NLP-annotate documents on Surfsaras Lisa computer'

Paul Huygen <paul.huygen@huygen.nl>

1st July 2016
14:18 h.

Abstract

This is a description and documentation of a system that uses SurfSara's supercomputer Lisa to perform large-scale NLP annotation on Dutch or English documents. The documents should have the size of typical newspaper-articles and they should be formatted in the NAF format. The annotation-pipeline can be found on "Newsreader pipeline".

Contents

1 Introduction

This document describes a system for large-scale linguistic annotation of documents, using supercomputer Lisa. Lisa is a computer-system co-owned by the Vrije Universiteit Amsterdam. This document is especially useful for members of the Computational Lexicology and Terminology Lab (CLTL) of the Vrije Universiteit Amsterdam who have access to that computer. Currently, the documents to be processed have to be encoded in the *NLP Annotation Format* (NAF).

The annotation of the documents will be performed by a "pipeline" that has been set up in the Newsreader-project [1]. The installation of this pipeline is performed by script that can be obtained from Github.

1.1 How to use it

Quick user instruction:

1. http://www.newsreader-project.eu

1. Get an account on Lisa.
2. Clone the software from Github. This results in a directory-tree with root `Pipeline_NL_Lisa`.
3. "cd" to `Pipeline_NL_Lisa`.
4. Run `stripnw` and `nuweb`
5. Create a subdirectory `data/in` and fill it with (a directory-structure containing) raw NAF's that have to be annotated.
6. Run script `runit`.
7. Repeat to run `runit` on a regular bases (e.g. twice per hour) until subdirectory `data/in/` and subdirectory `data/proc` are both empty.
8. The annotated NAF files can be found in `data/out`. Documents on which the annotation failed (e.g. because the annotation took too much time) have been moved to directory `data/fail`.

1.2 How it works

1.2.1 Moving files around

The system expects a subdirectory `data` and a subdirectory `data/in` in it's root directory. It expects the NAF files to be processed to reside in `data/in`, possibly distributed up in a directory-structure below `data/in`. The NAF files and the logfiles are stored in the following subdirectories of the `data` subdirectory:

proc: Temporary storage of the input files while they are being processed.
fail: For the input NAF's that could not be processed.
log: For logfiles.
out: The annotated files appear here.

From now on we will call these directories *trays* (e.g. intray, proctray).

if `data/in` has a directory-substructure, the structure is copied in the other directories. In other words, when there exists a file `data/in/aap/noot/mies.naf`, the system generates the annotated naf `data/out/aap/noot/mies.naf` and logfile`data/log/aap/noot/mies.naf` (although the latter is not in NAF format). When processing fails, the system does not generate `data/out/aap/noot/mies.naf`, but it moves `data/in/aap/noot/mies.naf` to `data/fail/aap/noot/mies.naf`.

The file in the log-tray contains the error-output that the NLP-modules generated when they operated on the NAF.

Processing the files is performed by jobs. Before a job processes a document, it moves the document from `data/in` to `data/proc`, to indicate that processing this document has been started. When the job is not able to perform processing to completion (e.g. because it is aborted), the NAF file remains in the `proc` subdirectory. At regular intervals a management script runs, and this moves NAF's of which processing has not been completed back to `in`.

While processing a document, a job generates log information and stores this in a log file with the same name as the input NAF file in directory `log`. If processing fails, the job moves the input NAF file from `proc` to `fail`. Otherwise, the job stores the output NAF file in `out` and removes the input NAF file from `proc`.

⟨ *parameters* 3 ⟩ ≡

```
export root=/home/phuijgen/nlpt/Pipeline-NL-Lisa
export intray=/home/phuijgen/nlpt/Pipeline-NL-Lisa/data/in
export proctray=/home/phuijgen/nlpt/Pipeline-NL-Lisa/data/proc
export outtray=/home/phuijgen/nlpt/Pipeline-NL-Lisa/data/out
export failtray=/home/phuijgen/nlpt/Pipeline-NL-Lisa/data/fail
export logtray=/home/phuijgen/nlpt/Pipeline-NL-Lisa/data/log
```

◇

Fragment defined by 3, 7b, 8d, 11b, 14b, 16ac, 17b, 20c.
Fragment referenced in 4.
Defines: `failtray` 6afl, 25a, `intray` 6bkl, 9b, 10c, 23a, `logtray` 6ahl, `outtray` 6al, `root` 8a, 9c, 13b, 23a, 28c.

1.2.2 Management-script

When a user has put NAF files in `data/in`, something has to take care of starting jobs ot annotate the files and moving abandoned files from the proctray back to the intray. This is performed by a script named `runit`, that should be started from time to time. When there are files present in the intray, runit should be started 2-3 times per hour.

1.2.3 The NLP-modules

In the annotation process a series of NLP modules operate in sequence on the NAF. The annotation-process is described in section 7.3.

1.2.4 Set parameters

The system has several parameters that will be set as Bash variables in file `parameters`. The user can edit that file to change parameters values

`"../parameters"` 4≡
⟨ *parameters* 3, ... ⟩
◇

2 File management

Viewed from the surface, what the pipeline does is reading, creating, moving and deleting files. The input is a directory tree with NAF files, the outputs are similar trees with NAF files and log files. The system generates processes that run at the same time, reading files from the input tree. It must be made certain that each file is processed by only one process. This section describes and builds the directory trees and the "stopos" system that supplies paths to input NAF files to the processes.

2.1 Move NAF-files around

The user may set up a structure with subdirectories to store the input NAF files. This structure must be copied in the other data directories.

The following bash functions copy resp. move a file that is presented with it's full path from a source data directory to a similar path in a target data-directory. Arguments:

1. Full path of sourcefile.
2. Full path of source tray.
3. Full path of target tray

The functions can be used as arguments in `xargs`.

⟨ *functions* 5a ⟩ ≡

```
function movetotray () {
local file=$1
local fromtray=$2
local totray=$3
local frompath=${file%/*}
local topath=$totray${frompath##$fromtray}
mkdir -p $topath
mv $file $totray${file##$fromtray}
}

export -f movetotray

◇
```

Fragment defined by 5ab, 26b, 27a.
Fragment referenced in 25c, 28c.
Defines: `movetotray` 10c, 23a, 25a.

⟨ *functions* 5b ⟩ ≡

```
function copytotray () {
local file=$1
local fromtray=$2
local totray=$3
local frompath=${file%/*}
local topath=$totray${frompath##$fromtray}
mkdir -p $topath
cp $file $totray${file##fromtray}
}

export -f copytotray

◇
```

Fragment defined by 5ab, 26b, 27a.
Fragment referenced in 25c, 28c.
Defines: `copytotray` Never used.

2.2 Count the files and manage directories

When the management script starts, it checks whether there is an input directory. If that is the case, it generates the other directories if they do not yet exist and then counts the files in the directories. The variable `unreadycount` is for the total number of documents in the intray and in the proctray.

⟨ *check/create directories* 6a ⟩ ≡
```
mkdir -p $outtray
mkdir -p $failtray
mkdir -p $logtray
mkdir -p $proctray
⟨ count files in tray (6b intray,6c incount ) 6j ⟩
⟨ count files in tray (6d proctray,6e proccount ) 6j ⟩
⟨ count files in tray (6f failtray,6g failcount ) 6j ⟩
⟨ count files in tray (6h logtray,6i logcount ) 6j ⟩
unreadycount=$((incount + $proccount))
◇
```
Fragment defined by 6ak.
Fragment referenced in 28c.
Uses: logcount 6a.

⟨ *count files in tray* 6j ⟩ ≡
```
@2=‘find $@1 -type f -print | wc -l‘
◇
```
Fragment referenced in 6a.
Uses: print 36a.

The processes empty the directory-structure in the intray and the proctray. So, it might be a good idea to clean up the directory-structure itself.

⟨ *check/create directories* 6k ⟩ ≡
```
find $intray -depth -type d -empty -delete
find $proctray -depth -type d -empty -delete
mkdir -p $intray
mkdir -p $proctray
◇
```
Fragment defined by 6ak.
Fragment referenced in 28c.
Uses: intray 3.

2.3 Generate pathnames

When a job has obtained the name of a file that it has to process, it generates the full-pathnames of the files to be produced, i.e. the files in the proctray, the outtray or the failtray and the logtray:

⟨ *generate filenames* 6l ⟩ ≡
```
filtrunk=${infile##$intray/}
export outfile=$outtray/${filtrunk}
export failfile=$failtray/${filtrunk}
export logfile=$logtray/${filtrunk}
export procfile=$proctray/${filtrunk}
export outpath=${outfile%/*}
export procpath=${procfile%/*}
export logpath=${logfile%/*}
◇
```
Fragment referenced in 12a.
Defines: filtrunk Never used, logfile 22a, logpath 23a, outfile 12a, 22a, 23c, 25a, outpath 23a, 25a, procfile 23abc, 24ab, 25a, procpath Never used.
Uses: failtray 3, intray 3, logtray 3, outtray 3.

2.4 Manage list of files in Stopos

The processes in the jobs that do the work pick NAF files from `data/in` in order to process them. There must be a system that arranges that each NAF file is picked up only once, by only one job-process. To do this, we use the "Stopos" system that has been implemented in Lisa. The management script makes a list of the files in `\data\in` and passes it to a "stopos pool" where the work processes can find them.

A difficulty is, that there is no way to look into stopos, other than to pick a file. The intended way of using Stopos is, to fill it with a given set of parameters and then start jobs that process the parameters one-by-one until there are no unused parameters left. In our system however, we would like to add new input-files while the system is already working, and there is no direct way to tell whether the name of a given input-file has already been added to Stopos or not. Therefore we need a kind of shadow-bookkeeping, listing the files that have already been added to Stopos and removing processed files from the list.

In order to be able to use stopos, first we have to "load" the "stopos module":

⟨ *load stopos module* 7a ⟩ ≡
```
module load stopos
```
◇

Fragment referenced in 25c, 28c.
Defines: `module` 21b, 22a, `stopos` 9a, 10a, 11ac, 12b.

A list of parameters like the filenames in our problem is called a "Stopos pool". Give our pool a name:

⟨ *parameters* 7b ⟩ ≡
```
export stopospool=dptpool
```
◇

Fragment defined by 3, 7b, 8d, 11b, 14b, 16ac, 17b, 20c.
Fragment referenced in 4.
Defines: `stopospool` 9a, 10a, 11ac, 12b.

2.4.1 Set up/reset pool

In this section filenames are added to the Stopos pool. Adding a large amount of filenames takes much time, so we do this sparingly. We do it as follows:

1. First look how many filenames are still available in the pool. If there are still sufficient filenames in the pool to keep the jobs working for the next half hour, we do nothing. On the other hand. If the pool is empty, we renew it (i.e. purge it and re-generate a new, empty pool). In this way the contents of the pool is aligned with the shadow-bookkeeping of the filenames. Also when there are no jobs or when there are no files in the intray, we renew the pool. If the pool is running out, we add filenames to the pool.
2. Generate a file `infilelist` that contains the paths to the files in the intray.
3. Assume file `old.filenames`, if it exists, contains the filenames that have been inserted in the Stopos pool.
4. Delete from `old.filenames` the names of the files that are no longer in the intray. They have probably been processed or are being processed.
5. Move the files in the proctray that are not actually being processed back the intray. We know that these files are not being processed because either there are no running jobs or the files reside in the proctray for a longer time than jobs are allowed to run.
6. Make file `infilelist` that lists files that are currently in the intray.
7. Remove the filenames that can also be found in `old.infilelist` from `infilelist`. After that `infilelist` contains names of files that are not yet in the pool.

8. Add the files in `infilelist` to the pool.
9. Add the content of `infilelist` to `old.infilelist`.

⟨ *update the stopos pool* 8a ⟩ ≡
```
cd $root
⟨ is the pool full or empty? (8b pool_full,8c pool_empty ) 9a ⟩
if
  [ $pool_full -ne 0 ]
then
  ⟨ make a list of filenames in the intray 9b ⟩
  ⟨ decide whether to renew the stopos-pool 9c ⟩
  ⟨ clean up pool and old.filenames 10a ⟩
  ⟨ clean up proctray 10c ⟩
  ⟨ add new filenames to the pool 11a ⟩
fi
◇
```
Fragment referenced in 28c.
Uses: `pool_empty` 8a.

When we run the job -manager twice per hour, Stopos needs to contain enough filenames to keep Lisa working for the next half hour. Probably Lisa's job-control system does not allow us to run more than 100 jobs at the same time. Typically a job runs seven parallel processes. Each process will probably handle at most one NAF file per minute. That means, that if stopos contains $100 \times 7 \times 30 = 21 \times 10^3$ filenames, Lisa can be kept working for half an hour. Let's round this number to 30000.

First let us see whether we will update the existing pool or purge and renew it. We renew it:

1. When there are no files in the intray, so the pool ought to be empty;
2. When there are no jobs around, so renewing the pool does not interfere with jobs running:
3. When the pool status tells us that the pool is empty.

The following macro sets the first argument variable (`pool-full`) to "1" if the pool does not exist or if it contains less then 30000 filenames. Otherwise, it sets the variable to "0" (true). It sets the second argument variable similar when there no filenames left in the pool.

⟨ *parameters* 8d ⟩ ≡
```
stopos_sufficient_filecount=30000
◇
```
Fragment defined by 3, 7b, 8d, 11b, 14b, 16ac, 17b, 20c.
Fragment referenced in 4.
Defines: `stopos_sufficient_filecount` 9a.

⟨ *is the pool full or empty?* 9a ⟩ ≡

```
@1=1
@2=0
stopos -p $stopospool status >/dev/null
result=$?
if
  [ $result -eq 0 ]
then
  if
    [ $STOPOS_PRESENT0 -gt $stopos_sufficient_filecount ]
  then
    @1=0
  fi
  if
    [ $STOPOS_PRESENT0 -gt 0 ]
  then
    @2=1
  fi
fi
◇
```

Fragment referenced in 8a.
Uses: `stopos` 7a, `stopospool` 7b, `stopos_sufficient_filecount` 8d.

⟨ *make a list of filenames in the intray* 9b ⟩ ≡

```
find $intray -type f -print | sort >infilelist
◇
```

Fragment referenced in 8a.
Defines: `infilelist` 10abc, 11a.
Uses: `intray` 3, `print` 36a.

Note that variable `jobcount` needs to be known before running the following macro. The macro set variable `regen_pool_condition` to `true` (i.e. zero) when the conditions renew the pool are fulfilled.

⟨ *decide whether to renew the stopos-pool* 9c ⟩ ≡

```
cd $root
regen_pool_condition=1
if
  [ $incount -eq 0 ] || [ $jobcount -eq 0 ] || [ $pool_empty -eq 0 ]
then
  regen_pool_condition=0
fi
◇
```

Fragment referenced in 8a.
Defines: `regen_pool_condition` 10a.
Uses: `incount` 6a, `pool_empty` 8a, `root` 3.

When the conditions are fulfilled, make a new pool and empty `old.infileist`. Otherwise, remove from `old.infilelist` the names of files that are no longer present in the intray.

⟨ *clean up pool and old.filenames* 10a ⟩ ≡

```
if
  [ $regen_pool_condition -eq 0 ]
then
  stopos -p $stopospool purge
  stopos -p $stopospool create
  rm -f old.infilelist
  touch old.infilelist
else
    ⟨ clean up old.infilelist 10b ⟩
fi

◇
```

Fragment referenced in 8a.
Uses: `infilelist` 9b, `regen_pool_condition` 9c, `stopos` 7a, `stopospool` 7b.

Update the content of `old.infilelist` so that, as far as we know, it contains only names of files that are still in the pool. Update `infilelist` so that it only contains names of files that reside in the intray but not yet in the pool.

⟨ *clean up old.infilelist* 10b ⟩ ≡

```
comm -12 old.infilelist infilelist >temp.infilelist
cp temp.infilelist old.infilelist
comm -13 old.infilelist infilelist >temp.infilelist
cp temp.infilelist infilelist
◇
```

Fragment referenced in 10a.
Uses: `infilelist` 9b.

Make a list of names of files in the proctray that should be moved to the intray, either because they reside longer in the proctray than the lifetime of jobs or because there are no running jobs. Move the files in the list back to the intray and add the list to `infilelist`. **Note:** that after this `infilelist` is no longer sorted.

⟨ *clean up proctray* 10c ⟩ ≡

```
if
  [ $running_jobs -eq 0 ]
then
  find $proctray -type f -print | sort >oldprocfilelist
else
  find $proctray -type f -cmin +$maxproctime -print | sort  >oldprocfilelist
fi
cat oldprocfilelist | xargs -iaap  bash -c 'movetotray aap $proctray $intray'
cat infilelist oldprocfilelist >temp.infilelist
mv temp.infilelist infilelist
◇
```

Fragment referenced in 8a.
Uses: `infilelist` 9b, `intray` 3, `maxproctime` 11b, `movetotray` 5a, `print` 36a, `running_jobs` 13c.

Add the names of the files in the intray that are not yet in the pool to the pool. Then update `old.infilelist`.

⟨ *add new filenames to the pool* 11a ⟩ ≡
```
stopos -p $stopospool add infilelist
cat infilelist old.infilelist | sort >temp.infilelist
mv temp.infilelist old.infilelist
rm infilelist
◇
```
Fragment referenced in 8a.
Uses: `infilelist` 9b, `stopos` 7a, `stopospool` 7b.

⟨ *parameters* 11b ⟩ ≡
```
maxproctime=30
◇
```
Fragment defined by 3, 7b, 8d, 11b, 14b, 16ac, 17b, 20c.
Fragment referenced in 4.
Defines: `maxproctime` 10c.

2.4.2 Get a filename from the pool

To get a filename from Stopos, perform:

```
stopos -p $stopospool next
```

When this instruction is successfull, it sets variable `STOPOS_RC` to `OK` and puts the filename in variable `STOPOS_VALUE`.

Get next input-file from stopos and put its full path in variable `infile`. If Stopos is empty, put an empty string in `infile`.

It seems that sometimes stopos produces the name of a file that is not present in the intray. In that case, get another filename from Stopos.

⟨ *get next infile from stopos* 11c ⟩ ≡
```
repeat=0
while
  [ $repeat ]
do
  stopos -p $stopospool next
  if
    [ ! "$STOPOS_RC" == "OK" ]
  then
    infile=""
    repeat=1
  else
    infile=$STOPOS_VALUE
    if
      [ -e "$infile" ]
    then
      repeat=1
    fi
  fi
done
◇
```
Fragment referenced in 12a.
Uses: `stopos` 7a, `stopospool` 7b.

2.4.3 Function to get a filename from Stopos

The following function, getfile, reads a file from stopos, puts it in variable `infile` and sets the paths to the outtray, the logtray and the failtray. When the Stopos pool turns out to be empty, the variable is made empty.

⟨ *functions in the jobfile* 12a ⟩ ≡
```
function getfile() {
  infile=""
  outfile=""
  ⟨ get next infile from stopos 11c ⟩
  if
    [ ! "$infile" == "" ]
  then
    ⟨ generate filenames 6l ⟩
  fi
}
```
◇

Fragment defined by 12a, 19a, 20a.
Fragment referenced in 25c.
Defines: `getfile` 18b.
Uses: `outfile` 6l.

2.4.4 Remove a filename from Stopos

⟨ *remove the infile from the stopos pool* 12b ⟩ ≡
```
stopos -p $stopospool remove
```
◇

Fragment referenced in 25a.
Uses: `stopos` 7a, `stopospool` 7b.

3 Jobs

3.1 Manage the jobs

The management script submits jobs when necessary. It needs to do the following:

1. Count the number of submitted and running jobs.
2. Count the number of documents that still have to be processed.
3. Calculate the number of extra jobs that have to be submitted.
4. Submit the extra jobs.

Find out how many submitted jobs there are and how many of them are actually running. Lisa supplies an instruction `showq` that produces a list of running and waiting jobs. However, the list is not always complete. Therefore we need to make job bookkeeping.

File `jobcounter` lists the number of jobs. When extra jobs are submitted, the number is increased. When logfiles are found that jobs produce when they end, the number is decreased.

⟨ *count jobs* 13a ⟩ ≡
```
if
  [ -e jobcounter ]
then
  export jobcount=‘cat jobcounter‘
else
  jobcount=0
fi
◇
```
Fragment defined by 13abc, 14a.
Fragment referenced in 28c.

Count the logfiles that finished jobs produce. Derive the number of jobs that have been finished since last time. Move the logfiles to directory `joblogs`. It is possible that jobs finish and produce logfiles while we are doing all this. Therefore we start to make a list of the logfiles that we will process.

⟨ *count jobs* 13b ⟩ ≡
```
cd $root
ls -1 dutch_pipeline_job.[eo]* >jobloglist
finished_jobs=‘cat jobloglist | grep "\.e" | wc -l‘
mkdir -p joblogs
cat jobloglist | xargs -iaap mv aap joblogs/
if
  [ $finished_jobs -gt $jobcount ]
then
  jobcount=0
else
  jobcount=$((jobcount - $finished_jobs))
fi
◇
```
Fragment defined by 13abc, 14a.
Fragment referenced in 28c.
Uses: `root` 3.

Extract the summaries of the numbers of running jobs and the total number of jobs from the job management system of Lisa.

⟨ *count jobs* 13c ⟩ ≡
```
joblist=‘mktemp -t jobrep.XXXXXX‘
rm -rf $joblist
showq -u $USER | tail -n 1 > $joblist
running_jobs=‘cat $joblist | gawk ’
    { match($0, /Active Jobs:[[:blank:]]*([[:digit:]]+)[[:blank:]]*Idle/, arr)
      print arr[1]
    }’‘
total_jobs_qn=‘cat $joblist | gawk ’
    { match($0, /Total Jobs:[[:blank:]]*([[:digit:]]+)[[:blank:]]*Active/, arr)
      print arr[1]
    }’‘
rm $joblist
◇
```
Fragment defined by 13abc, 14a.
Fragment referenced in 28c.
Defines: `running_jobs` 10c, 29c, `total_jobs_qn` Never used.
Uses: `print` 36a.

If `showq` reports more jobs than `jobcount` lists, something is wrong. The best we can do in that case is to make `jobcount` equal to `running_jobs`. The same repair must be performed when `jobcount` reports that there are jobs around while Sara maintains that this isn't the case.

⟨ *count jobs* 14a ⟩ ≡
```
if
  [ $total_jobs -gt $jobcount ] || [ $total_jobs -eq 0 ]
then
  jobcount=$total_jobs
fi
```
◇

Fragment defined by 13abc, 14a.
Fragment referenced in 28c.

Currently we aim at one job per 100 waiting files.

⟨ *parameters* 14b ⟩ ≡
```
filesperjob=100
```
◇

Fragment defined by 3, 7b, 8d, 11b, 14b, 16ac, 17b, 20c.
Fragment referenced in 4.

Calculate the number of jobs that have to be submitted.

⟨ *determine how many jobs have to be submitted* 14c ⟩ ≡
```
⟨ determine number of jobs that we want to have 14d, ... ⟩
jobs_to_be_submitted=$((jobs_needed - $jobcount))
```
◇

Fragment referenced in 15b.
Uses: `jobs_needed` 15b, `jobs_to_be_submitted` 15b.

Variable `jobs_needed` will contain the number of jobs that we want to have submitted, given the number of unready NAF files.

⟨ *determine number of jobs that we want to have* 14d ⟩ ≡
```
jobs_needed=$((unreadycount / $filesperjob))
if
  [ $unreadycount -gt 0 ] && [ $jobs_needed -eq 0 ]
then
  jobs_needed=1
fi
```
◇

Fragment defined by 14d, 15a.
Fragment referenced in 14c.
Uses: `jobs_needed` 15b, `unreadycount` 6a.

Let us not flood the place with millions of jobs. Set a max of 200 submitted jobs.

⟨ *determine number of jobs that we want to have* 15a ⟩ ≡

```
if
  [ $jobs_needed -gt 200 ]
then
  jobs_needed=200
fi
◇
```

Fragment defined by 14d, 15a.
Fragment referenced in 14c.
Uses: `jobs_needed` 15b.

⟨ *submit jobs when necessary* 15b ⟩ ≡

```
⟨ determine how many jobs have to be submitted 14c ⟩
if
  [ $jobs_to_be_submitted -gt 0 ]
then
   ⟨ submit jobs (15c $jobs_to_be_submitted ) 16b ⟩
   jobcount=$((jobcount + $jobs_to_be_submitted))
fi
echo $jobcount > jobcounter
◇
```

Fragment referenced in 28c.
Uses: `jobs_to_be_submitted` 15b.

3.2 Generate and submit jobs

A job needs a script that tells what to do. The job-script is a Bash script with the recipe to be executed, supplemented with instructions for the job control system of the host. In order to perform the Art of Making Things Unccesessary Complicated, we have a template from which the job-script can be generated with the M4 pre-processor.

Generate job-script template `job.m4` as follows:

1. Open the job-script with the wall-time parameter (the maximum duration that is allowed for the job).
2. Add an instruction to change the M4 “quote” characters.
3. Add the M4 template `dutch_pipeline_job`.

Process the template with `M4`.

⟨ *generate jobscript* 15d ⟩ ≡

```
echo "m4_define(m4_walltime, $walltime)m4_dnl" >job.m4
echo 'm4_changequote(‘<!’"’"’,‘!>’"’"’)m4_dnl' >>job.m4
cat dutch_pipeline_job.m4 >>job.m4
cat job.m4 | m4 -P >dutch_pipeline_job
# rm job.m4
◇
```

Fragment referenced in 16b.
Uses: `walltime` 16a.

A wall-time of 30 minutes seems suitable for the jobs. It is sufficiently large to be productive and it is small enough to be scheduled flexible in the job-system of Lisa.

⟨ *parameters* 16a ⟩ ≡
```
export walltime=30:00
```
◇

Fragment defined by 3, 7b, 8d, 11b, 14b, 16ac, 17b, 20c.
Fragment referenced in 4.
Defines: `walltime` 15d.

Submit the jobscript. The argument is the number of times that the jobscript has to be submitted.

⟨ *submit jobs* 16b ⟩ ≡
```
⟨ generate jobscript 15d ⟩
jobid=‘qsub -t 1-@1 /home/phuijgen/nlpt/Pipeline-NL-Lisa/dutch_pipeline_job‘
```
◇

Fragment referenced in 15b.

4 Logging

There are three kinds of log-files:

1. Every job generates two logfiles in the directory from which it has been submitted (job logs).
2. Every job writes the time that it starts or finishes processing a naf in a *time log*.
3. For every NAF a file is generated in the log directory. This file contains the standard error output of the modules that processed the file.

4.1 Time log

Keep a time-log with which the time needed to annotate a file can be reconstructed.

⟨ *parameters* 16c ⟩ ≡
```
export timelogfile=/home/phuijgen/nlpt/Pipeline-NL-Lisa/data/log/timelog
```
◇

Fragment defined by 3, 7b, 8d, 11b, 14b, 16ac, 17b, 20c.
Fragment referenced in 4.

⟨ *add timelog entry* 16d ⟩ ≡
```
echo ‘date +%s‘: @1 >> $timelogfile
```
◇

Fragment referenced in 16eg, 18b.

⟨ *log that the job starts* 16e ⟩ ≡
⟨ *add timelog entry* (16f `Start job $jobname`) 16d ⟩
◇

Fragment referenced in 25c.

⟨ *log that the job finishes* 16g ⟩ ≡
⟨ *add timelog entry* (16h `Finish job $jobname`) 16d ⟩
◇

Fragment referenced in 25c.

5 Processes

A job runs in computer that is part of the Lisa supercomputer. The computer has a CPU with multiple cores. To use the cores effectively, the job generates parallel processes that do the work. The number of processes to be generated depends on the number of cores and the amount of memory that is available.

5.1 Calculate the number of parallel processes to be launched

The stopos module, that we use to synchronize file management, supplies the instructions `sara-get-num-cores` and `sara-get-mem-size` that return the number of cores resp. the amount of memory of the computer that hosts the job.

Actually we could do with a more accurate estimation of the amount of memory that is available for the processes. Sometimes we need to install Spotlight servers and sometimes we can use external servers. The same goes for the `e-SRL` server. It would be better if we could measure how much memory is actually available.

Note that the stopos module has to be loaded before the following macro can be executed succesfully.

⟨ *determine amount of memory and nodes* 17a ⟩ ≡
```
export ncores=‘sara-get-num-cores‘
#export MEMORY=‘head -n 1 < /proc/meminfo | gawk ’{print $2}’‘
export memory=‘sara-get-mem-size‘
```
◇

Fragment referenced in 18a.
Defines: `memory` 17c, `ncores` 17c.
Uses: `print` 36a.

We want to run as many parallel processes as possible, however we do want to have at least one node per process and at least an amount of 3 GB of memory per process.

⟨ *parameters* 17b ⟩ ≡
```
mem_per_process=3
```
◇

Fragment defined by 3, 7b, 8d, 11b, 14b, 16ac, 17b, 20c.
Fragment referenced in 4.

Calculate the number of processes to be launched and write the result in variable `maxprogs`.

⟨ *determine number of parallel processes* 17c ⟩ ≡
```
export memchunks=$((memory / mem_per_process))
if
  [ $ncores -gt $memchunks ]
then
  maxprocs=$memchunks
else
  maxprocs=ncores
fi
```
◇

Fragment referenced in 18a.
Defines: `maxprocs` 18a.
Uses: `memory` 17a, `ncores` 17a.

5.2 Start parallel processes

After determining how many parallel processes we can run, start processes as Bash subshells. If it turns out that processes have no work to do, they die. In that case, the job should die too. Therefore a processes-counter registers the number of running processes. When this has reduced to zero, the macro expires.

```
⟨ run parallel processes 18a ⟩ ≡
    ⟨ determine amount of memory and nodes 17a ⟩
    ⟨ determine number of parallel processes 17c ⟩
    procnum=0
    ⟨ init processescounter 27b ⟩
    for ((i=1 ; i<=$maxprocs ; i++))
    do
      ( procnum=$i
        ⟨ increment the processes-counter 27c ⟩
        ⟨ perform the processing loop 18b ⟩
        ⟨ decrement the processes-counter, kill if this was the only process 28a ⟩
      )&
    done
    ⟨ wait for working-processes 28b ⟩
    ◇
```

Fragment referenced in 25c.
Defines: `procnum` Never used.
Uses: `maxprocs` 17c.

5.3 Perform the processing loop

In a loop, the process obtains the path to an input NAF and processes it.

```
⟨ perform the processing loop 18b ⟩ ≡
    while
       getfile
       [ ! -z $infile ]
    do
       ⟨ add timelog entry (18c Start $infile ) 16d ⟩
       ⟨ process infile 23a ⟩
       ⟨ add timelog entry (18d Finished $infile with result: $pipelineresult ) 16d ⟩

    done
    ◇
```

Fragment referenced in 18a.
Uses: `pipelineresult` 23a.

6 Servers

Some NLP-modules need to consult a Spotlight-server. If possible, we will use an existing server somewhere on the Internet, but if this is not possible we will have to set up our own Spotlight server.

The eSRL module has been built as a server-client structure, hence we have to set up this server too.

We have the following todo items:

1. Determine the amount of free memory after the servers have been installed in order to calculate the number of parallel processes that we can start.
2. Look whether the eSRL server can be installed externally as well.

6.1 Spotlight server

Some of the pipeline modules need to consult a *Spotlight server* that provides information from DBPedia about named entities. If it is possible, use an external server, otherwise start a server on the host of the job. We need two Spotlight servers, one for English and the other for Dutch. We expect that we can find spotlight servers on host `130.37.53.33`, port 2060 for Dutch and 2020 for English. If it turns out that we cannot access these servers, we have to build Spotlightserver on the local host.

⟨ *functions in the jobfile* 19a ⟩ ≡

```
function check_start_spotlight {
  language=$1
  if
    [ language == "nl" ]
  then
    spotport=2060
  else
    spotport=2020
  fi
  spotlighthost=130.37.53.33
  ⟨ check spotlight on (19b $spotlighthost,19c $spotport ) 20b ⟩
  if
    [ $spotlightrunning -ne 0 ]
  then
    start_spotlight_on_localhost $language $spotport
    spotlighthost="localhost"
    spotlightrunning=0
  fi
  export spotlighthost
  export spotlightrunning
}
```

◇

Fragment defined by 12a, 19a, 20a.
Fragment referenced in 25c.

⟨ *functions in the jobfile* 20a ⟩ ≡

```
function start_spotlight_on_localhost {
   language=$1
   port=$2
   spotlightdirectory=/home/phuijgen/nlp/nlpp/env/spotlight
   spotlightjar=dbpedia-spotlight-0.7-jar-with-dependencies-candidates.jar
   if
     [ "$language" == "nl" ]
   then
     spotresource=$spotlightdirectory"/nl"
   else
     spotresource=$spotlightdirectory"/en_2+2"
   fi
   java -Xmx8g \
        -jar $spotlightdirectory/$spotlightjar \
        $spotresource \
        http://localhost:$port/rest \
   &
}
◇
```

Fragment defined by 12a, 19a, 20a.
Fragment referenced in 25c.

⟨ *check spotlight on* 20b ⟩ ≡

```
exec 6<>/dev/tcp/@1/@2
spotlightrunning=$?
exec 6<&-
exec 6>&-
◇
```

Fragment referenced in 19a.
Uses: spotlightrunning 19a.

7 Apply the pipeline

This section finally deals with the essential purpose of this software: to annotate a document with the modules of the pipeline.

The pipeline is installed in directory `/home/phuijgen/nlpt/nlpp`. For each of the modules there is a script in subdirectory `bin`.

⟨ *parameters* 20c ⟩ ≡

```
export pipelineroot=/home/phuijgen/nlpt/nlpp
export BIND=$pipelineroot/bin
◇
```

Fragment defined by 3, 7b, 8d, 11b, 14b, 16ac, 17b, 20c.
Fragment referenced in 4.

7.1 Language of the document

Our pipeline is currently bi-lingual. Only documents in Dutch or English can be annotated. The language is specified as argument in the `NAF` tag. The pipeline installation contains a Python script that returns the language of the document in the NAF. Put the language in variable `naflang`.

Select the model that the Nerc module has to use, dependent of the language.

⟨ *retrieve the language of the document* 21a ⟩ ≡
```
naflang=‘cat @1 | python /home/phuijgen/nlpt/nlpp/env/bin/langdetect.py‘
export naflang
#
⟨ set nercmodel 21c ⟩
◇
```
Fragment referenced in 23a.
Defines: `naflang` 21c.

By the way, the python script uses Python 2.7, so let us import the corresponding module.

⟨ *load python module* 21b ⟩ ≡
```
module load python/2.7.9
◇
```
Fragment referenced in 25c.
Uses: `module` 7a.

⟨ *set nercmodel* 21c ⟩ ≡
```
if
  [ "$naflang" == "nl" ]
then
  export nercmodel=nl/nl-clusters-conll02.bin
else
  export nercmodel=en/en-newsreader-clusters-3-class-muc7-conll03-ontonotes-4.0.bin
fi
◇
```
Fragment referenced in 21a.
Defines: `nercmodel` Never used.
Uses: `naflang` 21a.

7.2 Apply a module on a NAF file

For each NLP module, there is a script in the `bin` subdirectory of the pipeline-installation. This script reads a NAF file from standard in and produces annotated NAF-encoded document on standard out, if all goes well. The exit-code of the module-script can be used as indication of the success of the annotation.

To prevent that modules operate on the result of failed operation of a a previous module, the exit code will be stored in variable `moduleresult`.

The following function applies a module on the input naf file, but only if variable `moduleresult` is equal to zero. If the annotation fails, the function writes a fail message to standard error and it sets variable `failmodule` to the name of the module that failed. In this way the modules can easily be concatenated to annotate the input document and to stop processing with a clear message when a module goes wrong. The module's output of standard error is concatenated to the logfile that belongs to the input-file. The function has the following arguments:

1. Path of the input NAF.
2. Module script.
3. Path of the output NAF.

⟨ *functions in the pipeline-file* 22a ⟩ ≡

```
function runmodule {
infile=$1
modulecommand=$2
outfile=$3
if
  [ $moduleresult -eq 0 ]
then
  cat $infile | $modulecommand > $outfile 2>>$logfile
  moduleresult=$?
  if
    [ $moduleresult -gt 0 ]
  then
    failmodule=$modulecommand
     echo Failed: module $modulecommand";" result $moduleresult >>$logfile
     echo Failed: module $modulecommand";" result $moduleresult >&2
     echo Failed: module $modulecommand";" result $moduleresult
     cp $outfile out.naf
     exit $moduleresult
  else
     echo Completed: module $modulecommand";" result $moduleresult >>$logfile
     echo Completed: module $modulecommand";" result $moduleresult >&2
     echo Completed: module $modulecommand";" result $moduleresult
  fi
fi
}

export runmodule
```

◇

Fragment defined by 22ab, 24ab.
Fragment never referenced.
Uses: `logfile` 6l, `module` 7a, `moduleresult` 22b, `outfile` 6l.

Initialise `moduleresult` with value 0:

⟨ *functions in the pipeline-file* 22b ⟩ ≡

```
export moduleresult=0
```

◇

Fragment defined by 22ab, 24ab.
Fragment never referenced.
Defines: `moduleresult` 22a, 23a.

7.3 Perform the annotation on an input NAF

When a process has obtained the name of a NAF file to be processed and has generated filenames for the input-, proc-, log-, fail- and output files (section 2.3), it can start to process the file. Note the timeout instruction:

⟨ *process infile* 23a ⟩ ≡
```
export nlppscript=$BIND/nlpp
movetotray $infile $intray $proctray
mkdir -p $outpath
mkdir -p $logpath
export TEMPRES=‘mktemp -t tempout.XXXXXX‘
⟨ retrieve the language of the document (23b $procfile ) 21a ⟩
moduleresult=0
timeout 1500 bash -c "(cat $procfile | $nlppscript > $TEMPRES)"
pipelineresult=$?
⟨ move the processed naf around 25a ⟩
cd $root
rm -f $TEMPRES
◇
```
Fragment referenced in 18b.
Uses: procfile 6l.

We need to set a time-out on processing, otherwise documents that take too much time keep being recycled between the intray and the proctray. The bash timeout function executes the instruction that is given as argument in a subshell. Therefore, execute processing in a separate script, apply_pipeline. This script inherits the exported parameters from the environment from which the timeout instruction has been executed. In other words, it knows about infile, procfile etc.

The script applies the nlpp script on the input-file.

"../apply_pipeline" 23c≡
```
#!/bin/bash
export pipelinescript=/home/phuijgen/nlpt/nlpp/bin/nlpp
outtmp=‘mktemp -t outtmp.XXXXXX‘
cat $procfile | $pipelinescript > $outtmp
result=$?
if
  [ $result -eq 0 ]
then
  mv $outtmp $outfile
  rm $procfile
else
  rm -f $outtmp
  mv $procfile $failfile
fi
exit $result

◇
```
Uses: outfile 6l, procfile 6l.

⟨ *make scripts executable* 23d ⟩ ≡
```
chmod 775 /home/phuijgen/nlpt/Pipeline-NL-Lisa/apply_pipeline
◇
```
Fragment defined by 23d, 29a, 42b.
Fragment referenced in 42c.

⟨ *functions in the pipeline-file* 24a ⟩ ≡

```
function apply_dutch_pipeline {
  runmodule $procfile   $BIND/tok                tok.naf
  runmodule tok.naf     $BIND/mor                mor.naf
  runmodule mor.naf     $BIND/nerc               nerc.naf
  runmodule nerc.naf    $BIND/wsd                wsd.naf
  runmodule wsd.naf     $BIND/ned                ned.naf
  runmodule ned.naf     $BIND/heideltime         times.naf
  runmodule times.naf   $BIND/onto               onto.naf
  runmodule onto.naf    $BIND/srl                srl.naf
  runmodule srl.naf     $BIND/nomevent           nomev.naf
  runmodule nomev.naf   $BIND/srl-dutch-nominals psrl.naf
  runmodule psrl.naf    $BIND/framesrl           fsrl.naf
  runmodule fsrl.naf    $BIND/opinimin           opin.naf
  runmodule opin.naf    $BIND/evcoref            out.naf
}

export apply_dutch_pipeline
```

◇

Fragment defined by 22ab, 24ab.
Fragment never referenced.
Uses: `procfile` 6l.

⟨ *functions in the pipeline-file* 24b ⟩ ≡

```
function apply_english_pipeline {
  runmodule $procfile    $BIND/tok               tok.naf
  runmodule tok.naf      $BIND/topic             top.naf
  runmodule top.naf      $BIND/pos               pos.naf
  runmodule pos.naf      $BIND/constpars         consp.naf
  runmodule consp.naf    $BIND/nerc              nerc.naf
  runmodule nerc.naf     $BIND/coreference-base  coref.naf
  runmodule coref.naf    $BIND/ned               ned.naf
  runmodule ned.naf      $BIND/nedrer            nedr.naf
  runmodule nedr.naf     $BIND/wikify            wikif.naf
  runmodule wikif.naf    $BIND/ukb               ukb.naf
  runmodule ukb.naf      $BIND/ewsd              ewsd.naf
  runmodule ewsd.naf     $BIND/eSRL              esrl.naf
  runmodule esrl.naf     $BIND/FBK-time          time.naf
  runmodule time.naf     $BIND/FBK-temprel       trel.naf
  runmodule trel.naf     $BIND/FBK-causalrel     crel.naf
  runmodule crel.naf     $BIND/evcoref           ecrf.naf
  runmodule ecrf.naf     $BIND/factuality        fact.naf
  runmodule fact.naf     $BIND/opinimin          out.naf
}

export apply_english_pipeline
```

◇

Fragment defined by 22ab, 24ab.
Fragment never referenced.
Uses: `procfile` 6l.

When processing is ready, the NAF's involved must be placed in the correct location. When processing has been successful, the produced NAF, i.e. `out.naf`, must be moved to the outtray and the file in the proctray must be removed. Otherwise, the file in the proctray must be moved to the

failtray. Finally, remove the filename from the stopos pool

⟨ *move the processed naf around* 25a ⟩ ≡
```
if
 [ $pipelineresult -eq 0 ]
then
  mkdir -p $outpath
  mv $TEMPRES $outfile
  rm $procfile
else
  movetotray $procfile $proctray $failtray
fi
⟨ remove the infile from the stopos pool 12b ⟩
```
◇

Fragment referenced in 23a.
Uses: `failtray` 3, `movetotray` 5a, `outfile` 6l, `outpath` 6l, `pipelineresult` 23a, `procfile` 6l.

It is important that the computer uses utf-8 character-encoding.

⟨ *set utf-8* 25b ⟩ ≡
```
export LANG=en_US.utf8
export LANGUAGE=en_US.utf8
export LC_ALL=en_US.utf8
```
◇

Fragment referenced in 25c.

7.4 The jobfile template

Now we know what the job has to do, we can generate the script. It executes the functions `passeer` and `veilig` to ensure that the management script is not

"../dutch_pipeline_job.m4" 25c≡
```
m4_changecom()#!/bin/bash
#PBS -lnodes=1
#PBS -lwalltime=m4_walltime
source /home/phuijgen/nlpt/Pipeline-NL-Lisa/parameters
piddir=‘mktemp -d -t piddir.XXXXXXX‘
( $BIND/start_eSRL $piddir )&
export jobname=$PBS_JOBID
⟨ log that the job starts 16e ⟩
⟨ set utf-8 25b ⟩
⟨ load stopos module 7a ⟩
⟨ load python module 21b ⟩
⟨ functions 5a, ... ⟩
⟨ functions in the jobfile 12a, ... ⟩
check_start_spotlight nl
check_start_spotlight en
echo spotlighthost: $spotlighthost >&2
echo spotlighthost: $spotlighthost
starttime=‘date +%s‘
⟨ run parallel processes 18a ⟩
⟨ log that the job finishes 16g ⟩
exit

```
◇

Uses: `spotlighthost` 19a.

7.5 Synchronisation mechanism

Make a mechanism that ensures that only a single process can execute some functions at a time. Currently we only use this to make sure that only one instance of the management script runs. This is necessary because loading Stopos with a huge amount of filenames takes a lot of time and we don not want that a new instance of the management script interferes with this.

The script `sematree`, obtained from http://www.pixelbeat.org/scripts/sematree/ allows this kind of "mutex" locking. Inside information learns that sematree is available on Lisa (in `/home/phuijgen/usrlocal/bi`
To lock access Sematree places a file in a *lockdir*. The directory where the lockdir resides must be accessable for the management script as well as for the jobs. Its name must be present in variable `workdir`, that must be exported.

⟨ *initialize sematree* 26a ⟩ ≡
```
export workdir=/home/phuijgen/nlpt/Pipeline-NL-Lisa/env
mkdir -p $workdir
```
◇

Fragment referenced in 28c.
Uses: `workdir` 27b.

Now we can implement functions `passeer` (gain exclusive access) and `veilig` (give up access).

⟨ *functions* 26b ⟩ ≡
```
function passeer () {
  local lock=$1
  sematree acquire $lock
}

function runsingle () {
  local lock=$1
  sematree acquire $lock 0 || exit
}


function veilig () {
  local lock=$1
  sematree release $lock
}

```
◇

Fragment defined by 5ab, 26b, 27a.
Fragment referenced in 25c, 28c.
Defines: `passeer` Never used, `veilig` 28c.

Occasionally a process applies the `passeer` function, but is aborted before it could apply the `veilig` function.

⟨ *functions* 27a ⟩ ≡

```
function remove_obsolete_lock {
  local lock=$1
  local max_minutes=$2
  if
    [ "$max_minutes" == "" ]
  then
   local max_minutes=60
  fi
  find $workdir -name $lock -cmin +$max_minutes -print | xargs -iaap rm -rf aap
}
```

◇

Fragment defined by 5ab, 26b, 27a.
Fragment referenced in 25c, 28c.
Uses: `print` 36a, `workdir` 27b.

7.5.1 Count processes in jobs

When a job runs, it start up independent sub-processes that do the work and it may start up servers that perform specific tasks (e.g. a Spotlight server). We want the job to shut down when there is nothing to be done. The "wait" instruction of Bash does not help us, because that instruction waits for the servers that will not stop. Instead we make a construction that counts the number of processes that do the work and activates the exit instruction when there are no more left. We use the capacity of sematree to increment and decrement counters. The process that decrements the counter to zero releases a lock that frees the main process. The working directory of sematree must be local on the node that hosts the job.

⟨ *init processescounter* 27b ⟩ ≡

```
export workdir=‘mktemp -d -t workdir.XXXXXX‘
sematree acquire finishlock
```

◇

Fragment referenced in 18a.
Defines: `finishlock` 28ab, `workdir` 26a, 27a.

⟨ *increment the processes-counter* 27c ⟩ ≡

```
sematree acquire countlock
proccount=‘sematree inc countlock‘
sematree release countlock
```

◇

Fragment referenced in 18a.
Defines: `countlock` 28a.
Uses: `proccount` 6a.

⟨ *decrement the processes-counter, kill if this was the only process* 28a ⟩ ≡

```
sematree acquire countlock
proccount=‘sematree dec countlock‘
sematree release countlock
echo "Process $proccunt stops." >&2
if
  [ $proccount -eq 0 ]
then
  sematree release finishlock
fi
```

◇

Fragment referenced in 18a.
Uses: `countlock` 27c, `finishlock` 27b, `proccount` 6a.

⟨ *wait for working-processes* 28b ⟩ ≡

```
sematree acquire finishlock
sematree release finishlock
echo "No working processes left. Exiting." >&2
```

◇

Fragment referenced in 18a.
Uses: `finishlock` 27b.

7.6 The job management script

7.7 The management script

`"../runit"` 28c≡

```
#!/bin/bash
source /etc/profile
export PATH=/home/phuijgen/usrlocal/bin/:$PATH
source /home/phuijgen/nlpt/Pipeline-NL-Lisa/parameters
cd $root
⟨ initialize sematree 26a ⟩
⟨ get runit options 29b ⟩
⟨ functions 5a, ... ⟩
remove_obsolete_lock runit_runs
runsingle runit_runs
⟨ load stopos module 7a ⟩
⟨ check/create directories 6a, ... ⟩
⟨ count jobs 13a, ... ⟩
⟨ update the stopos pool 8a ⟩
⟨ submit jobs when necessary 15b ⟩
if
  [ $loud ]
then
  ⟨ print summary 29c ⟩
fi
veilig runit_runs
exit
```

◇

Uses: `root` 3, `veilig` 26b.

⟨ *make scripts executable* 29a ⟩ ≡
```
chmod 775 /home/phuijgen/nlpt/Pipeline-NL-Lisa/runit
```
◇

Fragment defined by 23d, 29a, 42b.
Fragment referenced in 42c.

7.8 Print a summary

The `runit` script prints a summary of the number of jobs and the number of files in the trays unless a `-s` (silent) option is given.

Use getopts to unset the `loud` flag if the `-s` option is present.

⟨ *get runit options* 29b ⟩ ≡
```
OPTIND=1
export loud=0
while getopts "s:" opt; do
    case "$opt" in
    s)  loud=
        ;;
    esac
done
shift $((OPTIND-1))
```
◇

Fragment referenced in 28c.

Print the summary:

⟨ *print summary* 29c ⟩ ≡
```
echo in          : $incount
echo proc        : $proccount
echo failed      : $failcount
echo processed   : $((logcount - $failcount))
echo jobs        : $jobcount
echo running     : $running_jobs
echo submitted   : $jobs_to_be_submitted
if
  [ ! "$jobid" == "" ]
then
  echo "job-id     : $jobid"
fi
```
◇

Fragment referenced in 28c.
Uses: `failcount` 6a, `incount` 6a, `jobs_to_be_submitted` 15b, `logcount` 6a, `proccount` 6a, `running_jobs` 13c.

A How to read and translate this document

This document is an example of *literate programming* [1]. It contains the code of all sorts of scripts and programs, combined with explaining texts. In this document the literate programming tool `nuweb` is used, that is currently available from Sourceforge (URL:`nuweb.sourceforge.net`). The advantages of Nuweb are, that it can be used for every programming language and scripting language, that it can contain multiple program sources and that it is very simple.

A.1 Read this document

The document contains *code scraps* that are collected into output files. An output file (e.g. `output.fil`) shows up in the text as follows:

"output.fil" 4a ≡
```
# output.fil
< a macro 4b >
< another macro 4c >
◇
```

The above construction contains text for the file. It is labelled with a code (in this case 4a) The constructions between the < and > brackets are macro's, placeholders for texts that can be found in other places of the document. The test for a macro is found in constructions that look like:

< a macro 4b > ≡
```
This is a scrap of code inside the macro.
It is concatenated with other scraps inside the
macro. The concatenated scraps replace
the invocation of the macro.
```

```
Macro defined by 4b, 87e
Macro referenced in 4a
```

Macro's can be defined on different places. They can contain other macro's.

< a scrap 87e > ≡
```
This is another scrap in the macro. It is
concatenated to the text of scrap 4b.
This scrap contains another macro:
< another macro 45b >
```

```
Macro defined by 4b, 87e
Macro referenced in 4a
```

A.2 Process the document

The raw document is named `a_Pipeline_NL_Lisa.w`. Figure 1 shows pathways to translate it into printable/viewable documents and to extract the program sources. Table 1 lists the tools that are

Tool	Source	Description
gawk	www.gnu.org/software/gawk/	text-processing scripting language
M4	www.gnu.org/software/m4/	Gnu macro processor
nuweb	nuweb.sourceforge.net	Literate programming tool
tex	www.ctan.org	Typesetting system
tex4ht	www.ctan.org	Convert TEX documents into `xml`/`html`

Table 1: *Tools to translate this document into readable code and to extract the program sources*

needed for a translation. Most of the tools (except Nuweb) are available on a well-equipped Linux system.

⟨ *parameters in Makefile* 30 ⟩ ≡
```
NUWEB=../env/bin/nuweb
◇
```
Fragment defined by 30, 32b, 34bc, 36d, 39a, 41d.
Fragment referenced in 31a.
Uses: `nuweb` 38b.

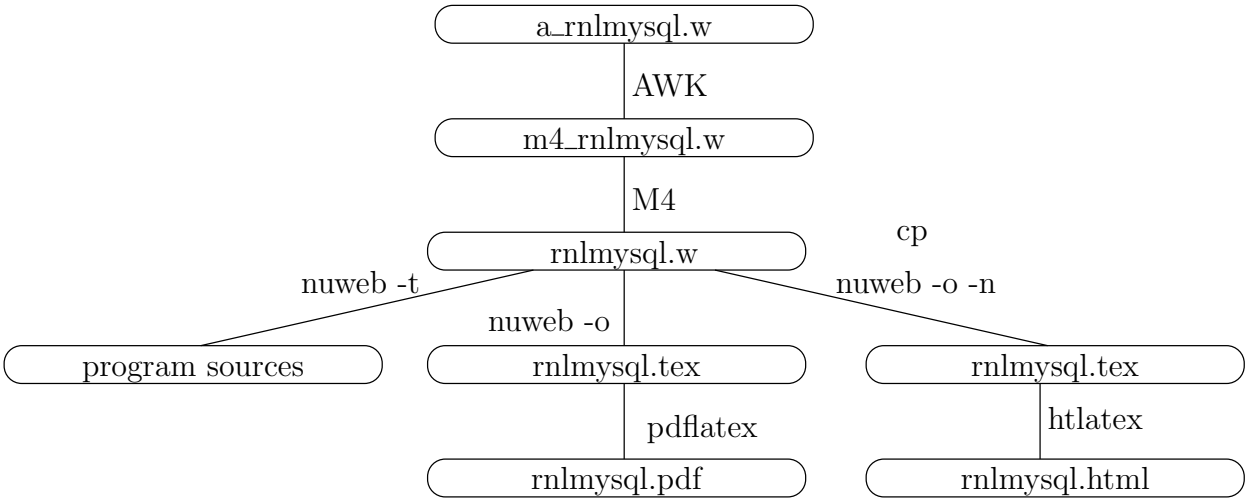

Figure 1: *Translation of the raw code of this document into printable/viewable documents and into program sources. The figure shows the pathways and the main files involved.*

A.3 The Makefile for this project.

This chapter assembles the Makefile for this project.

"Makefile" 31a≡
⟨ *default target* 31b ⟩

⟨ *parameters in Makefile* 30, ... ⟩

⟨ *impliciete make regels* 34a, ... ⟩
⟨ *expliciete make regels* 32c, ... ⟩
⟨ *make targets* 31c, ... ⟩
◇

The default target of make is `all`.

⟨ *default target* 31b ⟩ ≡
```
all : ⟨ all targets 32a ⟩
.PHONY : all
```
◇
Fragment referenced in 31a.
Defines: `all` Never used, `PHONY` 35b.

⟨ *make targets* 31c ⟩ ≡
```
clean:
        ⟨ clean up 32d ⟩
```
◇
Fragment defined by 31c, 36ab, 39e, 42ac.
Fragment referenced in 31a.

One of the targets is certainly the PDF version of this document.

⟨ *all targets* 32a ⟩ ≡
```
Pipeline_NL_Lisa.pdf◇
```
Fragment referenced in 31b.
Uses: pdf 36a.

We use many suffixes that were not known by the C-programmers who constructed the `make` utility. Add these suffixes to the list.

⟨ *parameters in Makefile* 32b ⟩ ≡
```
.SUFFIXES: .pdf .w .tex .html .aux .log .php

◇
```
Fragment defined by 30, 32b, 34bc, 36d, 39a, 41d.
Fragment referenced in 31a.
Defines: SUFFIXES Never used.
Uses: pdf 36a.

A.4 Get Nuweb

An annoying problem is, that this program uses nuweb, a utility that is seldom installed on a computer. Therefore, we are going to install that first if it is not present. Unfortunately, nuweb is hosted on sourceforge and it is difficult to achieve automatic downloading from that repository. Therefore I copied one of the versions on a location from where it can be downloaded with a script.

Put the nuweb binary in the nuweb subdirectory, so that it can be used before the directory-structure has been generated.

⟨ *expliciete make regels* 32c ⟩ ≡
```
nuweb: $(NUWEB)

$(NUWEB): ../nuweb-1.58
        mkdir -p ../env/bin
        cd ../nuweb-1.58 && make nuweb
        cp ../nuweb-1.58/nuweb $(NUWEB)

◇
```
Fragment defined by 32c, 33abc, 35b, 37a, 39bd.
Fragment referenced in 31a.
Uses: nuweb 38b.

⟨ *clean up* 32d ⟩ ≡
```
rm -rf ../nuweb-1.58
◇
```
Fragment referenced in 31c.
Uses: nuweb 38b.

⟨ *expliciete make regels* 33a ⟩ ≡

```
../nuweb-1.58:
        cd .. && wget http://kyoto.let.vu.nl/~huygen/nuweb-1.58.tgz
        cd .. &&  tar -xzf nuweb-1.58.tgz
```

◇

Fragment defined by 32c, 33abc, 35b, 37a, 39bd.
Fragment referenced in 31a.
Uses: `nuweb` 38b.

A.5 Pre-processing

To make usable things from the raw input `a_Pipeline_NL_Lisa.w`, do the following:

1. Process `$` characters.
2. Run the m4 pre-processor.
3. Run nuweb.

This results in a LaTeX file, that can be converted into a PDF or a HTML document, and in the program sources and scripts.

A.5.1 Process 'dollar' characters

Many "intelligent" TeX editors (e.g. the auctex utility of Emacs) handle `$` characters as special, to switch into mathematics mode. This is irritating in program texts, that often contain `$` characters as well. Therefore, we make a stub, that translates the two-character sequence `\$` into the single `$` character.

⟨ *expliciete make regels* 33b ⟩ ≡

```
m4_Pipeline_NL_Lisa.w : a_Pipeline_NL_Lisa.w
        gawk '{if(match($$0, "@%")) {printf("%s", substr($$0,1,RSTART-
1))} else print}' a_Pipeline_NL_Lisa.w \
          | gawk '{gsub(/[\\][\$$]/, "$$");print}'  > m4_Pipeline_NL_Lisa.w
```

◇

Fragment defined by 32c, 33abc, 35b, 37a, 39bd.
Fragment referenced in 31a.
Uses: `print` 36a.

A.5.2 Run the M4 pre-processor

⟨ *expliciete make regels* 33c ⟩ ≡

```
Pipeline_NL_Lisa.w : m4_Pipeline_NL_Lisa.w inst.m4
        m4 -P m4_Pipeline_NL_Lisa.w > Pipeline_NL_Lisa.w
```

◇

Fragment defined by 32c, 33abc, 35b, 37a, 39bd.
Fragment referenced in 31a.

A.6 Typeset this document

Enable the following:

1. Create a PDF document.

2. Print the typeset document.
3. View the typeset document with a viewer.
4. Create a HTMLdocument.

In the three items, a typeset PDF document is required or it is the requirement itself.

⟨ *impliciete make regels* 34a ⟩ ≡
```
%.pdf: %.w
	./w2pdf $<
```
◇
Fragment defined by 34a, 35a, 39c.
Fragment referenced in 31a.
Uses: pdf 36a.

A.6.1 Figures

This document contains figures that have been made by `xfig`. Post-process the figures to enable inclusion in this document.

The list of figures to be included:

⟨ *parameters in Makefile* 34b ⟩ ≡
```
FIGFILES=fileschema directorystructure
```
◇
Fragment defined by 30, 32b, 34bc, 36d, 39a, 41d.
Fragment referenced in 31a.
Defines: FIGFILES 34c.

We use the package `figlatex` to include the pictures. This package expects two files with extensions `.pdftex` and `.pdftex_t` for `pdflatex` and two files with extensions `.pstex` and `.pstex_t` for the `latex`/`dvips` combination. Probably tex4ht uses the latter two formats too.

Make lists of the graphical files that have to be present for latex/pdflatex:

⟨ *parameters in Makefile* 34c ⟩ ≡
```
FIGFILENAMES=$(foreach fil,$(FIGFILES), $(fil).fig)
PDFT_NAMES=$(foreach fil,$(FIGFILES), $(fil).pdftex_t)
PDF_FIG_NAMES=$(foreach fil,$(FIGFILES), $(fil).pdftex)
PST_NAMES=$(foreach fil,$(FIGFILES), $(fil).pstex_t)
PS_FIG_NAMES=$(foreach fil,$(FIGFILES), $(fil).pstex)
```
◇
Fragment defined by 30, 32b, 34bc, 36d, 39a, 41d.
Fragment referenced in 31a.
Defines: FIGFILENAMES Never used, PDFT_NAMES 36b, PDF_FIG_NAMES 36b, PST_NAMES Never used, PS_FIG_NAMES Never used.
Uses: FIGFILES 34b.

Create the graph files with program `fig2dev`:

⟨ *impliciete make regels* 35a ⟩ ≡

```
%.eps: %.fig
        fig2dev -L eps $< > $@

%.pstex: %.fig
        fig2dev -L pstex $< > $@

.PRECIOUS : %.pstex
%.pstex_t: %.fig %.pstex
        fig2dev -L pstex_t -p $*.pstex $< > $@

%.pdftex: %.fig
        fig2dev -L pdftex $< > $@

.PRECIOUS : %.pdftex
%.pdftex_t: %.fig %.pstex
        fig2dev -L pdftex_t -p $*.pdftex $< > $@
```

◇

Fragment defined by 34a, 35a, 39c.
Fragment referenced in 31a.
Defines: `fig2dev` Never used.

A.6.2 Bibliography

To keep this document portable, create a portable bibliography file. It works as follows: This document refers in the |bibliography| statement to the local `bib`-file `Pipeline_NL_Lisa.bib`. To create this file, copy the auxiliary file to another file `auxfil.aux`, but replace the argument of the command `\bibdata{Pipeline_NL_Lisa}` to the names of the bibliography files that contain the actual references (they should exist on the computer on which you try this). This procedure should only be performed on the computer of the author. Therefore, it is dependent of a binary file on his computer.

⟨ *expliciete make regels* 35b ⟩ ≡

```
bibfile : Pipeline_NL_Lisa.aux /home/paul/bin/mkportbib
        /home/paul/bin/mkportbib Pipeline_NL_Lisa litprog

.PHONY : bibfile
```

◇

Fragment defined by 32c, 33abc, 35b, 37a, 39bd.
Fragment referenced in 31a.
Uses: `PHONY` 31b.

A.6.3 Create a printable/viewable document

Make a PDF document for printing and viewing.

⟨ *make targets* 36a ⟩ ≡
```
pdf : Pipeline_NL_Lisa.pdf

print : Pipeline_NL_Lisa.pdf
        lpr Pipeline_NL_Lisa.pdf

view : Pipeline_NL_Lisa.pdf
        evince Pipeline_NL_Lisa.pdf

◇
```
Fragment defined by 31c, 36ab, 39e, 42ac.
Fragment referenced in 31a.
Defines: `pdf` 32ab, 34a, 36b, `print` 6j, 9b, 10c, 13c, 17a, 27a, 33b, `view` Never used.

Create the PDF document. This may involve multiple runs of nuweb, the LaTeX processor and the bibTeX processor, and depends on the state of the `aux` file that the LaTeX processor creates as a by-product. Therefore, this is performed in a separate script, `w2pdf`.

The w2pdf script The three processors nuweb, LaTeX and bibTeX are intertwined. LaTeX and bibTeX create parameters or change the value of parameters, and write them in an auxiliary file. The other processors may need those values to produce the correct output. The LaTeX processor may even need the parameters in a second run. Therefore, consider the creation of the (PDF) document finished when none of the processors causes the auxiliary file to change. This is performed by a shell script `w2pdf`.

⟨ *make targets* 36b ⟩ ≡
```
Pipeline_NL_Lisa.pdf : Pipeline_NL_Lisa.w $(W2PDF)  $(PDF_FIG_NAMES) $(PDFT_NAMES)
        chmod 775 $(W2PDF)
        $(W2PDF) $*

◇
```
Fragment defined by 31c, 36ab, 39e, 42ac.
Fragment referenced in 31a.
Uses: `pdf` 36a, `PDFT_NAMES` 34c, `PDF_FIG_NAMES` 34c.

The following is an ugly fix of an unsolved problem. Currently I develop this thing, while it resides on a remote computer that is connected via the `sshfs` filesystem. On my home computer I cannot run executables on this system, but on my work-computer I can. Therefore, place the following script on a local directory.

⟨ *directories to create* 36c ⟩ ≡
```
../nuweb/bin ◇
```
Fragment referenced in 42a.
Uses: `nuweb` 38b.

⟨ *parameters in Makefile* 36d ⟩ ≡
```
W2PDF=../nuweb/bin/w2pdf
◇
```
Fragment defined by 30, 32b, 34bc, 36d, 39a, 41d.
Fragment referenced in 31a.
Uses: `nuweb` 38b.

⟨ *expliciete make regels* 37a ⟩ ≡
```
$(W2PDF) : Pipeline_NL_Lisa.w $(NUWEB)
        $(NUWEB) Pipeline_NL_Lisa.w
```
◇
Fragment defined by 32c, 33abc, 35b, 37a, 39bd.
Fragment referenced in 31a.

"../nuweb/bin/w2pdf" 37b≡
```
#!/bin/bash
# w2pdf -- compile a nuweb file
# usage: w2pdf [filename]
# 20160701 at 1418h: Generated by nuweb from a_Pipeline_NL_Lisa.w
NUWEB=../env/bin/nuweb
LATEXCOMPILER=pdflatex
⟨ filenames in nuweb compile script 37d ⟩
⟨ compile nuweb 37c ⟩

```
◇
Uses: nuweb 38b.

The script retains a copy of the latest version of the auxiliary file. Then it runs the four processors nuweb, LaTeX, MakeIndex and bibTeX, until they do not change the auxiliary file or the index.

⟨ *compile nuweb* 37c ⟩ ≡
```
NUWEB=/home/phuijgen/nlpt/Pipeline-NL-Lisa/env/bin/nuweb
⟨ run the processors until the aux file remains unchanged 38c ⟩
⟨ remove the copy of the aux file 38a ⟩
```
◇
Fragment referenced in 37b.
Uses: nuweb 38b.

The user provides the name of the nuweb file as argument. Strip the extension (e.g. `.w`) from the filename and create the names of the LaTeX file (ends with `.tex`), the auxiliary file (ends with `.aux`) and the copy of the auxiliary file (add `old.` as a prefix to the auxiliary filename).

⟨ *filenames in nuweb compile script* 37d ⟩ ≡
```
nufil=$1
trunk=${1%%.*}
texfil=${trunk}.tex
auxfil=${trunk}.aux
oldaux=old.${trunk}.aux
indexfil=${trunk}.idx
oldindexfil=old.${trunk}.idx
```
◇
Fragment referenced in 37b.
Defines: auxfil 38c, 40c, 41a, indexfil 38c, 40c, nufil 38b, 40c, 41b, oldaux 38ac, 40c, 41a, oldindexfil 38c, 40c, texfil 38b, 40c, 41b, trunk 38b, 40c, 41bc.

Remove the old copy if it is no longer needed.

⟨ *remove the copy of the aux file* 38a ⟩ ≡
```
rm $oldaux
```
◇

Fragment referenced in 37c, 40b.
Uses: `oldaux` 37d, 40c.

Run the three processors. Do not use the option `-o` (to suppres generation of program sources) for nuweb, because `w2pdf` must be kept up to date as well.

⟨ *run the three processors* 38b ⟩ ≡
```
$NUWEB $nufil
$LATEXCOMPILER $texfil
makeindex $trunk
bibtex $trunk
```
◇

Fragment referenced in 38c.
Defines: `bibtex` 41bc, `makeindex` 41bc, `nuweb` 30, 32cd, 33a, 36cd, 37bc, 39a, 40a.
Uses: `nufil` 37d, 40c, `texfil` 37d, 40c, `trunk` 37d, 40c.

Repeat to copy the auxiliary file and the index file and run the processors until the auxiliary file and the index file are equal to their copies. However, since I have not yet been able to test the `aux` file and the `idx` in the same test statement, currently only the `aux` file is tested.

It turns out, that sometimes a strange loop occurs in which the `aux` file will keep to change. Therefore, with a counter we prevent the loop to occur more than 10 times.

⟨ *run the processors until the aux file remains unchanged* 38c ⟩ ≡
```
LOOPCOUNTER=0
while
  ! cmp -s $auxfil $oldaux
do
  if [ -e $auxfil ]
  then
   cp $auxfil $oldaux
  fi
  if [ -e $indexfil ]
  then
   cp $indexfil $oldindexfil
  fi
  ⟨ run the three processors 38b ⟩
  if [ $LOOPCOUNTER -ge 10 ]
  then
    cp $auxfil $oldaux
  fi;
done
```
◇

Fragment referenced in 37c.
Uses: `auxfil` 37d, 40c, `indexfil` 37d, `oldaux` 37d, 40c, `oldindexfil` 37d.

A.6.4 Create HTML files

HTML is easier to read on-line than a PDF document that was made for printing. We use `tex4ht` to generate HTML code. An advantage of this system is, that we can include figures in the same way as we do for `pdflatex`.

To create a HTML doc, we do the following:

1. Create a directory `../nuweb/html` for the HTML document.
2. Put the nuweb source in it, together with style-files that are needed (see variable `HTMLSOURCE`).
3. Put the script `w2html` in it and make it executable.
4. Execute the script `w2html`.

Make a list of the entities that we mentioned above:

⟨ *parameters in Makefile* 39a ⟩ ≡
```
htmldir=../nuweb/html
htmlsource=Pipeline_NL_Lisa.w Pipeline_NL_Lisa.bib html.sty artikel3.4ht w2html
htmlmaterial=$(foreach fil, $(htmlsource), $(htmldir)/$(fil))
htmltarget=$(htmldir)/Pipeline_NL_Lisa.html
```
◇

Fragment defined by 30, 32b, 34bc, 36d, 39a, 41d.
Fragment referenced in 31a.
Uses: `nuweb` 38b.

Make the directory:

⟨ *expliciete make regels* 39b ⟩ ≡
```
$(htmldir) :
        mkdir -p $(htmldir)
```
◇

Fragment defined by 32c, 33abc, 35b, 37a, 39bd.
Fragment referenced in 31a.

The rule to copy files in it:

⟨ *impliciete make regels* 39c ⟩ ≡
```
$(htmldir)/% : % $(htmldir)
        cp $< $(htmldir)/
```
◇

Fragment defined by 34a, 35a, 39c.
Fragment referenced in 31a.

Do the work:

⟨ *expliciete make regels* 39d ⟩ ≡
```
$(htmltarget) : $(htmlmaterial) $(htmldir)
        cd $(htmldir) && chmod 775 w2html
        cd $(htmldir) && ./w2html nlpp.w
```
◇

Fragment defined by 32c, 33abc, 35b, 37a, 39bd.
Fragment referenced in 31a.

Invoke:

⟨ *make targets* 39e ⟩ ≡
```
htm : $(htmldir) $(htmltarget)
```
◇

Fragment defined by 31c, 36ab, 39e, 42ac.
Fragment referenced in 31a.

Create a script that performs the translation.

"w2html" 40a≡
```
#!/bin/bash
# w2html -- make a html file from a nuweb file
# usage: w2html [filename]
#  [filename]: Name of the nuweb source file.
# 20160701 at 1418h: Generated by nuweb from a_Pipeline_NL_Lisa.w
echo "translate " $1 >w2html.log
NUWEB=/home/phuijgen/nlpt/Pipeline-NL-Lisa/env/bin/nuweb
⟨ filenames in w2html 40c ⟩

⟨ perform the task of w2html 40b ⟩

◇
```
Uses: nuweb 38b.

The script is very much like the w2pdf script, but at this moment I have still difficulties to compile the source smoothly into HTML and that is why I make a separate file and do not recycle parts from the other file. However, the file works similar.

⟨ perform the task of w2html 40b ⟩ ≡
```
⟨ run the html processors until the aux file remains unchanged 41a ⟩
⟨ remove the copy of the aux file 38a ⟩
◇
```
Fragment referenced in 40a.

The user provides the name of the nuweb file as argument. Strip the extension (e.g. .w) from the filename and create the names of the LaTeX file (ends with .tex), the auxiliary file (ends with .aux) and the copy of the auxiliary file (add old. as a prefix to the auxiliary filename).

⟨ filenames in w2html 40c ⟩ ≡
```
nufil=$1
trunk=${1%%.*}
texfil=${trunk}.tex
auxfil=${trunk}.aux
oldaux=old.${trunk}.aux
indexfil=${trunk}.idx
oldindexfil=old.${trunk}.idx
◇
```
Fragment referenced in 40a.
Defines: auxfil 37d, 38c, 41a, nufil 37d, 38b, 41b, oldaux 37d, 38ac, 41a, texfil 37d, 38b, 41b, trunk 37d, 38b, 41bc.
Uses: indexfil 37d, oldindexfil 37d.

⟨ *run the html processors until the aux file remains unchanged* 41a ⟩ ≡

```
while
  ! cmp -s $auxfil $oldaux
do
  if [ -e $auxfil ]
  then
   cp $auxfil $oldaux
  fi
  ⟨ run the html processors 41b ⟩
done
⟨ run tex4ht 41c ⟩

◇
```

Fragment referenced in 40b.
Uses: `auxfil` 37d, 40c, `oldaux` 37d, 40c.

To work for HTML, nuweb *must* be run with the `-n` option, because there are no page numbers.

⟨ *run the html processors* 41b ⟩ ≡

```
$NUWEB -o -n $nufil
latex $texfil
makeindex $trunk
bibtex $trunk
htlatex $trunk
◇
```

Fragment referenced in 41a.
Uses: `bibtex` 38b, `makeindex` 38b, `nufil` 37d, 40c, `texfil` 37d, 40c, `trunk` 37d, 40c.

When the compilation has been satisfied, run makeindex in a special way, run bibtex again (I don't know why this is necessary) and then run htlatex another time.

⟨ *run tex4ht* 41c ⟩ ≡

```
tex '\def\filename{{Pipeline_NL_Lisa}{idx}{4dx}{ind}} \input idxmake.4ht'
makeindex -o $trunk.ind $trunk.4dx
bibtex $trunk
htlatex $trunk
◇
```

Fragment referenced in 41a.
Uses: `bibtex` 38b, `makeindex` 38b, `trunk` 37d, 40c.

A.7 Create the program sources

Run nuweb, but suppress the creation of the LaTeX documentation. Nuweb creates only sources that do not yet exist or that have been modified. Therefore make does not have to check this. However, "make" has to create the directories for the sources if they do not yet exist. So, let's create the directories first.

⟨ *parameters in Makefile* 41d ⟩ ≡

```
MKDIR = mkdir -p

◇
```

Fragment defined by 30, 32b, 34bc, 36d, 39a, 41d.
Fragment referenced in 31a.
Defines: `MKDIR` 42a.

⟨ *make targets* 42a ⟩ ≡
```
DIRS = ⟨ directories to create 36c ⟩

$(DIRS) :
        $(MKDIR) $@
```
◇
Fragment defined by 31c, 36ab, 39e, 42ac.
Fragment referenced in 31a.
Defines: `DIRS` 42c.
Uses: `MKDIR` 41d.

⟨ *make scripts executable* 42b ⟩ ≡
```
chmod -R 775  ../bin/*
chmod -R 775  ../env/bin/*
```
◇
Fragment defined by 23d, 29a, 42b.
Fragment referenced in 42c.

⟨ *make targets* 42c ⟩ ≡
```
source : Pipeline_NL_Lisa.w $(DIRS) $(NUWEB)
        $(NUWEB) Pipeline_NL_Lisa.w
        ⟨ make scripts executable 23d, ... ⟩
```
◇
Fragment defined by 31c, 36ab, 39e, 42ac.
Fragment referenced in 31a.
Uses: `DIRS` 42a.

B References

B.1 Literature

References

[1] Donald E. Knuth. Literate programming. Technical report STAN-CS-83-981, Stanford University, Department of Computer Science, 1983.

C Indexes

C.1 Filenames

C.2 Macro's

⟨ add new filenames to the pool 11a ⟩ Referenced in 8a.
⟨ add timelog entry 16d ⟩ Referenced in 16eg, 18b.
⟨ all targets 32a ⟩ Referenced in 31b.
⟨ check spotlight on 20b ⟩ Referenced in 19a.
⟨ check/create directories 6ak ⟩ Referenced in 28c.
⟨ clean up 32d ⟩ Referenced in 31c.
⟨ clean up old.infilelist 10b ⟩ Referenced in 10a.
⟨ clean up pool and old.filenames 10a ⟩ Referenced in 8a.
⟨ clean up proctray 10c ⟩ Referenced in 8a.
⟨ compile nuweb 37c ⟩ Referenced in 37b.
⟨ count files in tray 6j ⟩ Referenced in 6a.
⟨ count jobs 13abc, 14a ⟩ Referenced in 28c.
⟨ decide whether to renew the stopos-pool 9c ⟩ Referenced in 8a.
⟨ decrement the processes-counter, kill if this was the only process 28a ⟩ Referenced in 18a.
⟨ default target 31b ⟩ Referenced in 31a.
⟨ determine amount of memory and nodes 17a ⟩ Referenced in 18a.
⟨ determine how many jobs have to be submitted 14c ⟩ Referenced in 15b.
⟨ determine number of jobs that we want to have 14d, 15a ⟩ Referenced in 14c.
⟨ determine number of parallel processes 17c ⟩ Referenced in 18a.
⟨ directories to create 36c ⟩ Referenced in 42a.
⟨ expliciete make regels 32c, 33abc, 35b, 37a, 39bd ⟩ Referenced in 31a.
⟨ filenames in nuweb compile script 37d ⟩ Referenced in 37b.
⟨ filenames in w2html 40c ⟩ Referenced in 40a.
⟨ functions 5ab, 26b, 27a ⟩ Referenced in 25c, 28c.
⟨ functions in the jobfile 12a, 19a, 20a ⟩ Referenced in 25c.
⟨ functions in the pipeline-file 22ab, 24ab ⟩ Not referenced.
⟨ generate filenames 6l ⟩ Referenced in 12a.
⟨ generate jobscript 15d ⟩ Referenced in 16b.
⟨ get next infile from stopos 11c ⟩ Referenced in 12a.
⟨ get runit options 29b ⟩ Referenced in 28c.
⟨ impliciete make regels 34a, 35a, 39c ⟩ Referenced in 31a.
⟨ increment the processes-counter 27c ⟩ Referenced in 18a.
⟨ init processescounter 27b ⟩ Referenced in 18a.
⟨ initialize sematree 26a ⟩ Referenced in 28c.
⟨ is the pool full or empty? 9a ⟩ Referenced in 8a.
⟨ load python module 21b ⟩ Referenced in 25c.
⟨ load stopos module 7a ⟩ Referenced in 25c, 28c.
⟨ log that the job finishes 16g ⟩ Referenced in 25c.
⟨ log that the job starts 16e ⟩ Referenced in 25c.
⟨ make a list of filenames in the intray 9b ⟩ Referenced in 8a.
⟨ make scripts executable 23d, 29a, 42b ⟩ Referenced in 42c.
⟨ make targets 31c, 36ab, 39e, 42ac ⟩ Referenced in 31a.
⟨ move the processed naf around 25a ⟩ Referenced in 23a.
⟨ parameters 3, 7b, 8d, 11b, 14b, 16ac, 17b, 20c ⟩ Referenced in 4.
⟨ parameters in Makefile 30, 32b, 34bc, 36d, 39a, 41d ⟩ Referenced in 31a.
⟨ perform the processing loop 18b ⟩ Referenced in 18a.
⟨ perform the task of w2html 40b ⟩ Referenced in 40a.
⟨ print summary 29c ⟩ Referenced in 28c.
⟨ process infile 23a ⟩ Referenced in 18b.
⟨ remove the copy of the aux file 38a ⟩ Referenced in 37c, 40b.
⟨ remove the infile from the stopos pool 12b ⟩ Referenced in 25a.
⟨ retrieve the language of the document 21a ⟩ Referenced in 23a.
⟨ run parallel processes 18a ⟩ Referenced in 25c.
⟨ run tex4ht 41c ⟩ Referenced in 41a.
⟨ run the html processors 41b ⟩ Referenced in 41a.
⟨ run the html processors until the aux file remains unchanged 41a ⟩ Referenced in 40b.
⟨ run the processors until the aux file remains unchanged 38c ⟩ Referenced in 37c.

⟨ run the three processors 38b ⟩ Referenced in 38c.
⟨ set nercmodel 21c ⟩ Referenced in 21a.
⟨ set utf-8 25b ⟩ Referenced in 25c.
⟨ submit jobs 16b ⟩ Referenced in 15b.
⟨ submit jobs when necessary 15b ⟩ Referenced in 28c.
⟨ update the stopos pool 8a ⟩ Referenced in 28c.
⟨ wait for working-processes 28b ⟩ Referenced in 18a.

C.3 Variables