Seminar Report

99 bottles of beer - Programming in TEX

Matthias Dieter Wallnöfer
matthias.wallnoefer@student.uibk.ac.at

19 February 2011

Supervisor: Ass.-Prof. Priv.-Doz. Dr. Georg Moser

Abstract

"TEX" (ancestor of "LATEX" in which also this text was written in) not only allows to format text but also encapsulates a powerful but probably not very easy understandable macro language.
This language is subject to various analysis in this seminar report. We will try to understand for which purpose this languages has been developed, how powerful it is and how it can be used to realise meaningful programs. We will notice that the TEX compiler can also be used as interpreter to interact with the user through input and output operations.
Finally we will be performing a comparison of TEX to the scripting language of "OpenOffice Writer", a popular word-processor.

Contents

1 Introduction

In this report there are presented the programming capabilities of TEX, which especially in the enhancement LATEX is one of the *most famous typesetting systems* on the world. It is widely used for technical documents in various arts as Mathematics, Computer Science, Physics, Engineering. Originally it has been developed by *Donald E. Knuth*, an important scientist in the world of computer science. In order to get a brief overview of the history the reading of Chapter 2 is suggested.

The mentioned typesetting system works with plain text and resource files (mainly images, templates and fonts) as input, compiles the lot and produces the output as DVI[1] or optionally in newer releases also as PDF files. Chapter 3 should help understanding this.

TEX and also the descendant LATEX incorporate a quite powerful macro language which works with *macro expansion*. That means command calls are expanded and get substituted with there definitions at compilation time. In this paper the focus lies in understanding these capabilities and to explain the most important constructs. There will also be a discussion about the reason why this language has been introduced. All together is described in Chapter 4.

The next point to analyse are the *extension capabilities*. We presume the knowledge about LATEX and huge macro collections of other frameworks (an example are the CTAN archives.[2]) So in the frame of this report there will be discussed questions as: "How capable is this macro language?" or "Is it even Turing complete?". These will be answered on behalf of a list framework which is based on λ-calculus. This can be reproduced in Chapter 6.

The last topic will try to draw a line between programming in TEX and *Basic scripting* known from popular Office suites as "MS Office", "StarOffice" and "OpenOffice". There will be explained why this scripting support has been added, for what good it is and how different it is to program the first or the latter. This part has been covered in Chapter 7.

2 TEX – Some Anecdotes

TEX is a popular typesetting system developed by Donald E. Knuth, a professor at the University of Stanford in California.[3]

Knuth has been working on the second volume of his monograph "The art of Computer Programming" and was very disappointed after receiving its print preview. Around that time (1977) he encountered for the first time a digital typesetting system which lead to the decision to write his own. He wanted a system which allows high-level text processing especially regarding fonts and math formulas. Another important goal was the output consistency between computer systems. That means each computer where Knuth's system is able to run, indifferently if it is older or newer should generate the exact same output.

[1] "DVI" is a format comparable to PDF.
[2] CTAN, `http://www.ctan.org`.
[3] Wikipedia, `http://en.wikipedia.org/wiki/TeX`.

After some thinking he coined the term TEX which has been chosen for the similarity to "technology". It references back to the word $\tau\epsilon\chi$ written in the Greek alphabet and it is also spelled like this (in German "Tech"). He launched the development in "WEB"[4], a programming language invented by himself. The difference regarding conventional languages is the mixture of source code and documentation in one file – this is called "literate programming".

The main engineering process ended back in the year 1989 at version 3. From there on only bugs had been fixed and no features had been added. After each change a post-comma number had been added in order to resemble step-by-step the value of π. Consequently the version at his time of death will finally receive exactly this name and all development will be definitely stopped. This is required to allow a tremendous back- and forward compatibility and each bug found afterwards will simply be called "feature".

Naturally this does not stop enhancements made in other directions, for example LATEX. LATEX is basically a huge macro collection for TEX which allows a big number of important tasks to be achieved much simpler – e.g. the document structure and picture management. Also this seminar report has been written with the help of it.

3 TEX As Text Processor

In order to give people a quick overview of the method of operation, we will try to explain it step by step. First we will have our plain text file which embeds text and control commands (comparable to HTML tags). Then we will be running it under the compiler which creates an output either in the DVI or in the much more popular PDF format.

3.1 The input file

In most cases this file is provided in form of a plain text document with the file extension ".tex". But it can also be a bunch of TEX commands which are specified interactively when the compiler runs. As the second use is much more scarce to find we will be focusing on the first one.

Our first example is as follows:

Example 3.1. TEX file *formula.tex*

```
The quadratic formula is $$-b \pm \sqrt{b^2 - 4ac} \over 2a$$
\bye
```

The result will be a document which embeds the text "The quadratic formula is" followed with the popular mathematics expression in dollar signs (they are used as identification symbols for the formula mode). The command "pm" means plus and minus together, "sqrt" square root and "over" a fraction. The braces outline the arguments. "bye" tells the TEX compiler to stop its interpretation and to start generating the output.

[4] More informations to "WEB" under `http://sunburn.stanford.edu/~knuth/cweb.html`.

3.2 The compiler

Per default newer TEX releases are shipped with two compiler executables: "tex" starts the variant which generates the traditional DVI output files, and "pdftex" generates files in the much more diffused PDF format – so conversion tools as "dvipdf" often become unnecessary.

Now we will be running the compilation procedure by following command:

```
$ tex formula.tex
This is TeX, Version 3.141592 (Web2C 7.5.6)
(./formula.tex [1] )
Output written on formula.dvi (1 page, 508 bytes).
Transcript written on formula.log.
```

And this might be the console output. We are getting informed that TEX has been able to generate a DVI output with one page.

3.3 The output

"How to open a DVI document?" the next question could be. Well, there exist a bunch of tools - one very popular for the X windowing system[5] is called "xdvi" (for completion we mention that there exists also the PDF pedant "xpdf").

Let us now start the viewer:

```
$ xdvi formula.dvi
```

And this is the output which we should be getting (Figure 1).

The quadratic formula is

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Figure 1: Output

This should give a brief overview and represent the base for understanding the next chapters. As we will be seeing later, the output file is not strictly necessary. We are even able to perform input and output operations at compilation time and convert TEX to an interpreted scripting language – Chapter 5 shows that.

4 The Macro Language

Originally the macro language of TEX was thought to speed up the writing process for documents to save repetitions, say to provide boiler plates. [6] But as already stated in the introduction this is not the only capability - otherwise the implementation of frameworks as LATEX would not have been possible. Below let us focus on some important characteristics from [2, Chapter 20].

[5] X windowing system, the popular UNIX GUI standard.

[6] "Boiler plates" is another term for "text macros".

4.1 The definition/macro

One of the most common instructions is the *definition*, often simply referenced as *macro*.

Definition 4.1. The definition

```
\def<control sequence><parameter text>{<replacement text>}
```

The scope of this construct is the replacement of a certain control sequence even including parameters (referenced as "#1" to "#9") through a text. The text can be plain-text or even be formed by other commands including the definition's name itself. Also these new commands will be *expanded* (=evaluated) until we get to base instructions or the parser's stack is full. Exactly this technique is often referenced as *macro expansion*. The definition is in some way comparable to procedures and functions known from other programming languages and even recursions are possible (explained later).

As an example think that we make much use of "$(x_1, \ldots, x_n)$". Since we are lazy to type let us introduce the following macro:

Example 4.2. Macro "xvec"

```
\def\xvec{(x_1,\ldots,x_n)}
```

To demonstrate the capabilities in the area of parameters let us change the "xvec" macro to be more generic regarding the vector's name and the last component. This might be the result:

Example 4.3. Macro "vec"

```
\def\vec#1#2{(#1_1,\ldots,#1_#2)}
```

The next example contains three definitions which do not yield a very useful output (ABCAB) but show how calls and definitions can be nested. The idea here is to redefine "a" multiple times. The first step when calling "puzzle" is to expand "a" which has been defined as "b". "b" prints out "A" and redefines "a". We get back to "puzzle" and get the next "a" definition as a printed "B" and a redefinition of "a". Then there is printed "C" and "a" is defined to "b" – that means the game restarts. After the five "a" expansions in "puzzle" we are done.

Example 4.4. Macro expansion

```
\def\a{\b}
\def\b{A\def\a{B\def\a{C\def\a{\b}}}}
\def\puzzle{\a\a\a\a\a}
```

4.2 The substitution

Now let us analyse the *substitution* command. It has the syntax:

Definition 4.5. The substitution

```
\let\<name>=<token>
```

The reader will directly start asking in which aspect lies the difference of commands like:

Example 4.6. Difference between "let" and "def"

```
\let\a=\b
\def\a{\b}
```

In the first case `\b` is suddenly expanded and assigned to `\a` – so it is tied statically. In the second case on each call of `\a` there will be performed a macro expansion; that means `\b` is always resolved as it is currently defined – so we have a dynamic linkage.

4.3 Data types

Before learning about control structures we need some knowledge about storage possibilities. Well, the TEX macro language provides some kind of variables or better *registers*. They are primarily thought for saving dimensions as distances, sizes, fonts sizes but also counters. But since we are focusing on general purpose programming we only discuss the *user-definable counters*.

These counters are able to save signed integer values, so floating point numbers are not possible at all. Sometimes we can avoid the restriction: money amounts could also be saved in cent units. Important operations include:

- Define them: `\newcount\<name>`
- Initialise them: `\<name>=<signed integer value>`
- Increment/decrement them:
  `\advance\<name> by <signed integer value>`
- Read the value: `\the\<name>`

Another much differently kind of storage are *non-parametrised definitions*. So TEX e.g. allows us to store strings (and much more – another example in Chapter 6).

4.4 Control structures

Using the mentioned constructs we are able to write very simple macros. But is not there also a possibility to perform some kind of distinctions, to evaluate expressions? After all this is provided by each reasonable programming system. And also in TEX there is – as known from other programming languages – an *if* construct:

Definition 4.7. General syntax of "if" commands

```
\if<condition><true text>\else<false text>\fi
```

But we should be aware that this is just a template – TEX in fact does not provide only one "if" keyword, no, there exist a bunch of them. An overview of some general-purpose ones is given in Table 1.

`\ifnum<number_1><relation><number_2>`	Compare two integers
`\ifodd<number>`	Test for odd integer
`\ifx<token_1><token_2>`	Test if tokens agree

Table 1: Various "if"s

Finally there exists also a conditional statement, which is capable of making a many-way branch:

Definition 4.8. Many-way branching

```
\ifcase<number><text for case 0>\or...\or<text for case \n>
\else<text for all other cases>\fi
```

As an example we should think about a macro which generates an output if we have to pay something or not. "balance" is a user-definable counter and "fullypaid", "overpaid" and "underpaid" are the respective macros which generate the output for each case.

Example 4.9. Macro "statement"

```
\def\statement{
 \ifnum\balance=0
  \fullypaid
 \else
  \ifnum\balance>0
   \overpaid
  \else
   \underpaid
  \fi
 \fi
}
```

4.5 Loops and recursion

TEX does not provide any loop constructs. So we have to work with *recursions*. Chapter 5 explains the method to use – hence we do not give an example here.

4.6 Result

Okay, this should be enough for a quick overview. We have seen that there do exist constructs which allow programming – but these are far less comfortable in comparison to conventional languages as C or Java. Fortunately there are extension possibilities: an example is the framework described in Chapter 6.

5 A Programming Example

This example motivates an implementation possibility of the well-known algorithm of "99 bottles of Beer".[7]

Example 5.1. Version of the "99 bottles of Beer" program

```
% TeX/LaTeX version of 99 bottles of Beer
%%
%% Craig J Copi - copi@oddjob.uchicago.edu
%% Modified by Matthias Wallnoefer for Vertiefungsseminar 2010/11
%%
```

This is a modified version which contrary to the original program (which generates a document) prints the content out to the screen at compilation time and does not generate any output document.

```
\def\myprint#1{\immediate\write16{#1}} % stdout printing
```

The "myprint" macro defines a possibility to print text using the standard output device (generally a terminal window) including a trailing line feed.

```
\newcount\beercurr
\def\beer#1{\beercurr=#1\let\next=\removebeer\removebeer}
```

We define a counting variable "beercurr" and a macro called "beer" which takes as argument the number of iterations. It assigns this information to the counter variable and the macro "removebeer" by a substitution to "next". Last, we call the macro "removebeer" to finally invoke the real recursion.

```
\def\removebeer{
 \ifnum\beercurr>1
  \myprint{\the\beercurr\space bottles of beer on the wall,}
  \myprint{\the\beercurr\space bottles of beer,}
  \myprint{take one down, pass it around,}
  \advance\beercurr by -1
  \myprint{\the\beercurr\space bottle\ifnum1<\beercurr{s}\fi\space
   of beer on the wall.}
 \else
  \myprint{1 bottle of beer on the wall,}
  \myprint{1 bottle of beer,}
  \myprint{take one down, pass it around,}
  \myprint{no bottles of beer on the wall.}
  \myprint{Time to buy some more beer....}
  \let\next=\relax
 \fi
 \myprint{}
 \next}
```

The definition "removebeer" checks how many iterations are left (by the counter "beercurr") and if there are more than one, the "then" block is launched. This one prints out the content of the counter by using "the" ("space" is needed to have a space before the next text piece – otherwise TEX ignores blanks after a command). We are decrementing the counter by one, print out an "s" if more

[7] "99 bottles of Beer" programs under `http://99-bottles-of-beer.net/`.

than one bottle is left, followed by a newline outside of the "if" and call "next", which is still set to be "removebeer". This is the *recursion step*.
But if the condition is wrong we run into the "else" part, print out the written lines and reassign "next" to be "relax" - that means do nothing. Hence the last call ("next") is useless and the recursion stops – so this is the *recursion base case*.

```
\beer{9}

\bye
```

The last part of the program starts the recursion by calling the macro with the parameter "9". "bye" as already explained in Chapter 3 tells the TEX compiler to stop.

Please keep in mind the algorithm – a similar implementation will be used in Chapter 7!

To complete the discussion how TEX can be told to interact with a user let us briefly analyse following code snippet. "read16" is a console input macro, "write18" however allows even to start external software if TEX was launched with the parameter "-shell-escape".

Example 5.2. User's name program

```
% Run command: tex -shell-escape <program>.tex
\def\myprint#1{\immediate\write16{#1}} % stdout printing
\myprint{Please type in your name:}
\read16 to\myname
\myprint{Hello \myname, here is xclock!}
\write18{xclock}
\bye
```

6 Programming Using Lambda-Calculus – List Support

The previous programming example might have focused the attention of the reader, but we know that this kind of programming is very cumbersome. So we would desire something more concrete, a paradigm which is more familiar to most part of computer scientists and programmers.

Mr. Alain Jeffrey invested quite some time roughly twenty years ago (1990) to have *list support* in TEX as he knew it from functional programming languages on the base of λ-calculus[1]. He managed to achieve that and fortunately he published his results.

Definition 6.1. (Untyped) λ-*calculus* is one of the computability models which can describe recursive enumerable languages – in other words it has the same power as Turing machines. The model is incredibly simple as it only embeds three base operations: *variables* (primitive data storage), *abstractions* (kind of functions) and *applications*. (kind of function invocations)[8]

[8] Abbreviated from Wikipedia, `http://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition`.

This report does not cover the handling of λ-terms itself, which incorporates many conventions regarding expressions, evaluation strategies and representations (e.g. "How do I encode a natural number?"). It is expected that the reader familiarises himself with these.

Proof. To be able to extend this support onto TEX we need to define a mapping on the base of the three operations. This can be done as shown in Table 2. □

λ-term		**TEX pedant**
Variable, e.g. x		`\def\<name>{...}`
Abstraction, e.g. $(\lambda x.t)$	$\Longrightarrow$	`\def\<name>#1..#n{...}`
Application, e.g. $(t\ t)$		`\<name>..\<name_n>`

Table 2: λ-calculus to TEX mapping

This definitely means that TEX incorporates the same power as λ-calculus. And since latter is Turing complete, also TEX is it as well. The ending of [1, Section 5.9] summarises it like this:

> "Interestingly, as we have implemented unbounded lists in TEX's mouth, this means we can implement a Turing Machine. So, if you believe the Church-Turing thesis, TEX's mouth is as powerful as any computer anywhere. Is not that good to know?"

After the discussion about the computability let us focus on a first example which uses the mentioned mapping:

Example 6.2. "First" and "Second" in λ-calculus
Given the following two λ-expressions:

$$\lambda xy.x \text{ as "First" and } \lambda xy.y \text{ as "Second"}$$

In TEX we can define the abstraction like this:

`\def\First#1#2{#1}` and `\def\Second#1#2{#2}`

And here the concrete example with the arguments "a" and "b":

$$\lambda \text{ calculus: } (\lambda xy.x)(a\ b) = a \text{ and } (\lambda xy.y)(a\ b) = b$$
In TEX: `\First{a}{b} = a` and `\Second{a}{b} = b`

Note: The braces around the parameters are used as delimiters (recall the "sqrt" macro in Chapter 3) and have to be specified in this case. Otherwise TEX would interpret command names "Firstab" and "Secondab".

Definition 6.3. *Lists* as known from functional languages are a data structure which consist by two components: the *head* which resembles the first entry and the rest, often referenced as *tail*. The tail, which again a list, is recursively divided into head and tail until the latter is empty. The connector between these components is called *Cons*, the empty list *Nil*.

The mentioned framework provides the most important *list constructors* (Listize, Unlistize, Nil, Cons, Singleton...) and *list operations* (Head, Tail, Fold left, Fold right, Concatenation, Insertion sort...). Some of these we will be discussed further.

Example 6.4. A list with the entries 1, 2, 3, 4, 5 in TEX notation

```
\Cons{1}\Cons{2}\Cons{3}\Cons{4}\Cons{5}\Nil
```

There exists also a constructor *Singleton* which creates a list with only one entry. In TEX this means: `\Cons{...}\Nil`.

Since most functional programming languages also provide a more user-friendly list representation which allows simpler treatment, also Jeffrey's TEX framework does so as well. In our case this means an enumeration of entries separated by colons and the whole enclosed in brackets (consider the next example).

Example 6.5. Interpret a list and return it back

```
\Show\Identity[3,5,2,4] => [3, 5, 2, 4]
```

The important operation is *Show* which takes two parameters: the first one is a list conversion function and the second one the list in the user-friendly format. It runs "Listize" (conversion to internal list representation) then the first parameter operation ("#1") and at the end "Unlistize" (conversion back to beginning) against our list.

```
\def\Show#1[#2] {\Unlistize{#1{\Listize[#2]}}}
```

We are performing the example using the operation *Identity* which means "do no conversion at all".

```
\def\Identity#1{#1}
```

The next step is to analyse *Listize*. Since this is much more tricky we limit ourself to give a rough overview. Basically the definition "Listizea" (helper) is done recursively: the recursion base is a single list entry (parameter 1) and the recursion step is a list with the head (parameter 1) concatenated ("Cons" operation) with the function's recursive call on the list tail. The selection of the case falls in the responsibility of the "TeXif" macro which defines a purpose-customised "if".

```
\def\Listize[#1] {\Listizea#1,\relax]}
\def\Listizea#1,#2] {\TeXif{\ifx\relax#2}%
  {\Singleton{#1}}%
  {\Cons{#1}{\Listizea#2]}}}
```

Unlistize however performs the unrolling. That means it gets the list in the internal (functional) format and performs a *fold right operation*[9] using "Commaize". As the definition of this macro states on each further entry we first

[9] "Fold right" is a right-associative list iterator.

print out a colon, followed by the entry's value (first argument) and the output of the list tail (second argument). In order to recess the use of folding we will also be discussing the operation with the help of Example 6.9. Please notice that the first entry is printed out separately by "Unlistizea" to prevent an introducing colon. And the "Unlistize" definition itself generates the opening and closing brackets.

```
\def\Unlistize#1{[#1\Unlistizea{}]}
\def\Unlistizea#1{#1\Foldr\Commaize{}}
\def\Commaize#1#2{, #1#2}
```

The next example demonstrates a persistent list. As mentioned in Chapter 4 a definition can also be used for data storage purposes – and this is not limited to text strings.

Example 6.6. Saved list

```
\def\list{\Listize[3,5,2,4]}
```

And now it should also be clear how to return the user-readable format:

Example 6.7. Print out a saved list

```
\Unlistize\list => [3, 5, 2, 4]
```

It is time to let us try to perform three basic list operations: "Head", "Tail" and "Reverse".

Example 6.8. Head, Tail and Reverse

```
\Head\list => 3
\Unlistize{\Tail\list} => [5, 2, 4]
\Unlistize{\Reverse\list} => [4, 2, 5, 3]
```

Please notice that "Head" does not require "Unlistize" – it yields only the first entry – and this alone is no list.

Here another folding example but the opposite way: we are using left-associative folding with a sum iterator macro called "sumfun". Contrary to right-associative folding the parameter's meaning is swapped: that means the first is the already handled list part (comparable to the tail) and the second the next list value. As we can observe the "Foldl" call gets as first argument the function, as second the start value (in this case zero – addition's neutral element) and the list. The result is "0+" and all list entries with plus signs between them.

Example 6.9. Folding

```
\def\sumfun#1#2{#1+#2}
\Foldl{\sumfun}{0}\list => 0+3+5+2+4
```

And at the end the most amazing feature: there is even a sorting macro based on *insertion sort*. We have to call "Insertsort" with the comparison function which we prefer to use. "Lessthan" for number ordering is already integrated. "Biggerthan" and similar ones could be added without much effort.

Example 6.10. Use of insertion sort

```
\Unlistize{\Insertsort\Lessthan{\list}} => [2, 3, 4, 5]
```

According the definition of "Insertsort" we base us on a right-hand folding and an insertion function called "Insert" (the argument is the comparator). The input is the list to sort and also the output should be a list as well. The output start value should be empty; hence "Nil".

```
\def\Insertsort#1{\Foldr{\Insert{#1}}\Nil}
```

The question which remains to discuss is about the macro "Insert". Also here we will only be giving an informal description due to place constraints. If the output list is still empty we simply return the actual list entry which is the head. Else we have to analyse the output list. If the value of its head is smaller than the actual list entry's value we have to insert the latter in the tail of the output list – that means head concatenated to a recursive call to "Insert" applied on the tail. In the other case it is easier: we put the list entry's value in front of the returned list and are done.

```
\def\Insert#1#2#3%
 {#3{\Inserta{#1}{#2}}{\Singleton{#2}}}
\def\Inserta#1#2#3#4%
 {#1{#2}{#3}%
  {\Cons{#2}{\Cons{#3}{#4}}}%
  {\Cons{#3}{\Insert{#1}{#2}{#4}}}}
```

These descriptions should be enough to give a rough idea how this framework has been built up and how it can be subject to enhancements.

7 Comparison To Macro Languages From Word-Processing Systems

After finishing the investigations about TEX as a programming language, let us focus our eyes on word processors, spreadsheet managers and desktop databases. What they share is some kind of *programming support* for *automation of periodically performed tasks*, for helping the user *filling out forms* or to allow *interaction with other applications*. In most cases this can be accomplished by two application programming interfaces (APIs):

- A more low-level oriented, not so user friendly one in a diffused programming language as *C, Java and C++*. This is primarily thought for developers of interacting applications.
- An easier, more user friendly one in *BASIC*[10] or another *scripting language*. This is thought for both developers and users.

Since TEX's purpose lies in typesetting we will limit us to perform the comparison using the scripting support of the word processor "OpenOffice Writer" which is called "StarOffice Basic" or also "OpenOffice Basic".[11] The lan-

[10] "BASIC" means "Beginners All-Purpose Symbolic Instruction Code".
[11] First name outlines the proprietary and the second one the open-source variant.

guage is interpreted, procedural and imperative and has had a strong influence of Microsoft BASIC products as "Visual Basic" and "Visual Basic for Applications"[3].

Visual Basic started back in the early nineties of the last millennium as a successor of "QuickBASIC", which itself was derived by the first plain BASIC dialects. Both QuickBASIC and Visual Basic added numerous improvements over the original language, e.g. compilation, enhanced procedures, functions, data types, multi-line control structures and finally also some aspects of Object Oriented Programming (OOP). A variant called Visual Basic for Applications (VBA) is used in Microsoft's Office suites as a BASIC API and therefore the most popular competitor of StarOffice Basic. [12]

We could have presented impressive features as the realisation of GUIs and event-driven programming but there is no real opponent in TEX in this area. Its macro language is mainly thought for document generation. Hence we thought to do the comparison by the *99 bottles of Beer* code. Let us read the original unmodified "99 bottles of Beer" program by Craig J Copi. Commands as "parindent" and "vskip" are typesetting specific and influence the text positioning.

Example 7.1. 99 bottles of beer in TEX

```
% TeX/LaTeX version of 99 bottles of Beer
%%
%% Craig J Copi - copi@oddjob.uchicago.edu
%%
\parindent=0pt
\newcount\beercurr
\def\beer#1{\beercurr=#1\let\next=\removebeer\removebeer}
\def\removebeer{
 \ifnum\beercurr>1
   \the\beercurr\ bottles of beer on the wall,\par
   \the\beercurr\ bottles of beer,\par
   take one down, pass it around,\par
   \advance\beercurr by -1
   \the\beercurr\ bottle\ifnum1<\beercurr{s}\fi\ of beer on the wall.\par
   \vskip 2ex\relax
 \else
   1 bottle of beer on the wall,\par 1 bottle of beer,
   take one down, pass it around,\par no bottles of beer on the wall.\par
   \vskip .5ex
   Time to buy some more beer\dots. \let\next=\relax
 \fi
 \next}

\beer{9}

\bye
```

Now the interesting thing is to simulate this in StarOffice Basic. The following code opens a new OpenOffice Writer window and prints out the same text.

Example 7.2. 99 bottles of beer in StarOffice Basic

```
REM  *****  BASIC  *****

REM StarOffice Basic version of "99 bottles of Beer"
REM
```

[12] Wikipedia, `http://en.wikipedia.org/wiki/Visual_Basic` and references.

```
REM Based on the TeX/LaTeX version written by Craig J Copi
REM Written by Matthias Dieter Wallnoefer for Vertiefungsseminar 2010/11

Sub Main
  Dim Dummy()
  Dim Url As String
  Dim Doc As Object

  Url = "private:factory/swriter"
  Doc = StarDesktop.loadComponentFromURL(Url, "_blank", 0, Dummy())

  Beer(9, Doc, Doc.Text.createTextCursor)
End Sub

Sub Beer(n as Integer, Doc as Object, Cursor as Object)
  If n > 1 Then
    Cursor.String = CStr(n) + " bottles of beer on the wall, "
    Cursor.gotoEnd(False)
    Cursor.String = CStr(n) + " bottles of beer, "
    Cursor.gotoEnd(False)
    Doc.Text.insertControlCharacter(Cursor, _
        com.sun.star.text.ControlCharacter.LINE_BREAK, False)
    Cursor.String = "take one down, pass it around, "
    Cursor.gotoEnd(False)
    n = n - 1
    If n > 1 Then
      Cursor.String = CStr(n) + " bottles of beer on the wall."
    Else
      Cursor.String = "1 bottle of beer on the wall."
    End If
    Cursor.gotoEnd(False)
    Doc.Text.insertControlCharacter(Cursor, _
        com.sun.star.text.ControlCharacter.PARAGRAPH_BREAK, False)
    Beer(n, Doc, Cursor)
  Else
    Cursor.String = "1 bottle of beer on the wall, 1 bottle of beer,"
    Cursor.gotoEnd(False)
    Doc.Text.insertControlCharacter(Cursor, _
        com.sun.star.text.ControlCharacter.LINE_BREAK, False)
    Cursor.String = "take one down, pass it around, no bottles of beer on the wall."
    Cursor.gotoEnd(False)
    Doc.Text.insertControlCharacter(Cursor, _
        com.sun.star.text.ControlCharacter.PARAGRAPH_BREAK, False)
    Cursor.gotoEnd(False)
    Cursor.String = "Time to buy some more beer..."
  End If
End Sub
```

The result: *much more difficult* – and very different to TEX. There we have simply to call the macro where we prefer to have the output and the document compiler expands everything. In StarOffice Basic and also in VBA we have to specify by a "Cursor object" where a certain text has to be written, changed, deleted ecc... – this more or less similar to a plain text cursor. And if the desired editing positions are unknown then certain find operations are available for use.

The different approaches are needed since OpenOffice is "WYSIWYG"-based[13] where TEX is not. An important implication is the fact that StarOffice Basic is capable of modifying existing documents, on TEX this is excluded by construction. On the other hand the document generation support from TEX (including

[13] "WYSIWYG" means "what you see is what you get".

third-party macros and packages) is hardly to be overtaken by Office word processors.

Hence it should be evident that there is no real winner and looser. Both systems embed their different aspects about which the user has to deal with – and it remains him to choose the right technique for the right application.

8 Conclusion

In order to give a summary, let us review the main chapter statements: TEX (and its descendant LATEX) incorporates a *powerful* – but a very *unconventional* – *macro language* (Chapter 4). To get an impression how a simple program does look like we have performed a detailed analysis with the help of the popular *99 bottles of Beer* algorithm in Chapter 5.

The macro language is located in the same formal language class as the *λ-calculus*; therefore recursive enumerable (type 1 in the Chomsky hierarchy). To demonstrate the capabilities on evaluation of Lambda expressions we have made use of a *list framework* by Alan Jeffrey which models lists as known in the world of functional programming languages. We have discussed the possibilities and got an idea how the language is subject to enhancements (Chapter 6).

The last part speaks about macro languages offered in popular *Office suites* which are realised in dialects of *BASIC*. We used StarOffice Basic as a representative and tried to make an expressive comparison to TEX (very hard to achieve) using the mentioned "99 bottles of Beer" code, which generates a document and prints a certain text for a fixed number of iterations. It is absolutely not certain if a system is better than the other one – the fitness has to be evaluated from task to task – message of Chapter 7.

Acknowledgement In particular I would like to thank my supervisor Georg Moser for support and very useful feedback.

8 Conclusion

References

[1] A. Jeffrey. Lists in TEX's Mouth. *TUGboat*, 11, 1990.

[2] D. E. Knuth. *The TEXbook*. Addison-Wesley, 1996.

[3] I. Sun Microsystems. *StarOffice 8 Programming Guide for BASIC*, 2005.