2. obligatoriske innlevering, høsten 2014

{Jonathan Feinberg, Joakim Sundnes}
{jonathf,sundnes}@simula.no

September 29, 2014

Innleveringskrav

Innlevering skal skje ved opplasting til github, som for den første obligatoriske oppgaven. Generelle krav til innleveringen er oppgitt på emnesidene. I tillegg kommer eventuelle krav til hver enkelt oppgave, som er spesifisert i oppgaveteksten.

Merk at denne obligatoriske oppgaven er omfattende og relativt krevende. Det anbefales sterkt å starte arbeidet så raskt som mulig. Ved å starte tidlig øker man også muligheten for å få hjelp av gruppelærere underveis.

Frist for innlevering: Fredag 24. oktober kl 16.00

Oversikt

I denne obligen ønsker vi å utvikle en automatisk preprosessor for LaTeX-dokumenter. Preprosessoren er et Python-skript som kjøres på LaTeX-koden før koden kompileres, og som skal gi tilleggsfunksjonalitet utover vanlig LaTeX.

For å beskrive hva preprosessoren skal gjøre, la oss begynne med et eksempel. Ta utgangspunkt iat man skriver et LaTeX-dokument `text.tex`. I dette dokumentet ønsker vi å inkludere en kode-snutt fra en kode-fil som er i samme mappe. Enkleste løsningen er å kopiere over innholdet fra filen manuelt, plassere snuttene `\begin{verbatim}` og `\end{verbatim}` på hver side av innlagt kode og si seg ferdig. Problemet med dette er at hvis man oppdaterer koden senere, må man samtidig gjøre ekstraarbeidet med å oppdatere LaTeX-dokumentet i samme slengen. I større prosjekter (som f.eks. masteroppgaver) kan dette fort bety veldig mye unødig ekstra-arbeid. Istedenfor å skrive inn ting manuelt, kand det være hensiktsmessig å importere ting direkte fra LaTeX. Det finnes noen løsninger internt i LaTeX, men de er klumsete og ikke spesielt egnet til det behovet en informatiker typisk vil ha.

Deloppgavene under beskriver forskjellig funksjonalitet som skal legges inn i preprosessoren kan. Det er 13 oppgaver, hvorav 7 er obligatoriske. Av de resterende 6, skal man velge seg ut minimum 3 oppgaver. De obligatoriske oppgavene er alle markert med '*'.

*Oppgave 1: Kodeimport

Istedetfor å lime kildekode manuelt inn i et LaTeX-dokument, la oss anta at vi kan skrive inn følgende snutt der vi ønsker å legge inn kode:

```
%@import script_example.py ( *out =(.|\n)*?return out)
```

Preprosessoren dere skal utvikle skal kunne finne snutter som dette inne i et LaTeX-dokument, åpne filen den referer til (`script_example.py`), finne kode som matcher regex (`( *out =(.|\n)*?return out)`) og lime dette inn som en `verbatim`-blokk i dokumentet hvor `%@import` ble kalt fra. LaTeX-koden som genereres av kommandoen over kan for eksempel se slik ut:

```
\begin{verbatim}
    out = 1.
    for o in xrange(2, order+1):
        out *= o
    return out
\end{verbatim}
```

Bruk gjerne flagget `re.MULTILINE` for å gjøre det regulære uttrykket enklere. Preprosessoren skal kunne kjøres som et selvstendig program fra kommandolinjen. Input skal være en utvidet LaTeX-fil, det vil si en LaTeX-fil med eventuelle `%@import`-kommandoer i teksten. For å holde orden på filtypene kan man gjerne go denne fila en egen endelse, for eksempel `.xtex, .3331tex` eller liknende. Output skal være en ren LaTeX-fil som vi kan prosessere med `pdflatex, latex` etc. For et eksempel på hva preprosessoren gjør, se filene `tex_before.xtex`, `tex_after.tex` og `script_example.py`.

Navn på skript: `prepro.py`

*Oppgave 2: Eksekvering av skript

I tillegg til å vise biter av en kode er det ofte interessant å vise hva et skript kan gjøre. Programmet dere skal lage må kunne kjøre et eksternt skript, returnere utskrift av kjøring og lime dette inn i LaTeX-dokumentet. I vårt eksempel kan vi forestille oss at snutten:

```
%@exec python script_example.py 4
```

gir oss utskriften:

```
\begin{verbatim}
$ python script_example.py 4
24.0
\end{verbatim}
```

Med andre ord, istedenfor å importere, skal koden kjøres og resultatet av kjøringen skal limes inn i dokumentet. Her kan det være nyttig å bruke `subprocess.Popen`-modulen. Ta en titt på oppgave 7 for et eksempel på bruk av denne modulen.

Navn på skript: tilleggfunksjonalitet i `prepro.py`

Oppgave 3: Kodeformatering

Koden over bruker `verbatim` som utgangspunkt for fremvisning av kode. Det er imidlertid mulig å gi koden mye penere formatering, med fargekoder og liknende. Siden vi har automatisert prosessen med å lime inn kode kan vi også automatisere formateringen. Det er ikke forventet at man skal kunne lage dette selv. Istedenfor kan man finne en mal i filen `format_template`, som inneholder oppskrifter på kodeformatteringen som er brukt i dette dokumentet. Et eksempel på LaTeX-kode før og etter preprosessering kan du se i filene `tex_before.tex` og `tex_after.tex`.

Kode-import er som brukt gjennom dokumentet i lyseblå og ser slik ut etter kompilering av det endelige LaTeX-dokumentet:

```
from __future__ import braces
```

filen inneholder også en tilsvarende snutt for pent formatert kode-eksekvering. Denne vil se slik ut etter kompilering:

Terminal

```
$ echo "help I'm stuck in a small shell-script"
help I'm stuck in a small shell-script
```

Navn på skript; tilleggsfunksjonalitet til `prepro.py`

Oppgave 4: Vanlig innliming av kode

Av og til er det ønskelig å lime inn kodesnutter på vanlig måte, uten å importere fra en ekstern fil. Dette kan vi gjøre i LaTeXmed `verbatim`, men vi ønsker å forenkle det ved å legge funksjonaliteten inn i preprosessoren. Ved å ikke legge på et filnavn etter `%@import` skal koden forvente at linjene etter er en kodesnutt, fram til `%@` er funnet. Her brukes `%@` på egen linje til å indikere at verbatim kode er slutt. Alt mellom `%@code` og `%@` skal legges inn i en `verbatim`-blokk, eventuelt med pen formatering som angitt i oppgaven over.

For eksempel:

```
%@import
Dette er en viktig kode-snutt.
%@
```

skal være nok for å plassere setningen inne i den formaterte lyse-blå verbatim-klassen. Tilsvarende vil:

```
%@exec
$ echo "Just another Perl hacker,"
%@
```

resultere i en utskrift:

Terminal

```
$ echo "Just another Perl hacker,"
```

Navn på skript: tilleggsfunksjonalitet til `prepro.py`

Oppgave 5: Kode-eksekvering

I denne del-oppgaven skal dere lese eksekverbar Python og Bash kode fra LaTeX-dokumentet, kjøre koden og lime utskrift fra kjøringen tilbake i dokumentet. Merk forskjellen fra Oppgave 2, hvor input var et eksternt skript som skulle kjøres. Her ønsker vi å legge inn gyldige Python- og Bash-linjer direkte i LaTeX-dokumentet, og få preprosessoren til å kjøre disse. For eksempel vil

```
%@python fake_name.py fake_arg
print 2+2
%@
```

gi resultatet

Terminal

```
$ python fake_name.py fake_arg
4
```

Tilsvarende vil

```
%@bash fake_name.sh fake_arg
echo "2+2" | bc
%@
```

gi resultatet

Terminal

```
$ bash fake_name.sh fake_arg
4
```

I dette eksempelet brukes `%@` både som start og slutt på blokk.

Oppgave 6: Skjult tekst

I denne oppgaven ønsker vi å kunne gjemme eller trekke frem tekst i LaTeX-dokumentet ved behov. For å definere dette må vi først introdusere variable. Disse defineres ved hjelp av `%@var`:

```
%@var lang python
```

Her vil variabelen `lang` få verdien `python`. Denne snutten er tenkt å være plassert i begynnelsen av dokumentet, slik at den lett kan justeres av brukeren, men den kan i utgangspunktet plasseres hvor som helst.

Med denne variablen definert kan man bruke `show` til å inkludere kode etter ønske. For eksempel:

```
%@show lang matlab
%@verb
for i=1:10
    i
end
%@
%@end
%@show lang python
%@verb
for i in xrange(10):
    print i+1
%@
%@end
```

I dette eksempelet vises den første kodesnutten hvis `lang` er satt til `matlab`, mens i den andre vil vises hvis tilsvarende `lang` er satt til `python`.

Det skal være mulig å legge andre `%@`-komandoer inne i blokker, så `%@show` skal eksplisitt avsluttes med `%@end`.

I tillegg til `show` skal man også lage en `hide`-funksjon som gjør det motsatte og gjemmer tekst når en variabel er satt.

Navn på skript: tilleggsfunksjonalitet i `prepro.py`

*Oppgave 7: Kompilering av preprosessert LaTeX-fil

Bruk Python-modulen `subprocess.Popen` for kompilere LaTeX-dokumentet:

```
>>> import subprocess
>>> proc = subprocess.Popen(
"pdflatex -file-line-error -interaction=nonstopmode path/to/file",
shell=True, stdout=subprocess.PIPE)
>>> out, err = proc.communicate()
```

De to variablene `out` og `err` inneholder standard-utskrift og feilmeldinger fra kjøring.

Siden argumentet `-file-line-error` er inkludert inneholder utskriften alle feilmeldinger på formatet:

```
filnavn:linjenummer:feilmelding
```

Hent ut alle feilmeldingene samt de to siste linjene i loggen og skriv dem ut til skjerm istedenfor den vanlige LaTeX-utskriften. En passende LaTeX-fil for å teste funksjonaliteten er `tex_error.tex`.

For de som er kjent med LaTeXog bryr seg om slikt: Hvis man ønsker å bruke sin egen variant av LaTeX, som `latex`, `xetex` eller `luatex` er dette også lov, men da må man være tydelig i rapporten på hva som gjør hva. Det skal være enkelt for en utenforstående å se hva man har gjort.

Navn på script: `compile.py`

*Oppgave 8: Inkludering av filer

LaTeXhar funksjonenen `\include` som lar deg importere andre LaTeXinnhold fra andre filer. Filene som inkluderes inneholder vanlig LaTeX-kode, og effekten er

den samme som om koden limes rett inn i hoved-dokumentet. Inkludering skjer med følgende linje:

```
\include{path/to/file}
```

Vi antar nå at også underfilene inneholder tilleggsfunksjonalitet, som må kjøres gjennom preprosessoren. Preprosessoren må derfor kjenne igjen `include`-kommandoen og behandle også disse filene riktig. Dette kan gjøres ved å lime innholdet fra filene inn i hoved-fila før de andre stegene i preprosesseringen skjer.

Navn på skript: tilleggsfunksjonalitet til `prepro.py`

Oppgave 9: Filtre

Istedetfor å samle alt i en fil, kan det være hensiktsmessig å å beholde filene separate, og kjøre preprosessoren på hver enkelt fil. Lag en mappe (eller et mappetre) og legg alle preposesserte inkluderte filer inn i denne mappen. Hoveddokumentet må også endres slik at de riktige filnavnene blir inkludert. Pass på at like navn på filer som er i forskjellige mapper ikke skaper problemer, og at alle LaTeX-dokumentene i mappen preprosesseres før man kompilerer.

Oppgave 10: Linjenummerering

Linjenummereringen referert til i oppgave 7 vil referere til den LaTeX-fila som ermanipulert av `prepro.py`. Normalt vil brukeren redigere i det opprinnelige dokumentet (før preprosessering), og det vil være mer hensiktsmessig om linjenummer refererer till dette dokumentet. I denne oppgaven skal man løse dette problemet ved å erstatte linjenummer i utskriften med tilsvarende linjer i det originale dokumentet.

Hint: For å gjøre det enklere å finne igjen linjenummer i den opprinnelige fila kan det være lurt om preprosessoren legger inn en LaTeX-kommentar på slutten av hver linje i prosessert fil:

```
\documentclass{article}%1:/path/to/file
%2:/path/to/file
\usepackage[T1]{fontenc}%3:/path/to/file
\usepackage[utf8]{inputenc}%4:/path/to/file
%5:/path/to/file
```

Her viser kommentaren til hvilken fil denne linjen er hentet fra, og hvilket linjenummer det var i denne fila. Disse linjene kan dermed bli funnet igjen etter preprosesering. Merk at hvis man har laget et filtre, må man også passe på at navnene refererer til riktig fil.

For denne oppgaven er det hensiktsmessig at skriptet `compile.py` også kaller preprosessoren `prepro.py`, men de to programmene bør også kunne kjøres uavhengig av hverandre.

Navn på skript: tilleggsfunksjonalitet i `prepro.py` og `compile.py`

*Oppgave 11: Front-end til preprosessor

Fra et brukerperspektiv kan det være greit å ha et velfungerende Bash-brukergrensesnitt til preprosessoren. Implementer dette ved hjelp av `argparse`-modulen. Følgende funksjoner forventes å være inkludert:

- Mulighet for velge navn på preprosessert fil. (mappe)
- Verbose-mode skriver ut på skjerm alle operasjoner som blir gjort.
- Mulighet for å bytte mellom enkel og fancy verbatim utskrift.
- Skru av og på `-interaction=nonstopmode` i pdflatex (eller tilsvarende).

Navn på skript: tilleggsfunksjonalitet til `prepro.py` og `compile.py`

*Oppgave 12: Testing og dokumentasjon

Alle interne funksjoner skal dokumenteres med doc-tester. I tillegg skal man implementere en test-suite som tester følgende kriterier:

- Kodeimport
- Eksekvering av skript
- Kode-eksekvering
- Skjult tekst
- Kompilering
- Linjenummerering

Testing av kode som ikke er implementert er selvsagt untatt.

*Oppgave 13: Rapport

Som angitt på emnesidene skal innleveringen inneholde en rapport skrevet i LaTeX. Rapporten skal inkludere relevante kodesnutter og kjøringer, som både viser til og i praksis bruker programmet du nettopp har laget. Det er for eksempel lov til å lage sikkelig dokumentasjon i et program, og bruke `%@import` til å inkludere den i rapporten.