Bilingual NLP pipeline

Paul Huygen <paul.huygen@huygen.nl>

28th June 2017
11:19 h.

Abstract

This is a description and documentation of the installation of the Newsreader-pipeline [1]. It is an instrument to annotate Dutch or English documents with NLP tags. The documents have to be stored in *Newsreader Annotation Format* (NAF [1]).

Contents

1. http://www.newsreader-project.eu/files/2012/12/NWR-D4-2-2.pdf

1 Introduction

This document describes the installation of a pipeline that annotates texts in order to extract knowledge. The pipeline has been set up as part of the newsreader [2] project. It accepts and produces texts in the NAF (Newsreader Annotation Format) format.

Apart from describing the pipeline set-up, the document actually constructs the pipeline. The pipeline has been installed on a (Ubuntu) Linux computer.

The installation has been parameterised. The locations and names that you read (and that will be used to build the pipeline) have been read from variables in file `inst.m4` in the nuweb directory.

The installed pipeline is bi-lingual. It is capable to annotate Dutch and English texts. It recognizes the language from the "lang" attribute of the `NAF` element of the document. Some of the modules are specific for a single language, other modules support both languages. s a result, there must be two pathways to lead a document through the pipeline, one for English and one for Dutch.

The pipeline is a concatenation of independent software modules, each of which reads a NAF document from standard input and produces another NAF document on standard output.

The aim is, to install the pipeline from open-source modules that can e.g. be obtained from Github. However, that aim is only partially fulfilled. Some of the modules still contain elements that are

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

not open-source ot data that are not freely available. Because of lack of time, the current version of the installer installs the English pipeline from a frozen repository of the Newsreader Project.

The NLPP pipeline can be seen as contructed in three parts: 1) The software that is needed to run the pipeline, e.g. compilers and interpreters; 2) the modules themselves and 3) scripts to to make the modules operate on a document.

1.1 Modules of the pipeline

Table 2 lists the modules in the pipeline. The column *source* indicates the origin of the module. The modules are obtained in one of the following ways:

1. If possible, the module is directly obtained from an open-source repository like Github.
2. Some modules have not been officially published in a repository. These modules have been packed in a tar-ball that can be obtained by the author. In table 2 this has been indicated as SNAPSHOT.

The modules themselves use other utilities like dependency-taggers and POS taggers. These utilities are listed in table 1.

Module	Version	Section	Source
KafNafParserPy	Feb 1, 2015	??	Github
Alpino	21088	??	RUG
Ticcutils	0.7	??	ILK
Timbl	6.4.6	??	ILK
Treetagger	3.2	??	Uni. München
Spotlight server	0.7	??	Spotlight

Table 1: *List of the utilities to be installed. Column description:* ***directory:*** *Name of the subdirectory below* `mod` *in which it is installed;* ***Source:*** *From where the module has been obtained;* ***script:*** *Script to be included in a pipeline.*

1.2 Reproducibility

An important goal of this pipeline is, to achieve reproducibility. It mean, that at some point in the future the annotation could be re-done on the document and it should produce a result that is identical as the result of the original annotation. In our casem reproducibility ivolves the following aspects:

- The annotated document ought to contain documentation about the annotation process: What modules have been applied, what was the version of the software of each module, Which resources have been used and what was the version of the resources.
- The source code of the modules as well as resources like data-sets and programming languages should be available from open repository.
- The repositories of the resources should use some versioning system enabling to re-use the version that has been used originally.

A problem in some cases is, that we need to use utilities that are supplied by external parties, and we do not have control about their methods of publication and version management. Examples of such utilities are the compilers for programming languages like Java, Python and parsers like Alpino.

Therefore, we have the following policy to achieve reproducibility:

- Each of the modules writes in the output NAF its own version, and details about the used resources in sufficient detail to enable re-processing.
- It is assumed that when a programming language (e.g. Java, Python) is used, annotion can be reproducible when the major versions coincide.

Module	Source	Resources	Section	Commit	Script	language
Tokenizer	Github	Java	6.1.1	1a69...	`tok`	en/nl
Topic detection	Github	Java	??	b2e0...	`topic`	en/nl
Morpho-syntactic parser	Github	Python, Alpino	??	52a9...	`mor`	nl
POS-tagger	snapshot		??	...	`pos`	en
Named-entity rec/class	Github		??	ca02...	`nerc`	en/nl
Dark-entity relinker	Github		??	90d7...	`nerc`	en/nl
Constituent parser	snapshot		??	...	`constpars`	en
Word-sense disamb. nl	Github		??	0300...	`wsd`	nl
Word-sense disamb. en	snapshot		??	...	`ewsd`	en
Named entity/DBP	snapshot		??	...	`ned`	en/nl
NED reranker	snapshot		??	...	`nedrerscript`	en
Wikify	snapshot		??	...	`wikify`	en
UKB	snapshot		??	...	`ukb`	en
Coreference-base	snapshot		??	...	`coreference-base`	en
Heideltime	Github		??	0fd3...	`heideltime`	nl
Onto-tagger	Github		??	9ea0...	`onto`	nl
Semantic Role labeling nl	Github		??	675d...	`srl`	nl
Semantic Role labeling en	snapshot		??	...	`eSRL`	en
Nominal Event ann.	Github		??	9ea0...	`nomevent`	nl
SRL dutch nominals	Github		??	6115...	`srl-dutch-nominals`	nl
Framenet-SRL	Github		??	9ea0...	`framesrl`	nl
FBK-time	snapshot		??	...	`FBK-time`	en
FBK-temprel	snapshot		??	...	`FBK-temprel`	en
FBK-causalrel	snapshot		??	...	`FBK-causalrel`	en
Opinion-miner	Github		??	40a7...	`opinimin`	en/nl
Event-coref	Github		??	a01f...	`evcoref`	en/nl
Factuality tagger	Github		??	58fa...	`factuality`	en
Factuality tagger	Github		??	cbad...	`factuality`	nl

Table 2: *List of the modules to be installed. Column description:* ***directory:*** *Name of the subdirectory below subdirectory* `modules` *in which it is installed;* ***source:*** *From where the module has been obtained;* ***commit:*** *Commit-name or version-tag* ***script:*** *Script to be included in a pipeline.*

- A script is constructed that reproducibly builds an environment for the pipeline on some software/hardware platform (e.g. Linux on X64 CPU), using utilities that vave been stored in some non-open repository (to preclude copyright-problems).

2 Structure of the pipeline

The finished pipeline consists of:

- A directory that contains for each module an directory with the module in installed form.
- A script that reads an input naf file or plain text file from standard in and produces an annotated NAF file on standard out.
- A script that must be "sourced" in order to find the resources that the modules need to find.

The directory with the modules must be relocatable and immutable. That means that scripts in modules do not have write permissions on the module directory and that they have to find other files on path-descriptions relative to the current path of the script itself.

2.1 Expected resources

In order to run the modules expect the following:

- Instruction `java` invokes Java 1.8;
- Instruction `python` invokes Python 3.6;
- Instruction `Perl` invokes Perl 5;
- Variable `TMPDIR` points to a user-writable directory.

3 Construct the infra-structure

In this section we will generate a script that set up an infra-structure in which the pipeline can be exploited. An attempt is made to make as little as possible presumptions about the services that the host provides.

We need to set up the following:

- Java Version 1.8
- Maven (Gradle?)
- Python version 3.6
- Python packages
- Autoconf
- ...

Let us generate a script to do the work:

```
"../env/bin/make_infrastructure" 5a≡
    #!/bin/bash
    ⟨ get location of the script (5b DIR ) 30a ⟩
    cd $DIR
    source ../../progenv
    ⟨ init make_infrastructure 6f, ... ⟩
    ⟨ set up Java 11a ⟩
    ⟨ set up Maven 12b ⟩
    ⟨ set up Python 13b, ... ⟩
    ⟨ set up autoconf 19e ⟩
    ⟨ set up Perl 17a ⟩
    ⟨ install shared libs 20b ⟩
    ⟨ install Alpino 21a ⟩

    ◇
```

⟨ *make scripts executable* 6a ⟩ ≡
```
chmod 775  ../env/bin/make_infrastructure
```
◇

Fragment defined by 6ad, 15a, 22a, 25a, 42c.
Fragment referenced in 42d.

Let us also make a script that cleans up the infra-structure after the installation.

"../env/bin/clean_infrastructure" 6b≡
```
#!/bin/bash
⟨ get location of the script (6c DIR ) 30a ⟩
cd $DIR
source ../../progenv
⟨ init make_infrastructure 6f, ... ⟩
⟨ clean up after installation 12g ⟩
```
◇

⟨ *make scripts executable* 6d ⟩ ≡
```
chmod 775  ../env/bin/clean_infrastructure
```
◇

Fragment defined by 6ad, 15a, 22a, 25a, 42c.
Fragment referenced in 42d.

Before we begin, we can try whether commands that we need to use actually exist and stop execution otherwise.

⟨ *test presence of command* 6e ⟩ ≡
```
which @1 >/dev/null
if
  [ $? -ne 0 ]
then
  echo "Please install @1"
  exit 4
fi
```
◇

Fragment referenced in 6f.
Uses: install 43a.

⟨ *init make_infrastructure* 6f ⟩ ≡
```
⟨ test presence of command (6g git ) 6e ⟩
⟨ test presence of command (6h tar ) 6e ⟩
⟨ test presence of command (6i unzip ) 6e ⟩
⟨ test presence of command (6j tcsh ) 6e ⟩
⟨ test presence of command (6k hg ) 6e ⟩

```
◇

Fragment defined by 6f, 9b.
Fragment referenced in 5a, 6b.

3.1 File-structure

Let us set up the pipeline in a directory-structure that looks like figure 1. The directories have the following functions.

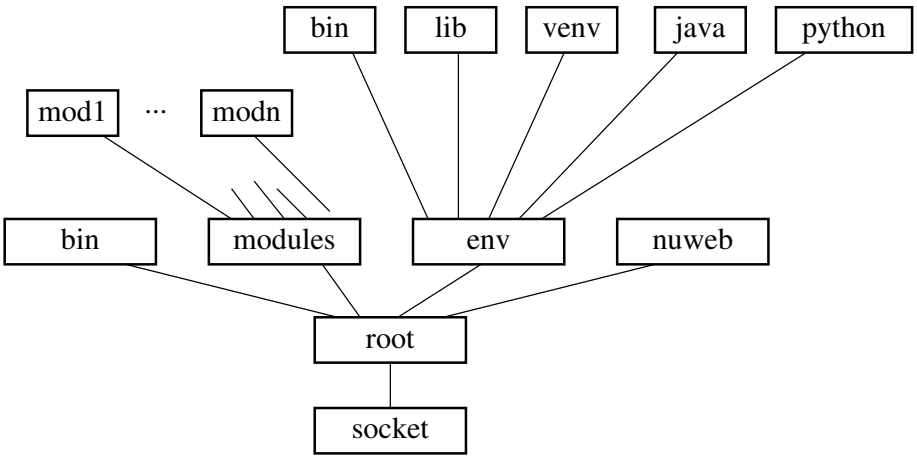

Figure 1: *Directory-structure of the pipeline (see text).*

socket: The directory in the host where the pipeline is to be implemented.
root: The root of the pipeline directory-structure.
nuweb: This directory contains this document and everything to create the pipeline from the open sources of the modules.
modules: Contains subdirectories with the NLP modules that can be applied in the pipeline.
bin: Contains for each of the applicable modules a script that reads NAF input, passes it to the module in the `modules` directory and produces the output on standard out. Furthermore, the subdirectory contains the script `install-modules` that performs the installation, and a script `test` that shows that the pipeline works in a trivial case.
env: The programming environment. It contains a.o. the Java development kit, Python, the Python virtual environment (`venv`), libraries and binaries.

⟨ *directories to create* 7a ⟩ ≡
```
../modules ◇
```
Fragment defined by 7abcd, 37b.
Fragment referenced in 42b.

⟨ *directories to create* 7b ⟩ ≡
```
../bin ../env/bin ◇
```
Fragment defined by 7abcd, 37b.
Fragment referenced in 42b.

⟨ *directories to create* 7c ⟩ ≡
```
../env/lib ◇
```
Fragment defined by 7abcd, 37b.
Fragment referenced in 42b.

⟨ *directories to create* 7d ⟩ ≡
```
../env/etc ◇
```
Fragment defined by 7abcd, 37b.
Fragment referenced in 42b.

It would be great if an installed pipeline could be moved to another directory while it would keep working. We are not yet sure whether this is possible. However, a minimum condition for this to work would be, that the location of the pipeline can be determined at run-time. To achieve this, let us place a script in the root-directory of the pipeline, that can find in run-time the absolute path to itself and that generates variables that point to the other directories.

```
"../progenv" 8a≡
      # Source this script
      ⟨ get location of the script (8b piperoot ) 30a ⟩
      ⟨ set variables that point to the directory-structure 8e, ... ⟩
      ⟨ set environment parameters 8c, ... ⟩
      if
        [ -e "$piperoot/progenvv" ]
      then
        source $piperoot/progenvv
      fi
      export progenvset=0
      ◇
```

Uses: piperoot 8d.

```
⟨ set environment parameters 8c ⟩ ≡
      export LC_ALL=en_US.UTF-8
      export LANG=en_US.UTF-8
      export LANGUAGE=en_US.UTF-8
      ◇
```

Fragment defined by 8c, 21b.
Fragment referenced in 8a.

The full path to the sourced script can be found in variable BASH_SOURCE[0].

```
⟨ find the nlpp root directory 8d ⟩ ≡
      piperoot="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"
      ◇
```

Fragment never referenced.
Defines: piperoot 8abe, 11e, 14ac, 18b, 19e, 20a, 22b, 29a.

Once we know piperoot, we know the path to the other directories of figure 1.

```
⟨ set variables that point to the directory-structure 8e ⟩ ≡
      export pipesocket=${piperoot%%/nlpp}
      export nuwebdir=$piperoot/nuweb
      export envdir=$piperoot/env
      export envbindir=$envdir/bin
      export envlibdir=$envdir/lib
      export modulesdir=$piperoot/modules
      export pipebin=$piperoot/bin
      export javadir=$envdir/java
      export jarsdir=$javadir/jars
      ◇
```

Fragment defined by 8e, 9a, 12f.
Fragment referenced in 8a.
Uses: nuweb 38d, piperoot 8d.

Add the environment bin directory to PATH:

⟨ *set variables that point to the directory-structure* 9a ⟩ ≡
```
export PATH=$envbindir:$pipebin:$PATH
◇
```
Fragment defined by 8e, 9a, 12f.
Fragment referenced in 8a.
Defines: `PATH` 11e, 12bf, 18b.

3.2 Download resources

To enhance speed of the installation we start to download all resources that we can download at the beginning of the installation in a single blow as parallel processes. We park the resources in a directory `v4.0.0.0_nlpp_resources`, located in the directory where the root of NLPP also resides.

⟨ *init make_infrastructure* 9b ⟩ ≡
```
⟨ download everything 9c, ... ⟩
wait
◇
```
Fragment defined by 6f, 9b.
Fragment referenced in 5a, 6b.

Hopefully there will be little to download.

Synchronize with a non-open snapshot-directory if possible. It is only possible if a valid `ssh` key resides in file `nrkey` in the directory in which the nlpp root directory resides.

⟨ *download everything* 9c ⟩ ≡
```
 mkdir -p $pipesocket/v4.0.0.0_nlpp_resources
if
  [ -e /home/huygen/projecten/pipelines/nrkey ]
then
  cd $pipesocket
  ( rsync -e "ssh -i /home/huygen/projecten/pipelines/nrkey" -
rLt newsreader@kyoto.let.vu.nl:v4.0.0.0_nlpp_resources . ) &
fi

◇
```
Fragment defined by 9c, 19a.
Fragment referenced in 9b.

Download other stuff using `wget`. The following macro downloads a resource into the snapshot-directory if it is not already there.

⟨ *need to wget* 9d ⟩ ≡
```
if
  [ ! -e $pipesocket/v4.0.0.0_nlpp_resources/@1 ]
then
  cd $pipesocket/v4.0.0.0_nlpp_resources
  ( wget @2 ) &
fi
◇
```
Fragment referenced in 11f, 16b, 19b.

3.3 Java

We need to have a Java JDK version 1.8 installed. In other words, when we issue the instruction `javac -version` within the pipeline environment, the response must be something like `javac 1.8.0_131`. We assume that if we find a correct Java 1.8, there will also be a proper `java`. Let us first test whether that is the case. If it is not the case, we can install java if a proper tarball is present in the "snaphot directory".

Let us perform the two tests:

Do we have a proper Java?

⟨ *check presence of javac in 1.8* 10a ⟩ ≡

```
javac -version 2>&1 | grep 1.8 >/dev/null
if
  [ $? == 0 ]
then
  @1="True"
else
  @1="False"
fi
```

◇

Fragment referenced in 11a.

Do we have a tarball to install Java? (in fact, the following macro can be used to check the presence of any tarball in the snapshot directory).

⟨ *check whether a tarball is present in the snapshot* 10b ⟩ ≡

```
if
  [ -e $pipesocket/v4.0.0.0_nlpp_resources/@1 ]
then
  @2="True"
else
  @2="False"
fi
```

◇

Fragment referenced in 11a, 12b, 13b, 17a.

Now do it:

⟨ *set up Java* 11a ⟩ ≡
```
⟨ check presence of javac in 1.8 (11b java_OK ) 10a ⟩
if
  [ ! "$java_OK" == "True" ]
then
  ⟨ check whether a tarball is present in the snapshot (11c jdk-8u131-linux-x64.tar.gz,11d tarball_present ) 10b ⟩
  if
    [ ! "$tarball_present" == "True" ]
  then
    echo "Please install Java 1.8 JDK"
    exit 4
  fi
  mkdir -p $javadir
  cd $javadir
  tar -xzf $pipesocket/v4.0.0.0_nlpp_resources/jdk-8u131-linux-x64.tar.gz
  ⟨ set up java environment 11e ⟩
fi
◇
```
Fragment referenced in 5a.

Adapt the `PATH` variable and set `JAVA_HOME`. Set these variables in the script that will be sourced in the running pipeline and set them in this script because we are going to need Java.

⟨ *set up java environment* 11e ⟩ ≡
```
echo 'export JAVA_HOME=$envdir/java/jdk1.8.0_131' >> $piperoot/progenvv
echo 'export PATH=$JAVA_HOME/bin:$PATH' >> $piperoot/progenvv
export JAVA_HOME=$envdir/java/jdk1.8.0_131
export PATH=$JAVA_HOME/bin:$PATH
◇
```
Fragment referenced in 11a.
Uses: `PATH` 9a, `piperoot` 8d.

3.4 Maven

Currently we need version `3.0.5` to compile the Java sources in some of the modules.

3.5 Maven

Some Java-based modules can best be compiled with Maven. So download and install Maven:

⟨ *download stuff* 11f ⟩ ≡
```
⟨ need to wget (11g apache-maven-3.0.5-bin.tar.gz,11h http://apache.rediris.es/maven/maven-3/3.0.5/binaries
◇
```
Fragment defined by 11f, 16b, 19b.
Fragment referenced in 19a.

First check whether maven is already present in the correct version.

⟨ *check presence of maven in 3.0.5* 12a ⟩ ≡

```
mvn -version | grep "Maven 3.0.5" >/dev/null
if
  [ $? == 0 ]
then
  @1="True"
else
  @1="False"
fi
```

◇

Fragment referenced in 12b.

⟨ *set up Maven* 12b ⟩ ≡

```
⟨ check presence of maven in 3.0.5 (12c mvn_OK ) 12a ⟩
if
  [ ! "$mvn_OK" == "True" ]
then
  ⟨ check whether a tarball is present in the snapshot (12d apache-maven-3.0.5-bin.tar.gz,12e tarball_present ) 10
  if
    [ ! "$tarball_present" == "True" ]
  then
    echo "Please install Maven version 3.0.5"
    exit 4
  fi
  cd $envdir
  tar -xzf /home/huygen/projecten/pipelines/v4.0.0.0_nlpp_resources/apache-maven-
3.0.5-bin.tar.gz
  export MAVEN_HOME=$envdir/apache-maven-3.0.5
  export PATH=${MAVEN_HOME}/bin:${PATH}
fi
```

◇

Fragment referenced in 5a.

⟨ *set variables that point to the directory-structure* 12f ⟩ ≡

```
export MAVEN_HOME=$envdir/apache-maven-3.0.5
export PATH=${MAVEN_HOME}/bin:${PATH}

```

◇

Fragment defined by 8e, 9a, 12f.
Fragment referenced in 8a.
Uses: PATH 9a.

When the installation has been finished, we do not need maven anymore.

⟨ *clean up after installation* 12g ⟩ ≡

```
cd $envdir
rm -rf apache-maven-3.0.5
```

◇

Fragment referenced in 6b.

3.6 Python

Several modules in the pipeline run on Python version 3.6. If the command `python` does not invoke that version, we can try install ActivePython, of which we have a tarball in the snapshot. Versioning in Python is very confusing. It is the official Python policy that `/usr/bin/env python` points to Python version 2 but that scripts with a shabang of `#<!!>! /usr/bin/env python` should be executable by Python version 2 as well as Python version 3.

Our policy will be as follows:

1. When installing, make sure that command `python3` starts a python 3.6 executable. If this is not the case, install ActivePython version 3.6. 2. Generate a virtual environment. 3. Make sure that in our environmen command `python` executes python from the virtual environment.

⟨ *check presence of python3 in 3.6* 13a ⟩ ≡

```
python3 --version 2>&1 | grep "Python 3.6" >/dev/null
if
  [ $? == 0 ]
then
  @1="True"
else
  @1="False"
fi
◇
```

Fragment referenced in 13b.

⟨ *set up Python* 13b ⟩ ≡

```
⟨ check presence of python3 in 3.6 (13c python_OK ) 13a ⟩
if
  [ ! "$python_OK" == "True" ]
then
  ⟨ check whether a tarball is present in the snapshot (13d ActivePython-3.6.0.3600-linux-x86_64-glibc-2.3.6-401
  if
    [ ! "$tarball_present" == "True" ]
  then
    echo "Please install Python version 3.6"
    exit 4
  fi
  ⟨ install ActivePython 14a ⟩
fi
◇
```

Fragment defined by 13b, 16a.
Fragment referenced in 5a.

Unpack the tarball in a temporary directory and install active python in the `env` subdirectory of nlpp. Activepython has a few peculiarities:

- It installs things in subdirectories `bin` and `lib` of the installation-directory (in our case subdirectory `env`).
- It installs scripts with names `python3` and `pip3`. We will make symbolic links from these scripts to `python` resp. `pip`.
- It writes self-starting scripts with a "shabang" containing the full absolute path to the `python3` script. In an attempt to make Active-python relocatable we will rewrite the Shabangs to have them contain `#!/usr/bin/env python`.

⟨ *install ActivePython* 14a ⟩ ≡

```
pytinsdir=‘mktemp -d -t activepyt.XXXXXX‘
cd $pytinsdir
tar -xzf $pipesocket/v4.0.0.0_nlpp_resources/ActivePython-3.6.0.3600-linux-x86_64-
glibc-2.3.6-401834.tar.gz
acdir=‘ls -1‘
cd $acdir
./install.sh -I $envdir
cd $piperoot
rm -rf $pytinsdir
⟨ create python script and pip script 14b ⟩
⟨ rewrite ActivePython shabangs 14c ⟩

◇
```

Fragment referenced in 13b.
Uses: `install` 43a, `piperoot` 8d.

⟨ *create python script and pip script* 14b ⟩ ≡

```
cd $envbindir
rm python
ln -s python3 python
rm pip
ln -s pip3 pip
◇
```

Fragment referenced in 14a.

To rewrite the shabangs of the ActivePython scripts do as follows:

1. Create a temporary directory.
2. Generate an AWK script that replaces the shabang line with a correct one.
3. Generate a script that moves a script from `env/bin` to the temporary directory and then applies the AWK script.
4. Apply the generated script on the scripts in `env/bin`.

⟨ *rewrite ActivePython shabangs* 14c ⟩ ≡

```
transfile=‘mktemp -t trans.XXXXXX‘
rm -rf $transfile
⟨ apply script tran on the scripts in (14d $envbindir,14e $transfile ) 15c ⟩
cd $piperoot
rm -rf $transfile
◇
```

Fragment referenced in 14a.

"../env/bin/tran" 14f≡

```
#!/bin/bash
⟨ get location of the script (14g trandir ) 30a ⟩
workfil=$1
tempfil=$2
mv $workfil $tempfil
gawk -f $trandir/chasbang.awk $tempfil>$workfil
chmod 775 $workfil
◇
```

⟨ *make scripts executable* 15a ⟩ ≡
```
chmod 775  ../env/bin/tran
```
◇

Fragment defined by 6ad, 15a, 22a, 25a, 42c.
Fragment referenced in 42d.

"../env/bin/chasbang.awk" 15b≡
```
#!/usr/bin/gawk -f
BEGIN { shabang="#!/usr/bin/env python3"}

/^\#\!.*python.*/ { print shabang
                    next
                  }
{print}
```
◇

Uses: print 36b.

The following looks complicated. The `find` command applies the `file` command on the files in the `env/bin` directory. The grep command filters out the names of the files that are scripts. it produces a filename, followed by a colon, followed by a description of the type of the file. The `gawk` command prints the filenames only and the `xargs` command applies the `tran` script on the file.

⟨ *apply script tran on the scripts in* 15c ⟩ ≡
```
find @1 -type f -exec file {} + \
   |  grep "Python script" | gawk '{print $1}' FS=':' \
   | xargs -iaap $envbindir/tran aap @2
```
◇

Fragment referenced in 14c.
Uses: print 36b.

3.6.1 Python packages

In order to be reproducible, we must make sure that Python packages are installed in the correct version. Therefore, we will install the packages beforehand and do not leave that to the install-scripts of the modules. Descriptions of the packages can be found on https://pypi.python.org. Install the following packages:

package	version	module
KafNafParserPy	1.87	
lxml	3.8.0	
pyyaml	3.12	
requests	2.18.1	networkx
networkx	1.11	corefbase

⟨ *set up Python* 16a ⟩ ≡

```
pip install KafNafParserPy==1.87
pip install lxml==3.8.0
pip install networkx==1.11
pip install pyyaml==3.12
pip install requests==2.18.1
pip install six==1.10.0.
```

◇

Fragment defined by 13b, 16a.
Fragment referenced in 5a.
Uses: `install` 43a.

3.7 Perl

One of the modules uses perl and needs `XML::LibXML`. However, installation of that package seems to be tricky and seems to depend on the availability of obscure stuff. So, we proceed as follows. First test whether Perl version 5 is present on the host. If that is not the case, check whether we have a tarball named `20160520_nlpp_perllib.tgz` in the snapshot. If that is the case, install Perl from scratch and unpack the tarball. Otherwise, fail, and tell the user to install Perl and `XML::LibXML`.

Install Perl locally, to be certain that Perl is available and to enable to install packages that we need (in any case: `XML::LibXML`).

⟨ *download stuff* 16b ⟩ ≡

⟨ *need to wget* (16c `perl-5.22.1.tar.gz`,16d `http://www.cpan.org/src/5.0/perl-5.22.1.tar.gz`) 9d ⟩

◇

Fragment defined by 11f, 16b, 19b.
Fragment referenced in 19a.

⟨ *check presence of perl in 5* 16e ⟩ ≡

```
perl -v 2>&1 | grep "perl 5," >/dev/null
if
  [ $? == 0 ]
then
  @1="True"
else
  @1="False"
fi
```

◇

Fragment referenced in 17a.

⟨ *set up Perl* 17a ⟩ ≡

```
⟨ check presence of perl in 5 (17b perl_OK ) 16e ⟩
if
  [ "$perl_OK" == "True" ]
then
  ⟨ check whether XML::LibXML is installed (17c lib_OK ) 17f ⟩
  if
    [ ! "$lib_OK" == "True" ]
  then
     perl_OK="False"
  fi
fi
if
  [ ! "$perl_OK" == "True" ]
then
  ⟨ check whether a tarball is present in the snapshot (17d 20160520_nlpp_perllib.tgz,17e tarball_present ) 10b ⟩
  if
    [ ! "$tarball_present" == "True" ]
  then
    echo "Please install Perl version 3.6 and XML::LXML"
    exit 4
  fi
  ⟨ install perl 18a, ... ⟩
fi

◇
```

Fragment referenced in 5a.

⟨ *check whether XML::LibXML is installed* 17f ⟩ ≡

```
perl -MXML::LibXML -e 1 2>/dev/null
if
  [ $? == 0 ]
then
  @1="True"
else
  @1="False"
fi
◇
```

Fragment referenced in 17a.

⟨ *install perl* 18a ⟩ ≡

```
tempdir=‘mktemp -d -t perl.XXXXXX‘
cd $tempdir
tar -xzf $pipesocket/v4.0.0.0_nlpp_resources/perl-5.22.1.tar.gz
cd perl-5.22.1
./Configure -des -Dprefix=$envdir/perl
make
make test
make install
cd $progroot
rm -rf $tempdir

◇
```

Fragment defined by 18abc.
Fragment referenced in 17a.
Uses: `install` 43a.

Make sure that modules use the correct Perl

⟨ *install perl* 18b ⟩ ≡

```
echo 'export PERL_HOME=$envdir/perl' >> $piperoot/progenvv
echo 'export PATH=$PERL_HOME/bin:$PATH' >> $piperoot/progenvv
export PERL_HOME=$envdir/perl
export PATH=$PERL_HOME/bin:$PATH
◇
```

Fragment defined by 18abc.
Fragment referenced in 17a.
Uses: `PATH` 9a, `piperoot` 8d.

Unpack the poor-man tarball with LibXML:

⟨ *install perl* 18c ⟩ ≡

```
cd $envdir/perl/lib
tar -xzf $pipesocket/v4.0.0.0_nlpp_resources/20160520_nlpp_perllib.tgz
◇
```

Fragment defined by 18abc.
Fragment referenced in 17a.

3.8 Download materials

This installer needs to download a lot from different sources:

- Most of the NLP-modules will be built up from their sources in Github. The sources must be cloned.
- Many modules need external resources, e.g. the Alpino tagger. Often these utilities must be downloaded from a location specified by the supplier.
- Many modules use extra resources like model-data, that must be obtained separately.
- Some of the resources are not publicly available. They must be obtained from a pass-word protected URL.
-

Usually downloads are slow, and the duration is only little determined by the resources in the installing computer, but by the network and the performance of the systems from which we download. Therefore, we may speed up by first downloading things, if possible in parallel processes.

We put the following the beginning of the install-script:

⟨ *download everything* 19a ⟩ ≡

```
⟨ download stuff 11f, ... ⟩
echo Waiting for downloads to complete ...
wait
echo Download completed
◇
```

Fragment defined by 9c, 19a.
Fragment referenced in 9b.

4 Shared libraries

When we do not want to rely on what the host can present to us, we need to make our own shared libraries. For the present, we will generate the shared libraries `libxslt` and `libxml2`. We do the following:

1. install autoconf, needed to compile the libs.
2. install libxslt
3. install libxml2

4.1 Autoconf

Gnu autoconf is a system to help configure the Makefiles for a software package. Softwarepackages that use this, supply a file `configure`, `configure.in` or `configure.ac`. To compile and install a package from source we can then perform 1) `./configure --prefix=<environment>`; 2) `make`; 3) `make install`.

Get autoconf:

⟨ *download stuff* 19b ⟩ ≡

```
⟨ need to wget (19c autoconf-2.69.tar.gz,19d http://ftp.gnu.org/gnu/autoconf/autoconf-2.69.tar.gz ) 9d ⟩
◇
```

Fragment defined by 11f, 16b, 19b.
Fragment referenced in 19a.

Install autoconf:

⟨ *set up autoconf* 19e ⟩ ≡

```
autoconfdir=‘mktemp -d -t autoconf.XXXXXX‘
cd $autoconfdir
tar -xzf $pipesocket/v4.0.0.0_nlpp_resources/autoconf-2.69.tar.gz
cd autoconf-2.69
./configure --prefix=$envdir
make
make install
cd $piperoot
rm -rf $autoconfdir
◇
```

Fragment referenced in 5a.
Uses: `install` 43a, `piperoot` 8d.

4.2 libxml2 and libxslt

Compilation and installation of `libxml2` and `libxslt` goes similar, according to the following template:

⟨ *install libxml2 or libxslt* 20a ⟩ ≡

```
shtmpdir=‘mktemp -d -t shl.XXXXXX‘
cd $shtmpdir
git clone @1
packagedir=‘ls -1‘
cd $packagedir
./autogen.sh --prefix=$envdir
make
make install
cd $piperoot
rm -rf $shtmpdir
```

◇

Fragment referenced in 20b.
Uses: `install` 43a, `piperoot` 8d.

⟨ *install shared libs* 20b ⟩ ≡

⟨ *install libxml2 or libxslt* (20c `git://git.gnome.org/libxml2`) 20a ⟩
⟨ *install libxml2 or libxslt* (20d `git://git.gnome.org/libxslt`) 20a ⟩
◇

Fragment referenced in 5a.

4.3 Alpino

Install Alpino as a utility because it is so big, and hard to install on different platforms. Users may choose to install the utilities (and Alpino) by hand and then still install the modules with the script from this file.

Alpino cannot be obtained from an open source repository and there does not seem to be a repository where all the older versions are stored. Therefore, if possible, we will use a copy from our secret archive if that is available. If that is not available, we will download the latest version of Alpino.

⟨ *install Alpino* 21a ⟩ ≡

```
alpinosrc=Alpino-x86_64-Linux-glibc-2.19-21088-sicstus.tar.gz
cd $envdir
if
[ -d "Alpino" ]
then
  echo "Not installing Alpino, because of existing directory $envdir/Alpino"
else
  if
    [ ! -e "$pipesocket/v4.0.0.0_nlpp_resources/$alpinosrc" ]
  then
    echo "Try to install the latest Alpino."
    alpinosrc=latest.tar.gz
    cd $pipesocket/v4.0.0.0_nlpp_resources
    wget http://www.let.rug.nl/vannoord/alp/Alpino/versions/binary/latest.tar.gz
    if
      [ $? -gt 0 ]
    then
      echo "Cannot install Alpino. Please install Alpino in $envdir/Alpino"
      exit 4
    fi
  fi
  cd $envdir
  tar -xzf $pipesocket/v4.0.0.0_nlpp_resources/$alpinosrc
fi
```

◇

Fragment referenced in 5a.
Uses: `install` 43a.

⟨ *set environment parameters* 21b ⟩ ≡

```
export ALPINO_HOME=$envdir/Alpino
```

◇

Fragment defined by 8c, 21b.
Fragment referenced in 8a.
Defines: `ALPINO_HOME` Never used.

5 Installation of the modules

6 Install the modules

We make a separate script to install the modules. By default, the modules will be installed in subdirectory `modules` of the NLPP root directory, but this is not necessarily so.

The script `install-modules` installs modules that are not yet present.

`"../env/bin/install-modules"` 21c≡

```
#!/bin/bash
⟨ get location of the script (21d DIR ) 30a ⟩
cd $DIR
source ../../progenv
⟨ functions of the module-installer 22b ⟩
⟨ install the modules 23f, ... ⟩
```

◇

⟨ *make scripts executable* 22a ⟩ ≡
```
chmod 775  ../env/bin/install-modules
```
◇

Fragment defined by 6ad, 15a, 22a, 25a, 42c.
Fragment referenced in 42d.
Uses: `install` 43a.

Installing a module from Github is very simple:

- Skip installation if the module is already present. Otherwise:
- Clone the module in subdirectory `modules`.
- `cd` to that module and perform script `install`.

⟨ *functions of the module-installer* 22b ⟩ ≡
```
function gitinst (){
   url=$1
   dir=$2
   commitset=$3
   echo "Install $dir" >&2
   cd $piperoot/modules
   if
     [ -e $dir ]
   then
     echo "Not installing existing module $dir"
   else
     git clone $url
     cd $dir
     git checkout $commitset
     ./install
   fi
}
```
◇

Fragment referenced in 21c.
Uses: `install` 43a, `piperoot` 8d.

6.1 Parameters in module-scripts

Some modules need parameters. All modules need a language specification. The language can be passed as exported variable `naflang`, but it can also be passed as argument `-l`. Furthermore, some modules need contact with a Spotlight server. With the arguments `-h` and `-b` the host and port of a running Spotlight-server can be passed.

Let us assess a "Parameter-passing" hierarchy for `run` scripts. Basically a "run" script uses default values encoded in the `run` script itself. These values can be overruled by environment parameters. Both default and enviroment parameter settings can be overruled by options that are provided to the `run` commands.

Let us adhere to the policy that we use short one-letter options in `run` scripts, that can be parsed with `getopts`.

The code to obtain command-line arguments in Bash has been obtained from Stackoverflow. The following fragment reads the arguments `-l language`, `-h spotlighthost` and `-p spotlightport`:

⟨ *start of module-script* 23a ⟩ ≡
⟨ *get location of the script* (23b `DIR`) 30a ⟩
```
cd $DIR
source ../../progenv
```
◇
Fragment referenced in 23dg, 24ad.

6.1.1 Tokeniser

The tokenizer is the simples of the modules. It needs Java version 1.8. On installation it compiles a Java JAR file, and this is used in the run script.

⟨ *install the tokenizer* 23c ⟩ ≡
```
gitinst https://github.com/PaulHuygen/ixa-pipe-tok.git ixa-pipe-
tok 1a69dbbf337aaf7a97bd21dffcfdbd7cb8ab0d83
```
◇
Fragment never referenced.

`"../bin/tok"` 23d≡
⟨ *start of module-script* (23e `$jarsdir`) 23a ⟩
```
cat | ../modules/ixa-pipe-tok/run
```
◇

6.1.2 Topic detection tool.

The topic detection tool uses Java.

⟨ *install the modules* 23f ⟩ ≡
```
gitinst https://github.com/PaulHuygen/ixa-pipe-topic.git ixa-pipe-
topic b2e0ef60badacd90b4f489bdf45f56a1956eb43e
```
◇
Fragment defined by 23fi, 24c.
Fragment referenced in 21c.

`"../bin/m4_topic"` 23g≡
⟨ *start of module-script* (23h `$jarsdir`) 23a ⟩
```
cat | ../modules/ixa-pipe-topic/run
```
◇

6.1.3 Morphosyntactic Parser and Alpino

The morphosyntactic parser is in fact a wrapper around Alpino. We have installed Alpino in section **??**. The morpho-syntactic parser expects Alpino to be located in `$envdir/Alpino`.

⟨ *install the modules* 23i ⟩ ≡
```
gitinst https://github.com/PaulHuygen/morphosyntactic_parser_nl.git morphosyntac-
tic_parser_nl 52a9ad750a71884b3d0a5430c8c0641035367c10
```
◇
Fragment defined by 23fi, 24c.
Fragment referenced in 21c.

"../bin/m4_morpharscript" 24a≡
⟨ *start of module-script* (24b $jarsdir) 23a ⟩
```
cat | ../modules/morphosyntactic_parser_nl/run
```
◇

6.1.4 Pos tagger

Use the pos-tagger from EHU for English documents.

⟨ *install the modules* 24c ⟩ ≡
```
gitinst git@github.com:PaulHuygen/ixa-pipe-pos.git ixa-pipe-
pos dea371c5c8c7cfca98104543e50c7d5daf456b5d
```
◇

Fragment defined by 23fi, 24c.
Fragment referenced in 21c.

"../bin/pos" 24d≡
⟨ *start of module-script* (24e $jarsdir) 23a ⟩
```
cat | ../modules/ixa-pipe-pos/run
```
◇

7 Utilities

7.1 Language detection

The following script `../env/bin/langdetect.py` discerns the language of the NAF document that it reads from standard in. If it cannot find the language, it prints `unknown`. The macro `set the language variable` uses this script to set variable `naflang`. All pipeline modules expect that this veriable has been set.

"../env/bin/langdetect.py" 24f≡
```
#!/usr/bin/env python
# langdetect -- Detect the language of a NAF document.
#
import xml.etree.ElementTree as ET
import sys
import re
xmldoc = sys.stdin.read()
#print xmldoc
root = ET.fromstring(xmldoc)
# print root.attrib['lang']
lang = "unknown"
for k in root.attrib:
   if re.match(".*lang$", k):
     language = root.attrib[k]
print(language)
```
◇

Uses: print 36b.

⟨ *make scripts executable* 25a ⟩ ≡
```
chmod 775  ../env/bin/langdetect.py
```
◇

Fragment defined by 6ad, 15a, 22a, 25a, 42c.
Fragment referenced in 42d.

The module-scripts depend on the existence of variable `naflang`. In most cases this is not a problem because the scripts run in a surrounding script that sets `naflang`. However, a users may occasionally run a module-script stand-alone e.g. to debug. In that case, we can read the language from the NAF, set variable `naflang`, and then run the module-script in a subshell. We assume that variable `scriptpath` contains the path of the script itself.

The macro does the following if `naflang` has not been set:

1. Save the content of standard in to a temporary file.
2. Run `langdetect` with the temporary file as input and set the `naflang` variable.
3. Run the script `$scriptpath` (i.e. itself) with the temporary file as input.
4. Remove the temporary file.
5. Exit itself with the errorcode of the sub-script that it has run.

⟨ *run in subshell when naflang is not known* 25b ⟩ ≡
```
if
  [ -z "${naflang+x}" ]
then
  naffile=‘mktemp -t naf.XXXXXX‘
  cat >$naffile
  naflang=‘cat $naffile | python $envbindir/langdetect.py‘
  export naflang
  cat $naffile | $scriptpath
  result=$?
  rm $naffile
  exit $result
fi
```
◇

Fragment never referenced.
Uses: `naflang` 27b.

⟨ *run only if language is English or Dutch* 25c ⟩ ≡
```
if
  [ ! "$naflang" == "nl" ] && [ ! "$naflang" == "en" ]
then
  exit 6
fi
```
◇

Fragment never referenced.
Uses: `naflang` 27b.

7.2 Run-script and test-script

The script `nlpp` reads a NAF document from standard in and produces an annotated NAF on standard out. The script `test` annotates either a test-document that resides in the nuweb directory or a user-provided document and leaves the intermediate results in its working directory `nlpp/test`, so that, in case of problems, it is easy traceable what went wrong.

The annotion process involves a sequence in which an NLP module reads a file that contains the output from a previous module (or the input NAF file), processes it and writes the result in another file.

The following function, `runmodule`, performs the action of a single module in the sequence. It needs three arguments: 1) the name of the NAF file that the previous module produced or the input file; 2) the name of directory in which the module resides and 3) the name of the output NAF.

The function uses variable `moduleresult` to decide whether it is really going to annotate. If this variable is "false" (i.e., not equal to zero), this means that one of the previous modules failed, and it is of no use to process the input file. In that case, the function leaves `moderesult` as it is and does not process the input-file. Otherwise, it will process the input-file and it sets `moduleresult` to the result of the processing module.

⟨ *function to run a module* 26a ⟩ ≡

```
export moduleresult=0

function runmodule {
  local infile=$1
  local modulecommand=$modulesdir/$2/run
  local outfile=$3
  if
    [ $moduleresult -eq 0 ]
  then
    cat $infile | $modulecommand > $outfile
    moduleresult=$?
    if
      [ $moduleresult -gt 0 ]
    then
      failmodule=$modulecommand
      echo "Failed: module $modulecommand; result $moduleresult" >&2
      exit $moduleresult
    else
       echo "Completed: module $modulecommand; result $moduleresult" >&2
    fi
  fi
}

◇
```

Fragment referenced in 29ab.
Defines: `moduleresult` 29ab, `runmodule` 26b, 27ab.

Use the function to annotate a NAF file that `infile` points to and write the result in a file that `outfile` points to:

⟨ *annotate dutch document* 26b ⟩ ≡

```
runmodule $infile    ixa-pipe-tok       tok.naf
runmodule tok.naf     ixa-pipe-topic       $outfile
◇
```

Fragment never referenced.
Uses: `runmodule` 26a.

Similar for an English naf:

⟨ *annotate english document* 27a ⟩ ≡
```
runmodule $infile    ixa-pipe-tok       tok.naf
runmodule tok.naf     ixa-pipe-topic       top.naf
runmodule top.naf     ixa-pipe-pos       $outfile
```
◇

Fragment referenced in 27b.
Uses: `runmodule` 26a.

Determine the language and select one of the above macro's to annotate the document. In fact, consider the document as an English document unless `naflang` is "nl"

⟨ *annotate* 27b ⟩ ≡
```
naflang=‘cat $infile | /home/huygen/projecten/pipelines/nlpp/env/bin/langdetect.py‘
export naflang
if
 [ "$naflang" == "nl" ]
then
runmodule $infile    ixa-pipe-tok       tok.naf
runmodule tok.naf     ixa-pipe-topic       top.naf
runmodule top.naf     morphosyntactic_parser_nl       $outfile
else
  ⟨ annotate english document 27a ⟩
fi
```
◇

Fragment referenced in 29ab.
Defines: `naflang` 25bc, 28.
Uses: `runmodule` 26a.

Use the above "annotate" macro in a test script and in a run script. The scripts set a working directory and put the input-file in it, and then annotate it.

The test-script uses a special test-directory and leaves it behind when it is finished. If the user specified a language, the script copies a NAF testfile from the nuweb directory as input-file. Otherwise, the script expects the test-directory to be present, with an input-file (named `in.naf`) in it.

⟨ *get a testfile and set naflang or die* 28 ⟩ ≡

```
cd $workdir
naflang=""
if
  [ "$1" == "en" ]
then
  cp $nuwebdir/test.en.in.naf $infile
  export naflang="en"
else
  if
    [ "$1" == "nl" ]
  then
    cp $nuwebdir/test.nl.in.naf  $infile
    export naflang="nl"
  fi
fi
if
  [ -e $infile ]
then
  if
    [ "$naflang" == "" ]
  then
    naflang=‘cat $infile | python $envbindir/langdetect.py‘
  fi
else
  echo "Please supply test-file $workdir/$infile or specify language"
  exit 4
fi
◇
```

Fragment referenced in 29a.
Uses: naflang 27b.

This is the test-script:

"../bin/test" 29a≡
```
#!/bin/bash
DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"
rdir=$(dirname "$DIR")
source $rdir/progenv
oldd=‘pwd‘
workdir=$piperoot/test
mkdir -p $workdir
cd $workdir
infile=in.naf
outfile=out.naf
⟨ get a testfile and set naflang or die 28 ⟩
⟨ function to run a module 26a ⟩
⟨ annotate 27b ⟩
if
  [ $moduleresult -eq 0 ]
then
  echo Test succeeded.
else
  echo Something went wrong.
fi
exit $moduleresult
◇
```
Uses: moduleresult 26a, piperoot 8d.

The run-script **nlpp** reads a “raw” naf from standard in and produces an annotated naf on standard out. It creates a temporary directory to store intermediate results from the modules and removes this directory afterwards.

"../bin/nlpp" 29b≡
```
#!/bin/bash
oldd=‘pwd‘
workdir=‘mktemp -d -t nlpp.XXXXXX‘
cd $workdir
cat >$workdir/$infile
⟨ function to run a module 26a ⟩
⟨ annotate 27b ⟩
if
  [ $moduleresult -eq 0 ]
then
  cat $outfile
fi
cd $oldd
rm -rf $workdir
exit $moduleresult
◇
```
Uses: moduleresult 26a.

8 Miscellaneous

8.1 Locate the path to the script itself

The following macro finds the directory in which the script itself or the sourced script itself is located.

⟨ *get location of the script* 30a ⟩ ≡
```
@1="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd)"
```
◇

Fragment referenced in 5a, 6b, 8a, 14f, 21c, 23a.

8.2 Logging

Write log messages to standard out if variable `LOGLEVEL` is equal to 1.

⟨ *variables of install-modules* 30b ⟩ ≡
```
LOGLEVEL=1
```
◇

Fragment never referenced.

⟨ *logmess* 30c ⟩ ≡
```
if
 [ $LOGLEVEL -gt 0 ]
then
 echo @1
fi
```
◇

Fragment never referenced.

A How to read and translate this document

This document is an example of *literate programming* [2]. 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_nlpp.w`. Figure 2 shows pathways to translate it into print-

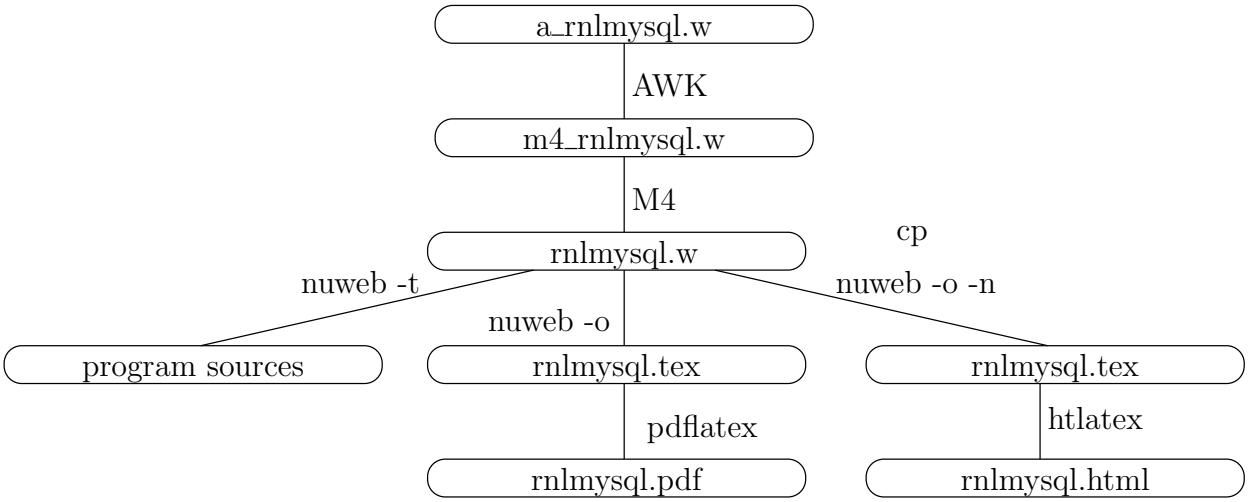

Figure 2: *Translation of the raw code of this document into printable/viewable documents and into program sources. The figure shows the pathways and the main files involved.*

able/viewable documents and to extract the program sources. Table 3 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 3: *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* 32a ⟩ ≡
```
NUWEB=../env/bin/nuweb
```
◇
Fragment defined by 32a, 33a, 35ab, 37c, 39b, 42a.
Fragment referenced in 32b.
Uses: `nuweb` 38d.

A.3 The Makefile for this project.

This chapter assembles the Makefile for the project.

`"Makefile"` 32b≡
⟨ *default target* 32c ⟩

⟨ *parameters in Makefile* 32a, ... ⟩

⟨ *impliciete make regels* 35c, ... ⟩
⟨ *expliciete make regels* 33b, ... ⟩
⟨ *make targets* 32d, ... ⟩
◇

The default target of make is `all`.

⟨ *default target* 32c ⟩ ≡
```
all : ⟨ all targets 32e ⟩
.PHONY : all

```
◇
Fragment referenced in 32b.
Defines: `all` Never used, `PHONY` 36a.

⟨ *make targets* 32d ⟩ ≡
```
clean:
	../env/bin/clean_infrastructure

```
◇
Fragment defined by 32d, 36b, 37a, 40c, 42bd, 43abc.
Fragment referenced in 32b.

The default is, to install nlpp.

⟨ *all targets* 32e ⟩ ≡
```
install◇
```
Fragment referenced in 32c.
Uses: `install` 43a.

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* 33a ⟩ ≡

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

◇

Fragment defined by 32a, 33a, 35ab, 37c, 39b, 42a.
Fragment referenced in 32b.
Defines: `SUFFIXES` Never used.
Uses: `pdf` 36b.

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* 33b ⟩ ≡

```
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 33bd, 34ab, 36a, 37d, 39c, 40b.
Fragment referenced in 32b.
Uses: `nuweb` 38d.

⟨ *clean up* 33c ⟩ ≡

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

◇

Fragment never referenced.
Uses: `nuweb` 38d.

⟨ *expliciete make regels* 33d ⟩ ≡

```
../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 33bd, 34ab, 36a, 37d, 39c, 40b.
Fragment referenced in 32b.
Uses: `nuweb` 38d.

A.5 Pre-processing

To make usable things from the raw input `a_nlpp.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* 34a ⟩ ≡

```
m4_nlpp.w : a_nlpp.w
        gawk '{if(match($$0, "@%")) {printf("%s", substr($$0,1,RSTART-
1))} else print}' a_nlpp.w \
          | gawk '{gsub(/[\\][\$$]/, "$$");print}'  > m4_nlpp.w
```

◇

Fragment defined by 33bd, 34ab, 36a, 37d, 39c, 40b.
Fragment referenced in 32b.
Uses: print 36b.

A.5.2 Run the M4 pre-processor

⟨ *expliciete make regels* 34b ⟩ ≡

```
nlpp.w : m4_nlpp.w inst.m4
        m4 -P m4_nlpp.w > nlpp.w
```

◇

Fragment defined by 33bd, 34ab, 36a, 37d, 39c, 40b.
Fragment referenced in 32b.

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.

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* 35a ⟩ ≡

```
FIGFILES=fileschema directorystructure
```

◇

Fragment defined by 32a, 33a, 35ab, 37c, 39b, 42a.
Fragment referenced in 32b.
Defines: `FIGFILES` 35b.

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* 35b ⟩ ≡

```
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 32a, 33a, 35ab, 37c, 39b, 42a.
Fragment referenced in 32b.
Defines: `FIGFILENAMES` Never used, `PDFT_NAMES` 37a, `PDF_FIG_NAMES` 37a, `PST_NAMES` Never used, `PS_FIG_NAMES` Never used.
Uses: `FIGFILES` 35a.

Create the graph files with program `fig2dev`:

⟨ *impliciete make regels* 35c ⟩ ≡

```
%.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 35c, 40a.
Fragment referenced in 32b.
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 `nlpp.bib`. To create this file, copy the auxiliary file to another file `auxfil.aux`, but replace the argument of the command `\bibdata{nlpp}` 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* 36a ⟩ ≡

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

.PHONY : bibfile
```

◇

Fragment defined by 33bd, 34ab, 36a, 37d, 39c, 40b.
Fragment referenced in 32b.
Uses: PHONY 32c.

A.6.3 Create a printable/viewable document

Make a PDF document for printing and viewing.

⟨ *make targets* 36b ⟩ ≡

```
pdf : nlpp.pdf

print : nlpp.pdf
        lpr nlpp.pdf

view : nlpp.pdf
        evince nlpp.pdf
```

◇

Fragment defined by 32d, 36b, 37a, 40c, 42bd, 43abc.
Fragment referenced in 32b.
Defines: `pdf` 33a, 37a, `print` 15bc, 24f, 34a, `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* 37a ⟩ ≡
```
nlpp.pdf : nlpp.w $(W2PDF)  $(PDF_FIG_NAMES) $(PDFT_NAMES)
        chmod 775 $(W2PDF)
        $(W2PDF) $*
```
◇
Fragment defined by 32d, 36b, 37a, 40c, 42bd, 43abc.
Fragment referenced in 32b.
Uses: `pdf` 36b, `PDFT_NAMES` 35b, `PDF_FIG_NAMES` 35b.

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* 37b ⟩ ≡
```
../nuweb/bin ◇
```
Fragment defined by 7abcd, 37b.
Fragment referenced in 42b.
Uses: `nuweb` 38d.

⟨ *parameters in Makefile* 37c ⟩ ≡
```
W2PDF=../nuweb/bin/w2pdf
◇
```
Fragment defined by 32a, 33a, 35ab, 37c, 39b, 42a.
Fragment referenced in 32b.
Uses: `nuweb` 38d.

⟨ *expliciete make regels* 37d ⟩ ≡
```
$(W2PDF) : nlpp.w $(NUWEB)
        $(NUWEB) nlpp.w
◇
```
Fragment defined by 33bd, 34ab, 36a, 37d, 39c, 40b.
Fragment referenced in 32b.

"../nuweb/bin/w2pdf" 37e≡
```
#!/bin/bash
# w2pdf -- compile a nuweb file
# usage: w2pdf [filename]
# 20170628 at 1119h: Generated by nuweb from a_nlpp.w
NUWEB=../env/bin/nuweb
LATEXCOMPILER=pdflatex
⟨ filenames in nuweb compile script 38b ⟩
⟨ compile nuweb 38a ⟩

◇
```
Uses: `nuweb` 38d.

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* 38a ⟩ ≡
```
NUWEB=/home/huygen/projecten/pipelines/nlpp/env/bin/nuweb
⟨ run the processors until the aux file remains unchanged 39a ⟩
⟨ remove the copy of the aux file 38c ⟩
◇
```
Fragment referenced in 37e.
Uses: nuweb 38d.

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* 38b ⟩ ≡
```
nufil=$1
trunk=${1%%.*}
texfil=${trunk}.tex
auxfil=${trunk}.aux
oldaux=old.${trunk}.aux
indexfil=${trunk}.idx
oldindexfil=old.${trunk}.idx
◇
```
Fragment referenced in 37e.
Defines: auxfil 39a, 41ab, indexfil 39a, 41a, nufil 38d, 41ac, oldaux 38c, 39a, 41ab, oldindexfil 39a, 41a, texfil 38d, 41ac, trunk 38d, 41acd.

Remove the old copy if it is no longer needed.

⟨ *remove the copy of the aux file* 38c ⟩ ≡
```
rm $oldaux
◇
```
Fragment referenced in 38a, 40e.
Uses: oldaux 38b, 41a.

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* 38d ⟩ ≡
```
$NUWEB $nufil
$LATEXCOMPILER $texfil
makeindex $trunk
bibtex $trunk
◇
```
Fragment referenced in 39a.
Defines: bibtex 41cd, makeindex 41cd, nuweb 8e, 32a, 33bcd, 37bce, 38a, 39b, 40d.
Uses: nufil 38b, 41a, texfil 38b, 41a, trunk 38b, 41a.

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* 39a ⟩ ≡

```
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 38d ⟩
  if [ $LOOPCOUNTER -ge 10 ]
  then
    cp $auxfil $oldaux
  fi;
done
◇
```

Fragment referenced in 38a.
Uses: `auxfil` 38b, 41a, `indexfil` 38b, `oldaux` 38b, 41a, `oldindexfil` 38b.

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* 39b ⟩ ≡

```
htmldir=../nuweb/html
htmlsource=nlpp.w nlpp.bib html.sty artikel3.4ht w2html
htmlmaterial=$(foreach fil, $(htmlsource), $(htmldir)/$(fil))
htmltarget=$(htmldir)/nlpp.html
◇
```

Fragment defined by 32a, 33a, 35ab, 37c, 39b, 42a.
Fragment referenced in 32b.
Uses: `nuweb` 38d.

Make the directory:

⟨ *expliciete make regels* 39c ⟩ ≡

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

◇
```

Fragment defined by 33bd, 34ab, 36a, 37d, 39c, 40b.
Fragment referenced in 32b.

The rule to copy files in it:

⟨ *impliciete make regels* 40a ⟩ ≡
```
$(htmldir)/% : % $(htmldir)
        cp $< $(htmldir)/
```
◇
Fragment defined by 35c, 40a.
Fragment referenced in 32b.

Do the work:

⟨ *expliciete make regels* 40b ⟩ ≡
```
$(htmltarget) : $(htmlmaterial) $(htmldir)
        cd $(htmldir) && chmod 775 w2html
        cd $(htmldir) && ./w2html nlpp.w
```
◇
Fragment defined by 33bd, 34ab, 36a, 37d, 39c, 40b.
Fragment referenced in 32b.

Invoke:

⟨ *make targets* 40c ⟩ ≡
```
htm : $(htmldir) $(htmltarget)
```
◇
Fragment defined by 32d, 36b, 37a, 40c, 42bd, 43abc.
Fragment referenced in 32b.

Create a script that performs the translation.

"w2html" 40d≡
```
#!/bin/bash
# w2html -- make a html file from a nuweb file
# usage: w2html [filename]
#  [filename]: Name of the nuweb source file.
# 20170628 at 1119h: Generated by nuweb from a_nlpp.w
echo "translate " $1 >w2html.log
NUWEB=/home/huygen/projecten/pipelines/nlpp/env/bin/nuweb
⟨ filenames in w2html 41a ⟩

⟨ perform the task of w2html 40e ⟩
```
◇
Uses: nuweb 38d.

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* 40e ⟩ ≡
```
⟨ run the html processors until the aux file remains unchanged 41b ⟩
⟨ remove the copy of the aux file 38c ⟩
```
◇
Fragment referenced in 40d.

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* 41a ⟩ ≡
```
nufil=$1
trunk=${1%%.*}
texfil=${trunk}.tex
auxfil=${trunk}.aux
oldaux=old.${trunk}.aux
indexfil=${trunk}.idx
oldindexfil=old.${trunk}.idx
◇
```
Fragment referenced in 40d.
Defines: `auxfil` 38b, 39a, 41b, `nufil` 38bd, 41c, `oldaux` 38bc, 39a, 41b, `texfil` 38bd, 41c, `trunk` 38bd, 41cd.
Uses: `indexfil` 38b, `oldindexfil` 38b.

⟨ *run the html processors until the aux file remains unchanged* 41b ⟩ ≡
```
while
  ! cmp -s $auxfil $oldaux
do
  if [ -e $auxfil ]
  then
   cp $auxfil $oldaux
  fi
  ⟨ run the html processors 41c ⟩
done
⟨ run tex4ht 41d ⟩

◇
```
Fragment referenced in 40e.
Uses: `auxfil` 38b, 41a, `oldaux` 38b, 41a.

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

⟨ *run the html processors* 41c ⟩ ≡
```
$NUWEB -o -n $nufil
latex $texfil
makeindex $trunk
bibtex $trunk
htlatex $trunk
◇
```
Fragment referenced in 41b.
Uses: `bibtex` 38d, `makeindex` 38d, `nufil` 38b, 41a, `texfil` 38b, 41a, `trunk` 38b, 41a.

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* 41d ⟩ ≡
```
tex '\def\filename{{nlpp}{idx}{4dx}{ind}} \input idxmake.4ht'
makeindex -o $trunk.ind $trunk.4dx
bibtex $trunk
htlatex $trunk
◇
```
Fragment referenced in 41b.
Uses: `bibtex` 38d, `makeindex` 38d, `trunk` 38b, 41a.

A.7 Perform the installation

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* 42a ⟩ ≡
```
MKDIR = mkdir -p
```
◇
Fragment defined by 32a, 33a, 35ab, 37c, 39b, 42a.
Fragment referenced in 32b.
Defines: MKDIR 42b.

⟨ *make targets* 42b ⟩ ≡
```
DIRS = ⟨ directories to create 7a, ... ⟩

$(DIRS) :
        $(MKDIR) $@
```
◇
Fragment defined by 32d, 36b, 37a, 40c, 42bd, 43abc.
Fragment referenced in 32b.
Defines: DIRS 42d.
Uses: MKDIR 42a.

⟨ *make scripts executable* 42c ⟩ ≡
```
chmod -R 775  ../bin/*
chmod -R 775  ../env/bin/*
```
◇
Fragment defined by 6ad, 15a, 22a, 25a, 42c.
Fragment referenced in 42d.

The target "sources" unpacks the nuweb file and creates the program scripts, i.e. the scripts that will apply modules on a NAF file and the script `install_modules` that installs the modules themselves and that creates the software environment the the modules need.

⟨ *make targets* 42d ⟩ ≡
```
sources : nlpp.w $(DIRS) $(NUWEB)
        $(NUWEB) nlpp.w
        ⟨ make scripts executable 6a, ... ⟩
```
◇
Fragment defined by 32d, 36b, 37a, 40c, 42bd, 43abc.
Fragment referenced in 32b.
Uses: DIRS 42b.

The "install" target performs the complete installation.

⟨ *make targets* 43a ⟩ ≡

```
install : sources
        ../bin/install-modules
```

◇

Fragment defined by 32d, 36b, 37a, 40c, 42bd, 43abc.
Fragment referenced in 32b.
Defines: `install` 6e, 11a, 12b, 13b, 14a, 16a, 17a, 18a, 19e, 20a, 21a, 22ab, 32e, 43b.

A.8 Test whether it works

The targets testnl and testen perform the test-script (section **??**) to test the dutch resp. english pipeline.

⟨ *make targets* 43b ⟩ ≡

```
testnl : install test.nl.in.naf
        rm -rf ../test
        mkdir ../test
        cd ../test && ../bin/test nl

testen : install test.en.in.naf
        rm -rf ../test
        mkdir ../test
        cd ../test && ../bin/test en
```

◇

Fragment defined by 32d, 36b, 37a, 40c, 42bd, 43abc.
Fragment referenced in 32b.
Defines: `testen` Never used, `testnl` Never used.
Uses: `install` 43a.

A.9 Restore paths after transplantation

When an existing installation has been transplanted to another location, many path indications have to be adapted to the new situation. The scripts that are generated by nuweb can be repaired by re-running nuweb. After that, configuration files of some modules must be modified.

⟨ *make targets* 43c ⟩ ≡

```
transplant :
        touch a_nlpp.w
        $(MAKE) sources
        ../env/bin/transplant
```

◇

Fragment defined by 32d, 36b, 37a, 40c, 42bd, 43abc.
Fragment referenced in 32b.

B References

B.1 Literature

References

[1] Rodrigo Agerri, Itziar Aldabe, Zuhaitz Beloki, Egoitz Laparra1, Maddalen Lopez de Lacalle1, German Rigau, Aitor Soroa, Antske Fokkens, Ruben Izquierdo, Marieke van Erp, Piek Vossen, Christian Girardi, and Anne-Lyse Minard. Event detection, version 2, deliverable d4.2.2. Technical report, University of the Basque Country, IXA NLP group, feb 2015. http://www.newsreader-project.eu/files/2012/12/NWR-D4-2-2.pdf.

[2] 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

⟨ filenames in w2html 41a ⟩ Referenced in 40d.
⟨ find the nlpp root directory 8d ⟩ Not referenced.
⟨ function to run a module 26a ⟩ Referenced in 29ab.
⟨ functions of the module-installer 22b ⟩ Referenced in 21c.
⟨ get a testfile and set naflang or die 28 ⟩ Referenced in 29a.
⟨ get location of the script 30a ⟩ Referenced in 5a, 6b, 8a, 14f, 21c, 23a.
⟨ impliciete make regels 35c, 40a ⟩ Referenced in 32b.
⟨ init make_infrastructure 6f, 9b ⟩ Referenced in 5a, 6b.
⟨ install ActivePython 14a ⟩ Referenced in 13b.
⟨ install Alpino 21a ⟩ Referenced in 5a.
⟨ install libxml2 or libxslt 20a ⟩ Referenced in 20b.
⟨ install perl 18abc ⟩ Referenced in 17a.
⟨ install shared libs 20b ⟩ Referenced in 5a.
⟨ install the modules 23fi, 24c ⟩ Referenced in 21c.
⟨ install the tokenizer 23c ⟩ Not referenced.
⟨ logmess 30c ⟩ Not referenced.
⟨ make scripts executable 6ad, 15a, 22a, 25a, 42c ⟩ Referenced in 42d.
⟨ make targets 32d, 36b, 37a, 40c, 42bd, 43abc ⟩ Referenced in 32b.
⟨ need to wget 9d ⟩ Referenced in 11f, 16b, 19b.
⟨ parameters in Makefile 32a, 33a, 35ab, 37c, 39b, 42a ⟩ Referenced in 32b.
⟨ perform the task of w2html 40e ⟩ Referenced in 40d.
⟨ remove the copy of the aux file 38c ⟩ Referenced in 38a, 40e.
⟨ rewrite ActivePython shabangs 14c ⟩ Referenced in 14a.
⟨ run in subshell when naflang is not known 25b ⟩ Not referenced.
⟨ run only if language is English or Dutch 25c ⟩ Not referenced.
⟨ run tex4ht 41d ⟩ Referenced in 41b.
⟨ run the html processors 41c ⟩ Referenced in 41b.
⟨ run the html processors until the aux file remains unchanged 41b ⟩ Referenced in 40e.
⟨ run the processors until the aux file remains unchanged 39a ⟩ Referenced in 38a.
⟨ run the three processors 38d ⟩ Referenced in 39a.
⟨ set environment parameters 8c, 21b ⟩ Referenced in 8a.
⟨ set up autoconf 19e ⟩ Referenced in 5a.
⟨ set up Java 11a ⟩ Referenced in 5a.
⟨ set up java environment 11e ⟩ Referenced in 11a.
⟨ set up Maven 12b ⟩ Referenced in 5a.
⟨ set up Perl 17a ⟩ Referenced in 5a.
⟨ set up Python 13b, 16a ⟩ Referenced in 5a.
⟨ set variables that point to the directory-structure 8e, 9a, 12f ⟩ Referenced in 8a.
⟨ start of module-script 23a ⟩ Referenced in 23dg, 24ad.
⟨ test presence of command 6e ⟩ Referenced in 6f.
⟨ variables of install-modules 30b ⟩ Not referenced.

C.3 Variables

`all`: 32c.
`ALPINO_HOME`: 21b.
`auxfil`: 38b, 39a, 41a, 41b.
`bibtex`: 38d, 41cd.
`DIRS`: 42b, 42d.
`fig2dev`: 35c.
`FIGFILENAMES`: 35b.
`FIGFILES`: 35a, 35b.
`indexfil`: 38b, 39a, 41a.
`install`: 6e, 11a, 12b, 13b, 14a, 16a, 17a, 18a, 19e, 20a, 21a, 22ab, 32e, 43a, 43b.
`makeindex`: 38d, 41cd.
`MKDIR`: 42a, 42b.
`moduleresult`: 26a, 29ab.
`naflang`: 25bc, 27b, 28.