NLP-annotate documents on Surfsaras Lisa computer'

Paul Huygen <paul.huygen@huygen.nl>

14th December 2018
07:33 h.

Abstract

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

Contents

1 Introduction

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

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

1.1 How to use it

Quick user instruction:

1. Get an account on Lisa.
2. Clone the software from Github. This results in a directory-tree with root `Pipeline_NL_Lisa`.

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

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

1.2 How it works

1.2.1 Moving files around

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

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

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

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

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

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

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

⟨ *parameters* 3 ⟩ ≡
```
export root=/home/phuijgen/nlp/test/Pipeline-NL-Lisa
export intray=/home/phuijgen/nlp/test/Pipeline-NL-Lisa/data/in
export proctray=/home/phuijgen/nlp/test/Pipeline-NL-Lisa/data/proc
export outtray=/home/phuijgen/nlp/test/Pipeline-NL-Lisa/data/out
export failtray=/home/phuijgen/nlp/test/Pipeline-NL-Lisa/data/fail
export logtray=/home/phuijgen/nlp/test/Pipeline-NL-Lisa/data/log
◇
```
Fragment defined by 3, 7b, 8d, 11b, 14b, 16b, 17b, 20b.
Fragment referenced in 4.
Defines: `failtray` 6afl, 21c, `intray` 6bkl, 9b, 10c, 21b, `logtray` 6ahl, `outtray` 6al, `root` 8a, 9c, 13b, 21b, 25c.

1.2.2 Management-script

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

1.2.3 The NLP-modules

In the annotation process a series of NLP modules operate in sequence on the NAF. The annotation-process is described in section 7.2.

1.2.4 Set parameters

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

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

2 File management

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

2.1 Move NAF-files around

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

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

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

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

⟨ *functions* 5a ⟩ ≡

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

export -f movetotray

◇
```

Fragment defined by 5ab, 23b, 24a.
Fragment referenced in 22b, 25c.
Defines: `movetotray` 10c, 21bc.

⟨ *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, 23b, 24a.
Fragment referenced in 22b, 25c.
Defines: `copytotray` Never used.

2.2 Count the files and manage directories

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

⟨ *check/create directories* 6a ⟩ ≡
```
mkdir -p $outtray
mkdir -p $failtray
mkdir -p $logtray
mkdir -p $proctray
⟨ count files in tray (6b intray,6c incount ) 6j ⟩
⟨ count files in tray (6d proctray,6e proccount ) 6j ⟩
⟨ count files in tray (6f failtray,6g failcount ) 6j ⟩
⟨ count files in tray (6h logtray,6i logcount ) 6j ⟩
unreadycount=$((incount + $proccount))
◇
```
Fragment defined by 6ak.
Fragment referenced in 25c.
Uses: logcount 6a.

⟨ *count files in tray* 6j ⟩ ≡
```
@2=‘find $@1 -type f -print | wc -l‘
◇
```
Fragment referenced in 6a.
Uses: print 33a.

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

⟨ *check/create directories* 6k ⟩ ≡
```
find $intray -depth -type d -empty -delete
find $proctray -depth -type d -empty -delete
mkdir -p $intray
mkdir -p $proctray
◇
```
Fragment defined by 6ak.
Fragment referenced in 25c.
Uses: intray 3.

2.3 Generate pathnames

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

⟨ *generate filenames* 6l ⟩ ≡
```
filtrunk=${infile##$intray/}
export outfile=$outtray/"${filtrunk}"
export failfile=$failtray/"${filtrunk}"
export logfile=$logtray/"${filtrunk}"
export procfile=$proctray/"${filtrunk}"
export outpath=${outfile%/*}
export procpath=${procfile%/*}
export logpath=${logfile%/*}
◇
```
Fragment referenced in 12a.
Defines: filtrunk Never used, logfile 21b, logpath 21b, outfile 12a, 21c, outpath 21bc, procfile 21bc, procpath Never used.
Uses: failtray 3, intray 3, logtray 3, outtray 3.

2.4 Manage list of files in Stopos

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

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

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

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

Fragment referenced in 22b, 25c.
Defines: `module` Never used, `stopos` 9a, 10a, 11ac, 12b.

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

⟨ *parameters* 7b ⟩ ≡
```
export stopospool=magicpool2
```
◇

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

2.4.1 Set up/reset pool

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

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

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

⟨ *update the stopos pool* 8a ⟩ ≡
```
cd $root
⟨ is the pool full or empty? (8b pool_full,8c pool_empty ) 9a ⟩
if
  [ $pool_full -ne 0 ]
then
  ⟨ make a list of filenames in the intray 9b ⟩
  ⟨ decide whether to renew the stopos-pool 9c ⟩
  ⟨ clean up pool and old.filenames 10a ⟩
  ⟨ clean up proctray 10c ⟩
  ⟨ add new filenames to the pool 11a ⟩
fi
◇
```
Fragment referenced in 25c.
Uses: `pool_empty` 8a.

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

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

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 or it does not exist at all..

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

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

⟨ *is the pool full or empty?* 9a ⟩ ≡

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

◇

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

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

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

◇

Fragment referenced in 8a.
Defines: `infilelist` 10abc, 11a.
Uses: `intray` 3, `print` 33a.

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

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

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

◇

Fragment referenced in 8a.
Defines: `regen_pool_condition` 10a.
Uses: `incount` 6a, `lisa_jobcount` 14a, `pool_empty` 8a, `root` 3.

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

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

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

◇
```

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

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

⟨ *clean up old.infilelist* 10b ⟩ ≡

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

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

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

⟨ *clean up proctray* 10c ⟩ ≡

```
if
  [ $running_jobs -eq 0 ]
then
  find $proctray -type f -print | sort >oldprocfilelist
else
  find $proctray -type f -cmin +$maxproctime -print | sort  >oldprocfilelist
fi
cat oldprocfilelist | xargs -iaap  bash -c 'movetotray aap $proctray $intray'
gawk '{gsub(PROCTRAY, INTRAY); print}' PROCTRAY=$proctray IN-
TRAY=$intray <oldprocfilelist >>infilelist
◇
```

Fragment referenced in 8a.
Uses: `infilelist` 9b, `intray` 3, `maxproctime` 11b, `movetotray` 5a, `print` 33a, `running_jobs` 14a.

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

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

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

2.4.2 Get a filename from the pool

To get a filename from Stopos, perform:

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

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

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

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

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

2.4.3 Function to get a filename from Stopos

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

⟨ *functions in the jobfile* 12a ⟩ ≡

```
function getfile() {
  infile=""
  outfile=""
  ⟨ get next infile from stopos 11c ⟩
  if
    [ ! "$infile" == "" ]
  then
    ⟨ generate filenames 6l ⟩
  fi
}
```

◇

Fragment defined by 12a, 19ad.
Fragment referenced in 22b.
Defines: `getfile` 18b.
Uses: `outfile` 6l.

2.4.4 Remove a filename from Stopos

⟨ *remove the infile from the stopos pool* 12b ⟩ ≡

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

◇

Fragment referenced in 21c.
Uses: `stopos` 7a, `stopospool` 7b.

3 Jobs

3.1 Manage the jobs

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

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

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

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

⟨ *count jobs* 13a ⟩ ≡
```
if
  [ -e jobcounter ]
then
  export my_jobcount=‘cat jobcounter‘
else
  my_jobcount=0
fi
◇
```
Fragment defined by 13ab, 14a.
Fragment referenced in 25c.
Uses: `my_jobcount` 13b.

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

⟨ *count jobs* 13b ⟩ ≡
```
cd $root
finished_jobs=‘ls -1 slurm-*.out | wc -l‘
mkdir -p joblogs
mv slurm-*.out joblogs/
if
  [ $finished_jobs -gt $my_jobcount ]
then
  my_jobcount=0
else
  my_jobcount=$((my_jobcount - $finished_jobs))
fi
◇
```
Fragment defined by 13ab, 14a.
Fragment referenced in 25c.
Defines: `my_jobcount` 13a, 15c, 26c.
Uses: `root` 3.

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

The command `squeue` produces a list of jobs. Example of it's output:

```
phuijgen@login2:~/nlp/test$ squeue | head
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
          42307_21     short PLINK_CH jakalman CG       0:04      1 r26n4
          42307_15     short PLINK_CH jakalman CG       0:04      1 r25n11
          42307_18     short PLINK_CH jakalman CG       0:05      1 r27n2
          42307_12     short PLINK_CH jakalman CG       0:04      1 r26n29
```

It seems that `squeue` produces a table with the following columns:

Jobid: Job ID
Partition: Class of jobs in which the job has been classified.
Name: Name of the job
User:
St: Job status. when the job runs, the status seems to be `R`.
Time: Probably elapsed time since the job started to run.

Nodes:
Nodelist (reason): Don't know.

```
phuijgen@login2:~/nlp/test/test$ sbatch -J 'apekop' testjob
JOBID     PARTITION  NAME     USER ST       TIME  NODES NODELIST(REASON)
87294_63  short      PLINK_CH jakalman CG   0:02      1 r26n17
87452     short      apekop   phuijgen PD   0:00      1 (Resources)
```

This package will give submitted jobs the name `magicplace_2`. The following code-piece extracts the lines about jobs with this name from a job-report and counts the number of jobs and of the number of running jobs.

⟨ *count jobs* 14a ⟩ ≡

```
joblist=‘mktemp -t jobrep.XXXXXX‘
rm -rf $joblist
squeue | gawk '$3=="magicplace_2" {print}' > $joblist
lisa_jobcount=‘wc -l <$joblist‘
running_jobs=‘gawk '$5=="R" {runners++}; END {print runners}' $joblist‘
rm -rf $joblist
```

◇

Fragment defined by 13ab, 14a.
Fragment referenced in 25c.
Defines: `joblist` Never used, `lisa_jobcount` 9c, 14c, 26c, `running_jobs` 10c, 26c.
Uses: `print` 33a.

Currently we aim at one job per 100 waiting files.

⟨ *parameters* 14b ⟩ ≡

```
filesperjob=100
```

◇

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

Calculate the number of jobs that have to be submitted.

⟨ *determine how many jobs have to be submitted* 14c ⟩ ≡

```
⟨ determine number of jobs that we want to have 15a, ... ⟩
jobs_to_be_submitted=$((jobs_needed - $lisa_jobcount))
```

◇

Fragment referenced in 15c.
Defines: `jobs_to_be_submitted` 15cd, 26c.
Uses: `jobs_needed` 15ac, `lisa_jobcount` 14a.

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* 15a ⟩ ≡
```
jobs_needed=$((unreadycount / $filesperjob))
if
  [ $unreadycount -gt 0 ] && [ $jobs_needed -eq 0 ]
then
  jobs_needed=1
fi
```
◇
Fragment defined by 15ab.
Fragment referenced in 14c.
Defines: `jobs_needed` 14c, 15b.
Uses: `filesperjob` 14b, `unreadycount` 6a.

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

⟨ *determine number of jobs that we want to have* 15b ⟩ ≡
```
if
  [ $jobs_needed -gt 200 ]
then
  jobs_needed=200
fi
```
◇
Fragment defined by 15ab.
Fragment referenced in 14c.
Uses: `jobs_needed` 15ac.

⟨ *submit jobs when necessary* 15c ⟩ ≡
```
⟨ determine how many jobs have to be submitted 14c ⟩
if
  [ $jobs_to_be_submitted -gt 0 ]
then
   ⟨ submit jobs (15d $jobs_to_be_submitted ) 16a ⟩
   my_jobcount=$((my_jobcount + $jobs_to_be_submitted))
fi
echo $my_jobcount > jobcounter
```
◇
Fragment referenced in 25c.
Uses: `jobs_to_be_submitted` 14c, 15c.

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.

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

⟨ *submit jobs* 16a ⟩ ≡
```
if
  [ @1 -gt 1 ]
then
  sbatch -J magicplace_2 -a 1-@1 magicplace_2
else
  sbatch -J magicplace_2 magicplace_2
fi
```
◇

Fragment referenced in 15c.

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* 16b ⟩ ≡
```
export timelogfile=/home/phuijgen/nlp/test/Pipeline-NL-Lisa/data/log/timelog
```
◇

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

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

Fragment referenced in 16df, 18b.

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

Fragment referenced in 22b.

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

Fragment referenced in 22b.

5 Processes

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

5.1 Calculate the number of parallel processes to be launched

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

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

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

⟨ *determine amount of memory and nodes* 17a ⟩ ≡
```
export ncores=‘sara-get-num-cores‘
export memory=‘sara-get-mem-size‘
```
◇

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

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

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

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

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

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

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

5.2 Start parallel processes

After determining how many parallel processes we can run, start processes as Bash subshells. If it turns out that processes have no work to do, they die. In that case, the job should die too.

Therefore a processes-counter registers the number of running processes. When this has reduced to zero, the macro expires.

⟨ *run parallel processes* 18a ⟩ ≡
```
⟨ determine amount of memory and nodes 17a ⟩
⟨ determine number of parallel processes 17c ⟩
procnum=0
⟨ init processescounter 24c ⟩
for ((i=1 ; i<=$maxprocs ; i++))
do
  ( procnum=$i
    ⟨ increment the processes-counter 24d ⟩
    ⟨ perform the processing loop 18b ⟩
    ⟨ decrement the processes-counter, kill if this was the only process 25a ⟩
  )&
done
⟨ wait for working-processes 25b ⟩
◇
```
Fragment referenced in 22b.
Defines: `procnum` Never used.
Uses: `maxprocs` 17c.

5.3 Perform the processing loop

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

⟨ *perform the processing loop* 18b ⟩ ≡
```
while
   getfile
   [ ! -z "$infile" ]
do
   ⟨ add timelog entry (18c Start $infile ) 16c ⟩
   ⟨ process infile 21b ⟩
   ⟨ add timelog entry (18d Finished $infile with result: $pipelineresult ) 16c ⟩

done
◇
```
Fragment referenced in 18a.
Uses: `pipelineresult` 21b.

6 Servers

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

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

We have the following todo items:

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

6.1 Spotlight server

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

⟨ *functions in the jobfile* 19a ⟩ ≡

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

◇

Fragment defined by 12a, 19ad.
Fragment referenced in 22b.

⟨ *functions in the jobfile* 19d ⟩ ≡

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

◇

Fragment defined by 12a, 19ad.
Fragment referenced in 22b.

⟨ *check spotlight on* 20a ⟩ ≡
```
exec 6<>/dev/tcp/@1/@2
spotlightrunning=$?
exec 6<&-
exec 6>&-
◇
```
Fragment referenced in 19a.
Uses: `spotlightrunning` 19a.

7 Apply the pipeline

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

The pipeline is installed in directory `/home/phuijgen/nlp/nlpp`. Script bin/nlpp applies the pipeline on a NAF file that it reads from standard in.

⟨ *parameters* 20b ⟩ ≡
```
export pipelineroot=/home/phuijgen/nlp/nlpp
export BIND=$pipelineroot/bin
◇
```
Fragment defined by 3, 7b, 8d, 11b, 14b, 16b, 17b, 20b.
Fragment referenced in 4.
Defines: `BIND` 21ab, `pipelineroot` Never used.

7.1 The eSRL server

One of the modules, `eSRL`, is a bit problematic, because it runs as a client-server-system, hence we need to start-up the server. We choose to start up the server beforehand, although it will occupy memory, even if it will never be used in Dutch documents.

The `nlpp` package contans a script, `bin/start_eSRL` that does this. A complication is, that for some reason the script expects either that the variable `naflang` has been set, or that it can read a NAF file from standard in, so that it can determine this variable by itself.

We start the srl server at the beginning of the job and we do not wait intil it runs, hoping that it will be running by the time that the first eSRL client starts its work.

Note that we need to override the location of the "pidfile" that will contain the process-id of the server. The default location is in a subdirectory of the nlpp tree, but we need it to be on the local file-system of the node on which the job runs. Furthermore, we have to override the directory that the semaphore script in nlpp uses to gain or release exclusive access.

⟨ *set local parameters in the job* 20c ⟩ ≡
```
export eSRL_piddir=‘mktemp -d -t eSRL_piddir.XXXXXX‘
export semaworkdir=‘mktemp -d -t sema.XXXXXX‘
◇
```
Fragment referenced in 22b.

⟨ *Start the bloody eSRL server* 21a ⟩ ≡
```
piddir=‘mktemp -d -t piddir.XXXXXXX‘
( export naflang="en" ; $BIND/start_eSRL $piddir ) &
◇
```
Fragment referenced in 22b.
Defines: `piddir` Never used.
Uses: `BIND` 20b.

7.2 Perform the annotation on an input NAF

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

⟨ *process infile* 21b ⟩ ≡
```
export nlppscript=$BIND/nlpp
movetotray "$infile" "$intray" "$proctray"
mkdir -p $outpath
mkdir -p $logpath
export TEMPRES=‘mktemp -t tempout.XXXXXX‘
moduleresult=0
timeout 1500 bash -c "(cat \"$procfile\" | $nlppscript >$TEMPRES 2>\"$logfile\")"
pipelineresult=$?
⟨ move the processed naf around 21c ⟩
cd $root
rm -f $TEMPRES
◇
```
Fragment referenced in 18b.
Defines: `pipelineresult` 18d, 21c, `timeout` Never used.
Uses: `BIND` 20b, `intray` 3, `logfile` 6l, `logpath` 6l, `movetotray` 5a, `outpath` 6l, `procfile` 6l, `root` 3.

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* 21c ⟩ ≡
```
if
 [ $pipelineresult -eq 0 ]
then
  mkdir -p $outpath
  mv $TEMPRES "$outfile"
  rm "$procfile"
else
  movetotray "$procfile" "$proctray" "$failtray"
fi
⟨ remove the infile from the stopos pool 12b ⟩
◇
```
Fragment referenced in 21b.
Uses: `failtray` 3, `movetotray` 5a, `outfile` 6l, `outpath` 6l, `pipelineresult` 21b, `procfile` 6l.

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

⟨ *set utf-8* 22a ⟩ ≡
```
export LANG=en_US.utf8
export LANGUAGE=en_US.utf8
export LC_ALL=en_US.utf8
```
◇

Fragment referenced in 22b.

7.3 The jobfile template

Now we know what the job has to do, we can generate the script. It executes the functions `passeer` and `veilig`.

The job will be submitted into the SLURM job-control system of Lisa. Documentation of this system can be found in the documentation of Surfsara. There is also a cheat sheet with the differences between the Torque and the SLURM system, that seems more up-to-date.

`"../magicplace_2"` 22b≡
```
#!/bin/sh
#SBATCH --nodes=1
#SBATCH  --time=30
⟨ set local parameters in the job 20c ⟩
source /home/phuijgen/nlp/test/Pipeline-NL-Lisa/parameters
⟨ Start the bloody eSRL server 21a ⟩
export jobname=$SLURM_JOB_NAME
⟨ log that the job starts 16d ⟩
⟨ set utf-8 22a ⟩
⟨ load stopos module 7a ⟩
⟨ functions 5a, ... ⟩
⟨ functions in the jobfile 12a, ... ⟩
check_start_spotlight nl
check_start_spotlight en
echo spotlighthost: $spotlighthost >&2
echo spotlighthost: $spotlighthost
starttime=‘date +%s‘
⟨ run parallel processes 18a ⟩
⟨ log that the job finishes 16f ⟩
exit

```
◇

Uses: `spotlighthost` 19a.

7.4 Synchronisation mechanism

This software allows parallel processes to run simultaneously, which can cause unwanted phenomena like two processes that try to annotate the same input-file at the same time.

In fact, we know of two problems that can occur due to processes running in parallel. The first problem, two processes that pick the same input-file for processing, is prevented by using the “Stopos” utility (see section 2.4). The other parallelisation problem might be, that the `runit` script takes a very long time to complete and in the mean time the script is started again, causing two instances of the script to run at the same time. This situation is not imaginary, because loading Stopos with a huge amount of filenames takes a lot of time.

The script `sematree`, obtained from `http://www.pixelbeat.org/scripts/sematree/` allows “mutex” locking. Inside information learns that sematree is available on Lisa (in `/home/phuijgen/usrlocal/bin/sematree`

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* 23a ⟩ ≡
```
export workdir=/home/phuijgen/nlp/test/Pipeline-NL-Lisa/env
mkdir -p $workdir
```
◇

Fragment referenced in 25c.
Defines: `workdir` 24ac.

Now we can implement functions `passeer` (gain exclusive access), `veilig` (give up access) and `runsingle` (gain immediate exclusive access). The difference between function `passeer` and `runsingle` is, that the former function waits until it can gain exclusive access and the latter tries to get immediate exclusive access and abort the process that called it if the attempt is not successful.

⟨ *functions* 23b ⟩ ≡
```
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, 23b, 24a.
Fragment referenced in 22b, 25c.
Defines: `passeer` Never used, `veilig` 25c.

Occasionally a process applies the `passeer` function, but is aborted before it could apply the `veilig` function. In that case, the resource that has been shielded by the synchronisation mechanism would no longer be available for other processes. To prevent this, the "lock" for that resource must eventually be removed in some other way. The following function `remove_obsolete_lock` removes a lock if it has been present for a long time. The maximum time that a lock is allowed to exist, `max_minutes`, has a default value of 60 minutes.

⟨ *functions* 24a ⟩ ≡

```
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, 23b, 24a.
Fragment referenced in 22b, 25c.
Defines: `remove_obsolete_lock` 24b.
Uses: `print` 33a, `workdir` 23a, 24c.

The following macro, applied in the `runit` script, makes sure that the script will not continue to run when another instance of the script is still running..

⟨ *die if another instance of runit is running* 24b ⟩ ≡

```
remove_obsolete_lock runit_runs
runsingle runit_runs
```

◇

Fragment referenced in 25c.
Uses: `remove_obsolete_lock` 24a.

7.4.1 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* 24c ⟩ ≡

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

◇

Fragment referenced in 18a.
Defines: `finishlock` 25ab, `workdir` 23a, 24a.

⟨ *increment the processes-counter* 24d ⟩ ≡

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

◇

Fragment referenced in 18a.
Defines: `countlock` 25a.
Uses: `proccount` 6a.

⟨ *decrement the processes-counter, kill if this was the only process* 25a ⟩ ≡

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

Fragment referenced in 18a.
Uses: `countlock` 24d, `finishlock` 24c, `proccount` 6a.

⟨ *wait for working-processes* 25b ⟩ ≡

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

Fragment referenced in 18a.
Uses: `finishlock` 24c.

7.5 The management script

`"../runit"` 25c≡

```
#!/bin/bash
source /etc/profile
export PATH=/home/phuijgen/usrlocal/bin/:$PATH
source /home/phuijgen/nlp/test/Pipeline-NL-Lisa/parameters
cd $root
⟨ initialize sematree 23a ⟩
⟨ get runit options 26b ⟩
⟨ functions 5a, ... ⟩
⟨ die if another instance of runit is running 24b ⟩
⟨ load stopos module 7a ⟩
⟨ check/create directories 6a, ... ⟩
⟨ count jobs 13a, ... ⟩
⟨ update the stopos pool 8a ⟩
⟨ submit jobs when necessary 15c ⟩
if
  [ $loud ]
then
  ⟨ print summary 26c ⟩
fi
veilig runit_runs
exit
◇
```

Uses: `root` 3, `veilig` 23b.

⟨ *make scripts executable* 26a ⟩ ≡
```
chmod 775 /home/phuijgen/nlp/test/Pipeline-NL-Lisa/runit
```
◇

Fragment defined by 26a, 39b.
Fragment referenced in 39c.

7.6 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* 26b ⟩ ≡
```
OPTIND=1
export loud=0
while getopts "s:" opt; do
    case "$opt" in
    s)  loud=
        ;;
    esac
done
shift $((OPTIND-1))
```
◇

Fragment referenced in 25c.

Print the summary:

⟨ *print summary* 26c ⟩ ≡
```
echo "in            : $incount"
echo "proc          : $proccount"
echo "failed        : $failcount"
echo "processed     : $((logcount - $failcount))"
echo "jobs (Lisa)   : $lisa_jobcount"
echo "jobs (shad)   : $my_jobcount"
echo "running jobs : $running_jobs"
echo "submitted     : $jobs_to_be_submitted"
if
  [ ! "$jobid" == "" ]
then
  echo "job-id      : $jobid"
fi
```
◇

Fragment referenced in 25c.
Uses: `failcount` 6a, `incount` 6a, `jobs_to_be_submitted` 14c, 15c, `lisa_jobcount` 14a, `logcount` 6a, `my_jobcount` 13b, `proccount` 6a, `running_jobs` 14a.

A How to read and translate this document

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

A.1 Read this document

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

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

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

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

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

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

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

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

A.2 Process the document

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

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

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

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

⟨ *parameters in Makefile* 27 ⟩ ≡
```
NUWEB=../env/bin/nuweb
```
◇

Fragment defined by 27, 29b, 31bc, 33d, 36a, 38d.
Fragment referenced in 28a.
Uses: `nuweb` 35b.

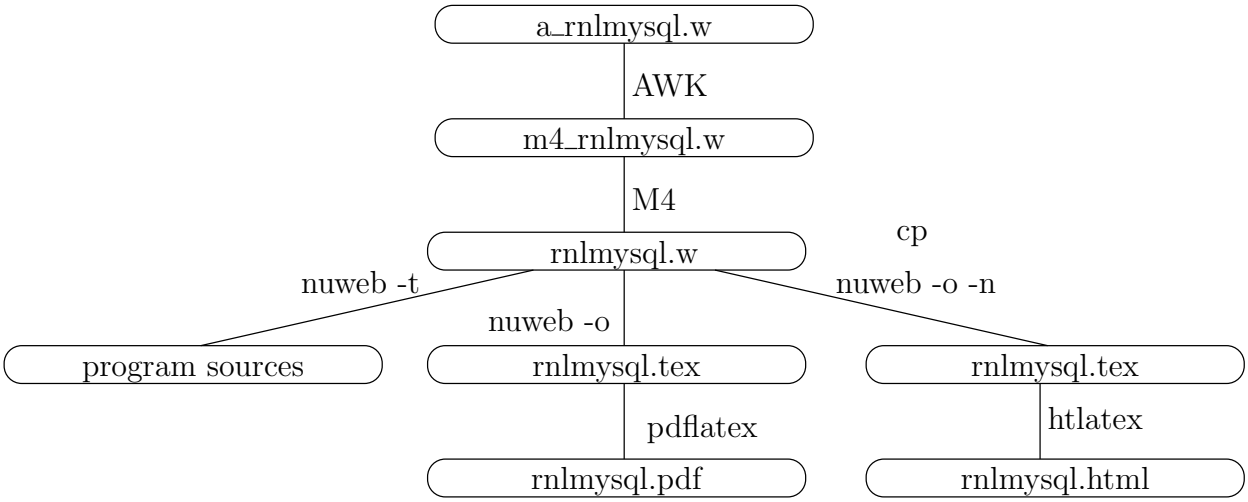

Figure 1: *Translation of the raw code of this document into printable/viewable documents and into program sources. The figure shows the pathways and the main files involved.*

A.3 The Makefile for this project.

This chapter assembles the Makefile for this project.

```
"Makefile" 28a≡
    ⟨ default target 28b ⟩

    ⟨ parameters in Makefile 27, ... ⟩

    ⟨ impliciete make regels 31a, ... ⟩
    ⟨ expliciete make regels 29c, ... ⟩
    ⟨ make targets 28c, ... ⟩
    ◇
```

The default target of make is `all`.

```
⟨ default target 28b ⟩ ≡
    all : ⟨ all targets 29a ⟩
    .PHONY : all

    ◇
```

Fragment referenced in 28a.
Defines: `all` Never used, `PHONY` 32b.

```
⟨ make targets 28c ⟩ ≡
    clean:
        ⟨ clean up 29d ⟩

    ◇
```

Fragment defined by 28c, 33ab, 36e, 39ac.
Fragment referenced in 28a.

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

⟨ *all targets* 29a ⟩ ≡
```
Pipeline_NL_Lisa.pdf◇
```
Fragment referenced in 28b.
Uses: pdf 33a.

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* 29b ⟩ ≡
```
.SUFFIXES: .pdf .w .tex .html .aux .log .php

◇
```
Fragment defined by 27, 29b, 31bc, 33d, 36a, 38d.
Fragment referenced in 28a.
Defines: SUFFIXES Never used.
Uses: pdf 33a.

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* 29c ⟩ ≡
```
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 29c, 30abc, 32b, 34a, 36bd.
Fragment referenced in 28a.
Uses: nuweb 35b.

⟨ *clean up* 29d ⟩ ≡
```
rm -rf ../nuweb-1.58
◇
```
Fragment referenced in 28c.
Uses: nuweb 35b.

⟨ *expliciete make regels* 30a ⟩ ≡

```
../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 29c, 30abc, 32b, 34a, 36bd.
Fragment referenced in 28a.
Uses: `nuweb` 35b.

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* 30b ⟩ ≡

```
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 29c, 30abc, 32b, 34a, 36bd.
Fragment referenced in 28a.
Uses: `print` 33a.

A.5.2 Run the M4 pre-processor

⟨ *expliciete make regels* 30c ⟩ ≡

```
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 29c, 30abc, 32b, 34a, 36bd.
Fragment referenced in 28a.

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* 31a ⟩ ≡
```
%.pdf: %.w
        ./w2pdf $<
```
◇

Fragment defined by 31a, 32a, 36c.
Fragment referenced in 28a.
Uses: pdf 33a.

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* 31b ⟩ ≡
```
FIGFILES=fileschema directorystructure
```
◇

Fragment defined by 27, 29b, 31bc, 33d, 36a, 38d.
Fragment referenced in 28a.
Defines: FIGFILES 31c.

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* 31c ⟩ ≡
```
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 27, 29b, 31bc, 33d, 36a, 38d.
Fragment referenced in 28a.
Defines: FIGFILENAMES Never used, PDFT_NAMES 33b, PDF_FIG_NAMES 33b, PST_NAMES Never used, PS_FIG_NAMES Never used.
Uses: FIGFILES 31b.

Create the graph files with program `fig2dev`:

⟨ *impliciete make regels* 32a ⟩ ≡

```
%.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 31a, 32a, 36c.
Fragment referenced in 28a.
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* 32b ⟩ ≡

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

.PHONY : bibfile
```

◇

Fragment defined by 29c, 30abc, 32b, 34a, 36bd.
Fragment referenced in 28a.
Uses: `PHONY` 28b.

A.6.3 Create a printable/viewable document

Make a PDF document for printing and viewing.

⟨ *make targets* 33a ⟩ ≡
```
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 28c, 33ab, 36e, 39ac.
Fragment referenced in 28a.
Defines: `pdf` 29ab, 31a, 33b, `print` 6j, 9b, 10c, 14a, 24a, 30b, `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* 33b ⟩ ≡
```
Pipeline_NL_Lisa.pdf : Pipeline_NL_Lisa.w $(W2PDF)  $(PDF_FIG_NAMES) $(PDFT_NAMES)
        chmod 775 $(W2PDF)
        $(W2PDF) $*

◇
```
Fragment defined by 28c, 33ab, 36e, 39ac.
Fragment referenced in 28a.
Uses: `pdf` 33a, `PDFT_NAMES` 31c, `PDF_FIG_NAMES` 31c.

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* 33c ⟩ ≡
```
../nuweb/bin ◇
```
Fragment referenced in 39a.
Uses: `nuweb` 35b.

⟨ *parameters in Makefile* 33d ⟩ ≡
```
W2PDF=../nuweb/bin/w2pdf
◇
```
Fragment defined by 27, 29b, 31bc, 33d, 36a, 38d.
Fragment referenced in 28a.
Uses: `nuweb` 35b.

⟨ *expliciete make regels* 34a ⟩ ≡
```
$(W2PDF) : Pipeline_NL_Lisa.w $(NUWEB)
        $(NUWEB) Pipeline_NL_Lisa.w
```
◇
Fragment defined by 29c, 30abc, 32b, 34a, 36bd.
Fragment referenced in 28a.

"../nuweb/bin/w2pdf" 34b≡
```
#!/bin/bash
# w2pdf -- compile a nuweb file
# usage: w2pdf [filename]
# 20181214 at 0733h: Generated by nuweb from a_Pipeline_NL_Lisa.w
NUWEB=../env/bin/nuweb
LATEXCOMPILER=pdflatex
⟨ filenames in nuweb compile script 34d ⟩
⟨ compile nuweb 34c ⟩

```
◇
Uses: nuweb 35b.

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* 34c ⟩ ≡
```
NUWEB=/home/phuijgen/nlp/test/Pipeline-NL-Lisa/env/bin/nuweb
⟨ run the processors until the aux file remains unchanged 35c ⟩
⟨ remove the copy of the aux file 35a ⟩
```
◇
Fragment referenced in 34b.
Uses: nuweb 35b.

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* 34d ⟩ ≡
```
nufil=$1
trunk=${1%%.*}
texfil=${trunk}.tex
auxfil=${trunk}.aux
oldaux=old.${trunk}.aux
indexfil=${trunk}.idx
oldindexfil=old.${trunk}.idx
```
◇
Fragment referenced in 34b.
Defines: auxfil 35c, 37c, 38a, indexfil 35c, 37c, nufil 35b, 37c, 38b, oldaux 35ac, 37c, 38a, oldindexfil 35c, 37c, texfil 35b, 37c, 38b, trunk 35b, 37c, 38bc.

Remove the old copy if it is no longer needed.

⟨ *remove the copy of the aux file* 35a ⟩ ≡
```
rm $oldaux
```
◇

Fragment referenced in 34c, 37b.
Uses: `oldaux` 34d, 37c.

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* 35b ⟩ ≡
```
$NUWEB $nufil
$LATEXCOMPILER $texfil
makeindex $trunk
bibtex $trunk
```
◇

Fragment referenced in 35c.
Defines: `bibtex` 38bc, `makeindex` 38bc, `nuweb` 27, 29cd, 30a, 33cd, 34bc, 36a, 37a.
Uses: `nufil` 34d, 37c, `texfil` 34d, 37c, `trunk` 34d, 37c.

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* 35c ⟩ ≡
```
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 35b ⟩
  if [ $LOOPCOUNTER -ge 10 ]
  then
    cp $auxfil $oldaux
  fi;
done
```
◇

Fragment referenced in 34c.
Uses: `auxfil` 34d, 37c, `indexfil` 34d, `oldaux` 34d, 37c, `oldindexfil` 34d.

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* 36a ⟩ ≡
```
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 27, 29b, 31bc, 33d, 36a, 38d.
Fragment referenced in 28a.
Uses: `nuweb` 35b.

Make the directory:

⟨ *expliciete make regels* 36b ⟩ ≡
```
$(htmldir) :
        mkdir -p $(htmldir)
```
◇

Fragment defined by 29c, 30abc, 32b, 34a, 36bd.
Fragment referenced in 28a.

The rule to copy files in it:

⟨ *impliciete make regels* 36c ⟩ ≡
```
$(htmldir)/% : % $(htmldir)
        cp $< $(htmldir)/
```
◇

Fragment defined by 31a, 32a, 36c.
Fragment referenced in 28a.

Do the work:

⟨ *expliciete make regels* 36d ⟩ ≡
```
$(htmltarget) : $(htmlmaterial) $(htmldir)
        cd $(htmldir) && chmod 775 w2html
        cd $(htmldir) && ./w2html nlpp.w
```
◇

Fragment defined by 29c, 30abc, 32b, 34a, 36bd.
Fragment referenced in 28a.

Invoke:

⟨ *make targets* 36e ⟩ ≡
```
htm : $(htmldir) $(htmltarget)
```
◇

Fragment defined by 28c, 33ab, 36e, 39ac.
Fragment referenced in 28a.

Create a script that performs the translation.

```
"w2html" 37a≡
      #!/bin/bash
      # w2html -- make a html file from a nuweb file
      # usage: w2html [filename]
      #  [filename]: Name of the nuweb source file.
      # 20181214 at 0733h: Generated by nuweb from a_Pipeline_NL_Lisa.w
      echo "translate " $1 >w2html.log
      NUWEB=/home/phuijgen/nlp/test/Pipeline-NL-Lisa/env/bin/nuweb
      ⟨ filenames in w2html 37c ⟩

      ⟨ perform the task of w2html 37b ⟩

      ◇
```

Uses: `nuweb` 35b.

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 37b ⟩ ≡
      ⟨ run the html processors until the aux file remains unchanged 38a ⟩
      ⟨ remove the copy of the aux file 35a ⟩
      ◇
```

Fragment referenced in 37a.

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 37c ⟩ ≡
      nufil=$1
      trunk=${1%%.*}
      texfil=${trunk}.tex
      auxfil=${trunk}.aux
      oldaux=old.${trunk}.aux
      indexfil=${trunk}.idx
      oldindexfil=old.${trunk}.idx
      ◇
```

Fragment referenced in 37a.
Defines: `auxfil` 34d, 35c, 38a, `nufil` 34d, 35b, 38b, `oldaux` 34d, 35ac, 38a, `texfil` 34d, 35b, 38b, `trunk` 34d, 35b, 38bc.
Uses: `indexfil` 34d, `oldindexfil` 34d.

⟨ *run the html processors until the aux file remains unchanged* 38a ⟩ ≡

```
while
  ! cmp -s $auxfil $oldaux
do
  if [ -e $auxfil ]
  then
   cp $auxfil $oldaux
  fi
  ⟨ run the html processors 38b ⟩
done
⟨ run tex4ht 38c ⟩

◇
```

Fragment referenced in 37b.
Uses: `auxfil` 34d, 37c, `oldaux` 34d, 37c.

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

⟨ *run the html processors* 38b ⟩ ≡

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

Fragment referenced in 38a.
Uses: `bibtex` 35b, `makeindex` 35b, `nufil` 34d, 37c, `texfil` 34d, 37c, `trunk` 34d, 37c.

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* 38c ⟩ ≡

```
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 38a.
Uses: `bibtex` 35b, `makeindex` 35b, `trunk` 34d, 37c.

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* 38d ⟩ ≡

```
MKDIR = mkdir -p

◇
```

Fragment defined by 27, 29b, 31bc, 33d, 36a, 38d.
Fragment referenced in 28a.
Defines: `MKDIR` 39a.

⟨ *make targets* 39a ⟩ ≡
```
DIRS = ⟨ directories to create 33c ⟩

$(DIRS) :
        $(MKDIR) $@
```
◇
Fragment defined by 28c, 33ab, 36e, 39ac.
Fragment referenced in 28a.
Defines: DIRS 39c.
Uses: MKDIR 38d.

⟨ *make scripts executable* 39b ⟩ ≡
```
chmod -R 775  ../env/bin/*
```
◇
Fragment defined by 26a, 39b.
Fragment referenced in 39c.

⟨ *make targets* 39c ⟩ ≡
```
source : Pipeline_NL_Lisa.w $(DIRS) $(NUWEB)
        $(NUWEB) Pipeline_NL_Lisa.w
        ⟨ make scripts executable 26a, ... ⟩
```
◇
Fragment defined by 28c, 33ab, 36e, 39ac.
Fragment referenced in 28a.
Uses: DIRS 39a.

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

⟨ submit jobs when necessary 15c ⟩ Referenced in 25c.
⟨ update the stopos pool 8a ⟩ Referenced in 25c.
⟨ wait for working-processes 25b ⟩ Referenced in 18a.

C.3 Variables