Creating HTML and LaTeXdocuments using UNIX shell scripts

Andreas Harnack

Version 0.9.6
(Chapters 1–6 reviewed.)

August 2024

Abstract

The article describes a way to generate HTML and LaTeX documents using shell scripts and other basic programming tools. It is intended for people with a considerable set of programming skills ready at hand, who want to deploy this skills in an unconventional way. It fosters the idea manifested in the Unix operating systems, to base development on simple, yet flexible tools instead of complex packages. The techniques introduced here are not limited to HTML and and LaTeX documents, they can be used to generate any form of structured documents, and even source code.

Contents

Chapter 1

How it all started – an Introduction

This little project began around 1996, in the dawning days of the Internet. Anyone wanted to have his or her own home page, and so did I. The problem was, there were hardly any HTML editors available at that time, and those that were[1] had apparently entirely misunderstood the idea of hypertext and its mark-up language.

The idea of HTML is to describe the structure of a document, rather then its appearance. The layout is supposed to be left to the browser, according to the technical possibilities it has. This often seems to have been forgotten regarding the amount of formatting information littering the output of even contemporary HTML editors. Most of that stuff I never even specified, it were merely the default setting of the editor I happened to use and hence – from my point of view – more or less random. Admittedly, nowadays I wouldn't insist any more on a Web page being Lynx compatible, but I still prefer to stick to the old idea to specify contents rather then appearance and I want to have control over what actually goes into my documents.

But writing Web pages using a simple text editor is tedious and error prone. So what are possible alternatives? When I was looking for them I came across a utility I never expected to be suitable for that task – it certainly was never intended to be – but it turned out to be quite up to the job: the Unix shell.

That was the moment the geek went back to the terminal and started to type in a few things. The results were astonishing. It took no more then five lines of code to generate a simple *hello world* Web page. With a bit of

[1]most prominently Word for Windows

tweaking and refining I soon got a tool set to describe Web pages as shell scripts. Even after 25 years that code still works. It has been adjusted a bit over the years to makes use of new shell features, to be more readable and maintainable, and obey to new standards, but in the essence it is still the same code and still no more that about 200 lines.

While that tool set is very handy to describe the *structure* of a Web page, it is not well suited to produce longer text. As a non-native speaker publishing mostly in English, I first and foremost missed a spell checker. Also, proofreading a text intermingled with mark-ups is not exactly a pleasure. A standard office suite does a much better job here. Wouldn't it be possible to extract the text, and its features I'm interested in, from an office document and feed it into my scripts? It is. Since version 2.2, I'm using OpenOffice to write the texts for my Web pages, and a Basic macro to transform it into shell scripts. These than can be called from within my script framework. Of course, Basic is usually not my first choice for any programming project, but out of the available APIs of OpenOffice, the Basic API is the best documented one, and that's the most important point here.

At some point the need arose to publish documents not just as HTML, but also in a form intended for printing, i.e. in PostScript or PDF format. TEX as a type setting system with the document preparation macro library LATEX on top of it are my preferred tools for that. LATEX is a mark-up language, in essence very much like HTML. So, is it possible to use the same shell script framework to produce LATEX documents as well? Of course it is. Since the tagging is encode in a kind of function library, all that's required is to load a different library, in order to produce different kind of documents. Technically it's not a problem to get HTML and LATEX out of the same script.[2]

This document has been created using the script framework. For a complete document many other features might be required, like lists, tables, formulas, and even charts. All this can be written in scripts as well as extracted from an OpenOffice document. How this can be done will be described in the upcoming chapters.

[2]Conceptually it is a bit trickier, since HTML and LATEX documents often have a different structure. So at least the top level script is likely to require some adjustments.

Chapter 2

Requirements

Most of the presented ideas are based on the concepts of pipes and filters. These are Unix concepts, so you will need a Unix system. Any Linux distribution will do.

If you, for whatever reason, are bound to use a Windows system, don't panic! There are several options you have. One of them is *Cygwin*[1]. Cygwin is an implementation of the Unix standard library based on Windows system calls. Many Unix applications have been compiled and linked against this library and can now run on Windows. Cygwin also emulates pipes. I used it for a while and anything I tried worked fine. It's a bit annoying, though, to coop with the different text file formats used in Windows and Unix.

Another option you have is to use virtualization. Virtualization allows you to run multiple virtual machines on a single hardware platform. Each virtual machine has its own operating system installed, which can be different from the one of the host. Each Linux distribution I've tried worked flawlessly. You can even run several Linux systems simultaneously. Of course, you need to have enough hardware resources to do that, since the hardware is shared amongst all the systems. In particular you should have enough memory and disk space. However, Linux systems are modest. One Gigabyte of memory and 12 GB of disk space should be enough for a development system. Even low end notebooks have a multiple of this installed nowadays. A solid state disk is a great advantage as well. A mechanical hard disk can easily become a bottleneck when it has to handle the disk-IO of several systems at once. If that's the case for you, you can try to run your virtual machine from an external USB-disk. In my cases that improved performance significantly.

[1]https://www.cygwin.com

To run a virtual machine, you need a virtualization software installed. Originally I've been using *VMware Server*, which worked quite well. However, around 2008 the replacement of the GUI by a web-based user-interface made it really painful to use. As far as I know it has been discontinued anyway without an adequate free replacement. I've been using *Oracle VirtualBox*[2] ever since, which works equally well. It comes under a GNU General Public License and therefore can be used free of charge even for commercial use, apart from an extension pack, which is licenced as freeware for personal use only. But that's not required for our purpose anyway.

After installing VirtualBox, which is straight forward, you can set up and start your virtual machine. It will ask you for an installation medium. You can use a physical DVD or a DVD-image on disk. Linux then installs like on any physical system.

You should have all the Unix standard tools implemented, like `sed`, `sort`, `uniq`, `awk`, etc. That's not an issue on a standard Unix/Linux implementation since they are, well, standard, but you'll have to select many of them individually when installing Cygwin. The same goes for a standard C-compiler. If you have a choice, use the GNU version of `awk`, it has some extension that might come in handy occasionally.

All the shell scripts have been implemented for the *bash* shell. That's the default shell for all Linux distributions I'm aware of. The Korne-shell should work equally well. Even the Bourne-shell is potentially suitable. Unfortunately it doesn't support functions, implying that the code in each function body would have to go into its own executable. That's neither very maintainable nor efficient.

Creating HTML documents with dynamic content doesn't require anything special as far as the document itself is concerned. To see it actually work, however, you'll need a web server with PHP support and a supported database of your choice.

A second central idea in this framework is implemented around *OpenOffice*, so you'll need this as well. Any version above 2.2 should do. If you prefer *LibreOffice*, that should work as well, though I never tested it.

In a heterogeneous environment with Unix and Windows systems you can run OpenOffice smoothly on both platforms to write documents. Exporting works equally well on both platforms, but moving the generated scripts from one system to another raises character encoding and file format issues. Nothing, that can't be solved, but it's avoidable if you transfer OpenOffice

[2]https://www.virtualbox.org/

documents from Windows to Unix instead of scripts. The `Makefile`-based export works probably just on Unix systems.

To produce LaTeX documents you'll obviously need a working TeX system installed on your system, including LaTeX and all the packages, the language support and the fonts required for the kind of documents you want to create. I work with *TeXLive* on Linux and *MiKTeX* on Windows. A tool to transform the TeX output into PDF is convenient. To translate OpenOffice formulas to TeX you'll further need a C++ compiler.

Chapter 3

The Unix Shell Script Framework

Unix shell scripts are programs. Writing Web pages as shell scripts means to write programs that produce HTML output. So, how can we produce HTML output? Let's start with the simplest thing, a single tag:

```
$ function tag { echo "<$@>"; }
$ tag hr
<hr>
$
```

The use of `"$@"` is a bit of shell magic. The effect is to replace it by all parameters passed to the function. This allows to specify tag attributes:

```
$ tag hr width=25% align=left
<hr width=25% align=left>[1]
$
```

For convenience, there is also a version, that allows multiple tags on a line:

```
$ function tags { for i; do echo -n "<$i>"; done; echo; }
$ tags br hr
<br><hr>
$ tags br 'hr width=25% align=left'
<br><hr width=25% align=left>
$
```

[1]Depreciated, should be `<hr style="width:25%;margin-left:0">` nowadays, but still kept here as an example.

HTML documents have a structure of nested blocks. Each HTML document should have at least a HEAD and a BODY block, enclosed into the HTML block. To create blocks, we use the Unix pipe mechanism. The idea is to consider a block creating function as being a filter, that takes some input, encloses it into a block structure and passes it onto the output. Here is a first version:

```
$ function block { echo "<$@>"; sed 's/^/\t[2]/'; echo "</$1>"; }
$ echo hello world | block p
<p>
        hello world
</p>
$ echo hello world | block p align=center
<p align=center>
        hello world
</p>
$
```

The function first creates the opening tag as seen above. Then the input is copied to the output by the `sed`, inserting a tab character at the beginning of each line, so the output will be nested neatly. Finely, the closing tag is appended, this time containing just the first argument, i.e. the tag name.

There are two more variants to create blocks:

```
$ function blck { echo -n "<$@>"; sed "\$s/\$/<\/$1>/"; }
$ function bl { LINE="$1"; shift; echo "<$@>${LINE}</$1>"; }
$ echo hello world | blk p align=center
<p align=center>hello world</p>
$ bl 'hello world' p align=center
<p align=center>hello world</p>
$
```

The first form inserts the opening and closing tags directly at the beginning and at the end of the first and last line respectively, without performing any line shift. This yields to a more compact result and avoids layout problems on the browsers side caused by the additional white space otherwise inserted at the beginning (and the end) of the text. It is intended for paragraphs and similar blocks. The second form produces the same result but takes the input from the command line, instead of the standard input. This is handy if you want specify a short output text directly in the script, like a headline for example.

Armed with this five simple functions we can go ahead and provide some more complex stuff. As mentioned above, each valid HTML document has

[2]The syntax to get a tab in there might vary, depending on your shell.

at least three nested blocks, so here is a filter, that takes any arbitrary piece of text and wraps it into the correct form:

```
$ function html {
>       (
>               bl "${1:-no title}" title | block head;
>               shift;
>               block body "$@";
>        ) | block html;
>}
$
$ bl 'hello world' p | html 'My first HTML page' 'bgcolor=lightgray'
<html>
        <head>
                <title>My first HTML page</title>
        </head>
        <body bgcolor=lightgray>
                <p>hello world</p>
        </body>
</html>
$
```

Et voila, only 12 lines of code and we have a result that gets very close to a compliant HTML page.

The first argument passed to the function is going to be the document title, something that's required by the standard. If there is no argument, *'no title'* is used instead. The title text is enclosed into the TITLE block and becomes the only element of the HEAD block. The `shift` command removes the first parameter from the parameter list – we just used it as title –, all remaining parameters (if any) become attributes of the BODY block. Note the use of parentheses in contrast to curly braces. While curly braces are used to group command execution within the same execution thread – very much like the compound statement in C does –, the code in parentheses is executed in its own sub-process. This allows to group the output of several commands and feed it into another pipeline, before it is included into the output of the main thread.

A productive version of an `html` filter will provide some more functionality. It's considered to be good practice to leave a copyright note, a contact address and the modification date somewhere on each page. This goes into the `html` filter, so you'll never have to worry about it again:

```
$ cat html
#!/bin/bash

source html.include

function trailer {
        echo "&copy; $(ln "$USER" "$(mailto "$MAILTO" "${MAILOFF:-32}")"),"
        LANG=en_GB date "+%e %B %Y" \
        | awk '
                BEGIN   {
                        ext[0]="th";
                        ext[1]="st";
                        ext[2]="nd";
                        ext[3]="rd";
                }

                {
                        i = $1%10;
                        if ( i>3 || i+10==$1 ) i = 0;
                        print ($1 ext[i], "of", $2, $3);
                }
        '
}

echo '<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">'
(
        (
                bl "${1-no title}" title
                test -n "$BASE" && tag base "href='$BASE'"
                test -n "$ICON" && tag link "rel='icon'" "href='$ICON'"
                test -n "$CSS" && tag link "rel='stylesheet'" \
                                        "type='text/css'" "href='$CSS'"
                bl '' script "type='text/javascript'" "src='mailto.js'"

        ) | block head

        shift

        (
                cat -
                echo
                tag hr
                trailer | block address

        ) | block body "$@"

) | block html

$
```

The filter has been put into its own file, so it can be called from the command line. All the helper functions went into an include file and are loaded with the `source` command.

The `trailer` function produces the copyright note, containing an email address and the modification date. The email address is encoded to make it harder for spammers to harvest it. Decoding is done by JavaScript. The decoding function essentially just shifts the code points of a string by a certain offset:

```
function decChar(n, m, off) {
        n = n - off;
        while ( n > m ) n = n - m;
        while ( n < 0 ) n = n + m;
        return n;
}

function decode(str, off) {
        var code = "ABC ... XYZabc ... wxyz0123456789@:.";
        var dec = "";
        for ( var i=0; i < str.length; i++ ) {
                var n = str.charAt(i);
                for ( var j=0; j < code.length; j++ ) {
                        if ( n == code.charAt(j) ) {
                                off = (106 * off + 1283) % 6075;
                                n = code.charAt(decChar(j, code.length, off));
                                break;
                        }
                }
                dec += n;
        }
        return dec;
}
```

That works because spam has to be sheep. A harvester could easily run the decoding script to get the address, but that increases the costs in terms of computation time, and that scales. As far as I know, no harvester has bothered yet to invest that time.[3]

Another JavaScript function calls the users mail agent and passes the decode email address to it:

```
function mailto(str, off) { location.href = "mailto:" + decode(str, off); }
```

All the JavaScript code goes into a file `mailto.js` which will be included in the HEAD block.

[3]And if they should, it's easy enough for us to increase the costs to any extend we'd like. :-)

The little bash helper called from within the trailer function compiles the correct JavaScript call:

```
function mailto() {
        echo "javascript:mailto($(dquote "$(encode "-$2" "$1")"),"$2")"
}
```

encode is a simple command line utility providing the inverse function to decode above, this time implemented in C. It encodes the users email address passed as argument using the offset passed as an option. The resulting string is put in double quotes by dquote – the use of that function avoids ugly escapes – and, together with the offset, compiled as parameter into a JavaScript call. That call becomes the target of a link, so that pressing the link triggers the decoding and the launch of the mail agent.

The ln function produces the correct link tag syntax:

```
function ln {
        text="$1"; shift; href="$1"; shift
        bl "${text}" 'a' "href='${href}'" "$@"
}
```

The current date for the trailer is retrieved from the systems date command and filtered through an awk program to get the output a bit nicer. The logic in the awk program works only for English but the output of date will depend on the users language settings. Therefore the language for date is specified explicitly. That ignores the user settings but makes sure the logic doesn't break if the user uses a different language. The solution is good enough for me, it can easily be adjusted if more flexibility is needed.

The main part adds the DOCTYPE comment, recommended by the standard to inform the browser about the encoding scheme used for the document. The header specifies the document title and the location of the email decoding script. It also demonstrates a technique to include additional information into the header, based on the existence and contents of some variables. The BODY part just copies the document contents from the standard input and appends the trailer, separated by a horizontal ruler.

The script can complete any content formatted as an HTML fragment into a valid HTML page, completed with some stuff that each page should have. If you want more on any of your pages you can easily extend it.

The variables used in the script can be defined in the user's environment settings, in a superordinate script, a makefile or wherever it might be convenient. Sometimes it's most convenient to pass them as options, overriding values that might have been set elsewhere. That's achieved easily:

```
while test "$1"; do
        case "$1" in
                -b|--base)      shift; export BASE="$1"; shift;;
                -c|--css)       shift; export CSS="$1"; shift;;
                -i|--icon)      shift; export ICON="$1"; shift;;
                -u|--user)      shift; export USER="$1"; shift;;
                -m|--mailto)    shift; export MAILTO="$1"; shift;;
                -o|--offset)    shift; export MAILOFF="$1"; shift;;
                -h|--help)      less "$0"; exit 0;;
                --)             shift; break;;
                *)              break;;
        esac
done
```

That loop goes on top of the script. It iterates over the command line parameters left to right, and for each known option the next parameter in line is assigned to the respective variable. The default case at the end terminates the loop if a parameter can't be recognized as an option. The next to last case can explicitly enforce loop termination. That's useful if a known option shall be passed to a subordinate script, which is not the case for the `html` script but kept here for consistency. Please note also the `--help` option. That simply lists the code of the script itself, being a very simple way to get the most accurate help text possible: the code itself.

It's also good practice to test for variables that are not optional. That avoids unexpected results and can spare you a lot of headaches. It's done after the options have been evaluated:

```
test "$USER" || undefined USER 'site maintainer'
test "$MAILTO" || undefined USER "site maintainer's email address"
```

The error handling functions are defined in `html.include`:

```
function error { echo "$0: Error: $@" 1>&2; exit 1; }
function undefined { error "'$1' ($2) undifined"; }
```

It's also possible to define default values:

```
test "$MAILOFF" || export MAILOFF=32
```

The use of the *or*-operator instead of an *if*-clause is a very handy abbreviation, relying on the short-cut evaluation of logical expressions. If the left hand side yields true, the result of the overall expression is already set, so there is no need to evaluate the right hand side; therefore it's simply skipped. Only if the left hand side yields false, the right hand side can have an impact on the result and therefore only then it is evaluated. However, it's the side effects we're interested in, the final result is discarded. That trick works in many imperative languages.

You might find it useful to have several different page formats available, depending on the purposes of a page. Since filters can be concatenated to any length, this is not a problem. I have a filter to generate pages intended to be published on *sourceforge.net*. The site requires each page to display the *sourceforge* logo together with a link to the *SourceForge* home page somewhere on the page. I decided to put it on top:

```
$ cat sf
#!/bin/bash

source html.include

(
        (
                if test -n "$SF_TOC"; then
                        cat "$SF_TOC" | while read href title; do
                                if test "$title" = "$WWW_TITLE"; then
                                        bl "$title" span "class='nav-item'"
                                else
                                        ln "$title" "$href" "class='nav-item'"
                                fi
                                echo
                        done
                fi

                (
                        bl 'hosted by:' em
                        img "$SF_LOGO" "$SF_NAME" 'SFLogo' |
                                lnk "$SF_LINK" "rel='nofollow'"

                ) | blk span "style='float:right;'"

        ) | block div "class='nav'"

        tag hr
        echo
        cat

) | html "$WWW_TITLE "$@"

$
```

Apart from the two helper functions `img` and `lnk` there's nothing new in here. Like `ln`, so does `lnk` produce a link tag. The difference between `ln` and `lnk` and is the same as between `bl` and `blk`:

```
function lnk {
        href="$1"; shift
        blk 'a' "href='${href}'" "$@"
}
```

The `img` function creates an image tag, taking an image source reference and an alternative descriptive text. An optional third parameter is interpreted as identifier:

```
function img {
        src="$1"; shift
        alt="$1"; shift
        if test "$1" = ''; then
                tag img "src='${src}'" "alt='${alt}'" "$@"
        else
                id="$1"; shift
                tag img "id='${id}'" "src='${src}'" "alt='${alt}'" "$@"
        fi
}
```

The `sf` script as a whole places a navigation section on top of its input, separated by a horizontal ruler, before the so extended content is turned into a complete page by `html`. The navigation section contains at least the *SourceForge* logo referring to the *sourceforge.net* project page. Optionally a table of content can be specified. If the variable `SF_TOC` is set, its value is taken as the name of file, containing the reference and title of each page to be referenced. Each title becomes a navigation item in the navigation section pointing to the specified reference, unless the title equals to the title of the page being process, in that case the link is skipped. If `SF_TOC` is specified, `WWW_TITLE` is required as well. That's checked easily:

```
test "$SF_TOC" && { test "$WWW_TITLE" || error "'SF_TOC' requires 'WWW_TITLE'"; }
```

The remaining compulsory parameters can be checked the usual way:

```
test "$SF_NAME" || undefined SF_NAME 'project name'
test "$SF_LINK" || undefined SF_LINK 'project link'
test "$SF_LOGO" || undefined SF_LOGO 'project logo'
```

And, of course, it's convenient to add command line options:

```
while test "$1"; do
        case "$1" in
                -p|--name)      shift; export SF_NAME="$1"; shift;;
                -l|--link)      shift; export SF_LINK="$1"; shift;;
                -i|--logo)      shift; export SF_LOGO="$1"; shift;;
                -c|--toc)       shift; export SF_TOC="$1"; shift;;
                -t|--title)     shift; export WWW_TITLE="$1"; shift;;
                -h|--help)      less "$0"; exit 0;;
                --)             shift; break;;
                *)              break;;
        esac
done
```

Here is an *'hello world'* example output:[4]

```
$ echo hello world | sf -p DocScript  -t 'Layout Guide' -c docscript.toc \
> -l https://sourceforge.net/projects/docscript/ \
> -i 'https://sourceforge.net/sflogo.php?type=11&group_id=28364' \
> -- -c docscript.css
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
       <head>
               <title>Layout Guide</title>
               <link rel='stylesheet' type='text/css' href='docscript.css'>
               <script type='text/javascript' src='mailto.js'></script>
       </head>
       <body>
               <div class='nav'>
                       <a href='overview.html' class='nav-item'>Home</a>
                       <a href='config.html' class='nav-item'>Configuration</a>
                       <span class='nav-item'>Layout Guide</span>
                       <span style='float:right;'><em>hosted by:</em>
                       <a href='...' rel='...'><img id='...' src='...' alt='...'></a></span>

               </div>
               <hr>

               hello world

               <hr>
               <address>
                       &copy; <a href='javascript:mailto("...",32)'>Andreas Harnack</a>,
                       10th of December 2021
               </address>
       </body>
</html>
$
```

[4]Some parameters skipped to fit line length.

assuming that:

```
$ cat docscript.toc
overview.html   Home
config.html     Configuration
layout.html     Layout Guide
$
```

Most likely you will have a dedicated script for each of your projects, specifying all of the project specific parameters:

```
$ cat docscript.sh
#!/bin/bash

export SF_NAME='DocScript'
export SF_LINK='https://sourceforge.net/projects/docscript/'
export SF_LOGO='https://sourceforge.net/sflogo.php?type=11&group_id=28364'
export SF_TOC='docscript.toc'
export CSS='docscript.css'

sf "$@"

$
```

The call

```
$ echo hello world | docscript.sh -t 'Layout Guide'
```

would yield the same result.

Chapter 4

Getting the Content

In the previous chapter we've developed some tools to create standard conforming HTML pages (`html`) and give them a specific layout (`sf`). However, we do not yet have any content for our pages, without any content piped through our filters we'll only get empty pages.

So how can we get the content? So far, we have only seen plain text, but the content needs to be formatted in HTML. Our filters will only nest it neatly, not format it in any way. This leaves the question, where more sophisticated content might come from.

Actually we've already seen a possible way. Anything between the BODY tags is technically speaking content. The trailer and the navigation section in the examples above are content. It has been created using the same tools as for the document framework. The data contained was either

- hard coded into the script (like the name of the `mailto.js` script),
- taken from a script parameter (like the document title),
- taken from an environment variable (like the name of the style sheet),
- taken from a file (like the table of content), or
- produced by a program (like the current date).

All that data was either simple or structured in the sense, that it was either single a data item (like the date or document title), or simple list and table (like the table of content). Our tools are well suited for this kind of data. But web pages will also contain data with a more complex structure as well as running text. That will require different techniques. In the next chapter we'll discuss a way to produce running text of any kind and length. Here we'll look at a technique to put more complex data or short text directly into a script.

4.1 Here Documents

Many shells, including the *bash*, offer a feature that comes in handy for our purpose: the *here document*. Technically it is a form of redirection, that feeds a part of the script content itself as input to a command or function within the same script.

Data and representation are still separated, but not so strictly as in different files. Keeping both in the same file makes it often easier to maintain either of them. Here is the syntax:

```
command [n]<<[-] delimiter
        data
delimiter
```

The *delimiter* can be any string. The shell copies data, starting from the next line, into *command*, until it finds a line containing only the *delimiter*, then normal command execution is resumed:

```
$cat here.sh
#!/bin/bash

cat << .
Line 1
Line 2
Line 3
.

echo '### done ###'

$ here.sh
Line 1
Line 2
Line 3
### done ###
$
```

I often just use a dot as delimiter, like the SMTP protocol does, since it is fairly unlikely to find a single dot alone on a line in any real text. However, any other string would do as well. The optional *n* before the `<<` specifies the file descriptor, the data is fed to. The default is standard input, which is usually what we want, so we can skip it.

Parameter expansion, command substitution, and arithmetic expansion are applied to all input lines, before they are passed on:

```
$ cat here.sh
#!/bin/bash
cat << .
Line 1 $SHELL
Line 2 $(date)
Line 3 $((2+3))
.
$ here.sh
Line 1 /bin/bash
Line 2 Mon 13 Dec 23:40:58 CET 2021
Line 3 5
$
```

Quoting the delimiter or any character in it will suppress expansion:

```
$ cat here.sh
#!/bin/bash
cat << '.'
Line 1 $SHELL
Line 2 $(date)
Line 3 $((2+3))
.
$ here.sh
Line 1 $SHELL
Line 2 $(date)
Line 3 $((2+3))
$
```

That's a convenient way to include literal content into a script, especially if the content stretches over several lines, like running text:

```
$ cat example
#!/bin/bash

source html.include

blk p << .
Lorem ipsum dolor sit amet, consectetur adipisici elit,
sed eiusmod tempor incidunt ut labore et dolore magna aliqua.
.

$
```

```
$ example
<p>Lorem ipsum dolor sit amet, consectetur adipisici elit,
sed eiusmod tempor incidunt ut labore et dolore magna aliqua.</p>
$
```

It even allows to use some of the tools we've defined in the previous chapter:

```
$ cat example
#!/bin/bash

source html.include

blk p << .
$(ln 'Lorem ipsum' 'myref') dolor sit amet, consectetur adipisici
elit, sed eiusmod tempor incidunt ut labore et dolore magna aliqua.
.

$ example
<p><a href='myref'>Lorem ipsum</a> dolor sit amet, consectetur adipisici
elit, sed eiusmod tempor incidunt ut labore et dolore magna aliqua.</p>
$
```

The `<<-` version of the redirect operator stripes away all leading tabs from the input lines, including the delimiter line, before passing it on. That allows to nest the literal input in a natural way:

```
$ cat example
#!/bin/bash

source html.include

(
        bl 'Lorem ipsum' h1

        blk p <<- .
                $(ln 'Lorem ipsum' 'myref') dolor sit amet,
                consectetur adipisici elit, sed eiusmod tempor
                incidunt ut labore et dolore magna aliqua.
        .

) | html 'Lorem ipsum'

$
```

```
$ example
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
        <head>
                <title>Lorem ipsum</title>
                <script type='text/javascript' src='mailto.js'></script>
        </head>
        <body>
                <h1>Lorem ipsum</h1>
                <p><a href='myref'>Lorem ipsum</a> dolor sit amet,
                consectetur adipisici elit, sed eiusmod tempor
                incidunt ut labore et dolore magna aliqua.</p>

                <hr>
                <address>
                        &copy; <a href='javascript:mailto("...",32)'>...</a>,
                        14th of December 2021
                </address>
        </body>
</html>
$
```

4.2 From Data to Content

Armed with this techniques we can create nearly any content we want. As an example lets assume we want to create a questionnaire. The questionnaire shall present a list of questions with a possible choice of answers. We start by just sketching down a first draft:

```
$ cat bridgekeeper
1. What is your name?
a) Sir Lancelot of Camelot
b) Sir Robin of Camelot
c) Sir Galahad of Camelot
d) It is 'Arthur', King of the Britons

2. What is your quest?
a) To seek the Holy Grail.

3. What is your favourite colour?
a) Blue
b) No, yellow

$
```

That's not yet a script, just an ordinary text file, though already structured in a way suitable for what we have in mind. It contains the questions, one at a line, followed by the possible answers and a blank line as a separator. Each data line starts with a number or letter, which later will serve as an identifier.

Now let's transform the data file into a script:

```
$ cat bridgekeeper
#!/bin/bash

function questionnaire { cat; }

questionnaire << '.'    # no expansion required
1. What is your name?
a) Sir Lancelot of Camelot
b) Sir Robin of Camelot
c) Sir Galahad of Camelot
d) It is 'Arthur', King of the Britons

2. What is your quest?
a) To seek the Holy Grail.

3. What is your favourite colour?
a) Blue
b) No, yellow

.

$
```

That skeleton doesn't do anything more then the simple `cat` command yet, bit it gives us the function `questionnaire`, that can now be filled with the formatting functionality we want.

Next we have to retrieve the structure of the text input. The questionnaire contains of two nested lists, so the we'll see two nested loops in the script:

```
function questionnaire {
        while IFS='. ' read question questiontext; do
                echo "($question) >$questiontext<" 1>&2
                while IFS=') ' read answer answertext; do
                        test "$answer" || break
                        echo "  [$answer] >$answertext<" 1>&2
                done
        done
}
```

Most of this is shell magic again: The `read` command reads a line of input, splits it into columns and assigns the values to the variables passed in the argument list in the order of appearance. If there are more columns then arguments, the process will stop before the last argument and all the rest of the line goes into that single remaining variable. If there are fewer columns, the remaining variables will hold the empty string. One variable is mandatory. We have two variables here, the first gets the identifier, the second the rest of the line, which is the question respectively answer. The input field separator `IFS` has been chosen to eliminate characters from the input intended for the human reader only. That includes white spaces before the beginning of the text.

The outer loop will read the first line of input, containing the first question. The following lines, containing the answers to the current question, are consumed by the inner loop. The first empty line terminates the inner loop, so the next line, representing the next question, will be consumed by the outer loop again. The whole process stops if there's no more input available. This is simple and straight forward, though rather sensitive to errors: There has to be exactly one empty line at the end of the input and between the questions, anything else will lead to unexpected results. The same applies to the character separating the columns in the data, if they do not exactly match the `IFS`. That's quite OK for our purposes, it's far easier to fix the input rather that making the analysis unnecessary complex.

The print commands are just a simple way to check the correct analysis of the input text:

```
$ bridgekeeper
(1) >What is your name?<
  [a] >Sir Lancelot of Camelot<
  [b] >Sir Robin of Camelot<
  [c] >Sir Galahad of Camelot<
  [d] >It is 'Arthur', King of the Britons<
(2) >What is your quest?<
  [a] >To seek the Holy Grail.<
(3) >What is your favourite colour?<
  [a] >Blue<
  [b] >No, yellow<
$
```

Note that they are redirected to `stderr`, so the dumps won't go into the result and hence they can remain there even in a productive version. That is a nice possibility to show the progress of the formatting and comes quite handy for longer documents.

Having the input analysed successfully it's time to format it. A questionnaire is a list of questions, so lets start with creating a list and the corresponding entries. We choose an ordered list with numerically numbered entries[1]:

```
source html.include

function questionnaire {
        while IFS='. ' read question questiontext; do
                (
                        bl "$questiontext" div "class='question'"
                        while IFS=') ' read answer answertext; do
                                test "$answer" || break
                        done
                ) | entry li
        done | block ol type=1
}
```

Note, there's an additional pair of grouping parentheses required around the code for a list entry. This is because the answers text is going to be part of the entry. The `entry` function works very similar to the `blk` function, but it indents all but the first line:

```
function entry      { echo -n "<$@>"; sed "1!s/^/\t/; \$s/\$/<\/$1>/"; }
```

This is best when having single-line entries and provides still fairly readable results in case of multi-line entries:

```
$ bridgekeeper
<ol type=1>
        <li><div class='question'>What is your name?</div></li>
        <li><div class='question'>What is your quest?</div></li>
        <li><div class='question'>What is your favourite colour?</div></li>
</ol>
$
```

The `div` tag is there to assign a name to a part of the document, that is than used to specify a layout:

```
.question { margin-top: 1em; margin-bottom: 1em; font-weight: bold; }
```

That can be done in a style sheet file or within the final document between the `<style></style>` tags in the document header. So far we've only considered the style sheet file.

[1]The dump output has been removed temporarily. It would mix on the console with the output on *stdout*, which can be confusing. Parts that have been added are in italic.

Note, that the identifiers assigned to the questions are not part of the document. Instead, the browser will create its own item numbering. That's actually a good thing. You are likely to store the user response in a database or something similar. There the questions will be identified by the identifiers provided with the input text. Let's assume you want to remove a question or reorder them for some reason. In that case the reader will still see a properly numbered list without you having to reorder your database.

Adding the corresponding answers is straight forward:

```
function questionnaire {
        while IFS='. ' read question questiontext; do
                (
                        bl "$questiontext" div "class='question'"
                        while IFS=') ' read answer answertext; do
                                test "$answer" || break
                                bl "$answertext" li
                        done | block ol type=a
                ) | entry li
        done | block ol type=1
}
```

Now we can see the effect of the `entry` function:

```
$ bridgekeeper
<ol type=1>
        <li><div class='question'>What is your name?</div>
                <ol type=a>
                        <li>Sir Lancelot of Camelot</li>
                        <li>Sir Robin of Camelot</li>
                        <li>Sir Galahad of Camelot</li>
                        <li>It is 'Arthur', King of the Britons</li>
                </ol></li>
        <li><div class='question'>What is your quest?</div>
                <ol type=a>
                        <li>To seek the Holy Grail.</li>
                </ol></li>
        <li><div class='question'>What is your favourite colour?</div>
                <ol type=a>
                        <li>Blue</li>
                        <li>No, yellow</li>
                </ol></li>
</ol>
$
```

There is no line break or white space after the `<li>` tag since then the list entry would technically start with a space.

To get a fully featured web page, just pipe it through the `html` script as described above and watch the result:

```
$ bridgekeeper | html -c bridgekeeper.css bridgekeeper
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
      <head>
            <title>bridgekeeper</title>
            <link rel='stylesheet' type='text/css' href='bridgekeeper.css'>
            <script type='text/javascript' src='mailto.js'></script>
      </head>
      <body>
            <ol type=1>
                  <li><div class='question'>What is your name?</div>
                        <ol type=a>
                              <li>Sir Lancelot of Camelot</li>
                              <li>Sir Robin of Camelot</li>
                              <li>Sir Galahad of Camelot</li>
                              <li>It is 'Arthur', King of the Britons</li>
                        </ol></li>
                  <li><div class='question'>What is your quest?</div>
                        <ol type=a>
                              <li>To seek the Holy Grail.</li>
                        </ol></li>
                  <li><div class='question'>What is your favourite colour?</div>
                        <ol type=a>
                              <li>Blue</li>
                              <li>No, yellow</li>
                        </ol></li>
            </ol>

            <hr>
            <address>
                  &copy; <a href='javascript:mailto("...",32)'>...</a>,
                  14th of December 2021
            </address>
      </body>
</html>
$
```

We have demonstrated a simple way to convert a text file into a properly formatted nested list. The technique is restricted neither to lists nor to literal text. Tables can be created in a similar way and any command providing line oriented output can serve as a data source. This holds in particular for scripts written in any other scripting language. *Perl* for example offers excellent opportunities to access relational data bases and format the results into a form suitable for processing the way we showed.

4.3 Making the Content Dynamic

Having the questionnaire nicely formatted as an HTML document, is the next step to have it answered and evaluated online. To do so requires the questionnaire to be transformed into an INPUT form to allow the user to enter input, as well as some means to process that input once the user presses the submit button. The former is standard HTML, the later depends on the options your internet service provider offers.

Let's start with the former one. Each possible answer needs to get an input radio button attached to it. Further, we need a submit button to send, and optionally a reset button to clear the form. Finally, the whole form is going to be enclosed in an FORM tag pair. All that is not much hassle, with only one small exception which we postpone for a moment:

```
function questionnaire
{
        (
                while IFS='. ' read question questiontext; do
                        (
                                bl "$questiontext" div "class='question'"
                                while IFS=') ' read answer answertext; do
                                        test "$answer" || break
                                        bl "$(tag input type=radio \
                                                "name='${question}'" \
                                                "value='${answer}'" \
                                        ) $answertext" li
                                done | block ol type=a
                        ) | blk li
                done | block ol type=1
                tag input type=submit "value='Submit'" "name='submit'"
                tag input type=reset "value='Reset'"
        ) | block form method=get "action='...'"
}
```

We won't list the output, since it's getting rather long (and wide!) but go ahead and try it out! Note that the submit button has an attribute called *name* attached to it. We'll come to that in a moment.

The next thing to decide is the value for the *action* attribute of the FORM tag. The *action* attribute specifies what to do with the user input. Its value is the path to a script that is executed with the input fed to it. The choice you have here, as mentioned, mainly depends of your ISP.

In my case I found that the best choice would be to use PHP. PHP is a scripting language intended for producing web pages dynamically on the server side. A PHP script has a C-like syntax and is typically embedded into an HTML document, enclosed in the special delimiters `<?php ... ?>`. The web server parses the document and executes any script code it may find.[2] The script code is removed from the document and replaced by the output it produces. The code itself is never be seen by the browser. An HTML page containing PHP code can process user data, so it can be the target of the *action* attribute of the FORM tag.

To insert PHP code into our documents we start again with a simple function. This, as well as all the following PHP-related functions, goes into a dedicated include file `php.include`:

```
function php {
        if test "$1"; then
                echo "<?php $@ ?>"
        else
                echo "<?php"; sed 's/^/\t/'; echo "?>"
        fi;
}
```

The function takes the PHP code from the command line arguments, if provided there, otherwise from standard input. Note, that the two branches of the *if*-statement are very similar to the block functions `bl` and `block`.

When the user submits the input, the questionnaire should be re-presented to the user with either an acknowledgement or an error message. We can achieve this by sending the input data to the page itself, rather then some different scripts. This way the user will get the feedback on the same page, together with the option to correct and eventually re-submit it. Hence, the *action* attribute of the FORM tag needs to point to the location where the current document is stored. We might not know that yet, but the web server will know when it delivers the document and we have a way to find that out:

```
(
        ...
) | block ... "action='$(php 'echo htmlspecialchars($_SERVER["PHP_SELF"]);')'"
```

This procduces:

```
<form ... action='<?php echo htmlspecialchars($_SERVER["PHP_SELF"]); ?>'>
        ...
</form>
```

The `$` sign marks a variable in PHP. `$_SERVER` is an associative array providing server related information, the string `"PHP_SELF"` is an index into

[2]For this to work the document needs to have the file name extension `.php` instead of `.html`.

that array providing the path to the current document. That path is simply echoed into the document sent to the browser, filling in the value for the action argument.

The `htmlspecialchars()` function converts characters with a special meaning in HTML like `<` and `>` into their HTML entities `&lt;` and `&gt;`. This prevents cross-site scripting attacks. Without this precaution attackers could use the URL to inject malicious HTML or PHP code into the page.[3]

When the questionnaire is re-presented to the user we should make sure, that all selections already made are preserved. The HTML radio button provides a *checked* attribute, that when present causes the button to be preselected. The information, if a button is selected, can be retrieved by another associative array, `$_GET` in our case, since we specified the *HTTP* GET method in the FORM tag. This array will provide all input values, indexed by the *name* attribute associated to the input elements. To create an expression to retrieve the values, we use another little function:

```
function php_env        { echo '$_'"$1['$2']";[4] }
```

Since we're on it, let's create a few more helper functions to create PHP code, all of them, as mentioned, go into `php.include`:

```
function php_block      { echo "$@" '{'; sed 's/^/\t/'; echo '}'; }
function php_cmd        { echo "$@;"; }
function php_var        { echo '$'"$1"; }

function php_if         { php_block if "( $1 )"; }
function php_elseif     { php_block else if "( $1 )"; }
function php_else       { php_block else; }
function php_foreach    { php_block foreach "( $1 )"; }

function php_assign     { php_cmd "$(php_var "$1") = $2"; }
function php_print      { php_cmd "print($@)"; }

...
```

[3]Let's assume our URL is *http://www.mysite.org/bridgekeeper.php*. Then our form generated would be `<form ... action='/bridgekeeper.php'>`. But now assume, an attacker appends the string */%27%3E%3Cscript%3Ealert('bang')%3C/script%3E* to that URL: `<form ... action='/bridgekeeper.php/'><script>alert('bang')<script>'>`!

[4]This are two concatenated strings: The first string in single quotes `'$_'` is passed unchanged to the result string, while the double quoted string is subject to parameter substitution. The single quotes within the double quotes, however, are literals and passed unchanged to the result too.

With this filters we can generate, amongst other things, any PHP control structure.[5] Just the notation is a bit unusual, since first comes the block, then the condition. But it fits nicely into the document script and makes sure we have the right syntax.[6]

With this at hand it's easy to write a simple test, that inserts the *checked* attribute whenever an answer has already been selected:

```
function is_checked {
        php_print '"checked"' \
        | php_if "isset($(php_env GET "$1")) && $(php_env GET "$1") == '$2'" \
        | php
}
```

Here is an example of the output it produces:

```
<?php
        if ( isset($_GET['1']) && $_GET['1'] == 'a' ) {
                print("checked");
        }
?>
```

The name of a radio button appears as index in `$_GET` only if one of the buttons with that name has been selected. The `isset()` functions is there to handle this situation gracefully, since accessing an undefined variable would lead to a PHP notice in the web servers error log.[7]

This piece of code needs to be added to each possible answer. A nightmare, if to be done manually, for any non trivial questionnaire. But we are doing scripting. All we have to do is to call this function as an additional attribute whenever an input radio button is created:

```
bl "$(tag input type=radio "name='${question}'" "value='${answer}'" \
        "$(is_checked "$question" "$answer")") $answertext" li
```

That's not even a whole additional line of code plus four lines for the check function, creating three lines of PHP code per answer.[8]

[5]In fact, we can generate the control structures of any programming language. That opens a whole range of meta-programming opportunities including the chance to specify algorithms in a language independent way. To change the target language, just change the filters. That works for all languages with a similar set of control structures, which applies to nearly all traditional imperative languages. However, things are getting fishier as soon as expressions are involved.

[6]I'm not exactly an expert in PHP and hence not too familiar with the syntax. I just looked it up and put it into the filter. It worked.

[7]The `&&` operator is a shortcut operator, remember? The right hand side is evaluated only if the left hand side succeeded.

[8]I'm not counting lines containing only a closing delimiter.

What's left to do is providing feedback to the user. For this we need to know, whether the document is going to be the empty questionnaire initially presented to the user, or an acknowledgement to some submitted input. That's were the *submit* buttons *name* attribute comes in. To distinguish between a blank questionnaire sent initially and annotated questionnaire sent as a response it's sufficient to check for the existence of the submit button's name; it will be present only if the user submitted some input and hence the document is sent as a response.

We've already met the `isset()` function for such cases. The skeleton of our response creating function might look like this:

```
function response
{
        (
                ...
        ) | php_if "isset($(php_env GET submit))" | php
}
```

This will produces the following code:

```
<?php
        if ( isset($_GET['submit']) ) {
                ...
        }
?>
```

The code within this code block can create output only in response to the user pressing the submit button. In the initial document that part will be empty.[9]

Once we know we're dealing with a user's response we need to process the input. How exactly that is done depends on our intention. Let's assume we want to conduct a user survey. That means, the answers will be recorded for later statistical analysis. (OK, the questions in the example above don't really fit, but they are easily enough to change, aren't they?)

The first thing we should probably do is to perform some consistency checks. How many and what checks should be performed depends on the nature of the questions we asked. To make life simple we could just sketch down a list of error conditions, each together with a corresponding error

[9]By the way, if your using the POST method instead of GET, you could check for the submit method as well: `if ($_SERVER["REQUEST_METHOD"] == "POST")`. This, however, will work for POST only, since that variable `$_SERVER["REQUEST_METHOD"]` seems to default to GET.

message, and have a script generating the required code. What I have in mind looks something like this:

```
response bridgekeeper << .
Please specify your name!	!isset($(php_env GET 1))
Please specify your quest!	!isset($(php_env GET 2))
Please specify your favourite colour!	!isset($(php_env GET 3))
```

Here we have the error message on the left, the condition triggering it on the right, separated by a tab. The conditions say it's an error if the answer to one of the questions 1, 2 or 3 respectively is the empty string. The numbers are the ones we assigned to the questions in our question list. They became names of a radio button group in the questionnaire's input form and appear as indices in the response array.

We can iterate through the list of error conditions and create a piece of code for each entry to check if that condition is met. If it is, the error message is presented to the user. It would be annoying for the user to have to correct and acknowledge each error individually. It's better to present all error messages at once, therefore they are collected in an array:

```
function response
{
	(
		php_assign errors 'array()'
		while IFS=$'\t' read message condition; do
			php_assign 'errors[]' "'$message'" | php_if "$condition"
		done
	) | php_if "isset($(php_env GET submit))" | php
}
```

Here is the result:

```
<?php
	if ( isset($_GET['submit']) ) {
		$errors = array();
		if ( !isset($_GET['1']) || $_GET['1'] == '' ) {
			$errors[] = 'Please specify your name!';
		}
		if ( !isset($_GET['2']) || $_GET['2'] == '' ) {
			$errors[] = 'Please specify your quest!';
		}
		if ( !isset($_GET['3']) || $_GET['3'] == '' ) {
			$errors[] = 'Please specify your favourite colour!';
		}
	}
?>
```

The array we created will initially be empty. Assigning a value to an array without specifying an index will append a new index to the array with the value assigned to it. To find out if there were any errors, we check for the size of the array:

```
function response
{
	(
		php_assign errors 'array()'
		while IFS=$'\t' read message condition; do
			test "$message" || break
			php_assign 'errors[]' "'$message'" | php_if "$condition"
		done
		(
			# print message in $error
			php_print "$(dquote "$(img $icon_warning WARNING)$(tag br)")"
				(
				php_print "$(php_var error)"
				php_print "'$(tag br)'"
			) | php_foreach "$(php_var errors) as $(php_var error)"
		) | php_if "count($(php_var errors)) > 0"
		(
			# print acknowledgement
			php_print "$(dquote "$(img $icon_ok OK)$(tag br)")"
			while read line; do
				php_print "'$line'"
			done
		) | php_else
	) | php_if "isset($(php_env GET submit))" | php
}
```

The error messages to be printed are ready at hand in the `errors` array, but we haven't got an acknowledgement to be printed yet. That can be pass it in the same way like the error messages, separated by an empty line:

```
response bridgekeeper "$(php_env GET 1 2 3)" << .
Please specify your name!	!isset($(php_env GET 1))
Please specify your quest!	!isset($(php_env GET 2))
Please specify your favourite colour!	!isset($(php_env GET 3))

Thank you for your support.$(tag br)
You answers have been recorded.
.
```

By the way, please note that we checked for unset variables rather then empty strings. Checking for unset variable is a better indicator for an unchecked button. It avoids warnings in the web servers log as described above.

The code also adds an icon to signal error or success. The variables `$icon_ok` and `$icon_warning` specify the URLs of the icon images. They may come from the shell environment or be specified somewhere in the script:

```
php_print "$(dquote "$(img $icon_warning WARNING)$(tag br)")"
php_print "$(dquote "$(img $icon_ok OK)$(tag br)")"
```

There is certainly room for improvement, especially regarding layout and appearance of the feedback. But I think the basic idea came across. Here is all the code to produce the feedback for the user generated so far:

```
<?php
        if ( isset($_GET['submit']) ) {
                $errors = array();
                if ( !isset($_GET['1']) ) {
                        $errors[] = 'Please specify your name!';
                }
                if ( !isset($_GET['2']) ) {
                        $errors[] = 'Please specify your quest!';
                }
                if ( !isset($_GET['3']) ) {
                        $errors[] = 'Please specify your favourite colour!';
                }
                if ( count($errors) > 0 ) {
                        print("<img src='warning.gif' alt='WARNING'><br>");
                        foreach ( $errors as $error ) {
                                print($error);
                                print('<br>');
                        }
                }
                else {
                        print("<img src='ok.gif' alt='OK'><br>");[10]
                        print('Thank you for your support.<br>');
                        print('You answers have been recorded.');
                }
        }
?>
```

The acknowledgement the user receives it not quite true yet: So far, we haven't recorded anything. That has to happen after the consistency check

[10]A word regarding quoting, which is always a sensitive topic in shell scripting. In both, HTML and PHP, single and double quotes are equivalent, as long as they match. Pick the one that's easier to specify. Whenever shell variable expansion is involved, that's likely to be the single quotes, since they can appear as literal in double quoted strings, which are required for shell code for variable expansion. In this line, however, single quotes are part of the string to be printed, so double quoted strings are required in PHP code, which are added by the `dquote` shell function.

but before the acknowledgement, since recording the data still offers plenty of opportunities to go wrong.

Where should the results be recorded? My ISP runs a MySQL server where I'm allowed to create my own databases. That's a perfect option, not least since PHP offers comfortable database support. Recording a user response requires connecting to the database server, selecting the right database and inserting a record of data. All these operations can fail for several reasons and need to be checked for errors. This always follows the same pattern and hence calls for another function:

```
function sql_error {
        php_print "$(dquote "$(img $icon_error ERROR)$(tag br)")"
        php_print "$1"
}

function sql_elseif {
        sql_error "$2" | php_elseif "! $1";
}
```

The first function produces the error message, the second one puts it into an *if*-statement performing the actual error checking, which itself is in the *else*-branch of the previous check.

All this goes, as mentioned above, between the consistency check and the acknowledgement:

```
        ...
        (
                # print message in $error
                ...
        ) | php_if "count($(php_var errors)) > 0"
        sql_elseif "($(php_var dbh) = mysqli_connect(...))" \
                "'Could not connect to database: ' . mysqli_connect_error()"
        sql_elseif "mysqli_query($(php_var dbh),$sql_query)" \
                "'Could not insert values: ' . mysqli_error($(php_var dbh))"
        (
                # print acknowledgement
                ...
        ) | php_else
        ...
```

The omitted parameters for *mysqli_connect* are:

```
        mysqli_connect('$sql_host','$sql_user','$sql_pw','$sql_db')
```

and this is the code snippet being produced:

```
<?php
        if ( isset($_GET['submit']) ) {
                ...
                if ( count($errors) > 0 ) {
                        ...
                }
                else if ( ! ($dbh = mysqli_connect('...')) ) {
                        print("<img src='error.gif' alt='ERROR'><br>");
                        print('Could not ... : ' . mysqli_connect_error());
                }
                else if ( ! mysqli_query($dbh,"...") ) {
                        print("<img src='error.gif' alt='ERROR'><br>");
                        print('Could not ... : ' . mysqli_error($dbh));
                }
                else {
                        ...
                }
        }
?>
```

Most of the shell variables used above should be defined in a place, where they can be changed easily. You might, for example, want to run a test site on your local computer, in which case host name and user credentials are likely to be different. Or you might run a test page parallel to your productive page at your ISPs site, in which case the database name will differ. On top of the main script or in a makefile might be a good place.

Compiling the database query string, which will be some SQL INSERT statement, is a little trickier. That's done on top of the `result` function:

```
function response
{
        table="$1"; shift
        sql_query="$(dquote "INSERT INTO $table VALUES $(sql_tuple "$@")")"
        (
                ...

        ) | php_if "isset($(php_env GET submit))" | php
}
```

The resulting PHP string has to be double quoted, since it will contain values of type string, which require single quotes in SQL. The table name is passed as first argument in this example, equally well it could be defined somewhere else.

The interesting bit here is the creation of the value tuple. What columns needed to be inserted can change whenever something changes in the questionnaire, so that's yet another thing that should be adaptable. Therefore, the information about the values to be inserted is taken from the rest of the `response` function's parameter list.

The functions to form the individual items into a tuple are straight forward:

```
function sql_list {
        echo -n "$1"; shift; for i; do echo ','; echo -n "$i"; done; echo
}

function sql_tuple {
        echo '('; sql_list "$@" | sed 's/^/\t/'; echo ')'
}
```

That's perfectly enough to pass literal values or SQL functions – like `now()` to get the current date and time – to the INSERT statement:[11]

```
$ echo $(dquote "INSERT INTO ... VALUES $(sql_tuple 22 "'a'" 'now()')")
"INSERT INTO ... VALUES ( 22, 'a', now() )"
```

However, the values to be inserted will usually come from some PHP variables when the PHP script is executed, while the query string is compiled when the PHP script is being created. To insert a value at runtime, the query string has to be split and the parts being concatenated with the value you want to insert, meaning, the value item needs to be a piece of PHP code. For string values a suitable function might look like:

```
function sql_str {
        for i; do echo "'$(dquote ".($i).")'"; done
}
```

This will produce:

```
$ sql_str '$a'
'".($a)."'
```

Within the query string this becomes:

```
$ echo $(dquote "INSERT ... VALUES $(sql_tuple 'now()' $(sql_str '$a') 22)")
"INSERT ... VALUES ( now(), '".($a)."', 22 )"
```

Only the first single quote of the item goes as literal part into the query string, starting a new SQL string (within the PHP string). The following

[11]Note that the top level command substitution `$(...)` is not quoted, making the result subject to shell's word splitting, eliminating newlines. That has been done to get the output in one line.

double quote actually terminates the (first part of the) query string. What follows is PHP code: The PHP string concatenate operator `.`, then – in parenthesis, to be on the safe side – the actual PHP expression providing the string value, and then another concatenate operator. The following double quote starts the (second part of the) query string again, then comes the second single quote to terminate the SQL string (within the PHP string). The overall result will be a PHP string expression concatenating all the predefined bits of the query string with the string values provided at runtime.

What's left to be defined is nice way to extract the values we want to insert out of the PHP environment. Basically we need something similar to `php_env`, but with a bit more checking and – for brevity – the ability to accept a sequence of arguments:

```
function sql_env {
        var="$1"; shift
        for i; do
                j="$(php_env "$var" "$i")"
                sql_str "isset($j)?$j:''"
        done
}
```

This allows us to specify something like

```
response survey "$(sql_env GET id)" 'now()' $(sql_env GET 1 2 3) << .[12]
        ...
.
```

to produce

```
"REPLACE INTO survey VALUES (
        '".(isset($_GET['id'])?$_GET['id']:'')."',
        now(),
        '".(isset($_GET['1'])?$_GET['1']:'')."',
        '".(isset($_GET['2'])?$_GET['2']:'')."',
        '".(isset($_GET['3'])?$_GET['3']:'')."'
)"
```

as value of `$sql_query` shell variable.

[12]Note that the expression `$(sql_env GET 1 2 3)` is not quoted, meaning that it's result will be subject to the shell's word splitting and pathname expansion. If any of the values returned contains one or more spaces or wild card characters matching a valid file name, the following steps will fail. Then you'll have to specify each value individually and have them quoted.

The use of REPLACE instead of INSERT is not a mistake, it's intentional and related to one of the two additional values we're going to record in the database. REPLACE is a MySQL extension of SQL and works exactly like INSERT except that a row with the same primary key, that might already exist in the table, is deleted before the new row is inserted. The additional *id* is supposed to be such a primary key. That has been done to avoid a problem: What happens if a user presses the submit button multiple times? Using plain INSERT without *id* we'd end up with multiple entries in the database. That's probably not what we want. PHP has a function to create a unique request id. That can be embedded into the questionnaire as an additional hidden input field. A hidden input field can include data to a form that can't be seen or modified by users when the form is submitted:

```
function questionnaire
{
        (
                tag input type=hidden "name='id'" "value='$(get_id)'"

                ...

        ) | block form method=get "action='$(php '...')'"
}
```

This data can be retrieved like any other input value, as already demonstrated above. Before we set the unique id, we check if there is one already set for that form, if it is, we re-use that, if not, then we're about to send a new, empty form and create a new one:

```
function get_id {
        id="$(php_env GET id)"
        php "$(php_print "isset($id) ? $id : uniqid('', true)")"
}
```

If a user submits the form multiple times now, the use of REPLACE instead of INSERT will make sure, that only the latest version of the answers will be recorded in the database.
The `now()` function, as already mentioned, is an SQL function returning the current date and time, which is recorded as a time stamp in the database entry. You might want to record more additional data, like the users IP address. That might allow you to asses the reliability of your collected data. But please respect your users privacy!

You'll probably find more problems to be addressed, depending on your individual needs, but I expect most of them to be handled quite easily. Here is a summary of the `response` function so far. Again we won't list the output but you are encouraged to try it out:

```
function response
{
      table="$1"; shift
      sql_query="$(dquote "REPLACE INTO $table VALUES $(sql_tuple "$@")")"
      (
            php_assign errors 'array()'
            while IFS=$'\t' read message condition; do
                  test "$message" || break
                  php_assign 'errors[]' "'$message'" | php_if "$condition"
            done
            (
                  # print message in $error
                  php_print "$(dquote "$(img $icon_warning WARNING)$(tag br)")"
                  (
                        php_print "$(php_var error)"
                        php_print "'$(tag br)'"
                  ) | php_foreach "$(php_var errors) as $(php_var error)"
            ) | php_if "count($(php_var errors)) > 0"
            sql_elseif "($(php_var dbh) = mysqli_connect(...))" \
                  "'Could not connect to database: ' . mysqli_connect_error()"
            sql_elseif "mysqli_query($(php_var dbh),$sql_query)" \
                  "'Could not insert values: ' . mysqli_error($(php_var dbh))"
            (
                  # print acknowledgement
                  php_print "$(dquote "$(img $icon_ok OK)$(tag br)")"
                  while read line; do
                        php_print "'$line'"
                  done
            ) | php_else
      ) | php_if "isset($(php_env GET submit))" | php
}
```

The functions we discussed in this and the previous section won't stand alone. They produce HTML text that is likely to be only some part of a document, i.e. that will be completed by some other text. As we moved all the HTML and PHP related code into their dedicated include files, so all the stuff specifically dedicated to creating and evaluating a questionnaire can go into a its own dedicated file, call `questionnaire.include` below.

With all this in place, a script to create a complete questionnaire becomes surprisingly short. In the following example it only specifies the name and location of the icons used, and the credentials to access the database, then it sources the include files and adds some supplementary text. The rest is the questionnaire itself and the error checking as described above:

```
$ cat bridgekeeper
#!/bin/bash

icon_warning='warning.gif'
icon_error='error.gif'
icon_ok='ok.gif'

sql_host='...'
sql_user='...'
sql_pw='...'
sql_db='bridgekeeper'

source html.include
source php.include
source questionnaire.include

bl 'Bridge Keeper' h1

blk p << '.'
Who would cross the Bridge of Death must answer me
these questions three, ere the other side he see.
.

questionnaire << '.'
1. What is your name?
a) Sir Lancelot of Camelot
b) Sir Robin of Camelot
c) Sir Galahad of Camelot
d) It is 'Arthur', King of the Britons

2. What is your quest?
a) To seek the Holy Grail.

3. What is your favourite colour?
a) Blue
b) No, yellow

.

response survey "$(sql_env GET id)" 'now()' $(sql_env GET 1 2 3) << .
Please specify your name!       !isset($(php_env GET 1))
Please specify your quest!      !isset($(php_env GET 2))
Please specify your favourite colour!   !isset($(php_env GET 3))

Thank you for your support.$(tag br)
You answers have been recorded.
.

$
```

Chapter 5

Producing Text

We have created a nice little tool set to format any data into a piece of HTML encoded content, to complete that content with some pre-formatted text and then wrap the result into a standard conforming HTML document. The source of that data might be a plain text file, some command output, a here-document, or whatever. We could even make the content dynamic. Particularly effectively we could described structured content, like lists, layouts, or script code.

What we haven't discussed yet is how to produce larger quantities of textual content, let alone documents containing primarily running text. The problem seems trivial, since all that's required would be a simple text editor. Unfortunately that's not quite as comfortable as we'd wish.

Consider spell checking, for example. I don't want to miss a good spell checker any more, especially when writing documents in a language that's not my mother tongue (like this one). Nowadays it's not uncommon even for simple editors to come with a spell checker, but they are nowhere near the capabilities of a good office product.

Or how to handle things like text highlighting? There are simple shell techniques allowing to have a piece of script code executed within plain text – occasionally they could have been seen in the previous sections. They are perfectly capable of inserting marked text, an image or a hyper link into the text flow, but the very least thing to expect is to have them interfering with the spell checking. A spell checker permanently complaining about script code can be getting quite annoying. Also, text littered with script code is not exactly a pleasure when being proof read.

A third problem is the handling of non-ASCII characters in text content. That is particularly important for languages containing accents or umlauts, but English texts too will every now and then contain things like *en*- or *em*-dashes or non-breakable spaces. Text editors typically handle this using an extended character set. HTML is not limited to ASCII characters any more, but for compatibility reasons it's still best to transform non-ASCII characters into their respective HTML character reference. We will see how to do that in due course. More problematic is an often missing convenient way to enter them, if they are not to be found on the keyboard.

We can summarize that the shell environment has a number of tools available which are perfectly capable of addressing all these problems – as long as the text we're dealing with is sufficiently short. For longer texts things are starting to get unpleasant.

So, back to the office products? They offer the most advanced spell checking capabilities available, easy to use means to highlight text or insert special characters, while the document remains easy to read and to work with. However, we already ruled out the HTML code produced by most office products, for the reasons we discussed in the introduction. That's why we started the whole project in the first place. But isn't there any other possibility to benefit from the convenience office products offer? They are a tempting alternative for writing documents, if only we could find a way to extract the information we need. Since nearly all office products nowadays come with some kind of macro programming, there should be a way.

My favourite office suite is OpenOffice[1]. It is by no means the only one, but it is open source – meaning not only, that it comes free of charge but also that we have access to the source code, which will come in handy later – and has a good programming interface, allowing to access the document's content.

The programming is done in OpenOffice BASIC. Of course, BASIC is usually not exactly my first choice for any serious programming project, and OpenOffice supports other languages as well, Python and JavaScript being the most prominent among them. However, the document API for BASIC is much simpler to deal with and much better documented than it is for other languages. That is – I believe – the most crucial point here. So I decided to stick to BASIC despite of its other deficiencies, and it's good enough for the job anyway.

[1]ApacheOpenOffice, to use the correct term, LibreOffice should work equally well for all examples shown in this document.

5.1 Retrieving the OpenOffice Document Structure

An OpenOffice document is essentially a sequence of paragraphs and tables. Tables are two dimensional arrays of cells, were each cell again contains a sequence of paragraphs and tables.[2] A paragraph in turn is a sequence of text portions. A text portion is a string of characters with no change of text attributes inside. That string can be extracted an printed. Both paragraphs and text portions have numerous attributes describing text properties. These properties can be evaluated.

That doesn't sound bad at all. We have the text and we have the properties, all we need to do is to extract them and write them to a file. Asking the user for a file name and opening it for output is a good point to start coding:

```
Sub Main
    Dim sDocName, sFileName as String
    Dim oDialog as Object
    DialogLibraries.LoadLibrary("DocScript")
    sDocName = convertFromURL(ThisComponent.URL)
    if len(sDocName) > 0 then
        sFileName = fileName(sDocName)
    end if
    oDialog = createUnoDialog(DialogLibraries.DocScript.FileOpen)
    oDialog.getControl("FileName").text = sFileName
    If oDialog.execute() = 1 Then
        exportToFile(oDialog.getControl("FileName").text, ThisComponent)
    End If
End Sub
```

This will be the main function of our script and hence its main entry point. It's convenient to link it to a menu entry or toolbar button into the OpenOffice user interface. Then it can be call by a simple mouse click.[3]

We assume you're familiar with BASIC programming here, we'll only explain what's specific to our purpose: A simple dialogue has been set up to ask the user for a file name.[4] That dialogue resides in a dialogue library called "`DocScript`" and needs to be loaded first. Then we extract the document's name from the document's URL and use it to derive a proposal for the file name. We open the dialogue, set our proposal as default and wait for the user to make a choice. If the user has not aborted the operation, we call a function to export the document to a file.

[2]Actually I'm not sure about tables but it doesn't hurt to assume it's possible.

[3]I use to put it closed the item 'Export as PDF', with AOO4: Tools → Customize...

[4]OpenOffice has a tool for it and to do so is quite straight forward.

Specifically for the use in makefiles, there is batch version as well, expecting the document's file name to be convert as parameter:

```
Sub Batch(sDocName as String)
    Dim sURL as String
    Dim oDoc as Object
    sURL = ConvertToURL(sDocName)
    If CreateUnoService("com.sun.star.ucb.SimpleFileAccess").exists(sURL) Then
        oDoc = StarDesktop.LoadComponentFromURL(sURL, "_blank", 0, Array())
        exportToFile(fileName(sDocName), oDoc)
        oDoc.close(true)
    Else
        msgbox "Document doesn't exist: " & sDocName
    End If
End Sub
```

This can be called like:

```
openoffice4 "macro:///DocScript.export.batch⁵(<full-path-to-document.odt>)"
```

It checks for the existence of the file, then loads it and calls the same export function as the interactive version.

The `fileName` function is used to construct a file name to be proposed as target for the export. That's essentially done by replacing, respectively appending, `.sh` as file extension:

```
Function fileName(sDocName as String) as String
    Dim vPath as Variant
    Dim vName as Variant
    sExtension$ = "sh"
    vPath = split(sDocName, "/")
    vName = split(vPath(UBound(vPath())), ".")
    If LBound(vName) < UBound(vName) Then
        vName(UBound(vName)) = sExtension
    Else
        vName(LBound(vName)) = vName(LBound(vName)) & "." & sExtension
    End If
    vPath(UBound(vPath())) = join(vName, ".")
    fileName = join(vPath, "/")
End Function
```

The function `exportToFile` just opens the output file. That's done by a little helper function, primarily to ensure a proper error handling. The real

[5] The path says, that we call the macro `batch` in the module `Export` of the library `DocScript`

work is done by the function exportDocument, which gets as parameters the file number of open output file and a reference to the current document. After the export function has terminated, the file is closed and we're done:

```
Function openFile(sFileName as String) as Integer
    On Error GoTo OnError
    iFileNumber% = FreeFile
    Open sFileName for Output as #iFileNumber
    openFile = iFileNumber
    Exit Function
OnError:
    MsgBox sFileName & CHR(13) & Error$, 16, "Can't open file"
    openFile = -1
End Function

Sub exportToFile(sFileName as String, oDoc as Object)
    iFileNumber% = openFile(sFileName)
    If iFileNumber >= 0 Then
        oDoc.refresh 'needed to avoid race condition in OO 3.4.1
        exportDocument(iFileNumber, oDoc)
        Close #iFileNumber
    End If
End Sub
```

The files we're creating have the extension .sh. We're going to create shell scripts! We could create HTML code straight away, but scripts retain the whole range of flexibility we've seen in the previous sections and that, as we'll see later, might be more then we have bargained for.

The scripts to be generated need a header. It should specify the correct interpreter in it's first line. We'll need to include some stuff too, in particular everything we'd like to keep adaptable. This goes into the appropriate inlcude file. Writing the header is the chief duty of the exportDocument function:

```
Sub exportDocument(iFile%, oDoc as Object)
    print #iFile "#!/bin/bash"
    print #iFile
    print #iFile "source oo2html.include"
    print #iFile

    REM export document content
    exportContent(iFile, oDoc, "")
    print #iFile
End Sub
```

In case you're wondering why we would bother to create a dedicated function for a trivial task like this, apart from having a nicely structured, textbook-like design :-), please bear in mind that we're also working at a kind of skeleton here. Right now we're only interested in the documents main contents. Later, however, we'll be also looking into more advanced stuff. Document-wide components like endnotes or a bibliography are likely to be handled here.

The next level down the document structure is a sequence of paragraphs and tables. OpenOffice handles such sequences in sequence objects. There is one sequence object per document representing the main text flow.

Iterating through the elements of a sequence object requires an enumeration object. That might look a bit strange, but it's the way it is. To distinguish between paragraphs and tables we can check the so called services, a concept that can be studied in the OpenOffice documentation:

```
Sub exportContent(iFile%, oContent as Object, sShift$)
    print #iFile sShift & "# exportContent"
    Dim oParaEnum, oPara as Object
    oParaEnum = oContent.getText().createEnumeration()
    ' iterate through all paragraphs
    Do While oParaEnum.hasMoreElements()
        oPara = oParaEnum.nextElement()
        If oPara.supportsService("com.sun.star.text.Paragraph") Then
            ' normal paragraphs
            exportParagraph(iFile, oPara, sShift)
        ElseIf oPara.supportsService("com.sun.star.text.TextTable") Then
            ' Tables
            exportTable(iFile, oPara, sShift)
        Else
            ' anything else, should not happen
            MsgBox "Unsupported Text Element"
        End If
    Loop
End Sub
```

Tables are not our main interest right now, but in a first approach they can be dealt with rather easily so we can as well get rid of that issue straight away. That will also show the first piece of shell code produced and illustrates the usage of the shift argument to get nicely nested output. To ease its use we define a little helper function:

```
Function shift(sShift as String) as String
    shift = sShift & CHR(9)
End Function
```

It just appends a tab character to the shift string, which will be prepended to every line we write to the output.

Exporting the table is nothing more then to iterate over rows and columns, and recursively call the `exportContent` function again to deal with the cell content:

```
Sub exportTable(iFile%, oTable as Object, s0Shift$)
    print #iFile s0Shift & "# exportTable"
    s1Shift$ = shift(s0Shift)
    s2Shift$ = shift(s1Shift)
    s3Shift$ = shift(s2Shift)
    iRows% = oTable.getRows().getCount()
    iColumns% = oTable.getColumns().getCount()
    print #iFile s0Shift & "( :"
    For i = 0 to iRows-1
        print #iFile s1Shift & "( :"
        For j = 0 to iColumns-1
            print #iFile s2Shift & "( :"
            exportContent(iFile, oTable.getCellByPosition(j,i), s3Shift)
            print #iFile s2Shift & ") | column"
        Next
        print #iFile s1Shift & ") | row"
    Next
    print #iFile s0Shift & ") | table"
    print #iFile
End Sub
```

We can't really produce any output before the paragraph part is implemented. However, what can be expected so far (from a document containing only a single table with one row and one column) would look like this:

```
#!/bin/bash

source oo2html.include

(
        (
                (
                        ...

                ) | column
        ) | row
) | table
```

As we can see, the shift parameter specifies the nesting level of the document components.

The commands `table`, `row` and `column` are the first three shell commands – well, actually they're going to be shell functions – specifically written for the transformation of OpenOffice documents. There will be more to come. All of theses will go into their own dedicated `oo2html.include` file:

```
function column         { blck td; }
function row            { block tr; }
function table          { block table; echo; }
```

This three commands called from the script above will eventually produce the following HTML fragment:

```
<table>
        <tr>
                <td>...</td>
        </tr>
</table>
```

There will be a lot more to be said about tables, which will be postponed to later. We aren't there yet anyway, we need to take care of paragraphs first. Note that we consider non-empty paragraphs only. That's done since empty paragraphs are sometimes unavoidable, for example, if you want to have a page break straight after a table. You probably don't want to have them in your final document though and getting rid of them here is much easier then doing so later:

```
Sub exportParagraph(iFile%, oPara as Object, sShift$)
    print #iFile sShift & "# exportParagraph"
    If len(trim(oPara.getString())) > 0 Then
        print #iFile sShift & "("
        exportParagraphContent(iFile, oPara, shift(sShift))
        print #iFile sShift & ") | paragraph " & paragraphStyle(oPara)
        print #iFile
    End If
End Sub
```

Paragraphs in OpenOffice represent a more general concept then they do in HTML. In OpenOffice, every element in the main text flow, apart from a table, is a paragraph.[6] This also applies to things like headlines and list items. Consequently, we need to figure out what kind of paragraph we're currently dealing with and pass that information on to the script.

The most important property of a paragraph is its style. Like with style sheets in web pages, nearly any formatting information can be bound to the style. Each style has a name and that name can be retrieved. We don't care

[6]What's not a paragraph of its own is paragraph content, as we'll see in a minute.

about the actual layout details of a specific style – they will be defined later and are likely to differ anyway – we only pass on the name to have something we can bind our own specification to.

However, paragraphs might also carry direct formatting information which might be of interest. Here we're looking for align settings. We only consider values which differ from the current default settings. Using styles is usually the better option compared to direct formatting and usually the later one is considered to be bad style, but occasionally it might be helpful. It's also in here for historical reasons, since I used this code before style sheets became available:

```
Function paragraphStyle(oPara as Object) as String
    Dim oStyles, oStyle as Object
    oStyles = ThisComponent.StyleFamilies.getByName("ParagraphStyles")
    oStyle = oStyles.getByName(oPara.ParaStyleName)
    sStyle$ = "'" & makeName(oPara.ParaStyleName) & "'"
    If oPara.ParaAdjust <> oStyle.ParaAdjust Then
        If oPara.ParaAdjust = com.sun.star.style.ParagraphAdjust.CENTER Then
            sStyle = sStyle + " " + dquote("$(align center)")
        ElseIf oPara.ParaAdjust = com.sun.star.style.ParagraphAdjust.LEFT Then
            sStyle = sStyle + " " + dquote("$(align left)")
        ElseIf oPara.ParaAdjust = com.sun.star.style.ParagraphAdjust.RIGHT Then
            sStyle = sStyle + " " + dquote("$(align right)")
        ElseIf oPara.ParaAdjust = com.sun.star.style.ParagraphAdjust.BLOCK Then
            sStyle = sStyle + " " + dquote("$(align justify)")
        End If
    End If
    paragraphStyle = sStyle
End Function
```

The `makeName` function makes sure, there are no whitespace characters inside a style name. We want to turn the style name into a function name eventually, so the are replaced by underscore characters:

```
Function makeName(str as String) as String
    makeName = join(split(str),"_")
End Function
```

Note the shell command substitution `$(...)` used for the alignment parameter, so `align` is actually a command to be called:

```
function align      { echo "style='text-align:$1;'"; }
```

Having dealt with a paragraph's properties, let's turn to its content. As mentioned previously, a paragraph is a sequence of text portions, where a text portion is a string in such a manner, that all characters in the string share the same attributes. We can iterate through the text portions, looking at each text portion's type and branching to a type specific export function:

```
Sub exportParagraphContent(iFile%, oPara as Object, sShift$)
    Dim oTextEnum
    otextEnum = oPara.createEnumeration()
    ' iterat through all text portions of a paragraph
    Do While otextEnum.hasMoreElements()
        Dim oText as Object
        Dim sType as String
        ' get next portion
        oText = oTextEnum.nextElement()
        ' get portion type
        sType = LCase(oText.TextPortionType)
        If sType = "text" Then
            exportText(iFile, oPara, charStyle(oText, oPara), sShift)
        Else
            unknown(iFile, oText, "text portion: " & sType, sShift)
        End If
    Loop
End Sub
```

A text portion can be much more then a simple string, formatted in a specific way. It could provide dynamic content, like text fields do, reference attached information, like done by footnotes, bookmarks, or annotations, or even represent entirely no-textual content, like images, or formulas.[7]

This again is just a skeleton, only being able to deal with simple text yet. In due course we'll extend it and add step by step more text portion types as we might find them useful for our documents.

Due to the diversity of text portion types we always have to anticipate the possibility to come across a type we haven't considered yet:

```
Sub unknown(iFile%, oObj as Object, sObj$, sShift$)
    print #iFile
    print #iFile "debug 'unknown Object: " & sObj & "' << '.'"
    inspect(iFile, oObj)
    print #iFile "."
    print #iFile
End Sub
```

The `debug` command is one of several commands defined to help us to deal with exceptional situations. Out of the bunch, we'll only need the `debug`

[7]In the last case, it's only the attributes, that counts; the string itself is empty.

command here, but since they all serve a similar purpose, namely to inform the user about some exceptional condition, let's introduce them in one go:

```
function msg            { echo "### $@" 1>&2; }
function warning        { echo "### WARNING: $@" 1>&2; }
function error          { echo "### ERROR: $@" 1>&2; exit 1; }
function debug          { echo "### ERROR: $@" 1>&2; cat - 1>&2; exit 1; }
```

The first two commands merely print a message to the user, flagged with a marker to be easier recognized in the output. That's particularly helpful if you run the script in a more complex build process, which can produce quite an amount of logging output. The third function prints a message and terminates the script. The fourth and last function additionally accepts input from the standard input to be append to the message, before it terminates the script.

The additional debug input expected by the `debug` command is provided by the `inspect` command and attached as here-document. It is information about the object yet unknown to our script. A simple version might look like this:[8]

```
Sub inspect(iFile%, obj as Object)
    print #iFile "# debug: " & obj.ImplementationName
    print #iFile "Properties:"
    dumpStr(iFile, obj.dbg_properties, shift("#"))
    print #iFile "Methods:"
    dumpStr(iFile, obj.dbg_methods, shift("#"))
    print #iFile "supported Interfaces:"
    dumpStr(iFile, obj.dbg_supportedInterfaces, shift("#"))
End Sub
```

That is quite useful, since the strategy behind our scripting framework is to grow along with our documents. Whenever the need arises to use a new type of text element, we'll most likely need to implement a new export function for it. The debug output provides a good starting point.

Note, that neither the command nor the here-document are indented within the script, to make them stand-out against the document's content.

By the way, in case you're wondering why we used a message box in the `exportContent` function but dump debug information into the script here, there is a simple reason: According to the OpenOffice specification the main text flow of a document can not contain anything but paragraphs and tables. Therefore that box should never pop up. If it does it means

[8]A more detailed approach can be found in [1].

we hit a real OpenOffice bug. Interrupting the export seems an appropriate reaction in that case. Running out of options when dealing with text portions only means we've encountered an OpenOffice object we can't handle yet, but there's nothing wrong with the document itself, so we just provide the information helping us to extend our export script in a smooth way.

The three debug properties we're using provide a string, listing all the properties, methods or supported interfaces of the object, respectively. The list starts with a header, separated by a colon which we ignore, followed by the items, separated by either a semicolon or a newline. We normalize that by replacing newlines by semicolons and then print one item per line:

```
Sub dumpStr(iFile%, sDbg$, sShift$)
    Dim oStr as Object
    oStr = split(join(split(Mid(dbgStr,InStr(sDbg,":")+1),CHR(10)),";"),";")
    For i = LBound(oStr) to UBound(oStr)
        If Len(Trim(oStr(i))) > 0 Then
            print #iFile sShift & Trim(oStr(i))
        End If
    Next
End Sub
```

With all this done we can turn to actually exporting text:

```
Sub exportText(iFile%, oText as Object, sStyle$, sShift$)
    sText$ =oText.getString()
    If len(sText) > 0 Then
        print #iFile sShift & "text" & sStyle & " '' " &  quoteStr(sText)
    End If
End Sub
```

Text is ultimately printed using the `text` command. Again, we only consider non-empty text strings. The `text` command gets the text attributes as well as the text string passed as arguments. There's one thing to watch out for, though. A document can contain any printable character including those having a special meaning to the shell. To prevent the shell from interpreting the content of a text string it needs to be quoted properly:

```
Function quoteStr(str as String) as String
    quoteStr = "'" & join(split(str, "'"), "'"&CHR(34)&"'"&CHR(34)&"'") & "'"
End Function
```

Enclosing a string in single quotes will preserve the literal meaning of each character in the string, even that of the backslash, which normally acts as an escape character. Therefore, the only character left to be not allowed in the string is the single quote itself, not even after a backslash, for obvious reasons. To solve this issue we split the string at each eventually occurring single quote into – obviously single quote free – substrings. These are quoted individually and then re-joined by a single quote enclosed double quotes. Within the double quotes the single quote has no special meaning. The shell concatenates these sequence of differently quoted strings (without spaces in between) back into on single string.

Like for paragraphs, the text attributes are extracted by a dedicated function, and not surprisingly, both functions look rather similar:

```
Function charStyle(oText as Objec, oPara as Object) as String
    ' get the character style properties
    If oText.CharStyleName <> "" Then
        ' check for style name first
        charStyle = " '" & makeName(oText.CharStyleName) & "'"
    Else
        iItalic% = com.sun.star.awt.FontSlant.ITALIC
        iBold% =  com.sun.star.awt.FontWeight.BOLD
        iFixed% = com.sun.star.awt.FontPitch.FIXED
        iSmallCaps% = com.sun.star.style.CaseMap.SMALLCAPS
        sStyle$ = ""
        If oText.CharPosture = iItalic And oPara.CharPosture <> iItalic Then
            sStyle = sStyle & " italic"
        End If
        If oText.CharWeight = iBold And oPara.CharWeight <> iBold Then
            sStyle = sStyle & " bold"
        End If
        If  oText.CharFontPitch = iFixed And oPara.CharFontPitch <> iFixed Then
            sStyle = sStyle & " fixedfont"
        End If
        If  oText.CharCaseMap = iSmallCaps And oPara.CharCaseMap <> iSmallCaps Then
            sStyle = sStyle & " smallcaps"
        End If
        If oText.charEscapement < 0 Then
            sStyle = sStyle & " subscript"
        ELseIf oText.charEscapement > 0 Then
            sStyle = sStyle & " superscript"
        End If
        charStyle = sStyle
    End If
End Function
```

Technically, determining the character attributes is part of text exporting function. However, factorizing it out and pass the complete attribute string to the export function turned out to improve re-usability.

All functions defined so far put together are the most basic version of our OpenOffice export script. Assuming you open a new OpenOffice text document now and put the following sentence centred into its only paragraph:

A **bold** `fixed font` in *italic.*

you'd get the following little shell script:

```
#!/bin/bash

source oo2html.include

(
        text '' 'A '
        text bold '' 'bold'
        text '' ' '
        text fixedfont '' 'fixed font'
        text '' ' in '
        text italic '' 'italic'
        text '' '.'

) | paragraph 'Text_body' `align center`
```

That perfectly pictures the structure of our example sentence as we'd like to see it. We have the text itself and all attributes we're interested in. Of course, real text might look rather more complex and not at all as neat as in this little example. A paragraph without any text highlighting for instance might come out as a single text command with the whole text passed as one single argument. That could sooner or later cause trouble; I'm pretty sure there is some upper limit to the length of a shell argument string. In that case we need to think of way to break down the text portions into several pieces. However, I never actually came across that limit and I've already written some texts this way. On the other hand, OpenOffice might every now and then present a piece of apparently equally formatted text as several text portions for no apparent reason. That shouldn't bother us at all, the text will come out all right in the end.

5.2 A Character Filter written in C

We have managed to extract the structure of an OpenOffice text document and encode it into a script. We can use the OpenOffice user interface to assign format information to text portions and we can spell check the document before exporting it. So we have a solution for two of the problems outlined at the beginning of this section. We haven't yet addressed the third one, the problem of of non ASCII characters in the document text. We could do that in the BASIC script as well. The problem, however, is better solved in a little C-program. Not just since I expect the implementation to be easier; a stand alone filter can be used to process text from other sources as well, so it's far more flexible.

The program itself is rather trivial. A single character can represent 256 different values. Up to 255 of them, i.e. excluding the `NULL`-character, might appear in a string. Each of these has either to be passed unchanged or to be mapped to a replacement string. We just take an appropriately initialized, character indexed array of strings pointers. The size of 256 entries is perfectly feasible. Each entry is either a null pointer, in which case the character is left untouched, or points to a replacement string which is simply printed instead of the input. The replacement string might be empty, implying the corresponding character to be ignored. As an option we allow to print a warning message in such cases.

```
void text2codes(char *charcode[], int warn, FILE *in, FILE *out)
{
        int ch;
        while ( (ch = getc(in)) != EOF ) {
                char *code = charcode[ch&0xff];
                if ( ! code )
                        putc(ch, out);
                else if ( *code )
                        fputs(code, out);
                else if ( warn )
                        fprintf(stderr, "ignored charcode %02x\n", ch);
        }
}
```

The translation table is a simple C array hard coded into the source files. We could fill in the 256 entries manually, but for script gurus like us that would be a shame. So it's time to have a closer look at a tool we have already met briefly, the `awk`.

`awk` is a C-like scripting language specifically designed to process line oriented data in text files. An `awk`-script typically consist of a sequence of pattern/action pairs. It scans the input and whenever a pattern matches a line, the corresponding action is performed. Special patterns allow to specify actions before and after the input is processed.

Let's assume we have a text file `wwwcodes` containing the HTML character references with one character number and its corresponding reference per line. This mapping can be found in any HTML documentation. We might even find one in the Web that only needs a bit of tweaking to get it in the correct format. This file needs to be transformed into a C-array of string literals. Here's a proposal (see file `charcodes.sh`):

```
#!/bin/bash

function c_block        { echo "$@" '{'; sed 's/^/\t/'; echo '}'; }
function c_list         { sed '$!s/$/,/'; }
function c_aggregate    { sed '$s/$/;/'; }

cat "$@" | awk '

        BEGIN   {
                        for ( i=0; i<256; i++ )
                                charcode[i] = "\"\"";
                        for ( i=0x20; i<0x7f; i++ )
                                charcode[i] = 0;
                }

                {       charcode[$1] = "\""$2"\""; }

        END     {
                        for ( i=0; i<256; i++ )
                                printf("/* 0x%02x */\t%s\n", i, charcode[i]);
                }

' | c_list | c_block 'char *charcode[256] =' | c_aggregate
```

We start by initializing an associative array with exactly 256 elements, each of them being a string containing two double quotes, which denotes in C an empty string. Hence, by default all characters are going to be ignored. Then we replace the range of known printable characters by zero, meaning they're going to be passed unchanged. When that's done the input file(s) are being processes, which simply sets the reference string for each explicitly specified character. Finally the array is printed, together with some C-comments. Encapsulating that list into the right C-syntax is only a cinch by now.

If you want, put the three function dedicated to generating C-code into a separate include file and have it ready to be re-used. They make another nice example of how simple and effective meta programming can be.

The main program only checks a few options and arranges for the right file descriptors to be open. It follows the UNIX convention, that any non-option argument is an input file name. We have added an additional option that allows us to let a line end character pass unchanged. We'll need it soon (file `text2codes.c`):

```
#include <stdio.h>

extern char *charcode[];

void text2codes(char *charcode[], int warn, FILE *in, FILE *out)
{
   ... /* see above */
}

int main(int argc, char *argv[])
{
        int warn = 0;
        char *progname;
        for(progname = *(argv++); *argv!=NULL && **argv=='-'; argv++) {
                if((*((*argv)+1))=='\0')
                        break; // stdin
                else if(strcmp(*argv, "-i") == 0)
                        warn = 1;
                else if(strcmp(*argv, "-br") == 0)
                        charcode['\n'] = 0;
        }
        if( *argv == NULL )
                text2codes(charcode, warn, stdin, stdout);
        else for(; *argv != NULL; argv++) {
                FILE *fp;
                if ( strcmp(*argv, "-") == 0 )
                        text2codes(charcode, warn, stdin, stdout);
                else if( (fp=fopen(*argv, "r")) != NULL ) {
                        text2codes(charcode, warn, fp, stdout);
                        fclose(fp);
                }
                else {
                        fprintf(stderr, "%s: can't open %s\n", prog, *argv);
                        return 2;
                }
        }
        return 0;
}
```

Generate the character reference table, compile and bind it together with the main program, and our little tool is ready to use:

```
$ cat wwwcodes | bash charcodes.sh >charcodes.c
$ cc -o text2html text2codes.c charcodes.c
```

You might want to add some more stuff, like a better error reporting or some more options, but for the time being this little program does everything we need.

5.3 A Word Wrapper in C

An OpenOffice paragraph typically does not contain any line breaks. Even if it does, they are likely to be replaced by the line break tag or some other character reference. That's what the character filter above was meant for. Consequently a paragraph in the resulting HTML document is likely to be a single, rather long line. As far as I'm aware of, a browser wouldn't mind, but it's rather ugly and uncomfortable if you are forced, for whatever reason, to look at the HTML code. It'd be much nicer to have the text broken into lines. That's what the following filter is there for.

The filter is implemented as a finite state machine. A finite state machine is a mathematical model defined by an input alphabet I, a set of states S, a state transition function $t\colon S \times I \to S$ and an initial state s_0. A finite state machine works cyclic: In each cycle it reads an input character and goes into the next state, according to the transition function. Eventually it reaches a terminal state, which ceases the process. In our case that will be equivalent to executing the `return` statement. Each transition can be associated with some actions to be performed. Sometimes the model is extended by an output alphabet O and a result function $r\colon S \times I \to O$. In that case the associated action of each state transition will be to output an output character. This model, however, doesn't really fit here, since a state transition might output none, one, or more characters.

Our word wrapper state machine has 4 states, closely associated to the current position in the output line:

- State 0 (initial state): We are at the beginning of an output line, potentially reading whitespace characters.
- State 1: We are reading the first token on the output line.
- State 2: We are reading whitespace characters between two token on the output line.
- State 3: We are reading a subsequent token on the output line.

The term *token* is a synonym to *word* in this context.

In each state we have to distinguish between three possible kinds of input characters:

- printable characters,
- whitespace characters, and
- end of file.

In state 0 we are just before the beginning of a new output line. Each new line starts with a printable character, so we stay in this state until we find one. Withespaces are ignored. In case of an end of line we're done, the new line will never be started. If we find a printable character, we print it as the first character on the new line, initialize the character counter and go to state 1:

```
state 0:
    ch:=input();
    if eof(ch) then
        return;
    elseif whitespace(ch) then
        goto state 0;
    else
        char count:=1; output(ch);
        goto state 1;
    end;
```

In state 1 we are reading the first token of the current output line. We stay in this state as long as we read printable characters:

```
state 1:
    ch:=input();
    if eof(ch) then
        output(line end);
        return;
    elseif whitespace(ch) then
        char count:=char count + 1;
        if char count < line length then
            goto state 2;
        else
            goto state 0;
        end;
    else
        char count:=char count + 1; output(ch);
        goto state 1;
    end;
```

Characters are just counted and passed on to the output. We are not bothered about exceeding the line length while remaining in this state. We can't make any assumptions about the structure of a token, hence we have no chance for hyphenation. If a token should be longer then a line, tough luck; the best we can do is to put it on a line on its own.

A whitespace terminates state 1. We add one more the character counter to take into account the space character that might follow the current token and check the line length now. If we've already reached or exceeded the line length limit, we terminate the output line and return to state 0 to start a new line, otherwise we proceed into state 2. The transition to state 2 implies that there's enough space left on the current line for at least one more character. In case of an end of file, we terminate the output line and return.

In state 2 we are reading whitespaces between two tokens on the output line. Similar to state 0, they are ignored. In case of an end of file, we return, terminating the output line first. If we find a printable character, we can't print it straight away. We don't know yet if the new token is going to fit on the current line. Instead, we initialize a buffer, push the character to it, initialize the buffer count and go to state 3:

```
state 2:
    ch:=input();
    if eof(ch) then
        output(line end);
        return;
    elseif whitespace(ch) then
        goto state 2;
    else
        buffer[0]:=ch; buff count:=1;
        goto state 3;
    end;
```

The bulk of the work is done in state 3. While we are in it, we know we are reading a token that so far still fits on the current output line. If we see the end of file, we know we can safely print it (including the separating space), terminate the output line, and return. Similar for a whitespace, in that case we add the buffer size to the character count (including an extra count for the next potentially following space) and check the line length. If there is still space left for at least one more character, then we continue with state 2, otherwise we terminate the current line and start all over again in state 0.

If we see a printable character, we check for the line length to see if the current token would still fit. If it does, we add the character to the buffer and remain in state 3. If it does not, we need to start a new line. On the new line there is no point for buffering any more. The current token will go onto that line, no matter how long it's going to be. So, we print all we've got so far (i.e. the content of the buffer and the character we've just seen), set the character counter accordingly, and continue in state 1:

```
state 3:
    ch:=input();
    if eof(ch) then
        output(space); print(buffer); output(line end);
        return;
    elseif whitespace(ch) then
        output(space); print(buffer);
        char count:=char count + buff count + 1;
        if char count < line length then
            goto state 2;
        else
            output(line end);
            goto state 0;
        end;
    else
        if char count + buff count < line length then
            buffer[buff count]:=ch; buff count:=buff count + 1;
            goto state 3;
        else
            output(line end); print(buffer); output(ch)
            char count:=buff count + 1;
            goto state 1;
        end;
    end;
```

The number of characters we store will never exceed the line length, even if individual tokens can be longer. That's good news, since it spares us the trouble to dynamically adjust the buffer length.

You might be wondering about the the large number of `goto`s in the pseudo code. Aren't they supposed to be bad? No, not really. The `goto` statement is just a tool, and like any other tool it can't be bad *per se.* It can only be used or abused. `goto`, unfortunately, is mostly abused to violate the structured programming paradigm. That's bad indeed, so you are always well advised to consider its usage carefully. In most cases it can – and really should be – avoided. However, there is no rule without an exception and structured programming is not always an appropriate choice. Structured programming tries to handle complexity be dividing a problem into smaller sub-problems. A finite state machine, however, already is a sound mathematical model. There is little point in ripping it apart.[9] On the other hand, `goto` provides exactly the pattern a finite state machine requires: read some input, perform some action, and the *go to* the next state. So `goto` is exactly the right tool for the job here. There are alternatives, but none of them I'm aware of offers a simpler and more effective way.

We won't list the C source code here, it's getting rather long and it's not really difficult. The finite state machine is a function implementing the four states above, while `main` allocates the line buffer and takes care of the file handling stuff. It might evaluate some options too. That's it.

There's on issue, though, that's worth to watch out for: As mentioned above, the word wrapper doesn't know anything about the structure of a token. This applies also to quoted strings; the wrapper will not recognize a quoted string and might recklessly wrap at any whitespace that might be in there. That's hardly a problem in ordinary text, but might cause trouble if the text contains HTML tags with quoted attribute values. We haven't seen them yet, and even with all what's still coming up that's quite unlikely, but it can't hurt to bear that fact in mind. The easiest workaround is to replace vulnerable whitespaces by their corresponding character reference before piping them trough the word wrapper.

5.4 Running a Document Script

We managed to retrieve the text of an OpenOffice document, including the interesting aspects of its structure, and turn it into a shell script. We also have two text filters now, that will assist us in processing the text. What's left to do is to put this things together and evaluate what we've got. Let's recall that little script fragment from our example sentence:

```
(
        text '' 'A '
        text bold '' 'bold'
        ...

) | paragraph 'Text_body' `align center`
```

We still need to provide a `text` and a `paragraph` command. We start with the former, it'll be the more demanding one. The challenge here is that it can contain any number of format specifiers, including none at all. Each

[9]In most cases that would not even be possible, unless the state transition graph can be structured into a block diagram. That, however, is likely to be a rare exception.

of them will require it's own filter, all concatenated in a pipeline. That calls for a recursive function definition:

```
function text
{
	TAG="$1"; shift
	if test "$TAG" != ''; then
		text "$@" | char_$TAG
	elif test $# -gt 0; then
		echo -n "$@" | text2html
	else
		text2html
	fi
}
```

Now it becomes clear why we arranged the arguments in such a strange way, in particular, what the empty argument is used for. It makes evaluation easier by allowing us to process arguments from left to right. We extract a parameter from the parameter list and look what we've got. If it is not empty, we interpret it as a format specifier and use it to create a filter name. The prefix creates a dedicated name space for character styles, to distinguish them from other types of style functions, like such for paragraph styles. With the remaining arguments we call the text command again, piping whatever that will produce into our previously specified filter. The recursion stops at the first empty argument, or if there are no arguments left. All remaining arguments, if any, are considered to be text and echoed into our character filter to get non-ASCII characters replaced by there HTML references. If there are no arguments left, we read from standard input.

The format specifying functions are also filters, wrapping the text into the appropriate tags. If we don't have an HTML equivalent, we just pass the text on unchanged:

```
function char_bold            { blck b; }
function char_italic          { blck i; }
function char_smallcaps       { cat; }        # no smallcaps in HTML
```

Styles are handled precisely the same way, only that they adhere more to the idea of using logical mark-up, rather than direct formatting:

```
function char_Strong_Emphasis   { blck strong; }
function char_Emphasis          { blck em; }
function char_Source_Text       { blck code; }
```

By the way, both can contain much more complex code than shown here in the examples, as we might see later.

Paragraphs are easier to deal with. They don't require the recursive structure and always read from `sdtin`. What remains is a simple wrapper:

```
function paragraph      { style="$1"; shift; para_$style "$@"; }
```

As for character styles, an own namespace is defined for paragraph styles. The real work is done in the style functions passed as argument. Many of them follow the same pattern, so it is quite handy to have some helper functions:

```
function para           { blck "$@" | wordwrap; echo; }
function hl             { blck "$@" | wordwrap; }
```

The `echo` in the first line is just there to improve the script layout. This two little functions make it fairly simple for most paragraph styles to assign a layout:

```
function Text_body      { para p "$@"; }
function Standard       { para p "$@"; }
function Quotations     { para blockquote "$@"; }

function Heading_1      { hl h1 "$@"; }
function Heading_2      { hl h2 "$@"; }
function Heading_3      { hl h3 "$@"; }
```

All this put together and applied to our example sentence produces the following HTML code:

```
<p style='text-align:center;'>A <b>bold</b> <tt>fixed font</tt> in
<i>italic</i>.</p>
```

I use to put the function definitions into separate include files, one for the general stuff, one for OpenOffice related functions and one for style declarations. Style names can vary from document to document, or perhaps you want to change the appearance of certain styles for a particular document or project. All you need to do then is to make sure that the project specific specification are found in your search path before the more general once:

```
html.include           # basic HTML function
oo2html.include        # genaral OpenOffice to HTML functions
oo2html.styles         # template specific functions
```

A word regarding efficiency: You might have noticed that what we are doing here is not exactly a text book example of economical resource utilisation. To get even a simple piece of text requires an `echo`, a pipe, and a C-program. That means at least one fully fledged operating system process,[10] and creating a process is really expensive in term of system resources. Any additional tag adds to that costs. Don't worry too much, nowadays any PC has enough power to handle this. You shouldn't use this approach to generate dynamic content, though. A web site having a few thousands hits a minute will certainly be knocked out by it, no matter how much CPU power or memory you might have.[11]

5.5 Unicode Support

The two C-filters had been working stable for quite a while when one day the time had come again to update my operating system. Suddenly strange characters appeared in places where previously umlauts and accents had been coded correctly. What had happened? The new Linux distribution turned out to be the first one I came across that uses *Unicode* as the default encoding scheme.

Before the invention of Unicode the most important computer code was ASCII, the *American Standard Code for Information Interchange*. It encodes the ten digits, the letters of the Latin alphabet in both, upper in lower cases, the punctuation marks typically found on a typewriter, and a set of control characters. That's perfectly OK for languages using an alphabet based exclusively on Latin characters, like English. Most programming languages are based on ASCII, too. But it's just not good enough when a language uses characters beyond that, and that are most of the other human languages spoken and written in the world.

ASCII is a seven bit code while computers typically work with bytes of eight bit, so only the lower 128 of the 256 characters encodable in a byte are actually being used. The first attempt to offer language specific characters was to encode them into the free upper positions. This worked fairly well for most European languages, in fact, that's the approach my Linux used before Unicode.

But this way the encoding is language dependent. Different languages use different character sets, so reading a German text on a French computer is bound to cause trouble. Combining different character sets in one document is entirely impossible. And what about languages, that use more than 256 characters, like Asian languages?

[10] Assuming the shell uses an internal `echo`, otherwise it would be two.

[11] And even if you don't reach that hit rate, your service provider will probably not be happy about the load you are creating, even with moderate traffic :-)

The Unicode project is the modern approach to address this problems. It strives to provide a single encoding scheme for all languages worldwide.[12] That, of course, implies a number of representable characters our simple filters are not capable to deal with yet.

For a fast solution, you could simply set the character encoding back to a scheme without Unicode, all Unicode supporting software should gracefully acknowledge this.[13] Character encoding is typically specified in one of the environment variables `LANG`, `LC_TYPE` or `LC_ALL`. Unicode support can be recognized by a suffix like `UTF-8`. Just setting the appropriate variable to the same value with that ending removed should work in most cases:

```
$ echo $LANG
de_DE.UTF-8
$ LANG=de_DE
```

It would of course be much better if our filter would be aware of Unicode. Apart from being more portable it would offer all the benefits of an enlarged character set.

Supporting more characters requires more bits to encode them. For some time 16 bits have been considered sufficient for the foreseeable future, but it seems, that future has already passed. By the end of 2021 Unicode defined some 144.000 characters and arguments were that up to 32 bit will be required soon. However, just replacing each byte by a 32 bit word would quadruple the size of existing ASCII files without adding any information. Most texts in a computer are – and are likely to remain – in plain ASCII, so that's not such a great idea.

Unicode is strictly speaking not a code, it only defines numerical values for characters, so called code points, but no bit patterns or bit sequences. That's left to an encoding scheme, of which several are possible. The currently most popular one is UTF-8, which stands for *Unicode Transformation Format* based on 8-bit words. It is a variable-length code, encoding code points into sequences of 1 to 4 bytes. All ASCII characters reappear unchanged in UTF-8, consequently each text file in plain ASCII is a valid UTF-8 file too. That's a tremendous advantage.

[12]Frankly, I have certain doubts about this ultimate goal: Finding an encoding for all languages expressly includes things like runes and hieroglyphs, even fictional languages. (Klingon has been proposed twice, Tengwar – known from Tolkien's *Lord of the Rings* – even made it to the Unicode road map, as far as I know.) Providing universal solution covering all writing systems in all the world sounds to me like the search for the philosophers stone. That has, as far as I'm aware of, never yet succeeded. However, Unicode supports many more languages than any other coding system yet and that's something.

[13]The Unicode folks are likely to kill me for suggesting it :-)

We don't want to dig into the encoding details here, we can rely on libraries to do the job for us. They'll take care of all the encoding and decoding tasks. All we have to do is to include the appropriate header files and initialize the library:

```
...
#include <locale.h>
#include <wchar.h>
...

int main(int argc, char *argv[])
{
        ...
        if ( ! setlocale(LC_CTYPE, "") )
                fprintf(stderr, "locale not specified");
        ...
}
```

That takes care of evaluating the environment variables and switching to the correct encoding schemes. Even turning of Unicode support completely, like indicated above, should work smoothly this way.

The fundamentals of the filter routine doesn't change that much. It mainly uses a different type for variables, functions and constants concerned with input characters. They have to be capable of dealing with the larger range of values. The precise type is implementation dependent and shouldn't concern us:[14]

```
int text2codes(char *charcode[], int warn, FILE *in, FILE *out)
{
        wint_t ch;
        while ( (ch = getwc(in)) != WEOF ) {
                char *code = (ch&~0xff) ? lookup(ch) : charcode[ch&0xff];
                ...
        }
        return ferror(in);
}
```

The only really new bit is the character lookup: There are characters now with a code point beyond the 8-bit range. Extending the lookup table has its limits, Unicode characters can have 16 or even 32 bits. A table of such size is clearly infeasible and, since only a few entries would be needed for our purpose, a huge waste of space. We use a dictionary lookup instead

[14]It's probably an 32-bit unsigned integer.

for all characters represented by more than 8-bit. The dictionary is an array where each entry is a key-value pair:

```
struct lookup_entry { wint_t key; char *code; };
```

If that array is sorted, we can use an efficient binary search. There is an appropriate function available in the C standard library:

```
char *lookup(wint_t ch) {
        void *result = bsearch(
                &ch, lookup_dictionary,
                sizeof(lookup_dictionary)/sizeof(struct lookup_entry),
                sizeof(struct lookup_entry),
                lookup_cmp
        );
        return result ? ((struct lookup_entry*) result)->code : "";
}
```

The details of the `bsearch` function can be found in the man page. Apart from some size information, the only things we need to provide is the dictionary itself, a reference to the key we're looking for, and a function comparing the key with an entry, returning a value that is positive, negative or zero if the key we're looking for is larger, smaller or equal respectively to the key of the entry currently under consideration:

```
int lookup_cmp(const void *key, const void *elem) {
        return *(wint_t*)key - ((struct lookup_entry*) elem)->key;
}
```

The dictionary is created using a list of character references in the same format as we already used for the lookup table. We don't need to worry about default entries, so the process is going to be slightly simpler. We only have to make sure the entries in the dictionary are sorted properly, otherwise the search function will fail. With standard tools, that's next to trivial:

```
function c_block        { echo "$@" '{'; sed 's/^/\t/'; echo '}'; }
function c_list         { sed '$!s/$/,/'; }
function c_aggregate    { sed '$s/$/;/'; }

cat "$@" | sort -n | awk '{ printf("{   0x%04x,\t%-24s }\n", $1, $2);}' \
| c_list | c_block 'struct lookup_entry lookup_table[] =' | c_aggregate
```

The size of the directory is not known in advance, it depends on the number of entries in the input file. It's possible to retrieve it from the array size, but only within the same translation unit. Hence, all the lookup bits and pieces above have to go into the same file:

```
#!/bin/bash

function c_block        { echo "$@" '{'; sed 's/^/\t/'; echo '}'; }
function c_list         { sed '$!s/$/,/'; }
function c_aggregate    { sed '$s/$/;/'; }

cat << _END_OF_CODE_

#include <stdlib.h>     /* for bsearch */
#include <wchar.h>      /* for wint_t */

struct lookup_entry { wint_t key; char *code; };

_END_OF_CODE_

cat "$@" | sort -n | awk '{ printf("{   0x%04x,\t%-24s }\n", $1, $2);}' \
| c_list | c_block 'static struct lookup_entry lookup_table[] =' | c_aggregate

cat << _END_OF_CODE_

static int lookup_cmp(const void *key, const void *elem) {
        ...
}

char *lookup(wint_t ch) {
        ...
}

_END_OF_CODE_
```

The C parts are just copied to the target file while the missing bits are generated. All objects are declared static, i.e. local to the file; only the directory lookup function itself is being exported. Don't forget to declare it in the main file, like we declared the lookup table:

```
extern char *charcode[256], *lookup(wint_t ch);
```

To create an executable we only need to build all C files and compile:

```
$ cat wwwcodes | bash charcodes.sh >charcodes.c
$ cat wwwlookup | bash lookup.sh >lookup.c
$ cc -o text2html text2codes.c charcodes.c lookup.c
```

That should do it. Right now there is no need to adopt the word wrapper as well. It will only see input that has passed through the character filter first and that is is guaranteed to be plain ASCII. It can't hurt, of course, to make the word wrapper fit for Unicode as well. If you plan to use it outside our tool set you should seriously consider it. The way is essentially the same.

If you're used to work on different computers and move your documents around a new issue arises: Different systems might use different character encoding schemes. That's not a problem for the OpenOffice documents itself, since OpenOffice uses it's own internal encoding, but for the generated scripts. They will be encoded in whatever scheme was active when OpenOffice had been started. The target system can use a different set-up, so we have to tell it, how the script once has been encoded. The easiest way to do so is to set the `LANG` variable within any document script. For good measure we add some further settings as well. Some day they might be of interest:

```
Sub exportDocument(iFile%, oDoc as Object)
    print #iFile "#!/bin/bash"
    print #iFile
    print #iFile "source oo2html.include"
    print #iFile

    print #iFile "export OPENOFFICE_SOLAR_VERSION=" + GetSolarVersion()
    print #iFile "export OPENOFFICE_VERSION=" + ooVersion()
    print #iFile "export OPENOFFICE_GUI=" + GetGUIType()
    print #iFile "export LANG=" + Environ("LANG")

    REM export document content
    exportContent(iFile, oDoc, "")
    print #iFile
End Sub
```

The value is taken from the current environment. On Unix-like systems that should be set correctly. On Windows you'll have to set it explicitly, set it to something like `en_US.cp1252`.[15]

Your target system needs to know the encoding scheme. Use `locale -a` to check what's already available. If a particular one is missing you can create it. In case of the Window encoding above that's done with:[16]

```
# localedef -f CP1252 -i en_US en_US.cp1252
```

For details please check the `locale` man page.

[15] On XP: My Computer → Properties → Advanced → Environment Variables; sorry, didn't use Windows after XP

[16] Run this as root.

Chapter 6

Advanced Text Features

We have demonstrated how shell scripting can be used to create HTML documents and how data, that's already available in one form or another, can be processed to be part of these documents. Now we've shown, how a standard office application like OpenOffice can be used to generate running text to be included in our scripting framework.

However, office applications are not only good for running text. The data for lists and tables will not always be available ready to use; often it might just be convenient to enter it directly into the document text. There are other document features as well, like hypertext links or images, that we might want to put into the text flow. We are about to show, how this could be achieved.

Till now, our path was rather straight forward. So far, there has been a not necessarily unique, but more or less obvious choice for each of our problems. The more advanced the features we're dealing with are going to be, the more likely that is going to change. Modern office applications offer a wide range of features, but only a few of them can be mapped directly to HTML. For some, that mapping is not unambiguous. That means, we usually have to make a choice. The tricky bit is to find the OpenOffice feature, that is most suitable to represent what we want to achieve and is still easy enough to be analysed. The choice can be different from one project to another, or even from one document to another.

We're going to present a selection of possible choices. All of them have been used in previous projects; some of them turned out to be quite stable, yet some of them have undergone considerable changes over time or are likely to do so in the future.

6.1 Tables

We've already met tables. OpenOffice treats tables like paragraphs, so we were forced to deal with them when we were extracting the main text flow. We've already demonstrated how to transform a table into a piece of script code. There was still a part missing by the time we looked at it, namely the one to fill in the table cell contents, but since table cells contain a sequence of paragraphs, which we are perfectly capable of processing now, there's nothing more left to do, at least for the most simple of tables. Just one thing to bear in mind: If a table cell contains a sequence of ordinary paragraphs, then each non-empty table item would be enclosed in paragraph tags. That's, in most cases, not what we want. Luckily, there is a simple solution. By default, OpenOffice assigns a special paragraph style to table cell contents, named something like `Table_Contents`. That can easily be used to alter the standard paragraph behaviour.

```
function para_Table_Contents        { wordwrap; }
```

This approach will ignore all paragraph-related formatting information. It has to, since in this standard situation, there is no tag left to bind them to. Alignment information for a cell typically goes into the `tr`-tag, and that's one level up.

We've hit two tricky problems here. The first is a technical one: How to pass information from within a nested block back to its enclosing block. Within shell scripts, that's next to impossible unless you're prepared to deal with advanced techniques of process communication, which clearly overstresses the abilities of the shell. Hence, such problems need to be dealt with in the OpenOffice macro script, which unfortunately might add a considerable amount of complexity.

The second problem is a structural one: The paragraphs in the cells can carry individual alignment information; even multiple paragraphs with the same cell can be aligned differently. If that's really what you want, then sticking to paragraph tags within the table cells is probably the best choice (and the simplest and most straight-forward solution). Otherwise, you'll need to decide which alignment should go into the table tags. That might be based on all cells of a row, of a column, or of the whole table. Of course, all these can be decided in the macro script prior to generating any code, but you see what I mean by increased complexity?

6.1.1 Simple and Complex Tables

Tables – or, more precisely, text tables – can be simple or complex. A table is simple if each cell spawns exactly only one row and one column, i.e. without any merged or split cells. A table that is not simple is complex. Unfortunately, not all methods of a table object work as expected for complex tables. This includes the method to obtain a cell by its position, as we did in our introductory example. Some indices might not exist; others might return empty cells. Cells in a complex table can be reliably accessed only by their names. Since we never know in advance if a table is complex, we'll have to use names, which complicates things a bit.

Each column is labelled alphabetically, starting with the letter A, and each row is labelled numerically, starting with the number 1; the concatenation of these two strings is a cell's name. There is the function `getCellNames` to retrieve an array of all cell names of a table. The first task will be to sort this array into rows and to determine the column spawn, since the cell attributes include only the row spawn.

I've never actually gone that far. Up to now, I've been quite happy with simple tables, which are perfectly easy to create with the techniques we have. If you want to have a paragraph sequence in a table cell, you can achieve this simply by changing the paragraph styles within the cells. If you just want a line break within a table cell, then a line break character should be the preferred choice.

6.2 Lists

While the rows and columns of a table are easy to extract from a document, lists and list entries are trickier to deal with. OpenOffice maps list items to certain paragraph attributes, but these items are never bundled into a single list. Hence, we have to do the job ourselves. The idea is to have all consecutive paragraphs with similar list attributes identified as items of the same list and bundle them together before they find their way into the script.

6.2.1 Numbering Level and Numbering Type

The first step, as always, is to find a paragraph attribute suitable to distinguish list items from normal paragraphs or items of another list. If a paragraph is a list item, then it is numbered – even if it is a list with bullets – and hence has some *numbering rules* attached to it. With numbering rules that do not refer to outline numbering, it also has a valid *numbering level.*

The numbering level seems to be a suitable candidate. Here is a function to retrieve it:

```
Function paraNumberingLevel(oPara as Object) as Integer
    paraNumberingLevel = 0
    If oPara.supportsService("com.sun.star.text.Paragraph") Then
        If Not isEmpty(oPara.NumberingRules) Then
            If Not oPara.NumberingRules.NumberingIsOutline Then
                paraNumberingLevel = abs(oPara.NumberingLevel) + 1
            End If
        End If
    End If
End Function
```

We check if the current object is a paragraph (it might be a table), if it has some numbering rules attached to it, and if these numbering rules are line numbering rules as opposed to *outline numbering*. Outline numbering is used to structure a document and is of no relevance here. Per default, each headline, for example, has an outline numbering rule set attached to it. Because we don't want to see them as list items, we need to exclude outline numbering.

For tables, like for paragraphs without line numbering, the functions defaults to zero. Hence, they can never be part of a list. All other values are incremented by one to move them to a range larger than zero. The `abs` function is a safety precaution. It makes sure the numbering level returned is non-negative, even if, for some obscure reason, OpenOffice should present a negative numbering level one day. In that case, our document layout might be screwed up, but our script will not run wild.

If a paragraph has a numbering level, it also has a *numbering type*. The numbering type determines the kind of numbering used for the list, or zero if it is a bullet list.

```
Function paraNumberingType(oPara as Object, iParaNumberingLevel%) as Integer
    paraNumberingType = 0
    If ( iParaNumberingLevel > 0 ) Then
        Dim oRule()
        Dim i As Integer
        oRule()= oPara.NumberingRules.getByIndex(oPara.NumberingLevel)
        For i = LBound(oRule()) To Ubound(oRule())
            If oRule(i).Name = "NumberingType" Then
                paraNumberingType = abs(oRule(i).Value) + 1
            End If
        Next
    End If
End Function
```

Again, we make sure we get a value larger than zero for a numbering type or zero if we don't have one.

The paragraph numbering type is an enumeration type and can easily be translated into a more readable representation:

```
Function listType(iType%) as String
    If iType <> 0 Then
        Select Case iType - 1
            Case com.sun.star.style.NumberingType.ARABIC:
                listType = "arabic"
            Case com.sun.star.style.NumberingType.ROMAN_UPPER:
                listType = "roman_upper"
            Case com.sun.star.style.NumberingType.ROMAN_LOWER:
                listType = "roman_lower"
            Case com.sun.star.style.NumberingType.CHARS_UPPER_LETTER:
                listType = "letter_upper"
            Case com.sun.star.style.NumberingType.CHARS_LOWER_LETTER:
                listType = "letter_lower"
            Case com.sun.star.style.NumberingType.CHAR_SPECIAL:
                listType = "char_special"
            Case com.sun.star.style.NumberingType.BITMAP:
                listType = "bitmap"
            Case Else
                listType = trim(str(iType-1))
        End Select
    Else
         listType = ""
    End If
End Function
```

6.2.2 Simple Lists

To assemble paragraphs into lists, we keep track of the numbering level of the list we are currently processing. To allow for nested lists, that's done on a stack. We need a stack here since in OpenOffice the numbering levels of nested lists do not need to follow a strict numerical order, i.e. list levels can be skipped when going down to the next nested list. We don't want such gaps in our target document, so we maintain our own list level counter, which serves as an index into the stack, keeping the OpenOffice numbering levels there. We will also need the numbering type, so we keep that on a stack too, and – for the sake of simplicity – also the shift string. OpenOffice currently supports up to ten list levels, so with a stack size of 32, we should be on the safe side.

Before processing a new paragraph, we adjust the list stack as necessary: While the numbering level of the current paragraph is lower than that of the current list, we close any previous list; if it is higher, we open a new one. If the numbering level remains the same, but the numbering type changes it also marks the beginning of a new list, so we have to close the current and open a new one. Last but not least, we also have to make sure all open lists are closed at the end of the document.

Note the stoppers we put on the stacks. These stoppers make sure the stack never gets empty, so we don't have to check for that explicitly. On the one hand, a paragraph's numbering level can't be smaller than zero, so stack unwinding in the loop stops here for sure. On the other hand, a numbering level of zero implies a numbering type of zero, which therefore will never be different from the type if we are at the bottom of the stack. We know that the numbering levels are equal here, since any level larger than that of the current paragraph would have been taken off the stack during stack unwinding, while the paragraph's level larger than that on top of the stack would have been caught in the previous branch.

If, after all eventual stack adjustments, we end up with a numbering style larger than zero, we are still in a list and output the current paragraph as a list item; otherwise, we are processing a paragraph containing normal text. All that has to happen while we are processing the paragraph sequence of the document's main content:[1]

```
Sub exportContent(iFile%, oContent as Object, sParagraph$, sShift$)
    Dim oParaEnum, oPara as Object
    Dim iParaLevel(32) as Integer
    Dim iListType(32) as Integer
    Dim sListShift(32) as String

    iListLevel% = 0
    iParaLevel(0) = 0
    iListType(0) = 0
    sListShift(0) = sShift
    oParaEnum = oContent.getText().createEnumeration()

    Do While oParaEnum.hasMoreElements()
        oPara = oParaEnum.nextElement()
        iParaNumberingLevel% = paraNumberingLevel(oPara)
        iParaNumberingType% = paraNumberingType(oPara, iParaNumberingLevel)

        While iParaNumberingLevel < iParaLevel(iListLevel)
```

[1]This version has been simplified somewhat compared to the one released in the code to make the basic idea more transparent.

```
            closeList(iFile, sListShift(iListLevel-1), iListType(iListLevel))
            iListLevel = iListLevel - 1
        Wend
        If iParaNumberingLevel > iParaLevel(iListLevel) Then
            openList(iFile, sListShift(iListLevel))
            iListLevel = iListLevel + 1
            iParaLevel(iListLevel) = iParaNumberingLevel
            iListType(iListLevel) = iParaNumberingType
            sListShift(iListLevel) = shift(sListShift(iListLevel-1))
        ElseIf iParaNumberingType <> iListType(iListLevel) Then
            closeList(iFile, sListShift(iListLevel-1), iListType(iListLevel))
            iListType(iListLevel) = iParaNumberingType
            openList(iFile, sListShift(iListLevel-1))
        End If

        If iListLevel > 0 Then
            print #iFile sListShift(iListLevel) & "( : # item"
            exportParagraphContent(iFile, oPara, 1, shift(sListShift(iListLevel)))
            print #iFile sListShift(iListLevel) & ") | item "
        ElseIf oPara.supportsService("com.sun.star.text.Paragraph") Then
            exportParagraph(iFile, oPara, sParagraph, sShift)
        ElseIf oPara.supportsService("com.sun.star.text.TextTable") Then
            ' a table can never be part of a list
            exportTable(iFile, oPara, sShift)
        Else
            MsgBox "Unsupported Text Element"
        End If
    Loop

    While iListLevel > 0
        closeList(iFile, sListShift(iListLevel-1), iListType(iListLevel))
        iListLevel = iListLevel - 1
    Wend

End Sub
```

The function to open or close a list respectively are simply:

```
Sub openList(iFile%, sShift$)
    print #iFile sShift & "( : # list"
End Sub

Sub closeList(iFile%, sShift$, iType%)
    print #iFile sShift & ") | list " & quoteStr(listType(iType))
End Sub
```

The resulting script code might look like this:

```
(
        (
                ...
        ) | item
        ...
) | list 'char_special'
```

The new code requires some new functions:

```
function item   { echo -n "<li>"; cat; }
function list   { style="$1"; shift; block $(list_$style); echo; }

function list_arabic            { echo ol "type='1'"; }
function list_roman_upper       { echo ol "type='I'"; }
function list_roman_lower       { echo ol "type='i'"; }
function list_letter_upper      { echo ol "type='A'"; }
function list_letter_lower      { echo ol "type='a'"; }
function list_char_special      { echo ul; }
function list_char_bitmap       { echo ul; }
```

A list can easily be created by pressing the *Numbering On/Off* or *Bullet On/Off* icon in the OpenOffice standard formatting toolbar.

6.2.3 Advanced Lists

For simple lists, the previous procedure is perfectly sufficient. Even nested lists are handled correctly. This, however, is more by coincidence than by design and the reason we omitted the closing tag of a list item. Nested lists shall be part of a list item, but in the current version, only single paragraphs can become list items. By omitting the closing tag, the item of the previous paragraph stretches until the beginning of the next one, including any possibly nested list. The result is technically perfectly legal, but not very nice; we probably could do better here.

Also, lists can be more complex than that. For example, a list item can contain more than one paragraph. In OpenOffice, such additional paragraphs are set with the same indention, but without a number or bullet, i.e. they have the same numbering level, but no numbering type.[2] If a list item can contain multiple paragraphs, then we need another logical level below the item. Let's call that level an *item element*. Item elements exist to add markup to the entities within an item. That implies, that we not only have to cumulate items into lists, but also item elements into items.

[2]They can be created by pressing either *Insert Unnumbered Entry* icon in the *Bullets and Numbering* toolbar before *Enter* to terminate the preceding paragraph, or *Backspace* just after having pressed *Enter* to create the new paragraph.

Item elements will be mapped to additional `p` or `div` tags within a list item. However, according to the principle of minimal markup such block tags should only be added if there actually are several blocks. If an item contains only a single block, it should be left as plain content. We can't make that decision for the first element in an item before we've seen the next one. Luckily, the filter to process an item element goes into the script after its content, so we can postpone adding it until we know what's coming next and pass a suitable parameter telling it what to do. That makes creating all the blocks a two-step process. In the first step, we just open the required blocks before processing a paragraph:

```
Sub openListItemElem(iFile%, sShift2$)
    print #iFile sShift2 & "( : # itemelem"
End Sub

Sub openListItem(iFile%, sShift1$)
    print #iFile sShift1 & "( : # item"
    openListItemElem(iFile, shift(sShift1))
End Sub

Sub openList(iFile%, sShift0$)
    print #iFile sShift0 & "( : # list"
    openListItem(iFile, shift(sShift0))
End Sub
```

Closing an item element is postponed until we know what's coming next and passes a parameter to the filter.

In the case of a single paragraph in the list item, the filter should pass on its content unchanged.

Usually, if a list item contains several blocks, each one of them should be wrapped into tags. However, there is the special case of nested lists. Lists are flow content and therefore count as a logical block. If there is a nested list, then there is also some preceding text for the previous list item.[3] This text becomes the first item element of the list item. Most style guides leave this text untagged – despite the fact, that it is a block – which feels right, since it's rarely a paragraph of its own. It's easy to instruct the first item element to pass on the text unchanged, once we see that it's followed by a list. Unfortunately, we can't see beyond that list. There might be more paragraphs coming, in which case probably all should be tagged the same way, but there is no way to know that yet. Therefore, if the second block is

[3]There has to be a list item preceding an nested list that started the outer list, otherwise there wouldn't be an outer list at all.

a list, the first will be left untagged, regardless what's coming. According to the standard, stand-alone text is valid flow content, so that's perfectly legal, just not very nice.

Finally, HTML allows *definition lists*, a feature OpenOffice doesn't have natively. A definition list represents a list of terms and their corresponding definitions. It consists of zero or more term-description groups, where each group consists of one or more terms followed by one or more descriptions. To emulate them in an OpenOffice document, we could use the facts that the type of a list is defined by the numbering type of the first paragraph in the list and that paragraphs might have a numbering level but no numbering type. Hence, whenever a paragraph has an increased numbering level but not a numbering type, then we consider it to start a definition list. That paragraph itself and all following paragraphs with the same numbering level (but no numbering type) become the term entries of the new list. The only idea to mark descriptions that I have come up with so far – unless to use styles – is to indent them even further.

Opening a list is a multi-level process now:

```
Sub openListItemElem(iFile%, sShift2$)
    print #iFile sShift2 & "( : # itemelem"
End Sub

Sub openListItem(iFile%, sShift1$)
    print #iFile sShift1 & "( : # item"
    openListItemElem(iFile, shift(sShift1))
End Sub

Sub openList(iFile%, sShift0$, iListLevel%, iNumberingLevel%)
    print #iFile sShift0 & "# list level " & iListLevel & ":" & iNumberingLevel
    print #iFile sShift0 & "( : # list"
    openListItem(iFile, shift(sShift0))
End Sub
```

The same goes for closing a list:

```
Sub closeListItemElem(iFile%, sShift2$, iElemCount%, sElemType$)
    print #iFile sShift2 & ") | itemelem " & iElemCount & sElemType
    print #iFile
End Sub

Sub closeListItem(iFile%, sShift1$, iElemCount%, sElemType$, sItemType$)
    closeListItemElem(iFile, shift(sShift1), iElemCount, sElemType)
    print #iFile sShift1 & ") | item " & quoteStr(sItemType)
    print #iFile
End Sub
```

```
Sub closeList(iFile%, sShift0$, iElemCount%, sElemType$, sItemType$, sListType%)
    closeListItem(iFile, shift(sShift0), iElemCount, sElemType, sItemType)
    print #iFile sShift0 & ") | list " & quoteStr(listType(sListType))
    print #iFile
End Sub
```

Evaluating the list related paragraph properties essentially follows the same pattern as before. First, we close any lists with a higher level list than the current paragraph. The parameters for the filter function are all on the stack:

```
    While iParaNumberingLevel < iParaLevel(iListLevel)
        closeList(iFile, sListShift(iListLevel-1), _
                         iElemCount(iListLevel), sElemType(iListLevel), _
                         sItemType(iListLevel), iListType(iListLevel) )
        iListLevel = iListLevel - 1
    Wend


iParaLevel(iListLevel) = iParaNumberingLevel = paraNumberingLevel(oPara)
iListType(iListLevel) = iParaNumberingType = paraNumberingType(oPara, iParaNumberingLevel)
sItemType(iListLevel) = paraNumberingCharStyle(oPara, iParaNumberingLevel)
sElemType(iListLevel) = " True|False " & paraStyle(oPara)
iElemCount(iListLevel) = 0


    Do While oParaEnum.hasMoreElements()

        ...

        ' check if there are lists to be closed
        While iParaNumberingLevel < iParaLevel(iListLevel)
            ...
        Wend

        'iParaNumberingLevel is >= iParaLevel(iListLevel) here
        'check if there is a new list to be opened
        If iParaNumberingLevel > iParaLevel(iListLevel) Then
            If iListLevel > 0 Then
                ' we are in a list
                If iListType(iListLevel) = 0 Then
                    ' we are in a defintion list
                    If iItemLevel(iListLevel) = 0 Then
                        ' definition list term
                    Else
                        ' definition list description
                    End If
            Else
                ' normal list
                ' close curent item
                ' open next one
        Else
            ' open a new list
        End If
    ElseIf iListLevel > 0 Then
        ' we are in a list, check if the list type has changed
        If (iParaNumberingType <> iListType(iListLevel) Or _
            iListType(iListLevel) = 0) And oPara.NumberingIsNumber _
        Then
            ' new list
            ...
        ElseIf oPara.NumberingIsNumber Or iListType(iListLevel) = 0 Then
            ' same list, new list item
            ...
        Else
            ' same list, same list item, new item element
            ...
        End If
    End If

    ...

Loop
```

6.3 Pre-formatted Text

The HTML browser considers text typically as flowing, i.e. any sequence of spaces, tabs and newline characters is meaningful only to separate words, but has no impact on the layout. Setting the line breaks and the spacing between words is entirely up to the browser. Mostly that's what we want, but there are situations where the layout of the input text should be retained; like in program code for example.

HTML offers a pair of tags to enclose such texts: `<pre>...</pre>`. Within these tags all wide space characters are significant. Typically the browser will display the text in a fixed font. In an OpenOffice document pre-formatted text can be marked by a dedicated paragraph style and exported just like any other paragraph, but in the scripts there are a few things to watch out for.

Firstly and obviously, pre-formatted text must not go through a word wrapper. That's easily omitted, we only need to ensure the output is terminated by a line end. That's required since the paragraph terminating line break is not part of the text in the script, it is usually appended by `wordwrap` instead. Here's a suitable function:

```
function preformatted           { blk pre; echo; }
```

Secondly, the pre-formatted text itself must not contain any line breaks. That's not quite as obvious and has something to do with the way our block building functions work. Most of them indent the input to achieve a neatly nested document structure. That's OK for flowing text but backfires for pre-formatted text, where the additional tabs suddenly become significant. Only the first line of a pre-formatted text block is protected from this effect: It is considered to start behind the opening tag. Any indention eventually inserted by filters would appear before the opening tag and hence still be discardable. If we could cover all the text of a pre-formatted paragraph in one single line behind the opening tag our problem would be solved. Luckily, our character filter already does exactly this for us by replacing all line breaks by the line break tag. With the line break characters removed from the output a paragraph is guaranteed to appear in a single line. (This line is then terminated by the `echo` above.)

While working perfectly fine technically, putting an entire code snipped into one single line is prone to result in rather long lines, which are difficult to read and might cause trouble at some point. Is there a way to have line breaks and tabs in pre-formatted text without affecting the layout? There is, the trick is to put them into comments:

```
function preformatted   { sed 's/<br>/<br><!--\n-->/g' | blk pre; echo; }
```

Since the line end character won't be visible, we still need the line break tag. Now any tab character, inserted immediately after a line break to indent a line, will not have an impact to the layout either:

```
...
                <pre>$ cat source_code<br><!--
                -->for i in; do<br><!--
                -->        somthing<br><!--
                -->done</pre>
...
```

Now all we need is to map the helper function above to the paragraph style name and – perhaps – add a few more formatting features:

```
function Preformatted_Text      { preformatted | blck blockquote; }
```

The remaining issue concerns the typing of pre-formatted text. A contiguous piece of pre-formatted text – like a code snipped – should go into a single paragraph. Hence line breaks must not be entered as *Enter*, that would start a new paragraph, but as *Shift+Enter*.

The use of line breaks in normal text is a bit uncommon and their proper handling requires some getting used to. It becomes really annoying when you cut and paste text, that always interprets line ends as paragraph ends. For such situations I use a little script that replaces all paragraph ends in the current selection by line breaks. It is a useful little helper and by no means limited to the scope of this project; it doesn't hurt to have it in your OpenOffice installed even if you don't do any scripting at all. We wont list it here, you can find it in the appendix. There might be alternative solutions on the Internet as well.

6.4 More Text Attributes

6.4.1 Assembled Text Attributes

So far, we have treated each text portion individually, without considering the text portions next to it. Therefore it can happen, that a certain property is marked up several times, even if it stretches over a sequence of strings. For example: *An italic text* ***with something bold*** *in it*

would be marked up as

```
<i>An italic text </i><i><b>with something bold</b></i><i> in it</i>
```

rather then simply

```
<i>An italic text <b>with something bold</b> in it</i>.
```

While technically there's nothing wrong with that in this example, it leads to ugly code at the very least and might cause more severe trouble in more complex cases, for instance, if we want to assign ids to our markup, which are supposed to be unique and therfore must not be split into several chunks of markup.

6.4.2 Hypertext Links

Links are one of the key features of the World Wide Web. In an OpenOffice document a hyperlink can be assigned to any character sequence within the text flow. They are handled like other character attributes. The *hypertext link dialogue* offers several masks to specify a URL, the most important of which are properly the *Internet* → *Web*, the *Document* → *Path*, and the *Document* → *Target in Document* options. The first one is there to specify a link to a document on any server in the WWW, the second to any document on the same server and the last one to a target within the same document.

If you specify a URL to a server without the protocol part, OpenOffice will automatically prepend it with `http://`; you probably want to change that to `https://` nowadays. Any legal URL is possible, including the path to a specific document and an anchor within the document.

If you specify a path to document on the same server, make sure you have selected the *Save URLs relative to file system* option under *Tool* → *Options...* → *Load/Save* → *General*. Otherwise OpenOffice will store the file's location as absolute path. It is rather unlikely that the absolute path on your web server will be the same as the one on device where you're writing your document. You can open a file browser to pick a file or enter the path manually. The path might contain an anchor within the document.

Even if the path is stored relative to the documents location, when obtained it will be presented as an absolute path, aligned to the current location of our document. It induces some efforts for us to revert that:

```
Function convert2relative(sRef as String) as String
    Dim aRef(), aDoc()
    Dim iRef, iDoc as Integer
    aRef = split(convertFromURL(sRef), "/")
    aDoc = split(convertFromURL(thisComponent.URL), "/")
    iRef = LBound(aRef())
    iDoc = LBound(aDoc())
    While iRef < UBound(aRef()) And iDoc < UBound(aDoc) And aRef(iRef) = aDoc(iDoc)
        iRef = iRef + 1 : iDoc = iDoc + 1
    Wend
    sRef = ""
    While iDoc < UBound(aDoc)
        sRef = sRef & "../" : iDoc = iDoc + 1
    Wend
    While iRef < UBound(aRef)
        sRef = sRef & aRef(iRef) & "/" : Ref = iRef + 1
    Wend
    convert2relative = sRef & aRef(iRef)
End Function
```

If you want to specify a target within the document you need an anchor. OpenOffice offers a choice of predefined anchors for:

- headings
- tables
- text frames
- graphics
- OLE objects
- sections
- bookmarks

You can open a dialogue to pick any of these. However, apart from the last option, you don't really have control over the naming of the target. You have to stick to an OpenOffice internal naming scheme for your anchors and you have to retrieve the names for each type of object individually.

Note, that you can specify a name for each hyperlink. A name is intended to be passed to the *name* attribute of the link tag in an HTML document. That would be the traditional way to declare an anchor. Unfortunately, you can't specify a link with only a name while omitting the URL. That makes it pretty useless, since only very few anchors are likely to be links as well. To specify an anchor, a bookmark is probably the better option.

6.4.3 Bookmarks

A Bookmark is a text content that marks a position inside a text. In an HTML document, bookmarks are converted to anchors that you can jump to from a hyperlink. So, in a way they are the counterpart of a hyperlink within the same document.

6.4.4 Cross-references

Cross-references insert text into a document that is functionally depended on some other text passages or object in the same document[4]. A cross-reference consists of a target and one or more references, that are determined by the target. A target can be:

[4]or within sub-documents of a master document

- a text passages set explicitly as target
- a headline
- the caption of an object like a table or graphic
- a bookmark

A reference, i.e. the text inserted can be there referred text passage itself, but also meta information like the page or chapter number of the target.

To mark some text as target for a cross-reference, select it and choose *Insert* → *Cross-refrence...*, this will open the cross-reference dialogue. In the *Type* list, select *Set Reference*. Type a name for the target in the *Name* box. The selected text is displayed in the *Value* box. Click *Insert*, the name of the target is added to the *Selection* list.

To create a reference to a target, position the cursor in the text where you want to insert it. choose *Insert* → *Cross-refrence...*, this will open the cross-reference dialogue. In the *Type* list, select category of the target you want to insert, in case of a previously marked text passage choose *Insert Reference*. In the *Selection* list, select the target that you want to reference. In the *Insert reference to* list, select the format for the reference. The format specifies the type of information that is displayed, to include the referred text itself choose *Reference*. Click *Insert*, then *Close* to finish.

Technically, a cross-reference is a field. That has the advantage that the reference can be updated automatically, once the target has changed. Pressing *F9* will update all references in the document. To remove a reference, delete the field.

References are not hyperlinks. Hyperlinks are there to transfer the user to a different location, either in the same document or possibly somewhere completely different on the WWW. There is usually not relation from the target back to the link. References on the other hand establish a relation in the opposite direction, they transfer information from (or about) the target back to the reference. That in OpenOffice the user can jump from a reference to the target by a simple click is a mere side effect to ease editing. Consequently, cross-references are not made hyperlinks when the document is saved in HTML. We can, however, still extract the relevant information to create links on our own. This might be quite useful, for example to create a small self-made table of content.

6.5 Hypertext Links

Links are one of the key features of the World Wide Web. In an OpenOffice document you can assign a hypertext link to any character sequence within the text flow. It can be both, a hypertext name as well as a URL. Both values are available as text portion attributes. These behave very much like character style properties and hence can be extracted in the same way. We have to make a choice which one has the preference in case both values are assigned:

```
Function href(oText as Object) As String
    If oText.HyperLinkName <> "" Then
        href = " 'hrefName " + CHR(34) + oText.HyperLinkName + CHR(34) + "'"
    ElseIf oText.HyperLinkURL <> "" Then
        href = " 'hrefURL " + CHR(34) + oText.HyperLinkURL + CHR(34) + "'"
    Else
        href = ""
    End If
End Function
```

The result is passed as an element of the argument list of a text portion, just like any other text attribute:

```
Sub exportText(iFile%, oPara as Object, oText as Object, sShift$)
        ...
        sAttributes = href(oText) + charStyle(oText, oPara)
        ...
End Sub
```

The only thing left to do now is to define the two new style functions:

```
function hrefURL                { blck a "href='$1'"; }
function hrefName               { blck a "name='$1'"; }
```

We pass different function names to the script. That's because names and URLs might have to receive different treatments. You might, for example, want to look up a name in a database to find the target while a URL stands for itself.

Unfortunately, while it's rather simple to retrieve the parameters of a hypertext link once it is in the document, to get it there in the first place is not quite as easy: OpenOffice likes to temper with the URLs typed into the input masks. I guess this is supposed to be user friendly, but in our case it just makes live harder.[5]

[5]There is currently a issue open at OpenOffice.org regarding some of the problems.

6.6 Images

Besides links, non-textual content included into documents was a branding feature of the World Wide Web. Meanwhile nearly all office applications support multi-media content to a varying degree and OpenOffice is no exception.

In OpenOffice non-textual content is put into frames, and so are images. Frames are a fundamental tool to control a document's layout. They can contain nearly everything, including their own text flow. Frames are particular useful to represent document content that is not part of the main text flow – though that's not the only way to use them. We'll find some interesting use cases for frames in due course, for now we're only interested in them as far as images are concerned.

Images, like all frames, are anchored. They can be anchored to a page, to a paragraph, to a character or *as* a character. The first three of these options make the frame effectively part of the page, either at a fixed position on a specific page, or floating, following a reference point in the text flow. However, the frame becomes not a part of the text flow itself, rather the text wraps around the frame. Only in the last case the frame is lined up according to its height and width like any other character in the line.

HTML doesn't know anything about pages, so the first three options don't have an HTML equivalent. If you want to include an image into your text, insert it as a character. Then they appear as non-textual objects in the document's main content enumeration where they are ready for us to be inspected, sparing any additional overhead to find them in the document structure.[6]

```
Sub exportParagraphContent(iFile%, oPara as Object, sShift$)
    ...

    Do While otextEnum.hasMoreElements()
        ...

        If sType = "text" Then
            exportText(iFile, oPara, oText, sShift)
        ElseIf sType = "frame" Then
            exportFrameContent(iFile, oText, sShift)
        Else
            ...
```

[6]Frames anchored to pages or paragraph have their own collection, that can be iterated through their own enumeration objects.

```
        End If
    Loop
End Sub
```

Frames are a general concept and by no means restricted to images. In particular, frames can contain content objects of virtually any type. These objects are accessible via an enumeration. An image inserted into the text flow typically will have its own dedicated frame, so it will be the only object in the collection. However, the collection is still a collection and we have to iterate through it to get any content. Of course, each object we find is checked for the type we're interested in. Later we will handle other frame content as well. Unknown objects are reported the usual way:

```
Sub exportFrameContent(iFile%, oText as Object, sShift$)
    Dim oFrameEnum, oElem as Object
    REM Use an empty string to enumerate all content
    oFrameEnum = oText.createContentEnumeration("")
    do while oFrameEnum.hasMoreElements()
        oElem = oFrameEnum.nextElement()
        If oElem.supportsService("com.sun.star.text.TextGraphicObject") Then
            REM bitmaps or vector oriented images
            exportTextGraphicObject(iFile, oElem, sShift)
        Else
            unknown(iFile, oElem, "frameContent", oText.String, sShift)
        End If
    Loop
End Sub
```

An image needs a reference to the image source in order to allow the browser to find and load it. Often this is file path relative to the including page's location. It can, however, be any URL, possibly referring a location on an entirely different server. Unfortunately, OpenOffice does not allow to specify an arbitrary URL as image source, so what alternatives do we have?

If you have selected the *Link* option in the *Insert picture* dialogue while inserting the image, OpenOffice will have stored the path to the image instead of the image itself.[7] You can retrieve that path and use it as an image source. As already discussed in the section *Hypertext Links*, you probably want to make sure to save paths relative to the document.

Inserting a reference to the file containing an image rather than a copy of the image itself implies that you must make sure to always transfer the image

[7]If you haven't done it, you can correct that later: Select the image and insert it again with the `Link` option checked. OpenOffice will replace the image by a link, keeping all other image setting.

files along with the document. For normal office documents that might be a disadvantage. In our case, however, we'll have to take care of the image files anyway, otherwise our web site won't work, so that's rarely an additional burden.

For simple cases that scheme might be sufficient. However, what if the path to the files shall be a different one on the server than it is in your develop environment, or if you want to specify a hyperlink as an image source instead of a file path?[8] In such cases we might resort to the images name. That name is guaranteed to be unique within an OpenOffice document and therefore can be used for lookup in some kind of key-value database. The default lookup procedure looks for a function of the image name in an emulated image namespace, but you can implement any other lookup scheme, including file or database lookup. However, beware the fact, that OpenOffice will rename images in case a name is used twice, which can lead to some unpleasant surprises. It also implies, that the same image used multiple times in a document needs multiple entries in your data base.

Each image should have an alternative text to give the reader some idea about what's shown in the image in case it can't be displayed or for users using a screen reader. OpenOffice offers attributes for that. Some times back this was only one attribute, nowadays there are two, *Title* and *Description*. Thereby, the *Title* attribute in the picture's *Description* dialogue is identical to the *Alternative Text* attribute in the *Picture*'s *Option* tab. Apparently this scheme has been introduced by Microsoft for whatever reason. That's of course a little confusing, but it offers us another option to select an image sources: Since only one of the attributes can be used to fill the image's alternate text attribute, the other one can serve as an alternative key for image source lookup. Therefore, whenever the *Description* attribute is filled, that content is used as alternate text for the image. If also the *Title* attribute is filled, then it replaces the image name for lookup. This way we make sure OpenOffice doesn't interfere with our naming and we can use the same title multiple times in the document if the image is used multiple times. If the *Description* attribute is not filled, then the *Title* is used as alternative text (since it is identical to the *Alternative Text* attribute in the *Option* tab anyway). If you want to use the *Description* and the pictures *Name* instead of the *Title*, just leave the *Title* empty.

```
Sub exportTextGraphicObject(iFile%, oObject as Object, sShift$)
```

[8]Actually, OpenOffice will not complain if you specify a hyperlink as reference while inserting an image, but it won't be able to retrieve and display it, leaving the document in a rather ugly look.

```
    Dim sURL, sAlt as String
    If len(trim(oObject.HyperLinkName)) <> 0 Then
        sURL = quoteStr(trim(oObject.HyperLinkName))
    ElseIf len(trim(oObject.HyperLinkURL)) <> 0 Then
        sURL = quoteStr(trim(oObject.HyperLinkURL))
    Else
        sURL = CHR(34) & "`imgName " & oObject.Name & "`" & CHR(34)
    End If
    If len(oObject.AlternativeText) <> 0 Then
        sAlt = oObject.AlternativeText
    Else
        sAlt = trim(oObject.Name)
    End If
    print #iFile sShift & "image " & sURL & " " & quoteStr(sAlt)
End Sub
```

In the early days of this tool set I also used the hyperlink URL and the hyperlink name attributes of an image to specify the source URL or some lookup key respectively, since this was the only way to specify a URL directly. Of course that's a hack, since this attributes are intended to specify the target if a user clicks on the image, rather then the image source themselves. I don't use it any more, but both attributes are still passed to the image function. This has partly historical reasons, since some existing documents still use this scheme. However, the default nowadays is to use them in the proper way and wrap the image into a link.[9]

Other than for hypertext links, URL or link name of an image are passed directly to the image function without any further processing. That has mainly historical reasons. The image function as one of the earliest in the tool set does the support any attribute processing. Feel free to change that. If you set up your own environment from the scratch it might be a better choice.

Occasionally you'll want to place an image outside a paragraph. According to the HTML standard that's strictly speaking illegal. Images are a type of content that always should be surrounded by a block, like a paragraph, a block quote, a list, a table etc. It's not a problem, neither in HTML nor in OpenOffice, to put an image as the only content into a dedicated paragraph. Unfortunately, such a paragraph wouldn't compile properly with our tool. Images don't count as text and hence our check for non-empty paragraphs will exclude the paragraph from processing.

[9]Note that the hyperlink name is only stored by OpenOffice if a hyperlink URL is specified. Therefore, the name has to take precedence over the URL. If you only want to use the name, specify something like `file:///`.

I didn't bother to change that, not least because there is a simple workaround: Just put a tab into the paragraph behind the image, that is not stripped away the by trim function, so it makes sure the paragraph is exported, but will be eliminated later by the word wrapper – nice and easy.

If you insist to get an image outside the paragraph tags, there is a simple option too:

```
Sub exportParagraph(iFile%, oPara as Object, sShift$)
    If len(trim(oPara.getString())) > 0 Then
        print #iFile sShift & "("
        exportParagraphContent(iFile, oPara, shift(sShift))
        print #iFile
        print #iFile sShift & ") | paragraph " + paragraphStyle(oPara)
        print #iFile
    Else
        REM no text: check for images and frames
        exportParagraphContent(iFile, oPara, sShift)
    End If
End Sub
```

This will skip the paragraph tags in case there is only non-textual contents.

6.7 Rulers

Plain HTML doesn't know anything about pages, but it has horizontal rulers, a feature that can be used to separate document parts and hence to create a page-like effect. Before OpenOffice offered rulers it seemed only natural to map page breaks to rulers. Nowadays OpenOffice has rulers, but I still stick to the old way. Even if you prefer to explore the new options, being able to handle page breaks is still useful.

A manually inserted page break is marked either by its break type or by the page style assigned to the new page. Both are easily checked. They become properties of the paragraph immediately following that break:

```
Function isBreak(iBreakType As Integer) as Boolean
     isBreak = (iBreakType = com.sun.star.style.BreakType.PAGE_BEFORE)
End Function


Sub exportContent(iFile%, oContent as Object, sShift$)
    Dim iBreakType as Integer
    iBreakType =
    ...
    Do While oParaEnum.hasMoreElements()
        ...
        If iListLevel > 0 Then
            ...
        ElseIf oPara.supportsService("com.sun.star.text.Paragraph") Then
            REM check for page breaks;
            If isBreak(oPara.BreakType) OR len(oPara.PageDescName) > 0 Then
                print #iFile "pagebreak '" & oPara.PageDescName & "'"
            End If
            REM paragraphs
            exportParagraph(iFile, oPara, sShift)
        ElseIf oPara.supportsService("com.sun.star.text.TextTable") Then
           ...
        Else
            ...
        End If
    Loop
End Sub
```

If interpreted as a ruler, the pagebreak function only inserts the ruler tag:

```
function pagebreak        { tag hr; }
```

This simple solution ignores page breaks before tables and lists. That can easily be solved by inserting an empty paragraph just after the break and before the list or table. Within a table or list a page break doesn't make much sense. OpenOffice doesn't allow page breaks in tables anyway and within list we just don't check.

6.8 Footnotes

Footnotes are not directly supported in HTML, but sometimes they come in handy and it's easy enough to emulate them. OpenOffice supports footnotes and endnotes. The appearance a of both is nearly identical, only that footnotes appear on the bottom of the page while endnotes are collected at the end of the document. Because HTML doesn't know anything about pages we have no chance to position a footnote properly, hence endnotes are a better choice here. Nevertheless, we keep talking about footnotes whenever the difference doesn't matter – the term is simply more common – bearing in mind that whenever we're talking about footnotes in the context of HTML we're actually referring to endnotes in OpenOffice.

A footnote consists of two components, a footnote marker and the footnote text.[10] The footnote text comments on a part of the main body text. The marker is normally a superscript number inserted into the text flow just behind the point the footnote is referring to, as well as a flag just in front of the footnote text. Within the main text flow it appears as special text portion:

```
Sub exportParagraphContent(iFile%, oPara as Object, sShift$)
    ...

    Do While otextEnum.hasMoreElements()
        ...

        If sType = "text" Then
            exportText(iFile, oPara, oText, sShift)
        ElseIf sType = "footnote" Then
            exportFootNote(iFile, oPara, oText, sShift)
        ElseIf sType = "frame" Then
            exportFrameContent(iFile, oText, sShift)
        Else
            ...

        End If
    Loop
End Sub
```

The export itself looks very much like that of an ordinary text portion, only that it gets an additional endnote reference attribute:

[10]Note, that OpenOffice occasionally refers to them as footnote text and footnote area respectively.

```
Sub exportFootNote(iFile%, oPara as Object, oText as Object, sShift$)
    If len(oText.getString()) > 0 then
        Dim sText, sAttributes as String
        sText = quoteStr(oText.getString())
        sAttributes = "endnoteref " & dquoteStr(oText.Footnote.ReferenceId)
        sAttributes = quoteStr(sAttributes) & charStyle(oText, oPara)
        print #iFile sShift & "text " & sAttributes & " '' " & sText
    End If
End Sub
```

The `endnoteref` function formats the marker in an appropriate way by setting it into superscript and transforms it into a hypertext link. The link target is an anchor within the same document marking the endnote text. As target name we use the endnote reference identifier maintained by OpenOffice:

```
function endnoteref        { blck a "href='#$1'" | blck sup; }
```

The endnote reference identifier is an integer number. It starts counting at 1 for each document. Hence it will be unique only within the same OpenOffice document. While OpenOffice allows you to add an offset to the marker, that doesn't apply to the identifier. If you intend to use several OpenOffice documents as sources for a single HTML document you'll need to take care of that.

The endnotes are accessible through their own document wide collection. They are processed after the main body and appended at the end of the document. Note, that the end of the document actually means the end of the current OpenOffice text, that is not necessarily the end of the final HTML document:

```
Sub exportDocument(iFile%, oDoc as Object)
    ...
    REM export document content
    exportContent(iFile, oDoc, "")
    REM add endnotes at the end of the document
    exportEndNotes(iFile, oDoc.getEndnotes(), "")
    ...
End Sub
```

The endnote export function collects the text of each endnote, their marker, and their identifier and hands all these to an endnote formatting function:

```
Sub exportEndNotes(iFile%, oEndNotes as Object, s0Shift as String)
    Dim sText, sMarker, sID, s1Shift, s2Shift as String
    If oEndNotes.getCount() > 0 Then
        s1Shift = shift(s0Shift)
        s2Shift = shift(s1Shift)
        print #iFile s0Shift & "( :"
        For i = 0 To oEndNotes.getCount() - 1
            sText = quoteStr(oEndNotes.getByIndex(i).getString())
            sMarker = quoteStr(oEndNotes.getByIndex(i).getAnchor().getString())
            sID = quoteStr(oEndNotes.getByIndex(i).ReferenceId)
            print #iFile s1Shift & "( :"
            print #iFile s2Shift & "text '' " & sText
            print #iFile s1Shift & ") | endnote " & sID & " " & sMarker
        Next
        print #iFile s0Shift & ") | endnotes"
        print #iFile
    End If
End Sub
```

Each endnote is preceded by its marker, which is turned into an anchor. All endnotes go into a single paragraph separated by line breaks. They are preceded by a thin left aligned line and set slightly smaller then the main text.

```
function endnote {
        bl "$2" a "name='$1'" | blck sup
        wordwrap; tag br
}

function endnotes {
        tag hr noshade color=black size=1 width=100 align=left
        block small | blck p
}
```

6.9 Bibliography

A bibliography is a list of books, articles, web pages and other publications that have been cited or referred to in the current document. Each entry in the bibliography is a citation. Technically, citations have a lot in common with footnotes: There is a maker in the text flow – typically a number or some abbreviated identifier – pointing to the appropriate entry in the bibliography. The entry then provides sufficient information to identify the cited source uniquely. Unlike the footnote, however, the bibliographic reference is not in free text form, but a record in a predefined format.

OpenOffice comes with its own means to maintain bibliographic references, either as part of a document or in a system wide database. No matter which one we use, within the text flow a citation appears as a text field:

```
Sub exportParagraphContent(iFile%, oPara as Object, sShift$)
    ...
    Do While otextEnum.hasMoreElements()
        ...
        If sType = "text" Then
            ...
        ElseIf sType = "footnote" Then
            ...
        ElseIf sType = "textfield" Then
            exportTextField(iFile, oPara, oText, sShift)
        ElseIf sType = "frame" Then
            ...
        Else
            ...
        End If
    Loop
End Sub
```

A text field is a special type of text content, used to insert information into a document, that comes from some other source, possibly an external one. Text fields can include the current date or time, document meta information like title or author, the result of a database query and so on. We don't need to bother where the text actually comes from, we can take and treat it as any other text. That's the default. Occasionally, however, it might be worth to consider the additional information a text field offers. Currently we're only interested in citations, though sooner or later we might find good uses for some of the others as well:

```
Sub exportTextField(iFile%, oPara as Object, oText as Object, sShift$)
    Const Bibliography = "com.sun.star.text.TextField.Bibliography"
    If oText.TextField.supportsService(Bibliography) Then
        iValId$ = com.sun.star.text.BibliographyDataField.IDENTIFIER
        sText$ = quoteStr(oText.TextField.Fields(iValId).Value)
        print #iFile sShift & "bibref " & sText & charStyle(oText, oPara)
    Else
        exportText(iFile, oPara, oText, sShift)
    End If
End Sub
```

Usually the form of a citation marker follows a certain pattern. The required formatting is done in the `bibref` function before the result is handed to the familiar `text` function, where the remaining formatting is done. Finally, the marker is transformed into a link, just like a footnote.

```
function bibref {
        label="$1"; shift;
        text "$@" '' "[${label}]" | blck a "href='#${label}'";
}
```

Assembling the bibliography itself as a list of citations sets an unexpected challenge: Just like endnotes, text fields are accessible through their own collection, but the entries do not appear to be in any particular order. Also, sources cited more than once in the document will appear multiple times in the bibliography. We need to think of a way to sort them and remove duplicate entries.

A second problem is the mere number of attributes that can be specified for bibliographic reference; the input mask offers about 30 possible choices. Passing all of them as a positional parameter is confusing and error prone, especially since most of them are optional and only a few might actually be specified. Passing them as named arguments would avoid this problems.

One possibility to achieve named arguments are environment variables. If we could define an environment variable for each specified attribute, then they could easily be tested for existence and the output format adopted accordingly:

```
function bibentry {
        bl "[${Identifier}]" a "name='${Identifier}'" | blck dt
        (
                test -n "$Author" && { text bold '' "$Author:"; echo; }
                test -n "$Title" && { text italic '' "$Title"; }
                test -n "$Address" && { echo ","; text '' "$Address"; }
                test -n "$Publisher" && { echo ","; text '' "$Publisher"; }
                test -n "$Year" && { echo; text '' "($Year)"; }
                test -n "$ISBN" && { echo ","; text '' "ISBN: $ISBN"; }
                echo "."
        ) | blck dd
}
```

This example formats a citation as definition list entry. That might not be the best choice, but it is simple and it's easy enough to change.

The formatting of the entry record itself is rather simple. It is, however, about the level of complexity we can reach with reasonable effort in a shell. We might actually have reached the point to think about other formatting tools. The next tool to explore are properly style sheets, but for the sake of simplicity we spare it for now.

To get the attributes into the environment we iterate through all text fields in the document, select the bibliographic entries, iterate over there attributes and create an assignment command for each attribute we find, using the attribute name as variable name and quoting the value[11]. The commands are collected into a single line before printing and go as a *here document* into the script, feed to a bibliography formatting function.

```
Sub exportBibliography(iFile%, oTextFields as Object, sShift$)
    Const Bibliography = "com.sun.star.text.TextField.Bibliography"
    Dim sName, sValue, sRecord as String
    Dim oTextFieldsEnum, oTextField as Object
    oTextFieldsEnum = oTextFields.createEnumeration
    If oTextFieldsEnum.hasMoreElements() Then
        print  #iFile "bibliography" + " << $"
        Do While oTextFieldsEnum.hasMoreElements()
            oTextField = oTextFieldsEnum.nextElement()
            If oTextField.supportsService(Bibliography) Then
                sRecord = ""
                For i = LBound(oTextField.Fields) To Ubound(oTextField.Fields)
                    If len(trim(oTextField.Fields(i).Value)) > 0 Then
                        sName = oTextField.Fields(i).Name
                        sValue = quoteStr(oTextField.Fields(i).Value)
                        sRecord = sRecord & sName & "=" & sValue & ";"
                    End If
                Next
                print #iFile sRecord
            End If
        Loop
        print #iFile "$"
        print #iFile
    End If
End Sub
```

The result would look something like this:

```
bibliography << $
Identifier="B02"; Author="Kirk, James T."; Publisher="starfleet"; ...;
Identifier="..."; Author="..."; ...
...
$
```

Having all attributes of an entry in a single line makes it easy to use UNIX standard utilities to get a sorted list without duplicate entries. Since this happens during the runtime of the script it inserts an additional layer of

[11]If you intend to run the entry formatting function into a separate shell, then you might have to use the `export` syntax, that has been skipped here for simplicity.

meta programming, in this case implemented using the shell `eval` command. The `unset` is used to clean out the environment before the attribute variables are defined, otherwise we might evaluate variables that have actually been defined outside our script or by a previous entry:

```
function bibliography        {
        sort | uniq | while read para; do
                unset Identifier Author Title Address Publisher Year ISBN
                eval $para
                bibentry
        done | block dl
}
```

Last but not least, the bibliography is exported at the end of the document, just after all endnotes:

```
Sub exportDocument(iFile%, oDoc as Object)
    ...
    REM export document content
    exportContent(iFile, oDoc, "")
    REM add endnotes at the end of the document
    exportEndNotes(iFile, oDoc.getEndnotes(), "")
    REM add bibliography at the end of the document
    exportBibliography(iFile, oDoc.getTextFields(), "")
    ...
End Sub
```

Chapter 7

Publishing in Print

Assuming you've created an interesting web site that has become quite popular. One fine day you get a phone call from a well respected magazine. The editor offers to publish one of your articles if you can provide it in a form more suitable for printing. Or you want to have some more control over the layout of your document in the first place, to have more options to underline your ideas. Or you need features HTML simply doesn't offer, let's say you need to display complex mathematical formulas. In all that cases you might prefer to use a more sophisticated type setting system.

Whoever wants to talk about type setting needs to talk about TEX. TEX is a software package designed to produce documents of the finest quality. Many people would agree that the results are the best you can currently expect from a computer; and above all: it's for free.

TEX has been developed by Donald E. Knuth, one of the fathers of computer sciences and probably best known for his classic series of books on *The Art of Computer Programming*. He started working on TEX in 1977, after being rather unhappy with the galley proofs he had received for the second volume of the very same series. They looked awful—because printing technology had improved dramatically. How was that possible?

In traditional type setting an author hands a typed or even hand-written manuscript to the publisher. The publisher's layout designer decides how the text should be formatted, how much space to leave for margins and between lines and sections, which font to use and so on. Then, manuscript and layout are handed to a typesetter, who fills the spaces defined by the designer with the text provided by the author. Designer and typesetter used to be well trained staff. They have learned to design a layout and set a text to get a result that is not just pleasing the eye, but also not tiring it more then necessary. They were trained to spot problems and pitfalls, based on the experience gained in about 500 years of book printing.

Meanwhile, photographic typesetting was available, allowing to produce a lead matrix directly from the original. Also, the first word processors and high-resolution printers came up, so that authors could provide there manuscripts as a ready-to-use master copy. That cut printing cost tremendously. Unfortunately, it also took the most skilled experts out of the production process. Authors hardly ever get deeper into the finer points of book printing.

TEX tries to fill that gap. Donald E. Knuth really started to study book printing. He has had help from the most respected experts in the field when designing the package. It is said, that some 9500 rules found there way into that program. As such, Knuth's *TEXbook* makes an interesting reading, even if you never intend to use TEX.

In 1985, Leslie Lamport published LATEX, a macro library build on top of TEX. While TEX is very good at type setting, it still leaves many layout tasks to the user. That's were LATEX comes in. It offers a higher level of abstraction to describe layout aspects and uses this information to instruct TEX how to set the text: LATEX is your layout designer, TEX your typesetter. LATEX too has been designed with extensive help from professional book printers and layout designers, so it's usually a good idea to trust it and not to temper too much with layout design yourself, unless—of course—you know exactly what you're doing.

If TEX and LATEX are that good, then why is not anybody using them? Why are still other text processors out there and—let's face it—far more common and popular? Well, there is nothing like a free lunch in this world. TEX is a computer program with no real intelligence. It can't guess the meaning or the structure of your document, but it needs to know such things in order to do a proper formatting job, so you need to tell and that induces efforts.

TEX uses a mark-up language to describe a document structure, just like HTML does. The TEX-language is designed to ease typing, so it's far easier to type a TEX-manuscript into a terminal than it is for an HTML document. But the fundamental disadvantages of a mark-up language remain: they are designed more for computers then for human beings. Format tags and special character encodings require keywords to be learned. Their effect is not apparent in the manuscript, they interfere with spell checkers and disrupt the text flow when proof reading. But hey, we spent the last two sections striving to solve exactly that kind of problems, and we have achieved a great deal with respect to HTML documents. Wouldn't it be possible to go the same way to produce TEX documents? It is, and it's only a little step to go.

We're going to produce LATEX documents here. You may consider LATEX as one specific TEX dialect. The LATEX commands differ slightly from plain TEX, but the basics remain the same, regardless which version you're going to use. In the text below we'll refer to both terms, depending whether an issue or a solution is LATEX specific or applies to TEX in general.

7.1 Producing TEX Documents with OpenOffice

All the OpenOffice scripting we did in the previous sections aims to retrieve the structure of an OpenOffice document, pick the bits of it we are interested in, encode it into a shell script and run it to produce the desired output. The document structure is not going to change, neither is the analysis part and hence our OpenOffice code. What's going to change are the shell script functions producing the actual result. Of course, we don't want to lose the ability to write HTML documents, so we are not going to throw away what we've already got; we provide an additional set of shell script functions instead. So far, with only one output format available, it was hard coded into the scripts. Now we need to make a choice and that's about the only thing we have to change on the OpenOffice part: making sure our script can be parametrized:

```
Sub exportDocument(iFile as Integer, oDoc as Object)
    print #iFile "#!/bin/bash"
    print #iFile
    print #iFile "case " & dquoteStr("$1") & " in"
    print #iFile "    report)   source oo2report.styles; shift;;"
    print #iFile "    article)  source oo2article.styles; shift;;"
    print #iFile "    html)     source oo2html.styles; shift;;"
    print #iFile "    '')       source oo2html.styles;;"
    print #iFile "esac"
    print #iFile
    print #iFile "for i; do"
    print #iFile "    source " & dquoteStr("$i")
    print #iFile "done"
    print #iFile

    ...

End Sub
```

This prologue looks at the script's argument at runtime and loads the appropriate include files. Without any parameter it falls back to HTML, maintaining backwards compatibility. You can also specify additional or entirely different include files.

In case you're wondering why there are predefined includes for reports and articles but none for LaTeX: both are LaTeX document classes. The availability and the effect of certain features in LaTeX depends on the document class: A level-1 headline, for example, requires different LaTeX commands in different document classes. We'll see the details soon. Such dependencies go into the class-specific include file, the general LaTeX stuff is included from there.

Let's turn to actually producing some text. LaTeX uses a different encoding scheme for special characters, so the first thing to be adopted is the character filter. That's pretty easy. All we need is a new list of encodings. We can copy that from any text book or from the Internet. We use, of course, the Unicode version of our filter, hence we have to split it into two files. They are translated into a lookup table and lookup directory respectively—using the very same scripts we already used for the HTML filter—, compiled and linked:

```
$ cat latexcodes | bash charcodes.sh >latexcodes.c
$ cat latexlookup | bash lookup.sh >latexlookup.c
$ cc -o text2latex text2codes.c latexcodes.c latexlookup.c
```

When preparing the character code list please bear in mind that the entries become C-strings without any further processing. TeX heavily uses the backslash and occasionally double quotes. Both must be quoted properly in a C-string and hence in the character encoding input files.

The character filter is called by the `text` function, which OpenOffice compiles into the document scripts for each text portion. The LaTeX equivalent of that function looks exactly the same as for HTML documents, only with the filter being replaced:

```
function text
{
        TAG="$1"; shift
        if test "$TAG" != ''; then
                text "$@" | eval $TAG
        elif test $# -gt 0; then
                echo -n "$@" | text2latex
        else
                text2latex
        fi
}
```

The next level up in the OpenOffice document structure are paragraphs. Like for HTML documents, the `paragraph` function itself is just a wrapper to the appropriate paragraph style functions. That wrapper doesn't change at all, we can reuse the version we already have:

```
function paragraph              { eval "$@"; }
```

In TEX a simple paragraph doesn't need any explicit mark-up. Any sequence of words followed by one or more blank lines is considered to be a paragraph. That would make the traditional helper function, on which standard paragraph styles are based, very simple:

```
function para                   { wordwrap; echo; }
```

Unfortunately, that's not the whole story, because it ignores alignment attributes. Unlike in HTML, in TEX the alignment is not just a parameter to a tag; there's not even a tag the parameter could be bound to. Expressing alignment requires an additional filter to be inserted into the pipe line and that leads to structure very similar to the `text` function:

```
function para
{
        TAG="$1"; shift
        if test "$TAG" != ''; then
                eval $TAG | para "$@"
        else
                wordwrap; echo;
        fi
}
```

The `wordwrap` filter doesn't need to change for TEX documents. Note, however, that filters are inserted from left to right, in contrast to the `text` function above. The implementation of the filters itself shall be postponed for a moment, for now we only need the `align` function that appears in the document script and has to provide the filter name. The name is derived from the alignment attribute. The prefix shall avoid name clashes:

```
function align                  { echo "align_$1"; }
```

Having this, he paragraph style functions are as simple as always:

```
function Text_body              { para "$@"; }
function Standard               { para "$@"; }
```

For plain text without any highlighting or specific alignment requirements that would already be sufficient. Running such a script created from an OpenOffice document with an appropriate include file containing the functions above will create a LaTeX source fragment. But that's not yet a complete LaTeX source file and there comes the bad news: Providing a filter to transform a fragment into a source file is by far not as elegant as it has been for HTML. LaTeX requires much more information to be put into the header part of a document then HTML does, and collecting all this in, or passing it to the filter is possible, but it doesn't seem to pay off. There are simpler ways to do this.

TeX comes with its own means to structure source files and we are going to use them. We just write a master document the traditional way using a good old text editor, and include our generated fragments where appropriate. While HTML tags are quite tedious and error prone to type, typing TeX is much easier and TeX is likely to complain if we made mistakes. Also, for HTML documents the master file needs to be a script since some program has to run in order to actually perform the inclusion. Now, the TeX program does the job for us. You can still write a script to produce the master file if you like, but I found it's not worth the effort.

The minimal version of a master document suitable for experimenting is fairly simple:

```
$ cat main.tex
\documentclass{article}
\begin{document}
\input{...}
\end{document}
$
```

The `\input` command gets the file name of your generated fragment file as a parameter. That's it.

For a real document there is more to be put into the master file: Additional packages, layout parameters, the title page, the abstract and the table of contents are only a few examples. You'll find the details in any good LaTeX book. Here's the master document I used for this documentation:

```
$ cat main.tex
\documentclass[11pt, notitlepage]{article}

\usepackage[ascii]{inputenc}
\usepackage[T1]{fontenc}
\usepackage{amsmath,amssymb,amsfonts,textcomp}
```

```
\widowpenalty=6000
\clubpenalty=4500

\title{Creating HTML documents\\using UNIX shell scripts}
\author{Andreas Harnack}

\begin{document}
\maketitle
\input{abstract}
\eject

\tableofcontents
\eject

\include{chapter1}
\include{chapter2}
\include{chapter3}
...
\end{document}
$
```

We assume you know how to run LaTeX on your computer. If in doubt, please consult your local manual for details. To get a PDF document on my system I simply enter:

```
$ pdflatex main.tex
```

Congratulations! If you have typed a simple, plain text in OpenOffice you should now have derived your first TeX document. Of course, that's not really exciting yet, the interesting stuff is still to come. We'll go through most of the features we've discussed for HTML and see how we can use them in TeX. Not all of them will make sense while eventually we'll reach the point where we feel some others should be added. That's natural: Different forms of publication will require different features.

Please pause for a moment and remember that we never intended to write a universal tool to translate any OpenOffice document into some other format. That won't work. The intention is rather the other way round: using OpenOffice as a front-end to create documents in a specific (foreign) format. Specific formats do have specific features. So don't expect to take any arbitrary text in OpenOffice and get a nice TeX document. Technically that might work, but to produce the best result you will have to invest some editorial work.

7.2 Text and Paragraph Attributes

One of the major advantages of LaTeX over TeX is the high level elements available to describe a document's structure. These elements can be commands or environments. Commands can have optional or mandatory arguments enclosed in brackets and braces respectively. We start with two helper functions to get that syntax straight:[1]

```
function args {
        echo -n "$1"; shift
        for i; do echo -n ",$i"; done
}

function cmd {
        CMD="$1"; shift; PAR="$1"; shift; ARG="`args $@`"
        test "$CMD" == "" || echo -n "\\$CMD"
        test "$ARG" == "" || echo -n "[$ARG]"
        test "$PAR" == "" || echo -n "{$PAR}"
}
```

Most commands appear as part of the text flow, some however stand on a line of there own:

```
function command        { cmd "$@"; echo; }
```

Environments are named document parts. They define some special treatment for the text they contain and are enclosed in a pair of `begin`/`end` commands:

```
function begin          { CMD="$1"; shift; command "begin{$CMD}" "$@"; }
function end            { command end "$@"; }
```

With this helper functions we're ready to implement the block building tools we're already used to:

```
function bl             { cmd "$@"; echo -n '{'; sed "\$s/\$/\}/"; }
function blck           { cmd begin "$@"; sed "\$s/\$/\\`cmd end $1`/"; }
function block          { begin "$@"; cat; end "$1"; }
```

They in turn are used to define the formatting functions for hard and soft character formatting:

[1]The following functions are among the useful helpers that come in handy in a script to create the master document, just in case you'd prefer to do that.

```
function bold                   { bl textbf; }
function italic                 { bl textit; }
function fixedfont              { bl texttt; }

function Strong_Emphasis        { bl textbf; }
function Emphasis               { bl textit; }
function Source_Text            { bl texttt; }

...
```

The paragraph alignment filters look pretty much the same:

```
function align_left             { blck flushleft; }
function align_right            { blck flushright; }
function align_center           { blck center; }
```

Remember our little example sentence? Here is how it looks in LaTeX:

```
\begin{center}A \textbf{bold} \texttt{fixed font} in
\textit{italic}.\end{center}
```

Here the effect of the word wrap is apparent: The second line contains one single token that wouldn't fit on the previous line, so the text is wrapped appropriately.

7.3 Headlines

In OpenOffice a headline is just a special paragraph style. To a certain extend that holds for HTML as well. Not so for LaTeX. LaTeX emphasise the structure of a document. Here the term *headline* doesn't even exist. Instead, a document is organized as a hierarchy of sectional units, like parts, chapters and sections. Each sectional unit has a title and begins with a sectioning command. The title is passed as an argument and represents the equivalent of an OpenOffice headline.

Technically, however, there is not much of a difference. We can define a headline helper function the same way we did before:

```
function hl             { bl "$@" | wordwrap; }
```

Which sectioning commands are available and how the title appears in the document depends on the document class. Not all commands are available for all classes and the mapping between headline levels and sectioning commands will vary from class to class. That's the reason for the document class specific include files we've seen above. Here is the one I use for reports:

```
$ cat record.styles

source oo2latex.styles

function Heading_1              { hl chapter; }
function Heading_2              { hl section; }
function Heading_3              { hl subsection; }
function Heading_4              { hl subsubsection; }
function Heading_5              { hl paragraph; }
function Heading_6              { hl subparagraph; }

$
```

Note how the general, non-class-specific functions are inserted.

7.4 Lists

LaTeX offers three list environments, two of them are supported by OpenOffice: `enumerate`d and `itemize`d lists. It took some effort to extract the list structure from a given document and create some suitable script output, but we have managed to do that, so it shouldn't be a big deal to create at least simple list:

```
function list                   { block `"$@"`; echo; }
function item                   { bl item | wordwrap; }

function arabic                 { echo enumerate; }
function roman_upper            { echo enumerate; }
function roman_lower            { echo enumerate; }
function letter_upper           { echo enumerate; }
function letter_lower           { echo enumerate; }
function char_special           { echo itemize; }
function char_bitmap            { echo itemize; }
```

This approach can only distinguish between the two basic list types, the list layout—and specifically the numbering layout—are left to LaTeX. That might not be the worst of all choices, the result typically is quite satisfying and this solution has the advantage of being simple and straight forward. If you should have special requirements for your list layouts: the possibilities LaTeX offers are really rich, but it requires some LaTeX programming and that is beyond the scope of this document.[2]

[2]An alternative approach would be to extract the label string OpenOffice generates and use this to format your lists, but I have to admit I did not yet find an easy way to do that.

7.5 Pre-formatted Text

LaTeX provides the `verbatim` environment to typeset things like source code. I was, however , not too pleased with the results and prefer to do it my own way:

```
function linebreaks    { sed 's/\\\\/'"\n"'/g'; }
function notabs        { sed 's/'"\t"'/        /g'; }
function truespaces    { sed 's/ /~/g'; }

function hbox  {
        awk '{printf("\\hbox{\\texttt{%s\\strut}}\n", $0)}';
}

function sourcecode
{
        echo '\bigskip'
        linebreaks | notabs | truespaces | hbox
        echo '\bigskip'
}
```

The pipe line in the script changes all encoded line ends back into real ones, replaces each tabulator by 8 spaces, changes all spaces in non-vulnerable once and finally sets each line in typewriter font into a horizontal box. The `\strut` at the end of that box adds an item with zero width and the maximum height of the font, making sure all horizontal boxes have the same height. All that is done by some nice little helper functions. The horizontal boxes are lined up vertically with some additional space added above and below. There is, of course, no point to use `wordwrap`:

```
\bigskip
\hbox{\texttt{for~i~in;~do\strut}}
\hbox{\texttt{~~~~~~~~do_somthing\strut}}
\hbox{\texttt{done\strut}}
\bigskip
```

This provides the core functionality for the respective OpenOffice paragraph styles. They can be extended to specify additional formatting. The following example adjusts the font size:

```
function Preformatted_Text      { sourcecode | block footnotesize ; }
```

For actually typing pre-formatted text in OpenOffice the same guidelines apply as for HTML documents.

7.6 Footnotes

For HTML documents we deployed endnotes to add remarks to a document. It is absolutely no problem to apply the same technique to TEX documents. It's even an added bonus since plain TEX doesn't offer support for endnotes:[3]

```
function endnote        { bl footnotetext '' "$1" | wordwrap; }
function endnotes       { cat; echo; }
function endnoteref     { cmd footnotemark '' "$1"; }
```

This uses low level commands to set a reference mark and the endnote text. The numbering is done by OpenOffice. No special treatment is required for the end-of-document listing of endnotes, so the corresponding functions just forwards anything it gets.

But we want is more: TEX breaks the text into pages and hence is perfectly capable to add footnotes at the bottom of a page, we only need to adjust the export accordingly. The footnote text must be known by the time the page break occurs. We don't know in advance when this is going to be. It might happen to be straight after the point where the footnote reference has been inserted. Listing the footnote text at the end of the document will definitely be too late for most footnotes.

TEX provides a footnote command that inserts a foot reference and gets the footnote text as argument. The footnote numbering and their positioning on the page is done automatically. We only need to pass the footnote text whenever a reference is inserted instead of at the end of the document:

```
Sub exportNote(iFile%, oNote as Object, sNote$, sStyle$, sShift$)
    Dim sID, sLable as String
    If len(oNote.getAnchor().getString()) > 0 Then
        sID = " " & quoteStr(oNote.ReferenceId)
        sLable = " " & quoteStr(oNote.getAnchor().getString())
        print #iFile sShift & "( :"
        exportContent(iFile, oNote, shift(sShift)
        print #iFile sShift & ") | " & sNote & sID & sLable & sStyle
    End If
End Sub
```

Technically there is little difference between footnotes and endnotes, so it should be possible to treat both the same way. That's the reason the function above has been parametrized. To distinguish endnotes from footnotes we can look for a set of additional services it offers:

[3]There is an add-on package available for LATEX but it seems to me to be a bit cumbersome, so I never really bothered to try it.

```
Sub exportFootNote(iFile%, oText as Object, sStyle$, sShift$)
    If len( oText.getString()) > 0 Then
        If oText.Footnote.supportsService("com.sun.star.text.Endnote") Then
            exportNote(iFile, oText.Footnote, "endnoteref", sStyle, sShift)
        Else
            exportNote(iFile, oText.Footnote, "footnoteref", sStyle, sShift)
        End If
    End If
End Sub
```

This will generate the code inserting the reference marks, ensuring the footnote text is available if needed.

Apart from simplifying our export macro, the common treatment of footnotes and endnotes allows us to actually defer the decision about the handling of remarks in the final document until the execution of the generated script. To keep that option open we're going to add the footnote text at the end of the document as well. We want to use the same exporting function as for endnotes, so it needs to be parametrized first:

```
Sub exportNotes(iFile%, oNotes as Object, sNote$, sNotes$, sShift$)
    Dim sLabel, sID as String
    If oNotes.getCount() > 0 Then
        print #iFile sOShift & "( :"
        For i = 0 To oNotes.getCount() - 1
            exportNote(iFile, oNotes.getByIndex(i), sNote, "", shift(sShift))
        Next
        print #iFile sShift & ") | " & sNotes
        print #iFile
    End If
End Sub
```

Like for endnotes, OpenOffice provides a document wide collection for all footnotes in the document:

```
    ...
    REM add footnotes at the end of the document
    exportNotes(iFile, oDoc.getFootnotes(), "footnote", "footnotes", "")
    REM add endnotes at the end of the document
    exportNotes(iFile, oDoc.getEndnotes(), "endnote", "endnotes", "")
    ...
```

Of course, now the text for each footnote or endnote appears twice in our document script, once at the place where the reference is inserted and once at the end of the document. But unless you put half the things you have to say into notes—which is a bad idea anyway—that won't hurt.

To produce endnotes the same way as done so far, we only need to make sure the reference function gets rid of the unwanted input:

```
function endnoteref     { cat >/dev/null; cmd footnotemark '' "$1"; }
```

The rest remains the same. Footnotes are handled exactly vice versa. Now the reference function processes the input while the footnote text at the end of the document is thrown away:

```
function footnoteref    { bl footnote; }
function footnote       { cat >/dev/null; }
function footnotes      { cat >/dev/null; }
```

But we have all option to change that default behaviour. To change all footnotes into endnotes, for example, we can just define:

```
function footnoteref    { endnoteref "$@"; }
function footnote       { endnote "$@"; }
function footnotes      { endnotes "$@"; }
```

This will be particularly interesting for HTML documents. We are no longer forced to write them using endnotes. The final HTML document will still contain only endnotes, but we are free to use footnotes to create them. Or you can deliberately put remarks into a source document that will come out as footnotes in print but as endnotes in HTML. Only mixing both forms doesn't work too well since the numbering schemes are not compatible. I'm sure that can be fixed, but then, having both, footnotes and endnotes in the same document seems rather unlikely.

7.7 Text Fields

A text field is a special text content item. It inserts text content coming from somewhere outside the document. That could be document meta information or data maintained in an external database. Often such information is not known at the time of writing or might change in future.

In most cases we don't need to worry about the specifics of a text field. OpenOffice will do the right job behind the scene. We only need to take the result and insert it as text string into the target document, that's the approach we used for HTML documents. Occasionally, however, as we've seen when compiling bibliographies, the additional information a text field carries might be of interest.

7.7.1 Bibliographic References and the Bibliography

A bibliographic reference is a text field holding a key identifying a record in a bibliographic database. We used the key to create the reference mark in HTML documents, that only required a bit of formatting.

LaTeX offers advanced techniques to create bibliographies. This includes its own bibliographic data bases. We could use them, of course, but we are already using the OpenOffice bibliographic data base for our HTML documents, We better stick to it until we have a very good reason to change that. We leave the formatting to LaTeX, though.

There is a LaTeX command to insert a citation reference. It gets a key and produces the reference mark. The key is used internally to provide the correct mapping between the reference and the corresponding entry in the bibliography. It can be any string. The reference mark will be a number, unless specified otherwise. The numbering is done automatically:

```
function bibref         { echo -n "$1" | bl cite; }
```

Formatting an entry for the bibliography implies a certain complexity, due mainly to the varying number of components:

```
function bibentry {
    command bibitem "$Identifier"
    test -n "$Author" && { text '' "$Author"; echo ":"; }
    test -n "$Title" && { text italic '' "$Title"; }
    test -n "$Address" && { echo ","; text 'bl hbox' '' "$Address"; }
    test -n "$Publisher" && { echo ","; text 'bl hbox' '' "$Publisher"; }
    test -n "$Organizations" && { echo ","; text '' "$Organizations"; }
    test -n "$Year" && { echo; text 'bl hbox' '' "($Year)"; }
    test -n "$ISBN" && {
        echo ","; echo -n ", ISBN: $ISBN" | sed 's/-/{--}/g' | bl hbox;
    }
    echo "."
}
```

Compiling the whole bibliography, on the other hand, is fairly simple:

```
function bibliography {
    sort | uniq | while read para; do
        unset Author Title Address Publisher Organizations Year ISBN
        eval $para; bibentry | wordwrap
    done | block thebibliography 99 | blck flushleft
}
```

It does the sorting and the proper set-up of the environment, as we've discussed for HTML documents. The parameter for the `thebibliography` block is a formatting hint. It is used to specify the right indent and should be as wide or slightly wider then the widest reference marker. Two digits should be enough for most documents.

7.7.2 User defined Variables

There are situations where some text requires special functionality to be set. The logos *TEX* and *LATEX* are the best examples. It needs a sequence of low level operations to achieve such effects. TEX offers macros to handle them conveniently. Macros have names, so we need a way to assign a name to a piece of text. Furthermore we need a fall-back representation, that appears in the OpenOffice document or eventually in other formats we want to create, where the special TEX facilities are not available. That's where user defined variables come in. They represent *name/value* pairs. The value appears as default representation, the name can be used to bind whatever functionality we want to it in TEX. That is not what variables are intended for but it works fairly well:

```
Sub exportTextField(iFile%, oText as Object, sStyle$, sShift$)
    Const Bibliography = "com.sun.star.text.TextField.Bibliography"
    Const UserField = "com.sun.star.text.TextField.User"
    If oText.TextField.supportsService(Bibliography) Then
        ...
    ElseIf oText.TextField.supportsService(UserField) Then
        sName$ = oText.TextField.TextFieldMaster.Name
        sValue$ = quoteStr(oText.TextField.Anchor.String)
        print #iFile sShift & "userfield " & sName & " " & sValue & sStyle
    Else
        ...
    End If
End Sub
```

There are several ways to map the variable name to a TEX macro. The most natural one seems to be shell variables:

```
export tex='{\TeX}'
export latex='{\LaTeX}'
...
```

This definitions might appear anywhere in the style documents. The braces are necessary since we can't be sure in which the context the text might appear.

I'm not aware of an explicit way to test for the existence of a variable, but there is the parameter substitution mechanism of the shell instead. An appropriate expression can be built and then evaluated:

```
function userfield {
        name="$1"; text="$2"; shift; shift
        text '' "${text}" | eval echo -n '$'"{${name}:-'`cat`'}" | text "$@"
}
```

Macro names are TEX meta code. They must not be treated like ordinary text. Specifically it's not a good idea to run them through a character filter replacing special characters, while this should be done for the fall-back text. Text attributes like highlighting, on the other hand, apply to all text regardless of it's source.

7.8 Tables

'Printers charge extra when you ask them to typeset tables, and they do so for good reasons.' That's what Donald E. Knuth says in his TeXbook about tables, and he says so for good reasons. Unlike for HTML documents, the layout matters in printed documents, and there is no simple way to find a solution that pleases the eye and communicates well. It's probably impossible to find an automatism to do that.

Nevertheless it's desirable to support at least simple tables. The basics are already there and all we need to do is to fill in the required functions:

```
function column { cat; test "$1" -lt "$2" && { echo '&'; } }
function row    { cat; test "$1" -lt "$2" && { echo '\\'; } }
function table  { block tabular $3; echo; }
```

Entries in the tabular environment are separated rather then marked-up, consequently there is no separator after the last cell in a row and after the last row in the table. The functions appending the separator need to know which entry is the last one. Therefore they get two additional parameters: the index of the current cell and the maximal index in the row respectively column. For the table function the number of rows and columns might be interesting too:

```
Sub exportTable(iFile%, oTable as Object, sShift0$)
    iRows% = oTable.getRows().getCount()
    iColumns% = oTable.getColumns().getCount()
    sShift1$ = shift(sShift0)
    sShift2$ = shift(sShift1)
    sShift3$ = shift(sShift2)
    print #iFile sShift0 & "( :"
    For i = 0 to iRows-1
        print #iFile sShift1 & "( :"
        For j = 0 to iColumns-1
            print #iFile sShift2 & "( :"
            exportContent(iFile, oTable.getCellByPosition(j,i), sShift3)
            print #iFile sShift2 & ") | column"; j; iColumns-1
        Next
        print #iFile sShift1 & ") | row"; i; iRows-1
    Next
    print #iFile sShift0 & ") | table"; iRows; iColumns
    print #iFile
End Sub
```

This, however, is only part of the story. The LaTeX tabular environment requires an argument describing the column alignments. In OpenOffice such information is bound to paragraphs. Not only that the paragraphs of a column can be formatted differently in different rows, there can be even several differently aligned paragraphs in one cell. To keep things simple we just take the first paragraph of the topmost cell of each column to pick the setting.

```
Sub exportTable(iFile%, oTable as Object, sOShift$)
    ...
    Dim oCell, oPara as Object
    sPara$ = iRows & " " & iColumns & " '"
    For j = 0 to iColumns-1
        oCell = oTable.getCellByPosition(j,0)
        oPara = oCell.getText().createEnumeration().nextElement()
        If oPara.ParaAdjust = com.sun.star.style.ParagraphAdjust.CENTER Then
            sPara = sPara + "c"
        ElseIf oPara.ParaAdjust = com.sun.star.style.ParagraphAdjust.LEFT Then
            sPara = sPara + "l"
        ElseIf oPara.ParaAdjust = com.sun.star.style.ParagraphAdjust.RIGHT Then
            sPara = sPara + "r"
        Else
            sPara = sPara + "p{" + columnWidth(oTable,j,iColumns)/100 + "cm}"
        End If
    Next
    sPara = sPara & "'"
    print #iFile sShift0 & "( :"
    ...
    print #iFile sShift0 & ") | table " & sPara
    print #iFile
End Sub
```

Columns set in block alignment need a column width to be specified. We can set the value in OpenOffice, though getting it out of there is somewhat tricky. It can be retrieved through so called column separators. Column separators appear between the columns of a table. Each table has a collection to provide access to its separators. Each separator has a relative position in the table. The difference between the positions of the two neighbouring separators give the width of the column relative to the width of the table. To calculate the column width we need to distinguish four cases:

1. The table has only one single column. In that case the separator collection is empty and the column width equal to the table width.

2. The table has more then one column and we're dealing with the leftmost column. In that case the relative width of the column is the relative position of the first entry in the separator collection.

3. The table has more then one column and we're dealing with the rightmost column. In that case the relative width of the column is given by the difference of the relative position of the last entry in the separator collection and the reference width of the table, which is available through a property called relative width sum.

4. Otherwise the table must have more than two columns and we're dealing with a column somewhere in between the first and the last one. In that case we calculate the relative width as the distance between the two neighbouring entries in the separator collection.

The column width is then the table width multiplied by the relative column width divided by the relative column width sum:

```
Function columnWidth(oTable as Object, j%, iColumns%) as Integer
    iRelWidthSum% = oTable.TableColumnRelativeSum
    iRelWidth% = iRelWidthSum
    iWidth% = oTable.Width
    If iColumns-1 > 0 Then
        iLower% = LBound(oTable.TableColumnSeparators)
        iUpper% = Ubound(oTable.TableColumnSeparators)
        If j = iColumns-1 Then
            iRelWidth = iRelWidth - oTable.TableColumnSeparators(iUpper).Position
        ElseIf j = 0 Then
            iRelWidth = oTable.TableColumnSeparators(iLower).Position
        Else
            iRelWidth = oTable.TableColumnSeparators(iLower+j-1).Position
            iRelWidth = oTable.TableColumnSeparators(iLower+j).Position - iRelWidth
        End If
    End If
    columnWidth = Int(iWidth*iRelWidth/iRelWidthSum/10+0.5)
End Function
```

The LaTeX tabular environment also allows to draw vertical and horizontal lines between and around table cells. It's not too difficult to add this feature to our script. Vertical lines are specified in the column format argument of the environment and apply to all rows. Like for the alignment, only the borders of the to topmost cells are evaluated:

```
Sub exportTable(iFile%, oTable as Object, sOShift$)
    ...
    For j = 0 to iColumns-1
        oCell = oTable.getCellByPosition(j,0)
        If oCell.LeftBorder.OuterLineWidth > 0 Then
           sPara = sPara + "|"
        End If
        ...
        If oCell.RightBorder.OuterLineWidth > 0 Then
           sPara = sPara + "|"
        End If
    Next
    ...
End Sub
```

Horizontal lines can be specified on a per row base and can span the whole table or only a subset of (consecutive) columns. We determine the first and the last cell that has a border and draw a line between them:

```
Sub exportTable(iFile%, oTable as Object, sOShift$)
    ...
    iMinCol = iColumns
    iMaxCol = 0
    sPara = sPara  & " '"
    For j = 0 to iColumns-1
        oCell = oTable.getCellByPosition(iColumns-1-j,0)
        If oCell.TopBorder.OuterLineWidth > 0 Then
            iMinCol = iColumns-1-j
        End If
        oCell = oTable.getCellByPosition(j,0)
        If  oCell.TopBorder.OuterLineWidth > 0 Then
            iMaxCol = j
        End If
        ...
    Next
    sPara = sPara & "' " & iMinCol+1 & " " & iMaxCol+1
    ...
End Sub
```

That's done once for the topmost cells to be over-lined and then once for each row to be under-lined.

```
Sub exportTable(iFile%, oTable as Object, sOShift$)
    ...
    For i = 0 to iRows-1
        print #iFile sShift1 & "( :"
        iMinCol = iColumns
        iMaxCol = 0
        For j = 0 to iColumns-1
            w = oTable.getCellByPosition(iColumns-1-j,i).BottomBorder.OuterLineWidth
            If w > 0 Then iMinCol = iColumns-1-j
            w = oTable.getCellByPosition(j,i).BottomBorder.OuterLineWidth
            If w > 0 Then iMaxCol = j
            ...
        Next
        print #iFile sShift1 & ") | row"; i; iRows-1; iMinCol+1; iMaxCol+1
    Next
    ...
End Sub
```

The formatting functions need to decide if a line has to be drawn and which command has to be used:

```
function row {
        cat
        if test "$1" -lt "$2" -o $4 -ge $3; then
                echo '\\'
                if test $4 -ge $3; then
                        if test $3 -gt 1 -o $4 -lt $1; then
                                command cline $3-$4
                        else
                                command hline; fi; fi; fi
}

function table {
        (
                if test $5 -ge $4; then
                        if test $4 -gt 1 -o $5 -lt $2; then
                                command cline $4-$5
                        else
                                command hline; fi; fi
                cat
        ) | block tabular $3
        echo
}
```

In case of underlining a row this might require an additional separator to be include at the end of the table, the only case in witch it is allowed and required.

7.9 Frames and Floating Objects

Normally TEX breaks the text into lines and across pages wherever necessary. Occasionally that might not be such a good idea. Some things are better kept apiece, be it on the same line or on the same page. LATEX has two ways to achieve this: boxes and floating environments.

Boxes come in three flavours: LR boxes, parboxes and rule boxes. *LR boxes* line up the text on a single line from left to right, hence the name. The LR box is typically just as high and wide as its content requires, its width, however, can also be specified explicitly. In both cases a frame can be put around the box. A *parbox* breaks its contents into lines, just as in ordinary text. That requires the line width to be known, so a parbox has always a width argument. There is a special environment to create parboxes, the `minipage` environment. It is more flexible than the `parbox` command. *Rule boxes* are rectangular areas filled with ink. If they are rather thin they make up horizontal or vertical rulers, hence the name. Further there are the plain TEX equivalents of boxes, the `hbox`, lining up text horizontally, and the `vbox`, lining up `hbox`es vertically. Common to all boxes is that they are treated like a single character. They become part of the current line and page, no matter how big they are. That might lead to badly spaced lines and partially filled spaces.

Floating environments are a way to avoid such effects by moving a larger chunk of material to a more convenient place, on top of the next page, for example. There are two such environments in standard LATEX, the `figure` environment and the `table` environment. Special document styles might provide more. Essentially all floating environments are similar, they only differ in the way captions are treated.

The OpenOffice equivalent of boxes are frames, though this statement deserves to be treated with some care: In OpenOffice frames are a more general concept then in TEX. While boxes in TEX primarily aggregate text into a single unit, frames in OpenOffice are used to represent and structure all kind of textual and non-textual contents, of which text is just one special case. They are a powerful and flexible tool to control the layout of a document.

We've already met frames when we were talking about images. We have seen that content represented by a frame might be outside the main text flow and that each frame has its own enumeration to access its content. So far, we were only interested in graphical content, now we are primarily interested in frames representing special textual content. Such frames are called text frames:

```
Sub exportFrameContent(iFile%, oText as Object, sFrame$, isPara%, sShift$)
    ...
    Do While ...
        ...
        ElseIf oElem.supportsService("com.sun.star.text.TextFrame") Then
            exportTextFrame(iFile, oElem, sShift)
        ...
    Loop
End Sub
```

Text frames have their own text flow, similar to the main document. Hence, we could just call the same function (recursively) to get their contents exported:

```
Sub exportTextFrame(iFile%, oFrame as Object, sShift$)
    print #iFile sShift + "("
    exportContent(iFile, oFrame, shift(sShift))
    print #iFile sShift + ") | frame " & quoteStr(oFrame.Name)
    print #iFile
End Sub
```

That approach is simple and straight forward, but leaves one problem: How to pick the appropriate LaTeX element to put the contents in? There are several types of boxes in LaTeX, but only one kind of text frame in OpenOffice. There is always the option to use the frame name to provide that information, in fact, it can be used to define an individual function for any single frame in the document. As a general approach, however, that seems rather tedious.

A distinguishing feature that could serve a a selector is the anchor type. We already know that frames can be anchored to a page, to a paragraph, to a character or *as* a character. LR boxes are always treated a single character, even if they are the only object in a line or paragraph. It seems plausible to map them to frames bound as character—or rather vice versa, map frames bound as character to LR boxes. LR boxes appear in the normal text flow. They usually contain only short pieces of text, they are mostly used to avoid undesired hyphenation.

More of a problem is to find a suitable anchor type for minipages. They might contain much more material then LR boxes but are still treated as a single character and hence expected to be part of the text flow. The only other other frames appearing in OpenOffice text flow are frames bound to a character, so that would be the only available choice left. Unfortunately that doesn't really reflect the LaTeX behaviour. Also it is quite a bit of guesswork to get the positioning right.

One way to get around that problem would be to allow frames to be used in a way that will not make them become an object of the target script, but considers them to be there for the sole purpose of structuring the source document. We've already done this with paragraphs: paragraphs containing only a single image should not show up as paragraph, it was only the image we were interested in. As criteria serves the size of the textual representation.

Now we apply the same approach to frames, specifically to frames bound as a character. If such a frame contains any text string, then it is exported as a `framebox`, which will be mapped to an LR box. Otherwise, the contents is exported without being put in a frame. If the frame is bound to character, it's exported as `charframe`. This will become a minipage. Other frames do not appear in the text flow, so we won't come across them here:

```
Sub exportTextFrame(iFile%, oFrame as Object, sShift$)
    DIM asChar as Integer
    asChar = com.sun.star.text.TextContentAnchorType.AS_CHARACTER
    If oFrame.AnchorType = asChar Then
        If len(trim(oFrame.getString())) = 0 Then
            exportContent(iFile, oFrame, sShift )
        Else
            exportFrame(iFile, oFrame, "framebox", sShift)
        End If
    Else
        exportFrame(iFile, oFrame, "charframe", sShift)
    End If
End Sub
```

This way it's possible to put a frame bound to the *end-of-paragraph* character in an otherwise empty frame, which in turn is bound as a character into the text flow. The inner frame will become a minipage while the outer frame will vanish, but still making sure that the whole construct behaves like a single character in the OpenOffice text. We're going to use the same technique to distinguish between text and display mode of mathematical formulas later on.

To distinguish between the various kinds of LR boxes or set box parameters some additional frame properties might be useful. They are passed as environment variables:

```
Function frameBorder(oFrame as Object) as String
    frameBorder = "0"
    iTop% = oFrame.TopBorder.OuterLineWidth
    iLeft% = oFrame.LeftBorder.OuterLineWidth
    iRight% = oFrame.RightBorder.OuterLineWidth
    iBottom% = oFrame.BottomBorder.OuterLineWidth
    If iTop+iBottom * iLeft+iRight > 0 Then
        frameBorder="1"
    End IF
End Function

Function frameWidthType(oFrame as Object) as String
    frameWidthType = "''"
    If oFrame.WidthType = 1 Then
        frameWidthType="fixed"
    ElseIf oFrame.WidthType = 2 Then
        frameWidthType="auto"
    End IF
End Function

Sub exportFrame(iFile%, oFrame as Object, sFrame$, sShift$)
    sFrame = " " & sFrame & " "
    sFrame = " BORDER=" + frameBorder(oFrame) + sFrame
    sFrame = " AUTOHEIGHT=" + oFrame.FrameIsAutomaticHeight + sFrame
    sFrame = " WIDTHTYPE=" + frameWidthType(oFrame) + sFrame
    sFrame = " RELHEIGHT=" + oFrame.FrameHeightPercent*0.01 + sFrame
    sFrame = " RELWIDTH=" + oFrame.FrameWidthPercent*0.01 + sFrame
    sFrame = " FRAMEHEIGHT=" + oFrame.FrameHeightAbsolute*0.001 + sFrame
    sFrame = " FRAMEWIDTH=" + oFrame.FrameWidthAbsolute*0.001 + sFrame
    print #iFile sShift + "("
    exportContent(iFile, oFrame, shift(sShift))
    print #iFile sShift + ") |" & sFrame & quoteStr(oFrame.Name)
    print #iFile
End Sub

function framebox {
        test "`type -t $1`" == 'function' && { eval "$@"; return; }
        if test "$WIDTHTYPE" = 'auto'; then
                if test "$BORDER" = '1'; then bl fbox; else bl mbox; fi
        else
                if test "$RELWIDTH" = '0'; then
                        SIZE="${FRAMEWIDTH}cm"
                else
                        SIZE="${RELWIDTH}\hsize"
                fi
                if test "$BORDER" = '1'; then
                        bl framebox '' "$SIZE"
                else
                        bl makebox '' "$SIZE"
                fi
        fi
}

function charframe {
        test "`type -t $1`" == 'function' && { eval "$@"; return; }
        if test "$RELWIDTH" = '0'; then
                block minipage "${FRAMEWIDTH}cm"; echo
        else
                block minipage "${RELWIDTH}\hsize"; echo
        fi
}
```

Floating objects on the other hand are outside the normal text flow, so it doesn't seem to be an unreasonable choice to bind them to a paragraph instead. Paragraphs have there own content enumeration object to access content bound to it. We can use the same export function as for text portions. However, we need to tell which kind of anchor we are dealing with, so all the functions above get an additional parameter passing the frame type:[4]

```
Sub exportParagraph(iFile%, oPara as Object, sShift$)
    REM handle attached objects first
    exportFrameContent(iFile, oPara, "paraframe", 0, sShift)
    REM then deal with text
    If len(trim(oPara.getString())) > 0 Then
        ...
    Else
        ...
    End If
End Sub
```

The frame type becomes the name of the shell function dealing with the content:

```
function paraframe {
        test "`type -t $1`" == 'function' && { eval "$@"; return; }
        block figure '' htp; echo
}
```

Having gone that far we can take the only remaining step and export the frames bound to a page as well. They are accessible through a document wide collection of all frames, which requires some filtering:

```
Sub exportPageFrames(iFile%, oFrames as Object, sShift$)
    atPage% = com.sun.star.text.TextContentAnchorType.AT_PAGE
    If oFrames.getCount() > 0 Then
        print #iFile sShift & "( :"
        For i = 0 To oFrames.getCount()-1
            Set oFrame = oFrames.getByIndex(i)
```

[4]This applies only to frames bound *to* something, frames bound *as* characters remain to be mapped to a `framebox`.

```
            If oFrame.AnchorType = atPage Then
                exportFrame(iFile, oFrame, "pageframe", shift(sShift))
            End If
        Next
        print #iFile sShift & ") | pageframes"
        print #iFile
        End If
End Sub
```

This is likely to be useful in very special circumstances only, though. There is hardly a way to map pages in OpenOffice to the pages generated by TEX, so it's very difficult to know where such a frame will turn up. You can only be sure if page frames are bound to the first page (or pages) of the document. That's the reason they are exported first:

```
Sub exportDocument(iFile%, oDoc as Object)
    ...
    REM check for page frames
    exportPageFrames(iFile, oDoc.getTextFrames(), "")
    REM export document content
    exportContent(iFile, oDoc, "")
    REM add footnotes at the end of the document
    exportNotes(iFile, oDoc.getFootnotes(), "footnote", "footnotes", "")
    REM add endnotes at the end of the document
    exportNotes(iFile, oDoc.getEndnotes(), "endnote", "endnotes", "")
    REM add bibliography at the end of the document
    exportBibliography(iFile, oDoc.getTextFields(), "")
    ...
End Sub
```

Since this feature is so special there is no default behaviour:

```
function pageframe {
        test "`type -t $1`" == 'function' && { eval "$@"; return; }
        echo "###" unknown pageframe: "$1" 1>&2
}

function pageframes {
        cat;
        echo;
}
```

The probably most useful application of page frames is to deal with document parts that are typically generated by LATEX. Let's say you want to put a table of contents in your OpenOffice document. It might help you while proofreading your work but can obviously not reflect the page numbering in the final document. If you put it in a page frame you can just get rid of it and let LaTeX do the job for you.

7.10 Embedded Objects

Embedded objects are containers for complex content, that comes from an external application. This can be formulas, charts, drawings or spread sheets. Like pictures and text frames they are a special type of frames:

```
Sub exportFrameContent(iFile%, oText as Object, sFrame$, isPara%, sShift$)
    ...
    Do While ...
        ...
        ElseIf oElem.supportsService("com.sun.star.text.TextEmbeddedObject") Then
            exportEmbeddedObject(iFile, oElem.getEmbeddedObject(), isPara, sShift)
        ...
    Loop
End Sub
```

Each type of embedded object gets its own exporting function:

```
Sub exportEmbeddedObject(iFile%, oObj as Object, isPara%, sShift$)
    If oObj.supportsService("com.sun.star.formula.FormulaProperties") Then
        exportFormula(iFile%, oObj.Formula(), isPara, sShift$)
    ElseIf oObj.supportsService("com.sun.star.chart.ChartDocument") Then
        exportChartDocument(iFile, oObj, oObject.Name, sShift)
    ElseIf oObj.supportsService("com.sun.star.sheet.SpreadsheetDocument") Then
        exportSpreadsheetDocument(iFile, oObj, oObject.Name, sShift)
    Else
        unknown(iFile, oObj, "embedded object", "", sShift)
    End If
End Sub
```

Not all of them are implemented yet. The one that are are described in the next chapters.

7.11 Formulas

Even if you can't think of any other reason to use TeX, there is still it's superb ability to set mathematical formulas. Though nowadays any modern word processor comes with some kind of formula editor, non of them I've seen so far comes anywhere near the quality TeX produces. OpenOffice is no exception here. But we're not interested in the type setting qualities of

OpenOffice anyway, all we want is a convenient way to get our document into a computer for further processing.

The OpenOffice formula editor is a standalone program within the OpenOffice suite, so formulas are the first kind of embedded objects we come across. The formula editors bulk functionality is a graphical representation of a formula of which we are only interested for the feedback it gives in the editing process. Our primary interest is the textual representation of the result—a simple string in the end, similar in syntax to the form $\TeX$ is using. Retrieving it is next to trivial:

```
Sub exportFormula(iFile%, oFormula$, lineMode%, sShift$)
    print #iFile sShift & "formula " & lineMode & " " & quoteStr(oFormula)
End Sub
```

Unfortunately the syntax is only similar, not identical to $\TeX$. We can't just take the string an insert it into the output, something I originally was hoping for. Further, the differences are not limited to just tokens and keywords, the extend to the syntax itself. Hence a simple search and replace wouldn't suffice. What we need is a program that is capable of reading a text in one representation and transform it into another, i.e. a compiler.

This is the point where things are getting challenging, but don't worry, it sounds harder than it its. There is a nice little book by Niklas Wirth, that explains the basics of compiler construction on some fifty pages. And we have the added bonus of dealing with open source software: we have the OpenOffice source code which we can use and reuse.

We won't go into all the details of building a suitable compiler, but outline the general procedure. The fully fledged code should have come along with this document.

To illustrate the basics of compiler construction, let's assume we want to write a simple desktop calculator evaluating arithmetic expression. It shall obey to the common precedence rules and allow parenthesis to an arbitrary level. Something like

$$\begin{array}{rcl} 2+3\times 4 & \to & 24 \\ (2+3)\times 4 & \to & 20 \end{array}$$

would be valid expressions.

Writing a compiler starts with finding a grammar. The heart of each grammar is a set of production rules. The rules for the calculator example could be written as:

$$\begin{array}{ccl} \textit{expression} & \to & \textit{term} \mid \textit{expression}\,\texttt{+}\,\textit{term} \mid \textit{expression}\,\texttt{-}\,\textit{term} \\ \textit{term} & \to & \textit{factor} \mid \textit{term}\,\texttt{*}\,\textit{factor} \mid \textit{term}\,\texttt{/}\,\textit{factor} \\ \textit{factor} & \to & \textit{number} \mid \texttt{(}\;\textit{expression}\;\texttt{)} \end{array}$$

The first rule says, that each expression is either a single term, or an expression followed by a `+` operator and a term, or an expression followed by a `-` operator and a term. The rule is recursive. It reduces an expression to either a single term, or a simpler expression and a term, allowing an expression to be an arbitrary long sequence of terms joined by additive operators. It is left-recursive,[5] so expressions are evaluated from left to right.

Similar, a term is either a single factor, or a simpler term followed by a multiplicative operator and a factor. A multiplicative operator can be either `*` or `/`. The fact, that a sequence of factors is reduced to a term before the term becomes part of an expression ensures that multiplicative operators have a higher precedence then additive ones.

Finally, a factor is either a number, or an expression enclosed in parenthesis. There is another recursion in here allowing for an arbitrary level of nesting. We skip the definition of a number for a moment. For now let's assume it's just a single digit:

$$\textit{number} \rightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}$$

The process of building up a valid expression starts with an *starting symbol* appearing on the left hand side of some rule. In our case that would be *expression*. This symbol is replaced by one of the alternatives on the right hand side. The result contains symbols either re-appearing on the left hand side of some rule, or standing for themselves as part of the final expression. The former are called non-terminal symbols, replacing continues with the appropriate rule. The later are called terminal symbols since they terminate the deduction process. Non-terminals are set italic, terminal symbols in fixed font. The remaining symbols are meta symbols to describe the grammar.

Any sequence of terminal symbols that can be deduced from the initial symbol forms a valid expression. The job of the parser is to find the sequence of rules leading to a particular expression.

An efficient parser requires the grammar to obey to certain constrains. Left recursion, for example, is a bad thing and must be avoided. We can do this by transforming the left recursions into iterations:

$$\begin{array}{ccl} \textit{expression} & \rightarrow & \textit{term} \; \{ \; (\texttt{+} \mid \texttt{-}) \; \textit{term} \; \} \\ \textit{term} & \rightarrow & \textit{factor} \; \{ \; (\texttt{*} \mid \texttt{/}) \; \textit{factor} \; \} \\ \textit{factor} & \rightarrow & \textit{number} \mid \texttt{(} \; \textit{expression} \; \texttt{)} \end{array}$$

Braces denote parts that can appear arbitrarily often including not at all, parentheses are there for grouping. There is still a recursion in there, but that

[5]Meaning the left hand side of the rule appears on the very left of a branch of the rule body on the right hand side.

one doesn't hurt. The meaning of the rewritten grammar remains precisely the same, but it is much more suitable to derive a parser.

The parser type we're interested in is called a predictive *recursive descendent parser*. Such a parser is build from a set of mutually recursive functions. Typically each function implements one of the production rules of the grammar:

```
class Input;

double expression(Input&);
double term(Input&);
double factor(Input&);
```

The `Input` class delivers the input to be parsed. It provides two important services to the parser. It keeps a copy of the *current input symbol* ready for inspection and *shifts to the next* symbol when told so. The initial version is fairly simple:

```
class Input {
                char const *ch;
        public:
                int operator()()        { return *ch; }
                Input& next()           { ++ch; return *this; }
                Input(char const *ch):  ch(ch) {}
};
```

The properties of the grammar allow the parser to be implemented very efficiently. The value of the expression is calculated along the way:

```
double expression(Input& input) {
        double value = term(input);
        while ( 1 )
                if ( input() == '+' )
                        value += term(input.next());
                else if ( input() == '-' )
                        value -= term(input.next());
                else break;
        return value;
}

double term(Input& input) {
        double value = factor(input);
        while ( 1 )
                if ( input() == '*' )
                        value *= factor(input.next());
                else if ( input() == '/' )
```

```
                        value /= factor(input.next());
                else break;
        return value;
}

double factor(Input& input) {
        double value;
        if ( isdigit(input()) )
                value = input()-'0', input.next();
        else if ( input() == '(' ) {
                value = expression(input.next());
                if ( input() == ')' )
                        input.next();
                else
                        throw "')' expected";
        }
        else
                throw "unexpectd character";
        return value;
}
```

We still need a `main` function to initialize the input, run the parser, catch any error conditions and print the results. Here is one:

```
int main(int argc, char *argv[]) {
        try {
                while ( *++argv ) {
                        Input input(*argv);
                        cout << *argv << '=' << expression(input) << endl;
                }
        }
        catch ( const char* exception ) {
                cout << "Error: " << exception << endl;
                return 1;
        }
        return 0;
}
```

Putting this together we can run the parser to evaluate expressions. Assuming we have compiled it all into a binary called `expr` we'd get:

```
$ expr '2*(3+3)' '2*3+3'
2*(3+3)=12
2*3+3=9
```

So far we have assumed, that each number is just a single digit and that each terminal symbol matches exactly one character in the input stream, and vice versa. That is not a realistic assumption. Real languages have symbols consisting of more than one character, take key words for example, or certain operators like <=, == and >=. In our case it's a number that can be longer than just a single digit. On the other hand there might be characters in the input stream, which are convenient for humans but not really part of the input. Our parser doesn't tolerate spaces or line breaks for example yet. Such character should be allowed, but ignored.

All this is the job of the scanner. The scanner reads the input, assembles character sequences into symbols and removes any unwanted wide spaces. The scanner is a finite state machine, though often that is not immediately apparent from its structure. A realistic scanner for our example parser is still fairly simple:

```
class Input {
                char const *ch;
                int current;
                double num;
        public:
                enum { Number = 256 };
                int operator()()        { return current; }
                double number()         { return num; }
                Input& next();
                Input(char const *ch):  ch(ch) { next(); }
};

Input& Input::next() {
        while ( isspace(*ch) )
                ++ch;
        if ( isdigit(*ch) ) {
                num = *ch++ - '0';
                while ( isdigit(*ch) )
                        num = num * 10 + (*ch++ - '0');
                current = Number;
        }
        else switch ( *ch ) {
                case '+':   current = *ch++; break;
                case '-':   current = *ch++; break;
                case '*':   current = *ch++; break;
                case '/':   current = *ch++; break;
                case '(':   current = *ch++; break;
                case ')':   current = *ch++; break;
                case '\0':  current = *ch; break;
                default:    throw "invalid character";
        }
        return *this;
}
```

Note that a number must not contain any sign, that would cause ambiguities. In order to allow for signed numbers we need to extend the grammar

$$\textit{factor} \quad \rightarrow \quad \texttt{NUMBER} \mid \texttt{-}\ \textit{expression} \mid \texttt{(}\ \textit{expression}\ \texttt{)}$$

and adopt the parser accordingly:

```
double factor(Input& input)
{
        double value;
        if ( input() == Input::Number )
                value = input.number(), input.next();
        else if ( input() == '-' )
                value = - expression(input.next());
        else if ( input() == '(' ) {
                value = expression(input.next());
                if ( input() == ')' )
                        input.next();
                else
                        throw "')' expected";
        }
        else
                throw "factor expected";
        return value;
}
```

So, what's got all this to do with formulas? As mentioned above, OpenOffice has its own language to represent mathematical formulas. This language is implemented in a textbook-like, hand-coded recursive descendent parser written in C++. It follows the same principles as our expression parser, only that it will be considerably larger. The parser consists of some 30 functions and the scanner distinguishes some 220 symbols, including about 200 keywords. And there's nothing for us to calculate along the way. Instead we're interested in the structure of a formula to print it in the TEX syntax. To illustrate this let's do the same with our expressions.

The structure of an expression is stored in a tree, which is build up along the parsing process.[6] Each node in the tree represents a syntactical entity in the parsed sentence. Consequently, the class hierarchy of the objects forming the tree is closely related—though not identical—to the grammar.

The root of the class hierarchy describes what we want to do with each formula. We just want to print it:

[6]Presumably we could print the result parallel to parsing, like we did the calculation above, but having the complete syntax tree is more flexible.

```
struct Formula {
        virtual std::ostream& print(std::ostream&) const = 0;
        virtual ~Formula() {}
};

typedef std::ostream ostream;
typedef std::auto_ptr<Formula> fp;

ostream& operator<<(ostream& os, Formula const& f) { return f.print(os); }
ostream& operator<<(ostream& os, fp const& f) { return f->print(os); }
```

We've got four sub-elements of a formula:

```
class BinaryOp: public Formula {
                const char* op;
                fp lhs, rhs;
        public:
                ostream& print(ostream& os) const {
                        return os << '{' << lhs << '}' << op << '{' << rhs << '}'; }
                BinaryOp(fp lhs, const char* op, fp rhs) :
                        op(op), lhs(lhs), rhs(rhs) {}
};

class UnaryOp: public Formula {
                const char* op;
                fp expr;
        public:
                ostream& print(ostream& os) const {
                        return os << op << '{' << expr << '}'; }
                UnaryOp(const char* op, fp expr) :
                        op(op), expr(expr) {}
};

class Number: public Formula {
                double value;
        public:
                ostream& print(ostream& os) const { return os << value; }
                Number(double value) : value(value) {}
};

class Parentheses: public Formula {
                fp expr;
        public:
                ostream& print(ostream& os) const {
                        return os << "(" << expr << ")"; }
                Parentheses(fp expr) :
                        expr(expr) {}
};
```

The parser functions have to be modified accordingly to build up and return the syntax tree:

```
fp expression(Input& input) {
        fp expr = term(input);
        while ( 1 )
                if ( input() == '+' )
                        expr = fp(new BinaryOp(expr, "+" , term(input.next())));
                else if ( input() == '-' )
                        expr = fp(new BinaryOp(expr, "-" , term(input.next())));
                else break;
        return expr;
}

fp term(Input& input) {
        ...
}

fp factor(Input& input) {
        fp expr;
        if ( input() == Input::Number )
                expr = fp(new Number(input.number())), input.next();
        else if ( input() == '-' )
                expr = fp(new UnaryOp("-", expression(input.next())));
        else if ( input() == '(' ) {
                expr = fp(new Parentheses(expression(input.next())));
                if ( input() == ')' )
                        input.next();
                else throw "')' expected"; }
        else throw "factor expected";
        return expr;
}
```

The use of auto pointers guaranties the proper clean-up. The result of the parsing process is a syntax tree that can be printed:

```
        cout << *expression(input) << endl;
```

Assuming we have substituted that line in the `main` function above we'd get the following result:

```
$ expr '2+3*-4' '-(2+3)*4'
{2}+{{3}\times{-{4}}
-{{({2}+{3})}\times{4}}
```

This might be slightly more braces then necessary, but better save then sorry.

To get OpenOffice formulas printed in TEX syntax we modify the OpenOffice formula parser in exactly the same way. The parser is already there, thanks to open source software. It still contains all the functionality required for the OpenOffice-internal stuff. We need to get rid of that first. To illustrate the procedure consider the following function:

```
void SmParser::Table()
{
        SmNodeArray  LineArray;

        Line();
        while (CurToken.eType == TNEWLINE)
        {
                NextToken();
                Line();
        }

        if (CurToken.eType != TEND)
                Error(PE_UNEXPECTED_CHAR);

        ULONG n = NodeStack.Count();
        LineArray.SetSize(n);

        for (ULONG i = 0; i < n; i++)
                LineArray.Put(n - (i + 1), NodeStack.Pop());

        SmStructureNode *pSNode = new SmTableNode(CurToken);
        pSNode->SetSubNodes(LineArray);
        NodeStack.Push(pSNode);
}
```

It has been copied from the file `starmath/source/parse.cxx` in the OpenOffice source tree and implements the production rule for a *Table*, which happens to be the starting symbol of the formula grammar. Anything not related to parsing can be removed, of the function cited above would remain only:

```
void ooMath::Parser::Table(Input& input)
{
        Line( input );
        while ( input() == NEWLINE )
                Line( input.next() );
        if ( input() != END )
                throw "Table: unexpected character";
}
```

We apply some modifications to the naming scheme and use our own name spaces. We further adopt the input handling and the error reporting mechanism. The structure of the parser and the syntax it accepts, however, remains untouched.

This has to be done for the remaining 28 functions of the grammar as well. Some of them are rather complex, while the majority falls into the category of the example above.

The error reporting mechanism is already familiar and rather simple. Whenever the parser encounters a syntax error it just throws an exception. Whenever a function terminates normally the input could be parsed correctly.

The scanner is going to have significantly more to do then we're used to so far. Not just that it has to deal with new classes of tokens like identifiers and keywords, also a token is no longer presented by just an integer. It carries additional information to make the parse process easier:

```
struct Token {
        TokenType   type;
        TokenGroup  group;
        TokenLevel  level;
        const char* desc;
};
```

The `TokenType` is a symbolic representation of a token. It is an enumeration and uniquely identifies each token or token class. The `TokenGroup` puts tokens into groups, which simplifies certain decisions in the parser. Technically it is an enumeration to, though it implements a bitset semantic. The `TokenLevel` is a number used to decide about operator precedences. Finally there is a human readable description primarily intended for debugging.

In the source code the definition of `Token` is actually a class, providing a set of helper functions to make implementing the parser more convenient.

The scanner itself lives in its own `Scanner` class, providing the core functionality including the keyword lookup. The parsers `Input` classes is reduced to a simple wrapper to get the interface the parser expects.

The modified parser gives us a simple acceptor, i.e. an executable that's able to parse an input string without doing anything with it except telling us, if it's syntactically correct or not. It's called from within the `main` function we already know:

```
        Scanner scanner(*argv);
        Parser()(scanner);
```

This creates a temporary Parser object. The parse process is started by calling the function call operator with an initialized scanner. The function call operator merely initializes the Input class and calls the function standing for the grammar's starting symbol:

```
void ooMath::Parser::operator()(Scanner& scanner) {
        Input input(scanner);
        Table(input);
}
```

Now it is high time to define some test cases. Ultimately we'll have to do this in an OpenOffice documents, like for all other OpenOffice features, but for now a simple shell script will do:

```
#!/bin/bash

function test
{
        while read line; do
                echo "### $line"
                math2tex "$line" || { echo 'ERROR'; exit 1; }
        done
}

### Examples
test << $$$
{df(x)} over {dx} = ln(x) + tan^{-1}(x^2)
...
$$$
```

It defines a number of test cases and parses them line by line. The parser binary has been named math2tex. The test terminates with a message if an error occurs. If the script passes without reporting an error we can at least be sure, that no syntax errors have been detected. I compiled test cases from the OpenOffice documentation, the TEXbook, and my own documents. The test set is certainly not comprehensive, but covers the most common cases.

Almost there. What remains to do after having the plain parser is to fill in the semantics. It starts with defining the root of a class hierarchy. Like anything else it goes into our dedicated name space:

```
namespace ooMath {
        class Formula;
        typedef std::auto_ptr<Formula> formula;
};
```

```
class ooMath::Formula {
        public:
                class BinaryOp;
                ...
                static const char* newline();
                ...
                virtual std::ostream& print(std::ostream&) const = 0;
                virtual ~Formula() {}
};
```

I prefer to put all subclasses into the root class, that helps keeping the name space tidy. The root class also defines functions to provide the keyword encodings, like:

```
const char* ooMath::Formula::newline()  { return "\\atop"; }
```

One node type we meet again is that of a binary operator, which looks nearly exactly as the one we already know:

```
class ooMath::Formula::BinaryOp: public ooMath::Formula {
        private:
                const char* op;
                formula lhs, rhs;
        public:
                virtual std::ostream& print(std::ostream& os) const;
                BinaryOperator(const char* op, formula& lhs, formula& rhs)
                        : op(op), lhs(lhs), rhs(rhs) {}
};

std::ostream& ooMath::Formula::BinaryOp::print(std::ostream& os) const {
        return os << '{' << *lhs << '}' << op << '{' << *rhs << '}'; }
```

Getting it into the parser functions is fairly straight forward, at least potentially. We've already demonstrated the general idea:

```
ooMath::formula ooMath::Parser::Table(Input& input)
{
        formula lhs = Line( input );
        while ( input() == NEWLINE ) {
                const char* op = Formula::newline();
                formula rhs = Line( input.next() );
                lhs = formula(new Formula::BinaryOp(op, lhs, rhs));
        }
        if ( input() != END )
                throw "Table: unexpected character";
        return lhs;
}
```

Sometimes, however, it might be necessary to rewrite the parser functions to exactly reflect the meaning of an expression. A sequence of lines, for example, is better represented by a list:

```
ooMath::formula ooMath::Parser::Table(Input& input)
{
        formula lhs = Line( input );
        if ( input() == NEWLINE ) {
                lhs = formula(new Formula::Head(lhs));
                do {
                        const char* op = Formula::newline();
                        formula rhs = Line( input.next() );
                        lhs = formula(new Formula::Tail(op, lhs, rhs));
                } while ( input() == NEWLINE );
        }
        if ( input() != END )
                throw "Table: unexpected character";
        return lhs;
}
```

The code has been changed to reflect the fact, that a line can either stand for itself, or become an element of a list. A line in a list needs to be treated differently then a line standing alone, for example list items are likely to have to be enclosed in braces. A list head, in turn, needs to be treated differently then the elements in the tail. Sometimes this changes are purely cosmetic, sometimes they are required to reflect the real semantic. We skip the details of the `Head` and `Tail` class here. They are here mainly there to demonstrate the idea, the real code looks slightly different.

There is a fair number of such corrections necessary to get the correct semantic. Going through all the details of the formula compiler would be beyond the scope of this document, not least because it's still evolving. So we'll leave it there. The point was to outline the general procedure of deriving our special purpose parser from the published source code, which can be summarized as follows:

1. Extract the parser code and remove all semantic-related parts, so that only a plain acceptor remains.

2. Put it into a C++ `main` function, compile it an run it on some test cases. It should provide an accept/non-accept output on each test string.

3. Create a C++ class hierarchy reflecting the formula syntax's structure. Some of the classes will correspond to a parser reduction rule, others

won't. The constructor of each class takes pointers of the nodes corresponding to the sub-symbols of the rule and stores them in the newly created instance. The use of `auto_ptr` spares the trouble of worrying about memory management.

4. Modify the parser so that each rule creates and returns an instance of the corresponding syntax tree class, holding references to all sub-symbols. The result of a successfully parsed string will be am object tree corresponding to the syntax tree of the pares string. If an error occurs an exception is thrown and no tree is return.

5. Add a virtual `print` function to the class hierarchy, that takes an output stream and prints the representation of each class in correct TeX syntax to it. Modify the `main` function, so that each returned top-level object is printed to `cout`.

There is one more interesting detail: The OpenOffice formula editor allows user defined special symbols. Since they can change any time it's not such a good idea to rely on them to be hard coded into the compiler. Instead we allow them to be read in from one or more files. The file structure is very simple. It's just a key/value pair per line, where the key represents the special symbol and the value its corresponding TeX macro. The formula compiler performs a simple linear search through all files in the order they are specified. There are certainly more advanced techniques but they don't seem to pay of. We keep the built-in lookup table as a fall-back, though. It is searched whenever a special symbol can't be found in a file, including the case that no lookup files are specified. If a symbol can't be found there either it is a syntax error.

The fully fledge source code should have come along with this document.

Chapter 8

Managing Projects

In this chapter we discuss some of the issues that might become relevant once your documentation efforts grow into projects and are expected to live over a longer time.

8.1 Scripts versus Software

There is a difference between scripts and software. Software always has a documentation an a life cycle, scripts usually don't. Scripts are often written, used, and then thrown away. They don't need a documentation, since they're short and simple enough for the code to document itself. They don't need version control, they simply don't live long enough to bother.

Scripting languages strive for rapid development and simplicity. There is no safety net like a type system that checks for consistency or certain conditions. Usually that doesn't matter. The programmer is identical to user, he or she understands the code best and is capable to detect and correct errors without further ado. If that doesn't apply to your scripts then chances are that scripting is not the best solution to your problem.[1] Actually I'm not sure that it is such a clever idea to implement large projects in—let's say—Perl.

There's no rule without exception, though. Sometimes you're bound to use a certain scripting language for some rare feature it offers. In case of our Unix shell scripts it's the pipe that is the crucial feature. It allows a kind of parallelism found in very few other languages.

[1]I've seen a script working for ages as a batch job, until someone tried to run it manually, incidentally from a rather unusual directory. Suddenly an unquoted wild card matched a file name and led to really unexpected results.

While the simplicity and rapidity of scripting was helpful to get the project started, it becomes more and more of a problem now that the technique is maturing. It's true, the scripts are still fairly simple and (mostly) well structured, so they fairly well document them self. But they depend to a high degree on the Unix environment in place, like the behaviour of the standard tools. Any change here is likely to affect the output you'll get.

8.2 Version Control

Upgrades of your Unix system, migrating to a different distribution or the evolution of your scripts—of both, shell scripts and OpenOffice macros—all this is likely to effect the output you'll produce. That can backfire one day. Usually you'll want to keep an electronic version of your documents, and expect it to be still usable even after years. In a way the scripts become apart of the document, so you need to make sure they remain stable.

One approach to achieve this is to archive scripts along with your documents. That ensures at least, that any modifications you make to add new feature will not affect existing documents. It works fine for shell scripts, but causes problems for OpenOffice macros. OpenOffice comes with its own macro management. I'm not aware on any means to call a script from outside the macro manager, so all the versions of all the scripts need to be kept in one predefined place. That is at least inconvenient.

A more promising approach seems to be to rely on one set of scripts only and put your target documents under version control. The idea is not so much to record the document history—though you might even be consider this as an added bonus—but to detect changes. Whenever you restart working on a documentation you haven't been working on for a while, just recreate all your target documents and let the version control software tell you what has changed. That prevents any impact of possible interim changes to remain undetected and allows you to act accordingly. This approach will also cover changes on the system's side, not just modifications you made intentionally. It has the advantage that all new features are available in all projects and spares you the trouble to keep the development in different projects in sync.

It shouldn't really matter which version control system you're using, as long as it can detect changes, which is fairly standard. Use the one of your choice. But use one! Ignoring this hint is likely to cause you a lot of pain one day.

8.3 Build tools

Rebuilding a set of documents frequently will soon call for some kind of build tool. I'm using `make`, but fell free to use whatever tool suits you.

Writing a makefile is straight forward and not different from any other makefile. The scripts generated by OpenOffice can be called like any other executable program, as long as it is found in the search path. Beware, however, that make doesn't expect scripts to be generated. It has some funny built-in rules that can be really counter-productive. Best get rid of all built-in rules first:

```
%: ;
```

To allow OpenOffice documents to be exported from within a build tool you need a procedure that can be called in batch mode:

```
Sub Batch(sDocName as String)
    Dim sURL as String
    Dim oDoc as Object
    sURL = ConvertToURL(sDocName)
    oDoc = StarDesktop.LoadComponentFromURL(sURL, "_blank", 0, Array())
    exportToFile(fileName(sDocName, "sh"), oDoc)
    oDoc.close(true)
End Sub
```

It gets the name of the document to be exported as an argument. There is no need to run a dialogue. The function `filename` derives the output file name from the document name. It's exactly the same code we've already seen in the `main` procedure, put I it's own function:

```
Function fileName(sDocName as String, sExtension as String) as String
    Dim vPath as Variant
    Dim vName as Variant
    vPath = split(sDocName, "/")
    vName = split(vPath(UBound(vPath())), ".")
    If LBound(vName) < UBound(vName) Then
        vName(UBound(vName)) = sExtension
    Else
        vName(LBound(vName)) = vName(LBound(vName)) & sExtension
    End If
    vPath(UBound(vPath())) = join(vName, ".")
    fileName = join(vPath, "/")
End Function
```

The `exportToFile` is new as well. It opens the output file and starts the export:

```
Sub exportToFile(sFileName as String, oDoc as Object)
    Dim iFile as Integer
    iFileNumber = FreeFile
    Open sFileName for Output as #iFileNumber
    exportDocument(iFileNumber, oDoc)
    Close #iFileNumber
End Sub
```

The two new functions factorize functionality from the main procedure as well, so that one is getting a bit simpler:

```
Sub Main
    Dim sDocName, sFileName as String
    Dim oDialog as Object
    DialogLibraries.LoadLibrary("DocScript")
    sDocName = convertFromURL(ThisComponent.URL)
    if len(sDocName) > 0 then
        sFileName = fileName(sDocName, "sh")
    end if
    oDialog = createUnoDialog(DialogLibraries.DocScript.FileOpen)
    oDialog.getControl("FileName").text = sFileName
    If oDialog.execute() = 1 Then
        exportToFile(oDialog.getControl("FileName").text, ThisComponent)
    End If
End Sub
```

Unfortunately OpenOffice batch procedure expects the full path name as argument. I haven't found a way around that yet, but there's a simple workaround on shell level:

```
$(OPENOFFICE) "macro:///HTML.export.batch(`pwd`/$<)"
```

This is a line from the makefile. The environment variable `$(OPENOFFICE)` is there to coop with different names of the OpenOffice binary in different distributions. It should be set in the `.profile`. `HTML.export.batch` is the fully qualified procedure name, consisting of the library name, the name of the module in the library and finally the name of the procedure in the module.

If you plan to use several different environments to work on you documents it is a good idea to specify a language locale in the makefile. That ensures the same output in every environment and avoids a lot of unwanted change reports from your version control system each time you change to an environment with different default setting.

There is one hitch to watch out for, though. It seems OpenOffice is designed to run one instance only per user at a time. If the build tool starts a job to re-export a document while OpenOffice is already running, then the batch job will try to connect to the existing instance, implying that it will use the environment settings which were in place when the first instance has been started. These settings are likely to be the default settings of the system, not the one specified in the makefile. If they differ the simplest option you have is to close all OpenOffice windows before running the build tool. If that's not feasible because it's to inconvenient to re-open OpenOffice each time, you can start the first OpenOffice instance from a shell with the environment set correctly.

There is an even more severe problem on Windows. It seems, in the Windows version of OpenOffice an external macro call returns before the macro actually terminates. A build tool will assume the output file to be created completely—what's not the case yet—and start processing it—what's definitely bound to cause trouble. That makes the use of build tools on windows next to impossible.

8.4 Environment Settings

In general it's not a problem to export an OpenOffice document on one system and run the resulting script on another. Potentially that works even between Windows and Unix. If you move scripts between systems with different default character encoding schemes, however, you're likely to run into problems. Your target system will not know it has use a different encoding scheme.

The simplest way around this problem is to tell the target system the scheme that was in place when the script has been generated. It's just a matter of setting the correct environment variable. To do the job properly we add some more figures that might be of interest one day, like the OpenOffice version used to create the script, or the operating system it was running on:

```
Sub exportDocument(iFile%, oDoc as Object)
    print #iFile
    print #iFile "export OPENOFFICE_SOLAR_VERSION=" + GetSolarVersion()
    print #iFile "export OPENOFFICE_VERSION=" + ooVersion()
    print #iFile "export OPENOFFICE_GUI=" + GetGUIType()
    print #iFile "export LANG=" + Environ("LANG")
    ...
End Sub
```

If you want to move scripts from Windows to Unix you have to take care of the different line-end schemes as well. The `dos2unix` tool is your friend. It's easier, however, to transfer the OpenOffice documents and run the export on the Unix side.

8.5 A Status Bar

One of our first steps towards exporting OpenOffice documents has been to create a menu entry and a tool bar button to start the export. These are still there and the most convenient way to get a script while documents are open for editing. The further processing is then started from a shell.

For longer documents the export can take quite a while, so you'll have to wait some time until it has completed. The problem is: for how long? How do you know when it is save to continue?

To monitor the progress we can implement a status bar:

```
Sub exportShowProgress(iFileNumber%, sFileName$, oDoc as Object)
    On Error GoTo OnError
    Dim oBar as Object
    oBar = oDoc.getCurrentController().getFrame().createStatusIndicator()
    oBar.start("Exporting script to "+sFileName, oDoc.ParagraphCount)
    exportDocument(iFileNumber, oBar, oDoc)
    oBar.end()
    Exit Sub
OnError:
    MsgBox "An Error occurred in Line: " & Erl & CHR(13) & Error$, 16, "Error"
    oBar.end()
End Sub
```

This creates a status bar object in the OpenOffice status line and sets its size to the number of paragraphs in the document. The actual counting is done while iterating through the documents main text flow:

```
Sub exportContent(iFile%, oContent as Object, sShift$, optional oBar as Object)
    ...
    iPara% = 0
    Do While oParaEnum.hasMoreElements()
        ...
        If not IsMissing(oBar) then
            iPara = iPara + 1
            oBar.setValue(iPara)
        End If
    Loop
    ...
End Sub
```

The `exportContent` function can be used to export any text flow, but only while exporting the main flow it gets access to a status bar object. Hence the parameter is optional and we have to check for it's existence before operating on it.

The error handling above is necessary to remove of the progress bar in case of an error. Otherwise the object remains in the status line even when the macro has terminated. Closing the OpenOffice window would be the only way to get rid of it.

8.6 Backups

You spent a lot of time and effort to create your documents and you want to protect them from data loss, right? Unfortunately, managing backups is an ungrateful job. It's a bit like buying an insurance: you pay for something you'll hopefully never need. Thereby it's not so much the money you pay for the equipment that matters, it's the time it costs you. If you work on a server that's in a data centre, or on a file system shared from there, then you're in luck. The operator will take care of backups. If, however, you're working on your own desktop PC, you'll have to do the job yourself. You don't have an operator, who takes care of replacing the tapes, there is no IT-manager checking the backup statistics and pushing you in case there's something wrong, and there is no time slot in the middle of the night when you can run a regularly scheduled backup that wouldn't bother anyone. You need to run your backups—manually, regularly and during your working time. That's annoying. To avoid things that are annoying is only human, and since there's little use in a backup that's not run regularly you're likely to end up soon with no backup at all.

The key to solve the dilemma is to be as selective as possible regarding the data to be backed up. I never saw a point in backing up the system disk. If it crashes you can reinstall it from CD-ROM. It might take a bit longer then a restore, but better to invest the time once when necessary then often just in case. A good installation log it much better here then a backup. It not just helps in case of data loss but also when upgrading to the next OS version.

No, the really important stuff is your user data, data you created on your own, that is unique and changes rapidly. That needs to be protected and it needs to be protected from more then disk failures. In fact, a disk failure is fairly unlikely nowadays. Modern disks are very reliable and often replaced long before they reach the end of their physical live, simply because they become outdated by new developments. But how often had you incidentally deleted a file you then wanted to have back? Or how often had you made some changes you wanted to revert before the previous state had been save in your version control system? Or may be you had to suffer data loss after a software crash? I any of these sounds familiar to you then you'll agree that backups on user data need to be run *very* frequently.

For this reason I decided to run a backup after each successful build. That applies to software development in the same way as for documentation projects. This backup spared me a lot of headache and I would call it the most successful backup concept I've ever implemented. Here is a little script that does the job:

```
BACKUPNAME=${projectname}
BACKUPDIR=/backup/$USER/$BACKUPNAME
BACKUPLABEL=$BACKUPDIR/.backup
BACKUPLIST=$BACKUPDIR/.newfiles
BACKUPEXCLUDES='-e *.o -e *.aux -e *.dvi -e *.log -e *.toc'

function backup
{
        BACKUPSESSION=`date +%Y%m%d.%H%M%S`
        if test -f ${BACKUPLABEL}; then
                find "$1" -xdev -depth -newer ${BACKUPLABEL}
                BACKUPSESSION=${BACKUPSESSION}.diff
        else
                find "$1" -xdev -depth
                BACKUPSESSION=${BACKUPSESSION}.full
                echo '### no backup timestamp found, full backup forced' 1>&2
        fi | grep -v ${BACKUPEXCLUDES} >${BACKUPLIST}
        echo ${BACKUPSESSION} >${BACKUPLABEL}
        if test -s ${BACKUPLIST}; then
                BACKUPFILE=${BACKUPDIR}/${BACKUPNAME}.cpio.${BACKUPSESSION}.gz
                cat ${BACKUPLIST} | cpio -o -a | gzip >${BACKUPFILE}
                echo '### backup:' `ls -sh ${BACKUPFILE}`
        else
                echo '### no backup neccessary' 1>&2
        fi
}

backup "${1:-.}"
```

The environment variable are set in the makefile, the rest goes into a script. The script is called from the makefile every time the project reaches a somewhat consistent state. Consistency is assumed if a binary could be compiled correctly or—in case of printable documents —the TEX run completed successfully. Originally it was a simple `tar`. Now it is a bit smarter.

It expects a backup directory to be defined, one for each project. There it searches for a time stamp indicating the last time a backup has been run. If it finds one the time stamp is used to find all files that have changed since then, otherwise all files of the project tree are included. Then some files matching certain patterns are excluded, essentially all files created automatically. The remaining files are packed into a `cpio` archive, compressed and stored under a name containing the backup time.

The script is very selective in the data it picks, so it runs very fast and produces only small backup files. It doesn't hurt at all to call it after each successful build, in normal situations you won't even notice. If your edit–compile–test cyc1e is very short it can run as frequently as every few minutes or even more often. It backs up only a delta, so the more often it is called the smaller the backup size becomes. On the other hand it inc1ude temporary files and files not yet under version control. If you want a full backup, just delete the time stamp file. I have a special make target to do so.

On my systems I put the backup directories of all projects onto a dedicated disk partition of about 700 MB, i.e. the size of a standard recordable CD-ROM. Whenever the disk utilisation on the backup file system is getting close to 100% I burn the whole file system onto CD-ROM, delete the older half of the backup files and continue to work. So I have two copies of each backup file on CR-ROM.

If your backup partition resides on a different disk it will also protect you from disk failures. It doesn't need to be a disk to achieve that, an USB-stick, a SIM-card or even a floppy drive would do equally well. A very interesting idea is to use a virtual disk drive from your ISP. The data there will be save even if a power peak completely destroys your hardware or your flat burns down[2]. You'll want, however, encrypt your data before you sent it somewhere outside your house with no access control whatsoever.

One last thing: don't be tempted to use disk mirrors to protect your data. A mirror never replaces a backup! It more likely lulls you into a false sense of security, especially if you don't have a proper monitoring. A mirror can protect you from a disk failure but that chance is rather slim anyway. We've discussed it above. Human errors are on top of the list of reasons for data loss, followed by software failures. In neither case a mirror does any good. What a mirror does buy you is *availability.* You can continue to work even if on of your disks fails, which will—according to Murphy's Law—always happen on Friday night just after your favourite computer store was closed for the weekend, with your theses due on Monday morning. So if you often have

[2]Though in that case you'll probably have other worries than your data.

tough deadlines, which are likely to ruin you if not met, then a mirror is the right choice for you; but not as a replacement for backups, as an addition. Be sure, however, that you have a proper monitoring. Mirror software is there to hide a hardware failure from the user, and modern systems do this extremely well. Often you won't even notice a performance drop. That's good for your work but also implies that you won't notice there's something wrong until your second disk fails as well; when everything is too late. You need a piece of software that monitors the `syslog`-file and notifies you in time if there's anything unusual, so you can act accordingly. If you use a RAID-controller you need to be sure that failures are communicated from the controller to the driver and then logged somewhere you can monitor it, which might or might not be the `syslog`. You see there's much more bear in mind when operating a mirror then setting up the file systems. I tried it for a while and it can be fun, but apart from that I found it's hardly worth the effort.

Bibliography

[1] Andrew D. Pitonyak. *OpenOffice.org Macros Explained.* 3rd. edition, 2016. `http://www.pitonyak.org/book`.

[2] The UNIX Programming Environment. *Brian W. Kernighan and Rob Pike.* Prentice Hall, Englewood Cliffs, New Jersey, 1984.

[3] The AWK Programming Languag. *Alfred V. Aho and Brian W. Kernighan and Peter J. Weinberger.* Addidon-Wesley, 1988.

[4] The C Programming Languag. *Brian W. Kernighan and Dennis M. Ritchie.* Addidon-Wesley, 1988.

[5] The TEXbook. *Donald E. Knuth.* Addidon-Wesley, 1986.

[6] LATEX: A document Preparation System. *Leslie Lamport.* Addidon-Wesley, 1986.

[7] N. Wirth. *Compilerbau.* Teubner, Stuttgart, 1981.

[8] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design.* Addidon-Wesley, 1977.

[9] Bjarne Sroustrup. *The C++ Programming Language.* Addidon-Wesley, 3rd. edition, 2000.