INF 2820 V2016: Obligatorisk innleveringsoppgave 3

- **Besvarelsene skal leveres i devilry innen torsdag 21.4 kl 18.00**
- **Filene det vises til finner du i**
    - **/projects/nlp/inf2820/cfg**

Oppgave 1: Shift-reduce-effektivisering (40 poeng)

Før du begynner på denne oppgaven bør du ha arbeidet deg gjennom oppgave 2 og oppgave 3 på ukesoppgavene til 8.3, sett 7!

a) (**20 poeng**)Vi har sett at Shift-Reduce-parsing for generelle kontekstfrie grammatikker er ineffektiv. Vi skal se på et grep vi kan ta for å gjøre den noe mindre ineffektiv. En forutsetning for at vi skal kunne gjøre dette, er at grammatikken har en spesiell form der hver regel er på en av følgende to former:

1. Høyresiden inneholder null eller flere ikke- terminaler, men ingen terminaler.
2. Høyresiden inneholder nøyaktig en terminal (og ikke noe annet).

Dette er et vanlig format for grammatikker for naturlige språk. Jeg har ikke funnet noe navn for denne i litteraturen, men la oss si at en slik grammatikk er på S*tandardform*. Vi ser spesielt at en grammatikk på Chomsky Normal Form også er på Standardform. Det følger at ethvert kontekstfritt språk har en grammatikk på Standardform.

Vi vil først sjekke om en grammatikk er på Standardform. NLTK inneholder en metode for å sjekke dette.

```
>>> grammar.is_nonlexical()
```

For å forstå `grammar.is_nonlexical()`, skal vi prøve den på forskjellige eksempelgrammatikker. Sjekk først at demogrammatikken i `sr_recognizer` gir `True` på denne testen. Legg etter tur til en av følgende regler til grammatikken, og se om nå `grammar.is_nonlexical()`:

```
NP -> 'New' 'York'
NP -> NP 'og' NP
ADJ ->
```

Merk at du ikke kan bruke NLTK-prosedyren `grammar.is_lexical()` for dette. Prosedyrene `grammar.is_lexical()` og `grammar.is_nonlexical()` er ikke motsatte. En grammatikk kan få `False` for begge prosedyrene, en annen grammatikk kan få `True` for begge. Du kan se dokumentasjonen f.eks. ved å kjøre

```
>>> help(grammar.is_nonlexical)
```

Siden SR-algoritmen er en ren bottom-up algoritme, vil vi ikke bruke grammatikker med tomme høyresider. Det tilsvarer at første betingelse skjerpes til

1+. Høyresiden inneholder ingen terminaler, bare en eller flere ikke- terminaler.

En grammatikk hvor hver regel er på form (1+) eller på form (2) kan vi si er på *Standardform+*.

Gitt at grammatikken har denne formen. Da vet vi at hvert «shift» må følges av en (leksikalsk) reduksjon, siden konstellasjoner som

```
NP som kjente | læreren som smilte sov
```

aldri kan føre til suksess. Det vil ikke bli noen senere mulighet til å redusere «som», siden den alltid vil ha noe til høyre for seg. Vi kan derfor modifisere SR-algoritmen slik at ethvert «shift» umiddelbart følges av en reduksjon. For å gjøre forskjellen mellom den opprinnelige og den modifiserte algoritmen tydeligere, har vi laget et eksempel i Eksempel 1.

Du skal først modifisere fila `sr_recognizer.py` slik at ethvert «shift» følges umiddelbart av en leksikalsk reduksjon. Kall resultatet `sr_recognizer_modifies.py`.

Deretter skal du modifisere `sr_parser.py` på tilsvarende måte. For resten av oppgaven vil det være hensiktsmessig å beholde alle prosedyrene i `sr_parser.py` og så utvide fila med modifiserte prosedyrer som `sr_parse_modified()` og `parse_modified().` Kall fila for `sr_parser_modified.py.`

Det er ikke et krav at trace-utskriftene blir riktige. Men for de som ønsker en litt større utfordring, anbefales også å se på dette.

For å teste algoritmen din skal du helst bruke grammatikken `pp.cfg` som du lagde på oppgave 2 i obligatorisk oppgavesett 2. Hvis du trenger en ny start, kan du bruke eksempelløsningen som legges ut. Sjekk på noen setninger at den modifiserte parseren gir de samme analysene som den opprinnelige.

Innlevering: Filene `sr_recognizer_modified.py` og `sr_parser_modified.py`. Noen eksempler som viser at `sr_parser_modified()` og `sr_parser()` lager de samme trærne.

b) (**10 poeng**) For å undersøke hvor mye vi har spart, vil vi bruke funksjonen `timing()` som er inkludert i `sr_parser.py`. Den kan bli kalt for eksempel som

```
>>> timing("""sr_parse(grammar,"Kari likte dyret i huset".split())""")
```

Vi skal sammenlikne tidsforbruket for `sr_parse()` med `sr_parse_modified()`. For begge parserne, se hva som skjer:

- Med eksempelet tre linjer over
- Når du legger til en PP til, for eksempel «Kari likte dyret i huset ved vannet».
- Når du legger til 2 PP-er, deretter 3 PP-er og til slutt 4 PP-er til den første setningen slik at det er til sammen 5 PP-er.

Innlevering: De 5 eksempelsetningene du brukte. Tidsforbruket for de 5 eksempelsetningen og de to prosedyrene i en forståelig 2*5 tabell.

c) (**10 poeng**) Men selv den raffinerte parseren blir langsom når setningene blir lengre. Først repeterer vi litt notasjon. Prøv (Merk en blank før *bak*.)

```
>>> a="Kari likte dyret"+2*" bak huset"
>>> a
```

Deretter prøv

```
>>> timing("""sr_parse_modified(grammar, ("Kari likte dyret"+2*" bak
huset").split())""")
```

og bytt så i tur og orden ut 2 med 3, 4, …, 8

Innlevering: Tidsforbruket for kjøringer med 2 til 8 PP-ledd og forklaring av hvordan tabellen skal leses

Eksempel1:
For å vise forskjellen mellom den opprinnelige SR-parseren og den modifiserte, kan vi ta et eksempel.

Anta at det er to regler med 'vannet' på høyreside

```
V -> 'vannet'
N -> 'vannet'
```

Da vil konstellasjonen

```
DET | vannet er rent
```

med den opprinnelige algoritmen ha 3 mulige fortsettelser to trekk frem:

Alt. 1

```
DET | vannet er rent
DET vannet | er rent
DET V | er rent
```

Alt. 2

```
DET | vannet er rent
DET vannet | er rent
DET N | er rent
```

Alt. 3

```
DET | vannet er rent
DET vannet | er rent
DET vannet er | rent
```

Med den modifiserte algoritmen vil det i stedet være to mulige fortsettelser (et trekk frem):

Alt. 4

```
DET | vannet er rent
DET V | er rent
```

Alt. 5

```
DET | vannet er rent
DET N | er rent
```

.

Oppgave 2: Utviding av grammatikken (20 poeng)

Vi skal nå utvide grammatikk `pp.cfg` (din egen eller modelløsningen) med en del flere konstruksjoner. For det første skal vi utvide med koordinasjon av setninger som i (a), NP-er som i (b) og VP-er som i (c). For koordinasjon bør du tilstrebe mest mulig generelle (og korrekte) regler, som dekker flest mulig tilfeller.

a) Kari smilte og barnet sov.
b) Ola og Kari smilte.
c) Kari smilte og ga barnet dyret.

Dernest skal vi inkludere noen relativsetninger. Vi vil her nøye oss med bare noen former for relativsetninger og med forholdsvis lite prinsipielle løsninger. De to formene av relativsetninger vi vil se på, er på formen (d) som vi finner i (e) og (f), og på formen (g) som vi finner i (h) og (k).

d) som VP
e) Kari som smilte likte dyret
f) Kari likte barnet som likte dyret **som sov**

g) som NP TV
h) Dyret som Kari likte sov
i) Kari likte dyret som barnet som sov likte

Kall den utvidete grammatikken for `my.cfg`. Du kan nå teste grammatikken med `sr_parser.py`. Hvor mange trær tilordner grammatikken til setningen

> Setning 1: Ola som fortalte at Kari sov og smilte likte barnet og huset ved vannet.

Innlevering: Grammatikken `my.cfg`. Svar på spørsmål om hvor mange analyser setningen får og lever trærne som grammatikken tilskriver setningen.

Oppgave 3: CNF (20 poeng)

Vi vil nå skifte til en tabellbasert parser og vi vil bruke CKY-algoritmen. Da må grammatikken være på Chomsky-normalform (CNF). Finn en grammatikk på CNF som dekker de samme konstruksjonene som `my.cfg` og som gir like mange forskjellige trær til en setning som det `my.cfg` gjør. Lagr den i en fil `cnf.cfg`. Det skal også være en naturlig korrespondanse mellom trærne de to grammatikkene tilskriver til en setning. Sjekk at grammatikken din er på riktig form ved

```
>>> <grammatikknavn>.is_chomsky_normal_form()
```

Test `cnf.cfg` på en del setninger, bl.a.

> Setning 2: Ola fortalte at Kari ga barnet dyret i huset.

Innlevering: Grammatikken `cnf.cfg`. Trærne som `cnf.cfg` tilordner til setning2 og trærne `my.cfg` tilordner til setning 2 sammen med en forklaring av hvilke par av trær som svarer til hverandre.

Oppgave 4: CNF (0 poeng)

Dette er en oppgave for de som ønsker litt større utfordringer. Lag et program som omformer en generell CFG til en CNF. Test den på my.cfg og se at resultatet stemmer. (Jfr. Exercise 13.1 i Jurafsky & Martin). En belønning for å gjøre denne oppgaven er at du vil huske algoritmen mye bedre.

Innlevering: Kode. Resultatet av å kjøre programmet på my.cfg.

Oppgave 5: CKY (20 poeng)

Vi kan nå bruke CKY-algoritmen. Til eksamen blir du bedt om å parse for hånd, og det er lurt å trene på dette. Konstruer CKY-tabell for setning 2 med grammatikk `cnf.cfg.` Du kan deretter bruke programmet `cky.py`. til å teste løsningen din.

Innlevering: CKY-tabell for setning 2.

Vi kan nå teste tidsforbruket for CKY-algoritmen og sammenlikne med SR-parseren. Men siden vi også har endret grammatikkene, vil vi først se på tidsforbruket for SR-parseren og om utvidelsen av `pp.cfg` til `my.cfg` eller omformingen til `cnf.cfg` påvirker tidsforbruket med denne parseren. Test

```
>>> timing("""sr_parse_modified(…, ("Kari likte dyret"+6*" bak
huset").split())""")
```

med de tre forskjellige grammatikkene og `sr_parse_refined.py`.

Dernest test tidsforbuket for prosedyren `cky` med samme setning og samme grammatikk.

Gjenta deretter testingen av tidsforbruket når tallet «6» skiftes med 10, 20, 30, 40 og 50.

Innlevering: To tabeller. En som viser tidsforbruket for de tre grammatikkene for samme setning og `sr_parse_modified()`. Dernest en tabell som viser tidsforbruket for `cky` med de forskjellige setningslengdene.

SLUTT