# Standardised Dutch NLP pipeline

Paul Huygen <paul.huygen@huygen.nl>

3rd December 2015
08:54 h.

**Abstract**

This is a description and documentation of a system that uses SurfSara's supercomputer Lisa to perform large-scale linguistic annotation of dutch documents with the "Newsreader pipeline".

## Contents

# 1 Introduction

This document describes a system for large-scale linguistic annotation of Dutch 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) who have access to that computer.

The annotation of the documents will be performed by a "pipeline" that has been set up in the Newsreader-project [1].

## 1.1 How to use it

Quick user instruction:

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. Create a subdirectory `in` and fill it with (a directoy-structure containing) raw NAF's that have to be annotated.
5. Run script `runit`.
6. Wait until it has finished.

The following is a demo script that performs the installation and annotates a set of texts:

```
"../demoscript" 2≡
      #!/bin/bash
      gitrepo=https://github.com/PaulHuygen/Pipeline-NL-Lisa.git
      xampledir=/home/phuijgen/nlp/data/examplesample/
      #
      git clone $gitrepo
      cd Pipeline_NL_Lisa
      mkdir -p data/in
      mkdir -p data/out
      cp $xampledir/*.naf data/in/
      ./runit
      ◇
```

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

# 2 Elements of the job

## 2.1 How it works

The user stores a directory-tree that contains “raw” NAF files in an “intray” and then starts a management script. The management script generates a list of the paths to the naf-files in the intray and stores this in a “Stopos pool” (section 2.4.2). “Stopos” enables parallel running jobs to get the filenames and precludes that two or more parallel processes obtain the same filename.

The management script submits a number of jobs to the queue of the supercomputer.

Eventually the jobs start on individual nodes, They are allowed to run for a certain duration, the “wall time”, after which they are aborted. Each job starts a number of parallel processes. Each process is a cycle of 1) obtain a filename from stopos; 3) annotate the file; 3) store the resulting NAF in the outtray and remove the input-file from the .; 4) remove the filename from the stopos pool.

If a cycle has been completed, the result is:

1. The number of files in the Stopos pool is reduced by one.
2. The number of files in the intray is reduced by one.
3. Either the failtray or the outtray contains a file with the same name as the file that has been removed from the intray.
4. There are entries in log-files

A “todo” item is, to manage files that fail to be annotated. Currently this results in an unusable file in the outtray.

If the cycle could not be completed, the result is:

1. The Stopos pool contains a file-name that cannot be accessed.
2. The intray contains a file that will not be processed using the current pool.

The management script has to be run periodically in order to regenerate the pool and to submit extra jobs to process the remaining files.

Define parameters for the items that have been introduced in this section:

⟨*parameters* 3⟩ ≡

```
export walltime=3:00
export root=/home/phuijgen/nlp/Pipeline-NL-Lisa
export intray=/home/phuijgen/nlp/Pipeline-NL-Lisa/data/in
export outtray=/home/phuijgen/nlp/Pipeline-NL-Lisa/data/out
export failtray=/home/phuijgen/nlp/Pipeline-NL-Lisa/data/fail
export logtray=/home/phuijgen/nlp/Pipeline-NL-Lisa/data/log
◇
```

Fragment defined by 3, 4b, 6b, 8a, 10c, 11f, 12c, 18a, 21d.
Fragment referenced in 4a.
Defines: `failtray` Never used, `intray` 5b, 6a, 7abc, 10b, 21a, `logtray` 5b, 6a, `outtray` 5b, 6a, 21a, `root` Never used, `walltime` 22c.

## 2.2 Still to be done

1. Handle log files from the job system.
2. Recognize when annotation fails.

## 2.3 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" 4a≡

```
⟨ parameters 3, . . . ⟩
◇
```

## 2.4 Moving NAF-files around

A job is a Bash script that finds raw NAF files in the intray, feeds the files through an NLP pipeline and stores the result as NAF file in the outtray. A complication is, that a job runs until it's "wall-time" has been expired, after which the operation system aborts the job. The input files that the job was annotating at that moment will not be completed, and stopos will not pass these files to other jobs. To solve this problem, before starting to annotate, the job moves the inputfile to a "proc" directory. The management script can move these files back to the input tray when it finds out that no job is processing them.

⟨ *parameters* 4b ⟩ ≡

```
export proctray=/home/phuijgen/nlp/Pipeline-NL-Lisa/data/proc
◇
```

Fragment defined by 3, 4b, 6b, 8a, 10c, 11f, 12c, 18a, 21d.
Fragment referenced in 4a.
Defines: `proctray` 5b, 6a, 7bcd, 10b, 21a.

In the pool the input nafs are stored by their full path. The following code scraps copy or move a file that is presented with it's full path from one tray to another tray. Arguments:

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

⟨ *copy file* 4c ⟩ ≡

```
cp @1 $@3/${@1##$@2}◇
```

Fragment never referenced.

⟨ *move file* 4d ⟩ ≡

```
mv @1 $@3/${@1##$@2}◇
```

Fragment never referenced.

⟨ *functions* 4e ⟩ ≡

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

Fragment defined by 4e, 5a, 7c, 17b, 18bc.
Fragment referenced in 12b, 21a.
Defines: `movetotray` 7bc, 10b.

⟨*functions* 5a⟩ ≡

```
function copytotray () {
local file=$1
local fromtray=$2
local totray=$3
local frompath=${file%/*}
local topath=$totray${frompath##$fromtray}
mv $file $totray${file##fromtray}
}
◇
```

Fragment defined by 4e, 5a, 7c, 17b, 18bc.
Fragment referenced in 12b, 21a.
Defines: `copytotray` Never used.

To enable this moving-around of NAF files, a management script has to perform the following:

1. Check whether there are raw NAF's to be processed.
2. Generate the output-tray to store the processed NAF's
3. Generate a Stopos pool with a list of the filenames of the NAF files or update an existing Stopos pool.

A job performs the following:

1. Obtain the path to a raw naf in the intray.
2. Write a processed naf in a directory-tree on the outtray
3. Move a failed inputfile to the fail-tree

Generate the directories to store the files when they are not yet there.

### 2.4.1 Look whether there are input-files

When the management script starts, it checks whether there is actually something to do.

⟨*check/create directories* 5b⟩ ≡

```
infilesexist=1
if
  [ ! -d "$intray" ]
then
  echo "No input-files."
  echo "Create $intray and fill it with raw NAF's."
  veilig
  exit 4
fi
mkdir -p $outtray
mkdir -p $logtray
mkdir -p $proctray
if
  [ ! "$(ls -A $intray)" ] &&  [ ! "$(ls -A $proctray)" ]
then
  echo "Finished processing"
  veilig
  exit
fi
◇
```

Fragment referenced in 21a.
Defines: `infilesexist` Never used.
Uses: `intray` 3, `logtray` 3, `outtray` 3, `proctray` 4b, `veilig` 18bc.

In the next section we will see that Stopos stores the full paths to raw NAF's. When variable `infile` contains the full path to a raw NAF, the following code derives the full path to the annotated NAF that will be created in the outtray:

⟨ *generate filenames* 6a ⟩ ≡

```
filtrunk=${infile##$intray/}
outfile=$outtray/${filtrunk}
logfile=$logtray/${filtrunk}
procfile=$proctray/${filtrunk}
outpath=${outfile%/*}
procpath=${procfile%/*}
logpath=${logfile%/*}
◇
```

Fragment referenced in 9a.
Defines: `filtrunk` Never used, `logfile` Never used, `logpath` Never used, `outfile` 9a, 10b, `outpath` 10b, `procfile` 7c, 10b, `procpath` Never used.
Uses: `intray` 3, `logtray` 3, `outtray` 3, `proctray` 4b.

### 2.4.2 Stopos: file management

Stopos stores a set of parameters (in our case the full paths to NAF files that have to be processed) in a named "pool". A process in a job can read a parameter value from the pool and the Stopos system makes sure that from that moment no other process is able to obtain that parameter value. When the job has finished processing the parameter value, it removes the parameter value from the pool.

Set the name of the Stopos pool:

⟨ *parameters* 6b ⟩ ≡

```
export stopospool=dppool
◇
```

Fragment defined by 3, 4b, 6b, 8a, 10c, 11f, 12c, 18a, 21d.
Fragment referenced in 4a.
Defines: `stopospool` 7ad, 8bc, 10b.

Load the stopos module in a script:

⟨ *load stopos module* 6c ⟩ ≡

```
module load stopos
◇
```

Fragment referenced in 12b, 21a.

### 2.4.3 Generate a Stopos pool

When the script is started for the first time, hopefully raw NAF files are present in the intray, but there are no submitted jobs. When there are no jobs, generate a new Stopos pool. Otherwise, there ought to be a pool. To update the pool, restore files that resided for longer time in the proctray into the intray and re-introduce them in the pool.

⟨ *set up new stopos pool* 7a ⟩ ≡

```
⟨ move all procfiles to intray 7b ⟩
find $intray -type f -print >filelist
stopos -p $stopospool purge
stopos -p $stopospool create
stopos -p $stopospool add filelist
stopos -p $stopospool status
◇
```

Fragment referenced in 21a.
Uses: `intray` 3, `print` 28b, `stopospool` 6b.

⟨ *move all procfiles to intray* 7b ⟩ ≡

```
find $proctray -type f -print | xargs -iaap movetotray aap $proctray $intray
◇
```

Fragment referenced in 7a.
Uses: `intray` 3, `movetotray` 4e, `print` 28b, `proctray` 4b.

Move files that reside longer than `maxproctime` minutes back to the intray. This works as follows:

1. function `restoreprocfile` moves a file back to the intray and adds the path in the intray to a list in file `restorefiles`.
2. The Unix function `find` the old procfiles to function `restoreprocfile`.
3. When the old procfiles have been collected, the filenames in `restorefiles` are passed to Stopos.

⟨ *functions* 7c ⟩ ≡

```
function restoreprocfile {
  procf=$1
  filelist=$2
  inf=$intray/${procfile##$proctray}
  echo $inf >>$filelist
  movetotray $procf $proctray $intray
}
◇
```

Fragment defined by 4e, 5a, 7c, 17b, 18bc.
Fragment referenced in 12b, 21a.
Defines: `restoreprocfile` 7d.
Uses: `intray` 3, `movetotray` 4e, `procfile` 6a, `proctray` 4b.

⟨ *restore old procfiles* 7d ⟩ ≡

```
restorefilelist=‘mktemp -t restore.XXXXXX‘
find $proctray -type f -ctime +$maxproctime -print | \
   xargs -iaap restoreprocfile aap $restorefilelist
stopos -p $stopospool add $restorefilelist
rm $restorefilelist
◇
```

Fragment referenced in 21a.
Uses: `maxproctime` 8a, `print` 28b, `proctray` 4b, `restoreprocfile` 7c, `stopospool` 6b.

⟨ *parameters* 8a ⟩ ≡

```
maxproctime=15
◇
```

Fragment defined by 3, 4b, 6b, 8a, 10c, 11f, 12c, 18a, 21d.
Fragment referenced in 4a.
Defines: `maxproctime` 7d.

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, try to recover old procfiles and try again. If Stopos is still empty, undefine `infile`.

⟨ *get next infile from stopos* 8b ⟩ ≡

```
stopos -p $stopospool next
if
  [ "$STOPOS_RC" == "OK" ]
then
   infile=$STOPOS_VALUE
else
  infile=""
fi
◇
```

Fragment referenced in 9a.
Uses: `stopospool` 6b.

### 2.4.4 Get Stopos status

Find out whether the stopos pool exists and create it if that is not the case.

Find out how many filenames are still present in the Stopos pool. Store the number of input-files that have not yet been given to a processing job in variable `untouched_files` and the number of files that have been given to a processing job but have not yet been finished in variable `busy_files`.

⟨ *get stopos status* 8c ⟩ ≡

```
stopos pools
if [ -z "‘echo $STOPOS_VALUE | grep $stopospool ‘" ]
then
   stopos -p $stopospool create
fi
stopos -p $stopospool status
untouched_files=$STOPOS_PRESENT0
busy_files=$STOPOS_PRESENT
◇
```

Fragment referenced in 21a.
Uses: `stopospool` 6b.

### 2.4.5 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, variable is made empty.

⟨ *function getfile* 9a ⟩ ≡

```
function getfile() {
  infile=""
  outfile=""
  ⟨ get next infile from stopos 8b ⟩
  if
    [ ! "$infile" == "" ]
  then
    ⟨ generate filenames 6a ⟩
  fi
}

◇
```

Fragment referenced in 12b.
Uses: `outfile` 6a.

## 2.5 The pipeline

The raw NAF's will be processed with the Dutch Newsreader Pipeline. It has been installed on the account **`phuijgen`** on Lisa. The installation has been performed using the Github repository .

⟨ *directories of the pipeline* 9b ⟩ ≡

```
export piperoot=/home/phuijgen/nlp/nlpp
export pipebindir=/home/phuijgen/nlp/nlpp/bin
◇
```

Fragment referenced in 9c.

The following script processes a raw NAF from standard in and produces the result on standard out.:

"../pipenl" 9c≡

```
#!/bin/bash
source /home/phuijgen/nlp/Pipeline-NL-Lisa/parameters
⟨ directories of the pipeline 9b ⟩
⟨ set utf-8 10a ⟩
OLDD=‘pwd‘
TEMPDIR=‘mktemp -t -d ontemp.XXXXXX‘
cd $TEMPDIR
cat             | $pipebindir/tok
rm -rf $TEMPDIR
◇
```

⟨ *make scripts executable* 9d ⟩ ≡

```
chmod 775 /home/phuijgen/nlp/Pipeline-NL-Lisa/pipenl
◇
```

Fragment defined by 9d, 21b, 34c.
Fragment referenced in 34d.

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

⟨ *set utf-8* 10a ⟩ ≡

```
export LANG=en_US.utf8
export LANGUAGE=en_US.utf8
export LC_ALL=en_US.utf8
◇
```

Fragment referenced in 9c.

Actually, we do not yet handle failed files separately.

⟨ *process infile* 10b ⟩ ≡

```
movetotray $infile $intray $proctray
mkdir -p $outpath
cat $procfile | /home/phuijgen/nlp/Pipeline-NL-Lisa/pipenl >$outfile
rm $procfile
stopos -p $stopospool remove
◇
```

Fragment referenced in 11b.
Uses: `intray` 3, `movetotray` 4e, `outfile` 6a, `outpath` 6a, `procfile` 6a, `proctray` 4b, `stopospool` 6b.

## 2.6 Time log

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

⟨ *parameters* 10c ⟩ ≡

```
export timelogfile=/home/phuijgen/nlp/Pipeline-NL-Lisa/data/log/timelog
◇
```

Fragment defined by 3, 4b, 6b, 8a, 10c, 11f, 12c, 18a, 21d.
Fragment referenced in 4a.

⟨ *add timelog entry* 10d ⟩ ≡

```
echo ‘date +%s‘: @1 >> $timelogfile
◇
```

Fragment referenced in 11b.

## 2.7 General log mechanism

Write to a log file if logging is set to true.

⟨ *init logfile* 10e ⟩ ≡

```
LOGGING=true
LOGFIL=/home/phuijgen/nlp/Pipeline-NL-Lisa/data/log/log
PROGNAM=@1
◇
```

Fragment referenced in 21a.
Defines: `LOGFIL` 11a, `LOGGING` 11a.

⟨ *write log* 11a ⟩ ≡

```
if LOGGING=true
then
  echo ‘date‘";" $PROGNAM":" @1 >>$LOGFIL
fi
◇
```

Fragment referenced in 13ac, 20a.
Uses: `LOGFIL` 10e, `LOGGING` 10e.

## 2.8 Parallel processes

When a job runs, it determines how many resources it has (CPU nodes, memory) and from that it deterines how many parallel processed it can start up.

⟨ *start parallel processes* 11b ⟩ ≡

```
⟨ determine amount of memory and nodes 11e ⟩
⟨ determine number of parallel processes 12a ⟩
procnum=0
for ((i=1 ; i<=$maxprocs ; i++))
do
  ( procnum=$i
    while
       getfile
       [ ! -z $infile ]
    do
       ⟨ add timelog entry (11c Start $infile ) 10d ⟩
       ⟨ process infile 10b ⟩
       ⟨ add timelog entry (11d Finished $infile ) 10d ⟩

    done
  )&
done

◇
```

Fragment referenced in 12b.

⟨ *determine amount of memory and nodes* 11e ⟩ ≡

```
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 11b.
Uses: `print` 28b.

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 `memchunk` GB of memory per process.

⟨ *parameters* 11f ⟩ ≡

```
mem_per_process=5
◇
```

Fragment defined by 3, 4b, 6b, 8a, 10c, 11f, 12c, 18a, 21d.
Fragment referenced in 4a.

⟨ *determine number of parallel processes* 12a ⟩ ≡

```
export memchunks=$((memory / mem_per_process))
if
  [ $ncores -gt $memchunks ]
then
  maxprocs=$memchunks
else
  maxprocs=ncores
fi
◇
```

Fragment referenced in 11b.

### 2.9 The job

"../dutch_pipeline_job.m4" 12b≡

```
m4_changecom
#!/bin/bash
#PBS -lnodes=1
#PBS -lwalltime=m4_walltime
source /home/phuijgen/nlp/Pipeline-NL-Lisa/parameters
⟨ functions 4e, ... ⟩

⟨ function getfile 9a ⟩
⟨ load stopos module 6c ⟩
starttime=‘date +%s‘
⟨ start parallel processes 11b ⟩
wait
exit

◇
```

### 2.10 Manage the jobs

When we have received files to be parsed we have to submit the proper amount of jobs. To determine whether new jobs have to be submitted we have to know the number of waiting and running jobs. Unfortunately it is too costly to often request a list of running jobs. Therefore we will make a bookkeeping. File `/home/phuijgen/nlp/Pipeline-NL-Lisa/.jobcount` contains a list of the running and waiting jobs.

⟨ *parameters* 12c ⟩ ≡

```
JOBCOUNTFILE=/home/phuijgen/nlp/Pipeline-NL-Lisa/.jobcount
◇
```

Fragment defined by 3, 4b, 6b, 8a, 10c, 11f, 12c, 18a, 21d.
Fragment referenced in 4a.
Defines: `JOBCOUNTFILE` 13af, 14b, 15b, 16a.

It is updated as follows:

- When a job is submitted, a line containing the job-id, the word “wait” and a timestamp is added to the file.
- A job that starts, replaces in the line with its job-id the word “waiting” by running and replaces the timestamp.
- A job that ends regularly removes the line with its job-id.

- A job that ends leaves a log message. The filename consists of a concatenation of the jobname, a dot, the character “o” and the job-id. At a regular basis the existence of such files is checked and `$JOBCOUNTFILE` updated.

Submit a job and write a line in the jobcountfile. The line consists of the jobnumber, the word “wait” and the timestamp in universal seconds.

⟨ *submit a job* 13a ⟩ ≡

```
qsub /home/phuijgen/nlp/Pipeline-NL-Lisa/dutch_pipeline_job | \
 gawk -F"." -v tst=‘date +%s‘  ’{print $1 " wait " tst}’ \
 >> $JOBCOUNTFILE
⟨ write log (13b Updated jobcountfile ) 11a ⟩
◇
```

Fragment referenced in 22d.

When a job starts, it performs some bookkeeping. It finds out its own job number and changes `wait` into `run` in the bookeepfile.

⟨ *perform jobfile-bookkeeping* 13c ⟩ ≡

```
⟨ find out the job number 13e ⟩
prognam=dutch_pipeline_job$JOBNUM
⟨ write log (13d start ) 11a ⟩
⟨ change “wait” to “run” in jobcountfile 13f ⟩
◇
```

Fragment never referenced.

The job ID begins with the number, e.g. `6670732.batch1.irc.sara.nl`.

⟨ *find out the job number* 13e ⟩ ≡

```
JOBNUM=${PBS_JOBID%%.*}
◇
```

Fragment referenced in 13c.

⟨ *change “wait” to “run” in jobcountfile* 13f ⟩ ≡

```
if [ -e $JOBCOUNTFILE ]
then
  passeer
  mv $JOBCOUNTFILE $tmpfil
  gawk -v jid=$JOBNUM -v stmp=‘date +%s‘ \
    ’⟨ awk script to change status of job in joblist 14a ⟩’ \
    $tmpfil >$JOBCOUNTFILE
  veilig
  rm -rf $tmpfil
fi
◇
```

Fragment referenced in 13c.
Uses: `JOBCOUNTFILE` 12c, `passeer` 18bc, `veilig` 18bc.

⟨ *awk script to change status of job in joblist* 14a ⟩ ≡

```
BEGIN {WRIT="N"};
{ if(match($0,"^"jid)>0) {
     print jid " run  " stmp;
     WRIT="Y";
  } else {print}
};
END {
  if(WRIT=="N") print jid " run  " stmp;
}◇
```

Fragment referenced in 13f.
Uses: `print` 28b.

When a job ends, it removes the line:

⟨ *remove the job from the counter* 14b ⟩ ≡

```
passeer
mv $JOBCOUNTFILE $tmpfil
gawk -v jid=$JOBNUM  '$1 !~ "^"jid {print}' $tmpfil >$JOBCOUNTFILE
veilig
rm -rf $tmpfil
◇
```

Fragment never referenced.
Uses: `JOBCOUNTFILE` 12c, `passeer` 18bc, `print` 28b, `veilig` 18bc.

Periodically check whether jobs have been killed before completion and have thus not been able to remove their line in the jobcountfile. To do this, write the jobnumbers in a temporary file and then check the jobcounter file in one blow, to prevent frequent locks.

⟨ *do brief check of expired jobs* 14c ⟩ ≡

```
obsfil=`mktemp --tmpdir obs.XXXXXXX`
rm -rf $obsfil
⟨ make a list of jobs that produced logfiles (14d $obsfil ) 15a ⟩
⟨ compare the logfile list with the jobcounter list (14e $obsfil ) 15b ⟩
rm -rf $obsfil
◇
```

Fragment referenced in 14f.

⟨ *do the frequent tasks* 14f ⟩ ≡

```
⟨ do brief check of expired jobs 14c ⟩
◇
```

Fragment never referenced.

When a job has ended, a logfile, and sometimes an error-file, is produced. The name of the logfile is a concatenation of the jobname, a dot, the character `o` and the jobnumber. The error-file has a similar name, but the character `o` is replaced by `e`. Generate a sorted list of the jobnumbers and remove the logfiles and error-files:

⟨ *make a list of jobs that produced logfiles* 15a ⟩ ≡

```
for file in dutch_pipeline_job.o*
do
  JOBNUM=${file##dutch_pipeline_job.o}
  echo ${file##dutch_pipeline_job.o} >>$tmpfil
  rm -rf dutch_pipeline_job.[eo]$JOBNUM
done
sort < $tmpfil >@1
rm -rf $tmpfil
◇
```

Fragment referenced in 14c.

Remove the jobs in the list from the counter file if they occur there.

⟨ *compare the logfile list with the jobcounter list* 15b ⟩ ≡

```
if [ -e $JOBCOUNTFILE ]
then
  passeer
  sort < $JOBCOUNTFILE >$tmpfil
  gawk -v obsfil=@1 ’
    BEGIN {getline obs < obsfil}
    { while((obs<$1) && ((getline obs < obsfil) >0)){}
      if(obs==$1) next;
      print
    }
  ’ $tmpfil >$JOBCOUNTFILE
  veilig
fi
rm -rf $tmpfil
◇
```

Fragment referenced in 14c.
Uses: `JOBCOUNTFILE` 12c, `passeer` 18bc, `print` 28b, `veilig` 18bc.

From time to time, check whether the jobs-bookkeeping is still correct. To this end, request a list of jobs from the operating system.

⟨ *verify jobs-bookkeeping* 15c ⟩ ≡

```
actjobs=‘mktemp --tmpdir act.XXXXXX‘
rm -rf $actjobs
qstat -u  phuijgen | grep dutch_pipeline_job | gawk -F"." ’{print $1}’ \
 | sort  >$actjobs
⟨ compare the active-jobs list with the jobcounter list (15d $actjobs ) 16a ⟩
rm -rf $actjobs
◇
```

Fragment referenced in 15e.

⟨ *do the now-and-then tasks* 15e ⟩ ≡

```
⟨ verify jobs-bookkeeping 15c ⟩
◇
```

Fragment never referenced.

⟨ *compare the active-jobs list with the jobcounter list* 16a ⟩ ≡

```
if [ -e $JOBCOUNTFILE ]
then
  passeer
  sort < $JOBCOUNTFILE >$tmpfil
  gawk -v actfil=@1 -v stmp=`date +%s` '
    ⟨ awk script to compare the active-jobs list with the jobcounter list 16b ⟩
  ' $tmpfil >$JOBCOUNTFILE
  veilig
  rm -rf $tmpfil
else
  cp @1 $JOBCOUNTFILE
fi
◇
```

Fragment referenced in 15c.
Uses: `JOBCOUNTFILE` 12c, `passeer` 18bc, `veilig` 18bc.

Copy lines from the logcount file if the jobnumber matches a line in the list actual jobs. Write entries for jobnumbers that occur only in the actual job list.

⟨ *awk script to compare the active-jobs list with the jobcounter list* 16b ⟩ ≡

```
BEGIN {actlin=(getline act < actfil)}
{ while(actlin>0 && (act<$1)){
     print act " wait " stmp;
     actlin=(getline act < actfil);
  };
  if((actlin>0) && act==$1 ){
     print
     actlin=(getline act < actfil);
  }
}
END {
    while((actlin>0) && (act ~ /^[[:digit:]]+/)){
      print act " wait " stmp;
    actlin=(getline act < actfil);
 };
}
◇
```

Fragment referenced in 16a.
Uses: `print` 28b.

⟨ *check/perform every time* 16c ⟩ ≡

```
⟨ replace files from proctray when no processes are running ? ⟩
⟨ submit jobs when necessary ? ⟩
◇
```

Fragment never referenced.

⟨ *derive number of jobs to be submitted* 17a ⟩ ≡

```
      REQJOBS=$(( $(( $NRFILES / 150 )) ))
      if [ $REQJOBS -gt m4_maxjobs ]
      then
        REQJOBS=m4_maxjobs
      fi
      if [ $NRFILES -gt 0 ]
      then
        if [ $REQJOBS -eq 0 ]
        then
          REQJOBS=1
        fi
      fi
      @1=$(( $REQJOBS - $NRJOBS ))

      ◇
```

Fragment never referenced.

## 2.11 Synchronisation mechanism

Make a mechanism that ensures that only a single process can execute some functions at a time. For instance, if a process selects a file to be processed next, it selects a file name from a directory-listing and then removes the selected file from the directory. The two steps form a “critical code section” and only a single process at a time should be allowed to execute this section. Therefore, generate the functions `passeer` and `veilig` (cf. E.W. Dijkstra). When a process completes `passeer`, no other processes can complete `passeer` until the first process executes `veilig`.

Function `passeer` tries repeatedly to create a *lock directory*, until it succeeds and function `veilig` removes the lock directory.

Sometimes de-synchonisation is good, to prevent that all processes are waiting at the same time for the same event. Therefore, now and then a process should wait a random amount of time. We don't need to use sleep, because the cores have no other work to do.

⟨ *functions* 17b ⟩ ≡

```
      waitabit()
      { ( RR=$RANDOM
          while
            [ $RR -gt 0 ]
          do
          RR=$((RR - 1))
          done
        )

      }

      ◇
```

Fragment defined by 4e, 5a, 7c, 17b, 18bc.
Fragment referenced in 12b, 21a.
Defines: `waitabit` 18b.

⟨*parameters* 18a⟩ ≡

```
export LOCKDIR=/home/phuijgen/nlp/Pipeline-NL-Lisa/.lock
```

◇

Fragment defined by 3, 4b, 6b, 8a, 10c, 11f, 12c, 18a, 21d.
Fragment referenced in 4a.
Defines: `LOCKDIR` 18bc, 19a.

⟨*functions* 18b⟩ ≡

```
function passeer () {
 while ! (mkdir $LOCKDIR 2> /dev/null)
 do
   waitabit
 done
}

function veilig () {
  rmdir "$LOCKDIR"
}
```

◇

Fragment defined by 4e, 5a, 7c, 17b, 18bc.
Fragment referenced in 12b, 21a.
Defines: `passeer` 13f, 14b, 15b, 16a, 19bc, 20a, `veilig` 5b, 13f, 14b, 15b, 16a, 18c, 19bc, 20a, 21a.
Uses: `LOCKDIR` 18a, `waitabit` 17b.

Function `runsingle` is similar to `passeer`, but it exits when the lock is set.

⟨*functions* 18c⟩ ≡

```
function runsingle () {
 if ! (mkdir $LOCKDIR 2> /dev/null)
 then
    exit
 fi
}

function veilig () {
  rmdir "$LOCKDIR"
}
```

◇

Fragment defined by 4e, 5a, 7c, 17b, 18bc.
Fragment referenced in 12b, 21a.
Defines: `passeer` 13f, 14b, 15b, 16a, 18b, 19bc, 20a, `veilig` 5b, 13f, 14b, 15b, 16a, 18b, 19bc, 20a, 21a.
Uses: `LOCKDIR` 18a.

The processes that execute these functions can crash and they are killed when the time alotted to them has been used up. Thus it is possible that a process that executed `passeer` is not able to execute `veilig`. As a result, all other processes would come to a halt. Therefore, check the age of the lock directory periodically and remove the directory when it is older than, say, two minutes (executing critical code sections ought to take only a very short amount of time).

⟨ *remove old lockdir* 19a ⟩ ≡

```
      find $LOCKDIR -amin 10 -print 2>/dev/null | xargs rm -rf
      ◇
```

Fragment referenced in 21a.
Uses: `LOCKDIR` 18a, `print` 28b.

The synchronisation mechanism can be used to have parallel processes update the same counter.

⟨ *increment filecontent* 19b ⟩ ≡

```
      passeer
      NUM=`cat @1`
      echo $((NUM + 1 )) > @1
      veilig
      ◇
```

Fragment never referenced.
Uses: `passeer` 18bc, `veilig` 18bc.

⟨ *decrement filecontent* 19c ⟩ ≡

```
      passeer
      NUM=`cat @1`
      echo $((NUM - 1 )) > @1
      veilig
      ◇
```

Fragment never referenced.
Uses: `passeer` 18bc, `veilig` 18bc.

We will need a mechanism to find out whether a certain operation has taken place within a certain past time period. We use the timestamp of a file for that. When the operation to be monitored is executed, the file is touched. The following macro checks such a file. It has the following three arguments: 1) filename; 2) time-out period; 3) result. The result parameter will become true when the file didn't exist or when it had not been touched during the time-out period. In those cases the macro touches the file.

⟨ *check whether update is necessary* 20a ⟩ ≡

```
⟨ write log (20b now: ‘date +%s‘ ) 11a ⟩
arg=@1
stamp=‘date -r @1 +%s‘
⟨ write log (20c $arg: $stamp ) 11a ⟩
passeer
if [ ! -e @1 ]
then
  @3=true
elif [ $((‘date +%s‘ - ‘date -r @1 +%s‘)) -gt @2 ]
then
  @3=true
else
  @3=false
fi
if $@3
then
  echo ‘date‘ > @1
fi
veilig
if $@3
then
  ⟨ write log (20d yes, update ) 11a ⟩
else
  ⟨ write log (20e no, no update ) 11a ⟩
fi
◇
```

Fragment never referenced.

## 2.12 The management script

"../runit" 21a≡

```
#!/bin/bash
source /home/phuijgen/nlp/Pipeline-NL-Lisa/parameters
⟨ functions 4e, ... ⟩
⟨ remove old lockdir 19a ⟩
runsingle
⟨ init logfile 10e ⟩
⟨ load stopos module 6c ⟩
⟨ check/create directories 5b ⟩
⟨ get stopos status 8c ⟩
waitingfilecount=‘find $intray -type f -print | wc -l‘
readyfilecount=‘find $outtray -type f -print | wc -l‘
procfilecount=‘find $proctray -type f -print | wc -l‘
unprocessedfilecount=$((waitingfilecount + $procfilecount))
submitted_job_count=‘qstat -u  $USER | grep dutch | wc -l‘
if
  [ $submitted_job_count -eq 0 ]
then
   ⟨ set up new stopos pool 7a ⟩
else
   ⟨ restore old procfiles 7d ⟩
fi
⟨ submit jobs 22a ⟩

veilig
◇
```

Uses: `intray` 3, `outtray` 3, `print` 28b, `proctray` 4b, `veilig` 18bc.

⟨ *make scripts executable* 21b ⟩ ≡

```
chmod 775 /home/phuijgen/nlp/Pipeline-NL-Lisa/runit
◇
```

Fragment defined by 9d, 21b, 34c.
Fragment referenced in 34d.

Regenerate the stopos pool if it is empty but there are still input-files.

⟨ *regenerate pool if it is prematurely empty* 21c ⟩ ≡

```
if
  [ $untouched_files -eq 0 ]
then
  ⟨ (re-)generate stopos pool ? ⟩
fi
◇
```

Fragment never referenced.

Make sure that enough jobs are submitted. Currently we aim at one job per 150 waiting files.

⟨ *parameters* 21d ⟩ ≡

```
filesperjob=150
◇
```

Fragment defined by 3, 4b, 6b, 8a, 10c, 11f, 12c, 18a, 21d.
Fragment referenced in 4a.

⟨ *submit jobs* 22a ⟩ ≡

```
jobs_needed=$((unprocessedfilecount / $filesperjob))
if
  [ $jobs_needed -lt 1 ]
then
  jobs_needed=1
fi
jobs_to_be_submitted=$((jobs_needed - $submitted_job_count))
if
  [ $jobs_to_be_submitted -gt 0 ]
then
   ⟨ generate jobscript 22c ⟩
   ⟨ submit extra jobs (22b $jobs_to_be_submitted ) 22d ⟩
fi

◇
```

Fragment referenced in 21a.

⟨ *generate jobscript* 22c ⟩ ≡

```
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 22a.
Uses: `walltime` 3.

⟨ *submit extra jobs* 22d ⟩ ≡

```
for ((a=1; a <= @1;  a++))
do
  ⟨ submit a job 13a ⟩
done
◇
```

Fragment referenced in 22a.

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

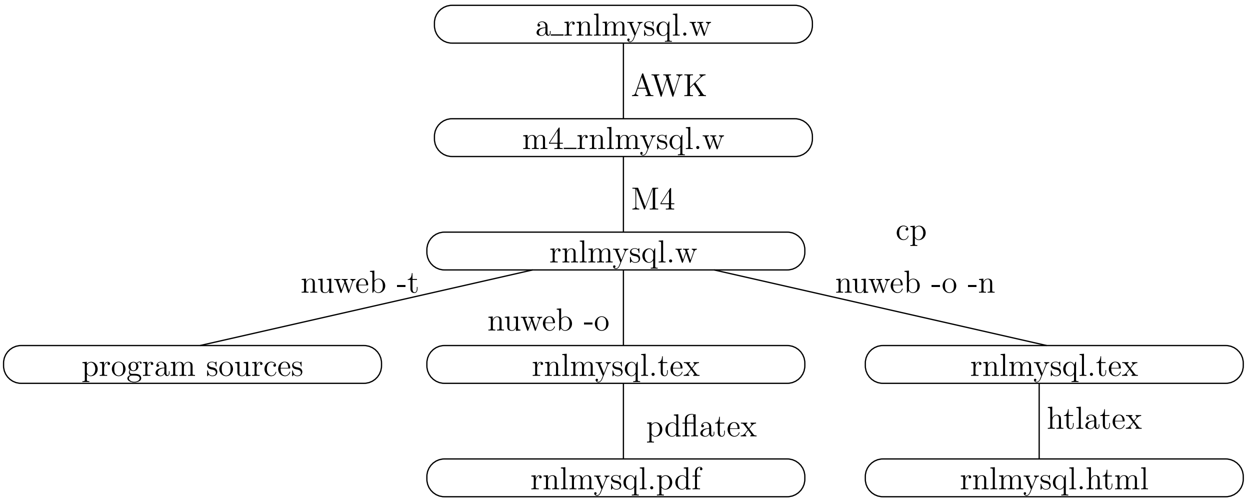

Figure 1: *Translation of the raw code of this document into printable/viewable documents and into program sources. The figure shows the pathways and the main files involved.*

| **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*

⟨ *parameters in Makefile* 24a ⟩ ≡

```
NUWEB=../env/bin/nuweb
◇
```

Fragment defined by 24a, 25a, 27ab, 29c, 31b, 34a.
Fragment referenced in 24b.
Uses: `nuweb` 30d.

### A.3 The Makefile for this project.

This chapter assembles the Makefile for this project.

`"Makefile"` 24b≡

```
⟨ default target 24c ⟩

⟨ parameters in Makefile 24a, ... ⟩

⟨ impliciete make regels 26c, ... ⟩
⟨ expliciete make regels 25b, ... ⟩
⟨ make targets 24d, ... ⟩
◇
```

The default target of make is `all`.

⟨ *default target* 24c ⟩ ≡

```
all : ⟨ all targets 24e ⟩
.PHONY : all

◇
```

Fragment referenced in 24b.
Defines: `all` Never used, `PHONY` 28a.

⟨ *make targets* 24d ⟩ ≡

```
clean:
        ⟨ clean up 25c ⟩

◇
```

Fragment defined by 24d, 28b, 29a, 32c, 34bd, 35a.
Fragment referenced in 24b.

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

⟨ *all targets* 24e ⟩ ≡

```
Pipeline_NL_Lisa.pdf◇
```

Fragment referenced in 24c.
Uses: `pdf` 28b.

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* 25a ⟩ ≡

```
.SUFFIXES: .pdf .w .tex .html .aux .log .php

◇
```

Fragment defined by 24a, 25a, 27ab, 29c, 31b, 34a.
Fragment referenced in 24b.
Defines: `SUFFIXES` Never used.
Uses: `pdf` 28b.

## 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* 25b ⟩ ≡

```

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 25bd, 26ab, 28a, 29d, 31c, 32b.
Fragment referenced in 24b.
Uses: `nuweb` 30d.

⟨ *clean up* 25c ⟩ ≡

```
rm -rf ../nuweb-1.58
◇
```

Fragment referenced in 24d.
Uses: `nuweb` 30d.

⟨ *expliciete make regels* 25d ⟩ ≡

```
../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 25bd, 26ab, 28a, 29d, 31c, 32b.
Fragment referenced in 24b.
Uses: `nuweb` 30d.

## 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* 26a ⟩ ≡

```
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 25bd, 26ab, 28a, 29d, 31c, 32b.
Fragment referenced in 24b.
Uses: `print` 28b.

### A.5.2 Run the M4 pre-processor

⟨ *expliciete make regels* 26b ⟩ ≡

```
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 25bd, 26ab, 28a, 29d, 31c, 32b.
Fragment referenced in 24b.

## 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* 26c ⟩ ≡

```
%.pdf: %.w
        ./w2pdf $<

◇
```

Fragment defined by 26c, 27c, 32a.
Fragment referenced in 24b.
Uses: `pdf` 28b.

### 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* 27a ⟩ ≡

```
FIGFILES=fileschema directorystructure

◇
```

Fragment defined by 24a, 25a, 27ab, 29c, 31b, 34a.
Fragment referenced in 24b.
Defines: `FIGFILES` 27b.

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* 27b ⟩ ≡

```
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 24a, 25a, 27ab, 29c, 31b, 34a.
Fragment referenced in 24b.
Defines: `FIGFILENAMES` Never used, `PDFT_NAMES` 29a, `PDF_FIG_NAMES` 29a, `PST_NAMES` Never used, `PS_FIG_NAMES` Never used.
Uses: `FIGFILES` 27a.

Create the graph files with program `fig2dev`:

⟨ *impliciete make regels* 27c ⟩ ≡

```
%.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 26c, 27c, 32a.
Fragment referenced in 24b.
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* 28a ⟩ ≡

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

.PHONY : bibfile
◇
```

Fragment defined by 25bd, 26ab, 28a, 29d, 31c, 32b.
Fragment referenced in 24b.
Uses: `PHONY` 24c.

### A.6.3 Create a printable/viewable document

Make a PDF document for printing and viewing.

⟨ *make targets* 28b ⟩ ≡

```
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 24d, 28b, 29a, 32c, 34bd, 35a.
Fragment referenced in 24b.
Defines: `pdf` 24e, 25a, 26c, 29a, `print` 7abd, 11e, 13a, 14ab, 15bc, 16b, 19a, 21a, 26a, `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* 29a ⟩ ≡

```
Pipeline_NL_Lisa.pdf : Pipeline_NL_Lisa.w $(W2PDF)  $(PDF_FIG_NAMES) $(PDFT_NAMES)
        chmod 775 $(W2PDF)
        $(W2PDF) $*

◇
```

Fragment defined by 24d, 28b, 29a, 32c, 34bd, 35a.
Fragment referenced in 24b.
Uses: `pdf` 28b, `PDFT_NAMES` 27b, `PDF_FIG_NAMES` 27b.

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* 29b ⟩ ≡

```
../nuweb/bin ◇
```

Fragment referenced in 34b.
Uses: `nuweb` 30d.

⟨ *parameters in Makefile* 29c ⟩ ≡

```
W2PDF=../nuweb/bin/w2pdf
◇
```

Fragment defined by 24a, 25a, 27ab, 29c, 31b, 34a.
Fragment referenced in 24b.
Uses: `nuweb` 30d.

⟨ *expliciete make regels* 29d ⟩ ≡

```
$(W2PDF) : Pipeline_NL_Lisa.w $(NUWEB)
        $(NUWEB) Pipeline_NL_Lisa.w
◇
```

Fragment defined by 25bd, 26ab, 28a, 29d, 31c, 32b.
Fragment referenced in 24b.

"../nuweb/bin/w2pdf" 29e≡

```
#!/bin/bash
# w2pdf -- compile a nuweb file
# usage: w2pdf [filename]
# 20151203 at 0854h: Generated by nuweb from a_Pipeline_NL_Lisa.w
NUWEB=../env/bin/nuweb
LATEXCOMPILER=pdflatex
⟨ filenames in nuweb compile script 30b ⟩
⟨ compile nuweb 30a ⟩

◇
```

Uses: `nuweb` 30d.

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* 30a ⟩ ≡

```
NUWEB=/home/phuijgen/nlp/Pipeline-NL-Lisa/env/bin/nuweb
⟨ run the processors until the aux file remains unchanged 31a ⟩
⟨ remove the copy of the aux file 30c ⟩
◇
```

Fragment referenced in 29e.
Uses: `nuweb` 30d.

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* 30b ⟩ ≡

```
nufil=$1
trunk=${1%%.*}
texfil=${trunk}.tex
auxfil=${trunk}.aux
oldaux=old.${trunk}.aux
indexfil=${trunk}.idx
oldindexfil=old.${trunk}.idx
◇
```

Fragment referenced in 29e.
Defines: `auxfil` 31a, 33ab, `indexfil` 31a, 33a, `nufil` 30d, 33ac, `oldaux` 30c, 31a, 33ab, `oldindexfil` 31a, 33a, `texfil` 30d, 33ac, `trunk` 30d, 33acd.

Remove the old copy if it is no longer needed.

⟨ *remove the copy of the aux file* 30c ⟩ ≡

```
rm $oldaux
◇
```

Fragment referenced in 30a, 32e.
Uses: `oldaux` 30b, 33a.

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* 30d ⟩ ≡

```
$NUWEB $nufil
$LATEXCOMPILER $texfil
makeindex $trunk
bibtex $trunk
◇
```

Fragment referenced in 31a.
Defines: `bibtex` 33cd, `makeindex` 33cd, `nuweb` 24a, 25bcd, 29bce, 30a, 31b, 32d.
Uses: `nufil` 30b, 33a, `texfil` 30b, 33a, `trunk` 30b, 33a.

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* 31a ⟩ ≡

```
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 30d ⟩
  if [ $LOOPCOUNTER -ge 10 ]
  then
    cp $auxfil $oldaux
  fi;
done
◇
```

Fragment referenced in 30a.
Uses: `auxfil` 30b, 33a, `indexfil` 30b, `oldaux` 30b, 33a, `oldindexfil` 30b.

### 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* 31b ⟩ ≡

```
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 24a, 25a, 27ab, 29c, 31b, 34a.
Fragment referenced in 24b.
Uses: `nuweb` 30d.

Make the directory:

⟨ *expliciete make regels* 31c ⟩ ≡

```
$(htmldir) :
        mkdir -p $(htmldir)


◇
```

Fragment defined by 25bd, 26ab, 28a, 29d, 31c, 32b.
Fragment referenced in 24b.

The rule to copy files in it:

⟨ *impliciete make regels* 32a ⟩ ≡

```
      $(htmldir)/% : % $(htmldir)
              cp $< $(htmldir)/


      ◇
```

Fragment defined by 26c, 27c, 32a.
Fragment referenced in 24b.

Do the work:

⟨ *expliciete make regels* 32b ⟩ ≡

```
      $(htmltarget) : $(htmlmaterial) $(htmldir)
              cd $(htmldir) && chmod 775 w2html
              cd $(htmldir) && ./w2html nlpp.w


      ◇
```

Fragment defined by 25bd, 26ab, 28a, 29d, 31c, 32b.
Fragment referenced in 24b.

Invoke:

⟨ *make targets* 32c ⟩ ≡

```
      htm : $(htmldir) $(htmltarget)


      ◇
```

Fragment defined by 24d, 28b, 29a, 32c, 34bd, 35a.
Fragment referenced in 24b.

Create a script that performs the translation.

"w2html" 32d≡

```
      #!/bin/bash
      # w2html -- make a html file from a nuweb file
      # usage: w2html [filename]
      #  [filename]: Name of the nuweb source file.
      # 20151203 at 0854h: Generated by nuweb from a_Pipeline_NL_Lisa.w
      echo "translate " $1 >w2html.log
      NUWEB=/home/phuijgen/nlp/Pipeline-NL-Lisa/env/bin/nuweb
      ⟨ filenames in w2html 33a ⟩

      ⟨ perform the task of w2html 32e ⟩


      ◇
```

Uses: `nuweb` 30d.

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* 32e ⟩ ≡

```
      ⟨ run the html processors until the aux file remains unchanged 33b ⟩
      ⟨ remove the copy of the aux file 30c ⟩
      ◇
```

Fragment referenced in 32d.

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* 33a ⟩ ≡

```
nufil=$1
trunk=${1%%.*}
texfil=${trunk}.tex
auxfil=${trunk}.aux
oldaux=old.${trunk}.aux
indexfil=${trunk}.idx
oldindexfil=old.${trunk}.idx
◇
```

Fragment referenced in 32d.
Defines: `auxfil` 30b, 31a, 33b, `nufil` 30bd, 33c, `oldaux` 30bc, 31a, 33b, `texfil` 30bd, 33c, `trunk` 30bd, 33cd.
Uses: `indexfil` 30b, `oldindexfil` 30b.

⟨ *run the html processors until the aux file remains unchanged* 33b ⟩ ≡

```
while
  ! cmp -s $auxfil $oldaux
do
  if [ -e $auxfil ]
  then
   cp $auxfil $oldaux
  fi
  ⟨ run the html processors 33c ⟩
done
⟨ run tex4ht 33d ⟩

◇
```

Fragment referenced in 32e.
Uses: `auxfil` 30b, 33a, `oldaux` 30b, 33a.

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

⟨ *run the html processors* 33c ⟩ ≡

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

Fragment referenced in 33b.
Uses: `bibtex` 30d, `makeindex` 30d, `nufil` 30b, 33a, `texfil` 30b, 33a, `trunk` 30b, 33a.

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* 33d ⟩ ≡

```
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 33b.
Uses: `bibtex` 30d, `makeindex` 30d, `trunk` 30b, 33a.

## 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* 34a ⟩ ≡

```
MKDIR = mkdir -p

◇
```

Fragment defined by 24a, 25a, 27ab, 29c, 31b, 34a.
Fragment referenced in 24b.
Defines: `MKDIR` 34b.

⟨ *make targets* 34b ⟩ ≡

```
DIRS = ⟨ directories to create 29b ⟩

$(DIRS) :
        $(MKDIR) $@

◇
```

Fragment defined by 24d, 28b, 29a, 32c, 34bd, 35a.
Fragment referenced in 24b.
Defines: `DIRS` 34d.
Uses: `MKDIR` 34a.

⟨ *make scripts executable* 34c ⟩ ≡

```
chmod -R 775  ../bin/*
chmod -R 775  ../env/bin/*
◇
```

Fragment defined by 9d, 21b, 34c.
Fragment referenced in 34d.

⟨ *make targets* 34d ⟩ ≡

```
source : Pipeline_NL_Lisa.w $(DIRS) $(NUWEB)
        $(NUWEB) Pipeline_NL_Lisa.w
        ⟨ make scripts executable 9d, ... ⟩

◇
```

Fragment defined by 24d, 28b, 29a, 32c, 34bd, 35a.
Fragment referenced in 24b.
Uses: `DIRS` 34b.

## A.8 Restore paths after transplantation

When an existing installation has been transplanted to another location, many path indications have to be adapted to the new situation. The scripts that are generated by nuweb can be repaired by re-running nuweb. After that, configuration files of some modules must be modified.

⟨ *make targets* 35a ⟩ ≡

```
transplant :
        touch a_Pipeline_NL_Lisa.w
        $(MAKE) sources
        ../env/bin/transplant

◇
```

Fragment defined by 24d, 28b, 29a, 32c, 34bd, 35a.
Fragment referenced in 24b.

In order to work as expected, the following script must be re-made after a transplantation.

"../env/bin/transplant" 35b≡

```
#!/bin/bash
LOGLEVEL=1
⟨ set variables that point to the directory-structure ? ⟩
⟨ set paths after transplantation ? ⟩
⟨ re-install modules after the transplantation ? ⟩

◇
```

# 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

## C.3 Variables