Standardised Dutch NLP pipeline

Paul Huygen <paul.huygen@huygen.nl>

1st March 2016
15:40 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`.

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

6. Wait until it has finished.

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, 10c, 13b, 15b, 16b, 17e.
Fragment referenced in 4c.
Defines: `failtray` 6af, 7, 23a, `intray` 6bk, 7, 9ab, 10a, 21a, `logtray` 6ah, 7, `outtray` 6a, 7, `root` 8a, 9b, 12b, 21a, 26c, `walltime` 14d.

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, 10c, 13b, 15b, 16b, 17e.
Fragment referenced in 4c.
Defines: `stopospool` 8ad, 9c, 10bd, 11b.

Load the stopos module in a script:

⟨ *load stopos module* 4b ⟩ ≡

```
module load stopos
```

◇

Fragment referenced in 23c, 26c.
Defines: `module` 20a, `stopos` 8ad, 9c, 10bd, 11b.

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, 24b, 25a.
Fragment referenced in 23c, 26c.
Defines: `movetotray` 10a, 21a, 23a.

⟨*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, 24b, 25a.
Fragment referenced in 23c, 26c.
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 ⟩
◇
```

Fragment referenced in 26c.
Uses: `logcount` 6a.

⟨ *count files in tray* 6j ⟩ ≡

```
@2=`find $@1 -type f -print | wc -l`
◇
```

Fragment referenced in 6a.
Uses: `print` 34a.

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 11a.

Defines: `filtrunk` Never used, `logfile` 20a, `logpath` 21a, `outfile` 11a, 20a, 23a, `outpath` 21a, 23a, `procfile` 21b, 22ab, 23a, `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. First look how many filenames are still available in the pool. If the pool is empty, or there are no files in the intray, or there are no jobs, the pool must be renewed. On the other hand, if there are still lots of filenames in it, nothing has to be done.
2. If the pool is running out, something has to be done:
3. File `old.filenames` contains the filenames that have been inserted in the Stopos pool.
4. Delete from `old.filenames` the names of the files that are no longer in the intray. They have probably been processed or are being processed.
5. Move the files in the proctray that are not actually being processed back the intray. We know that these files are not being processed because either there are no running jobs or the files reside in the proctray for a longer time than jobs are allowed to run.
6. Make file `infilelist` that lists files that are currently in the intray.
7. Check whether the listed filenames are present in `old.filenames` and remove them from `infilelist` when that is the case. Put the result in `new.filenames`.
8. Add the files in `new.filenames` to the pool.
9. Add the content of `new.filenames` to `old.filenames`.

It seems that the file-bookkeeping that is external is sometimes flawed and therefore we renew the pool as often as we can.

When we run the job -manager twice per hour, Stopos needs to contain enough filenames to keep Lisa working for the next half hour. Probably Lisa's job-control system does not allow us to run more than 100 jobs at the same time. Typically a job runs seven parallel processes. Each process will probably handle at most one NAF file per minute. That means, that if stopos contains $100 \times 7 \times 30 = 2110^3$ filenames, Lisa can be kept working for half an hour.

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

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

⟨ *update the stopos pool* 8a ⟩ ≡

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


if
  [ $running_jobs -eq 0 ]
then
  old_procfiles_only=1
else
  old_procfiles_only=0
fi
  ⟨ move procfiles to intray ? ⟩
fi
nr_of_infiles=‘cat infilelist | wc -l‘
stopos -p $stopospool add new.infilelist
⟨ add contents of new.infilelist to old.infilelist ? ⟩
◇
```

Fragment referenced in 26c.

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

⟨ *is the pool full or empty?* 8d ⟩ ≡

```
@1=1
@2=0
stopos -p $stopospool status >/dev/null
result=$?
if
  [ $result -eq 0 ]
then
  if
    [ $STOPOS_PRESENT0 -gt 30000 ]
  then
    @1=0
  fi
  if
    [ $STOPOS_PRESENT0 -gt 0 ]
  then
    @2=1
  fi
fi
◇
```

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

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

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

Fragment referenced in 8a.
Uses: `intray` 3b, `print` 34a.

Note that variable `jobcount` needs to be known before running the following macro. When variable `regen_pool_condtion` is equal to zero, the pool has to be renewed.

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

```
cd $root
regen_pool_condition=1
empty_intray=‘find $intray -type f -print | head  | wc -l‘
if
  [ $empty_intray -eq 0 ] || [ $jobcount -eq 0 ] || [ $pool_empty -eq 0 ]
then
  regen_pool_condition=0
fi
◇
```

Fragment referenced in 8a.
Defines: `regen_pool_condition` 9c.
Uses: `intray` 3b, `print` 34a, `root` 3b.

⟨ *clean up pool and old.filenames* 9c ⟩ ≡

```
if
  [ $regen_pool_condition -eq 0 ]
then
  stopos -p $stopospool purge
  stopos -p $stopospool create
  rm -f old.infilelist
else
    ⟨ clean up old.infilelist 9d ⟩
fi

◇
```

Fragment referenced in 8a.
Uses: `regen_pool_condition` 9b, `stopos` 4b, `stopospool` 4a.

Remove from `old.filelist` the names of files that are no longer in the intray.

⟨ *clean up old.infilelist* 9d ⟩ ≡

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

Fragment referenced in 9c.

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

⟨ *clean up proctray* 10a ⟩ ≡

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

Fragment referenced in 8a.
Uses: `intray` 3b, `maxproctime` 10c, `movetotray` 5a, `print` 34a, `running_jobs` 12c.

⟨ *add new filenames to the pool* 10b ⟩ ≡

```
stopos -p $stopospool add infilelist
rm infilelist
◇
```

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

⟨ *parameters* 10c ⟩ ≡

```
maxproctime=30
◇
```

Fragment defined by 3b, 4a, 10c, 13b, 15b, 16b, 17e.
Fragment referenced in 4c.
Defines: `maxproctime` 10a.

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* 10d ⟩ ≡

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

Fragment referenced in 11a.
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* 11a⟩ ≡

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

◇
```

Fragment defined by 11a, 18ad.
Fragment referenced in 23c.
Defines: `getfile` 17b.
Uses: `outfile` 7.

2.4.4 Remove a filename from Stopos

⟨*remove the infile from the stopos pool* 11b⟩ ≡

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

Fragment referenced in 23a.
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. Unfortunately, it seems that this instruction shows only the running jobs in job arrays. Therefore we need to make job bookkeeping.

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

⟨ *count jobs* 12a ⟩ ≡

```
if
  [ -e jobcounter ]
then
  export jobcount=‘cat jobcounter‘
else
  jobcount=0
fi
◇
```

Fragment defined by 12abc, 13a.
Fragment referenced in 26c.

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

⟨ *count jobs* 12b ⟩ ≡

```
cd $root
ls -1 dutch_pipeline_job.[eo]* >jobloglist
finished_jobs=‘cat jobloglist | grep "\.e" | wc -l‘
mkdir -p joblogs
cat jobloglist | xargs -iaap mv aap joblogs/
if
  [ $finished_jobs -gt $jobcount ]
then
  jobcount=0
else
  jobcount=$((jobcount - $finished_jobs))
fi
◇
```

Fragment defined by 12abc, 13a.
Fragment referenced in 26c.
Uses: `root` 3b.

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

⟨ *count jobs* 12c ⟩ ≡

```
joblist=‘mktemp -t jobrep.XXXXXX‘
rm -rf $joblist
showq -u $USER | tail -n 1 > $joblist
running_jobs=‘cat $joblist | gawk ’
    { match($0, /Active Jobs:[[:blank:]]*([[:digit:]]+)[[:blank:]]*Idle/, arr)
      print arr[1]
    }’‘
total_jobs_qn=‘cat $joblist | gawk ’
    { match($0, /Total Jobs:[[:blank:]]*([[:digit:]]+)[[:blank:]]*Active/, arr)
      print arr[1]
    }’‘
rm $joblist
◇
```

Fragment defined by 12abc, 13a.
Fragment referenced in 26c.
Defines: `running_jobs` 8a, 10a, 13a, 27c, `total_jobs_qn` Never used.
Uses: `print` 34a.

If there are more running than `jobcount` lists, something is wrong. The best we can do in that case is to make `jobcount` equal to `running_jobs`.

⟨ *count jobs* 13a ⟩ ≡

```
if
  [ $running_jobs -gt $jobcount ]
then
  jobcount=$running_jobs
fi
◇
```

Fragment defined by 12abc, 13a.
Fragment referenced in 26c.
Uses: `running_jobs` 12c.

Currently we aim at one job per 30 waiting files.

⟨ *parameters* 13b ⟩ ≡

```
filesperjob=30
◇
```

Fragment defined by 3b, 4a, 10c, 13b, 15b, 16b, 17e.
Fragment referenced in 4c.

Calculate the number of jobs that have to be submitted.

⟨ *determine how many jobs have to be submitted* 13c ⟩ ≡

```
⟨ determine number of jobs that we want to have 13d, ... ⟩
jobs_to_be_submitted=$((jobs_needed - $jobcount))
◇
```

Fragment referenced in 14b.
Uses: `jobs_needed` 14b, `jobs_to_be_submitted` 14b.

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

⟨ *determine number of jobs that we want to have* 13d ⟩ ≡

```
jobs_needed=$((unreadycount / $filesperjob))
if
  [ $unreadycount -gt 0 ] && [ $jobs_needed -eq 0 ]
then
  jobs_needed=1
fi
◇
```

Fragment defined by 13d, 14a.
Fragment referenced in 13c.
Uses: `jobs_needed` 14b.

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

⟨ *determine number of jobs that we want to have* 14a ⟩ ≡

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

Fragment defined by 13d, 14a.
Fragment referenced in 13c.
Uses: `jobs_needed` 14b.

⟨ *submit jobs when necessary* 14b ⟩ ≡

```
⟨ determine how many jobs have to be submitted 13c ⟩
if
  [ $jobs_to_be_submitted -gt 0 ]
then
   ⟨ submit jobs (14c $jobs_to_be_submitted ) 15a ⟩
   jobcount=$((jobcount + $jobs_to_be_submitted))
fi
echo $jobcount > jobcounter
◇
```

Fragment referenced in 26c.
Uses: `jobs_to_be_submitted` 14b.

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* 14d ⟩ ≡

```
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 15a.
Uses: `walltime` 3b.

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

⟨ *submit jobs* 15a ⟩ ≡

```
  ⟨ generate jobscript 14d ⟩
  jobid=‘qsub -t 1-@1 /home/phuijgen/nlp/Pipeline-NL-Lisa/dutch_pipeline_job‘
◇
```

Fragment referenced in 14b.

4 Logging

There are three kinds of log-files:

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

4.1 Time log

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

⟨ *parameters* 15b ⟩ ≡

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

Fragment defined by 3b, 4a, 10c, 13b, 15b, 16b, 17e.
Fragment referenced in 4c.

⟨ *add timelog entry* 15c ⟩ ≡

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

Fragment referenced in 15df, 17b.

⟨ *log that the job starts* 15d ⟩ ≡

```
⟨ add timelog entry (15e Start job $jobname ) 15c ⟩
◇
```

Fragment referenced in 23c.

⟨ *log that the job finishes* 15f ⟩ ≡

```
⟨ add timelog entry (15g Finish job $jobname ) 15c ⟩
◇
```

Fragment referenced in 23c.

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* 16a ⟩ ≡

```
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 17a.
Defines: `memory` 16c, `ncores` 16c.
Uses: `print` 34a.

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* 16b ⟩ ≡

```
mem_per_process=4
◇
```

Fragment defined by 3b, 4a, 10c, 13b, 15b, 16b, 17e.
Fragment referenced in 4c.

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

⟨ *determine number of parallel processes* 16c ⟩ ≡

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

Fragment referenced in 17a.
Defines: `maxprogs` Never used.
Uses: `memory` 16a, `ncores` 16a.

5.2 Start parallel processes

```
⟨ run parallel processes 17a ⟩ ≡
        ⟨ determine amount of memory and nodes 16a ⟩
        ⟨ determine number of parallel processes 16c ⟩
        procnum=0
        ⟨ init processescounter 25b ⟩
        for ((i=1 ; i<=$maxprocs ; i++))
        do
          ( procnum=$i
            ⟨ increment the processes-counter 25c ⟩
            ⟨ perform the processing loop 17b ⟩
            ⟨ decrement the processes-counter, kill if this was the only process 26a ⟩
          )&
        done
        ⟨ wait for working-processes 26b ⟩
        ◇
```

Fragment referenced in 23c.
Defines: `procnum` Never used.

5.3 Perform the processing loop

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

```
⟨ perform the processing loop 17b ⟩ ≡
        while
            getfile
            [ ! -z $infile ]
        do
            ⟨ add timelog entry (17c Start $infile ) 15c ⟩
            ⟨ process infile 21a ⟩
            ⟨ add timelog entry (17d Finished $infile with result: $pipelineresult ) 15c ⟩

        done
        ◇
```

Fragment referenced in 17a.
Uses: `pipelineresult` 21a.

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/test/nlpp`. For each of the modules there is a script in subdirectory `bin`.

```
⟨ parameters 17e ⟩ ≡
        export pipelineroot=/home/phuijgen/nlp/test/nlpp
        export BIND=$pipelineroot/bin
        ◇
```

Fragment defined by 3b, 4a, 10c, 13b, 15b, 16b, 17e.
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* 18a⟩ ≡

```
function check_start_spotlight {
  language=$1
  if
    [ language == "nl" ]
  then
    spotport=2060
  else
    spotport=2020
  fi
  spotlighthost=130.37.53.38
  ⟨ check spotlight on (18b $spotlighthost,18c $spotport ) 19a ⟩
  if
    [ $spotlightrunning -ne 0 ]
  then
    start_spotlight_on_localhost $language $spotport
    spotlighthost="localhost"
    spotlightrunning=0
  fi
  export spotlighthost
  export spotlightrunning
}
◇
```

Fragment defined by 11a, 18ad.
Fragment referenced in 23c.

⟨*functions in the jobfile* 18d⟩ ≡

```
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 11a, 18ad.
Fragment referenced in 23c.

⟨ *check spotlight on* 19a ⟩ ≡

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

Fragment referenced in 18a.

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* 19b ⟩ ≡

```
naflang=‘cat @1 | /home/phuijgen/nlp/test/nlpp/bin/langdetect‘
export naflang
#
⟨ set nercmodel 19c ⟩
◇
```

Fragment referenced in 21a.
Defines: `naflang` 19c, 21c.

⟨ *set nercmodel* 19c ⟩ ≡

```
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 19b.
Defines: `nercmodel` Never used.
Uses: `naflang` 19b.

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* 20a⟩ ≡

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

export runmodule
◇
```

Fragment defined by 20ab, 22ab.
Fragment referenced in 21c.
Uses: `logfile` 7, `module` 4b, `moduleresult` 20b, `outfile` 7.

Initialise `moduleresult` with value 0:

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

```
export moduleresult=0
◇
```

Fragment defined by 20ab, 22ab.
Fragment referenced in 21c.
Defines: `moduleresult` 20a, 21a.

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* 21a ⟩ ≡
```
movetotray $infile $intray $proctray
mkdir -p $outpath
mkdir -p $logpath
export TEMPDIR=`mktemp -d -t nlpp.XXXXXX`
cd $TEMPDIR
⟨ retrieve the language of the document (21b $procfile ) 19b ⟩
moduleresult=0
timeout 1500 $root/apply_pipeline
pipelineresult=$?
⟨ move the processed naf around 23a ⟩
cd $root
rm -rf $TEMPDIR
◇
```
Fragment referenced in 17b.
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" 21c≡
```
#!/bin/bash
⟨ functions in the pipeline-file 20a, ... ⟩

cd $TEMPDIR
if
  [ "$naflang" == "nl" ]
then
   apply_dutch_pipeline
else
   apply_english_pipeline
fi
◇
```
Uses: naflang 19b.

⟨ *make scripts executable* 21d ⟩ ≡
```
chmod 775 /home/phuijgen/nlp/Pipeline-NL-Lisa/apply_pipeline
◇
```
Fragment defined by 21d, 27a, 40b.
Fragment referenced in 40c.

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

```
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 20ab, 22ab.
Fragment referenced in 21c.
Uses: `procfile` 7.

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

```
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/ned                   ned.naf
  runmodule ned.naf      $BIND/nedrer                nedr.naf
  runmodule nedr.naf     $BIND/wikify                wikif.naf
  runmodule wikif.naf    $BIND/ukb                   ukb.naf
  runmodule ukb.naf      $BIND/ewsd                  ewsd.naf
  runmodule ewsd.naf     $BIND/coreference-base      coref.naf
  runmodule coref.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 20ab, 22ab.
Fragment referenced in 21c.
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* 23a ⟩ ≡

```
if
 [ $pipelineresult -eq 0 ]
then
  mkdir -p $outpath
  mv out.naf $outfile
  rm $procfile
else
  movetotray $procfile $proctray $failtray
fi
⟨ remove the infile from the stopos pool 11b ⟩
◇
```

Fragment referenced in 21a.
Uses: `failtray` 3b, `movetotray` 5a, `outfile` 7, `outpath` 7, `pipelineresult` 21a, `procfile` 7.

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

⟨ *set utf-8* 23b ⟩ ≡

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

Fragment referenced in 23c.

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"` 23c≡

```
m4_changecom()#!/bin/bash
#PBS -lnodes=1
#PBS -lwalltime=m4_walltime
source /home/phuijgen/nlp/Pipeline-NL-Lisa/parameters
piddir=`mktemp -d -t piddir.XXXXXXX`
( $BIND/start_eSRL $piddir )&
export jobname=$PBS_JOBID
⟨ log that the job starts 15d ⟩
⟨ set utf-8 23b ⟩
⟨ load stopos module 4b ⟩
⟨ functions 5a, ... ⟩
⟨ functions in the jobfile 11a, ... ⟩
check_start_spotlight nl
check_start_spotlight en
echo spotlighthost: $spotlighthost >&2
echo spotlighthost: $spotlighthost
starttime=`date +%s`
⟨ run parallel processes 17a ⟩
⟨ log that the job finishes 15f ⟩
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* 24a ⟩ ≡

```
export workdir=/home/phuijgen/nlp/Pipeline-NL-Lisa/env
mkdir -p $workdir
◇
```

Fragment referenced in 26c.
Uses: `workdir` 25b.

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

⟨ *functions* 24b ⟩ ≡

```
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, 24b, 25a.
Fragment referenced in 23c, 26c.
Defines: `passeer` Never used, `veilig` 26c.

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

⟨ *functions* 25a ⟩ ≡

```
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, 24b, 25a.
Fragment referenced in 23c, 26c.
Uses: `print` 34a, `workdir` 25b.

6.6.1 Count processes in jobs

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

⟨ *init processescounter* 25b ⟩ ≡

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

Fragment referenced in 17a.
Defines: `finishlock` 26ab, `workdir` 24a, 25a.

⟨ *increment the processes-counter* 25c ⟩ ≡

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

Fragment referenced in 17a.
Defines: `countlock` 26a.
Uses: `proccount` 6a.

⟨ *decrement the processes-counter, kill if this was the only process* 26a ⟩ ≡

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

Fragment referenced in 17a.
Uses: `countlock` 25c, `finishlock` 25b, `proccount` 6a.

⟨ *wait for working-processes* 26b ⟩ ≡

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

Fragment referenced in 17a.
Uses: `finishlock` 25b.

6.7 The job management script

6.8 The management script

"../runit" 26c≡

```
#!/bin/bash
source /etc/profile
export PATH=/home/phuijgen/usrlocal/bin/:$PATH
source /home/phuijgen/nlp/Pipeline-NL-Lisa/parameters
cd $root
⟨ initialize sematree 24a ⟩
⟨ get runit options 27b ⟩
⟨ functions 5a, ... ⟩
remove_obsolete_lock runit_runs
runsingle runit_runs
⟨ load stopos module 4b ⟩
⟨ check/create directories 6a ⟩
⟨ count jobs 12a, ... ⟩
⟨ update the stopos pool 8a ⟩
⟨ submit jobs when necessary 14b ⟩
if
  [ $loud ]
then
  ⟨ print summary 27c ⟩
fi
veilig runit_runs
exit
◇
```

Uses: `root` 3b, `veilig` 24b.

⟨ *make scripts executable* 27a ⟩ ≡

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

Fragment defined by 21d, 27a, 40b.
Fragment referenced in 40c.

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* 27b ⟩ ≡

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

Fragment referenced in 26c.

Print the summary:

⟨ *print summary* 27c ⟩ ≡

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

Fragment referenced in 26c.
Uses: `failcount` 6a, `incount` 6a, `jobs_to_be_submitted` 14b, `logcount` 6a, `proccount` 6a, `running_jobs` 12c.

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.

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* 28 ⟩ ≡

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

Fragment defined by 28, 30b, 32bc, 34d, 37a, 39d.
Fragment referenced in 29a.
Uses: `nuweb` 36b.

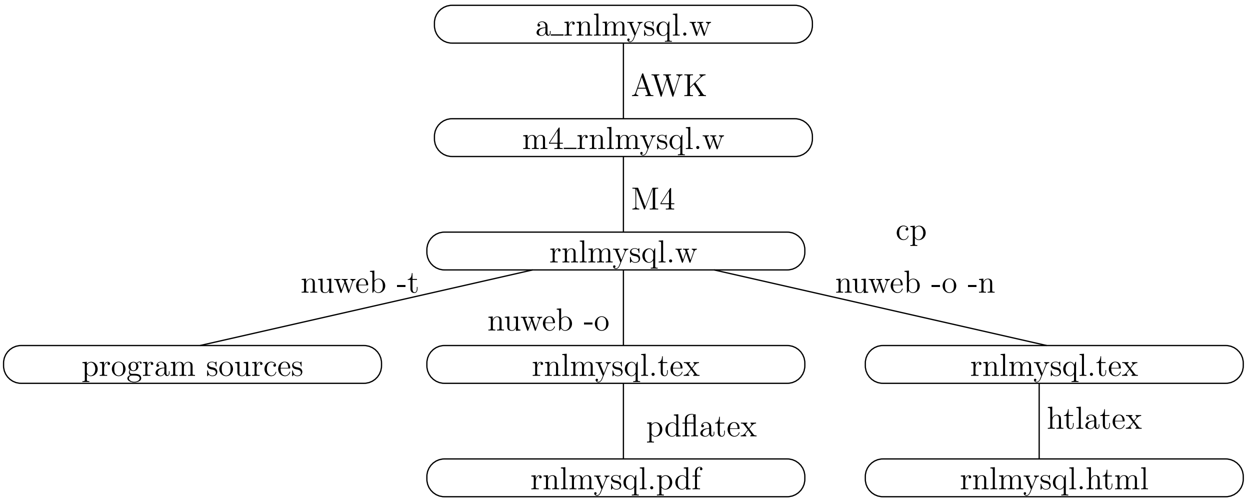

Figure 1: *Translation of the raw code of this document into printable/viewable documents and into program sources. The figure shows the pathways and the main files involved.*

A.3 The Makefile for this project.

This chapter assembles the Makefile for this project.

```
"Makefile" 29a≡
      ⟨ default target 29b ⟩

      ⟨ parameters in Makefile 28, ... ⟩

      ⟨ impliciete make regels 32a, ... ⟩
      ⟨ expliciete make regels 30c, ... ⟩
      ⟨ make targets 29c, ... ⟩
      ◇
```

The default target of make is `all`.

```
⟨ default target 29b ⟩ ≡
      all : ⟨ all targets 30a ⟩
      .PHONY : all

      ◇
```

Fragment referenced in 29a.
Defines: `all` Never used, `PHONY` 33b.

```
⟨ make targets 29c ⟩ ≡
      clean:
               ⟨ clean up 30d ⟩

      ◇
```

Fragment defined by 29c, 34ab, 37e, 40ac.
Fragment referenced in 29a.

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

⟨ *all targets* 30a ⟩ ≡

```
Pipeline_NL_Lisa.pdf◇
```

Fragment referenced in 29b.
Uses: `pdf` 34a.

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* 30b ⟩ ≡

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

◇
```

Fragment defined by 28, 30b, 32bc, 34d, 37a, 39d.
Fragment referenced in 29a.
Defines: `SUFFIXES` Never used.
Uses: `pdf` 34a.

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* 30c ⟩ ≡

```

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 30c, 31abc, 33b, 35a, 37bd.
Fragment referenced in 29a.
Uses: `nuweb` 36b.

⟨ *clean up* 30d ⟩ ≡

```
rm -rf ../nuweb-1.58
◇
```

Fragment referenced in 29c.
Uses: `nuweb` 36b.

⟨ *expliciete make regels* 31a ⟩ ≡

```
../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 30c, 31abc, 33b, 35a, 37bd.
Fragment referenced in 29a.
Uses: `nuweb` 36b.

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* 31b ⟩ ≡

```
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 30c, 31abc, 33b, 35a, 37bd.
Fragment referenced in 29a.
Uses: `print` 34a.

A.5.2 Run the M4 pre-processor

⟨ *expliciete make regels* 31c ⟩ ≡

```
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 30c, 31abc, 33b, 35a, 37bd.
Fragment referenced in 29a.

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* 32a ⟩ ≡

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

◇
```

Fragment defined by 32a, 33a, 37c.
Fragment referenced in 29a.
Uses: `pdf` 34a.

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* 32b ⟩ ≡

```
FIGFILES=fileschema directorystructure

◇
```

Fragment defined by 28, 30b, 32bc, 34d, 37a, 39d.
Fragment referenced in 29a.
Defines: `FIGFILES` 32c.

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* 32c ⟩ ≡

```
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 28, 30b, 32bc, 34d, 37a, 39d.
Fragment referenced in 29a.
Defines: `FIGFILENAMES` Never used, `PDFT_NAMES` 34b, `PDF_FIG_NAMES` 34b, `PST_NAMES` Never used,
`PS_FIG_NAMES` Never used.
Uses: `FIGFILES` 32b.

Create the graph files with program `fig2dev`:

⟨ *impliciete make regels* 33a ⟩ ≡

```
%.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 32a, 33a, 37c.
Fragment referenced in 29a.
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* 33b ⟩ ≡

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

.PHONY : bibfile
◇
```

Fragment defined by 30c, 31abc, 33b, 35a, 37bd.
Fragment referenced in 29a.
Uses: `PHONY` 29b.

A.6.3 Create a printable/viewable document

Make a PDF document for printing and viewing.

⟨ *make targets* 34a ⟩ ≡

```
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 29c, 34ab, 37e, 40ac.
Fragment referenced in 29a.
Defines: `pdf` 30ab, 32a, 34b, `print` 6j, 9ab, 10a, 12c, 16a, 25a, 31b, `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* 34b ⟩ ≡

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

◇
```

Fragment defined by 29c, 34ab, 37e, 40ac.
Fragment referenced in 29a.
Uses: `pdf` 34a, `PDFT_NAMES` 32c, `PDF_FIG_NAMES` 32c.

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* 34c ⟩ ≡

```
../nuweb/bin ◇
```

Fragment referenced in 40a.
Uses: `nuweb` 36b.

⟨ *parameters in Makefile* 34d ⟩ ≡

```
W2PDF=../nuweb/bin/w2pdf
◇
```

Fragment defined by 28, 30b, 32bc, 34d, 37a, 39d.
Fragment referenced in 29a.
Uses: `nuweb` 36b.

⟨ *expliciete make regels* 35a ⟩ ≡

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

Fragment defined by 30c, 31abc, 33b, 35a, 37bd.
Fragment referenced in 29a.

"../nuweb/bin/w2pdf" 35b≡

```
#!/bin/bash
# w2pdf -- compile a nuweb file
# usage: w2pdf [filename]
# 20160301 at 1540h: Generated by nuweb from a_Pipeline_NL_Lisa.w
NUWEB=../env/bin/nuweb
LATEXCOMPILER=pdflatex
⟨ filenames in nuweb compile script 35d ⟩
⟨ compile nuweb 35c ⟩

◇
```

Uses: `nuweb` 36b.

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* 35c ⟩ ≡

```
NUWEB=/home/phuijgen/nlp/Pipeline-NL-Lisa/env/bin/nuweb
⟨ run the processors until the aux file remains unchanged 36c ⟩
⟨ remove the copy of the aux file 36a ⟩
◇
```

Fragment referenced in 35b.
Uses: `nuweb` 36b.

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* 35d ⟩ ≡

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

Fragment referenced in 35b.
Defines: `auxfil` 36c, 38c, 39a, `indexfil` 36c, 38c, `nufil` 36b, 38c, 39b, `oldaux` 36ac, 38c, 39a, `oldindexfil` 36c, 38c, `texfil` 36b, 38c, 39b, `trunk` 36b, 38c, 39bc.

Remove the old copy if it is no longer needed.

⟨ *remove the copy of the aux file* 36a ⟩ ≡

```
rm $oldaux
◇
```

Fragment referenced in 35c, 38b.
Uses: `oldaux` 35d, 38c.

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* 36b ⟩ ≡

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

Fragment referenced in 36c.
Defines: `bibtex` 39bc, `makeindex` 39bc, `nuweb` 28, 30cd, 31a, 34cd, 35bc, 37a, 38a.
Uses: `nufil` 35d, 38c, `texfil` 35d, 38c, `trunk` 35d, 38c.

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* 36c ⟩ ≡

```
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 36b ⟩
  if [ $LOOPCOUNTER -ge 10 ]
  then
    cp $auxfil $oldaux
  fi;
done
◇
```

Fragment referenced in 35c.
Uses: `auxfil` 35d, 38c, `indexfil` 35d, `oldaux` 35d, 38c, `oldindexfil` 35d.

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* 37a ⟩ ≡

```
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 28, 30b, 32bc, 34d, 37a, 39d.
Fragment referenced in 29a.
Uses: `nuweb` 36b.

Make the directory:

⟨ *expliciete make regels* 37b ⟩ ≡

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


◇
```

Fragment defined by 30c, 31abc, 33b, 35a, 37bd.
Fragment referenced in 29a.

The rule to copy files in it:

⟨ *impliciete make regels* 37c ⟩ ≡

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


◇
```

Fragment defined by 32a, 33a, 37c.
Fragment referenced in 29a.

Do the work:

⟨ *expliciete make regels* 37d ⟩ ≡

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


◇
```

Fragment defined by 30c, 31abc, 33b, 35a, 37bd.
Fragment referenced in 29a.

Invoke:

⟨ *make targets* 37e ⟩ ≡

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


◇
```

Fragment defined by 29c, 34ab, 37e, 40ac.
Fragment referenced in 29a.

Create a script that performs the translation.

"w2html" 38a≡

```
#!/bin/bash
# w2html -- make a html file from a nuweb file
# usage: w2html [filename]
#  [filename]: Name of the nuweb source file.
# 20160301 at 1540h: 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 38c ⟩

⟨ perform the task of w2html 38b ⟩

◇
```

Uses: nuweb 36b.

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* 38b ⟩ ≡

```
⟨ run the html processors until the aux file remains unchanged 39a ⟩
⟨ remove the copy of the aux file 36a ⟩
◇
```

Fragment referenced in 38a.

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* 38c ⟩ ≡

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

Fragment referenced in 38a.
Defines: `auxfil` 35d, 36c, 39a, `nufil` 35d, 36b, 39b, `oldaux` 35d, 36ac, 39a, `texfil` 35d, 36b, 39b, `trunk` 35d, 36b, 39bc.
Uses: `indexfil` 35d, `oldindexfil` 35d.

⟨ *run the html processors until the aux file remains unchanged* 39a ⟩ ≡

```
while
  ! cmp -s $auxfil $oldaux
do
  if [ -e $auxfil ]
  then
   cp $auxfil $oldaux
  fi
  ⟨ run the html processors 39b ⟩
done
⟨ run tex4ht 39c ⟩

◇
```

Fragment referenced in 38b.
Uses: `auxfil` 35d, 38c, `oldaux` 35d, 38c.

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

⟨ *run the html processors* 39b ⟩ ≡

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

Fragment referenced in 39a.
Uses: `bibtex` 36b, `makeindex` 36b, `nufil` 35d, 38c, `texfil` 35d, 38c, `trunk` 35d, 38c.

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* 39c ⟩ ≡

```
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 39a.
Uses: `bibtex` 36b, `makeindex` 36b, `trunk` 35d, 38c.

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* 39d ⟩ ≡

```
MKDIR = mkdir -p

◇
```

Fragment defined by 28, 30b, 32bc, 34d, 37a, 39d.
Fragment referenced in 29a.
Defines: `MKDIR` 40a.

⟨ *make targets* 40a ⟩ ≡

```
      DIRS = ⟨ directories to create 34c ⟩

      $(DIRS) :
              $(MKDIR) $@

      ◇
```

Fragment defined by 29c, 34ab, 37e, 40ac.

Fragment referenced in 29a.

Defines: `DIRS` 40c.

Uses: `MKDIR` 39d.

⟨ *make scripts executable* 40b ⟩ ≡

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

Fragment defined by 21d, 27a, 40b.

Fragment referenced in 40c.

⟨ *make targets* 40c ⟩ ≡

```
      source : Pipeline_NL_Lisa.w $(DIRS) $(NUWEB)
              $(NUWEB) Pipeline_NL_Lisa.w
              ⟨ make scripts executable 21d, ... ⟩

      ◇
```

Fragment defined by 29c, 34ab, 37e, 40ac.

Fragment referenced in 29a.

Uses: `DIRS` 40a.

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