A Modern Computational Linguistics Course Using Dutch

Gosse Bouma
Groningen University
PO Box 716
NL 9700 AS Groningen
gosse@let.rug.nl

Abstract

This paper describes material for a course in computational linguistics which concentrates on building (parts of) realistic language technology applications for Dutch. We present an overview of the reasons for developing new material, rather than using existing text-books. Next we present an overview of the course in the form of six exercises, covering advanced use of finite state methods, grammar development, and natural language interfaces. The exercises emphasise the benefits of special-purpose development tools, the importance of testing on realistic data-sets, and the possibilities for web-applications based on natural language processing.

1 Introduction

This paper describes a set of exercises in computational linguistics. The material was primarily developed for two courses: an general introduction to computational linguistics, and a more advanced course focusing on natural language interfaces. Students who enter the first course have a background in either humanities computing or cognitive science. This implies that they possess some general programming skills and that they have at least some knowledge of general linguistics. Furthermore, all students entering the course are familiar with logic programming and Prolog. The native language of practically all students is Dutch.

The aim of the introductory course is to provide a overview of language technology applications, of the concepts and techniques used to develop such applications, and to let students gain practical experience in developing (components) of these applications. The second course focuses on computational semantics and the construction of natural language interfaces using computational grammars.

Course material for computational linguistics exists primarily in the form of text books, such as Allen (1987), Gazdar and Mellish (1989) and Covington (1994). They focus primarily on basic concepts and techniques (finite state automata, definite clause grammar, parsing algorithms, construction of semantic representations, etc.) and the implementation of toy systems for experimenting with these techniques. If course-ware is provided, it consists of the code and grammar fragments discussed in the text-material. The language used for illustration is primarily English.

While attention for basic concepts and techniques is indispensable for any course in this field, one may wonder whether implementation issues need to be so prominent as they are in the text-books of, say, Gazdar and Mellish (1989) and Covington (1994). Developing natural language applications from scratch may lead to maximal control and understanding, but is also time-consuming, requires good programming skills rather than insight in natural language phenomena, and, in tutorial settings, is restricted to toy-systems. These are disadvantages for an introductory course in particular. In such a course, an attractive alternative is to skip most of the implementation issues, and focus instead on what can be achieved if one has the right tools and data available. The advantage is that the emphasis will shift naturally to a situation where students must concentrate primarily on developing accounts for linguistic data, on exploring data available in the form of corpora or word-lists, and on using real high-level tools. Consequently, it becomes feasible to consider not only toy-systems and toy-fragments, but to develop more or less realistic components of natural language applications. As the target language of the course is Dutch, this also implies that at least some attention has to be paid to specific properties of Dutch grammar, and to (electronic) linguistic resources for Dutch. Since students nowadays have access to powerful hardware and both tools and data can be distributed easily over the internet, there are no real practical obstacles.

Text-books which are concerned primarily with computational semantics and natural language interfaces, such as Pereira and Shieber (1987) and Blackburn and Bos (1998), tend to introduce a toy-domain, such as a geography database or an excerpt of a movie-script, as application area. In trying to develop exercises which are closer to real applications, we have explored the possibilities of using web-accessible databases as back-end for a natural language interface program.

More in particular, we hope to achieve the following:

- *Students learn to use high-level tools.* The development of a component for morphological analysis requires far more than what can be achieved by specifying and implementing the underlying finite state automata directly. Rather, abstract descriptions of morphological rules should be possible, and software should be provided to support development and debugging. Similarly, while a programming language such as Prolog offers possibilities for relatively high-level descriptions of natural language grammars, the advantages of specialised languages for implementing unification-based grammars and accompanying tools are obvious. Furthermore, the availability of graphical interfaces and visualisation in tutorial situations is a bonus which should not be underestimated.

- *Students learn to work with real data.* In developing practical, robust, wide-coverage, language technology applications, researchers have found that the use of corpora and electronic dictionaries is absolutely indispensable. Students should gain at least some familiarity with such sources, learn how to search large datasets, and how to deal with exceptions, errors, or unclear cases in real data.

- *Students become familiar with quantitative evaluation methods.* One advantage of developing components using real data is that one can use the evaluation metrics dominant in most current computational linguistics research. That is, an implementation of hyphenation-rule or a grammar for temporal expressions can be tested by measuring its accuracy on a list of unseen words or utterances. This provides insight in the difficulty of solving similar problems in a robust fashion for unrestricted text.

- *Students develop language technology components for Dutch.* In teaching computational linguistics to students whose native language is not English, it is common practice to focus primarily on the question how the (English) examples in the text book can be carried over to a grammar for one's own language. As this may take considerable time and effort, more advanced topics are usually skipped. In a course which aims primarily at Dutch, and which also contains material describing some of the peculiarities of this language (hyphenation rules, spelling rules relevant to morphology, word order in main and subordinate clauses, verb clusters), there is room for developing more elaborate and extended components.

- *Students develop realistic applications.* The use of tools and real data makes it easier to develop components which are robust and which have relatively good coverage. Applications in the area of computational semantics can be made more interesting by exploiting the possibilities offered by the internet. The growing amount of information available on the internet provides opportunities for accessing much larger databases (such as public transport time-tables or library catalogues), and therefore, for developing more realistic applications.

The sections below are primarily concerned with a number of exercises we have developed to achieve the goals mentioned above. A accompanying text is under development.[1]

2 Finite State Methods

A typical course in computational linguistics starts with finite state methods. Finite state techniques can provide computationally efficient solutions to a wide range of tasks in natural language processing. Therefore, students should be familiar with the basic concepts of automata (states and transitions, recognizers and transducers, properties of automata) and should know how to solve

[1] See `www.let.rug.nl/~gosse/tt` for a preliminary version of the text and links to the exercises described here.

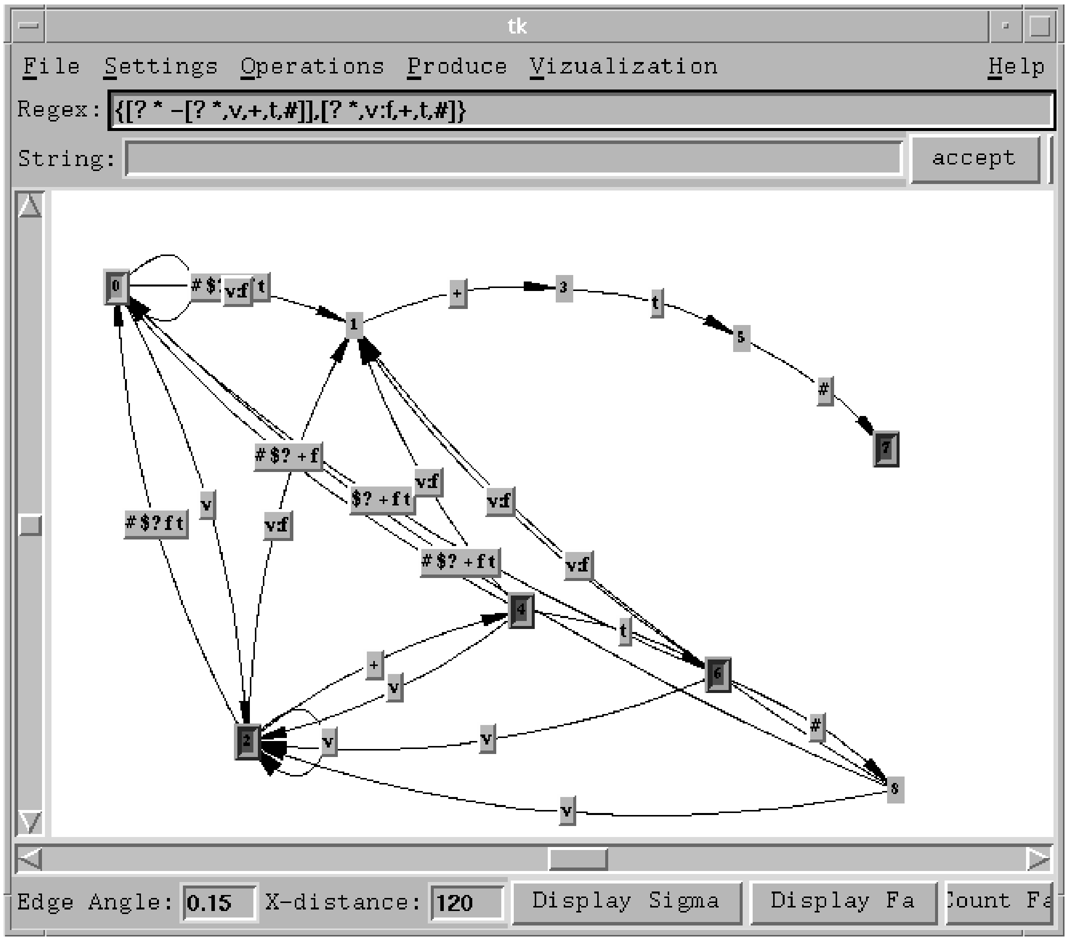

Figure 1: FSA. The regular expression and transducer are an approximation of the rule for realizing a final `-v` in abstract stems as `-f` if followed by the suffix `-t` (i.e. `leev+t` → `leeft`). `[A,B]` denotes `A` followed by `B`, `{A,B}` denotes `A` or `B`, '?' denotes any single character, and `A - B` denotes the string defined by `A` minus those defined by `B`. `A:B` is the transduction of `A` into `B`. '+' is a morpheme boundary, and the hash-sign is the end of word symbol.

toy natural language processing problems using automata.

However, when solving 'real' problems most researchers use software supporting high-level descriptions of automata, automatic compilation and optimisation, and debugging facilities. packages for two-level morphology, such as PC-KIMMO (Antworth, 1990), are well-known examples. As demonstrated in Karttunen et al. (1997), an even more flexible use of finite state technology can be obtained by using a calculus of regular expressions. A high-level description language suited for language engineering purposes can be obtained by providing, next to the standard regular expression operators, a range of operators intended to facilitate the translation of linguistic analyses into regular expressions. Complex problems can be solved by composing automata defined by simple regular expressions.

We have developed a number of exercises in which regular expression calculus is used to solve more or less 'realistic' problems in language technology. Students use the FSA-utilities package [2] (van Noord, 1997), which provides a powerful language for regular expressions and possibilities for adding user-defined operators and macros, compilation into (optimised) automata, and a graphical user-interface. Automata can be displayed graphically, which makes it easy to learn the meaning of various regular expression operators (see figure 1).

Exercise I: Dutch Syllable Structure

Hyphenation for Dutch (Vosse, 1994) requires that complex words are split into morphemes, and mor-

[2] `www.let.rug.nl/~vannoord/fsa/`

```
macro(syll, [ onset^, nucleus, coda^ ] ).
macro(onset, {[b,{l,r}^],[c,h^,{l,r}^]}).
macro(nucleus, { [a,{[a,{i,u}^],u}^],
                 [e,{[e,u^],i,o,u}^]  }).
macro(coda, {[b, {s,t}^], [d,s^,t^]}).
```

Figure 2: First approximation of a regular expression defining Dutch syllables. `A^` means that `A` is optional.

phemes are split into syllables. Each morpheme or syllable boundary is a potential insertion spot for a hyphen. Whereas one would normally define the notion 'syllable' in terms of phonemes, it should be defined in terms of character strings for this particular task. The syllable can easily be defined in terms of a regular expression. For instance, using the regular expression syntax of FSA, a first approximation is given in figure 2. The definition in 2 allows such syllables as `[b,a,d]`, `[b,l,a,d]`, `[b,r,e,e,d,s,t]`, etc.

Students can provide a definition of the Dutch syllable covering all perceived cases in about twenty lines of code. The quality of the solutions could be tested in two ways. First, students could test which words of a list of over 5000 words of the form `[C*,V+,C*]` (where `C` and `V` are macros for consonants and vowels, respectively) are accepted and rejected by the syllable regular expression. A restrictive definition will reject words which are bisyllabic (`geijkt`) and foreign words such as `crash`, `sfinx`, and `jazz`. Second, students could test how accurate the definition is in predicting possible hyphenation positions in a list of morphophonemic words. To this end, a list of 12000 morphophonemic words and their hyphenation properties was extracted from the CELEX lexical database (Baayen et al., 1993).[3] The best solutions for this task resulted in a 5% error rate (i.e. percentage of words in which a wrongly placed hyphenation point occurs).

Exercise II: Verbal Inflection

A second exercise concentrated on finite state transducers. Regular expressions had to be constructed for computing the surface form of abstract verbal stem forms and combinations of a stem and a verbal inflection suffix (see figure 3). Several spelling rules need to be captured. Examples (b) and (c) show that single consonants following a short vowel are doubled when followed by the `'+en'` suffix, while long vowels (normally represented by two identical characters) are written as a single character when followed by a single consonant and `'+en'`. Examples (d-g) illustrate that the rule which requires doubling of a consonant after a short vowel is not applied if the preceding vowel is a schwa. Note that a single `'e'` (sometimes `'i'`) can be either a stressed vowel or a schwa. This makes the rule hard to apply on the basis of the written form of a word. Examples (h-j) illustrate the effect of devoicing on spelling. Examples (i-l) illustrate several other irregularities in present tense and infinitive forms that need to be captured. Examples (n-q), finally, illustrate past tense formation of weak verbal stems. Past tenses are formed with either a `'+te'` or `'+de'` ending (`'+ten'`/`'+den'` for plural past tenses). The form of the past tense is predictable on the basis of the preceding stem, and this a single underlying suffix `'+Te'` is used. Stems ending with one of the consonants `'c,f,h,k,p,s,t'` and `'x'` form a past tense with `'+te'`, while all other stems receive a `'+de'` ending. Note that the spelling rule for devoicing applies to past tenses as well (p-q). In the exercise, only past tenses of weak stems were considered.

	Underlying	**Surface**	**Gloss**
a.	werk+en	werken	work[INF]
b.	bak+en	bakken	bake[INF]
c.	raak+en	raken	hit[INF]
d.	verwen+en	verwennen	pamper[INF]
e.	teken+en	tekenen	draw[INF]
f.	aanpik+en	aanpikken	catch up[INF]
g.	zanik+en	zaniken	wine[INF]
h.	leev+en	leven	live[INF]
i.	leev	leef	live[PRES, 1, SG.]
j.	leev+t	leeft	live(s)[PRES, 2/3, SG.]
k.	doe+en	doen	do[INF]
l.	ga+t	gaat	go(es)[PRES, 2/3, SG.]
m.	zit+t	zit	sit(s)[PRES, 2/3, SG.]
n.	werk+Te	werkte	worked[PAST, SG]
o.	hoor+Te	hoorde	heard[PAST, SG]
p.	blaf+Te	blafte	barked[PAST, SG]
q.	leev+Te	leefde	lived[PAST, SG]

Figure 3: Dutch verbal inflection

The implementation of spelling rules as transducers is based on the `replace`-operator (Kart-

[3]The hyphenation task itself was defined as a finite state transducer:

```
macro(hyph, replace([] :  - , syll, syll))
```

The operator `replace (Target, LeftContext, RightContext)` implements 'leftmost' (and 'longest match') replacement (Karttunen, 1995). This ensures that in the cases where a consonant could be either final in a coda or initial in the next onset, it is in fact added to the onset.

```
macro(verbal_inflection,
  shorten o double o past_tense).
macro(shorten,
  replace([a,a]:a ,[],[cons,+,e,n])).
macro(double,
  replace(b:[b,b],
              [cons,vowel],[+,e,n])).
macro(past_tense,
  te_suffix o past_default).
macro(te_suffix,
  replace([T]:[t],
        [{c,f,h,k,s,p,t,x},+],[])).
macro(past_default,
  replace([T]:[d],[],[])).
```

Figure 4: Spelling rules for Dutch verbal inflection. `A o B` is the composition of transducers `A` and `B`.

tunen, 1995). A phonological or spelling rule

$$\texttt{U} \rightarrow \texttt{S / L \_ R}$$

can be implemented in FSA as:

```
replace(Underlying:Surface, Left, Right)
```

An example illustrating the rule format for transducers is given in figure 4. Most solutions to the exercise consisted of a collection of approximately 30 `replace`-rules which were composed to form a single finite state transducer. The size of this transducer varied between 4.000 and over 16.000 states, indicating that the complexity of the task is well beyond reach of text-book approaches.

For testing and evaluation, a list of almost 50.000 pairs of underlying and surface forms was extracted from Celex.[4] 10 % of the data was given to the students as training material. Almost all solutions achieved a high level of accuracy, even for the '`verwennen/tekenen`' cases, which can only be dealt with using heuristics. The best solutions had less than 0,5% error-rate when tested on the unseen data.

[4]Reliable extraction of this information from Celex turned out to be non-trivial. Inflected forms are given in the database, and linked to their (abstract) stem by means of an index. However, the distinction between weak and strong past tenses is not marked explicitly in the database and thus we had to use the heuristic that weak past tense singular forms always end in '`te`' or '`de`', while strong past tense forms do not. Another problem is the fact that different spellings of a word are linked to the same index. Thus, '`scalperen`' (to scalp) is linked to the stem '`skalpeer`'. For the purposes of this exercise, such variation was largely eliminated by several ad-hoc filters.

3 Grammar Development

Natural language applications which perform syntactic analysis can be based on crude methods, such as key-word spotting and pattern matching, more advanced but computationally efficient methods, such as finite-state syntactic analysis, or linguistically motivated methods, such as unification-based grammars. At the low-end of the scale are systems which perform partial syntactic analysis of unrestricted text (*chunking*), for instance for recognition of names or temporal expressions, or NP-constituents in general. At the high-end of the scale are wide-coverage (unification-based) grammars which perform full syntactic analysis, sometimes for unrestricted text. In the exercises below, students develop a simple grammar on the basis of real data and students learn to work with tools for developing sophisticated, linguistically motivated, grammars.

3.1 Exercise III: Recognizing temporal expressions

A relatively straightforward exercise in grammar development is to encode the grammar of Dutch temporal expressions in the form of a context-free grammar.

In this particular case, the grammar is actually implemented as a Prolog definite clause grammar. While the top-down, backtracking, search strategy of Prolog has certain drawbacks (most notably the fact that it will fail to terminate on left-recursive rules), using DCG has the advantage that its relationship to context-free grammar is relatively transparent, it is easy to use, and it provides some of the concepts also used in more advanced unification-based frameworks. The fact that the non-terminal symbols of the grammar are Prolog terms also provides a natural means for adding annotation in the form of parse-trees or semantics.

The task of this exercise was to develop a grammar for Dutch temporal expressions which covers all instances of such expressions found in spoken language. The more trivial part of the lexicon was given and a precise format was defined for semantics. The format of the grammar to be developed is illustrated in figure 5. The top rule rewrites a `temp_expr` as a weekday, followed by a date, followed by an hour. An hour rewrites as the ad-hoc category `approximately` (containing several words which are not crucial for semantics but which frequently occur in spontaneous utterances), and an `hour1` category, which in turn can rewrite as a category `hour_lex` followed by the word `uur`, followed

```
temp_expr(date(Da,Mo,Ye),day(We),
                    hour(Ho,Mi)) -->
  weekday(We), date(Da,Mo,Ye),
                        hour(Ho,Mi).

weekday(1) --> [zondag].

date(Date,Month) -->
   date_lex(Date), month_lex(Month).

hour(Hour,Min) -->
   approximately, hour1(Hour,Min).

approximately -->
      [ongeveer] ; [om] ; [rond] ;
      [omstreeks] ; [].

hour1(Ho,Mi) -->
   hour_lex(Ho), [uur], min_lex(Mi).
hour1(Ho,Mi) -->
   min_lex(Mi), [over], hour_lex(Ho).
```

Figure 5: DCG for temporal expressions.

by a `min_lex`. Assuming suitable definitions for the lexical (pre-terminal) categories, this will generate such strings as `zondag vijf januari omstreeks tien uur vijftien` (*Sunday, January the fifth, at ten fifteen*). A more or less complete grammar of temporal expressions of this sort typically contains between 20 and 40 rules.

A test-corpus was constructed by extracting 2.500 utterances containing at least one lexical item signalling a temporal expression (such as a weekday, a month, or words such as `uur`, `minuut`, `week`, `morgen`, `kwart`, `omstreeks`, etc.) from a corpus of dialogues collected from a railway timetable information service. A subset of 200 utterances was annotated. The annotation indicates which part of the utterance is the temporal expression, and its semantics. An example is given below.

```
sentence(42,[ja,ik,wil,reizen,op,
   zesentwintig,januari,s_morgens,om,
   tien,uur,vertrekken], [op,
   zesentwintig, januari,s_morgens,om,
   tien,uur], temp(date(_,1,26),
              day(_,_,2),hour(10,_))).
```

The raw utterances and 100 annotated utterances were made available to students. A grammar can now be tested by evaluating how well it manages to spot temporal phrases within an utterance and assign the correct semantics to it. To this end, a parsing scheme was used which returned the (leftmost) maximal sub-phrase of an utterance that could be parsed as a temporal expression. This result was compared with the annotation, thus providing a measure for 'word accuracy' and 'semantic accuracy' of the grammar. The best solutions achieved over 95 % word and semantic accuracy.

```
head_complement_struct(Mthr,Hd,Comp) :-
    head_feature_principle(Mthr,Hd),
    Hd:comp <=> Comp.

rule(np_pp,vp/VP,[np/NP,pp/PP,v/V]) :-
    head_complement_struct(VP,V,np_pp),
    case(NP,acc),
    PP:head:pform <=> aan.
```

Figure 6: A fragment of the grammar for Dutch

Exercise IV: Unification grammar

Linguistically motivated grammars are almost without exception based on some variant of unification grammar (Shieber, 1986). Head-driven phrase structure grammar (HPSG) (Pollard and Sag, 1994) is often taken as the theoretical basis for such grammars. Although a complete introduction into the linguistic reasoning underlying such a framework is beyond the scope of this course, as part of a computational linguistics class students should at least gain familiarity with the core concepts of unification grammar and some of the techniques frequently used to implement specific linguistic analyses (underspecification, inheritance, gap-threading, unary-rules, empty elements, etc.).

To this end, we developed a core grammar of Dutch, demonstrating how subject-verb agreement, number and gender agreement within NP's, and subcategorization can be accounted for. Furthermore, it illustrates how a simplified form of gap-threading can be used to deal with unbounded dependencies, how the movement account for the position of the finite verb in main and subordinate clauses can be mimicked using an 'empty verb' and some feature passing, and how auxiliary-participle combinations can be described using a 'verbal complex'. The design of the grammar is similar to the OVIS-grammar (van Noord et al., 1999), in that it uses rules with a relatively specific context-free backbone. Inheritance of rules from more general 'schemata' and 'principles' is used to add feature constraints to these rules without redundancy. The schemata and principles, as well as many details of the analysis, are based on HPSG. Figure 6 illustrates the general format of phrase structure schemata and feature constraints.

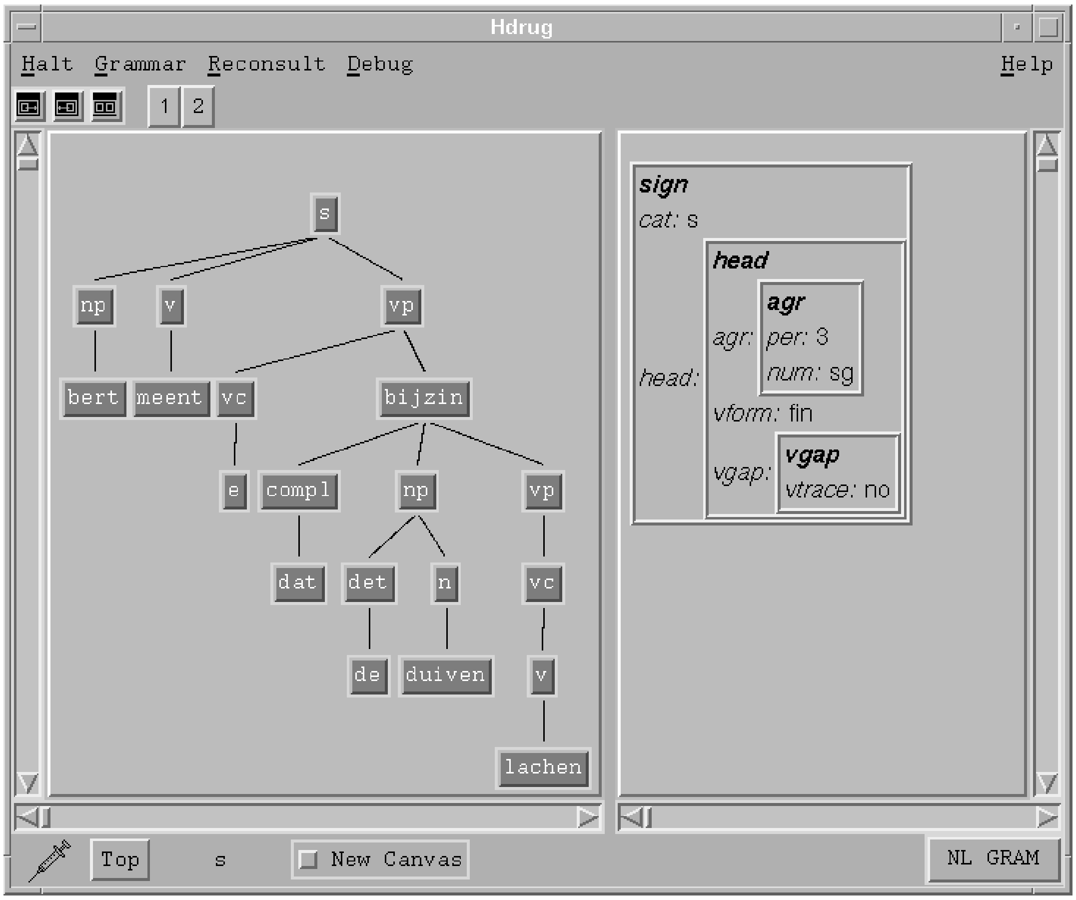

Figure 7: Screenshot of Hdrug

The grammar fragment is implemented using the HDRUG development system[5] (van Noord and Bouma, 1997). HDRUG provides a description language for feature constraints, allows rules, lexical entries, and ‘schemata’ or ‘principles’ to be visualised in the form of feature matrices, and provides an environment for processing example sentences which supports the display of derivation trees and partial parse results (chart items). A screen-shot of HDRUG is given in figure 7.

As an exercise, students had to extend the core fragment with rules and lexical entries for additional phrasal categories (PP's), verbal subcategorization types (verbs selecting for a PP-complement), NP constructions (determiner-less NP's), verb-clusters (modal+infinitive combinations), and WH-words (*wie, wat, welke, wiens, hoeveel, ...* (*who, what, which, whose, how many, ...*). To test the resulting fragment, students were also given a suite of example sentences which had to be accepted, as well as a suite of ungrammatical sentences. Both test suites were small (consisting of less than 20 sentences each) and constructed by hand. This reflects the fact that this exercise is primarily concerned with the implementation of a sophisticated linguistic analysis.

[5] `www.let.rug.nl/~vannoord/hdrug/`

4 Natural Language Interfaces

Practical courses in natural language interfaces or computational semantics (Pereira and Shieber, 1987; Blackburn and Bos, 1998) have used a toy database, such as geographical database or an excerpt of a movie script, as application domain. The growing amount of information available on the internet provides opportunities for accessing much larger databases (such as public transport time-tables or library catalogues), and therefore, for developing more realistic applications. In addition, many web-sites provide information which is essentially dynamic (weather forecasts, stock-market information, etc.), which means that applications can be developed which go beyond querying or summarising pre-defined sets of data. In this section, we describe two exercises in which a natural language interface for web-accessible information is developed. In both cases we used the PILLOW package[6] (Cabeza et al., 1996) to access data on the web and translate the resulting HTML-code into Prolog facts.

4.1 Exercise V: Natural Language Generation

Reiter and Dale (1997) argue that the generation of natural language reports from a database with numerical data can often be based on low-tech processing language engineering techniques such as pattern matching and template filling. Sites which provide access to numerical data which is subject to change over time, such as weather forecasts or stock quotes, provide an excellent application domain for a simple exercise in language generation.

For instance, in one exercise, students were asked to develop a weather forecast generator, which takes the long-term (5 day) forecast of the Dutch meteorological institute, KNMI, and produces a short text describing the weather of the coming days. Students were given a set of pre-collected numerical data as well as the text of the corresponding weather forecasts as produced by the KNMI. These texts served as a 'target corpus', i.e. as an informal definition of what the automatic generation component should be able to produce.

To produce a report generator involved the implementation of 'domain knowledge' (a 70% chance of rain means that it is 'rainy', if maximum and minimum temperatures do not vary more than 2 degrees, the temperature remains the same, else there is a change in temperature that needs to be reported, etc.) and rules which apply the domain knowledge to produce a coherent report. The latter rules could be any combination of format or write instructions and more advanced techniques based on, say, definite clause grammar. The completed system can not only be tested on pre-collected material, but also on the information taken from the current KNMI web-page by using the Prolog-HTTP interface.

A similar exercise was developed for the AEX (stock market) application described below. In this case, students we asked to write a report generator which reports the current state of affairs at the Dutch stock market AEX, using numerical data provided by the web-interface to the Dutch news service 'NOS *teletext*' and using similar reports on *teletext* itself as 'target-corpus'.

[6] `http://www.clip.dia.fi.upm.es/miscdocs/pillow/pillow.html`

4.2 Exercise VI: Question answering

Most natural language dialogue systems are interfaces to a database. In such situations, the main task of the dialogue system is to answer questions formulated by the user.

The construction of a question-answering system using linguistically-motivated techniques, requires (minimally) a domain-specific grammar which performs semantic analysis and a component which evaluates the semantic representations output by the grammar with respect to the database. Once these basic components are working, one can try to extend and refine the system by adding (domain-specific or general) disambiguation, contextual-interpretation (of pronouns, elliptic expressions, etc), linguistically-motivated methods for formulating answers in natural language, and scripts for longer dialogues.

In the past, we have used information about railway time-tables as application domain. Recently, a rich application domain was created by constructing a stock-market game, in which participants (the students taking the class and some others) were given an initial sum of money, which could be invested in shares. Participants could buy and sell shares at wish. Stock quotes were obtained on-line from the news service 'NOS *teletext*'. Stock-quotes and transactions were collected in a database, which, after a few weeks, contained over 3000 facts.

The unification-based grammar introduced previously (in exercise IV) was adapted for the current domain. This involved adding semantics and adding appropriate lexical entries. Furthermore, a simple question-answering module was provided, which takes the semantic representation for a question assigned by the grammar (a formula in predicate-logic), transforms this into a clause which can be evaluated as a Prolog-query, calls this query, and returns the answer.

The exercise for the students was to extend the grammar with rules (syntax and semantics) to deal with adjectives, with measure phrases (*vijf euro/procent* (*five euro/percent*), with date expressions (*op vijf januari* (*on January, 5*)), and constructions such as *aandelen Philips* (*Philips shares*), and *koers van VNU* (*price of VNU*) which were assigned a non-standard semantics Next, the question system had to be extended so as to handle a wider range of questions. This involved mainly the addition of domain-specific translation rules. Upon completion of the exercise, question-answer pairs of the sort illustrated in 8 were possible.

```
Q: wat is de koers van ABN AMRO
   what is the price of ABN AMRO
A: 17,75
Q: is het aandeel KPN gisteren gestegen
   have the KPN shares gone up yesterday
A: ja
   yes
Q: heeft Rob enige aandelen Baan verkocht
   has Rob sold some Baan shares
A: nee
   no
Q: welke spelers bezitten aandelen Baan
   Which players possess Baan shares
A: gb, woutr, pieter, smb
Q: hoeveel procent zijn de aandelen kpn
   How many percent have the KPN shares
   gestegen
   gone up
A: 5
```

Figure 8: Question-answer pairs in the AEX dialogue system.

5 Concluding remarks

Developing realistic and challenging exercises in computational linguistics requires support in the form of development tools and resources. Powerful tools are available for experimenting with finite state technology and unification-based grammars, resources can be made available easily using internet, and current hardware allows students to work comfortably using these tools and resources. The introduction of such tools in introductory courses has the advantage that it provides a realistic overview of language technology research and development. Interesting application area's for natural language dialogue systems can be obtained by exploiting the fact that the internet provides access to many on-line databases. The resulting applications give access to large amounts of actual and dynamic information. For educational purposes, this has the advantage that it gives a feel for the complexity and amount of work required to develop 'real' applications.

The most important problem encountered in developing the course is the relative lack of suitable electronic resources. For Dutch, the CELEX database provides a rich source of lexical information, which can be used to develop interesting exercises in computational morphology. Development of similar, data-oriented, exercises in the area of computational syntax and semantics is hindered, however, by the fact that resources, such as electronic dictionaries proving valence and concept information, and corpora annotated with part of speech, syntactic structure, and semantic information, are missing to a large extent. The development of such resources would be most welcome, not only for the development of language technology for Dutch, but also for educational purposes.

Acknowledgements

I would like to thank Gertjan van Noord for his assistance in the development of some of the materials and Rob Koeling for teaching the course on natural language interfaces with me. The material presented here is being developed as part of the module *natuurlijke taalinterfaces* of the (Kwaliteit & Studeerbaarheids-) project *brede onderwijsinnovatie kennissystemen* (BOK), which develops (electronic) resources for courses in the area of knowledge based systems. The project is carried out by several Dutch universities and is funded by the Dutch ministry for Education, Culture, and Sciences.

References

James F. Allen. 1987. *Natural Language Understanding.* Benjamin Cummings, Menlo Park CA.

Evan L. Antworth. 1990. *PC-KIMMO : a two-level processor for morphological analysis.* Summer Institute of Linguistics, Dallas, Tex.

R. H. Baayen, R. Piepenbrock, and H. van Rijn. 1993. *The CELEX Lexical Database (CD-ROM).* Linguistic Data Consortium, University of Pennsylvania, Philadelphia, PA.

Patrick Blackburn and Johan Bos. 1998. Representation and inference for natural language: A first course in computational semantics. Ms., Department of Computational Linguistics, University of Saarland, Saarbrücken.

D. Cabeza, M. Hermenegildo, and S. Varma. 1996. The pillow/ciao library for internet/www programming using computational logic systems. In *Proceedings of the 1st Workshop on Logic Programming Tools for INTERNET Applications*, JICSLP"96, Bonn, September.

Michael A. Covington. 1994. *Natural Language Processing for Prolog Programmers.* Prentice Hall, Englewood Cliffs, New Jersey.

Gerald Gazdar and Christopher S. Mellish. 1989. *Natural Language Processing in Prolog; an Introduction to Computational Linguistics.* Addison Wesley.

L. Karttunen, J.P. Chanod, G. Grefenstette, and A. Schiller. 1997. Regular expressions for language engineering. *Natural Lanuage Engineering*, pages 1–24.

Lauri Karttunen. 1995. The replace operator. In *33th Annual Meeting of the Association for Computational Linguistics*, pages 16–23, Boston, Massachusetts.

Fernando C.N. Pereira and Stuart M. Shieber. 1987. *Prolog and Natural Language Analysis*. Center for the Study of Language and Information Stanford.

Carl Pollard and Ivan Sag. 1994. *Head-driven Phrase Structure Grammar*. Center for the Study of Language and Information Stanford.

Ehud Reiter and Robert Dale. 1997. Building applied natural language generation systems. *Natural Language Engineering*, 3(1):57–87.

Stuart M. Shieber. 1986. *Introduction to Unification-Based Approaches to Grammar*. Center for the Study of Language and Information Stanford.

Gertjan van Noord and Gosse Bouma. 1997. Hdrug. a flexible and extendable environment for natural language processing. In Dominique Estival, Alberto Lavelli, and Klaus Netter, editors, *Computational Environments for Grammar Development and Linguistic Engineering*, pages 91–98, Somerset, NJ. Association for Computational Linguistics.

Gertjan van Noord, Gosse Bouma, Rob Koeling, and Mark-Jan Nederhof. 1999. Robust grammatical analysis for spoken dialogue systems. *Journal of Natural Language Engineering*. To appear.

Gertjan van Noord. 1997. FSA Utilities: A toolbox to manipulate finite-state automata. In Darrell Raymond, Derick Wood, and Sheng Yu, editors, *Automata Implementation*. Springer Verlag. Lecture Notes in Computer Science 1260.

Theo Vosse. 1994. *The Word Connection*. Ph.D. thesis, Rijksuniversiteit Leiden.