TrY

Ajabu Tex
ajabutex@gmail.com

January 2nd, 2014

Contents

Part I
Preliminaries

1 Introduction

This document describes the algorithms needed to implement a system for the automation of the compilation of (La)TEX documents.

The characteristics of the implementation allow to use the program for the automation of other procedures, such as the compilation of files of *Literate Programming*, and in all cases in which a sequence of `bash` commands is needed to process a text file.

The language used is *Python*, chosen for its expressive power, stability, large endowment of standard libraries and the fact to be natively installed on most Linux systems.

2 Scope of use

There are already some great programs aimed to automate the compilation of TEX and LATEX documents: *latexmk* by John Collins (http://www.phys.psu.edu/~collins/software/latexmk-jcc/), *Rubber* by Emmanuel Beffara (http://launchpad.net/rubber), *arara* by Paulo Cereda (http://cereda.github.io/arara/) and many others; you can find some of them on CTAN (http://www.ctan.org/topic/compilation).

arara is the program from which I took inspiration for the creation of *TrY*. I really like the idea behind the implementation of *arara*, namely that of making explicit in the document the list of commands needed to compile it.

arara is a nice piece of software but has some limitations that make it useless for my work, subtle limits relating to the handling of the Linux filters, pipes and subshells.

On the other hand *TrY* works only on Linux (and perhaps Mac OS), but on Linux practically has no limitation. *TrY* can do anything that can be done in a Linux terminal. In practice the *TrY* commands are nothing but `bash` commands inserted in the comment lines of a TEX document.

Last but not least, *TrY* could not be simpler than it is.

3 How to use

3.1 Installation

Be sure that the file `try` is executable and then copy it in the `/usr/bin` directory with the following commands:

```
chmod +x try
sudo cp try /usr/bin/try
```

3.2 Mechanics

TrY scans the document looking for *TrY* statements into the comment lines and executes them one after the other.

The typical form of the *TrY* commands is the following:

```
%$ <command> <parameter> <parameter> ...
```

1. The first element of the line must be the comment symbol (`%`);
2. immediately followed by the dollar sign (`$`);
3. then follows a space;
4. finally, a Linux command with its parameters.

So let's say you are writing a LaTeX document named `nicedoc.tex`; if you put at the beginning of the file the following line of comment:

```
%
%$ lualatex --enable-write18 nicedoc.tex
%
```

you can simply compile the document with the following command line:

```
try nicedoc.tex
```

As an added benefit, anyone who will read your manuscript will know that the required engine for the compilation is LuaLaTeX and that is required the activation of the `write18` option.

In a *TrY* statement you can give instructions to process any file, but if the file name passed to the command as a parameter is the same of the name of the document itself, the file name can be replaced by the placeholder '`$0`'.

So, in the example above, you can use into the manuscript the following statement:

```
%
%$ lualatex --enable-write18 $0
%
```

that is a little bit more compact.

Another example: you are writing `bigdoc.tex`, a complex LaTeX document, with management of many different fonts, extensive indexes and some parts computed on-the-fly by python. The typical cicle of compilation consists of several bash commands. But you can type those commands at the beginning of the file once and forever as *TrY* statements:

```
%
%$ xelatex -shell-escape $0
%$ bibtex -min-crossref=3 bigdoc.aux
%$ xelatex -shell-escape $0
%$ xelatex -shell-escape $0
%
```

and let *TrY* do the hard work for you. The nice thing is that this way the document show explicitly the engine needed, the number of compilations required and so on, making easier the collaborative editing of the document.

With *TrY* you can also process other files than TEX or LATEX documents. In this case you have to tell the program what is the comment symbol used, using the statement `$trycommentchar=<a_comment_char>$`. For example:

```
#
# Sample python listing
#
# $trycommentchar=#$
#
#$ python -B $0 -W message
#
```

As a last example, you can put at the beginning of a `noweb` file of *Literate Programming* the following statements. Let's say that the name of the file is `niceprog.nw`:

```
%
% extraction and compilation of the documentation
%$ noweave -index -latex $0 > niceprog-doc.tex
%$ pdflatex niceprog-doc.tex
%$ pdflatex niceprog-doc.tex
%$ pdflatex niceprog-doc.tex
%
% extraction and compilation of the program
%$ notangle -Rniceprog $0 > niceprog.pas
%$ fpc -Fudirectory -gh niceprog.pas
%
```

Now you have only to launch *TrY* in a terminal with the following command line

```
try niceprog.nw
```

to obtain the documentation, the listing and the executable file in just a single step.

3.3 The command line

To process a document with *TrY* open a terminal and use the following command line:

```
$ try [file [--log] [--safe] [--verbose] | --help | --version]
$ try [file [-l] [-s] [-v] | -h | -V]
```

`[file]` is the name of the document, including the extension, that you want to process;

`[-h | --help]` print a help message and exits;

`[-s | --safe]` print the list of commands before to execute them;

`[-l | --log]` produce a log file with the operations carried out by the program;

`[-v | --verbose]` print the output of the program;

`[-V | --version]` output version information and exit.

If you use a programmable text editor, it can be set for use with *TrY*. For example, to create a command that launches *TrY* from GEdit you can do this way:

1. menu `Tools -> Manage externals tools`
2. list `All Languages`. Click on `Add a new tool`. Name it `Compile TrY`
3. In the `Edit` pane write:

```
#!/bin/bash
try $GEDIT_CURRENT_DOCUMENT_NAME --log --verbose
```

4. `Shortcut Key -> Shift+Ctrl+%` (or whatever you want)
5. `Save -> Current document`
6. `Input -> Nothing`
7. `Output -> Display in bottom pane`
8. `Applicability -> All documents -> All Languages`

Now you can start building with *TrY* the document you are working on with GEdit simply by pressing the key combination `Shift+Ctrl+%`.

Part II
The code

Here is an outline of the program:

6a ⟨*try* 6a⟩≡
⟨*shabang* 15c⟩
⟨*license statements* 16⟩
⟨*standard libraries* 6b⟩
⟨*global constants* 8a⟩
⟨*procedure to notify messages on the screen and in the logfile* 9a⟩
⟨*procedure to get the list of command-line arguments* 11a⟩
⟨*procedures to collect the TrY statements* 11b⟩
⟨*procedure for the request for a confirmation* 14b⟩
⟨*procedure for the execution of the TrY statements* 15b⟩
⟨*main module* 7⟩

The main module collects the arguments in the variable `args` from the command line; if a filename is passed it scans the document to find lines of comment containing *TrY* statements that are then stored in the variable `try_statements`. At last executes the *TrY* statements itself, but only if has a confirmation from the user. Every step of this process is notified on the screen and in the logfile if the proper parameters are passed to the program via the command line.

6b ⟨*standard libraries* 6b⟩≡ (6a) 8b▷

```
import sys
```

7 ⟨*main module* 7⟩≡ (6a)

```
if __name__ == "__main__":
    args = get_args()
    if not args.filename ==’’:
        notify(’This is ’ + prog_name + ’ ver. ’ + prog_version + \
            ’ revision ’ + prog_revisiondate + ’\n’, args)
        try_statements = get_trystatements_from(args)
        if len(try_statements) == 0:
            notify(’No commands found in ’ + args.filename + ’.’, args)
            sys.exit(’Nothing to execute.’)
        else:
            if args.safe:
                if execution_confirmed_of(try_statements):
                    execute_statements(try_statements, args)
                else:
                    notify(’Execution stopped by the user.’, args)
                    sys.exit(’Execution aborted.’)
            else:
                execute_statements(try_statements, args)
```

Uses `execute_statements` 15b, `execution_confirmed_of` 14b, `get_args` 11a, `get_trystatements_from` 14a, `notify` 9b, `prog_name` 8a, `prog_revisiondate` 8a, and `prog_version` 8a.

Now we have to build every single command called by the main module. First of all we have to define some global constants, namely the strings used in the output of the program.

8a ⟨*global constants* 8a⟩≡ (6a)

```
prog_name = 'TrY'
prog_cmd = 'try'
prog_version = '4.0'
prog_author = 'Ajabu Tex'
prog_revisiondate = 'Jan 2, 2014'
prog_releasedate = '2014'
prog_license = prog_name + ' ' + prog_version + '\n' + \
    'Copyright 2013-' + prog_releasedate + ' ' + prog_author + '\n' + \
    'There is NO warranty.  Redistribution of this software is\n' + \
    'covered by the terms of both the ' + prog_name + ' copyright and\n' + \
    'the GNU General Public License.\n' + \
    'For more information about these matters, see the file\n' + \
    'named COPYING and the ' + prog_name + ' source.\n' + \
    'Author of ' + prog_name + ': ' + prog_author + '.'
```

Defines:
`prog_author`, used in chunk 11a.
`prog_cmd`, used in chunk 11a.
`prog_license`, used in chunk 11a.
`prog_name`, used in chunks 7, 9a, 11a, 14a, and 15b.
`prog_releasedate`, used in chunk 11a.
`prog_revisiondate`, used in chunks 7 and 9a.
`prog_version`, used in chunks 7 and 9a.

The function `append_to_logfile()` takes as input a file name and a string. If the string is that opening the file then the file is overwritten; otherwise is opened in 'append' mode. Then the function calculates the day and time, join them to the string passed as argument and writhe this new string into the log file. The name of the log file is the same name of the file to process –included the extension– plus the extension `.trylog`. Example: if I call the program with the following command line

```
try -l niceprog.nw
```

the name of the log file will be `niceprog.nw.trylog`.

8b ⟨*standard libraries* 6b⟩+≡ (6a) ◁6b 10▷

```
from datetime import datetime
```

9a ⟨*procedure to notify messages on the screen and in the logfile* 9a⟩≡ (6a) 9b▷

```
def append_to_logfile(a_file_name, a_string_line):
    if a_string_line.startswith('This is ' + prog_name + \
    ' ver. ' + prog_version + ' revision ' + prog_revisiondate):
        openmode = 'w'
    else:
        openmode = 'a'
    now = datetime.now()
    time_string = now.strftime("%d %b %Y %H:%M:%S.%f")
    log_string = time_string + ' ' + a_string_line
    logfile_name = a_file_name + '.trylog'
    logfile = open(logfile_name, openmode)
    logfile.write(log_string + '\n')
    logfile.close
```

Defines:
`append_to_logfile`, used in chunk 9b.
Uses `prog_name` 8a, `prog_revisiondate` 8a, and `prog_version` 8a.

Every time the program has to send something to the screen or to the logfile, it calls the function `notify()` passing to it the text to show and the arguments passed to the program by the command line.

9b ⟨*procedure to notify messages on the screen and in the logfile* 9a⟩+≡ (6a) ◁9a

```
def notify(text, args):
    if args.log:
        append_to_logfile(args.filename, text)
    if args.verbose:
        print (text)
```

Defines:
`notify`, used in chunks 7 and 13–15.
Uses `append_to_logfile` 9a.

The first operation the program has to do is to collect the parameters from the command line. This issue is exploited with the standard library `argparse`. The program can be called with the following parameters:

log send an output in a log file;

safe safe mode: before to execute the list of statements found in the processed file prints them out on the screen and asks the user for a confirmation;

verbose outputs on the screen the stages of processing;

version print on the screen the version number and copyright information;

file the name of the file to process;

where `file` is a positional parameter and the others are optional parameters. So we have the following collection of parameters, returned by the function `get_args` as a list. The function `get_args` collect the command line parameters and manage all the stage pertaining to the command line itself. If there are no parameters prints a help screen; if the program is called with the `--version` parameter prints a screen with the version informations.

To get useful documentation from the python help system type in a terminal:

```
$ pydoc argparse
$ pydoc argparse.ArgumentParser
```

10 ⟨*standard libraries* 6b⟩+≡ (6a) ◁8b 15a▷

```
import argparse
```

11a ⟨*procedure to get the list of command-line arguments* 11a⟩≡ (6a)

```
def get_args():
    parser = argparse.ArgumentParser(
        prog=prog_cmd,
        usage='%(prog)s [options] filename',
        formatter_class=argparse.RawDescriptionHelpFormatter,
        description='''\
            ''' + prog_name + '''
            TeX automation tool''',
        epilog='(C) 2013-' + prog_releasedate + ' ' + prog_author)
    parser.add_argument('-l', '--log',
        action='store_true',
        help='generate a log output')
    parser.add_argument('-s' ,'--safe',
        action='store_true',
        help='print the list of commands prior to execute them')
    parser.add_argument('-v', '--verbose',
        action='store_true',
        help='print the command output')
    parser.add_argument('-V', '--version',
        action='version',
    version=prog_license)
    parser.add_argument('filename', nargs='?', default='')
    args = parser.parse_args()
    if args.filename == '':
        parser.print_help()
    return args
```

Defines:
`get_args`, used in chunk 7.
Uses `prog_author` 8a, `prog_cmd` 8a, `prog_license` 8a, `prog_name` 8a, and `prog_releasedate` 8a.

Now the variable `args` contain a list whith the name of the file to process and the parameters to apply.

The extraction and the collection of the *TrY* statements from a file goes through three stages: first we have to know what is the comment `string.rpartition()` flag used in the file, then we have to extract from the file all the lines that are TrY statements, at last we must to return that statements as a list of string ready to be passed to the procedure of execution.

11b ⟨*procedures to collect the TrY statements* 11b⟩≡ (6a)

⟨*procedure for the recognition of the char used as comment flag* 12⟩
⟨*procedure to extract TrY statements from a list of strings* 13⟩
⟨*procedure to return the collected statements* 14a⟩

`get_commentchar_from()` takes as input a list of lines of text and returns the default comment character (‘`%`’) or the one defined by the statement `$trycommentchar=...$` that may be present.

The splitting of the statement is a bit tricky: the function `string.rpartition(sep)` search for the separator sep in a string, starting at the end of the string, and return a list containing the part before it, the separator itself, and the part after it.

In our case the ipothetyc statement `"% $trycommentchar=x$"` is splitted in the list `["% ", "$trycommentchar=", "x$"]`. We take the last element and split it again with the function `string.rsplit([sep]), that return a list of the words in the string, using sep as the delimiter string, starting at the end of the string and working to the front. In this way we obtain the list [[["x", "$"]`, where the first element is the one we were looking for.

For the python documentation:

```
$ pydoc str.rpartition
$ pydoc str.rsplit
```

12 ⟨*procedure for the recognition of the char used as comment flag* 12⟩≡ (11b)

```
def get_commentchar_from(str_list):
    result = '%'
    for str_line in str_list:
        if '$trycommentchar=' in str_line:
            str_line = str_line.rpartition('$trycommentchar=')[2]
            str_line = str_line.rsplit('$')[0]
            result = str_line
    return result
```

Defines:
`get_commentchar_from`, used in chunk 14a.

The function `get_trystatement_list_from()` takes as input a list of lines of text, the list of arguments and the comment flag. Returns a list of *TrY* stateents, which is a list of bash commands that are present on the lines that begin with the string `flag`, in which has been replaced any occurrence of the string '`$0`' with the name of the file to process.

A useful python documentation is in:

```
$ pydoc str.partition
```

13 ⟨*procedure to extract TrY statements from a list of strings* 13⟩≡ (11b)

```
def get_trystatement_list_from(str_list, args, flag):
    file_name = args.filename
    result = []
    for str_line in str_list:
        if str_line.startswith(flag):
            if '$0' in str_line:
                str_line = str_line.replace('$0', file_name)
            str_line = str_line.strip()
            str_line = str_line.partition(flag)[2]
            notify('Found command ' + str_line, args)
            result.append(str_line)
    return result
```

Defines:
`get_trystatement_list_from`, used in chunk 14a.
Uses `notify` 9b.

`get_trystatements_from()` takes in input the list of arguments passed to the program via the command line. Open the file and extract the whole its content as a list of strings. Those strings are filtered through the procedure `get_commentchar_from()` and `get_trystatement_list_from()` to obtain a list of bash commands to pass to the procedure of execution.

14a ⟨*procedure to return the collected statements* 14a⟩≡ (11b)

```
def get_trystatements_from(args):
    file_name = args.filename
    notify('Looking for ' + prog_name + ' commands in ' + file_name, args)
    f = open(file_name, 'r')
    str_lines = f.readlines()
    f.close()
    comment_char = get_commentchar_from(str_lines)
    notify('Comment char used: ' + comment_char, args)
    try_statement_flag = comment_char + '$ '
    result = get_trystatement_list_from(str_lines,
        args, try_statement_flag)
    return result
```

Defines:
`get_trystatements_from`, used in chunk 7.
Uses `get_commentchar_from` 12, `get_trystatement_list_from` 13, `notify` 9b, and `prog_name` 8a.

`execution_confirmed_of` takes the list of statements found into the processed file, print them out on the screen and ask the user a confirmation about the execution. The function returns a boolean value.

14b ⟨*procedure for the request for a confirmation* 14b⟩≡ (6a)

```
def execution_confirmed_of(str_lst):
    prompt = 'These are the commands that will be processed:\n'
    for str_line in str_lst:
        prompt += str_line + '\n'
    prompt += '\n\nDo you want to continue? [Yes/No]\n'
    answer = raw_input(prompt)
    result = answer in ['Y', 'y', 'Yes', 'yes']
    return result
```

Defines:
`execution_confirmed_of`, used in chunk 7.

`execute_statements()` takes as input a list of strings representing Unix commands and passes them to the operating system for the execution. If there's an error the execution is aborted.

Useful python documentation:

```
$ pydoc subprocess
$ pydoc subprocess.call
```

15a ⟨*standard libraries* 6b⟩+≡ (6a) ◁10

```
import os
import subprocess
```

15b ⟨*procedure for the execution of the TrY statements* 15b⟩≡ (6a)

```
def execute_statements(try_statements, args):
    if len(try_statements) == 0:
        notify('No commands found.', args)
        sys.exit()
    else:
        for statement in try_statements:
            notify('Executing instruction: ' + statement, args)
            exit_status = subprocess.call([statement],
                stdout=subprocess.PIPE, shell=True)
            if exit_status != 0:
                notify(prog_name + ': error at system level. \n' + \
                    'Execution aborted with exit status ' + \
                    str(exit_status) + '\n', args)
                sys.exit(exit_status)
            else:
                notify('SUCCESS', args)
```

Defines:
`execute_statements`, used in chunk 7.
Uses `notify` 9b and `prog_name` 8a.

That's all. At the end here are listed the shabang and the usual declarations about license and copyright, written as stated by the GNU GPL web page: http://www.gnu.org/licenses/gpl-howto.en.html

15c ⟨*shabang* 15c⟩≡ (6a)

```
#!/usr/bin/env python
```

16 ⟨*license statements* 16⟩≡ (6a)

```
# Copyright 2013-2014 Ajabu Tex
#
# This file is part of TrY.
#
# TrY is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# TrY is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with TrY.  If not, see <http://www.gnu.org/licenses/>.
```

Part III
What's new in version 4

The code was almost all rewritten to implement two main new features:

the safe mode: before of executing the instructions found in the processed file *TrY* prints them out on the screen and asks for a confirmation from the user;

the command line: now is implemented with the `argparse` library. The code is more solid and robust, easily readable and maintenable;

I've also rewritten almost all the documentation, now much more in a *Literate Programming* style.

At last the terms of license are no more those of LPPL but GNU GPL – General Public License.

Part IV
Credits and licence

Program: TrY
Version: 4.0
Release: 2013-2014
Author: Ajabu Tex <ajabutex@gmail.com>
Description: Automation tool for TeX documents
Terms of use: GNU General Public License.

The project repository is hosted on Bitbucket; feel free to contribute in any way. If you want to fork the project or want to send a pull request, or simply want to download the latest version, go to

```
https://bitbucket.org/ajabutex/try
```

The program is also kindly hosted as a package on CTAN:

```
http://www.ctan.org/pkg/try
```

To obtain the program code and the documentation the source file `try.nw` was compiled with `noweb`, a simple tool for *Literate Programming* written by Norman Ramsey. You can find informations about it at:

```
http://www.cs.tufts.edu/~nr/noweb/
http://www.ctan.org/tex-archive/web/noweb
```

Part V
Indexes

4 Chunks

⟨*global constants* 8a⟩
⟨*license statements* 16⟩
⟨*main module* 7⟩
⟨*procedure for the execution of the TrY statements* 15b⟩
⟨*procedure for the recognition of the char used as comment flag* 12⟩
⟨*procedure for the request for a confirmation* 14b⟩
⟨*procedure to extract TrY statements from a list of strings* 13⟩
⟨*procedure to get the list of command-line arguments* 11a⟩
⟨*procedure to notify messages on the screen and in the logfile* 9a⟩
⟨*procedure to return the collected statements* 14a⟩
⟨*procedures to collect the TrY statements* 11b⟩
⟨*shabang* 15c⟩
⟨*standard libraries* 6b⟩
⟨*try* 6a⟩

5 Identifiers