Standardised Dutch NLP pipeline

Paul Huygen <paul.huygen@huygen.nl>

8th December 2015
09:06 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=2:00
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, 11cd, 13d, 14b, 18a.
Fragment referenced in 4a.
Defines: `failtray` 6a, 11b, `intray` 5b, 6a, 7abc, 11b, 20e, `logtray` 5b, 6a, `outtray` 5b, 6a, 20e, `root` Never used, `walltime` 15a.

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 "walltime" 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, 11cd, 13d, 14b, 18a.
Fragment referenced in 4a.
Defines: `proctray` 5b, 6a, 7bcd, 11b, 20e.

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.

Here follows the same functionality, bu now as Bash function. The functions are exported in order to be able to use them in `xargs` constructions (See this Stack-exchange item.

```
⟨ functions 4e ⟩ ≡
        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 4e, 5a, 7c, 17b, 18bc.
Fragment referenced in 13f, 20e.
Defines: `movetotray` 7bc, 11b.

⟨*functions* 5a⟩ ≡

```
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 4e, 5a, 7c, 17b, 18bc.
Fragment referenced in 13f, 20e.
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 20e.
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}
failfile=$failtray/${filtrunk}
logfile=$logtray/${filtrunk}
procfile=$proctray/${filtrunk}
outpath=${outfile%/*}
procpath=${procfile%/*}
logpath=${logfile%/*}
◇
```

Fragment referenced in 9a.
Defines: `filtrunk` Never used, `logfile` 11b, `logpath` 11b, `outfile` 9a, 11b, `outpath` 11b, `procfile` 7c, 11b, `procpath` Never used.
Uses: `failtray` 3, `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, 11cd, 13d, 14b, 18a.
Fragment referenced in 4a.
Defines: `stopospool` 7ad, 8bc, 11b.

Load the stopos module in a script:

⟨ *load stopos module* 6c ⟩ ≡

```
module load stopos
◇
```

Fragment referenced in 13f, 20e.

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 20e.
Uses: `intray` 3, `print` 27a, `stopospool` 6b.

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

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

Fragment referenced in 7a.
Uses: `intray` 3, `movetotray` 4e, `print` 27a, `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
}
export -f restoreprocfile
◇
```

Fragment defined by 4e, 5a, 7c, 17b, 18bc.
Fragment referenced in 13f, 20e.
Defines: `restoreprocfile` 7d.
Uses: `intray` 3, `movetotray` 4e, `procfile` 6a, `proctray` 4b.

⟨ *restore old procfiles* 7d ⟩ ≡

```
restorefilelist=‘mktemp -t restore.XXXXXX‘
find $proctray -type f -cmin +$maxproctime -print | \
   xargs -iaap  bash -c 'restoreprocfile aap $restorefilelist'
stopos -p $stopospool add $restorefilelist
rm $restorefilelist
◇
```

Fragment referenced in 20e.
Uses: `maxproctime` 8a, `print` 27a, `proctray` 4b, `restoreprocfile` 7c, `stopospool` 6b.

⟨ *parameters* 8a ⟩ ≡

```
maxproctime=15
◇
```

Fragment defined by 3, 4b, 6b, 8a, 11cd, 13d, 14b, 18a.
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 20e.
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 13f.
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, 10b.

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 11a ⟩
OLDD=‘pwd‘
TEMPDIR=‘mktemp -t -d ontemp.XXXXXX‘
cd $TEMPDIR
cat             | $pipebindir/tok           > tok.naf
cat tok.naf     | $pipebindir/mor           > mor.naf
cat mor.naf     | $pipebindir/nerc_conll02  > nerc.naf
cat nerc.naf    | $pipebindir/wsd           > wsd.naf
cat wsd.naf     | $pipebindir/ned           > ned.naf
cat ned.naf     | $pipebindir/heideltime    > times.naf
cat times.naf   | $pipebindir/onto          > onto.naf
cat onto.naf    | $pipebindir/srl           > srl.naf
cat srl.naf     | $pipebindir/evcoref       > ecrf.naf
cat ecrf.naf    | $pipebindir/framesrl      > fsrl.naf
cat fsrl.naf    | $pipebindir/dbpner        > dbpner.naf
cat dbpner.naf  | $pipebindir/nomevent      > nomev.naf
cat nomev.naf   | $pipebindir/postsrl       > psrl.naf
cat psrl.naf    | $pipebindir/opinimin
rm -rf $TEMPDIR
◇
```

⟨ *make scripts executable* 10a ⟩ ≡

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

Fragment defined by 10a, 21a, 33b.
Fragment referenced in 33c.

Let us start a pipeline with more facilities.

- Create a log file that accepts the log info

"../newpipenl" 10b≡

```
#!/bin/bash
source /home/phuijgen/nlp/Pipeline-NL-Lisa/parameters
⟨ directories of the pipeline 9b ⟩
⟨ set utf-8 11a ⟩
OLDD=‘pwd‘
TEMPDIR=‘mktemp -t -d ontemp.XXXXXX‘
cd $TEMPDIR
cat | $pipebindir/tok >tok.naf
⟨ nextmodule (10c tok,10d mor,10e mor ) 10 ⟩
⟨ nextmodule (10f mor,10g nerc_conll02,10h nerc ) 10 ⟩
⟨ nextmodule (10i nerc,10j wsd,10k wsd ) 10 ⟩
⟨ nextmodule (10l wsd,10m ned,10n ned ) 10 ⟩
⟨ nextmodule (10o ned,10p heideltime,10q times ) 10 ⟩
⟨ nextmodule (10r times,10s onto,10t onto ) 10 ⟩
⟨ nextmodule (10u onto,10v srl,10w srl ) 10 ⟩
⟨ nextmodule (10x srl,10y evcoref,10z ecrf ) 10 ⟩
⟨ nextmodule (10 ecrf,10| framesrl,10 fsrl ) 10 ⟩
⟨ nextmodule (10 fsrl,10 dbpner,10 dbpner ) 10 ⟩
⟨ nextmodule (10 dbpner,10 nemevent,10 nomev ) 10 ⟩
⟨ nextmodule (10 nomev,10 postsrl,10 psrl ) 10 ⟩
⟨ nextmodule (10 psrl,10 opinimin,10 opinimin ) 10 ⟩
cd $OLDD
rm -rf $TEMPDIR
exit
◇
```

If a module has been passed, proceed with the next module unless previous module failed. The follosing macro, `nextmodule`, tests whether the last module has been successfull. If so, it writes a header to standard error (the logfile) and starts up next module. Otherwise, it exits the pipeline script with an error code.

⟨ *nextmodule* 10 ⟩ ≡

```
if
  [ $? -gt 0 ]
then
  err=$?
  cd $OLDD
  rm -rf $TEMPDIR
  exit $err
fi
echo ‘date +%s‘: @2: >&2
cat @1.naf | $pipebindir/@2 >@3.naf
◇
```

Fragment referenced in 10b.

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

⟨ *set utf-8* 11a ⟩ ≡

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

Fragment referenced in 9c, 10b.

Actually, we do not yet handle failed files separately.

⟨ *process infile* 11b ⟩ ≡

```
movetotray $infile $intray $proctray
mkdir -p $outpath
mkdir -p $logpath
cat $procfile | timeout 1500 /home/phuijgen/nlp/Pipeline-NL-
Lisa/newpipenl 2>$logfile  >$outfile
exitstat=$?
if
  [ $exitstat -gt 0 ]
then
  if
    [ $exitstat == 124 ]
  then
    echo ‘date +%s‘: Time-out >>$logfile
  fi
  movetotray $procfile $proctray $failtray
else
rm $procfile
fi
stopos -p $stopospool remove
◇
```

Fragment referenced in 12d.
Uses: `failtray` 3, `intray` 3, `logfile` 6a, `logpath` 6a, `movetotray` 4e, `outfile` 6a, `outpath` 6a, `procfile` 6a, `proctray` 4b, `stopospool` 6b.

Select a proper spotlighthost:

⟨ *parameters* 11c ⟩ ≡

```
export spotlighthost=130.37.53.38
◇
```

Fragment defined by 3, 4b, 6b, 8a, 11cd, 13d, 14b, 18a.
Fragment referenced in 4a.
Defines: `spotlighthost` Never used.

2.6 Time log

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

⟨ *parameters* 11d ⟩ ≡

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

Fragment defined by 3, 4b, 6b, 8a, 11cd, 13d, 14b, 18a.
Fragment referenced in 4a.

⟨ *add timelog entry* 12a ⟩ ≡

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

Fragment referenced in 12d.

2.7 General log mechanism

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

⟨ *init logfile* 12b ⟩ ≡

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

Fragment referenced in 20e.
Defines: `LOGFIL` 12c, `LOGGING` 12c.

⟨ *write log* 12c ⟩ ≡

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

Fragment referenced in 19d.
Uses: `LOGFIL` 12b, `LOGGING` 12b.

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* 12d ⟩ ≡

```
⟨ determine amount of memory and nodes 13c ⟩
⟨ determine number of parallel processes 13e ⟩
procnum=0
for ((i=1 ; i<=$maxprocs ; i++))
do
  ( procnum=$i
    while
       getfile
       [ ! -z $infile ]
    do
        ⟨ add timelog entry (13a Start $infile ) 12a ⟩
       ⟨ process infile 11b ⟩
       ⟨ add timelog entry (13b Finished $infile ) 12a ⟩

    done
  )&
done

◇
```

Fragment referenced in 13f.

⟨ *determine amount of memory and nodes* 13c ⟩ ≡
```
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 12d.
Uses: `print` 27a.

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* 13d ⟩ ≡
```
mem_per_process=4
◇
```
Fragment defined by 3, 4b, 6b, 8a, 11cd, 13d, 14b, 18a.
Fragment referenced in 4a.

⟨ *determine number of parallel processes* 13e ⟩ ≡
```
export memchunks=$((memory / mem_per_process))
if
  [ $ncores -gt $memchunks ]
then
  maxprocs=$memchunks
else
  maxprocs=ncores
fi
◇
```
Fragment referenced in 12d.

2.9 The job

`"../dutch_pipeline_job.m4"` 13f≡
```
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 12d ⟩
wait
exit

◇
```

2.10 Manage the jobs

Find out how many submitted jobs there are and how many are running.

⟨ *count jobs* 14a ⟩ ≡

```
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=‘cat $joblist | gawk ’
    { match($0, /Total Jobs:[[:blank:]]*([[:digit:]]+)[[:blank:]]*Active/, arr)
      print arr[1]
    }’‘
rm $joblist
◇
```

Fragment referenced in 20e.
Defines: `running_jobs` Never used, `total_jobs` 14c, 20e.
Uses: `print` 27a.

Make sure that enough jobs are submitted. Currently we aim at one job per 100 waiting files.

⟨ *parameters* 14b ⟩ ≡

```
filesperjob=100
◇
```

Fragment defined by 3, 4b, 6b, 8a, 11cd, 13d, 14b, 18a.
Fragment referenced in 4a.

The follwing code-piece submits jobs when necessary. Note that this piece will be used when it is already known that there are files waiting to be processed. So, there must be at least one job.

⟨ *submit jobs* 14c ⟩ ≡

```
jobs_needed=$((unprocessedfilecount / $filesperjob))
if
  [ $jobs_needed -lt 1 ]
then
  jobs_needed=1
fi
jobs_to_be_submitted=$((jobs_needed - $total_jobs))
if
  [ $jobs_to_be_submitted -gt 0 ]
then
   ⟨ generate jobscript 15a ⟩
   qsub -t 1-$jobs_to_be_submitted /home/phuijgen/nlp/Pipeline-NL-
Lisa/dutch_pipeline_job
fi

◇
```

Fragment referenced in 20e.
Defines: `jobs_needed` Never used, `jobs_to_be_submitted` Never used.
Uses: `total_jobs` 14a.

⟨ *generate jobscript* 15a ⟩ ≡

```
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 14c.
Uses: `walltime` 3.

2.10.1 Keep it going

The script **runit** performs job management. Therefore, this script must be started at regular intervals. We cannot install cron-jobs on Lisa to do this. Therefore, it would be a good idea to to have jobs starting runit now and then. I tried to do that over ssh, but it did not succeed (timed out).

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* 15b ⟩ ≡

```
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 never referenced.

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

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

```
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 never referenced.
Uses: `passeer` 18bc, `print` 27a, `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* 16a ⟩ ≡

```
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 (16b $actjobs ) 16d ⟩
rm -rf $actjobs
◇
```

Fragment referenced in 16c.

⟨ *do the now-and-then tasks* 16c ⟩ ≡

```
⟨ verify jobs-bookkeeping 16a ⟩
◇
```

Fragment never referenced.

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

```
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 16e ⟩
  ' $tmpfil >$JOBCOUNTFILE
  veilig
  rm -rf $tmpfil
else
  cp @1 $JOBCOUNTFILE
fi
◇
```

Fragment referenced in 16a.
Uses: `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* 16e ⟩ ≡

```
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 16d.
Uses: `print` 27a.

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

```
REQJOBS=$(( $(( $NRFILES / 100 )) ))
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 13f, 20e.
Defines: `waitabit` 18b.

⟨*parameters* 18a⟩ ≡

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

◇
```

Fragment defined by 3, 4b, 6b, 8a, 11cd, 13d, 14b, 18a.
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 13f, 20e.
Defines: `passeer` 15c, 16d, 19bcd, `veilig` 5b, 15c, 16d, 18c, 19bcd, 20e.
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 13f, 20e.
Defines: `passeer` 15c, 16d, 18b, 19bcd, `veilig` 5b, 15c, 16d, 18b, 19bcd, 20e.
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 20e.
Uses: `LOCKDIR` 18a, `print` 27a.

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 19d ⟩ ≡
      ⟨ write log (20a now: ‘date +%s‘ ) 12c ⟩
      arg=@1
      stamp=‘date -r @1 +%s‘
      ⟨ write log (20b $arg: $stamp ) 12c ⟩
      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 (20c yes, update ) 12c ⟩
      else
        ⟨ write log (20d no, no update ) 12c ⟩
      fi
      ◇
```

Fragment never referenced.

2.12 The management script

```
"../runit" 20e≡
      #!/bin/bash
      source /home/phuijgen/nlp/Pipeline-NL-Lisa/parameters
      ⟨ functions 4e, ... ⟩
      ⟨ remove old lockdir 19a ⟩
      runsingle
      ⟨ init logfile 12b ⟩
      ⟨ 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))
      ⟨ count jobs 14a ⟩
      if
        [ $total_jobs -eq 0 ]
      then
         ⟨ set up new stopos pool 7a ⟩
      else
         ⟨ restore old procfiles 7d ⟩
      fi
      ⟨ submit jobs 14c ⟩

      veilig
      ◇
```

Uses: `intray` 3, `outtray` 3, `print` 27a, `proctray` 4b, `total_jobs` 14a, `veilig` 18bc.

```
⟨ make scripts executable 21a ⟩ ≡
      chmod 775 /home/phuijgen/nlp/Pipeline-NL-Lisa/runit
      ◇
```

Fragment defined by 10a, 21a, 33b.
Fragment referenced in 33c.

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.

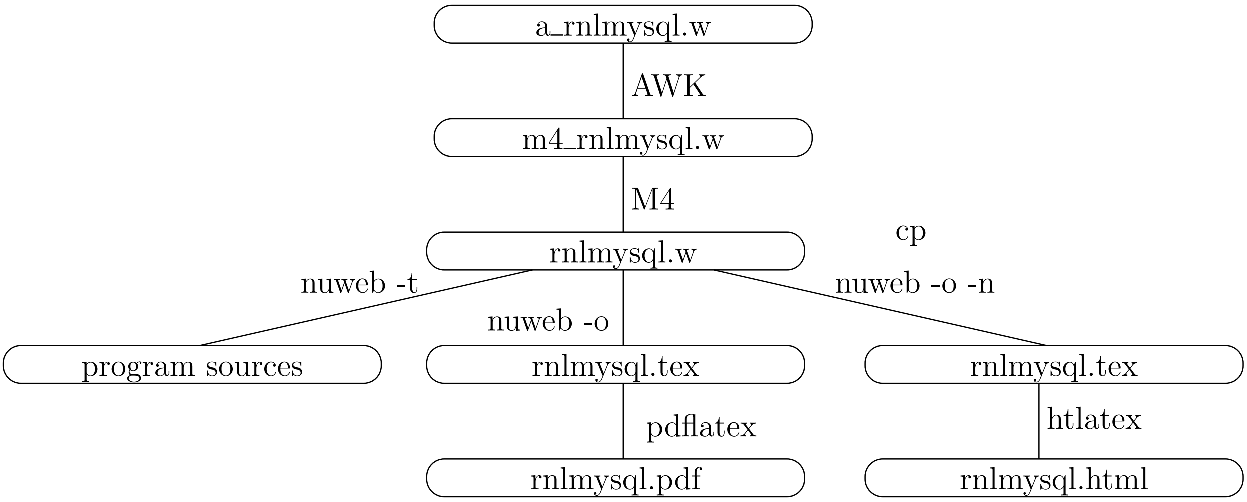

Figure 1: *Translation of the raw code of this document into printable/viewable documents and into program sources. The figure shows the pathways and the main files involved.*

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* 21b ⟩ ≡

```
NUWEB=../env/bin/nuweb
◇
```

Fragment defined by 21b, 23e, 25c, 26a, 28b, 30a, 32d.
Fragment referenced in 23a.
Uses: `nuweb` 29c.

A.3 The Makefile for this project.

This chapter assembles the Makefile for this project.

`"Makefile"` 23a≡

```
⟨ default target 23b ⟩

⟨ parameters in Makefile 21b, ... ⟩

⟨ impliciete make regels 25b, ... ⟩
⟨ expliciete make regels 24a, ... ⟩
⟨ make targets 23c, ... ⟩
◇
```

The default target of make is `all`.

⟨ *default target* 23b ⟩ ≡

```
all : ⟨ all targets 23d ⟩
.PHONY : all

◇
```

Fragment referenced in 23a.
Defines: `all` Never used, `PHONY` 26c.

⟨ *make targets* 23c ⟩ ≡

```
clean:
        ⟨ clean up 24b ⟩

◇
```

Fragment defined by 23c, 27ab, 31b, 33ac.
Fragment referenced in 23a.

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

⟨ *all targets* 23d ⟩ ≡

```
Pipeline_NL_Lisa.pdf◇
```

Fragment referenced in 23b.
Uses: `pdf` 27a.

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* 23e ⟩ ≡

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

```

◇

Fragment defined by 21b, 23e, 25c, 26a, 28b, 30a, 32d.
Fragment referenced in 23a.
Defines: `SUFFIXES` Never used.
Uses: `pdf` 27a.

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* 24a ⟩ ≡

```

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 24acd, 25a, 26c, 28c, 30b, 31a.
Fragment referenced in 23a.
Uses: `nuweb` 29c.

⟨ *clean up* 24b ⟩ ≡

```
rm -rf ../nuweb-1.58
```

◇

Fragment referenced in 23c.
Uses: `nuweb` 29c.

⟨ *expliciete make regels* 24c ⟩ ≡

```
../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 24acd, 25a, 26c, 28c, 30b, 31a.
Fragment referenced in 23a.
Uses: `nuweb` 29c.

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* 24d ⟩ ≡

```
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 24acd, 25a, 26c, 28c, 30b, 31a.
Fragment referenced in 23a.
Uses: `print` 27a.

A.5.2 Run the M4 pre-processor

⟨ *expliciete make regels* 25a ⟩ ≡

```
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 24acd, 25a, 26c, 28c, 30b, 31a.
Fragment referenced in 23a.

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* 25b ⟩ ≡

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

◇
```

Fragment defined by 25b, 26b, 30c.
Fragment referenced in 23a.
Uses: `pdf` 27a.

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* 25c ⟩ ≡

```
FIGFILES=fileschema directorystructure

◇
```

Fragment defined by 21b, 23e, 25c, 26a, 28b, 30a, 32d.
Fragment referenced in 23a.
Defines: `FIGFILES` 26a.

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* 26a ⟩ ≡

```
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 21b, 23e, 25c, 26a, 28b, 30a, 32d.
Fragment referenced in 23a.
Defines: `FIGFILENAMES` Never used, `PDFT_NAMES` 27b, `PDF_FIG_NAMES` 27b, `PST_NAMES` Never used, `PS_FIG_NAMES` Never used.
Uses: `FIGFILES` 25c.

Create the graph files with program `fig2dev`:

⟨ *impliciete make regels* 26b ⟩ ≡

```
%.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 25b, 26b, 30c.
Fragment referenced in 23a.
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* 26c ⟩ ≡

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

.PHONY : bibfile
◇
```

Fragment defined by 24acd, 25a, 26c, 28c, 30b, 31a.
Fragment referenced in 23a.
Uses: `PHONY` 23b.

A.6.3 Create a printable/viewable document

Make a PDF document for printing and viewing.

⟨ *make targets* 27a ⟩ ≡

```
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 23c, 27ab, 31b, 33ac.
Fragment referenced in 23a.
Defines: `pdf` 23de, 25b, 27b, `print` 7abd, 13c, 14a, 15c, 16ae, 19a, 20e, 24d, `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* 27b ⟩ ≡

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

◇
```

Fragment defined by 23c, 27ab, 31b, 33ac.
Fragment referenced in 23a.
Uses: `pdf` 27a, `PDFT_NAMES` 26a, `PDF_FIG_NAMES` 26a.

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* 28a ⟩ ≡

```
../nuweb/bin ◇
```

Fragment referenced in 33a.
Uses: `nuweb` 29c.

⟨ *parameters in Makefile* 28b ⟩ ≡

```
W2PDF=../nuweb/bin/w2pdf
◇
```

Fragment defined by 21b, 23e, 25c, 26a, 28b, 30a, 32d.
Fragment referenced in 23a.
Uses: `nuweb` 29c.

⟨ *expliciete make regels* 28c ⟩ ≡

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

Fragment defined by 24acd, 25a, 26c, 28c, 30b, 31a.
Fragment referenced in 23a.

"../nuweb/bin/w2pdf" 28d≡

```
#!/bin/bash
# w2pdf -- compile a nuweb file
# usage: w2pdf [filename]
# 20151208 at 0906h: Generated by nuweb from a_Pipeline_NL_Lisa.w
NUWEB=../env/bin/nuweb
LATEXCOMPILER=pdflatex
⟨ filenames in nuweb compile script 29a ⟩
⟨ compile nuweb 28e ⟩

◇
```

Uses: `nuweb` 29c.

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* 28e ⟩ ≡

```
NUWEB=/home/phuijgen/nlp/Pipeline-NL-Lisa/env/bin/nuweb
⟨ run the processors until the aux file remains unchanged 29d ⟩
⟨ remove the copy of the aux file 29b ⟩
◇
```

Fragment referenced in 28d.
Uses: `nuweb` 29c.

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* 29a ⟩ ≡

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

Fragment referenced in 28d.
Defines: `auxfil` 29d, 31e, 32a, `indexfil` 29d, 31e, `nufil` 29c, 31e, 32b, `oldaux` 29bd, 31e, 32a, `oldindexfil` 29d, 31e, `texfil` 29c, 31e, 32b, `trunk` 29c, 31e, 32bc.

Remove the old copy if it is no longer needed.

⟨ *remove the copy of the aux file* 29b ⟩ ≡

```
rm $oldaux
◇
```

Fragment referenced in 28e, 31d.
Uses: `oldaux` 29a, 31e.

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* 29c ⟩ ≡

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

Fragment referenced in 29d.
Defines: `bibtex` 32bc, `makeindex` 32bc, `nuweb` 21b, 24abc, 28abde, 30a, 31c.
Uses: `nufil` 29a, 31e, `texfil` 29a, 31e, `trunk` 29a, 31e.

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* 29d ⟩ ≡

```
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 29c ⟩
  if [ $LOOPCOUNTER -ge 10 ]
  then
    cp $auxfil $oldaux
  fi;
done
◇
```

Fragment referenced in 28e.
Uses: `auxfil` 29a, 31e, `indexfil` 29a, `oldaux` 29a, 31e, `oldindexfil` 29a.

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* 30a ⟩ ≡

```
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 21b, 23e, 25c, 26a, 28b, 30a, 32d.
Fragment referenced in 23a.
Uses: `nuweb` 29c.

Make the directory:

⟨ *expliciete make regels* 30b ⟩ ≡

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


◇
```

Fragment defined by 24acd, 25a, 26c, 28c, 30b, 31a.
Fragment referenced in 23a.

The rule to copy files in it:

⟨ *impliciete make regels* 30c ⟩ ≡

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

◇
```

Fragment defined by 25b, 26b, 30c.
Fragment referenced in 23a.

Do the work:

⟨ *expliciete make regels* 31a ⟩ ≡

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

◇
```

Fragment defined by 24acd, 25a, 26c, 28c, 30b, 31a.
Fragment referenced in 23a.

Invoke:

⟨ *make targets* 31b ⟩ ≡

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

◇
```

Fragment defined by 23c, 27ab, 31b, 33ac.
Fragment referenced in 23a.

Create a script that performs the translation.

"w2html" 31c≡

```
#!/bin/bash
# w2html -- make a html file from a nuweb file
# usage: w2html [filename]
#  [filename]: Name of the nuweb source file.
# 20151208 at 0906h: 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 31e ⟩

⟨ perform the task of w2html 31d ⟩

◇
```

Uses: nuweb 29c.

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* 31d ⟩ ≡

```
⟨ run the html processors until the aux file remains unchanged 32a ⟩
⟨ remove the copy of the aux file 29b ⟩
◇
```

Fragment referenced in 31c.

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* 31e ⟩ ≡

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

Fragment referenced in 31c.
Defines: `auxfil` 29ad, 32a, `nufil` 29ac, 32b, `oldaux` 29abd, 32a, `texfil` 29ac, 32b, `trunk` 29ac, 32bc.
Uses: `indexfil` 29a, `oldindexfil` 29a.

⟨ *run the html processors until the aux file remains unchanged* 32a ⟩ ≡

```
while
  ! cmp -s $auxfil $oldaux
do
  if [ -e $auxfil ]
  then
   cp $auxfil $oldaux
  fi
  ⟨ run the html processors 32b ⟩
done
⟨ run tex4ht 32c ⟩

◇
```

Fragment referenced in 31d.
Uses: `auxfil` 29a, 31e, `oldaux` 29a, 31e.

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

⟨ *run the html processors* 32b ⟩ ≡

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

Fragment referenced in 32a.
Uses: `bibtex` 29c, `makeindex` 29c, `nufil` 29a, 31e, `texfil` 29a, 31e, `trunk` 29a, 31e.

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* 32c ⟩ ≡

```
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 32a.
Uses: `bibtex` 29c, `makeindex` 29c, `trunk` 29a, 31e.

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* 32d ⟩ ≡

```
MKDIR = mkdir -p

◇
```

Fragment defined by 21b, 23e, 25c, 26a, 28b, 30a, 32d.
Fragment referenced in 23a.
Defines: `MKDIR` 33a.

⟨ *make targets* 33a ⟩ ≡

```
DIRS = ⟨ directories to create 28a ⟩

$(DIRS) :
        $(MKDIR) $@

◇
```

Fragment defined by 23c, 27ab, 31b, 33ac.
Fragment referenced in 23a.
Defines: `DIRS` 33c.
Uses: `MKDIR` 32d.

⟨ *make scripts executable* 33b ⟩ ≡

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

Fragment defined by 10a, 21a, 33b.
Fragment referenced in 33c.

⟨ *make targets* 33c ⟩ ≡

```
source : Pipeline_NL_Lisa.w $(DIRS) $(NUWEB)
        $(NUWEB) Pipeline_NL_Lisa.w
        ⟨ make scripts executable 10a, ... ⟩

◇
```

Fragment defined by 23c, 27ab, 31b, 33ac.
Fragment referenced in 23a.
Uses: `DIRS` 33a.

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