Squibs and Discussions

Dependency Unification Grammar for PROLOG

Friedrich Steimann*
Universität Karlsruhe

Christoph Brzoska*
Universität Karlsruhe

1. Introduction

The programming language PROLOG has proved to be an excellent tool for implementing natural language processing systems. Its built-in resolution and unification mechanisms are well suited to both accept and generate sentences of artificial and natural languages. Although supporting many different linguistic formalisms, its straightforwardness and elegance have perhaps best been demonstrated with definite clause grammars (DCGs) (Pereira and Warren 1980), an extension to PROLOG's syntax allowing direct implementation of rules of context-free grammars as Horn clauses.

While context-free grammars and DCGs—strongly related to the huge linguistic field of constituency or phrase structure grammars and their descendants—have become very popular among logic programmers, dependency grammars (DGs) have long remained a widely unnoticed linguistic alternative. DG is based on the observation that each word of a sentence has individual slots to be filled by others, its so-called dependents. Which dependents a particular word takes depends not only on its function within the sentence, but also on its meaning—like other contemporary linguistic frameworks, DG integrates both syntactic and semantic aspects of natural language.

DG was first formalized by Tesniére (1959) and later, among others, by Gaifman (1965) and Hays (1964). The formalization presented in this paper is based on Hellwig's Dependency Unification Grammar (DUG) (Hellwig 1986). We merely add a framework for automatic translation of DUG rules to Horn clauses that makes DUGs as easy to implement as classic DCGs.

2. Dependency Grammar as Context-Free Grammar

Whereas context-free grammars differentiate between terminals (coding the words of a language) and non-terminals (representing the constituents that are to be expanded), the symbols of a DG uniformly serve both purposes: like terminals they must be part of the sentence to be accepted (or generated), and like non-terminals, they call for additional constituents of the sentence. Despite this significant difference, DG can be defined in terms of context-free grammar, making the twofold role of its symbols explicit:

Definition
A context-free grammar $G = (T, N, P, S)$ where

> —terminals and non-terminals are related by a one-to-one mapping $f : T \rightarrow N \backslash \{S\}$ and

* Institut für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe, Germany. E-mail: {steimann,brzoska}@ira.uka.de

—each production in P is either of the form

$$s \rightarrow n_1 \ldots n_m$$

or of the form

$$n \rightarrow n_1 \ldots f^{-1}(n) \ldots n_m,$$

where $n, n_1, \ldots, n_m$ are elements of $N \backslash \{S\}$ and $s = S$

is a dependency grammar.

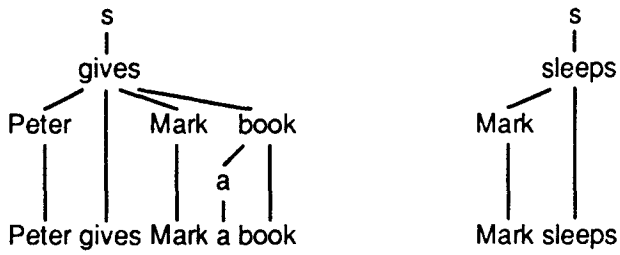
Accordingly, if atomic symbols are replaced by first-order terms, the following toy DG can be implemented in PROLOG using the DCG rule format:

```
s --> n(_, verb(_)).

n(give, verb(N)) -->
  n(_, noun(N)),
  [n(give, verb(N))],
  n(_, noun(_)),
  n(_, noun(_)).

n(sleep, verb(N)) -->
  n(_, noun(N)),
  [n(sleep, verb(N))].

n('Peter', noun(N)) -->
  [n('Peter', noun(N))].

n('Mark', noun(N)) -->
  [n('Mark', noun(N))].

n(book, noun(N)) -->
  n(_, det),
  [n(book, noun(N))].

n(a, det) -->
  [n(a, det)].
```

The terms $n(.,.)$ provide space for feature structures commonly employed to capture syntactic and semantic properties of words (Shieber 1986; Knight 1989). They serve only as an example here; other structures, including that used by Hellwig (1986), can also be employed.

Prior to parsing, each sentence must be converted to a string of terms holding the features derived through lexical analysis. This preprocessing step also resolves lexical ambiguities by representing words with alternative meanings through different symbols. Parsing the sentences "Peter gives Mark a book" and "Mark sleeps" with the

above DCG produces the following dependency trees:



3. Direct Transformation of DUG Rules to Horn Clauses

Although implementing DUG as DCG works acceptably, it makes no use of the rules' regular form: note how, when parsing the sentence "Mark sleeps," the parser calls several rules before it realizes that the rule for *give* must fail (because the sentence does not contain *give*), even though the head already indicates that *give* is required for the rule to succeed. If, however, the word partially specified as $n(\_, verb(\_))$ in the body of the start rule is accepted before the next rule is selected, an intelligent parser can exploit the fact that the sentence's verb is *sleep* and immediately call the appropriate rule. We therefore suggest an alternative syntax and translation scheme that produces a more efficient DUG parser.

In our DUG syntax, the head of a rule is separated from its body (holding the dependents of the word in the head) by the binary infix operator :>. The start rule

```
s :> n(_, verb(_)).
```

is translated to the Horn clause

```
s(_G1, _G2) :-
     accept(n(_G3, verb(_G4)), _G1, _G5),
     n(_G3, verb(_G4), _G5, _G2).
```

where the arguments appended to each predicate hold input and output sentence, respectively, and where an accept predicate is inserted before each literal of the rule body.[1] Accordingly,

```
n(sleep, verb(N)) :> n(_, noun(N)).
```

becomes

```
n(sleep, verb(N), _G1, _G2) :-
     accept(n(_G3, noun(N)), _G1, _G4),
     n(_G3, noun(N), _G4, _G2).
```

Note that the head literal of the *sleep* rule need not be repeated in the body because the respective word is removed from the input sentence before the rule is called (in this case in the start rule). The fact that a word has no dependent is coded by

```
n(well, adverb) :> [].
```

1 The implementation of $accept(\ldots)$ can be found in the appendix.

and translated to

```
n(well, adverb, _G1, _G1).
```

Like other contemporary grammar formalisms, DUG comes with syntactic extensions that code optionality and references.

3.1 Optionality

Many dependents are optional. Rather than providing an alternative rule for every possible combination of dependents, it is more convenient to declare a dependent optional, meaning that a sentence is correct independent of its presence. For example,

```
n(sleep, verb(N)) :> n(_, noun(N)), ? n(_, adverb).
```

where ? precedes the optional dependent, is implemented as

```
n(sleep, verb(N), _G1, _G2) :-
     accept(n(_G3, noun(N)), _G1, _G4), n(_G3, noun(N), _G4, _G5),
     ((accept(n(_G6, adverb)), _G5, _G7), n(_G6, adverb, _G7, _G2))
     ;
     _G5=_G2).
```

accepting "Mark sleeps" as well as "Mark sleeps well."

3.2 Referencing

References account for the fact that many words are similar in terms of the dependents they take. In order not to repeat the same set of rules over and over again, a reference operator ⇒ (read 'goes like') is introduced that causes branching to the rule of an analogous word, as in

```
n(yawn, verb(N)) :> ==> n(sleep, verb(N)).
```

In this case, the word *sleep* being referred to is not a dependent of *yawn*, the PROLOG translation

```
n(yawn, verb(N), _G1, _G2) :- n(sleep, verb(N), _G1, _G2).
```

therefore branches to the rule for *sleep* without accepting the word *sleep*.

As a side effect, references introduce quasi non-terminals to DUG. For example, by factoring out common dependency patterns, it is possible to generalize the rules for transitive verbs and allow for exceptions to the rule at the same time:

```
% standard dependents of transitive verbs in active voice
transverb(N, active) :>
     word(_, noun(N)),           % subject
     word(_, noun(_)).           % object

% standard dependents of transitive verbs in passive voice
transverb(N, passive) :>
     word(_, noun(N)),           % subject
     ? word(by, preposition).    % optional agent

% standard transitive verb
word(like, verb(N, Voice)) :>
     ==> transverb(N, Voice).
```

```
% transitive verb with additional indirect object
word(give, verb(N, Voice)) :>
     ==> transverb(N, Voice),
     word(_, noun(_)).
```

4. A Word about Word Order

Following Hellwig's DUG formalism, our PROLOG implementation does not code word order directly in the rules. Other DG formalisms, such as the one proposed by Gaifman (1965) and Hays (1964), mark the position of the head among its dependents by a special symbol in the body. The DUG parser can be adapted to follow this convention by accepting the symbol *self* in the rule body as in

```
n(sleep, noun(N)) :> n(_, noun(N)), self.
```

and by modifying both the preprocessor and the *accept* predicate so that the input sentence is split at the position of the dependent accepted and left and right remainders are passed to the next rules separately. However, many natural languages leave word order rather unconstrained, and its adequate handling is not a problem specific to DGs (see, for example, Pereira 1981, and Covington 1990).

5. Notes on Performance

The presented DUG formalism with free word order has successfully been employed to parse Latin sentences. Tracing showed that backtracking was considerably reduced as compared with an equivalent phrase structure grammar, although no good upper bound for complexity could be found (Steimann 1991). Although the pure DG formalism proved to be particularly practical for integration of idioms and exceptions, its lack of constituent symbols, i.e., non-terminals, would have lead to a grammar of enormous size and made it difficult to integrate special Latin constructs such as *accusative cum infinitive* or *ablative absolute.*

However, as shown above, DUG is a hybrid grammar: although dependency rules are the backbone of the formalism, it allows the introduction of quasi non-terminals that are integrated into the grammar via references. If desired, phrase structure rules can thus easily be combined with ordinary dependency rules.

The size of a grammar can be further reduced by introduction of order-sorted feature types (Aït-Kaci and Nasr 1986) supporting variable numbers of labeled arguments and subtyping. Using feature types instead of constructor terms for representing the words of a language increases readability and enables abstraction of rules as well as implementation of semantic type hierarchies supporting selectional restrictions (Steimann 1991).

References

Aït-Kaci, H., and Nasr, R. (1986). "LOGIN: A logic programming language with built-in inheritance." *The Journal of Logic Programming* 3:185–215.

Covington, M. A. (1990). "Parsing discontinuous constituents in dependency grammar." *Computational Linguistics* 16(4):234–236.

Gaifman, H. (1965). "Dependency systems and phrase-structure systems." *Information and Control* 8:304–337.

Hays, D. G. (1964). "Dependency theory: A formalism and some observations." *Language* 40(4):511–525.

Hellwig, P. (1986). "Dependency unification grammar." In *Proceedings, 11th International Conference on Computational Linguistics* (COLING 1986). University of Bonn, Bonn. 195–198.

Knight, K. (1989). "Unification: A multidisciplinary survey." *ACM Computing Surveys* 21(1):105–113.

Pereira, F. (1981). "Extraposition grammars." *American Journal of Computational Linguistics* 7(4):243–255.

Pereira, F., and Warren, D. H. D. (1980). "Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks." *Artificial Intelligence* 13:231–278.

Shieber, S. M. (1986). "An introduction to unification-based approaches to grammar." CLSI Lecture Notes, No. 4, Stanford University, Stanford, California.

Steimann, F. (1991). "Ordnungssortierte feature-Logik und Dependenzgrammatiken in der Computerlinguistik." Diplomarbeit Universität Karlsruhe, Fakultät für Informatik, Karlsruhe, Germany.

Tesnière, L. (1959). *Elements de syntaxe structurale.* Paris: Librairie Klincksiek.

Appendix A

The DUG Preprocessor

The following PROLOG source code implements a simple preprocessor that converts source files containing DUG rules into target files consisting of Horn clauses only. Automatic creation of the parse tree has also been implemented. However, it is omitted here for clarity.

Note that every call to a start rule must be extended by two list arguments: the input and the output sentence (the latter usually being the empty list []).

```
% operator directives (priorities must be adapted)
:- op(1200, xfx, ':>').
:- op(600, fx, '?').
:- op(500, fx, '==>').

dug(Source, Target) :-
     see(Source),
     tell(Target),
     convert,
     seen,
     told.

convert :-
     read(DUGClause),
        (DUGClause = end_of_file
        ;
        convert(DUGClause, PClause),
        displayq(PClause), write('.'), nl,
        convert).

% DUG rule
convert((HeadIn :> BodyIn), (HeadOut :- BodyOut)) :-
     !, HeadIn =.. [Pred|Args],
     append(Args, [In, Out], Expanded),
     HeadOut =.. [Pred|Expanded],
     convert(BodyIn, BodyOut, In, Out).

% other
convert(Clause, Clause).

% conjunction
convert((AIn, BIn), (AOut, BOut), In, Out) :-
     !, convert(AIn, AOut, In, Intermediate),
     convert(BIn, BOut, Intermediate, Out).

% option
convert(? AIn, ((AOut); In = Out), In, Out) :-
     !, convert(AIn, AOut, In, Out).

% reference
convert (==> AIn, AOut, In, Out) :-
```

```
     !, AIn =.. [Pred|Args],
     append(Args, [In, Out], Expanded),
     AOut =.. [Pred|Expanded].

% no dependents
convert([], true, In, In) :- !.

% dependent (introduces call to 'accept')
convert(AIn, (accept(AIn, In, Intermediate), AOut), In, Out) :-
     AIn =.. [Pred|Args],
     append(Args, [Intermediate, Out], Expanded),
     AOut =.. [Pred|Expanded].
```

The accept predicate that must be included in every program containing DUG rules can be implemented as follows:

```
accept(Element, [Element|String], String).

accept(Element, [Other|StringIn], [Other|StringOut]) :-
     accept(Element, StringIn, StringOut).
```