NLP-annotate documents on Surfsaras Lisa computer'

Paul Huygen <paul.huygen@huygen.nl>

26th December 2018
16:59 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 **2**
- 1.1 How to use it . . . 3
- 1.2 How it works . . . 3
  - 1.2.1 Moving files around . . . 3
  - 1.2.2 Management-script . . . 4
  - 1.2.3 The NLP-modules . . . 4
  - 1.2.4 Set parameters . . . 4

2 File management **4**
- 2.1 Move NAF-files around . . . 4
- 2.2 Count the files and manage directories . . . 5
- 2.3 Generate pathnames . . . 6
- 2.4 Clean up . . . 7
- 2.5 Manage list of files in Stopos . . . 7
  - 2.5.1 Set up/reset pool . . . 7
  - 2.5.2 Get a filename from the pool . . . 12
  - 2.5.3 Function to get a filename from Stopos . . . 13
  - 2.5.4 Remove a filename from Stopos . . . 13

3 Jobs **13**
- 3.1 Manage the jobs . . . 13
  - 3.1.1 Count the jobs . . . 13
  - 3.1.2 Count how many jobs are needed . . . 15
  - 3.1.3 Submit jobs . . . 16

4 Logging **17**
- 4.1 Job logs . . . 17
- 4.2 Time log . . . 18

5 Processes **18**
5.1 Calculate the number of parallel processes to be launched . . . 18
5.2 Start parallel processes . . . 19
5.3 Perform the processing loop . . . 20

6 Servers **20**
6.1 Spotlight server . . . 20

7 Apply the pipeline **22**
7.1 The eSRL server . . . 22
7.2 Perform the annotation on an input NAF . . . 23
7.3 The jobfile template . . . 24
7.4 Synchronisation mechanism . . . 24
7.4.1 Count processes in jobs . . . 26
7.5 The management script . . . 27
7.6 Print a summary . . . 28

A How to read and translate this document **28**
A.1 Read this document . . . 29
A.2 Process the document . . . 29
A.3 The Makefile for this project. . . . 30
A.4 Get Nuweb . . . 31
A.5 Pre-processing . . . 32
A.5.1 Process 'dollar' characters . . . 32
A.5.2 Run the M4 pre-processor . . . 32
A.6 Typeset this document . . . 32
A.6.1 Figures . . . 33
A.6.2 Bibliography . . . 34
A.6.3 Create a printable/viewable document . . . 34
A.6.4 Create HTML files . . . 37
A.7 Create the program sources . . . 40

B References **41**
B.1 Literature . . . 41

C Indexes **41**
C.1 Filenames . . . 41
C.2 Macro's . . . 41
C.3 Variables . . . 43

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

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. 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* 4a ⟩ ≡
```
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 4a, 7c, 9g, 12b, 15j, 18b, 19a, 22b.
Fragment referenced in 4b.
Defines: `failtray` 6afl, 7a, 23c, `intray` 6bkl, 9a, 10b, 11g, 23b, `logtray` 6ahl, 7a, `outtray` 6al, 7a, `root` 9ad, 10bc, 17c, 18a, 23b, 27c.

1.2.2 Management-script

When a user has put NAF files in `data/in`, something has to take care of starting jobs to 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 constructions of these modules and the script that invokes the annotation are not part of this package. In this document we assume that there exists a script named `nlpp`, located in

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"` 4b≡
```
⟨ parameters 4a, ... ⟩
```
◇

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, 25b, 26a.
Fragment referenced in 24b, 27c.
Defines: `movetotray` 11g, 23bc.

⟨ *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, 25b, 26a.
Fragment referenced in 24b, 27c.
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, 17b.
Fragment referenced in 27c.
Uses: logcount 6a.

⟨ *count files in tray* 6j ⟩ ≡
```
@2=‘find $@1 -type f -print | wc -l‘
◇
```
Fragment referenced in 6a.
Uses: print 35a.

The processes empty the directory-structure in the intray and the proctray, but they do not remove empty directories. Therefore, the `runit` script has to do this.

⟨ *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, 17b.
Fragment referenced in 27c.
Uses: intray 4a.

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 13a.
Defines: filtrunk Never used, logfile 23b, logpath 23b, outfile 13a, 23c, outpath 23bc, procfile 23bc, procpath Never used.
Uses: failtray 4a, intray 4a, logtray 4a, outtray 4a.

2.4 Clean up

Not everybody removes her output-files after she copied them to her own directory. Especially log-files tend to escape the attention of the users. Therefore, let us remove log-files and output-files that exist for longer that a month.

⟨ *clean up outputfiles and logfiles* 7a ⟩ ≡
```
find $logtray -ctime +30 -delete
find $outtray -ctime +30 -delete
find $failtray -ctime +30 -delete
```
◇

Fragment referenced in 27c.
Uses: `failtray` 4a, `logtray` 4a, `outtray` 4a.

2.5 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* 7b ⟩ ≡
```
module load stopos
```
◇

Fragment referenced in 24b, 27c.
Defines: `module` Never used, `stopos` 10a, 11a, 12ac, 13b.

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

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

Fragment defined by 4a, 7c, 9g, 12b, 15j, 18b, 19a, 22b.
Fragment referenced in 4b.
Defines: `stopospool` 10a, 11a, 12ac, 13b.

2.5.1 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.

If the Stopos pool contains enough filenames to keep the system working for the next thirty minutes, we leave it as it is.

When we want to add filenames to a running pool, we have to make sure that they are not already listed in the pool. Therefore, we need a shadow-bookkeeping that is accessible to us. When the `runit` script updates the pool it generates a file `old.infilelist` with a sorted listing of the contents of the pool.

The next time that `runit` runs, it generates a list of names of files that are in the intray, but not in the pool and then adds these filenames to the pool and to verb|old.infilelist|. A complication are the files in the proctray. Sometimes they are transferred back to the intray, and recognised as new file of which the name is not yet listed in the pool. Therefore runit starts to remove names of files in the proctray from `old.filenames`.

1. Remove the names of files in the proctray from `old.infilelist`.
2. Check whether the intray is empty. we need to know this later on.
3. Move files in the proctray of which we know that they are no longer being processed back to the intray. That means that, when there are jobs running, all the files in the proctray can be transferred, and otherwise, files that resided in the proctime for a longer time than the maximum lifetime of a job can be transferred.
4. Check whether we can renew the pool (purge its contents and re-fill). We can do that when it is empty or when there are no jobs in the system. When the pool is renewed, its contents is aligned with the shadow-bookkeeping.
5. Generate a sorted file `infilelist` that contains the paths to the files in the intray.
6. Remove the filenames in `old.infilelist` from `infilelist`.
7. Delete from `old.filenames` the names of the files that are no longer in the intray.
8. 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.
9. Add the files in `infilelist` to the pool.
10. Add the content of `infilelist` to `old.infilelist`.

First an aside: One of the tasks that we have to perform is, to compare two (sorted) lists of filenames, and then produce a list of the filenames that are present in the first list, but not in the second. The following macro does this. It has two arguments: 1) The name of the file with all the names, 2) the name of the file with filenames that must be removed. It replaces the file in the first argument with a file that contains the net result. and 3) the name of the file with the net result.

⟨ *subtract filenames* 8a ⟩ ≡

```
subfile=‘mktemp -t subfile.XXXXXX‘
comm -23 @1 @2 >$subfile
mv $subfile @1
◇
```

Fragment referenced in 9a, 11b.
Defines: `subfile` 8b.

⟨ *keep common filenames* 8b ⟩ ≡

```
subfile=‘mktemp -t subfile.XXXXXX‘
comm -12 @1 @2 >$subfile
mv $subfile @1
◇
```

Fragment referenced in 11b.
Uses: `subfile` 8a.

⟨ *subtract files in proctray from shadow bookkeeping* 9a ⟩ ≡

```
cd $root
proclist=‘mktemp -t proclist.XXXXXX‘
gawcomm="{gsub(\"$proctray\", \"$intray\"); print}"
find $proctray -type f -print \
  | gawk "$gawcomm" \
  | sort >$proclist
⟨ subtract filenames (9b old.infilelist,9c $proclist ) 8a ⟩
rm $proclist
◇
```

Fragment referenced in 9d.
Uses: proclist 9a.

⟨ *update the stopos pool* 9d ⟩ ≡

```
cd $root
⟨ is the pool full or empty? (9e pool_full,9f pool_empty ) 10a ⟩
if
  [ $pool_full -ne 0 ]
then
  ⟨ subtract files in proctray from shadow bookkeeping 9a ⟩
  ⟨ clean up proctray 11g ⟩
  ⟨ make a list of filenames in the intray 10b ⟩
  ⟨ decide whether to renew the stopos-pool 10c ⟩
  ⟨ clean up pool and old.filenames 11a ⟩
  ⟨ add new filenames to the pool 12a ⟩
fi
◇
```

Fragment referenced in 27c.
Uses: pool_empty 9d.

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* 9g ⟩ ≡

```
stopos_sufficient_filecount=30000
◇
```

Fragment defined by 4a, 7c, 9g, 12b, 15j, 18b, 19a, 22b.
Fragment referenced in 4b.
Defines: stopos_sufficient_filecount 10a.

⟨ *is the pool full or empty?* 10a ⟩ ≡

```
@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 9d.
Uses: `stopos` 7b, `stopospool` 7c, `stopos_sufficient_filecount` 9g.

⟨ *make a list of filenames in the intray* 10b ⟩ ≡

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

◇

Fragment referenced in 9d.
Defines: `infilelist` 9b, 11acdef, 12a.
Uses: `intray` 4a, `print` 35a, `root` 4a.

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* 10c ⟩ ≡

```
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 9d.
Defines: `regen_pool_condition` 11a.
Uses: `incount` 6a, `lisa_jobcount` 15e, `pool_empty` 9d, `root` 4a.

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* 11a ⟩ ≡

```
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 and infilelist 11b ⟩
fi

◇
```

Fragment referenced in 9d.
Uses: `infilelist` 10b, `regen_pool_condition` 10c, `stopos` 7b, `stopospool` 7c.

Update the content of `old.infilelist`. Names of files that are not in the intray, have probably also been removed from the pool. So remove them from the shadow-bookkeeping as well. After that, remove filenames that are already in `old.infilelist` from `infilelist`. After that, `infilelist` contains the names of new filenames that have to be added to the pool.

⟨ *clean up old.infilelist and infilelist* 11b ⟩ ≡

```
⟨ keep common filenames (11c old.infilelist,11d infilelist ) 8b ⟩
⟨ subtract filenames (11e infilelist,11f old.infilelist ) 8a ⟩
◇
```

Fragment referenced in 11a.
Uses: `infilelist` 10b.

If there exist jobs, move files that resided longer in the proctray than the lifetime of a job back to the in-tray. If there are no jobs, move all files in the proctray back to the intray. Before doing that, remove the names of the files in the proctray from the list of filenames that were stored in the Stopos pool last time (`old.infilelist`).

⟨ *clean up proctray* 11g ⟩ ≡

```
if
  [ $lisa_jobcount -eq 0 ]
then
  find $proctray -type f -print | xargs -iaap  bash -
c 'movetotray aap $proctray $intray'
else
  find $proctray -type f -cmin +$maxproctime -print | xargs -iaap  bash -
c 'movetotray aap $proctray $intray'
fi
◇
```

Fragment referenced in 9d.
Uses: `intray` 4a, `lisa_jobcount` 15e, `maxproctime` 12b, `movetotray` 5a, `print` 35a.

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* 12a ⟩ ≡
```
stopos -p $stopospool add infilelist
cat infilelist old.infilelist | sort >temp.infilelist
mv temp.infilelist old.infilelist
rm infilelist
◇
```
Fragment referenced in 9d.
Uses: `infilelist` 10b, `stopos` 7b, `stopospool` 7c.

⟨ *parameters* 12b ⟩ ≡
```
maxproctime=30
◇
```
Fragment defined by 4a, 7c, 9g, 12b, 15j, 18b, 19a, 22b.
Fragment referenced in 4b.
Defines: `maxproctime` 11g.

2.5.2 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* 12c ⟩ ≡
```
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 13a.
Uses: `stopos` 7b, `stopospool` 7c.

2.5.3 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* 13a ⟩ ≡

```
function getfile() {
  infile=""
  outfile=""
  ⟨ get next infile from stopos 12c ⟩
  if
    [ ! "$infile" == "" ]
  then
    ⟨ generate filenames 6l ⟩
  fi
}

◇
```

Fragment defined by 13a, 21ad.
Fragment referenced in 24b.
Defines: `getfile` 20b.
Uses: `outfile` 6l.

2.5.4 Remove a filename from Stopos

⟨ *remove the infile from the stopos pool* 13b ⟩ ≡

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

Fragment referenced in 23c.
Uses: `stopos` 7b, `stopospool` 7c.

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.

3.1.1 Count the jobs

Find out how many submitted jobs there are and how many of them are actually running. We will make a double bookkeeping. The first system is, to look in the reports from Lisa, as obtained by the instruction `squeue` (see the man-page of `squeue`). The second way is, that we store the number of submitted jobs in a file, add the number of newly submitted jobs to it and subtract the number of finished jobs from it.

Lisa supplies an instruction `squeue` that produces a list of running and waiting jobs. However, in the former job-control system torque I had the experience that the joblisting were not always adequate. Therefore, in the past we made a shadow job bookkeeping as well.

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.

We will submit jobs with job-name `magicplace_2`. The following code-piece extracts the lines with this job-name from a job-report and counts the number of jobs and of the number of running jobs.

Note, that the standard output format of squeue truncates the jobname to eight characters. Therefore we specify a different output-format in `joblist_format`, that extends the size reserved for the jobnames. (See the man page of squeue for details about the output format).

If we would submit every job with a separate statement, we could just count the lines that the output of `squeue` produced. However, the preferred way to submit a large quantity of jobs is, to submit them as an array. In that case the submitted jobs are represented on a single line, and we have to interpret the array boundaries that are appended to the job-id. e.g. a line that begins with `440454_[2-200]` represents 199 jobs. The following macro contains an `AWK` script that counts the jobs from the output of the `squeue` command:

⟨ *gawk command to count jobs from squeue report* 14 ⟩ ≡

```
BEGIN {nrjobs=0}
{ jobid=$1
  if (match(jobid, /\[([0-9]+)-([0-9]+)\]/, arr)) {
            firstjob=arr[1]; lastjob=arr[2]
            jobs_in_array = lastjob -firstjob +1
            nrjobs = nrjobs + jobs_in_array
   } else {
        nrjobs++
   }
}

END { print nrjobs }
```

◇

Fragment referenced in 15a.
Uses: `print` 35a.

Use this script here:

⟨ *count jobs in squeue report* 15a ⟩ ≡
```
@1='gawk '
  ⟨ gawk command to count jobs from squeue report 14 ⟩
  ' @2'
◇
```
Fragment referenced in 15be.

⟨ *count running jobs in squeue report* 15b ⟩ ≡
```
gawk '$5=="R" {print} <@2 >rjoblist
⟨ count jobs in squeue report (15c running_jobs,15d rjoblist ) 15a ⟩
rm -rf rjoblist
◇
```
Fragment never referenced.
Uses: rjoblist 15e.

⟨ *count jobs* 15e ⟩ ≡
```
joblist='mktemp -t jobrep.XXXXXX'
rjoblist='mktemp -t rjobrep.XXXXXX'
rm -rf $joblist
joblist_format="%.18i %.9P %.15j %.8u %.2t %.10M %.6D %R"
squeue -o "$joblist_format" | gawk '$3=="magicplace_2" {print}' > $joblist
gawk '$5=="R" {print}' <$joblist >$rjoblist
⟨ count jobs in squeue report (15f lisa_jobcount,15g $joblist ) 15a ⟩
⟨ count jobs in squeue report (15h running_jobs,15i $rjoblist ) 15a ⟩
rm -rf $joblist
rm -rf $rjoblist
◇
```
Fragment referenced in 27c.
Uses: rjoblist 15e.

3.1.2 Count how many jobs are needed

Let us make an estimation about the number of jobs that are needed the next half hour. Each job generates around 16 parallel processes to annotate files. Suppose annotation of a short file takes four minutes, a processes can annotate 7 files before the process aborts. So, under good circumstances, a single job is able to handle 112 files. Let this make the number round:

⟨ *parameters* 15j ⟩ ≡
```
filesperjob=100
◇
```
Fragment defined by 4a, 7c, 9g, 12b, 15j, 18b, 19a, 22b.
Fragment referenced in 4b.
Defines: filesperjob 16a.

The number of jobs that we need is the quotient of the number of files that still have to be processed and `filesperjob`. We have to make sure that, when there are less than `filesperjob` unready input files, we still need a job to process them. On the other hand, we do not want to flood the place with jobs, so we do not submit more than 200 jobs.

⟨ *determine number of jobs that we want to have* 16a ⟩ ≡
```
jobs_needed=$((unreadycount / $filesperjob))
if
  [ $unreadycount -gt 0 ] && [ $jobs_needed -eq 0 ]
then
  jobs_needed=1
fi
if
  [ $jobs_needed -gt 200 ]
then
  jobs_needed=200
fi
```
◇

Fragment referenced in 16b.
Defines: `jobs_needed` 16b.
Uses: `filesperjob` 15j, `unreadycount` 6a.

Subtract the number of existing jobs from `jobs_needed` to obtain the number of jobs that still have to be submitted.

⟨ *determine how many jobs have to be submitted* 16b ⟩ ≡
```
⟨ determine number of jobs that we want to have 16a ⟩
jobs_to_be_submitted=$((jobs_needed - $lisa_jobcount))
```
◇

Fragment referenced in 16c.
Defines: `jobs_to_be_submitted` 16cd, 28c.
Uses: `jobs_needed` 16ac, `lisa_jobcount` 15e.

3.1.3 Submit jobs

⟨ *submit jobs when necessary* 16c ⟩ ≡
```
⟨ determine how many jobs have to be submitted 16b ⟩
if
  [ $jobs_to_be_submitted -gt 0 ]
then
   ⟨ submit jobs (16d $jobs_to_be_submitted ) 17a ⟩
fi
```
◇

Fragment referenced in 27c.
Uses: `jobs_to_be_submitted` 16bc.

A job executes a "job script" that is constructed in section 7.3. "Submitting a job" means "Hand the job file over to the job-control system". If we want to have multiple jobs, we can submit the job as an array. The slurm job-control system uses instruction `sbatch` to submit jobs. Information about `sbatch` can be found in the Surfsara documentation or on the man page of `sbatch`.

The following macro submits the jobscript as a single job or as an array of multiple jobs. Its argument signifies the number of times that the job has to be submitted.

⟨ *submit jobs* 17a ⟩ ≡

```
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 16c.

4 Logging

There are three kinds of log-files:

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

4.1 Job logs

The names of job-logs have the form `slurm-<id>.out`, where `<id>` represents the job-id to which the file belonged. As soon as a process enters the running state the job-log is created.

To prevent flooding the place with job-log files, the run-script will move old job-logs to sub-directory `logs/joblogs`. There are two improvements that we want to perform in future:

1. Generate a log-rotating mechanism that compresses older logfiles and removes still older logfiles
2. Find out which of the log-files belonged to expired jobs and move only those files. Now we just move files that are older than an hour.

First, make sure that the directory to store the job-logs exists, to prevent an avalanche of error-messages.

⟨ *check/create directories* 17b ⟩ ≡

```
mkdir -p /home/phuijgen/nlp/test/Pipeline-NL-Lisa/logs/joblogs
◇
```

Fragment defined by 6ak, 17b.
Fragment referenced in 27c.
Defines: `joblogs` 17c, 18a.

⟨ *clean up joblogs* 17c ⟩ ≡

```
find $root \
  -maxdepth 1 -name "slurm-[0-9]*.out" \
  -cmin +60 \
  -execdir mv {} logs/joblogs/ \;
◇
```

Fragment defined by 17c, 18a.
Fragment referenced in 27c.
Uses: `joblogs` 17b, `root` 4a.

Remove old job-logs.

⟨ *clean up joblogs* 18a ⟩ ≡
```
find $root/logs/joblogs \
  -name "slurm-[0-9]*.out" \
  -cmin +1440 \
  -delete
```
◇
Fragment defined by 17c, 18a.
Fragment referenced in 27c.
Uses: `joblogs` 17b, `root` 4a.

4.2 Time log

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

⟨ *parameters* 18b ⟩ ≡
```
export timelogfile=/home/phuijgen/nlp/test/Pipeline-NL-Lisa/logs/timelog
```
◇
Fragment defined by 4a, 7c, 9g, 12b, 15j, 18b, 19a, 22b.
Fragment referenced in 4b.

⟨ *add timelog entry* 18c ⟩ ≡
```
echo `date +%s`: @1 >> $timelogfile
```
◇
Fragment referenced in 18df, 20b.

⟨ *log that the job starts* 18d ⟩ ≡
⟨ *add timelog entry* (18e `Start job $jobname`) 18c ⟩
◇
Fragment referenced in 24b.

⟨ *log that the job finishes* 18f ⟩ ≡
⟨ *add timelog entry* (18g `Finish job $jobname`) 18c ⟩
◇
Fragment referenced in 24b.

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

Our jobs run each on a single node. The number of parallel processes that we can set up in a job depends on the number of CPU's and the amount of memory that is available. In order to run effectively, we need at least one CPU and an estimated amount of 3 GB per process. It seemed that we read the number of CPU's and the amount of memory from the variables

SLURM_CPUS_ON_NODE resp. SLURM_MEM_PER_NODE. However, as of this week (december 14, 2018) variable SLURM_MEM_PER_NODE seems to be replaced by SLURM_MEM_PER_CPU. Moreover, command sara-get-mem-size dis not seem to exist today in the morning. In the afternoon the function works and specifies the amount of memory in Gb.

Let us use variable SLURM_MEM_PER_CPU today.

⟨ *parameters* 19a ⟩ ≡
```
required_mem_per_process_gb=3
required_mem_per_process_mb=$((required_mem_per_process_gb * 1000))
◇
```
Fragment defined by 4a, 7c, 9g, 12b, 15j, 18b, 19a, 22b.
Fragment referenced in 4b.
Defines: required_mem_per_process_gb Never used, required_mem_per_process_mb 19b.

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

⟨ *determine number of parallel processes* 19b ⟩ ≡
```
export memory_mb=$((SLURM_MEM_PER_CPU * $SLURM_CPUS_ON_NODE))
export memchunks=$((memory_mb / $required_mem_per_process_mb))
if
  [ $SLURM_CPUS_ON_NODE -gt $memchunks ]
then
  maxprocs=$memchunks
else
  maxprocs=$SLURM_CPUS_ON_NODE
fi

echo "Number of CPU's: $SLURM_CPUS_ON_NODE" >&2
echo "Memory:         : $memory_mb" >&2
echo "Memchunks:      : $memchunks" >&2
echo "calculated maxprocs: $maxprocs" >&2
◇
```
Fragment referenced in 20a.
Defines: maxprocs 20a.
Uses: required_mem_per_process_mb 19a.

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* 20a ⟩ ≡
```
⟨ determine number of parallel processes 19b ⟩
procnum=0
⟨ init processescounter 26c ⟩
for ((i=1 ; i<=$maxprocs ; i++))
do
  ( procnum=$i
    ⟨ increment the processes-counter 26d ⟩
    ⟨ perform the processing loop 20b ⟩
    ⟨ decrement the processes-counter, kill if this was the only process 27a ⟩
  )&
done
⟨ wait for working-processes 27b ⟩
◇
```
Fragment referenced in 24b.
Defines: `procnum` Never used.
Uses: `maxprocs` 19b.

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* 20b ⟩ ≡
```
while
   getfile
   [ ! -z "$infile" ]
do
   ⟨ add timelog entry (20c Start $infile ) 18c ⟩
   ⟨ process infile 23b ⟩
   ⟨ add timelog entry (20d Finished $infile with result: $pipelineresult ) 18c ⟩

done
◇
```
Fragment referenced in 20a.
Uses: `pipelineresult` 23b.

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* 21a ⟩ ≡

```
function check_start_spotlight {
  language=$1
  if
    [ language == "nl" ]
  then
    spotport=2060
  else
    spotport=2020
  fi
  spotlighthost=130.37.53.33
  ⟨ check spotlight on (21b $spotlighthost,21c $spotport ) 22a ⟩
  if
    [ $spotlightrunning -ne 0 ]
  then
    start_spotlight_on_localhost $language $spotport
    spotlighthost="localhost"
    spotlightrunning=0
  fi
  export spotlighthost
  export spotlightrunning
}
```

◇

Fragment defined by 13a, 21ad.
Fragment referenced in 24b.

⟨ *functions in the jobfile* 21d ⟩ ≡

```
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 13a, 21ad.
Fragment referenced in 24b.

⟨ *check spotlight on* 22a ⟩ ≡
```
exec 6<>/dev/tcp/@1/@2
spotlightrunning=$?
exec 6<&-
exec 6>&-
◇
```
Fragment referenced in 21a.
Uses: `spotlightrunning` 21a.

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* 22b ⟩ ≡
```
export pipelineroot=/home/phuijgen/nlp/nlpp
export BIND=$pipelineroot/bin
◇
```
Fragment defined by 4a, 7c, 9g, 12b, 15j, 18b, 19a, 22b.
Fragment referenced in 4b.
Defines: `BIND` 23ab, `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* 22c ⟩ ≡
```
export eSRL_piddir=‘mktemp -d -t eSRL_piddir.XXXXXX‘
export semaworkdir=‘mktemp -d -t sema.XXXXXX‘
◇
```
Fragment referenced in 24b.

⟨ *Start the bloody eSRL server* 23a ⟩ ≡
```
piddir=‘mktemp -d -t piddir.XXXXXXX‘
( export naflang="en" ; $BIND/start_eSRL $piddir ) &
◇
```
Fragment referenced in 24b.
Defines: `piddir` Never used.
Uses: `BIND` 22b.

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* 23b ⟩ ≡
```
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 23c ⟩
cd $root
rm -f $TEMPRES
◇
```
Fragment referenced in 20b.
Defines: `pipelineresult` 20d, 23c, `timeout` Never used.
Uses: `BIND` 22b, `intray` 4a, `logfile` 6l, `logpath` 6l, `movetotray` 5a, `outpath` 6l, `procfile` 6l, `root` 4a.

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* 23c ⟩ ≡
```
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 13b ⟩
◇
```
Fragment referenced in 23b.
Uses: `failtray` 4a, `movetotray` 5a, `outfile` 6l, `outpath` 6l, `pipelineresult` 23b, `procfile` 6l.

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

⟨ *set utf-8* 24a ⟩ ≡
```
export LANG=en_US.utf8
export LANGUAGE=en_US.utf8
export LC_ALL=en_US.utf8
```
◇
Fragment referenced in 24b.

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"` 24b≡
```
#!/bin/sh
#SBATCH --nodes=1
#SBATCH  --time=30
⟨ set local parameters in the job 22c ⟩
source /home/phuijgen/nlp/test/Pipeline-NL-Lisa/parameters
⟨ Start the bloody eSRL server 23a ⟩
export jobname=$SLURM_JOB_NAME
⟨ log that the job starts 18d ⟩
⟨ set utf-8 24a ⟩
⟨ load stopos module 7b ⟩
⟨ functions 5a, ... ⟩
⟨ functions in the jobfile 13a, ... ⟩
check_start_spotlight nl
check_start_spotlight en
echo spotlighthost: $spotlighthost >&2
echo spotlighthost: $spotlighthost
starttime=‘date +%s‘
⟨ run parallel processes 20a ⟩
⟨ log that the job finishes 18f ⟩
exit

```
◇
Uses: `spotlighthost` 21a.

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 2.5). 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* 25a ⟩ ≡
```
export workdir=/home/phuijgen/nlp/test/Pipeline-NL-Lisa/env
mkdir -p $workdir
```
◇

Fragment referenced in 27c.
Defines: `workdir` 26ac.

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* 25b ⟩ ≡
```
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, 25b, 26a.
Fragment referenced in 24b, 27c.
Defines: `passeer` Never used, `veilig` 27c.

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* 26a ⟩ ≡

```
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, 25b, 26a.
Fragment referenced in 24b, 27c.
Defines: `remove_obsolete_lock` 26b.
Uses: `print` 35a, `workdir` 25a, 26c.

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* 26b ⟩ ≡

```
remove_obsolete_lock runit_runs
runsingle runit_runs
```

◇

Fragment referenced in 27c.
Uses: `remove_obsolete_lock` 26a.

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* 26c ⟩ ≡

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

◇

Fragment referenced in 20a.
Defines: `finishlock` 27ab, `workdir` 25a, 26a.

⟨ *increment the processes-counter* 26d ⟩ ≡

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

◇

Fragment referenced in 20a.
Defines: `countlock` 27a.
Uses: `proccount` 6a.

⟨ *decrement the processes-counter, kill if this was the only process* 27a ⟩ ≡

```
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 20a.
Uses: `countlock` 26d, `finishlock` 26c, `proccount` 6a.

⟨ *wait for working-processes* 27b ⟩ ≡

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

Fragment referenced in 20a.
Uses: `finishlock` 26c.

7.5 The management script

`"../runit"` 27c≡

```
#!/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 25a ⟩
⟨ get runit options 28b ⟩
⟨ functions 5a, ... ⟩
⟨ die if another instance of runit is running 26b ⟩
⟨ load stopos module 7b ⟩
⟨ check/create directories 6a, ... ⟩
⟨ clean up joblogs 17c, ... ⟩
⟨ clean up outputfiles and logfiles 7a ⟩
⟨ count jobs 15e ⟩
⟨ update the stopos pool 9d ⟩
⟨ submit jobs when necessary 16c ⟩
if
  [ $loud ]
then
  ⟨ print summary 28c ⟩
fi
veilig runit_runs
exit
◇
```

Uses: `root` 4a, `veilig` 25b.

⟨ *make scripts executable* 28a ⟩ ≡
```
chmod 775 /home/phuijgen/nlp/test/Pipeline-NL-Lisa/runit
```
◇

Fragment defined by 28a, 41b.
Fragment referenced in 41c.

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* 28b ⟩ ≡
```
OPTIND=1
export loud=0
while getopts "s:" opt; do
    case "$opt" in
    s)  loud=
        ;;
    esac
done
shift $((OPTIND-1))
```
◇

Fragment referenced in 27c.

Print the summary:

⟨ *print summary* 28c ⟩ ≡
```
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 27c.
Uses: `failcount` 6a, `incount` 6a, `jobs_to_be_submitted` 16bc, `lisa_jobcount` 15e, `logcount` 6a, `proccount` 6a, `running_jobs` 15e.

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* 29 ⟩ ≡
```
NUWEB=../env/bin/nuweb
```
◇

Fragment defined by 29, 31b, 33bc, 35d, 38a, 40d.
Fragment referenced in 30a.
Uses: `nuweb` 37b.

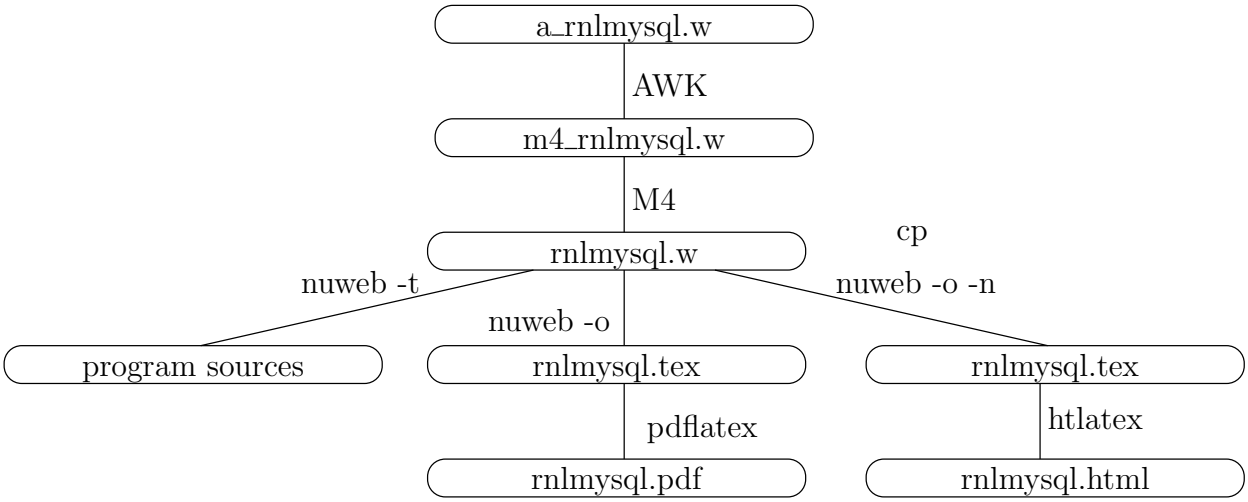

Figure 1: *Translation of the raw code of this document into printable/viewable documents and into program sources. The figure shows the pathways and the main files involved.*

A.3 The Makefile for this project.

This chapter assembles the Makefile for this project.

```
"Makefile" 30a≡
    ⟨ default target 30b ⟩

    ⟨ parameters in Makefile 29, ... ⟩

    ⟨ impliciete make regels 33a, ... ⟩
    ⟨ expliciete make regels 31c, ... ⟩
    ⟨ make targets 30c, ... ⟩
    ◇
```

The default target of make is `all`.

```
⟨ default target 30b ⟩ ≡
    all : ⟨ all targets 31a ⟩
    .PHONY : all

    ◇
```

Fragment referenced in 30a.
Defines: `all` Never used, `PHONY` 34b.

```
⟨ make targets 30c ⟩ ≡
    clean:
        ⟨ clean up 31d ⟩

    ◇
```

Fragment defined by 30c, 35ab, 38e, 41ac.
Fragment referenced in 30a.

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

⟨ *all targets* 31a ⟩ ≡
```
Pipeline_NL_Lisa.pdf◇
```
Fragment referenced in 30b.
Uses: pdf 35a.

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* 31b ⟩ ≡
```
.SUFFIXES: .pdf .w .tex .html .aux .log .php

◇
```
Fragment defined by 29, 31b, 33bc, 35d, 38a, 40d.
Fragment referenced in 30a.
Defines: SUFFIXES Never used.
Uses: pdf 35a.

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* 31c ⟩ ≡
```
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 31c, 32abc, 34b, 36a, 38bd.
Fragment referenced in 30a.
Uses: nuweb 37b.

⟨ *clean up* 31d ⟩ ≡
```
rm -rf ../nuweb-1.58
◇
```
Fragment referenced in 30c.
Uses: nuweb 37b.

⟨ *expliciete make regels* 32a ⟩ ≡

```
../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 31c, 32abc, 34b, 36a, 38bd.
Fragment referenced in 30a.
Uses: `nuweb` 37b.

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* 32b ⟩ ≡

```
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 31c, 32abc, 34b, 36a, 38bd.
Fragment referenced in 30a.
Uses: `print` 35a.

A.5.2 Run the M4 pre-processor

⟨ *expliciete make regels* 32c ⟩ ≡

```
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 31c, 32abc, 34b, 36a, 38bd.
Fragment referenced in 30a.

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* 33a ⟩ ≡
```
%.pdf: %.w
        ./w2pdf $<
```
◇

Fragment defined by 33a, 34a, 38c.
Fragment referenced in 30a.
Uses: `pdf` 35a.

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* 33b ⟩ ≡
```
FIGFILES=fileschema directorystructure
```
◇

Fragment defined by 29, 31b, 33bc, 35d, 38a, 40d.
Fragment referenced in 30a.
Defines: `FIGFILES` 33c.

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* 33c ⟩ ≡
```
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 29, 31b, 33bc, 35d, 38a, 40d.
Fragment referenced in 30a.
Defines: `FIGFILENAMES` Never used, `PDFT_NAMES` 35b, `PDF_FIG_NAMES` 35b, `PST_NAMES` Never used, `PS_FIG_NAMES` Never used.
Uses: `FIGFILES` 33b.

Create the graph files with program `fig2dev`:

⟨ *impliciete make regels* 34a ⟩ ≡

```
%.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 33a, 34a, 38c.
Fragment referenced in 30a.
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* 34b ⟩ ≡

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

.PHONY : bibfile
◇
```

Fragment defined by 31c, 32abc, 34b, 36a, 38bd.
Fragment referenced in 30a.
Uses: `PHONY` 30b.

A.6.3 Create a printable/viewable document

Make a PDF document for printing and viewing.

⟨ *make targets* 35a ⟩ ≡
```
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 30c, 35ab, 38e, 41ac.
Fragment referenced in 30a.
Defines: `pdf` 31ab, 33a, 35b, `print` 6j, 9a, 10b, 11g, 14, 15be, 26a, 32b, `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* 35b ⟩ ≡
```
Pipeline_NL_Lisa.pdf : Pipeline_NL_Lisa.w $(W2PDF)  $(PDF_FIG_NAMES) $(PDFT_NAMES)
        chmod 775 $(W2PDF)
        $(W2PDF) $*

◇
```
Fragment defined by 30c, 35ab, 38e, 41ac.
Fragment referenced in 30a.
Uses: `pdf` 35a, `PDFT_NAMES` 33c, `PDF_FIG_NAMES` 33c.

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* 35c ⟩ ≡
```
../nuweb/bin ◇
```
Fragment referenced in 41a.
Uses: `nuweb` 37b.

⟨ *parameters in Makefile* 35d ⟩ ≡
```
W2PDF=../nuweb/bin/w2pdf
◇
```
Fragment defined by 29, 31b, 33bc, 35d, 38a, 40d.
Fragment referenced in 30a.
Uses: `nuweb` 37b.

⟨ *expliciete make regels* 36a ⟩ ≡
```
$(W2PDF) : Pipeline_NL_Lisa.w $(NUWEB)
        $(NUWEB) Pipeline_NL_Lisa.w
```
◇
Fragment defined by 31c, 32abc, 34b, 36a, 38bd.
Fragment referenced in 30a.

"../nuweb/bin/w2pdf" 36b≡
```
#!/bin/bash
# w2pdf -- compile a nuweb file
# usage: w2pdf [filename]
# 20181223 at 1659h: Generated by nuweb from a_Pipeline_NL_Lisa.w
NUWEB=../env/bin/nuweb
LATEXCOMPILER=pdflatex
⟨ filenames in nuweb compile script 36d ⟩
⟨ compile nuweb 36c ⟩

```
◇
Uses: nuweb 37b.

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* 36c ⟩ ≡
```
NUWEB=/home/phuijgen/nlp/test/Pipeline-NL-Lisa/env/bin/nuweb
⟨ run the processors until the aux file remains unchanged 37c ⟩
⟨ remove the copy of the aux file 37a ⟩
```
◇
Fragment referenced in 36b.
Uses: nuweb 37b.

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* 36d ⟩ ≡
```
nufil=$1
trunk=${1%%.*}
texfil=${trunk}.tex
auxfil=${trunk}.aux
oldaux=old.${trunk}.aux
indexfil=${trunk}.idx
oldindexfil=old.${trunk}.idx
```
◇
Fragment referenced in 36b.
Defines: auxfil 37c, 39c, 40a, indexfil 37c, 39c, nufil 37b, 39c, 40b, oldaux 37ac, 39c, 40a, oldindexfil 37c, 39c, texfil 37b, 39c, 40b, trunk 37b, 39c, 40bc.

Remove the old copy if it is no longer needed.

⟨ *remove the copy of the aux file* 37a ⟩ ≡
```
rm $oldaux
```
◇

Fragment referenced in 36c, 39b.
Uses: `oldaux` 36d, 39c.

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* 37b ⟩ ≡
```
$NUWEB $nufil
$LATEXCOMPILER $texfil
makeindex $trunk
bibtex $trunk
```
◇

Fragment referenced in 37c.
Defines: `bibtex` 40bc, `makeindex` 40bc, `nuweb` 29, 31cd, 32a, 35cd, 36bc, 38a, 39a.
Uses: `nufil` 36d, 39c, `texfil` 36d, 39c, `trunk` 36d, 39c.

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* 37c ⟩ ≡
```
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 37b ⟩
  if [ $LOOPCOUNTER -ge 10 ]
  then
    cp $auxfil $oldaux
  fi;
done
```
◇

Fragment referenced in 36c.
Uses: `auxfil` 36d, 39c, `indexfil` 36d, `oldaux` 36d, 39c, `oldindexfil` 36d.

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* 38a ⟩ ≡
```
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 29, 31b, 33bc, 35d, 38a, 40d.
Fragment referenced in 30a.
Uses: `nuweb` 37b.

Make the directory:

⟨ *expliciete make regels* 38b ⟩ ≡
```
$(htmldir) :
        mkdir -p $(htmldir)
```
◇

Fragment defined by 31c, 32abc, 34b, 36a, 38bd.
Fragment referenced in 30a.

The rule to copy files in it:

⟨ *impliciete make regels* 38c ⟩ ≡
```
$(htmldir)/% : % $(htmldir)
        cp $< $(htmldir)/
```
◇

Fragment defined by 33a, 34a, 38c.
Fragment referenced in 30a.

Do the work:

⟨ *expliciete make regels* 38d ⟩ ≡
```
$(htmltarget) : $(htmlmaterial) $(htmldir)
        cd $(htmldir) && chmod 775 w2html
        cd $(htmldir) && ./w2html nlpp.w
```
◇

Fragment defined by 31c, 32abc, 34b, 36a, 38bd.
Fragment referenced in 30a.

Invoke:

⟨ *make targets* 38e ⟩ ≡
```
htm : $(htmldir) $(htmltarget)
```
◇

Fragment defined by 30c, 35ab, 38e, 41ac.
Fragment referenced in 30a.

Create a script that performs the translation.

```
"w2html" 39a≡
      #!/bin/bash
      # w2html -- make a html file from a nuweb file
      # usage: w2html [filename]
      #  [filename]: Name of the nuweb source file.
      # 20181223 at 1659h: 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 39c ⟩

      ⟨ perform the task of w2html 39b ⟩

      ◇
```

Uses: `nuweb` 37b.

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 39b ⟩ ≡
      ⟨ run the html processors until the aux file remains unchanged 40a ⟩
      ⟨ remove the copy of the aux file 37a ⟩
      ◇
```

Fragment referenced in 39a.

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 39c ⟩ ≡
      nufil=$1
      trunk=${1%%.*}
      texfil=${trunk}.tex
      auxfil=${trunk}.aux
      oldaux=old.${trunk}.aux
      indexfil=${trunk}.idx
      oldindexfil=old.${trunk}.idx
      ◇
```

Fragment referenced in 39a.
Defines: `auxfil` 36d, 37c, 40a, `nufil` 36d, 37b, 40b, `oldaux` 36d, 37ac, 40a, `texfil` 36d, 37b, 40b, `trunk` 36d, 37b, 40bc.
Uses: `indexfil` 36d, `oldindexfil` 36d.

⟨ *run the html processors until the aux file remains unchanged* 40a ⟩ ≡

```
while
  ! cmp -s $auxfil $oldaux
do
  if [ -e $auxfil ]
  then
   cp $auxfil $oldaux
  fi
  ⟨ run the html processors 40b ⟩
done
⟨ run tex4ht 40c ⟩

◇
```

Fragment referenced in 39b.
Uses: `auxfil` 36d, 39c, `oldaux` 36d, 39c.

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

⟨ *run the html processors* 40b ⟩ ≡

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

Fragment referenced in 40a.
Uses: `bibtex` 37b, `makeindex` 37b, `nufil` 36d, 39c, `texfil` 36d, 39c, `trunk` 36d, 39c.

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* 40c ⟩ ≡

```
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 40a.
Uses: `bibtex` 37b, `makeindex` 37b, `trunk` 36d, 39c.

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* 40d ⟩ ≡

```
MKDIR = mkdir -p

◇
```

Fragment defined by 29, 31b, 33bc, 35d, 38a, 40d.
Fragment referenced in 30a.
Defines: `MKDIR` 41a.

⟨ *make targets* 41a ⟩ ≡
```
DIRS = ⟨ directories to create 35c ⟩

$(DIRS) :
        $(MKDIR) $@
```
◇
Fragment defined by 30c, 35ab, 38e, 41ac.
Fragment referenced in 30a.
Defines: DIRS 41c.
Uses: MKDIR 40d.

⟨ *make scripts executable* 41b ⟩ ≡
```
chmod -R 775  ../env/bin/*
```
◇
Fragment defined by 28a, 41b.
Fragment referenced in 41c.

⟨ *make targets* 41c ⟩ ≡
```
source : Pipeline_NL_Lisa.w $(DIRS) $(NUWEB)
        $(NUWEB) Pipeline_NL_Lisa.w
        ⟨ make scripts executable 28a, ... ⟩
```
◇
Fragment defined by 30c, 35ab, 38e, 41ac.
Fragment referenced in 30a.
Uses: DIRS 41a.

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

"../magicplace_2" Defined by 24b.
"../nuweb/bin/w2pdf" Defined by 36b.
"../parameters" Defined by 4b.
"../runit" Defined by 27c.
"Makefile" Defined by 30a.
"w2html" Defined by 39a.

C.2 Macro's

⟨ add new filenames to the pool 12a ⟩ Referenced in 9d.

⟨ add timelog entry 18c ⟩ Referenced in 18df, 20b.
⟨ all targets 31a ⟩ Referenced in 30b.
⟨ check spotlight on 22a ⟩ Referenced in 21a.
⟨ check/create directories 6ak, 17b ⟩ Referenced in 27c.
⟨ clean up 31d ⟩ Referenced in 30c.
⟨ clean up joblogs 17c, 18a ⟩ Referenced in 27c.
⟨ clean up old.infilelist and infilelist 11b ⟩ Referenced in 11a.
⟨ clean up outputfiles and logfiles 7a ⟩ Referenced in 27c.
⟨ clean up pool and old.filenames 11a ⟩ Referenced in 9d.
⟨ clean up proctray 11g ⟩ Referenced in 9d.
⟨ compile nuweb 36c ⟩ Referenced in 36b.
⟨ count files in tray 6j ⟩ Referenced in 6a.
⟨ count jobs 15e ⟩ Referenced in 27c.
⟨ count jobs in squeue report 15a ⟩ Referenced in 15be.
⟨ count running jobs in squeue report 15b ⟩ Not referenced.
⟨ decide whether to renew the stopos-pool 10c ⟩ Referenced in 9d.
⟨ decrement the processes-counter, kill if this was the only process 27a ⟩ Referenced in 20a.
⟨ default target 30b ⟩ Referenced in 30a.
⟨ determine how many jobs have to be submitted 16b ⟩ Referenced in 16c.
⟨ determine number of jobs that we want to have 16a ⟩ Referenced in 16b.
⟨ determine number of parallel processes 19b ⟩ Referenced in 20a.
⟨ die if another instance of runit is running 26b ⟩ Referenced in 27c.
⟨ directories to create 35c ⟩ Referenced in 41a.
⟨ expliciete make regels 31c, 32abc, 34b, 36a, 38bd ⟩ Referenced in 30a.
⟨ filenames in nuweb compile script 36d ⟩ Referenced in 36b.
⟨ filenames in w2html 39c ⟩ Referenced in 39a.
⟨ functions 5ab, 25b, 26a ⟩ Referenced in 24b, 27c.
⟨ functions in the jobfile 13a, 21ad ⟩ Referenced in 24b.
⟨ gawk command to count jobs from squeue report 14 ⟩ Referenced in 15a.
⟨ generate filenames 6l ⟩ Referenced in 13a.
⟨ get next infile from stopos 12c ⟩ Referenced in 13a.
⟨ get runit options 28b ⟩ Referenced in 27c.
⟨ impliciete make regels 33a, 34a, 38c ⟩ Referenced in 30a.
⟨ increment the processes-counter 26d ⟩ Referenced in 20a.
⟨ init processescounter 26c ⟩ Referenced in 20a.
⟨ initialize sematree 25a ⟩ Referenced in 27c.
⟨ is the pool full or empty? 10a ⟩ Referenced in 9d.
⟨ keep common filenames 8b ⟩ Referenced in 11b.
⟨ load stopos module 7b ⟩ Referenced in 24b, 27c.
⟨ log that the job finishes 18f ⟩ Referenced in 24b.
⟨ log that the job starts 18d ⟩ Referenced in 24b.
⟨ make a list of filenames in the intray 10b ⟩ Referenced in 9d.
⟨ make scripts executable 28a, 41b ⟩ Referenced in 41c.
⟨ make targets 30c, 35ab, 38e, 41ac ⟩ Referenced in 30a.
⟨ move the processed naf around 23c ⟩ Referenced in 23b.
⟨ parameters 4a, 7c, 9g, 12b, 15j, 18b, 19a, 22b ⟩ Referenced in 4b.
⟨ parameters in Makefile 29, 31b, 33bc, 35d, 38a, 40d ⟩ Referenced in 30a.
⟨ perform the processing loop 20b ⟩ Referenced in 20a.
⟨ perform the task of w2html 39b ⟩ Referenced in 39a.
⟨ print summary 28c ⟩ Referenced in 27c.
⟨ process infile 23b ⟩ Referenced in 20b.
⟨ remove the copy of the aux file 37a ⟩ Referenced in 36c, 39b.
⟨ remove the infile from the stopos pool 13b ⟩ Referenced in 23c.
⟨ run parallel processes 20a ⟩ Referenced in 24b.
⟨ run tex4ht 40c ⟩ Referenced in 40a.
⟨ run the html processors 40b ⟩ Referenced in 40a.
⟨ run the html processors until the aux file remains unchanged 40a ⟩ Referenced in 39b.
⟨ run the processors until the aux file remains unchanged 37c ⟩ Referenced in 36c.

⟨ run the three processors 37b ⟩ Referenced in 37c.
⟨ set local parameters in the job 22c ⟩ Referenced in 24b.
⟨ set utf-8 24a ⟩ Referenced in 24b.
⟨ Start the bloody eSRL server 23a ⟩ Referenced in 24b.
⟨ submit jobs 17a ⟩ Referenced in 16c.
⟨ submit jobs when necessary 16c ⟩ Referenced in 27c.
⟨ subtract filenames 8a ⟩ Referenced in 9a, 11b.
⟨ subtract files in proctray from shadow bookkeeping 9a ⟩ Referenced in 9d.
⟨ update the stopos pool 9d ⟩ Referenced in 27c.
⟨ wait for working-processes 27b ⟩ Referenced in 20a.

C.3 Variables

`all`: 30b.
`auxfil`: 36d, 37c, 39c, 40a.
`bibtex`: 37b, 40bc.
`BIND`: 22b, 23ab.
`copytotray`: 5b.
`countlock`: 26d, 27a.
`DIRS`: 41a, 41c.
`failcount`: 6a, 6g, 28c.
`failtray`: 4a, 6afl, 7a, 23c.
`fig2dev`: 34a.
`FIGFILENAMES`: 33c.
`FIGFILES`: 33b, 33c.
`filesperjob`: 15j, 16a.
`filtrunk`: 6l.
`finishlock`: 26c, 27ab.
`getfile`: 13a, 20b.
`incount`: 6a, 6c, 10c, 28c.
`indexfil`: 36d, 37c, 39c.
`infilelist`: 9b, 10b, 11acdef, 12a.
`intray`: 4a, 6bkl, 9a, 10b, 11g, 23b.
`joblist`: 15e, 15g.
`joblogs`: 17b, 17c, 18a.
`jobs_needed`: 16a, 16b, 16c.
`jobs_to_be_submitted`: 16b, 16c, 16d, 28c.
`lisa_jobcount`: 10c, 11g, 15e, 15f, 16b, 28c.
`logcount`: 6a, 6i, 28c.
`logfile`: 6l, 23b.
`logpath`: 6l, 23b.
`logtray`: 4a, 6ahl, 7a.
`makeindex`: 37b, 40bc.
`maxprocs`: 19b, 20a.
`maxproctime`: 11g, 12b.
`MKDIR`: 40d, 41a.
`module`: 7b.
`movetotray`: 5a, 11g, 23bc.
`nufil`: 36d, 37b, 39c, 40b.
`nuweb`: 29, 31cd, 32a, 35cd, 36bc, 37b, 38a, 39a.
`oldaux`: 36d, 37ac, 39c, 40a.
`oldindexfil`: 36d, 37c, 39c.
`outfile`: 6l, 13a, 23c.
`outpath`: 6l, 23bc.
`outtray`: 4a, 6al, 7a.
`passeer`: 25b.
`pdf`: 31ab, 33a, 35a, 35b.
`PDFT_NAMES`: 33c, 35b.

PDF_FIG_NAMES: 33c, 35b.
PHONY: 30b, 34b.
piddir: 23a.
pipelineresult: 20d, 23b, 23c.
pipelineroot: 22b.
pool_empty: 9d, 9f, 10c.
pool_full: 9d, 9e.
print: 6j, 9a, 10b, 11g, 14, 15be, 26a, 32b, 35a.
proccount: 6a, 6e, 26d, 27a, 28c.
procfile: 6l, 23bc.
proclist: 9a, 9c.
procnum: 20a.
procpath: 6l.
PST_NAMES: 33c.
PS_FIG_NAMES: 33c.
regen_pool_condition: 10c, 11a.
remove_obsolete_lock: 26a, 26b.
required_mem_per_process_gb: 19a.
required_mem_per_process_mb: 19a, 19b.
rjoblist: 15bd, 15e, 15i.
root: 4a, 9ad, 10bc, 17c, 18a, 23b, 27c.
running_jobs: 15c, 15e, 15h, 28c.
spotlighthost: 21a, 21b, 24b.
spotlightrunning: 21a, 22a.
stopos: 7b, 10a, 11a, 12ac, 13b.
stopospool: 7c, 10a, 11a, 12ac, 13b.
stopos_sufficient_filecount: 9g, 10a.
subfile: 8a, 8b.
SUFFIXES: 31b.
texfil: 36d, 37b, 39c, 40b.
timeout: 23b.
trunk: 36d, 37b, 39c, 40bc.
unreadycount: 6a, 16a.
veilig: 25b, 27c.
view: 35a.
workdir: 25a, 26a, 26c.