nw2md: A Markdown Literate Programming Tool

Meik Teßmer*

2014

Dept. *Computergestützte Methoden* (CoMet), Faculty of Business Administration and Economics, Bielefeld University

Version: 1

TODO

Contents

1 noweb Made Easy 1
2 The Conversion Tool 3
3 The tang1e tool 9
   3.1 Common Code Snippets 11
4 Todo 12
5 Code Chunks 13

1 noweb Made Easy

noweb is an ideal tool for Literate Programming. It has a clean and simple syntax and generates good looking documents (HTML and TeX). However, it forces the author to write documentation in LaTeX. Usually using LaTeX is not very difficult, but for beginners the learning curve

*mtessmer@wiwi.uni-bielefeld.de*
can be a bit steep. To simplify the first steps in Literate Programming a less complicated markup language can be helpful.

Markdown is known as a simple, easy to read and write text formatting syntax, and is supported by many converters, e.g. pandoc. In combination with noweb’s syntax pandoc adds up to a lightweight Literate Programming tool:

- easy writing by using Markdown instead of LaTeX
- adequate syntax highlighting
- output to nearly every thinkable digital format

To use Markdown with noweb we need a conversion tool that translates noweb code chunks

```
<<my code chunk>>=
def my_function():
    pass

@ This function ...

to pandoc’s fenced code blocks syntax:

~~~
def my_function():
    pass

~~~
This function ...
```

Other than the standard Markdown markup for verbatim blocks (indentation by four spaces or one tab) the fenced code blocks can use attributes to specify how the code should be displayed. This is especially useful for syntax highlighting and identifier definitions.

Noweb itself does not support additional information along code chunk definitions. Therefore I propose a small syntax enhancement to allow pandoc to use a sensible syntax highlighting. Written as a regular expression a standard noweb code chunk starts with

```
^<<([^>]+)>$```

This definition will be extended by a pair of braces around the name of the used programming language in the block, e.g.

```
<<start reading file>>= (bash)
```

The according regular expression is

```
^<<([^>]+)>\s*\((\w+)\)$```

With this extension the conversion tool can define the language attribute of a fenced code block and enable syntax highlighting.
2 The Conversion Tool

nw2md converts a file with Markdown syntax and (enhanced) noweb code chunks. It

- preserves compatibility with noweb code chunk definitions
- adds a language attribute for better syntax highlighting if available
- generates a code chunk index at the end. By default it creates a section, with -s it will
generate a subsection.

Currently documentation chunk beginnings with explicit identifier definitions we ignored.
This may change in future versions.

```
<nw2md>=
~ {#nw2md .python} <
    # convert noweb files to pandoc # =>
```

```
``` nw2md - convert noweb with pandoc markdown to pandoc (from there to X)
usage:

nw2md <my_file.nw >my_file.md

In combination with pandoc:

```
cat my_file.nw | nwmd2x.py > my_file.md
pandoc -f markdown -s --toc my_file.md my_file.html
pandoc -f markdown -s --toc --latex-engine=xelatex \
    --bibliography=lit.bib my_file.md my_file.tex
```

```
``` import re import sys

```
# some globals
code_chunk_index_as_subsection = False
# chunk index
code_chunks = {}
# markers of code chunk definitions
open_mark = "<<"
close_mark = ">>="
end = "^@"
in_slide_chunk = False

def find_code_chunk_definition(line):
    """look for code chunk definition and return chunk name""
```
# look for annotated code chunk header, e.g. `<<coode>>= (make)
match = re.match(open_mark + "([>]+)" + close_mark + "\s*\((\w+)\)\", line)
if not match:
    # ok, use normal noweb syntax
    match = re.match(open_mark + "([>]+)" + close_mark, line)
    if match:
        return (match.group(1), None)
    else:
        return None
else:
    return (match.group(1), match.group(2))

def contains_doc_chunk_header(line):
    match = re.match(end + "[ ]?", line)
    if match:
        return True
    else:
        return False

def convert_code_chunk_header(chunk_name, language_hint=None):
    ""
    convert noweb code chunk header to pandoc markdown
    The preceeding whitespace disables the generation of ‘:‘ at the end of
    the paragraph.
    ""

    if chunk_name in code_chunks and code_chunks[chunk_name] == 0:
        name = chunk_name
    else:
        name = chunk_name + " %d" % code_chunks[chunk_name]

    #label = ".. _" + clean_links(name) + ":"
    #label = ".. _" + name + ":"
    label = clean_links(name)

    # docutils 0.9+ has a code directive
    #return "\n" + label + "\n\n" + "\"" + name + "\"\n.. code:: python\n\n" #return "\n" + label + "\n\n" + "\‘" + name + "\’\n.. code:: python\n\n"
```python
# old implementation without language_hint
#else:
    line = line + " ." + language_hint
    line += "}\\n"

# old implementation without language_hint
#line = "\\n" + '\'' + name + '\'' + } \n
~~~ \{#' + label + '}

return line

def convert_slide_chunk_header(chunk_name):
    name = chunk_name

    #line = "\\n" + '\'' + name + '\'' + } \n
    line = "\\n" + '\'' + name + '\'' + } .markdown \n
    line = "\\n" + '\'' + name + '\'' + } \n
    line = "\\n" + '\'' + name + '\'' + } markdown \n
    line = "\\n" + '\'' + name + '\'' + } markdown \n
    return line

def clean_links(link):
    """some chars are not allowed in labels and refs""

    link = link.replace(" ", "\_")
    link = link.replace("/", "\_<")
    link = link.replace(".", "\_<")

    return link

def build_reference(match):
    """match function for sub""

    words = match.group(0)

    # clean up title
    title = words
    title = title.replace("[[","\_"")
    title = title.replace("]]","\_"")

    # clean up reference string
    # this is Sphinx style!
    ref = words
    ref = ref.replace("[[","\_:ref:\_"")
    ref = ref.replace("]]","\_"")
    ref = ref.replace(" ", "\_<")
```

5
# build final reference
ref = ref + "<" + title + ">/""

return ref

def convert_inline_references(line):
    converted_line = "x"
    converted_line_ = ""

    # we need a loop here because re.sub replaces only the left-most
    # appearance of the pattern
    while True:
        converted_line = re.sub("\[[^\[\]]*\]\]", build_reference, line)
        if converted_line == converted_line_:
            break
        else:
            converted_line_ = converted_line

    return converted_line

def replace_escaped_angle_bracket(line):
    # we use string concatenation to avoid the replacement here
    return line.replace("@<" + "<", "<" + "")

def handle_code_chunk_definition(match, line):
    language_hint = None
    chunk_name = match[0]
    if len(match)==2:
        language_hint = match[1]

    if chunk_name == "slide": 
        # this is not a real code chunk, but a slide chunk
        in_slide_chunk = True
        line = convert_slide_chunk_header(chunk_name)
    else:
        # normal code chunk
        #
        # store code chunk for index
        if not chunk_name in code_chunks:
            code_chunks[chunk_name] = 0
elif chunk_name in code_chunks and code_chunks[chunk_name] == 0:
    code_chunks[chunk_name] = 2
else:
    code_chunks[chunk_name] += 1

line = convert_code_chunk_header(chunk_name, language_hint)

return line, chunk_name

def handle_doc_chunk(line):
    return line

def translate(in_, out):
    # state indicator:
    # None: doc chunk -> replace [[...]]
    # name: code chunk -> add two spaces
    #
    # we start in doc chunk mode
    chunk_name = None

    for line in in_:
        # look for a code chunk definition
        match = find_code_chunk_definition(line)
        if match:
            line, chunk_name = handle_code_chunk_definition(match, line)
        else:
            # is the a new doc chunk?
            # don’t forget to remove the @ marker
            if contains_doc_chunk_header(line):
                chunk_name = None
                line = "\n~~~\n" + convert_inline_references(line[1:])

            if chunk_name:
                # add two spaces
                line = " " + line
                # replace @<< with < <
                line = replace_escaped_angleBracket(line)
            else:
                line = convert_inline_references(line)

        out.write(line)

7
generate_index(sys.stdout)

def generate_index(out):
    # write header
    if len(code_chunks.keys()):
        out.write("\n\n----\n
Code Chunks\n
")
    # section or subsection?
    if code_chunk_index_as_subsection:
        out.write("----------\n")
    else:
        out.write("========\n")

    # generate list of chunks
    for chunk_name in sorted(code_chunks.keys()):
        if chunk_name in code_chunks and code_chunks[chunk_name] == 0:
            name = chunk_name
            #ref = "* `{0}`_\n".format(name)
            #ref = "* `{0}`\n".format(clean_links(name))
            ref = "* [{}][#{1}]\n".format(name, clean_links(name))
            out.write(ref)
        else:
            name = chunk_name
            ref = "* `{0}`\n".format(name)
            #ref += " * `{0}`_\n".format(name.replace(" ", "_"))
            #ref += " * '{0}'_\n".format(name)
            ref += " * [{}][#{1}]\n".format(name, clean_links(name))
            out.write(ref)

        for i in range(2, code_chunks[chunk_name]+1):
            name = chunk_name + " %d" % i

            # this is Sphinx style!
            #ref = "* :ref:`{0} <{1}>`\n".format(name.replace(" ", "_"), name)
            #ref = " * `{0}`\n".format(name)
            ref = " * [{0}][#{1}]\n".format(name, clean_links(name))
            #ref = " * `{0}`\n".format(clean_links(name))
            out.write(ref)

if __name__ == "__main__":
    if len(sys.argv) > 1:
        if sys.argv[1] == "-s":
            8
3 The tangle tool

According to the syntax enhancement a tangle is necessary that understand the code chunk definitions:

```python
<code>
code_chunk_index_as_subsection = True
translate(sys.stdin, sys.stdout)
</code>
```

According to the syntax enhancement a tangle is necessary that understand the code chunk definitions:

```python
<code>
code_chunk_index_as_subsection = True
translate(sys.stdin, sys.stdout)
</code>
```

```python
def collect_code_chunks(filenames):
    code_chunks = {}
    with fileinput.input(filenames) as f:
        collect_chunks_from_file(f, code_chunks)
    return code_chunks

def collect_chunks_from_file(file, code_chunks):
```
chunk_name = None
open_mark = "<<"
close_mark = ">>"

for line in file:
    # look for annotated code chunk header, e.g. <<coode>>= (make)
    # this is the enhanced noweb syntax defined by me
    match = re.match(open_mark + "([^{>]+)" + close_mark + "=" + \s*\(([^\w]+)\)\", line)
    if not match:
        # ok, use normal noweb syntax
        match = re.match(open_mark + "([^{>]+)" + close_mark + "=", line)
        if match:
            # found normal code chunk definition
            chunk_name = match.group(1)
            # create new entry if necessary
            # bug in original version!
            if not chunk_name in code_chunks:
                code_chunks[chunk_name] = []
        else:
            # no code chunk definition, maybe a new doc chunk?
            match = re.match("@", line)
            if match:
                chunk_name = None
            elif chunk_name:
                # if chunkName is defined, we are in a code chunk and
                # collect the line
                code_chunks[chunk_name].append(line)
            else:
                # found a new code chunk definition (enhanced syntax)
                chunk_name = match.group(1)
                # create new entry if necessary
                # bug in original version!
                if not chunk_name in code_chunks:
                    code_chunks[chunk_name] = []

def expand(chunk_name, code_chunks, indent):
    """expand given code chunk""

    open_mark = "<<"
close_mark = ">>"

    try:
        chunk_lines = code_chunks[chunk_name]
except KeyError:
    print("the given chunk name '{0}' was not found".format(chunk_name), file=sys.stderr)
    sys.exit(1)

expanded_chunk_lines = []
for line in chunk_lines:
    match = re.match("(\s*)" + open_mark + "([^>]+)" + close_mark + "\s*$", line)
    if match:
        expanded_chunk_lines.extend( expand(match.group(2), code_chunks, indent + match.group(1)) )
    else:
        expanded_chunk_lines.append(indent + line)
return expanded_chunk_lines

def tangle(chunk_name, filenames, out):
    """tangle filenames with given chunk name""

code_chunks = collect_code_chunks(filenames)
lines = expand(chunk_name, code_chunks, "")
sys.stdout.write("".join(lines))

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="tangle files with enhanced noweb structure")
    parser.add_argument("-R", dest="chunk_name", metavar="chunk_name",
                        help="chunk name to start with", required=True)
    parser.add_argument("filename", nargs="+", help="filename(s) to use")

    args = parser.parse_args()
    out = sys.stdout
    tangle(args.chunk_name, args.filename, out)

<<footer>>

3.1 Common Code Snippets

<author and version 2>=
4 Todo

- add noweb navigation (or a modern form of it)
- add %def syntax support
#!/bin/sh

if [ -z "${NOWEB_SOURCE}" ]; then
    NOWEB_SOURCE=nw2md.nw
fi

if [ -z "${NOWEB_CODE}" ]; then
    NOWEB_CODE='pwd'/code
fi

# check if we need to create target dirs
[ -d ${NOWEB_CODE} ] || mkdir -p ${NOWEB_CODE}

FILES="nw2md tangle"

for f in ${FILES}; do
    ./scripts/tangle -R"$f" "$NOWEB_SOURCE" > "$NOWEB_CODE/$f"
    chmod u+x "$NOWEB_CODE/$f"
done

5 Code Chunks

- author and version
  - author and version
  - author and version 2
- build-script
- copyright
- footer
- header
- nw2md
- tangle