Standardised Dutch NLP pipeline

Paul Huygen <paul.huygen@huygen.nl>

16th February 2016
15:03 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 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. Currently, the dopcuments to be processed have to be encoded in the *NLP Annotation Format* (NAF).

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

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.

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

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

"../demoscript" 3a≡
```
#!/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
◇
```

1.2 How it works

1.2.1 Moving files around

The NAF files and the logfiles are stored in the following subdirectories of the `data`:

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

The user stores the raw NAF files in directory `data/in`. She may construct a structure with subdirectories in `data/in` that contain the NAF files. If she does that, the system copies this file-structure in the other subdirectories of `data`. Processing the files is performed by jobs. Before a job processes a document, it moves the document from `in` to `proc`, to indicate that processing this document has been started.

When the job is not able to perform processing to completion (e.g. because it is aborted), the NAF file remains in the `proc` subdirectory. A management script moves NAF of which processing has not been completed back to `in`.

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

⟨ *parameters* 3b ⟩ ≡
```
export walltime=30:00
export root=/home/phuijgen/nlp/Pipeline-NL-Lisa
export intray=/home/phuijgen/nlp/Pipeline-NL-Lisa/data/in
export proctray=/home/phuijgen/nlp/Pipeline-NL-Lisa/data/proc
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 3b, 4a, 9e, 11b, 13b, 14b, 15e.
Fragment referenced in 4c.
Defines: `failtray` 6af, 7, 20b, `intray` 6abk, 7, 8a, 9cd, 18b, `logtray` 6ah, 7, `outtray` 6a, 7, `root` 8a, 13a, 23a, `walltime` 12c.

1.2.2 Managing the documents with Stopos

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

Periodically the management script moves unprocessed documents from `data/proc` to `data/in` and regenerate the infilelist in the Stopos pool.

A list of files to be processed is called a "Stopos pool".

⟨ *parameters* 4a ⟩ ≡
```
export stopospool=dppool
```
◇

Fragment defined by 3b, 4a, 9e, 11b, 13b, 14b, 15e.
Fragment referenced in 4c.
Defines: `stopospool` 8ab, 10ac.

Load the stopos module in a script:

⟨ *load stopos module* 4b ⟩ ≡
```
module load stopos
```
◇

Fragment referenced in 21a, 23a.
Defines: `module` 18a, `stopos` 8ab, 10ac.

1.2.3 Management script

A management script `runit` set the system to work and keep the system working until all input files have been processed until either successful completion or failure. The script must run periodically in order to restore unfinished input-files from `data/proc` to `data/in` and to submit enough jobs to the job-system.

1.2.4 Job script

The management-script submits a Bash script as a job to the job-management system of Lisa. The script contains special parameters for the job system (e.g. to set the maximum processing time). It generate a number of parallel processes that do the work.

To enhance flexibility the job script is generated from a template with the M4 pre-processor.

1.2.5 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"` 4c≡
```
⟨ parameters 3b, ... ⟩
```
◇

2 Files

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

2.1 Move NAF-files around

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

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

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

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

⟨ *functions* 5a ⟩ ≡

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

export -f movetotray

◇
```

Fragment defined by 5ab, 22ab.
Fragment referenced in 21a, 23a.
Defines: `movetotray` 9cd, 18b, 20b.

⟨ *functions* 5b ⟩ ≡

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

export -f copytotray

◇
```

Fragment defined by 5ab, 22ab.
Fragment referenced in 21a, 23a.
Defines: `copytotray` Never used.

2.2 Count the files and manage directories

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

⟨ *check/create directories* 6a ⟩ ≡
```
mkdir -p $outtray
mkdir -p $failtray
mkdir -p $logtray
mkdir -p $proctray
⟨ count files in tray (6b intray,6c incount ) 6j ⟩
⟨ count files in tray (6d proctray,6e proccount ) 6j ⟩
⟨ count files in tray (6f failtray,6g failcount ) 6j ⟩
⟨ count files in tray (6h logtray,6i logcount ) 6j ⟩
unreadycount=$((incount + $proccount))
⟨ remove empty directories 6k ⟩
if
  [ ! "$(ls -A $intray)" ] &&  [ ! "$(ls -A $proctray)" ]
then
  echo "Finished processing"
  exit
fi
◇
```
Fragment referenced in 23a.
Uses: `logcount` 6a.

⟨ *count files in tray* 6j ⟩ ≡
```
@2=‘find $@1 -type f -print | wc -l‘
◇
```
Fragment referenced in 6a.
Uses: `print` 30b.

Remove empty directories in the intray and the proctray.

⟨ *remove empty directories* 6k ⟩ ≡
```
find $intray -depth -type d -empty -delete
find $proctray -depth -type d -empty -delete
mkdir -p $intray
mkdir -p $proctray
◇
```
Fragment referenced in 6a.
Uses: `intray` 3b.

2.3 Generate pathnames

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

⟨ *generate filenames* 7 ⟩ ≡
```
filtrunk=${infile##$intray/}
export outfile=$outtray/${filtrunk}
export failfile=$failtray/${filtrunk}
export logfile=$logtray/${filtrunk}
export procfile=$proctray/${filtrunk}
export outpath=${outfile%/*}
export procpath=${procfile%/*}
export logpath=${logfile%/*}
◇
```
Fragment referenced in 10b.
Defines: `filtrunk` Never used, `logfile` 18ab, `logpath` 18b, `outfile` 10b, 18a, 20b, `outpath` 18b, 20b, `procfile` 18bc, 19b, 20ab, `procpath` Never used.
Uses: `failtray` 3b, `intray` 3b, `logtray` 3b, `outtray` 3b.

2.4 Manage list of files in Stopos

2.4.1 Set up/reset pool

The processes obtain the names of the files to be processed from Stopos. Adding large amount of filenames to the stopos pool take much time, so this must be done sparingly. We do it as follows:

1. File `old.filenames` contains the filenames that have been inserted in the Stopos pool.
2. When there is no pool or the pool is empty, generate a new pool and remove `old.filenames`.
3. Move the files in the proctray that are not actually being processed back the intray. We know that these files are not being processed because either there are no running jobs or the files reside in the proctray for a longer time than jobs are allowed to run.
4. Make file `filenames` that lists all the files that are currently in the intray.
5. Remove from `old.filenames` the names of the files that are no longer in the intray. Hopefully they heve been processed or are being processed.
6. Make file `new.filenames` that contains the names of the files in the intray that are not present in `old.filenames`. These filenames have to be added to the pool.
7. Add the files in `new.filenames` to the pool.
8. Add the content of `new.filenames` to `old.filenames`.

⟨ *update the stopos pool* 8a ⟩ ≡

```
cd $root
if
  [ $running_jobs -eq 0 ]
then
  ⟨ move all procfiles to intray 9c ⟩
else
  ⟨ move old procfiles to intray 9d ⟩
fi
find $intray -type f -print | sort >infilelist
nr_of_infiles=‘cat infilelist | wc -l‘
if
  [ $nr_of_infiles -gt 0 ]
then
  if
    [ $total_jobs -eq 0 ]
  then
    ⟨ (re)generate stopos pool 8b ⟩
    cp infilelist new.infilelist
  else
    ⟨ update old.infilelist 8c ⟩
    ⟨ generate new.infilelist 9a ⟩
  fi
  stopos -p $stopospool add new.infilelist
  ⟨ add contents of new.infilelist to old.infilelist 9b ⟩
fi
◇
```

Fragment referenced in 23a.
Defines: `nr_of_infiles` Never used.
Uses: `intray` 3b, `print` 30b, `root` 3b, `running_jobs` 11a, `stopos` 4b, `stopospool` 4a, `total_jobs` 11a.

When there are no jobs, we can re-generate the Stopos pool without risk to confuse running processes. So, in this case, remove the stopos pool if it exists, remove `old.infilelist` if it exists and generate a new pool.

⟨ *(re)generate stopos pool* 8b ⟩ ≡

```
stopos -p $stopospool purge
stopos -p $stopospool create
rm -f old.infilelist
◇
```

Fragment referenced in 8a.
Uses: `stopos` 4b, `stopospool` 4a.

Find the names of files that have been inserted in the pool and are still in the intray. Pre-requisite: `filenames` and `old.filenames` are both sorted. Replace `old.filenames` with this list.

⟨ *update old.infilelist* 8c ⟩ ≡

```
comm -12 old.infilelist infilelist >old_current.infilelist
cp old_current.infilelist old.infilelist
◇
```

Fragment referenced in 8a.

Find the names or the files that are in the intray but not yet in the pool. Replace `new.filenames` with this list.

⟨ *generate new.infilelist* 9a ⟩ ≡
```
comm -13 old.infilelist infilelist >new.infilelist
```
◇
Fragment referenced in 8a.

⟨ *add contents of new.infilelist to old.infilelist* 9b ⟩ ≡
```
cat new.infilelist >>old.infilelist
sort old.infilelist >old.infilelist.sorted
mv old.infilelist.sorted old.infilelist
```
◇
Fragment referenced in 8a.

When no jobs are running, the files in the proctray will never be annotated, so move them back to the intray.

⟨ *move all procfiles to intray* 9c ⟩ ≡
```
find $proctray -type f -print | xargs -iaap  bash -
c 'movetotray aap $proctray $intray'
```
◇
Fragment referenced in 8a.
Uses: `intray` 3b, `movetotray` 5a, `print` 30b.

However, when there are running jobs, move only the files that reside longer in the proctray than jobs can run.

⟨ *move old procfiles to intray* 9d ⟩ ≡
```
find $proctray -type f -cmin +$maxproctime -print | xargs -iaap  bash -
c 'movetotray aap $proctray $intray'
```
◇
Fragment referenced in 8a.
Uses: `intray` 3b, `maxproctime` 9e, `movetotray` 5a, `print` 30b.

⟨ *parameters* 9e ⟩ ≡
```
maxproctime=30
```
◇
Fragment defined by 3b, 4a, 9e, 11b, 13b, 14b, 15e.
Fragment referenced in 4c.
Defines: `maxproctime` 9d.

2.4.2 Get a filename from the pool

To get a filename from Stopos perform:

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

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

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

⟨ *get next infile from stopos* 10a ⟩ ≡

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

Fragment referenced in 10b.
Uses: `stopos` 4b, `stopospool` 4a.

2.4.3 Function to get a filename from Stopos

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

⟨ *functions in the jobfile* 10b ⟩ ≡

```
function getfile() {
  infile=""
  outfile=""
  ⟨ get next infile from stopos 10a ⟩
  if
    [ ! "$infile" == "" ]
  then
    ⟨ generate filenames 7 ⟩
  fi
}

◇
```

Fragment defined by 10b, 16ad.
Fragment referenced in 21a.
Defines: `getfile` 15b.
Uses: `outfile` 7.

2.4.4 Remove a filename from Stopos

⟨ *remove the infile from the stopos pool* 10c ⟩ ≡

```
stopos -p $stopospool remove
◇
```

Fragment referenced in 20b.
Uses: `stopos` 4b, `stopospool` 4a.

3 Jobs

3.1 Manage the jobs

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

1. Count the number of submitted and running jobs.

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

Find out how many submitted jobs there are and how many of them are actually running. Lisa supplies an instruction `showq` that produces a list of running and waiting jobs. Extract the summaries of the numbers of running jobs and the total number of jobs.

⟨ *count jobs* 11a ⟩ ≡
```
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 23a.
Defines: `running_jobs` 8a, 24, `total_jobs` 8a, 11c, 12a, 24.
Uses: `print` 30b.

Currently we aim at one job per 30 waiting files.

⟨ *parameters* 11b ⟩ ≡
```
filesperjob=30
◇
```
Fragment defined by 3b, 4a, 9e, 11b, 13b, 14b, 15e.
Fragment referenced in 4c.

Calculate the number of jobs that have to be submitted. Note that this code-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.

⟨ *determine how many jobs have to be submitted* 11c ⟩ ≡
```
jobs_needed=$((unreadycount / $filesperjob))
if
  [ $jobs_needed -lt 1 ]
then
  jobs_needed=1
fi
jobs_to_be_submitted=$((jobs_needed - $total_jobs))
◇
```
Fragment referenced in 12a.
Uses: `jobs_needed` 12a, `jobs_to_be_submitted` 12a, `total_jobs` 11a.

Submits jobs when necessary:

⟨ *submit jobs when necessary* 12a ⟩ ≡
```
⟨ determine how many jobs have to be submitted 11c ⟩
if
  [ $jobs_to_be_submitted -gt 0 ]
then
   ⟨ submit jobs (12b $jobs_to_be_submitted ) 12d ⟩
fi
total_jobs=$((total_jobs + $jobs_to_be_submitted))
◇
```
Fragment referenced in 23a.
Uses: jobs_to_be_submitted 12a.

3.2 Generate and submit jobs

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

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

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

Process the template with `M4`.

⟨ *generate jobscript* 12c ⟩ ≡
```
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 12d.
Uses: walltime 3b.

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

⟨ *submit jobs* 12d ⟩ ≡
```
⟨ generate jobscript 12c ⟩
qsub -t 1-@1 /home/phuijgen/nlp/Pipeline-NL-Lisa/dutch_pipeline_job
◇
```
Fragment referenced in 12a.

4 Logging

There are three kinds of log-files:

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

4.1 Job logs

While we are busy with file-bookkeeping, let us handle the job-logs too. When a job finishes it produces two files that contain standard output and standard error of the log. We remove logfiles that are more than a day old. Job-logs have the same name as the job. The extension begins with character `o` (output) or `e`, followed by a number.

⟨ *remove old joblogs* 13a ⟩ ≡
```
find $root -name "dutch_pipeline_job.[eo]*" -cmin +1440 -delete
```
◇

Fragment referenced in 23a.
Uses: `root` 3b.

4.2 Time log

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

⟨ *parameters* 13b ⟩ ≡
```
export timelogfile=/home/phuijgen/nlp/Pipeline-NL-Lisa/data/log/timelog
```
◇

Fragment defined by 3b, 4a, 9e, 11b, 13b, 14b, 15e.
Fragment referenced in 4c.

⟨ *add timelog entry* 13c ⟩ ≡
```
echo `date +%s`: @1 >> $timelogfile
```
◇

Fragment referenced in 13df, 15b.

⟨ *log that the job starts* 13d ⟩ ≡
⟨ *add timelog entry* (13e `Start job $jobname`) 13c ⟩
◇

Fragment referenced in 21a.

⟨ *log that the job finishes* 13f ⟩ ≡
⟨ *add timelog entry* (13g `Finish job $jobname`) 13c ⟩
◇

Fragment referenced in 21a.

5 Processes

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

5.1 Calculate the number of parallel processes to be launched

The stopos module, that we use to synchronize file management, supplies the instructions `sara-get-num-cores` and `sara-get-mem-size` that return the number of cores resp. the amount of memory of the computer that hosts the job. **Note** that the stopos module has to be loaded before the following macro can be executed succesfully.

⟨ *determine amount of memory and nodes* 14a ⟩ ≡

```
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 15a.
Defines: `memory` 14c, `ncores` 14c.
Uses: `print` 30b.

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 `4` GB of memory per process.

⟨ *parameters* 14b ⟩ ≡

```
mem_per_process=4
◇
```

Fragment defined by 3b, 4a, 9e, 11b, 13b, 14b, 15e.
Fragment referenced in 4c.

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

⟨ *determine number of parallel processes* 14c ⟩ ≡

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

Fragment referenced in 15a.
Defines: `maxprogs` Never used.
Uses: `memory` 14a, `ncores` 14a.

5.2 Start parallel processes

⟨ *start parallel processes* 15a ⟩ ≡
```
⟨ determine amount of memory and nodes 14a ⟩
⟨ determine number of parallel processes 14c ⟩
procnum=0
for ((i=1 ; i<=$maxprocs ; i++))
do
  ( procnum=$i
    ⟨ perform the processing loop 15b ⟩
  )&
done

◇
```
Fragment referenced in 21a.
Defines: `procnum` Never used.

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

⟨ *perform the processing loop* 15b ⟩ ≡
```
while
   getfile
   [ ! -z $infile ]
do
   ⟨ add timelog entry (15c Start $infile ) 13c ⟩
   ⟨ process infile 18b ⟩
   ⟨ add timelog entry (15d Finished $infile ) 13c ⟩

done
◇
```
Fragment referenced in 15a.

6 Apply the pipeline

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

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

⟨ *parameters* 15e ⟩ ≡
```
export pipelineroot=/home/phuijgen/nlp/nlpp
export BIND=$pipelineroot/bin
◇
```
Fragment defined by 3b, 4a, 9e, 11b, 13b, 14b, 15e.
Fragment referenced in 4c.

6.1 Spotlight server

Some of the pipeline modules need to consult a *Spotlight server* that provides information from DBPedia about named entities. If it is possible, use an external server, otherwise start a server on the host of the job. We need two Spotlight servers, one for English and the other for Dutch. We expect that we can find spotlight servers on host `130.37.53.38`, port 2060 for Dutch and 2020 for

English. If it turns out that we cannot access these servers, we have to build Spotlightserver on the local host.

⟨ *functions in the jobfile* 16a ⟩ ≡

```
function check_start_spotlight {
  language=$1
  if
    [ language == "nl" ]
  then
    spotport=2060
  else
    spotport=2020
  fi
  spothost=130.37.53.38
  ⟨ check spotlight on (16b $spothost,16c $spotport ) 16e ⟩
  if
    [ $spotlightrunning -ne 0 ]
  then
    start_spotlight_on_localhost $language $spotport
    spothost="localhost"
  fi
}
◇
```

Fragment defined by 10b, 16ad.
Fragment referenced in 21a.

⟨ *functions in the jobfile* 16d ⟩ ≡

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

Fragment defined by 10b, 16ad.
Fragment referenced in 21a.

⟨ *check spotlight on* 16e ⟩ ≡

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

Fragment referenced in 16a.

6.2 Language of the document

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

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

⟨ *retrieve the language of the document* 17a ⟩ ≡

```
naflang=‘cat @1 | /home/phuijgen/nlp/nlpp/bin/langdetect‘
export naflang
#
⟨ set nercmodel 17b ⟩
◇
```

Fragment referenced in 18b.
Defines: `naflang` 17b, 18b, 19a.

⟨ *set nercmodel* 17b ⟩ ≡

```
if
  [ "$naflang" == "nl" ]
then
  export nercmodel=nl/nl-clusters-conll02.bin
else
  export nercmodel=en/en-newsreader-clusters-3-class-muc7-conll03-ontonotes-4.0.bin
fi
◇
```

Fragment referenced in 17a.
Uses: `naflang` 17a.

6.3 Apply a module on a NAF file

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

To prevent that modules are applied on the result of a failed annotation by a previous module, the exit code will be stored in variable `moduleresult`.

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

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

⟨ *functions in the pipeline-file* 18a ⟩ ≡

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

export runmodule
◇
```

Fragment defined by 18a, 19b, 20a.
Fragment referenced in 19a.
Uses: `logfile` 7, `module` 4b, `outfile` 7.

6.4 Perform the annotation on an input NAF

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

⟨ *process infile* 18b ⟩ ≡

```
movetotray $infile $intray $proctray
mkdir -p $outpath
mkdir -p $logpath
TEMPDIR=‘mktemp -d -t nlpp.XXXXXX‘
cd $TEMPDIR
⟨ retrieve the language of the document (18c $procfile ) 17a ⟩
moduleresult=0
timeout 1500 apply_pipeline \
     $naflang $procfile $logfile $TEMPDIR $spotlighthost
pipelineresult=$?
⟨ move the processed naf around 20b ⟩
rm -rf $TEMPDIR
◇
```

Fragment referenced in 15b.
Uses: `procfile` 7.

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

"../apply_pipeline" 19a≡

```
#!/bin/bash
⟨ functions in the pipeline-file 18a, ... ⟩

cd $TEMPDIR
if
  [ "$naflang" == "nl" ]
then
   apply_dutch_pipeline
else
   apply_english_pipeline
fi
◇
```

Uses: naflang 17a.

⟨ functions in the pipeline-file 19b ⟩ ≡

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

export apply_dutch_pipeline

◇
```

Fragment defined by 18a, 19b, 20a.
Fragment referenced in 19a.
Uses: procfile 7.

⟨ *functions in the pipeline-file* 20a ⟩ ≡

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

export apply_english_pipeline
```

◇

Fragment defined by 18a, 19b, 20a.
Fragment referenced in 19a.
Uses: `procfile` 7.

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

⟨ *move the processed naf around* 20b ⟩ ≡

```
if
 [ $moduleresult -eq 0 ]
then
  mkdir -p $outpath
  mv out.naf $outfile
  rm $procfile
else
  movetotray $procfile $sourcetray $failtray
fi
⟨ remove the infile from the stopos pool 10c ⟩
```

◇

Fragment referenced in 18b.
Uses: `failtray` 3b, `movetotray` 5a, `outfile` 7, `outpath` 7, `procfile` 7.

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

⟨ *set utf-8* 20c ⟩ ≡

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

◇

Fragment referenced in 21a.

6.5 The jobfile template

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

```
"../dutch_pipeline_job.m4" 21a≡
      m4_changecom()#!/bin/bash
      #PBS -lnodes=1
      #PBS -lwalltime=m4_walltime
      source /home/phuijgen/nlp/Pipeline-NL-Lisa/parameters
      export jobname=$PBS_JOBID
      ⟨ log that the job starts 13d ⟩
      ⟨ set utf-8 20c ⟩
      ⟨ initialize sematree 21b ⟩
      ⟨ load stopos module 4b ⟩
      ⟨ functions 5a, ... ⟩
      ⟨ functions in the jobfile 10b, ... ⟩
      check_start_spotlight nl
      check_start_spotlight en
      starttime=‘date +%s‘
      ⟨ start parallel processes 15a ⟩
      wait
      ⟨ log that the job finishes 13f ⟩
      exit

      ◇
```

6.6 Synchronisation mechanism

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

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

```
⟨ initialize sematree 21b ⟩ ≡
      export workdir=/home/phuijgen/nlp/Pipeline-NL-Lisa/env
      mkdir -p $workdir
      ◇
```

Fragment referenced in 21a, 23a.

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

⟨ *functions* 22a ⟩ ≡

```
function passeer () {
  local lock=$1
  sematree acquire $lock
}

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


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

◇
```

Fragment defined by 5ab, 22ab.
Fragment referenced in 21a, 23a.
Defines: `passeer` Never used, `veilig` 23a.

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

⟨ *functions* 22b ⟩ ≡

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

Fragment defined by 5ab, 22ab.
Fragment referenced in 21a, 23a.
Uses: `print` 30b.

6.7 The job management script

6.8 The management script

"../runit" 23a≡

```
#!/bin/bash
source /etc/profile
export PATH=/home/phuijgen/usrlocal/bin/:$PATH
source /home/phuijgen/nlp/Pipeline-NL-Lisa/parameters
cd $root
⟨ initialize sematree 21b ⟩
⟨ get runit options 23c ⟩
⟨ functions 5a, ... ⟩
remove_obsolete_lock runit_runs
runsingle runit_runs
⟨ load stopos module 4b ⟩
⟨ check/create directories 6a ⟩
⟨ remove old joblogs 13a ⟩
⟨ count jobs 11a ⟩
⟨ update the stopos pool 8a ⟩
⟨ submit jobs when necessary 12a ⟩
if
  [ $loud ]
then
  ⟨ print summary 24 ⟩
fi
veilig runit_runs
◇
```

Uses: root 3b, veilig 22a.

⟨ make scripts executable 23b ⟩ ≡

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

Fragment defined by 23b, 36c.
Fragment referenced in 36d.

6.9 Print a summary

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

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

⟨ get runit options 23c ⟩ ≡

```
OPTIND=1
export loud=0
while getopts "s:" opt; do
    case "$opt" in
    s)  loud=
        ;;
    esac
done
shift $((OPTIND-1))
◇
```

Fragment referenced in 23a.

Print the summary:

⟨ *print summary* 24 ⟩ ≡

```
echo in         : $incount
echo proc       : $proccount
echo failed     : $failcount
echo processed  : $((logcount - $failcount))
echo jobs       : $total_jobs
echo running    : $running_jobs
```

◇

Fragment referenced in 23a.
Uses: `failcount` 6a, `incount` 6a, `logcount` 6a, `proccount` 6a, `running_jobs` 11a, `total_jobs` 11a.

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

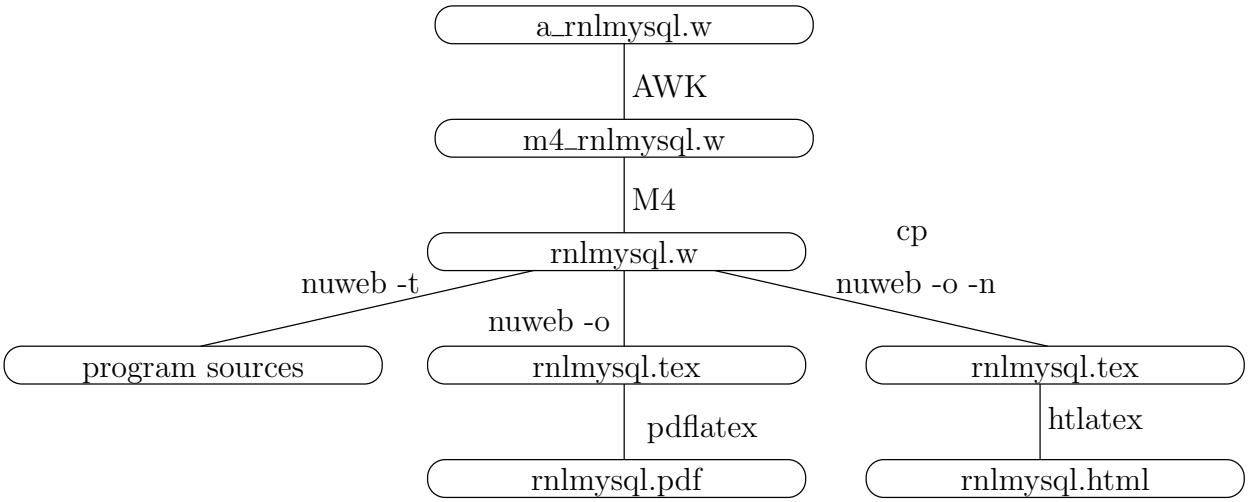

Figure 1: *Translation of the raw code of this document into printable/viewable documents and into program sources. The figure shows the pathways and the main files involved.*

printable/viewable documents and to extract the program sources. Table 1 lists the tools that are

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

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

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

⟨ *parameters in Makefile* 25 ⟩ ≡

```
NUWEB=../env/bin/nuweb
```

◇

Fragment defined by 25, 26e, 28d, 29a, 31b, 33b, 36a.
Fragment referenced in 26a.
Uses: `nuweb` 32c.

A.3 The Makefile for this project.

This chapter assembles the Makefile for this project.

```
"Makefile" 26a≡
    ⟨ default target 26b ⟩

    ⟨ parameters in Makefile 25, ... ⟩

    ⟨ impliciete make regels 28c, ... ⟩
    ⟨ expliciete make regels 27a, ... ⟩
    ⟨ make targets 26c, ... ⟩
    ◇
```

The default target of make is `all`.

```
⟨ default target 26b ⟩ ≡
    all : ⟨ all targets 26d ⟩
    .PHONY : all

    ◇
```

Fragment referenced in 26a.
Defines: `all` Never used, `PHONY` 30a.

```
⟨ make targets 26c ⟩ ≡
    clean:
        ⟨ clean up 27b ⟩

    ◇
```

Fragment defined by 26c, 30bc, 34c, 36bd.
Fragment referenced in 26a.

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

```
⟨ all targets 26d ⟩ ≡
    Pipeline_NL_Lisa.pdf◇
```

Fragment referenced in 26b.
Uses: `pdf` 30b.

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 26e ⟩ ≡
    .SUFFIXES: .pdf .w .tex .html .aux .log .php

    ◇
```

Fragment defined by 25, 26e, 28d, 29a, 31b, 33b, 36a.
Fragment referenced in 26a.
Defines: `SUFFIXES` Never used.
Uses: `pdf` 30b.

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* 27a ⟩ ≡

```
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 27ac, 28ab, 30a, 31c, 33c, 34b.
Fragment referenced in 26a.
Uses: `nuweb` 32c.

⟨ *clean up* 27b ⟩ ≡

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

◇

Fragment referenced in 26c.
Uses: `nuweb` 32c.

⟨ *expliciete make regels* 27c ⟩ ≡

```
../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 27ac, 28ab, 30a, 31c, 33c, 34b.
Fragment referenced in 26a.
Uses: `nuweb` 32c.

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* 28a ⟩ ≡

```
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 27ac, 28ab, 30a, 31c, 33c, 34b.
Fragment referenced in 26a.
Uses: `print` 30b.

A.5.2 Run the M4 pre-processor

⟨ *expliciete make regels* 28b ⟩ ≡

```
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 27ac, 28ab, 30a, 31c, 33c, 34b.
Fragment referenced in 26a.

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* 28c ⟩ ≡

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

◇

Fragment defined by 28c, 29b, 34a.
Fragment referenced in 26a.
Uses: `pdf` 30b.

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* 28d ⟩ ≡

```
FIGFILES=fileschema directorystructure
```

◇

Fragment defined by 25, 26e, 28d, 29a, 31b, 33b, 36a.
Fragment referenced in 26a.
Defines: `FIGFILES` 29a.

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* 29a ⟩ ≡

```
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 25, 26e, 28d, 29a, 31b, 33b, 36a.
Fragment referenced in 26a.
Defines: `FIGFILENAMES` Never used, `PDFT_NAMES` 30c, `PDF_FIG_NAMES` 30c, `PST_NAMES` Never used, `PS_FIG_NAMES` Never used.
Uses: `FIGFILES` 28d.

Create the graph files with program `fig2dev`:

⟨ *impliciete make regels* 29b ⟩ ≡

```
%.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 28c, 29b, 34a.
Fragment referenced in 26a.
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* 30a ⟩ ≡
```
bibfile : Pipeline_NL_Lisa.aux /home/paul/bin/mkportbib
        /home/paul/bin/mkportbib Pipeline_NL_Lisa litprog

.PHONY : bibfile
```
◇
Fragment defined by 27ac, 28ab, 30a, 31c, 33c, 34b.
Fragment referenced in 26a.
Uses: PHONY 26b.

A.6.3 Create a printable/viewable document

Make a PDF document for printing and viewing.

⟨ *make targets* 30b ⟩ ≡
```
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 26c, 30bc, 34c, 36bd.
Fragment referenced in 26a.
Defines: pdf 26de, 28c, 30c, print 6j, 8a, 9cd, 11a, 14a, 22b, 28a, 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* 30c ⟩ ≡
```
Pipeline_NL_Lisa.pdf : Pipeline_NL_Lisa.w $(W2PDF)  $(PDF_FIG_NAMES) $(PDFT_NAMES)
        chmod 775 $(W2PDF)
        $(W2PDF) $*
```
◇
Fragment defined by 26c, 30bc, 34c, 36bd.
Fragment referenced in 26a.
Uses: pdf 30b, PDFT_NAMES 29a, PDF_FIG_NAMES 29a.

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* 31a ⟩ ≡
```
../nuweb/bin ◇
```
Fragment referenced in 36b.
Uses: `nuweb` 32c.

⟨ *parameters in Makefile* 31b ⟩ ≡
```
W2PDF=../nuweb/bin/w2pdf
◇
```
Fragment defined by 25, 26e, 28d, 29a, 31b, 33b, 36a.
Fragment referenced in 26a.
Uses: `nuweb` 32c.

⟨ *expliciete make regels* 31c ⟩ ≡
```
$(W2PDF) : Pipeline_NL_Lisa.w $(NUWEB)
        $(NUWEB) Pipeline_NL_Lisa.w
◇
```
Fragment defined by 27ac, 28ab, 30a, 31c, 33c, 34b.
Fragment referenced in 26a.

`"../nuweb/bin/w2pdf"` 31d≡
```
#!/bin/bash
# w2pdf -- compile a nuweb file
# usage: w2pdf [filename]
# 20160216 at 1503h: Generated by nuweb from a_Pipeline_NL_Lisa.w
NUWEB=../env/bin/nuweb
LATEXCOMPILER=pdflatex
⟨ filenames in nuweb compile script 32a ⟩
⟨ compile nuweb 31e ⟩

◇
```
Uses: `nuweb` 32c.

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* 31e ⟩ ≡
```
NUWEB=/home/phuijgen/nlp/Pipeline-NL-Lisa/env/bin/nuweb
⟨ run the processors until the aux file remains unchanged 33a ⟩
⟨ remove the copy of the aux file 32b ⟩
◇
```
Fragment referenced in 31d.
Uses: `nuweb` 32c.

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* 32a ⟩ ≡

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

Fragment referenced in 31d.
Defines: `auxfil` 33a, 35ab, `indexfil` 33a, 35a, `nufil` 32c, 35ac, `oldaux` 32b, 33a, 35ab, `oldindexfil` 33a, 35a, `texfil` 32c, 35ac, `trunk` 32c, 35acd.

Remove the old copy if it is no longer needed.

⟨ *remove the copy of the aux file* 32b ⟩ ≡

```
rm $oldaux
◇
```

Fragment referenced in 31e, 34e.
Uses: `oldaux` 32a, 35a.

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* 32c ⟩ ≡

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

Fragment referenced in 33a.
Defines: `bibtex` 35cd, `makeindex` 35cd, `nuweb` 25, 27abc, 31abde, 33b, 34d.
Uses: `nufil` 32a, 35a, `texfil` 32a, 35a, `trunk` 32a, 35a.

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* 33a ⟩ ≡

```
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 32c ⟩
  if [ $LOOPCOUNTER -ge 10 ]
  then
    cp $auxfil $oldaux
  fi;
done
◇
```

Fragment referenced in 31e.
Uses: `auxfil` 32a, 35a, `indexfil` 32a, `oldaux` 32a, 35a, `oldindexfil` 32a.

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* 33b ⟩ ≡

```
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 25, 26e, 28d, 29a, 31b, 33b, 36a.
Fragment referenced in 26a.
Uses: `nuweb` 32c.

Make the directory:

⟨ *expliciete make regels* 33c ⟩ ≡

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

◇
```

Fragment defined by 27ac, 28ab, 30a, 31c, 33c, 34b.
Fragment referenced in 26a.

The rule to copy files in it:

⟨ *impliciete make regels* 34a ⟩ ≡
```
$(htmldir)/% : % $(htmldir)
        cp $< $(htmldir)/
```
◇
Fragment defined by 28c, 29b, 34a.
Fragment referenced in 26a.

Do the work:

⟨ *expliciete make regels* 34b ⟩ ≡
```
$(htmltarget) : $(htmlmaterial) $(htmldir)
        cd $(htmldir) && chmod 775 w2html
        cd $(htmldir) && ./w2html nlpp.w
```
◇
Fragment defined by 27ac, 28ab, 30a, 31c, 33c, 34b.
Fragment referenced in 26a.

Invoke:

⟨ *make targets* 34c ⟩ ≡
```
htm : $(htmldir) $(htmltarget)
```
◇
Fragment defined by 26c, 30bc, 34c, 36bd.
Fragment referenced in 26a.

Create a script that performs the translation.

"w2html" 34d≡
```
#!/bin/bash
# w2html -- make a html file from a nuweb file
# usage: w2html [filename]
#  [filename]: Name of the nuweb source file.
# 20160216 at 1503h: 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 35a ⟩

⟨ perform the task of w2html 34e ⟩
```
◇
Uses: nuweb 32c.

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* 34e ⟩ ≡
⟨ *run the html processors until the aux file remains unchanged* 35b ⟩
⟨ *remove the copy of the aux file* 32b ⟩
◇
Fragment referenced in 34d.

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* 35a ⟩ ≡
```
nufil=$1
trunk=${1%%.*}
texfil=${trunk}.tex
auxfil=${trunk}.aux
oldaux=old.${trunk}.aux
indexfil=${trunk}.idx
oldindexfil=old.${trunk}.idx
```
◇

Fragment referenced in 34d.
Defines: `auxfil` 32a, 33a, 35b, `nufil` 32ac, 35c, `oldaux` 32ab, 33a, 35b, `texfil` 32ac, 35c, `trunk` 32ac, 35cd.
Uses: `indexfil` 32a, `oldindexfil` 32a.

⟨ *run the html processors until the aux file remains unchanged* 35b ⟩ ≡
```
while
  ! cmp -s $auxfil $oldaux
do
  if [ -e $auxfil ]
  then
   cp $auxfil $oldaux
  fi
  ⟨ run the html processors 35c ⟩
done
⟨ run tex4ht 35d ⟩

```
◇

Fragment referenced in 34e.
Uses: `auxfil` 32a, 35a, `oldaux` 32a, 35a.

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

⟨ *run the html processors* 35c ⟩ ≡
```
$NUWEB -o -n $nufil
latex $texfil
makeindex $trunk
bibtex $trunk
htlatex $trunk
```
◇

Fragment referenced in 35b.
Uses: `bibtex` 32c, `makeindex` 32c, `nufil` 32a, 35a, `texfil` 32a, 35a, `trunk` 32a, 35a.

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* 35d ⟩ ≡
```
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 35b.
Uses: `bibtex` 32c, `makeindex` 32c, `trunk` 32a, 35a.

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* 36a ⟩ ≡
```
MKDIR = mkdir -p
```
◇
Fragment defined by 25, 26e, 28d, 29a, 31b, 33b, 36a.
Fragment referenced in 26a.
Defines: MKDIR 36b.

⟨ *make targets* 36b ⟩ ≡
```
DIRS = ⟨ directories to create 31a ⟩

$(DIRS) :
        $(MKDIR) $@
```
◇
Fragment defined by 26c, 30bc, 34c, 36bd.
Fragment referenced in 26a.
Defines: DIRS 36d.
Uses: MKDIR 36a.

⟨ *make scripts executable* 36c ⟩ ≡
```
chmod -R 775  ../bin/*
chmod -R 775  ../env/bin/*
```
◇
Fragment defined by 23b, 36c.
Fragment referenced in 36d.

⟨ *make targets* 36d ⟩ ≡
```
source : Pipeline_NL_Lisa.w $(DIRS) $(NUWEB)
        $(NUWEB) Pipeline_NL_Lisa.w
        ⟨ make scripts executable 23b, ... ⟩
```
◇
Fragment defined by 26c, 30bc, 34c, 36bd.
Fragment referenced in 26a.
Uses: DIRS 36b.

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

⟨ remove the copy of the aux file 32b ⟩ Referenced in 31e, 34e.
⟨ remove the infile from the stopos pool 10c ⟩ Referenced in 20b.
⟨ retrieve the language of the document 17a ⟩ Referenced in 18b.
⟨ run tex4ht 35d ⟩ Referenced in 35b.
⟨ run the html processors 35c ⟩ Referenced in 35b.
⟨ run the html processors until the aux file remains unchanged 35b ⟩ Referenced in 34e.
⟨ run the processors until the aux file remains unchanged 33a ⟩ Referenced in 31e.
⟨ run the three processors 32c ⟩ Referenced in 33a.
⟨ set nercmodel 17b ⟩ Referenced in 17a.
⟨ set utf-8 20c ⟩ Referenced in 21a.
⟨ start parallel processes 15a ⟩ Referenced in 21a.
⟨ submit jobs 12d ⟩ Referenced in 12a.
⟨ submit jobs when necessary 12a ⟩ Referenced in 23a.
⟨ update old.infilelist 8c ⟩ Referenced in 8a.
⟨ update the stopos pool 8a ⟩ Referenced in 23a.

C.3 Variables