTURE_PCOP | \pic{}()[]{} |
| SECTIONING | \command for a document section |
| COMMAND_OOP | \com[][]{} |
| COMMAND_OOOPP | \com[][][]{}{} |
| COMMAND_OPO | \com[]{}[] |
| COMMAND_POOOP | \com{}[][][]{} |
| COMMAND_POOP | \com{}[][]{} |
| COMMAND_POOPP | \com{}[][]{}{} |

Table 9: Special command type keywords.

| Keyword | LaTeX command form |
|---|---|
| SPECIAL | reserved for possible future use |
| SPECIAL_BEGIN_ENV | \begin{env} except for \item lists |
| SPECIAL_BEGIN_LIST | \begin{env} for \item lists |
| SPECIAL_COMMAND | \command |
| SPECIAL_END_ENV | \end{env} except for \item lists |
| SPECIAL_END_LIST | \end{env} for \item lists |
| SPECIAL_SECTIONING | \command for a document section |

where N is an integer number (with $10000 \leq N \leq 32767$ for a grammar special, or $N > 50000$ for a code special) that has been specified within *LTX2X* as being identified with the grammar and actions corresponding to the value of the NAME= command for the SPECIAL_.

## D.5 Tag specification commands

### D.5.1 Arguments

The commands relating to the specification of LaTeX command arguments are given in Table 10.

Table 10: Argument commands.

| Command | Value |
|---|---|
| OPT_PARAM= | FIRST or LAST |
| REQPARAMS= | Integer. The number of required arguments |

The OPT_PARAM= command specifies that the LaTeX command takes one optional argument and it is the FIRST or LAST in the argument list.

The REQPARAMS= command specifies that the LaTeX command has **Integer** number of required arguments. **Integer** must be between one and nine[9] inclusive.

Absence of these commands implies that the relevant LaTeX command has no arguments of the unspecified kind.

### D.5.2 Tag actions

The commands for specifying the tag actions are summarized in Table 11. The _ITEM_ commands can only be used within a BEGIN_LIST_ENV or a SPECIAL_ command specification.

Table 11: Tag commands.

| Command | Application |
|---|---|
| END_ITEM= | actions after `\item` text |
| END_ITEM_PARAM= | actions after `\item` optional argument |
| END_OPT= | actions after optional argument |
| END_TAG= | actions after all arguments processed |
| END_TAG_n= | actions after n'th required argument |
| START_ITEM= | actions before `\item` |
| START_ITEM_PARAM= | actions before `\item` optional argument |
| START_OPT= | actions before optional argument |
| START_TAG= | actions at start of command |
| START_TAG_n= | actions before n'th required argument |

[9] Or eight if OPT_PARAM= is specified.

Each of these commands can specify a list of actions to be performed; typically this is just to print a text string. A string is any set of characters enclosed in double quote marks. The string can include any special printing characters. The text string starts immediately after the first double quote and ends immediately before the last double quote. The string cannot include a physical linebreak within the command table file. If the first action is to print a string then the string may be placed on the same line as the keyword.

The actions are listed one per line and are performed in the order they are listed. Table 12 lists the action commands.

Table 12: Tag actions.

| Keyword | Value | Application |
|---|---|---|
| `STRING:` | text string | Print the string |
| `SOURCE:` | `BUFFER num` | Print the contents of buffer number `num` |
| `SOURCE:` | `FILE name` | Print the contents of file `name` |
| `SOURCE:` | `SYSBUF` | Print the contents of the system buffer |
| `RESET_BUFFER:` | `num` | Reset the buffer `num` |
| `RESET_FILE:` | `name` | Reset the file `name` |
| `RESET_SYSBUF:` | | Reset the system buffer |
| `SWITCH_TO_BUFFER:` | `num` | Print to buffer number `num` |
| `SWITCH_TO_FILE:` | `name` | Print to file called `name` |
| `SWITCH_TO_SYSBUF:` | | Print to the system buffer |
| `SWITCH_BACK:` | | Reset the print mode |
| `SET_MODE:` | `name` | Set the mode to `name` |
| `RESET_MODE:` | | Reset the mode to its prior value |
| `CODE:` | | Start of a set of EXPRESS-A statements |

### D.5.3 Print control

The print control commands are summarized in Table 13. These are used to set the print mode at the start and end of a command, and for each argument. The exception is the `PRINT_CONTROL=` command which can only be used within an `OTHER_` command type specification, and which is the only print control that can be specified for the `OTHER_` commands.

Table 13: Print control commands.

| Command | Application |
|---|---|
| `PRINT_CONTROL=` | printing of `OTHER_` commands |
| `PC_AT_START=` | set printing at start of command |
| `PC_AT_END=` | set printing at end of command |
| `PRINT_OPT=` | printing of optional argument |
| `PRINT_Pn=` | printing of n'th required argument |

The values that these commands may take are given in Table 14. These direct where any print output is to be directed. The default is to send all output the the file named as the output on the command line when starting *LTX2X*.

Table 14: Print control values.

| Value | Application |
|---|---|
| `NO_PRINT` | Do not print at all |
| `TO_SYSBUF` | Print to the system buffer |
| `TO_BUFFER num` | Print to buffer number `num` |
| `TO_FILE name` | Print to file called `name` |
| `NO_OP` | Do not do any processing |
| `RESET` | Reset the print mode |

`NO_PRINT` and `NO_OP` both produce no printed output. However, in the `NO_OP` case the lexer handles all the processing, and effectively just ignores the source document text. In the `NO_PRINT` case, the source text is processed as normal, but the printing is directed to a black hole.

### D.5.4 Sectioning

`SECTIONING` command specifications require a `SECTIONING_LEVEL=` command. The values that this can take are listed in Table 15.

Table 15: Sectioning level values.

| Value | Application |
|---|---|
| `PART` | sectioning equivalent to `\part` |
| `CHAPTER` | sectioning equivalent to `\chapter` |
| `SECT` | sectioning equivalent to `\section` |
| `SUBSECT` | sectioning equivalent to `\subsection` |
| `SUBSUBSECT` | sectioning equivalent to `\subsubsection` |
| `PARA` | sectioning equivalent to `\paragraph` |
| `SUBPARA` | sectioning equivalent to `\subprargraph` |

A sectioning command specification uses the `END_TAG=` text tag differently from its use by any other specification. In this case, the tag is printed at the closure of the text forming the body of the section of the document. A document section is considered to be closed when it is followed by a higher level sectioning command. The values in Table 15 are listed in decreasing level. That is, a section at level `CHAPTER` is at a higher level than a section at level `PARA`.

**NOTE** For the use of writers of `SPECIAL_` command specifications, `SECTIONING_LEVEL=` can be given some additional values. These are `PARTm2` and `PARTm1` for levels respectively two and one higher than `PART`, and `SUBPARAp1` and `SUBPARAp2` for levels respectively one and two lower than `SUBPARA`.

# E System installation

This section describes how to install the *LTX2X* program and some of the internal size limits within *LTX2X*.

The basic *LTX2X* system requires the following source files:

`l2x.l` the lexer source.

`l2x.y` the parser source.

`l2xlib.c`, `l2xlib.h` main program and support functions.

`l2xlibtc.h` header file containing keywords and their representations as strings.

`l2xcom.h` header file for all system components (except for getopt, srchenv and the interpreter).

`l2xacts.c`, `l2xacts.h` standard action functions.

`l2xusrlb.c`, `l2xusrlb.h` special actions and user-defined functions.

`strtypes.h` header file with some type definitions.

`getopt.c`, `getopt.h` functions for handling command line options [14, Chapter 6].

`srchenv.c`, `srchenv.h` functions for searching directories for files [15, page 747].

The EXPRESS-A interpreter also requires the following files:

`l2xistup.c` the interface between the main part of *LTX2X* and the interpreter.

`l2xicmon.h` header file for the interface.

`l2xirtne.c`, `l2xistd.c`, `l2xidecl.c`, `l2xistmt.c`, `l2xiexpr.c` the files that contain the code for parsing EXPRESS-A. Respectively they deal with functions and procedures, the built-in functions, declarations, statements, and expressions.

`l2xiprse.h` header file for parsing.

`l2xixutl.c`, `l2xiexec.h` utility routines supporting the execution module and for managing the interpreter's stack.

`l2xixstd.c`, `l2xixstm.c`, `l2xixxpr.c` functions for executing the EXPRESS-A built in functions, statements and expressions.

`l2xirexp.c`, `l2xirexpr.h` general functions for processing and executing regular expressions.

`listsetc.c`, `listsetc.h` general functions for processing lists.

`l2xiscan.c`, `l2xiscan.h` lexing routines for the interpreter.

`l2xisymt.c`, `l2xisymt.h` routines for managing the interpreter's symbol tables.

`l2xidbug.c` the source level debugger.

`l2xierr.c`, `l2xierr.h` EXPRESS-A language error handling and diagnostic output for the user.

`l2xiidbg.c`, `l2xiidbg.h`, `l2xisdcl.c` diagnostics for a developer of the interpreter.

`licomsym.h` general header file for the interpreter modules.

`l2xidftc.h`, `l2xiertc.h`, `l2xisctc.h`, `l2xisftc.h` header files containing keywords and their representations as strings.

The following files may be useful:

`man` the manpage

`printct.c` a program to print and update command table files;

`ltx2html.sty` a LaTeX package file to assist in retagging a LaTeX document to an HTML document.

Essentially, installing *LTX2X* consists of processing the file `l2x.l` through a lexer generator, processing the file `l2x.y` through a parser generator, and then compiling the results together with the other supplied source files.

The lexer source file `l2x.l` and the parser source file `l2x.y` have to be processed by flex (or equivalent) and bison (or equivalent) respectively to generate C code. This code, together with the code in the other source files must then be compiled and linked to form the executable.

The executable must then, after suitable testing, be moved to its final place in your system and the manpage (file `man`) also copied to its final position in your directory structure.

Included in the *LTX2X* distribution are several command table files. One is `detex.ct` which provides an example of commands for de-TeXing a document.[10] Another is `remcom.ct` which provides an example of commands to remove comments from a LaTeX document. The command table file `bye.ct` replaces a LaTeX document by "Goodbye document". Another is `ltx2x.ct` which does nothing except try and include another file named `ZiLcH`, which

[10] You may wish to try using `detex.ct` on the LaTeX source of this document to see what the effect is. This can also serve as a check on the system installation.

presumably is not on anyone's system. Running *LTX2X* with this file will prompt for another name of a file if it cannot find `ZiLcH`; enter an existing file (like `detex.ct`) at the prompt.[11]

The command table file `l2h.ct` has proven to be adequate for converting the LaTeX source of this manual, and other LaTeX documents without pictures and only limited mathematics, into an ASCII file with HTML tags instead.

The file `fun.ct` contains some test code for the EXPRESS-A interpreter. The contents are similar to the example shown at the end of section 4.

The `l2xusrlb` files are skeletons. The system does include the functions and parser constructs for the `\GRAMMspecial` and `\CODEspecial` commands used as examples previously. The last two entries in `remcom.ct` are the specification of these, and the implementation is as described previously.

### E.1 Command table printing

The grammar of the command table has been changed slightly since the initial release of *LTX2X*. The utility C program in `printct.c` may be used to:

- Pretty-print a command table;
- Convert an original command table to one that conforms to the new grammar.

The syntax for running `printct` is:

```
printct [-D dir_cat_char] [-P path_seperators] [-f table_file] [-t]
```

where elements in square brackets are options. These options are identical to the corresponding ones for *LTX2X* and are as follows:

`-f` By default, `printct` reads the command table from a file called `ltx2x.ct`. If the required command table is in a file with another name this option is used to change from the default file. For example,

```
> printct
```

reads a command table from `ltx2x.ct`, while

```
> printct -f detex.ct
```

reads a command table from file `detex.ct`.

`-t` This generates some diagnostics related to the processing of the command table file.

[11] This is one way of setting up *LTX2X* for interactive specification of the desired command table file(s).

-D The value of this option is the character that the operating system uses to catenate directory names to form a path (see 3.1.1). The default value is a slash (i.e. /). The default could be changed to a backslash, for example, by -D \.

-P The environment variable (see 3.1.1) contains a list of directories (also known as path names). In the operating system that I use, these are separated by the colon (:) character which, together with the semi-colon and space characters, form the *LTX2X* default separators. The path separator characters can be changed with this option. For example, -P : will make the separators be a colon or a space (space is automatically included in the separator list).

printct only reads a single command table file and outputs the pretty-printed and updated version to file printct.lis. It performs a very limited amount of error checking and writes error messages and statistics to the file printct.err.

### E.2 A make file

Here is a UNIX make file [16] for the *LTX2X* system.

```
# makefile for program ltx2x --- LaTeX to X autotagger
#
##################### Change the following for your setup
# The compiler
CC = cc

# We use flex (or equivalent, but not lex) to generate the lexer
LEX = flex
# and the options
LEXFLAGS = -v

# We use bison (or equivalent) to generate the parser
YACC = bison
# and the options
YACCFLAGS = -y -d -v

# Libraries to be used
LIBS = -ly -ll -lm

# The root directory for the installation (e.g., /usr/local )
ROOTDIR = /proj/ltx/teTeX033

# Where to place the running code (e.g. /usr/local/bin )
BINDIR = ${ROOTDIR}/bin

# Where to place the manpage (e.g., /usr/local/man/man1 )
MANEXT = 1
MANDIR = ${ROOTDIR}/man/man${MANEXT}
```

```
# Just in case you want to change the name of the binary
# (and then you should also change the man page and documentation).
# So, do not change this.
PROG = ltx2x

# Where to place the user documentation (e.g., /usr/local/doc/ltx2x )
DOCDIR = ${ROOTDIR}/doc/${PROG}

# Where to place the example command tables (e.g., /usr/local/lib/config/ltx2x )
CTDIR = ${ROOTDIR}/lib/config/${PROG}

# The file copy command (copy but do not delete original)
COPY = cp

# The file move command (move and delete original)
MOVE = mv

# The file delete command
DELETE = rm

# The make directory (hierarchy) command
MAKEDIR = mkdirhier

# The stream editor command
SED = sed

# Command to write to the terminal (stdout)
ECHO = echo

################### You should not have to change anything after this

# The source modules
L2XSRCS = l2xytab.c l2xlexyy.c l2xlib.c l2xacts.c l2xusrlb.c
getopt.c srchenv.c

INTSRCS = l2xirtne.c l2xistd.c l2xidecl.c l2xistmt.c l2xiexpr.c
l2xiscan.c l2xisymt.c l2xierr.c l2xiidbg.c l2xistup.c l2xistm.c
l2xixxpr.c l2xixstd.c l2xidbug.c l2xisdcl.c l2xirexp.c listsetc.c

# The object modules
L2XOBJS = l2xytab.o l2xlexyy.o l2xlib.o l2xacts.o l2xusrlb.o
          getopt.o srchenv.o

INTSRCS = l2xirtne.o l2xistd.o l2xidecl.o l2xistmt.o l2xiexpr.o
l2xiscan.o l2xisymt.o l2xierr.o l2xiidbg.o l2xistup.o l2xistm.o
l2xixxpr.o l2xixstd.o l2xidbug.o l2xisdcl.o l2xirexp.o listsetc.o
```

```
OBJS = ${L2XOBJS} ${INTOBJS}

# Link object code together into PROG
ltx2x : ${OBJS}
        ${CC} -o ${PROG} ${OBJS} ${LIBS}

# Compile C source code into object code
getopt.o : getopt.c getopt.h
        ${CC} -c getopt.c
l2xytab.o : l2xytab.c l2xlib.h l2xusrlb.h  l2xacts.h strtypes.h l2xcom.h
        ${CC} -c l2xytab.c
l2xlexyy.o : l2xlexyy.c l2xytab.h l2xlib.h  l2xusrlb.h l2xcom.h
        ${CC} -c l2xlexyy.c
l2xlib.o : l2xlib.c getopt.h l2xytab.h strtypes.h l2xcom.h
        ${CC} -c l2xlib.c
l2xusrlb.o : l2xusrlb.c l2xlib.h l2xytab.h strtypes.h l2xcom.h
        ${CC} -c l2xusrlb.c
l2xacts.o : l2xacts.c l2xusrlb.h l2xlib.h l2xytab.h strtypes.h l2xcom.h
        ${CC} -c l2xacts.c
srchenv.o : srchenv.c srchenv.h
        ${CC} -c srchenv.c

# Generate C code for parsing
l2xytab.c l2xytab.h: l2x.y
        @ ${ECHO} "Expect 10 shift/reduce conflicts to be reported"
        ${YACC} ${YACCFLAGS} l2x.y
        ${MOVE} y.tab.c l2xytab.c
        ${MOVE} y.tab.h l2xytab.h

# Generate C code for lexing
l2xlexyy.c : l2x.l
        ${LEX} ${LEXFLAGS} l2x.l
        ${MOVE} lex.yy.c l2xlexyy.c

# the interpreter modules

# compiler flags for analyze and execute modules
ANLFLAG = -Danalyze
RUNFLAG = -Dtrace

# interpreter header files
SOMEH = l2xicmon.h l2xierr.h l2xiscan.h l2xisymt.h licomsym.h l2xiidbg.h
MOSTH = ${SOMEH} l2xiprse.h
ALLH = ${MOSTH} l2xicpr.h l2xiexec.h

# interpreter interface

l2xistup.o : l2xistup.c ${ALLH}
```

```
        ${CC} -c ${ANLFLAG} ${RUNFLAG} l2xistup.c

# the parser module

l2xirtne.o : l2xirtne.c ${ALLH}
        ${CC} -c ${ANLFLAG} l2xirtne.c

l2xistd.o : l2xistd.c ${MOSTH}
        ${CC} -c ${ANLFLAG} ${RUNFLAG} l2xistup.c

l2xistup.o : l2xistup.c ${ALLH}
        ${CC} -c l2xistd.c

l2xidecl.o : l2xidecl.c ${MOSTH} l2xicpr.h
        ${CC} -c ${ANLFLAG} l2xisdecl.c

l2xistmt.o : l2xistmt.c ${ALLH}
        ${CC} -c ${ANLFLAG} l2xistmt.c

l2xiexpr.o : l2xiexpr.c ${MOSTH} l2xicpr.h
        ${CC} -c ${ANLFLAG} l2xiexpr.c

# the scanner module

l2xiscan.o : l2xiscan.c ${SOMEH} l2xicpr.h
        ${CC} -c ${ANLFLAG} l2xiscan.c

# symbol table module

l2xisymt.o : l2xisymt.c l2xicmon.h l2xierr.h l2xisymt.h licomsym.h l2xiidbg.h
        ${CC} -c l2xisymt.c

# executor module

l2xixutl.o : l2xixutl.c ${MOSTH} l2xiexec.h listsetc.h
        ${CC} -c ${RUNFLAG} l2xixutl.c

l2xixstm.o : l2xixstm.c ${MOSTH} l2xiexec.h listsetc.h
        ${CC} -c ${RUNFLAG} l2xixstm.c

l2xixxpr.o : l2xixxpr.c ${MOSTH} l2xiexec.h listsetc.h
        ${CC} -c ${RUNFLAG} l2xixxpr.c

l2xixstd.o : l2xixstd.c ${MOSTH} l2xiexec.h listsetc.h
        ${CC} -c ${RUNFLAG} l2xixstd.c

l2xidbug.o : l2xidbug.c ${SOMEH} l2xiexec.h listsetc.h
        ${CC} -c ${RUNFLAG} l2xidbug.c
```

```
# error and miscellaneous

l2xisdcl.o : l2xisdcl.c ${SOMEH}
	${CC} -c ${ANLFLAG} ${RUNFLAG} l2xisdcl.c

l2xiidbg.o : l2xiidbg.c ${SOMEH} l2xiexec.h
	${CC} -c l2xiidbg.c

l2xirexp.o : l2xirexp.c l2xirexp.h
	${CC} -c l2xirexp.c

listsetc.o : listsetc.c listsetc.h
	${CC} -c listsetc.c

# only call make install if BINDIR has been set
install : ltx2x
	${MAKEDIR} ${BINDIR}
	${MOVE} ${PROG} ${BINDIR}

# Edit the file man to replace DOCUMENTDIR by the actual directory
# where the user manual is to be placed, and CTDIR by the location
# of the example command table files.
# Then copy the manpage to the proper place
manpage :
	${SED} 's!DOCUMENTDIR!${DOCDIR}!; s!CTDIR!${CTDIR}!' man > tman
	${MAKEDIR} ${MANDIR}
	${COPY} tman ${MANDIR}/${PROG}.${MANEXT}

# Copy the user manuals to the proper place
doc :
	${MAKEDIR} ${DOCDIR}
	${COPY} ltx2x.tex ${DOCDIR}/${PROG}.tex
	${COPY} ltx2x.ps ${DOCDIR}/${PROG}.ps
	${COPY} ltx2x.txt ${DOCDIR}/${PROG}.txt
	${COPY} ltx2x.html ${DOCDIR}/${PROG}.html

# Copy the example command tables to their final location
ctables :
	${MAKEDIR} ${CTDIR}
	${COPY} ltx2x.ct ${CTDIR}/ltx2x.ct
	${COPY} detex.ct ${CTDIR}/detex.ct
	${COPY} remcom.ct ${CTDIR}/remcom.ct
	${COPY} l2h.ct ${CTDIR}/l2h.ct
	${COPY} bye.ct ${CTDIR}/bye.ct
	${COPY} fun.ct ${CTDIR}/fun.ct

# Do almost everything except clean up
```

```
all : ltx2x install manpage doc ctables

# call make clean to remove the object files, info from YACC,
# and the edited version of the manpage
clean :
	${DELETE}  *.o
	${DELETE} y.output
	${DELETE} tman

# Compile the command table printer
printct : printct.o getopt.o srchenv.o
	${CC} -o printct printct.o getopt.o srchenv.o

printct.o : printct.c getopt.h strtypes.h l2xcom.h
	${CC} -c printct.c
```

If you use the above makefile then the first part should be edited to reflect your system's configuration. You could do `make all` which should do everything for you, except the cleaning up. A more conservative approach is recommended. First just do `make` which will generate the executable. This can then be tested. When all is well do `make install` and `make manpage` which will put the executable and the manpage into their final positions. Finally, `make clean` will remove the intermediate files generated during the build process.

The above `make` file uses flex as the lexer generator. You can use your favorite one instead but it must, unlike lex, support exclusive start states. Also, bison is used above as the parser generator. Again, you can use your favorite one. As far as I am aware, there is nothing remarkable about the grammar, except that during early development I exceeded the size limits of yacc. The grammar has been simplified since then, so this may no longer be a problem.
NOTE: If bison is used it reports that there are 10 shift/reduce conflicts. It appears that these can be safely ignored.

One compilation problem has been noted by Uwe Sassenberg[12] on AIX 3.2 and IRIX 5.3 systems, but I could not reproduce it on a SunOS 4.1.3 system. This is when the main procedure of *LTX2X* is processing the optional command line arguments. For some reason it had difficulties with the C `EOF`. The symptom was that the program compiled but when it was run it sat there absorbing CPU cycles and doing nothing as it had got into an infinite `while` loop. The cure was to insert the following line of code in file `l2xlib.c`:

```
main(argc,argv)
...
  /* get command line optional parameters */
  opterr = 1;       /* getopt prints errors if opterr is 1 */
  while (EOF != (optchar =
        getopt(argc,argv, "l:ty:f:cp:wE:P:D:"))) {
```

[12] <sassen@hal1.physik.uni-dortmund.de>

```
/* insert this line of code:  if (optchar == 255) break;  end insert */
        switch(optchar) {
...
```

This code line which you may need to insert is supplied as a comment in the distributed source.

### E.3 Limits and errors

The *LTX2X* system has some built-in limits which are defined in `l2xlib.c`. The following is a listing of the relevant sizes.

`CLAUSE_STACK_SIZE` The maximum nesting depth of document sectioning. This is set somewhat larger than the number of standard LaTeX sectioning command types. (Default 10)

`EVERY_N_LINES` Controls the frequency of printing processed line numbers to the terminal. (Default 100)

`LIST_STACK_SIZE` The maximum nesting depth of list environments. This is set somewhat larger than the standard LaTeX nesting depth. (Default 10)

`MAX_BUFFER` The maximum number of characters that can be held in the system buffer, and also the maximum number of characters in a pretty-printed output line. (Default 2000)

`MAX_CT_STACK` The maximum nesting depth for included command table files. (Default 20)

`MAX_ERRORS` The maximum number of non-fatal errors discovered in command table processing or in source file processing before *LTX2X* quits. (Default 10)

`MAX_LINE` The maximum number of characters in a line of a LaTeX source file. (Default 2000)

`MAX_PRINT_STACK` The maximum nesting depth for print control commands. (Default 100)

`MAX_TABLE_ENTRIES` The maximum number of `TYPE` specifications in a command table (including the built in type specifications). (Default 1000)

`MAX_TABLE_LINE` The maximum number of characters in a line in a command table file. (Default 254)

`MAX_USER_BUFFS` The maximum number of user buffers. (Default 20)

`MAX_UBUFF_LEN` The maximum number of characters that can be stored in a user buffer. (Default 510)

`MAX_USER_FILES` The maximum number of user files. (Default 16)

*LTX2X* prints out a summary of the program statistices at the end of the `ltx2x.err` file. If the limits are not suitable for your purposes, then they may be changed and the system rebuilt.

*LTX2X* can produce a variety of error and warning messages, for example when any of the above limits are exceeded. Some of the messages are related to command table processing, while others are related to LaTeX document processing. Both these kinds of messages are targeted to the normal end user. There is another set of messages that are aimed at the implementor of new `SPECIAL_` commands. An implementor may also find some of the debugging options useful if things really fall apart.

## E.4 Availability

Source code and documentation for *LTX2X* is available from the NIST SOLIS (SC4 On-Line Information Service) system [17] in directory
`/subject/sc4/editing/latex/programs/ltx2x`.
SOLIS can be accessed by:

- Anonymous ftp to `ftp.cme.nist.gov` (`cd` to `/pub/subject/sc4...`)
- URL `http://www.nist.gov/sc4`

Any comments should be directed to `apde@cme.nist.gov`.

### E.4.1 Copyright

Development of this software was funded by the United States Government and is not subject to copyright. It was developed by the Manufacturing Systems Integration Division (MSID) of the Manufacturing Engineering Laboratory (MEL) of the National Institute of Standards and Technology (NIST).

### E.4.2 Disclaimer

There is no warranty for the *LTX2X* software. If the *LTX2X* software is modified by someone else and passed on, NIST requests that the software's recipients be notified that what they have is not what NIST distributed.

#### Policies

1. Anyone may copy and distribute verbatim copies of the source code as received in any medium.
2. Anyone may modify your copy or copies of the *LTX2X* source code or any portion of it, and copy and distribute such modifications provided that all modifications are clearly associated with the entity that performs the modifications.

**NO WARRANTY**

NIST PROVIDES ABSOLUTELY NO WARRANTY. THE *LTX2X* SOFTWARE IS PROVIDED 'AS IS' WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD ANY PORTION OF THE *LTX2X* SOFTWARE PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

IN NO EVENT WILL NIST BE LIABLE FOR DAMAGES, INCLUDING ANY LOST PROFITS, LOST MONIES, OR OTHER SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH PROGRAMS NOT DISTRIBUTED BY NIST) THE PROGRAMS, EVEN IF YOU HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

# F A grammar for the command table

## F.1 Notation

The syntactical constructs used correspond to a derivative of the Wirth Syntax Notation (WSN) [18]. The semantics of the enclosing braces are:

- curly braces '{ }' indicate xero or more repetitions;
- square brackets '[ ]' indicate an optional element;
- parenthesis '( )' indicates a group;
- vertical bar '|' indicates that exactly one of the terms in the expression shall be chosen.

Here is the grammar for WSN defined in itself.

```
syntax     = { production } .
production = identifier '=' expression '.' .
expression = term { '|' term } .
term       = factor { factor } .
factor     = identifier | literal | group | option | repetition .
identifier = character { character } .
literal    = '''' character { character } '''' .
group      = '(' expression ')' .
```

```
option     = '[' expression ']' .
repetition = '{' expression '}' .
```

We also use the following shorthand notation for particular characters:

- `\c` — any printable character
- `\n` — the end of line character(s)
- `eof` — the end of file character(s)

## F.2 Grammar

First, the keywords. Note that these are case insensitive.

```
AudibleAlertChar = 'AUDIBLE_ALERT_CHAR=' .
BackspaceChar = 'BACKSPACE_CHAR=' .
BeginDocument = 'BEGIN_DOCUMENT' .
BeginDollar = 'BEGIN_DOLAR' .
BeginEnv = 'BEGIN_ENV' .
BeginListEnv = 'BEGIN_LIST_ENV' .
BeginPictureCc = 'BEGIN_PICTURE_CC' .
BeginVenv = 'BEGIN_VENV' .
BeginVerb = 'BEGIN_VERB' .
BeginVerbatim = 'BEGIN_VERBATIM' .
Buffer = 'BUFFER' .
CarriageReturnChar = 'CARRIAGE_RETURN_CHAR=' .
Chapter = 'CHAPTER' .
CharCommand = 'CHAR_COMMAND' .
Command = 'COMMAND' .
CommandOop = 'COMMAND_OOP' .
CommandOoopp = 'COMMAND_OOOPP' .
CommandOpo = 'COMMAND_OPO' .
CommandPooop = 'COMMAND_POOOP' .
CommandPoop = 'COMMAND_POOP' .
CommandPoopp = 'COMMAND_POOPP' .
Comment = 'C=' .
EndCtfile = 'END_CTFILE=' .
EndDocument = 'END_DOCUMENT' .
EndDollar = 'END_DOLAR' .
EndEnv = 'END_ENV' .
EndItem = 'END_ITEM=' .
EndItemParam = 'END_ITEM_PARAM=' .
EndListEnv = 'END_LIST_ENV' .
EndMode = 'END_MODE' .
EndOpt = 'END_OPT=' .
EndPicture = 'END_PICTURE' .
EndTag = 'END_TAG=' .
EndTag1 = 'END_TAG_1=' .
```

```
EndTag2 = 'END_TAG_2=' .
EndTag3 = 'END_TAG_3=' .
EndTag4 = 'END_TAG_4=' .
EndTag5 = 'END_TAG_5=' .
EndTag6 = 'END_TAG_6=' .
EndTag7 = 'END_TAG_7=' .
EndTag8 = 'END_TAG_8=' .
EndTag9 = 'END_TAG_9=' .
EndType = 'END_TYPE' .
EndVenv = 'END_VENV' .
EndVerb = 'END_VERB' .
EndVerbatim = 'END_VERBATIM' .
EscapeChar = 'ESCAPE_CHAR=' .
File = 'FILE' .
First = 'FIRST' .
FormfeedChar = 'FORMFEED_CHAR=' .
HexChar = 'HEX_CHAR=' .
HorizontalTabChar = 'HORIZONTAL_TAB_CHAR=' .
Include = 'INCLUDE=' .
InMode = 'IN_MODE=' .
Last = 'LAST' .
Lbrace = 'LBRACE' .
Name = 'NAME=' .
NewlineChar = 'NEWLINE_CHAR=' .
NoOp = 'NO_OP' .
NoPrint = 'NO_PRINT' .
OptParam = 'OPT_PARAM=' .
OtherBegin = 'OTHER_BEGIN' .
OtherCommand = 'OTHER_COMMAND' .
OtherEnd = 'OTHER_END' .
Para = 'PARA' .
Paragraph = 'PARAGRAPH' .
Part = 'PART' .
Partm1 = 'PARTm1' .
Partm2 = 'PARTm2' .
PcAtEnd = 'PC_AT_END=' .
PcAtStart = 'PC_AT_START=' .
PictureCcpp = 'PICTURE_CCPP' .
PictureCo = 'PICTURE_CO' .
PictureCop = 'PICTURE_COP' .
PictureCp = 'PICTURE_CP' .
PictureOcc = 'PICTURE_OCC' .
PictureOccc = 'PICTURE_OCCC' .
PictureOco = 'PICTURE_OCO' .
PicturePcop = 'PICTURE_PCOP' .
PrintControl = 'PRINT_CONTROL=' .
PrintP1 = 'PRINT_P1=' .
PrintP2 = 'PRINT_P2=' .
```

```
PrintP3 = 'PRINT_P3=' .
PrintP4 = 'PRINT_P4=' .
PrintP5 = 'PRINT_P5=' .
PrintP6 = 'PRINT_P6=' .
PrintP7 = 'PRINT_P7=' .
PrintP8 = 'PRINT_P8=' .
PrintP9 = 'PRINT_P9=' .
PrintOpt = 'PRINT_OPT=' .
Rbrace = 'RBRACE' .
Reqparams = 'REQPARAMS=' .
Reset = 'RESET' .
ResetBuffer = 'RESET_BUFFER:' .
ResetMode = 'RESET_MODE:' .
Sect = 'SECT' .
Sectioning = 'SECTIONING' .
SectioningLevel = 'SECTIONING_LEVEL=' .
SetMode = 'SET_MODE:' .
SlashSpace = 'SLASH_SPACE' .
Source = 'SOURCE:' .
Special = 'SPECIAL' .
SpecialBeginEnv = 'SPECIAL_BEGIN_ENV' .
SpecialBeginList = 'SPECIAL_BEGIN_LIST' .
SpecialCommand = 'SPECIAL_COMMAND' .
SpecialEndEnv = 'SPECIAL_END_ENV' .
SpecialEndList = 'SPECIAL_END_LIST' .
SpecialSectioning = 'SPECIAL_SECTIONING' .
StartItem = 'START_ITEM=' .
StartItemParam = 'START_ITEM_PARAM=' .
StartOpt = 'START_OPT=' .
StartTag = 'START_TAG=' .
StartTag1 = 'START_TAG_1=' .
StartTag2 = 'START_TAG_2=' .
StartTag3 = 'START_TAG_3=' .
StartTag4 = 'START_TAG_4=' .
StartTag5 = 'START_TAG_5=' .
StartTag6 = 'START_TAG_6=' .
StartTag7 = 'START_TAG_7=' .
StartTag8 = 'START_TAG_8=' .
StartTag9 = 'START_TAG_9=' .
String = 'STRING:' .
SubPara = 'SUBPARA' .
SubParap1 = 'SUBPARAp1' .
SubParap2 = 'SUBPARAp2' .
SubSect = 'SUBSECT' .
SubSubSect = 'SUBSUBSECT' .
SwitchBack = 'SWITCH_BACL: ' .
SwitchToBuffer = 'SWITCH_TO_BUFFER: ' .
SwitchToFile = 'SWITCH_TO_FILE: ' .
```

```
SwitchToSysbuf = 'SWITCH_TO_SYSBUF: ' .
Sysbuf = 'SYSBUF' .
TexChar = 'TEX_CHAR' .
ToBuffer = 'TO_BUFFER' .
ToFile = 'TO_FILE' .
ToSysbuf = 'TO_SYSBUF' .
Type = 'TYPE=' .
Vcommand = 'VCOMMAND' .
VerticalTabChar = 'VERTICAL_TAB_CHAR=' .
```

Some utility productions.

```
latex_id = \c { \c } .
name = \c { \c } .
text = '"' { \c } '"' .
Eol = \n .
digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .
integer = digit { digit } .
ct_file_name = name .
file_id = name .
buffer_id = integer .
mode_id = name .
```

The starting production.

```
table = [ special_chars ] { specification | inclusion | comment } eof .
```

Productions for `inclusion` and `comment` and `eof`.

```
inclusion = Include ct_file_name Eol .
comment = Comment { \c } Eol .
eof = EndCtfile { \c } Eol .
```

Productions for `special_chars`.

```
special_chars = [ escape ] [ alert ] [ backspace ] [ return ] [ feed ]
                [ hex ] [ htab ] [ newline ] [ vtab ] { comment } .
escape = EscapeChar \c Eol Eol .
alert = AudibleAlertChar \c Eol .
backspace = BackspaceChar \c Eol .
return = CarriageReturnChar \c Eol .
feed = FormfeedChar \c Eol .
hex = HexChar \c Eol .
htab = HorizontalTabChar \c Eol .
newline = NewlineChar \c Eol .
vtab = VerticalTabChar \c Eol .
```

Productions for `specification`.

```
specification = built_in | normal | list | section | special | picture | odd .
built_in = (Type built_in_type Eol) [ built_in_body ] end_type .
end_type = EndType Eol .
built_in_type = BeginDocument | BeginDollar | BeginVerb | BeginVerbatim |
                EndDocument | EndDollar | EndVerb | EndVerbatim |
                Lbrace | OtherBegin | OtherCommand | OtherEnd |
                Paragraph | Rbrace | SlashSpace .
normal = (Type normal_type Eol) type_name [ normal_body ] end_type .
type_name = Name latex_id Eol .
normal_type = BeginEnv | BeginVenv | CharCommand | Command |
              EndEnv | EndVenv | TexChar | Vcommand .
list = (Type list_type Eol) type_name [ list_body ] end_type .
list_type = BeginListEnv | EndListEnv .
section = (Type Sectioning Eol) type_name [ section_body ] end_type .
special = (Type special_type Eol) type_name [ special_body ] end_type .
special_type = Special | SpecialBeginEnv | SpecialBeginList |
               SpecialCommand | SpecialEndEnv | SpecialEndList |
               SpecialSectioning .
picture = (Type picture_type Eol) type_name [ picture_body ] end_type  .
picture_type = BeginPictureCc | EndPicture | PictureCcpp | PictureCo |
               PictureCop | PictureCp | PictureOcc |
               PictureOccc | PictureOco | PicturePcop .
odd = (Type odd_type Eol) type_name [ odd_body ] end_type .
odd_type = CommandOop | CommandOoopp | CommandOpo |
           CommandPooop | CommandPoop | CommandPoopp .
```

The X_body productions.

```
built_in_body = [ basic_body ]
                { start_mode [ basic_body ] end_mode } .
start_mode = InMode mode_id Eol .
end_mode = EndMode Eol .
normal_body = [ basic_norm_body ]
              { start_mode [ basic_norm_body ] end_mode } .
sect_body = [ basic_sect_body ]
            { start_mode [ basic_sect_body ] end_mode } .
list_body = [ basic_list_body ]
            { start_mode [ basic_list_body ] end_mode } .
picture_body = [ basic_defarg_body ]
               { start_mode [ basic_defarg_body ] end_mode } .
odd_body =  [ basic_defarg_body ]
            { start_mode [ basic_defarg_body ] end_mode } .
special_body = [ basic_special_body ]
               { start_mode [ basic_special_body ] end_mode } .
```

Note: the ordering of the components of the following basic_X_body productions is immaterial.

```
basic_body = [ start_it ] [ end_it ] .
```

```
basic_norm_body = [ basic_body ] [ no_req_arg ] [ opt_arg_pos ]
             { arg_print } { arg_action } .
basic_sect_body = sect_level [ basic_norm_body ] .
sect_level = SectioningLevel div_level Eol .
div_level = Chapter | Para | Part | Partm1 | Partm2 | Sect | Subpara |
            Subparap1 | Subparap2 | Subsect | Subsubsect .
basic_defarg_body = [ basic_body ] { arg_print } { arg_action } .
basic_list_body = [ basic_norm_body ] { item_action } .
basic_special_body = [ sect_level ] [ basic_list_body ] .
no_req_arg = Reqparams integer Eol .
opt_arg_pos = OptParam ( First | Last ) Eol .
```

The `start_it` and `end_it` productions.

```
start_it = [ start_print ] [start_action ] .
start_print = PcAtStart ( basic_pc_kind | Reset ) Eol .
basic_pc_kind = NoPrint | ToSysbuf | print_to_buffer | print_to_file .
print_to_buffer = ToBuffer buffer_id .
print_to_file = ToFile file_id .
start_action = StartTag [ text ] Eol { tag_action } .
tag_action = ( String text |
               Source ( Sysbuf | user_buffer | user_file ) |
               ResetBuffer buffer_id |
               ResetFile file_id |
               ResetSysbuf |
               SwitchToBuffer buffer_id |
               SwitchToFile file_id |
               SwitchToSysbuf |
               SwitchBack |
               SetMode mode_id |
               ResetMode )
               Eol .
user_buffer = Buffer buffer_id .
user_file = File file_id .
end_it = [ end_print ] [ end_action ] .
end_print = PcAtEnd (basic_pc_kind | Reset) Eol .
end_action = EndTag [ text ] Eol { tag_action } .
```

The `arg_print` productions.

```
arg_print = print_arg_kind (basic_pc_kind | NoOp ) Eol .
print_arg_kind = PrintOpt | PrintP1 | PrintP2 | PrintP3 | PrintP4 |
                 PrintP5 | PrintP6 | PrintP7 | PrintP8 | PrintP9 .
```

The `arg_action` productions.

```
arg_action = arg_tag_kind [ text ] Eol { tag_action } .
arg_tag_kind = EndOpt | EndTag1 | EndTag2 | EndTag3 | EndTag4 |
               EndTag5 | EndTag6 | EndTag7 | EndTag8 | EndTag9 |
             StartOpt | StartTag1 | StartTag2 | StartTag3 | StartTag4 |
            StartTag5 | StartTag6 | StartTag7 | StartTag8 | StartTag9 .
```

The `item_action` productions.

```
item_action = item_tag_kind [ text ] Eol { tag_action } .
item_tag_kind = EndItem | EndItemParam | StartItem | StartItemParam .
```

The parser in *LTX2X* for the command table is very simple. For each `TYPE=` in the command table it creates a `struct` to hold the specification data. If any type is multiply defined, then which one will be finally used is somewhat random because of the sorting and searching algorithms employed internally. No checks are made for multiply defined entries.

Each command in the command table starts on a seperate line. The parser reads only as much of a table line as is necessary to parse that line according to the first token that it finds on the line. The data in each line after parsing is added to the current `struct` for the LaTeX command. If any of the command lines within an entry are multiply defined, then the latest one will overwrite any earlier ones.

This line-based parsing means that effectively anything between the end of the required data on the line is ignored by the parser, and so could be treated as a comment. There is no guarantee that this behaviour will be maintained in future releases of *LTX2X*.

## G A grammar for Express-A

The same WSN notation is used for the grammar for Express-A as for the command table grammar.

First the keywords. Note that these are case insensitive. Also not all of the keywords have been used in this implementation of Express-A; those that have not been used are reserved for the future.

```
ABS = 'abs' .
ABSTRACT = 'abstract' .
ACOS = 'acos' .
AGGREGATE = 'aggregate' .
ALIAS = 'alias' .
AND = 'and' .
ANDOR = 'andor' .
ARRAY = 'array' .
AS = 'as' .
ASIN = 'asin' .

ATAN = 'atan' .
BAG = 'bag' .
BEGIN = 'begin' .
BINARY = 'binary' .
BLENGTH = 'blength' .
BOOLEAN = 'boolean' .
BY = 'by' .
CALL = 'call' .
CASE = 'case' .
CONSTANT = 'constant' .
```

```
CONST_E = 'const_e' .
CONTEXT = 'context' .
COS = 'cos' .
CRITERIA = 'criteria' .
DERIVE = 'derive' .
DIV = 'div' .
ELSE = 'else' .
END = 'end' .
END_ALIAS = 'end_alias' .
END_CALL = 'end_call' .

END_CASE = 'end_case' .
END_CODE = 'end_code' .
END_CONSTANT = 'end_constant' .
END_CONTEXT = 'end_context' .
END_CRITERIA = 'end_criteria' .
END_ENTITY = 'end_entity' .
END_FUNCTION = 'end_function' .
END_IF = 'end_if' .
END_LOCAL = 'end_local' .
END_MODEL = 'end_model' .

END_NOTES = 'end_notes' .
END_OBJECTIVE = 'end_objective' .
END_PARAMETER = 'end_parameter' .
END_PROCEDURE = 'end_procedure' .
END_PURPOSE = 'end_purpose' .
END_REALIZATION = 'end_realization' .
END_REFERENCES = 'end_references' .
END_REPEAT = 'end_repeat' .
END_RULE = 'end_rule' .
END_SCHEMA = 'end_schema' .

END_SCHEMA_DATA = 'end_schema_data' .
END_TEST_CASE = 'end_test_case' .
END_TYPE = 'end_type' .
ENTITY = 'entity' .
ENUMERATION = 'enumeration' .
EOF = 'eof' .
EOLN = 'eoln' .
ESCAPE = 'escape' .
EXISTS = 'exists' .
EXP = 'exp' .
FALSE = 'false' .

FIXED = 'fixed' .
FOR = 'for' .
FORMAT = 'format' .
```

```
FROM = 'from' .
FUNCTION = 'function' .
GENERIC = 'generic' .
HIBOUND = 'hibound' .
HIINDEX = 'hiindex' .
IF = 'if' .
IMPORT = 'import' .

IN = 'in' .
INSERT = 'insert' .
INTEGER = 'integer' .
INVERSE = 'inverse' .
LENGTH = 'length' .
LIKE = 'like' .
LIST = 'list' .
LOBOUND = 'lobound' .
LOINDEX = 'loindex' .
LOCAL = 'local' .

LOG = 'log' .
LOG10 = 'log10' .
LOG2 = 'log2' .
LOGICAL = 'logical' .
MOD = 'mod' .
MODEL = 'model' .
NOT = 'not' .
NOTES = 'notes' .
NUMBER = 'number' .
NVL = 'nvl' .

OBJECTIVE = 'objective' .
ODD = 'odd' .
OF = 'of' .
ONEOF = 'oneof' .
OPTIONAL = 'optional' .
OR = 'or' .
ORD = 'ord' .
OTHERWISE = 'otherwise' .
PARAMETERi = 'parameter' .
PI = 'pi' .

PRED = 'pred' .
PROCEDURE = 'procedure' .
PURPOSE = 'purpose' .
QUERY = 'query' .
READ = 'read' .
READLN = 'readln' .
REAL = 'real' .
```

```
REALIZATION = 'realization' .
REFERENCE = 'reference' .
REFERENCES = 'references' .

REMOVE = 'remove' .
REPEAT = 'repeat' .
RETURN = 'return' .
REXPR = 'rexpr' .
ROLESOF = 'rolesof' .
ROUND = 'round' .
RULE = 'rule .
SCHEMA = 'schema' .
SCHEMA_DATA = 'schema_data' .
SELECT = 'select' .

SELF = 'self' .
SET = 'set' .
SIN = 'sin' .
SIZEOF = 'sizeof' .
SKIP = 'skip' .
SQRT = 'sqrt' .
STRING = 'string' .
SUBOF = 'subof' .
SUBTYPE = 'subtype' .
SUCC = 'succ' .

SUPERTYPE = 'supertype' .
SUPOF = 'supof' .
SYSTEM = 'system' .
TAN = 'tan' .
TEST_CASE = 'test_case' .
THE_DAY = 'the_day' .
THE_MONTH = 'the_month' .
THE_YEAR = 'the_year' .
THEN = 'then' .
TO = 'to' .

TRUE = 'true' .
TRUNC = 'trunc' .
TYPE = 'type' .
TYPEOF = 'typeof' .
UNIQUE = 'unique' .
UNKNOWN = 'unknown' .
UNTIL = 'until' .
USE = 'use' .
USEDIN = 'usedin' .
USING = 'using' .
```

```
VALUE = 'value' .
VALUE_IN = 'value_in' .
VALUE_UNIQUE = 'value_unique' .
VAR = 'var' .
WHERE = 'where' .
WHILE = 'while' .
WITH = 'with' .
WRITE = 'write' .
WRITELN = 'writeln' .
XOR = 'xor' .
```

The following rules define various classes of characters which are used in constructing the tokens.

```
digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .
digits = digit { digit } .
letter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' |
               'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' |
               'w' | 'x' | 'y' | 'z' .
lparen_not_star = '(' not_star .
not_lparen_star = not_paren_star | ')' .
not_paren_star = letter | digit | not_paren_star_special .
not_paren_star_quote_special = '!' | '"' | '#' | '$' | '%' | '&' | '+' |
                   ',' | '-' | '.' | '/' | ':' | ';' | '<' | '=' | '>' | '?' |
                   '@' | '[' | '\' | ']' | '^' | '_' | '`' | '{' | '|' | '}' |
                   '~' .
not_paren_star_special = not_paren_star_quote_special | '''' .
not_quote = not_paren_star_quote_special | letter | digit | '(' | ')' | '*' .
not_rparen = not_paren_star | '*' | '(' .

not_star = not_paren_star | '(' | ')' .
octet = hex_digit hex_digit .
special = not_paren_star_quote_special | '(' | ')' | '*' | '''' .
star_not_rparen = '*' not_rparen .
```

The following rules specify how certain combinations of characters are interpreted as lexical elements within the language.

```
integer_literal = digits .
real_literal = digits '.' [ digits ] [ 'e' [ sign ] digits ] .
simple_id = letter { letter | digit | '_' } .
simple_string_literal = \q { ( \q \q ) | not_quote | \s | \o } \q .
```

The following rules specify the syntax of comments in Express-A.

```
embedded_remark = '(*' { not_lparen_star | lparen_not_star |
                     star_not_rparen | embedded_remark } '*)' .
remark = embedded_remark | tail_remark .
tail_remark = '--' { \a | \s | \o } \n .
```

The following rules represent identifiers which are known to have a particular meaning (i.e., to be declared elsewhere as types or functions, etc.).

```
attribute_ref = attribute_id .
constant_ref = constant_id .
entity_ref = entity_id .
enumeration_ref = enumeration_id .
function_ref = function_id .
parameter_ref = parameter_id .
procedure_ref = procedure_id .
type_ref = type_id .
variable_ref = variable_id .
```

The following rules specify how the previous lexical elements may be combined into constructs of EXPRESS-A. White space and/or remark(s) may appear between any two tokens in these rules. The primary syntax rule for EXPRESS-A is `express_a`.

```
actual_parameter_list = '(' parameter { ',' parameter } ')' .
add_like_op = '+' | '-' | OR | XOR .
aggregation_types = array_type | bag_type | list_type | set_type .
algorithm_head = { declaration } [ local_decl ] .
array_type = ARRAY bound_spec OF base_type .
assignment_stmt = general_ref { qualifier } ':=' expression ';' .
attribute_decl = attribute_id .
attribute_id = simple_id .
attribute_qualifier = '.' attribute_ref .
bag_type = BAG [ bound_spec ] OF base_type .

base_type = aggregation_types | simple_types | named_types .
boolean_type = BOOLEAN .
bound_1 = numeric_expression .
bound_2 = numeric_expression .
bound_spec = '[' bound_1 ':' bound_2 ']' .
built_in_constant = CONST_E | PI | THE_DAY | THE_MONTH | THE_YEAR | '?' .
built_in_function = ABS | COS | EOF | EOLN | EXISTS | EXP |
                    HIBOUND | HIINDEX | LENGTH | LOBOUND | LOINDEX |
                    LOG | LOG2 | LOG10 | NVL | ODD | ORD | PRED |
                    REXPR | ROUND | SIN | SIZEOF |
                    SQRT | SUCC | TAN | TRUNC .
built_in_procedure = INSERT | PRINT | PRINTLN | READ | READLN | REMOVE |
                     SYSTEM | WRITE | WRITELN .
case_action = case_label { ',' case_label } ':' stmt .
case_label = expression .

case_stmt = CASE selector OF { case_action } [ OTHERWISE ':' stmt ]
                  END_CASE ';' .
compound_stmt = BEGIN stmt { stmt } END ';' .
constant_factor = built_in_constant .
```

```
constructed_types = enumeration_type .
declaration = entity_decl | function_decl | procedure_decl | type_decl .
entity_body = { explicit_attr } .
entity_decl = entity_head entity_body END_ENTITY ';' .
entity_head = ENTITY entity_id ';' .
entity_id = simple_id .
enum_id = simple_id .

enumeration_reference = enum_id .
enumeration_type = ENUMERATION OF '(' enum_id { ',' enum_id } ')' .
escape_stmt = ESCAPE ';' .
explicit_attr = attribute_decl { ',' attribute_decl } ':' base_type ';' .
express_a = { declaration } [ local_decl ] { stmt } END_CODE .
expression = simple_expression [ rel_op_extended simple_expression ] .
factor = simple_factor [ '**' simple_factor ] .
formal_parameter = parameter_id { ',' parameter_id } ':' parameter_type .
function_call = ( built_in_function | function_ref )
                      [ actual_parameter_list ] .
function_decl = function_head [ algorithm_head ] stmt { stmt }
                      END_FUNCTION ';' .

function_head = FUNCTION function_id [ '(' formal_parameter
                      { ';' formal_parameter } ')' ] ':' parameter_type ';' .
function_id = simple_id .
generalized_types = general_aggregation_types .
general_aggregation_types = general_array_type | general_bag_type |
                            general_list_type | general_set_type .
general_array_type = ARRAY [ bound_spec ] OF parameter_type .
general_bag_type = BAG [ bound_spec ] OF parameter_type .
general_list_type = LIST [ bound_spec ] OF parameter_type .
general_ref =  parameter_ref | variable_ref .
general_set_type = SET [ bound_spec ] OF parameter_type .
if_stmt = IF logical_expression THEN stmt { stmt } [ ELSE stmt { stmt } ]
                END_IF ';' .

increment = numeric_expression .
increment_control = variable_id ':=' bound_1 TO bound_2 [ BY increment ] .
index = numeric_expression .
index_1 = index .
index_2 = index .
index_qualifier = '[' index_1 [ ':' index_2 ] ']' .
integer_type = INTEGER .
interval = '{' interval_low interval_op interval_item interval_op
               interval_high '}' .
interval_high = simple_expression .
interval_item = simple_expression .

interval_low = simple_expression .
```

```
interval_op = '<' | '<=' .
list_type = LIST [ bound_spec ] OF base_type .
literal = integer_literal | logical_literal | real_literal |
                string_literal .
local_decl = LOCAL local_variable { local_variable } END_LOCAL ';' .
local_variable = variable_id { ',' variable_id } ':' parameter_type ';' .
logical_expression = expression .
logical_literal = FALSE | TRUE | UNKNOWN .
logical_type = LOGICAL .
multiplication_like_op = '*' | '/' | DIV | MOD | AND | '||' .

named_types = entity_ref | type_ref .
null_stmt = ';' .
numeric_expression = simple_expression .
parameter = expression .
parameter_id = simple_id .
parameter_type = generalized_types | named_types | simple_types .
population = entity_ref .
primary = literal | ( qualifiable_factor { qualifier } ) .
procedure_call_stmt = ( built_in_procedure | procedure_ref )
                            [ actual_parameter_list ] ';' .
procedure_decl = procedure_head [ algorithm_head ] { stmt } END_PROCEDURE ';' .

procedure_head = PROCEDURE procedure_id [ '(' [ VAR ] formal_parameter
                       { ';' [ VAR ] formal_parameter } ')' ] ';' .
procedure_id = simple_id .
qualifiable_factor = attribute_ref | constant_factor | function_call |
                           general_ref | population .
qualifier = attribute_qualifier | index_qualifier .
real_type = REAL .
referenced_attribute = attribute_ref | qualified_attribute .
rel_op = '<' | '>' | '<=' | '>=' | '<>' | '=' | ':<>:' | ':=:' .
rel_op_extended = rel_op | IN | LIKE .
repeat_control = [ increment_control ] [ while_control ] [ until_control ] .
repeat_stmt = REPEAT repeat_control ';' stmt { stmt } END_REPEAT ';' .

return_stmt = RETURN [ '(' expression ')' ] ';' .
selector = expression .
set_type = SET [ bound_spec ] OF base_type .
sign = '+' | '-' .
simple_expression = term { add_like_op term } .
simple_factor = enumeration_reference | interval |
                ( [ unary_op ] ( '(' expression ')' | primary ) ) .
simple_types = integer_type | logical_type | real_type | string_type .
skip_stmt = SKIP ';' .
stmt = assignment_stmt | case_stmt | compound_stmt | escape_stmt |
       if_stmt | null_stmt | procedure_call_stmt | repeat_stmt | return_stmt |
       skip_stmt .
string_literal = simple_string_literal .
```

```
string_type = STRING .
term = factor { multiplication_like_op factor } .
type_decl = TYPE type_id '=' underlying_type ';' END_TYPE ';' .
type_id = simple_id .
unary_op = '+' | '-' | NOT .
underlying_type = constructed_types | aggregation_types | simple_types |
                  type_ref .
until_control = UNTIL logical_expression .
variable_id = simple_id .
while_control = WHILE logical_expression .
```

## References


[1] Leslie Lamport. *LaTeX: A Document Preparation System.* Addison-Wesley Publishing Company, second edition, 1994.

[2] Donald E. Knuth. *The TeXbook.* Addison-Wesley Publishing Company, 1984.

[3] ISO 10303. *Industrial automation systems and integration — Product data representation and exchange*, 1994.

[4] C. A. Goldfarb. *The SGML Handbook.* Oxford University Press, 1990. (Edited and with a foreword by Yuri Rubinsky).

[5] Chuck Musciano and Bill Kennedy. *HTML — The Definitive Guide.* O'Reilly & Associates, Inc., 1996.

[6] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, second edition, 1988.

[7] Douglas A. Schenck and Peter R. Wilson. *Information Modeling the EXPRESS Way.* Oxford University Press (ISBN 0-19-308714-3), 1994.

[8] ISO 10303-11:1994. *Industrial automation systems and integration — Product data representation and exchange — Part 11: Description methods: The EXPRESS language reference manual*, 1994.

[9] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc.* O'Reilly & Associates, Inc., second edition, 1992.

[10] M. E. Lesk and E. Schmidt. 'LEX — A Lexical Analyser Generator'. In *UNIX Programmer's Manual 2.* AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[11] S. C. Johnson. *YACC — Yet Another Compiler Compiler.* C S Technical Report 32, Bell Telephone Laboratories, Murray Hill, NJ, 1975.

[12] ISO/TR 10303-12:1997. *Industrial automation systems and integration — Product data representation and exchange — Part 12: Description method: The EXPRESS-I language reference manual*, 1997.

[13] Peter R. Wilson. *'FLaTTeN: A Program to Flatten LaTeX Source Files'*. NIST, Gaithersburg, MD 20899, December 1994. (In draft).

[14] Don Libes. *Obfuscated C and Other Mysteries*. John Wiley & Sons, Inc., 1993.

[15] A. I. Holub. *Compiler Design in C*. Prentice-Hall, Inc., 1990.

[16] Andrew Oram and Steve Talbott. *Managing Projects with make*. O'Reilly & Associates, Inc., second edition, 1991.

[17] Gaylen R. Rinaudot. *The IGES/PDES Organization STEP On-Line Information Service*. NISTIR 5511, NIST, Gaithersburg, MD 20899, October 1994.

[18] N. Wirth. 'What Can We Do About the Unnecessary Diversity of Notation for Syntactic Definitions?'. *Communications of the ACM*, 20(11):822–823, November 1977.

[19] Ronald Mak. *Writing Compilers & Interpreters — An Applied Approach*. John Wiley & Sons, Inc., 1991.