Super-CGN, een CGN-parser op basis van Supertagging. Project in het kader van het vak Lexicale Kennisrepresentatie 2002

Marthe Dekker, Wieteke Dijkman, Stephan van der Feest, Arjen Hommerson, Vincent Jacobs, Khiet Truong, Eelco de Vries

October 14, 2002

Contents

1 Inleiding

1.1 Doel

Doel van het project SuperCGN is het ontwikkelen van een programma of serie van programma's die een ge-PoS-tagged corpus syntactisch ontleedt. Hierbij wilden we gebruik maken van de mogelijkheden die de lexicale kennisrepresentatietaal DATR (Default ATtribute Representation) biedt. Het is een project in het kader van de cursus Lexicale kennisrepresentatie. In deze cursus hebben we bekeken hoe we met grote lexicale databestanden kunnen omgaan aan de hand van de gespecialiseerde lexicale kennisrepresentatietaal DATR. We hebben kennis gemaakt met de eigenschappen en mogelijkheden van deze taal en een bijbehorende evaluatiesemantiek door de volgende literatuur te lezen:

Evans and Gazdar (1996) "DATR: a language for lexical knowledge representation". Computational Linguistics 22.2, 167-216. Keller (1995) "DATR theories ans DATR models". Proceedings ACL95. Kleer (1996) "An evaluation semantics for DATR theories". Proceedings COLING96.

De eerste paar weken hebben we gebruikt om de taal DATR te bestuderen, de daarop volgende weken om door middel van een eigen project in de praktijk te gaan werken met zo een groot databestand. De opdracht voor het project was een programma of serie van programma's te schrijven die bewerkingen uitvoeren op lexicale databestanden om met het verschijnsel lexicale kennisrepresentatie te leren werken.

De voorkennis en vaardigheden van de deelnemers was redelijk divers en we hebben geprobeerd het project onder te verdelen in deelprojecten die ieder een specifieke kennis vereisten.

We hebben voor ons project gebruik maken van de volgende bestaande bestanden en programma's:

- de vijfde release van het Corpus Gesproken Nederlands, in de vorm van .pos en .syn bestanden. Te vinden op: /users.sfinx/heleen/CGN/R5 (op host sfinx.let.uu.nl),
- PORTRAY, een in Utrecht in ****TODO*** 19.. door . . . ontwikkeld programma om . . .* ,
- Grail, ****TODO*** . . .* ,
- ****TODO*** etc.. . .*

input: zinnen voorzien van POStags
output: syntactisch onlede zinnen, weergeven als onleedbomen.

Het was de bedoeling om een DATRtheorie te gebruiken om POStags te koppelen aan categoriale typen. Met deze categoriale typen kan Grail een boom opbouwen. Deze DATRtheorie moest automatisch worden opgebouwd omdat hij moest kunnen omgaan met een oneindig grote databank van bovendien natuurlijke, gesproken, en daarom onvoorspelbare taal. En waarom dit alles, waarom deze in- en output. Over de keuzes die bij het schrijven van de verschillende onderdelen zijn gemaakt later meer. . . .

De opzet van ons project was als volgt:

****TODO***: invoegen schema. . .*

2 DATR

DATR (Evans en Gazdar 1989) is een kennisrepresentatietaal die speciaal ontworpen is voor het vastleggen van lexicale informatie, en wel in de vorm van een niet-monotoon overervingnetwerk (ook wel 'semantisch net' genoemd). Het is een taal die minder mogelijkheden heeft dan voor algemene kennisrepresentatie ontworpen talen en daardoor eenvoudiger, minimalistischer van opzet is.

Hoe onregelmatig lexicale informatie ook mag lijken, vaak bevat een lexicon toch zoveel regelmatige informatie dat een opsomming zou leiden tot een onnodig groot lexicon. DATR maakt het mogelijk redundante informatie uit lexica te persen door gebruik te maken van niet-monotone overerving (uitleggen). Door locale en globale overerving en definitie op basis van default kan de lexicale informatie zo compact mogelijk worden weergegeven. (voorbeeld van een DATRzin en theorie met enkele voorbeeldqueries, waarbij wordt aangegeven waar sprake is van locale of globale overerving, wat een zin is, wat een theorie is, waar sprake is van definitie door default).

DATR is een taal en geen theoretisch raamwerk, zoals HPSG dat bijvoorbeeld wel is. DATR is theorie-inspecifiek en doet geen uitspraken over het uiterlijk of de organisatie van ons mentale lexicon. We kunnen DATR toepassen om van alle natuurlijke talen ter wereld en binnen misschien wel alle taalkundige theorieen lexica te bouwen/ weer te geven. De vorm van een DATRtheorie is afhankelijk van welke eigenschappen de programmeur computationeel of uit het oogpunt van leesbaarheid belangrijk vindt.

3 CGN

3.1 Inleiding

We hebben gebruik gemaakt van data uit het CGN in de vorm van gePoS-tagde weergaven van gesproken taalfragmenten.

Het Corpus Gesproken Nederlands is een project dat tot doel heeft een databank aan te leggen van gesproken Standaardnederlands. Opgenomen gesprekken worden orthografisch, fonetich, syntactisch en/of prosodisch geanalyseerd om uiteindelijk te komen tot een corpus van 10.000.000 woorden. Het gehele corpus, het zogenaamde basiscorpus, is in het geheel orthografisch getranscribeerd, gelemmatiseerd en voorzien van woordsoortinformatie. Een deelverzameling van 1.000.000 woorden, het kerncorpus, is gedetailleerd fonetisch getranscribeerd, syntactisch geannoteerd en voor een kwart prosodisch geannoteerd.

Van het corpus wordt om de 6 maanden een release gedaan. Tot nu toe zijn 5 van de 7 releases een feit. Wij hebben voor ons project gebruik gemaakt van informatie uit het kerncorpus van de vijfde release.

3.2 Waarom het CGN gekozen als databank?

Omdat het project moest gaan over lexicale kennisrepresentatie was het voor de hand liggend om data uit het CGN te gebruiken. Onderdeel van het CGN project is naast het deels manueel en deels automatisch part-of-speech- taggen van taalfragmenten ook het automatisch syntactisch annoteren van deze gePoStagde fragmenten. Wij wilden iets soortgelijks doen als deze laatste vertaalstap en daarbij leek het ons bij het generaliseren over de complexe data handig een lexicale kennisrepresentatietaal te gebruiken. Later bleek dat de taal DATR voor ons eigenlijk een te krachtig middel was: er was minder te generaliseren dan verwacht waardoor DATR eigenlijk overbodig werd.

****TODO***: Hoe zien de .syn files eruit (plaatje/link?)...*

Hoe gaat de PoS-tagging binnen het CGNproject in zijn werk en hoe de automatische syntactische annotatie? ...

4 Tool 1. cgn2grail

****TODO***: Syntaxis is `cgntograil inputfile` en er wordt `cgn.pl` geproduceerd. Moet in de map met zijn Prolog bronnen draaien vanwege toegankelijkheid lexicon.*

Wat wij nodig hebben, is een catogoriaal type voor ieder woord in de zin. Hiervoor gebruiken we CGNtoGrail. De output van dit tool is niet geheel naar onze wens. De deelstrepen staan niet op de plaats waar we ze hebben willen. Bij sommige typen staan ze naar standaard naar links, bij andere standaard naar rechts. We willen deze 'willekeur'niet en kiezen er daarom voor om in het bestand parameter.pl, dat onderdeel uit maakt van CGNtoGRail, de waarde voor Position in de claus is_functor1(+EdgeLabel,+ParentCategory,+Position) standaard op inplace (en niet op initial of final) te zetten, opdat de deelstrepen op hun canonieke plaats staan.

5 Tool 2. Lexentry

****TODO***: Syntaxis is* `lexentry outputfile` *en* `cgn.pl` *wordt gelezen. Perl met bash-shellscript wat gebruik maakt van sed, awk en grep.*

Het prologfile dat we krijgen na het runnen van ToolI bevat meer informatie dan we nodig hebben. Echter, de postags zijn bij cgn2grail verloren gegaan. Deze zullen we dus weer te voorschijn moeten halen. Met behulp van een shell-script worden de overbodige gegevens uit het output file van ToolI verwijderd. Daarna zorgt een Perl script er voor dat de Postags uit het originele CGN synfile gehaald worden. Dit wordt gedaan aan de hand van synfile nummers, de zinnummers en de regelnummers welke verwijzen naar de positie waar het postag zich bevindt. De output van dit script bevat voor iedere postag alle (binnen het CGN) mogelijke categoriale types.

6 Tool 3. pos2dtr

****TODO***: Syntaxis is* `pos2dtr inputfile outputfile` *waarbij outputfile bijv. pos2datr.dtr voor gebruik met tool 5 is. Geschreven in Java.*

6.1 Inleiding

Tool3 is de naam van het derde gereedschap van het project dat hoort bij het vak Lexicale Kennisrepresentatie 2002.

De bedoeling van dit onderdeel is dat het de uitvoer van tool2 omzet in een DATR theorie die bruikbaar is voor tool4 van ditzelfde project.

Tool4 heeft de beschikking over een PoS-tag en wil alle lexicale typen die bij deze PoS-tag mogelijk zijn vinden. Het is daarom logisch elke PoS-tag een knoop in de DATR theorie te laten zijn.

De paden van zo een knoop voeren tot de lexicale typen die door tool4 op te vragen zijn. De informatievrager moet wel weten welke paden opvraagbaar zijn. Daarom bestaan de namen van de paden uit een vaste string “type” met daaraan geplakt een nummer. Dit nummer loopt per knoop op van 1 t/m het aantal lexicale typen dat bij die PoS tag gevonden is.

Een rudimentaire vorm van inheritance wordt aan de resulterende DATR uitvoer toegevoegd. De hoop is dat sommige woordgroepen sterk overeenkomen in hun lexicale typen. Hierover is meer te lezen in latere hoofdstukken.

Tool3 is geïmplementeerd in de programmeertaal Java. Dit is een objectgeoriënteerde taal. Dit wil zeggen dat gebruik wordt gemaakt van objecten die informatie bevatten en zij bevatten methoden om met deze informatie te kunnen werken.

6.2 Invoer van tool2

De invoer vanuit tool2 bestaat uit een tab-gelimiteerd tekstbestand met twee kolommen. In de eerste kolom staat telkens een PoS-tag. In de tweede kolom staat een formule die het lexicale type aangeeft.

6.2.1 PoS-tag

De PoS-tag is rechtstreeks over te nemen van de uitvoer van tool2. Een voorbeeld van een PoS-tag is het volgende:

"T301"

Deze begint altijd met een hoofdletter. Aan de voorwaarde dat knoopnamen in DATR altijd met een hoofdletter beginnen is bij deze dus voldaan.

Alle gevonden PoS-tags worden opgeslagen in een datastructuur. Zo komen we tot een object dat Tool_iiiData heet en een aantal POSTag objecten bevat.

Indien in een andere regel dezelfde PoS-tag gevonden wordt, dan kan er beter geen nieuw POSTag object aangemaakt worden: uiteindelijk moet elke POSTag precies één knoop vormen. De oplossing is het samensmelten van het bestaande POSTag object met de nieuw gevonden PoS-tag. Hiervoor heeft het object POSTag de methode 'join'.

Als het aangeboden lexicale type bij deze POSTag nog niet voorkomt, dan wordt het lexicale type als LexType object toegevoegd aan het POSTag object. Een POSTag object bevat dus een aantal LexType objecten.

6.2.2 Lexicaal type

In de invoer voor deze tool zien de formules die lexicale typen weergeven er als volgt uit:

```
lit(ww(7))
```

```
dia(hd,box(hd,dr(l(ld),dl(r(su),lit(n),dr(l(cnj),lit(conj),
lit(conj))),lit(pp))))
```

Bij het eerste voorbeeld wordt het lexicaal type hetgeen dat tussen haakjes staat. In dit geval “ww(7)”.

Voor het tweede voorbeeld moet de formule in stukken gehakt worden op de plaatsen waar de komma’s staan. Elk stukje is apart te vertalen. Als een stukje begint met “dia” of met “box”, dan wordt de vertaling een gelijknamig atoom. Delen als “dl(r(su)” en “dr(l(cnj)” vertalen tot een boog plus een label: respectievelijk “ su ” en “/ cnj ”.

Let op de spaties waarop de vertalingen eindigen. Deze worden gebruikt als atoomscheidingen in de uiteindelijke DATR theorie. De vertaalslag voor “lit” werkt nog steeds, mits er een spatie achteraan geplakt wordt. Het resultaat voor het tweede voorbeeld is dus:

```
dia box / ld  su n / cnj conj conj pp
```

Elke formule in de uitvoer van tool2 blijkt met “dia(hd,box(hd” te beginnen zonder dat dit daadwerkelijk informatie toevoegd. Indien deze diamants en boxes ongewenst zijn kunnen deze genegeerd worden. Hiervoor is een optie ingebouwd. Bovendien worden de labels van de bogen niet gebruikt in andere tools van dit project. Daarom kunnen ook deze optioneel genegeerd worden.

Vervolgens wordt een lexicaal type opgeslagen in een LexType object. Dit bevat een string zoals deze uit de vertaling van de formule tevoorschijn komt.

6.3 Inheritance

6.3.1 Wanneer erven?

Het idee van inheritance is dat een DATR theorie compacter wordt. bij de bouw van tool3 is nog onbekend welke vorm van inheritance hiertoe het beste zou dienen. Eerst uitproberen en dan pas bouwen gaat nu eenmaal niet met de hoeveelheid data die verwerkt moet worden.

Zoals in de inleiding gezegd is is er hoop dat sommige woordgroepen sterk overeenkomen in hun lexicale typen. Zo kunnen de persoonsvormen, PoS-tag T301 t/m T309, mogelijk veel van elkaar erven. Een eenvoudige vorm van automatisch genereerbare inheritance is de volgende. Knoop A erft alle paden van knoop B als al deze paden ook in knoop A zitten.

Zo kan in knoop A het lege pad naar knoop B wijzen.

6.3.2 Uitvoering

Voor de uitleg wordt hier het volgende, vereenvoudigde bestand als uitvoer van tool2 gegeven.

Vereenvoudigde tool2 uitvoer:

```
TA  lit(1)
TA  lit(2)
TA  lit(4)
TB  lit(1)
TB  lit(2)
TB  lit(3)
TB  lit(4)
TC  lit(1)
TC  lit(4)
TD  lit(3)
TD  lit(4)
TD  lit(6)
TD  lit(9)
TE  lit(4)
```

Inheritance wordt toegevoegd in drie stappen.

Ten eerste worden de POSTag objecten gesorteerd, zodat de POSTag objecten die meer LexType objecten bevatten eerder komen dan POSTag objecten die minder van deze objecten bevatten. Het voorbeeld hier zou er in DATR notatie zo uit zien:

Data gesorteerd op aantal LexType objecten per POSTag object:

```
TB:
   <type1>  ==  1
   <type2>  ==  2
   <type3>  ==  3
   <type4>  ==  4.
TD:
   <type1>  ==  3
   <type2>  ==  4
   <type3>  ==  6
   <type4>  ==  9.
TA:
   <type1>  ==  1
   <type2>  ==  2
   <type3>  ==  4.
TC:
   <type1>  ==  1
   <type2>  ==  4.
TE:
   <type1>  ==  4.
```

Vervolgens wordt gekeken of de bovenste knoop alle type van een knoop eronder kan erven. Hierbij worden de knopen van boven naar onder doorlopen. TB zal dus van TA erven. Hierna gaat het algoritme door met de tweede knoop enz. met onderstaand resultaat.

Data met inheritance:

```
TB:
   <> == TA
   <type4> == 3.
TD:
   <type1> == 3
   <type2> == 4
   <type3> == 6
   <type4> == 9.
TA:
   <> == TC
   <type3> == 2.
TC:
   <> == TE
   <type2> == 1.
TE:
   <type1> == 4.
```

De gevonden inheritance wordt opgeslagen in het `Tool_iiiData` object.

6.4 Uitvoer naar een DATR bestand

Gepoogd is de zoektocht naar inheritance te scheiden van de daadwerkelijk uitvoer naar een DATR bestand. Dit zal nooit helemaal lukken, doordat slechts één vorm van inheritance in het `Tool_iiiData` object mogelijk is. Als er andere vormen van inheritance gezocht worden, dan moeten zowel het datamodel als de manier om dit naar een DATR bestand om te zetten gewijzigd worden. Een `Tool_iiiData` object is per slot van rekening geen DATR object.

Softwareontwikkelaars streven vaak naar herbruikbaarheid van hun programma's (en de onderdelen ervan). Toch is er geen compleet DATR object in de taal Java ontworpen, omdat dit voor dit éénmalige project onnodig was.

6.5 Discussie

6.5.1 De mate aan inheritance

Bij een test met relatief weinig invoer bleek de gebruikte inheritance weinig extra compactheid op te leveren. Geen knoop erft meer dan drie paden van een ander, terwijl er knopen zijn met meer dan 20 paden.

Misschien is de hoeveelheid data te klein. Hierdoor zouden lexicale typen die in principe wel bij een PoS-tag voorkomen buiten de data vallen. Laten we voor de gedachtegang even aannemen dat de mogelijke lexicale typen van één PoS-tag een superset vormen van die van een andere. Ontbreekt in de data bij de eerste PoS-tag één lexicaal type uit de gezamelijke set dan is de gezochte vorm van inheritance al onmogelijk.

Kijkend naar de testuitvoer is van dit verschijnsel niets terug te vinden. Misschien komt dit doordat de hoeveelheid data veel te klein is en de gevonden lexicale typen een kleine, willekeurige selectie zijn.

Mogelijk is de structuur van de data verkeerd ingeschat. Een andere vorm van inheritance zou dan misschien beter werken. Bijvoorbeeld kunnen verschillende PoS-tags wel veel overeenkomende lexicale type hebben. Het zou dan logisch zijn een aparte knoop te maken waar de gevonden gezamelijke lexicale typen worden opgeslagen. Een knoop, genaamd 'T30x' zou bijvoorbeeld de gevonden, gezamelijke lexicale typen van PoS-tags T301 t/m T309 kunnen bevatten.

Als uitbreiding zouden knopen zoals 'T30x' en 'T31x' weer van 'T3xx' kunnen erven. Voor deze vorm van inheritance kunnen het datamodel en de methoden waarmee naar DATR geschreven wordt onveranderd blijven. Echter, ook hier geldt dat het onmogelijk is eerst te proberen en daarna te bouwen.

Het invoerbestand van de test is ongeveer drie maal zo groot als het resulterende DATR bestand. Dit komt voornamelijk doordat redundante informatie verwijderd is tijdens de vertaling van de formules. Alle syntactisch geannoteerde zinnen van release 5 vormen samen een bestand van circa 8 MB. Als dit bestand (eventueel in delen) als invoer voor tool1 en tool2 dient, dan voert tool2 een bestand uit dat kleiner is dan 8 MB. Zodra tool3 dit verwerkt heeft blijft in ieder geval minder dan 3 MB over. Dit is voor hedendaagse begrippen erg weinig. Eigenlijk heeft inheritance als compressiemethode in dit geval dus weinig meerwaarde.

6.5.2 Grote bestanden

Tijdens het zoeken naar inheritance moet alle informatie in het actieve geheugen aanwezig zijn. Met grotere hoeveelheden informatie kunnen hierdoor problemen ontstaan. Onverkend is de grens van het geheugengebruik. In de discussie over inheritance is geopperd dat meer data wenselijk zijn. Een uitgebreidere test zou moeten uitwijzen of met de volledige verzameling beschikbare data gewerkt kan worden.

6.6 Meer informatie over de implementatie

Details van de implementatie van tool3 zijn te vinden in de javadoc- en bronbestanden.

7 Tool 5. HDATR

****TODO***: Tool 4 is er niet, ook al was er een tool 4 in het oorspronkelijke opzet. De functionatileit is nu onderdeel van andere tools.*

****TODO***: Syntaxis is* `hdatr -p pos2datr.dtr finfile sentence >outputfile` *waarbij pos2datr.dtr output van tool3 (pos2dtr) is, finfile een FinFile en sentence een zin. Haskell bronnen zijn nog niet in de "Final" verzameling inbegrepen.*

HDATR is een implementatie van DATR in de programmeertaal Haskell. Zowel lokale als globale contexten worden bijgehouden. In dit gedeelte zal ik een gedeelte van de implementatie bespreken. De implementatie is volledig ten opzichte van de evaluatiesemantiek.

Het programma bestaat uit een aantal Haskell bestanden.

- `AbstractDATR.hs` : Dit dit bestand worden de datastructuren gedefinieerd. Denk hierbij aan een datastructuur voor een DATR bestand, interne definities van "context" etc.
- `ParseDATR.hs` : In dit bestand staan een aantal functies om DATR bestanden en DATR queries te ontleden. Bovendien wordt de QDATR extensie om andere DATR bestanden te laden ('# load') ondersteunt.
- `Evaluate.hs` : Dit bestand is de kern van HDATR. De logica van DATR is hierin gecodeerd.
- `DATR.hs` : Dit is het hoofdbestand. Dit roept de ontleed- en evaluatiefuncties aan.

Voor dit verslag is het aardig om iets van het detail van Evaluate.hs te bekijken. Zoals gebruikelijk bij functionele talen bekijk je het programma van achter naar voren. Hier zien we de functie 'selectPath'. Deze functie past het idee van 'default paths' toe. Het kiest, gegeven een concreet pad, het meest specifieke pad uit de mogelijke paden in een knoop. De functie 'select' is een functie die gebruik hiervan maakt.

Het kiest nu niet uit een aantal paden het beste pad, maar kiest het beste pad uit de gehele DATR-definitie. De functie waar het in dit bestand om gaat echter, is de functie 'evaluate'. Deze functie voert de daadwerkelijke evaluatie uit. Het doet dit, kort gezegd, door recursief de subelementen van het pad te evalueren, hiervan een pad te vormen en dit te zelf te evaluaren. Bovendien houd het rekening met een globale en lokale context die hierbij wordt gebruikt. Deze bepaalt de richting van de evaluatie. Voor meer detail zie de code.

De hoofdfunctie van dit bestand is 'query' en is slechts een wrapper voor de functie 'evaluate'. Het voegt een geaccumuleerde variable voor de context toe voor de aanroep van evaluate.

7.1 Gebruik

Het programma kan op twee manieren worden gebruikt, namelijk:

- Standalone DATR
- Als onderdeel van dit project :)

De eerste manier gaat als volgt:
`hdatr -d <datr-file> <query>`
dus bijvoorbeeld: `hdatr -d mijndatr.dtr "Test:<test>"`

Deze mogelijkheid zit in het programma voor het debuggen. Het programma werkt bijvoorbeeld goed met een complex DATR bestand die een logische formule op vervulbaarheid controleert. Dit garandeert niet dat er geen fouten in zitten, maar het garandeert wel dat het programma zeer uitgebreid getest is.

De tweede manier is een kleine uitbreiding die eigenlijk niet in HDATR thuis hoort, maar er in zit omdat het sneller is dan een script die HDATR constant aanroept.

Het wordt als volgt gebruikt:
`hdatr -p <datr-file> <fin-file> <#sentence>`
dus bijvoorbeeld: `hdatr -p mijndatr.dtr fn0007.fn 1`

Deze methode is compatible met de rest van de scripts en programma's uit ons project.

7.2 Snelheid

Benchmarks van grote DATR bestanden laten zien dat HDATR in elk geval veel sneller is dan een QDATR. Dit is geen eerlijke vergelijking omdat QDATR geinterpreteerd wordt en HDATR gecompileerd wordt. Wat opvalt is dat bij een grotere query QDATR in verhouding nog langzamer gaat in vergelijking met HDATR. Dit kan geen harde uitspraken opleveren, maar het lijkt er op dat de gemiddelde complexiteit van HDATR kleiner is dan die van QDATR. Een verklaring hiervoor kan zijn dat er Haskell wordt gebruikt. Deze programmeertaal maakt gebruik van zogenaamde lazy evaluation. Dit wil zeggen dat een element pas wordt uitgerekent als het ook werkelijk nodig is om te weten. Dit kan grote snelheidswinst opleveren bij een DATR implementatie, omdat de padlengte van het pad dat wordt geëvalueerd zeer groot kan wordt (bij een niet-triviaal DATR bestand). Lazy evaluation zorgt ervoor dat alleen het begin van het pad hoeft te worden berekent.

8 Tool 6. Mug

****TODO***: Syntaxis is `mug inputfile` of, om ook .tex van de bewijzen in outputfile opteslaan, `mug inputfile outputfile.tex` – zonder .tex output print `mug` slechts de hoeveelheid aan gevonden parses op het scherm. Geschreven in Haskell, te compileren met ghc.*

Mug – Lambek calculus prover (without semantics)
(uit: Manual - Version 0.1.5 van: Stephan A.B. van der Feest)

8.1 Installing Mug

The binary version of Mug does not need any installing; all you need is one file:

`mug` for UNIXes or `mug.exe` for Windows

To compile Mug from source, you need to have the Glasgow Haskell Compiler installed (see `http://www.haskell.org/ghc`).

You can then compile Mug by running:

```
ghc -make -o mug mug.hs
```

from the directory that contains `mug.hs` and `ParseLib.hs`

8.2 Using Mug

There are three ways to call mug:

```
mug <inputfile>
```

Tries to prove the sentence in `inputfile` , writes the number of provable lexical combinations to stdout.

```
mug <inputfile> <texfile>
```

Does the same, and writes the proves found as LaTeX to `texfile`.

```
mug <inputfile> <viewer> <texfile>
```

In addition, view the .dvi output from LaTeX in `viewer`.

8.2.1 Input file syntax

The input file should start with one line containing the sentence to be proven, in plain text (meaning words seperated by spaces), and preferably using only lower-case characters (because lexicon entries must match exactly).

The second line should contain the goal formula (probably often `s`)

After that, every following line should contain a lexicon entry, in the following format:

```
form: type
```

Where `type` can contain any atomical formula, or any combination of them using /, \ and `*` with both arguments seperated by spaces, such that:

`/ a b` = a/b

`\ a b` = $b\backslash a$

`* a b` = $a \bullet b$

Thus, a possible simple file will look as follows:

```
alice reads the book
s
alice: np
reads: \ s np
reads: / s \ s np
the: / np n
book: n
```

8.2.2 Output options

At the moment, Mug only supports output to a .tex file for LaTeX, which the program also calls after proving a sentence correct, so be sure that `latex` is in your `PATH`, because otherwise you will be stuck with just the .tex file.

The output file for the example from section 8.2.1 would contain the reading:

`alice:` $np \circ$ `reads:` $((np\backslash s)/np) \circ$ `the:` $(np/n) \circ$ `book:`

n

followed by 2 possible Gentzen-style derivations, the first looking like this:

$$\dfrac{\dfrac{\overline{n \Rightarrow n} \quad \overline{np \Rightarrow np}}{(np/n), n \Rightarrow np}[/L] \quad \dfrac{\overline{np \Rightarrow np} \quad \overline{s \Rightarrow s}}{np, (np\backslash s) \Rightarrow s}[\backslash L]}{np, ((np\backslash s)/np), (np/n), n \Rightarrow s}[/L]$$

If you don't need Mug to create the output file, you can omit the output file parameter when running it; if you only provide an input file name, the only thing Mug will do is print one line to standard output: the number of possible lexical substitutions that make the sentence provable.

This can be a useful feature if you want to use Mug in some bigger system, only to check if some sentence is provable (In fact, Mug was designed for such a system).

9 Resultaten

****TODO***: Voorbeelden van in- en output Score grammaticaliteit?*

9.1 Conclusie

****TODO****

9.2 Foutendiscussie

****TODO***: Hebben we, zoals beoogd, optimaal gebruik kunnen maken van de mogelijkheden van DATR?*

****TODO***: Struikelblokken.*

****TODO***: Wat hadden we beter kunnen doen?*

9.3 Samenvatting

****TODO****

10 Appendix

****TODO***: broncode inplakken of URL aangeven*

10.1 Tools

- Tool 1. `cgn2grail infile` (output is cgn.pl)
600 regels Prolog plus automatisch gegenereerde 6000 regels meer (Prolog, bij voorkeur Sicstus, nodig om het te draaien).
- Tool 2. `lexentry outfile` (input is cgn.pl)
Kort bash-shellscript (gebruikt `awk` , `sed` , `grep`) en 85 regels Perl voor `loc2pos` (Perl nodig en awk, sed, grep en natuurlijk bash).
- Tool 3. `pos2dtr infile outfile`
860 regels Java (Java VM nodig om het te draaien).
- Tool 4. is er niet!
- Tool 5. `pos2tag` : hiervoor draai je `hdatr -p pos2datr.dtr finfile sentence > outfile`
Haskell broncode is nog geen onderdeel van de “Final” verzameling. Ook als binary zonder Haskell of bronnen te gebruiken.
- Tool 6. `mug infile` of `mug infile texoutfile` 430 regels Haskell plus 520 regels meer voor ParseLib. Ook als binary zonder Haskell of bronnen te gebruiken.

10.2 .syn-files

****TODO***: .syn files inplakken of. . .*