COMPILING TRACE & UNIFICATION GRAMMAR FOR PARSING AND GENERATION

Hans Ulrich Block
Siemens AG, Corporate Research, ZFE IS INF 23
Otto Hahn-Ring 6
D-8000 München 83
Germany
block@ztivax.uucp

ABSTRACT

This paper presents Trace & Unification Grammar (TUG), a declarative and reversible grammar formalism that brings together Unification Grammar (UG) and ideas of Government & Binding Theory (GB) in an undogmatic way. A grammar compiler is presented that transforms a grammar written in the TUG formalism into two different forms, one being useful for parsing, the other being useful for generation.

1 INTRODUCTION

During the last years there has been a growing interest in NL systems that can be used for both parsing and generation. The invention of unification grammar that allows for a declarative description of language made it possible to use the same grammar for both tasks. The main goal of a grammar then is to describe a relation between normalized (semantic) representations and language strings. A grammar that can be used in both directions is called "reversible".

We can distinguish three levels of reversibility. On the first level, not only the same grammar is used for parsing and generation, but also the interpreter for parsing and generation is reversible. This approach is taken in Shieber (1988). Besides elegance the approach has the advantage that the reversibility is guaranteed. Further advantages are mentioned in Neumann (1991). As a disadvantage, it is yet unclear whether and how these systems can be made efficient.

On the second level we find systems where the same reversible grammar is processed by two different interpreters, one for parsing, one for generation. The advantage of these systems is that the grammar can be changed and tested easily, which helps to shorten the development cycle. The disadvantage again is that grammar interpreters are usually too slow to be used in realistic systems.

On the third level we finally find systems, where the linguistic description is given in a reversible declarative grammar. This grammar is then compiled into two different forms, one being usefull only for parsing, the other only for generation. Whereas here we have to face the disadvantage that compiling can take some time and therefore prolongs the development cycle, the advantage lies in the efficient processing that can be achieved with compiled grammars. Strzalkowski (1990) and Strzalkowski/Peng (1990) describe a compiler that transforms a grammar originally written for parsing into an efficient generator.

In the follwing section I will present a system of the third type and show by means of which compiling methods a grammar written in a perspicious formalism, TRACE AND UNIFICATION GRAMMAR (TUG) can be transformed to fast parsers and generators. The proposed compilers and their modular architecture have the further advantage that most of their parts can be used also for other formalisms than the one described, e.g. DCGs.

The whole system is part of a polyfunctional linguistic processor for German called LINGUISTIC KERNEL PROCESSOR (LKP). The LKP contains a grammar of German with broad coverage. The grammar describes the relation between a subset of German and a subset of QLF, the intermediate semantic form that is used in the *Core Language Engine* of SRI Cambridge (Alshawi 1990). The LKP has been implemented in PROLOG. Parsing and Generation of a sentence up to 15 words normally takes between 1 and 10 seconds, with a strong tendency to the lower bound.

2 FORMALISM

The design of Trace and Unification Grammar has been guided by the following goals:

- **Perspicuity**. We are convinced that the generality, coverage, reliability and development speed of a grammar are a direct function of its perspicuity, just as programming in Pascal is less errorprone than programming in assembler. In the optimal case, the grammar writer should be freed of reflections on how to code things best for processing but should only be guided by linguistic criteria. These goals led for example to the introduction of unrestricted disjunction into the TUG formalism.
- **Compatibility to GB Theory**. It was a major objective of the LKP to base the grammar on well understood and motivated grounds. As most of the newer linguistic descriptions on German are in the framework of GB theory, TUG was designed to be somehow compatible with this theory though it was not our goal to "hardwire" every GB principle.
- **Efficiency**. As the LKP is supposed to be the basis of products for interactive usage of natural language, efficiency is a very important goal. Making efficiency a design goal of the formalism led e.g. to the introduction of feature types and the separation of the movement rules into head movement and argument movement.

The basis of TUG is formed by a context free grammar that is augmented by PATR II-style feature equations. Besides this basis, the main features of TUG are feature typing, mixing of attribute-value-pair and (PROLOG-) term unification, flexible macros, unrestricted disjunction and special rule types for argument and head movement.

2.1 BASIC FEATURES

As a very simple example we will look at the TUG version of the example grammar in Shieber (1984).

```
% type definition

s       => f.
np      => f(agr:agrmnt).
vp      => f(agr:agrmnt).
v       => f(agr:agrmnt).

agrmnt => f(number:number,person:person).

number => {singular,plural}.
person => {first,second,third}.

% rules

s ---> np, vp |
     np:agr = vp:agr.

vp ---> v, np |
     vp:agr = v:agr.

% lexicon

lexicon('Uther',np) |
     agr:number = singular,
     agr:person = third.
lexicon('Arthur',np) |
     agr:number = singular,
     agr:person = third.
lexicon(knights,v) |
     agr:number = singular,
     agr:person = third.
lexicon(knight,v) |
     ( agr:number = singular,
       ( agr:person = first
       ; agr:person = second
       )
     ;
       agr:number = plural
     ).
```

The two main differences to PATR II in the basic framwork are that first, TUG is less flexible in that it has a "hard" contextfree backbone, whereas in PATR II categories of the context free part are placeholders for feature structures, their names beeing taken as the value of the **cat** feature in the structure. Second, TUG has a strict typing. For a feature path to be well defined, each of its attributes has to be declared in the type definition.

Besides defined attribute-value-pairs, TUG allows for the mixing of attribute-value-pair unification with arbitrary structures like PROLOG terms using a back-quote notation. This can be regarded as the unificational variant of the **BUILDQ** operation known from ATNs. As an example consider the following lexicon entry of *each* that constructs a predicate logic notation out of **det:base**,

det:scope and det:var.

```
lexicon(each,det) |
     det:sem =
          'all(det:var,det:base ->
                         det:scope)
```

During our work on the German grammar we found that this feature was very useful for the construction of semantic forms.

TUG provides templates for a clearer organization of the grammar. The agreement in the above mentioned grammar might have been formulated like this:

```
agree(X,Y) short_for
     X:agr = Y:agr.

...

s ---> np, vp |
     agree(np,vp).
```

TUG allows for arbitrary disjunction of feature equations. Disjunctions and Conjunction may be mixed freely. Besides well known cases as in the entry for *knight* above, we found many cases where disjunctions of path equations are useful, e.g. for the description of the extraposed relative clauses[1].

2.2 MOVEMENT RULES

Further to these more standard UG-features, TUG provides special rule formats for the description of discontinuous dependencies, so called "movement rules". Two main types of movement are distinguished: argument movement and head movement. The format and processing of argument movement rules is greatly inspired by Chen *e.a.* (1988) and Chen (1990), the processing of head movement is based on GPSG like slash features.

Head Movement

A head movement rule defines a relation between two positions in a parse tree, one is the landing site, the other the trace position. Head movement is constrained by the condition that the trace is the head of a specified sister (the root node) of the landing site[2]. Trace and Antecedent are identical with the exception that the landing site contains overt material, the trace does'nt. Suppose, that v is the head of vk, vk the head of vp and vp the head of s, then only the first of the following structures is a correct head movement, the second is excluded because np is not head of vp, the third because antecedent and trace are unequal.

$[_{s'}$ v_i $[_s$ ... $[_{vp}$ ...
  $[_{vk}$ ... trace(v)$_i$ ...]...]...]...]

$[_{s'}$ np_i $[_s$ ... $[_{vp}$ trace(np)$_i$ ...
  $[_{vk}$ ... v ...]...]...]

$[_{s'}$ np_i $[_s$ ... $[_{vp}$ ...
  $[_{vk}$ ... trace(v)$_i$ ...]...]...]...]

To formulate head movement in TUG the following format is used. First, a head definition defines which category is the head of which other.

```
v is_head_of vk.
vk is_head_of vp.
vp is_head_of s.
```

Second, the landing site is defined by a rule like

```
s' ---> v+s | ...
```

To include recursive rules in the head path, heads are defined by the following head definitions. In a structure $[_M$ D_1 ... $D_n]$ D_i is the head of M if either D_i is_head_of M is defined or D_i has the same category as M and either D_i is_head_of X or X is_head_of D_i is defined for any category X.

Head movement rules are very well suited for a concise description of the positions of the finite verb in German (sentence initial, second and final) as in

Hat$_i$ der Mann der Frau das Buch gegeben t$_i$?

Has$_i$ the man the woman the book given t$_i$

Der Mann hat$_i$ der Frau das Buch gegeben t$_i$

The man has$_i$ the woman the book given t$_i$

... daß der Mann der Frau das Buch gegeben hat

... that the man the woman the book given has

All that is needed are the head definitions and the rule that introduces the landing site[3].

[1] Block/Schmid (1991) describes our processing technique for disjunctions.

[2] Here, "head of" is a transitive relation s.t. if x is head of y and y is head of z then x is head of z.

[3] Even though verb movement is not supposed to be a topic for English grammar, one might think of describing English Subj-Aux inversion in terms of head movement.
Peter has been reading a book
Has$_i$ Peter t$_i$ been reading a book

Argument Movement

Argument movement rules describe a relation between a landing site and a trace. The trace is always c-commanded by the landing site, its antecedent. Two different traces are distinguished, anaphoric traces and variable traces. Anaphoric traces must find their antecedent within the same bounding node, variable trace binding is constrained by subjacency, e.a. the binding of the trace to its antecedent must not cross two bounding nodes. Anaphoric traces are found for example in English passive constructions [s [np The book of this author]$_i$ was read t_i] whereas variable traces are usually found in wh-constructions and topicalization. Similar to the proposal in Chen *e.a.* (1988), argument movement is coded in TUG by a rule that describes the landing site, as for example in

```
s2 ---> np:ante<trace(var,np:trace), s1 |
     ante:fx = trace:fx,
     ...
```

This rule states that `np:ante`[4] is the antecedent of an np-trace that is dominated by s1. This rule describes a leftward movement. Following Chen's proposal, TUG also provides for rightward movement rules, though these are not needed in the German grammar. A rightward movement rule might look like this.

```
s2 ---> s1, trace(var,np:trace)>np:ante |
     ante:fx = trace:fx,
     ...
```

The first argument in the trace-term indicates whether the landing site is for a variable (`var`) or for an anaphoric (`ana`) trace. Other than head movement, where trace and antecedent are by definition identical, the feature sharing of argument traces with their antecedents has to be defined in the grammar by feature equations (`ante:fx = trace:fx, ...`). Furthermore, it is not necessary that the antecedent and the trace have the same syntactic category. A rule for pronoun fronting in German might e.g. look like this:

```
spr ---> pron<trace(ana,np), s | ...
```

[4]The notation `Cat:Index` is used to distinguish two or more occurrences of the same category in the same rule in the equation part. `:ante` and `:trace` are arbitrary names used as index to refer to the two different nps.

The current version of the formalisms requires that the grammar contains a declaration on which categories are possible traces. In such a declaration it is possible to assign features to a trace, for example marking it as empty:

```
trace(np) | np:empty = yes.
```

Bounding nodes have to be declared as such in the grammar by statements of the form

```
bounding_node(np).
bounding_node(s) | s:tense = yes.
```

As in the second case, bounding nodes may be defined in terms of category symbols and features[5]. Typical long distance movement phenomena are described within this formalism as in GB by trace hopping. Below is a grammar fragment to describe the sentence *Which books$_i$ do you think t_i John knows t_i Mary did'nt understand t_i*:

```
bounding_node(s).
bounding_node(np).

s1 ---> np<trace(var,np), s | ...
s ---> np, vp | ...
s ---> aux, np, vp | ...
np ---> propernoun | ...
np ---> det, n |
vp ---> v, s1 | ...
vp ---> v, np | ...

trace(np).
```

The main difference of argument movement to other approaches for the description of discontinuities like extraposition grammars (Pereira 1981) is that argument movement is not restricted to nested rule application. This makes the approach especially atractive for a scrambling analysis of the relative free word order in the German *Mittelfeld* as in

Ihm$_i$ hat$_j$ das Buch$_k$ keiner t_i t_k gegeben t_j.

3 PROCESSING TRACE & UNIFICATION GRAMMAR

TUG can be processed by a parser and a generator. Before parsing and generation, the grammar is compiled to a more efficient form.

[5]Currently, only conjunction of equations is allowed in the definition of bounding nodes.

The first compilation step is common to generation and parsing. The attribute-value-pair structure is transformed to (PROLOG) term structure by a TUG-to-DCG converter. This transformation makes use of the type definitions. As an example consider the transformation of the grammar

```
a     => f(a1:t1).
b     => f(a1:t1).

t1     => f(a2:t2,a3:t3).

t2     => {1,2}.
t3     => {2,3}.

a ---> b |
     a:a1:a3 = 2,
     ( a:a1:a2 = 1; a:a1 = b:a1 ).
```

It is transformed to the following grammar in a DCG like format[6].

```
a(t1(A,2)) --->
     [b(B), {A = 1 ; t1(A,2) = B}].
```

The compilation steps following the TUG-to-DCG converter are different for parsing and generation.

3.1 THE PARSER GENERATOR

In the LKP, a TUG is processed by a Tomita parser (Tomita 1986). For usage in that parser the result of the TUG-to-DCG converter is compiled in several steps:

- expansion of head movement rules
- transformation of argument movement rules
- elimination of empty productions
- conversion to LR(K) format
- computation of LR tables

First, head movement rules are eliminated and the grammar is expanded by introducing slash rules for the head path by the head movement expander. Suppose the TUG-to-DCG converter has produced the following fragment:

[6]Note that the goal {A = 1 ; t1(A,2) = B} is interpreted as a constraint and not as a PROLOG goal as in DCGs. See Block/Schmid (1991) for the evaluation of the constraints.

```
v(_) is_head_of vk(_).
vk(_) is_head_of vp(_).
vp(_) is_head_of s(_).

s1(S1) ---> [v(V) + s(S)].
s(S) ---> [...,vp(VP),...].
vp(VP) ---> [...,vk(VK),...].
vk(VK) ---> [...,v(V),...].
```

Then, the head movement expander introduces slash rules[7] along the head-path, thereby introducing the empty nonterminals **push(X)** and **pop(X)**.

```
% rules sofar
s(S) ---> [...,vp(VP),...].
vp(VP) ---> [...,vk(VK),...].
vk(VK) ---> [...,v(V),...].

% newly introduced slash rules
s1(S1) ---> [v(V), push(v(V)), s_v(S)].
s_v(S) ---> [...,vp_v(VP),...].
vp_v(VP) ---> [...,vk_v(VK),...].
vk_v(VK) ---> [...,v_v(V),...].
v_v(V)---> [pop(v(V))].

% empty productions for push and pop
push(X) ---> [].
pop(X) ---> [].
```

push(X) and pop(X) are "marker rules" (Aho/Sethi/Ullman 1986) that invoke the parser to push and pop their argument onto and off a left-to-right stack. This treatment of head movement leads to a twofold prediction in the Tomita parser. First, the new slash categories will lead to LR parsing tables that predict that the verb will be missing if rule **s1 ---> ...** has applied. Second, the feature structure of the verb is transported to the right on the left-to-right stack. Therefore, as soon as a **v_v** is expected, the whole information of the verb, e.g. its subcategorization frame, is available. This strategy leads to a considerable increase in parsing efficiency.

In the next compilation phase, argument movement rules are transformed to the internal format. For the control of gaps a gap-threadding mechanism is introduced. Following Chen *e.a.* (1988), the gap features are designed as multisets, thus allowing crossing binding relations as mentioned in section 2.

[7]A slashed category **X/Y** is represented using the underscore character **X_Y**.

To see the effect of this compilation step, take the following fragment as output of the head movement expander.

```
bounding_node(s(_)).
s1(S1) ---> np(NP)<trace(var,np(Trace)),
     s(S).
s(S) ---> np(NP), vp(VP).
vp(VP) ---> v(V).
vp(VP) ---> v(V), np(NP).
trace(np(_)).
```

The argument movement expander transforms this to the following grammar.

```
s1(Gi,Go,S1) ---> np(Gi,Gt,NP),
     s(Gs,Go,S),
     {cut_trace(trace(var,np(Trace)),
                Gs,Gt)}.
s(Gi,Go,S) ---> np(Gi,Gt,NP),
       vp(Gt,Go,VP),
       {bound(Gi)}.
vp(Gi,Go,VP) ---> v(Gi,Go,V).
vp(Gi,Go,VP) ---> v(Gi,Gt,V),
      np(Gt,Go,NP).
np([trace(_,np(NP))|G],G,NP) ---> [].
```

The predicates `cut_trace/3` and `bound/1` are defined as in Chen *e.a.* (1988).

The next step, the empty production eliminater, eliminates all empty productions except those for `push` and `pop`. This transforms the output of the argument movement expander to the following grammar.

```
s1(Gi,Go,S1) ---> np(Gi,Gt,NP),
     s(Gs,Go,S),
     {cut_trace(trace(var,np(Trace)),
                Gs,Gt)}.
s1([trace(_,np(NP))|Gt],Go,S1) --->
     s(Gs,Go,S),
     {cut_trace(trace(var,np(Trace)),
                Gs,Gt)}.
s(Gi,Go,S) ---> np(Gi,Gt,NP),
     vp(Gt,Go,VP),
     {bound(Gi)}.
s([trace(_,np(NP))|Gt],Go,S) --->
     vp(Gt,Go,VP),
     {bound(Gi)}.
vp(Gi,Go,VP) ---> v(Gi,Go,V).
vp(Gi,Go,VP) ---> v(Gi,Gt,V),
     np(Gt,Go,NP).
vp(Gi,Go,VP) --->
     v(Gi,[trace(_,np(NP))|Go],V).
```

Elimination of empty productions allows for a simpler implementation of the Tomita parser, which again leads to an increased efficiency.

The next step, the DCG-to-LRK converter splits the grammar rules into a context free and a DCG part. A context free rule is represented as `rule(No,LHS,RHS)`, a DCG rule as `dcg_rule(No,LHS,RHS,Constraint)`. Rules are synchronized by their numbers. After this step the above grammar fragment is represented in the following format.

```
rule(1,s1,[np,s]).
rule(2,s1,[s]).
rule(3,s,[np,vp]).
rule(4,s,[vp]).
rule(5,vp,[v]).
rule(6,vp,[v,np]).

dcg_rule(1,s1(Gi,Go,S1),
     [np(Gi,Gt,NP),s(Gs,Go,S)],
     cut_trace(trace(var,np(Trace)),
               Gs,Gt)).
dcg_rule(2,s1([trace(_,np(NP))|Gt],
              Go,S1),
      [s(Gs,Go,S)],
      cut_trace(trace(var,np(Trace)),
                Gs,Gt)).
dcg_rule(3,s(Gi,Go,S),
      [np(Gi,Gt,NP),vp(Gt,Go,VP)],
      bound(Gi)).
dcg_rule(4,s([trace(_,np(NP))|Gt],
             Go,S),
      [vp(Gt,Go,VP)],
      bound(Gi)).
dcg_rule(5,vp(Gi,Go,VP),
      [v(Gi,Gv,V)],
      ( Gv = Go
      ; Gv = [trace(_,np(NP))|Go]
      ) ).
dcg_rule(6,vp(Gi,Go,VP),
      [v(Gi,Gt,V),np(Gt,Go,NP)],
      true).
```

Note that during this step, different rules that share the same context free backbone are transformed to a single context free rule. The difference in their feature structure is expressed in a disjunction in the Constraint (e.g. rule 5). As very often traces occur in optional positions (e.g. objects, as in `vp ---> v.  vp ---> v, np`), the elimination of empty productions (traces) considerably reduces the amount of edges the parser has to build.

After these compilation steps the context free rules are transformed to YACC format and YACC is used to compute the LR parsing table. Finally, YACC's `y.output` file is transformed to PROLOG.

3.2 THE GENERATOR GENERATOR

For generation with TUG an improved version of the semantic-head-driven generator (SHDG) (see Shieber *e.a.* 1990) is used. Before beeing useful for generation, the grammar is transformed in the following steps:

- expansion of head movement rules
- transformation to the semantic head driven generator format
- expansion of movement rules
- elimination of nonchainrules with uninstantiated semantics
- goal reordering and transformation to executable prolog code

First, the head movement expander transforms the head movement rules. As in the parser generator, slashed categories are generated along the head path, but no push and pop categories are introduces. Instead, the head movement rule and the trace are treated similar to argument movement. The resulting relevant new rules from the example above are:

```
% newly introduced slash rules
s1(S1) ---> [v(V)<trace(var,v_v(V)),
       s_v(S)].
s_v(S) ---> [...,vp_v(VP),...].
vp_v(VP) ---> [...,vk_v(VK),...].
vk_v(VK) ---> [...,v_v(V),...].
trace(_,v_v(V)).
```

In the next step rule symbols are transformed to the `node(Cat,S,S0)` format needed by the semantic-head-driven generator. Thereby disjunctions on the semantic argument as in the following example

```
a(Sem) ---> b(BSem), c(CSem),
      {BSem = Sem; CSem = Sem}.
```

are unfolded (multiplied out) to different rules. The output of this step for the above rule is:

```
node(a(Sem),S,S0) --->
     node(b(Sem),S,S1),
     node(c(CSem),S1,S0).
node(a(Sem),S,S0) --->
     node(b(BSem),S,S1),
     node(c(Sem),S1,S0).
```

Obviously, unfolding of semantic disjunctions is necessary for a correct choice of the semantic head.

The next compilation cycle expands the movement rules. Similar to the parser generator two arguments for gap threadding are introduced. The filling of the arguments and the transformation of the movement rules is different from the parser generator. It is a rather complicated operation which is sensitive to the semantics control flow.

Given a rule

```
a(A) ---> b(B)<trace(var,b(BT)), c(C)}
```

we can distinguish two cases:

1) The rule is a *nonchain rule* in the sense of Shieber *e.a.* (1990) or it is a *chain rule* and the antecedent of the trace is the semantic head. In this case the antecedent has to be generated prior to the trace. A typical example is a predicate logic analysis as in:

```
node(s1(Sem),S,S0) --->
     node(np(Sem,SemIn) <
            trace(var,np(NPSem,NPSem)),
          S,S1),
     node(s(SemIn),S1,S0).
```

As the antecedent carries the semantic information, it is expanded at the landing site, while the trace is just empty:

```
node(s1(Gi,Go,Sem),S,S0) --->
    node(np(Gi,Gt,Sem,SemIn),S,S1),
    node(s(Gt,Gs,SemIn),S1,S0),
    {cut_trace(trace(var,np(NPSem,NPSem),
                Gs,Gi)}.
node(np([trace(var,np(NPSem,NPSem))|Go],
    Go,NPSem,NPSem),S,S).
```

2) If any element other than the antecedent is the semantic head, then this head has to be generated prior to the antecedent. As the head might contain the trace, it also has to be generated prior to its antecedent. Consider the rule:

```
node(s1(Sem),S,S0) --->
```

```
node(np(NPSem)<trace(var,np(NPSem)),
     S,S1),
node(s(Sem),S1,S0).
```

In this rule s is generated prior to np. Within s, the trace of np will be generated. Following the suggestion in Shieber *e.a.* (1990), rules like this are compiled in such a way that an antecedent is generated in the trace position without linking it to the input string. This antecedent is then added to the set of gaps together with its starting and ending position (coded as a difference list). When generation comes to the landing site, the antecedent is cut out of the trace set. Thereby its starting and ending position is unified with the landing site's start and end positions. The translation of the above rule is:

```
node(s1(Gi,Go,Sem),S,S0) --->
    node(s(Gs,Go,Sem),S1,S0),
    {cut_trace(trace(var,np(NPSem),S,S1),
               Gs,Gi)}.
node(np([trace(var,np(NPSem),S,S0)|Go],
    Go,NPSem),SX,SX) --->
    node(np(G,G,NPSem),S,S0).
```

In the next step, a certain class of nonchain rules is eliminated from the grammar. One of the basic inefficiencies of the semantic-head-driven generator in Shieber *e.a.* (1990) has its origin in nonchain rules whose left-hand-side-semantics is a variable. This kind of nonchain rule often results from empty productions or lexicon entries of semantically empty words. For instance, in a grammar and lexicon fragment like

```
vk(SC)/Sem ---> aux(VKSem,SC,VKSC)/Sem,
    vk(VKSC)/VKSem.

aux(VKSem,SC,SC)/past(VKSem) ---> [has].
aux(Sem,SC,[Subj|SC])/Sem ---> [is].
```

the rule introducing *is* is a nonchain rule whose semantics is a variable and thus cannot be indexed properly. Rules like this one are eliminated by a partial evaluation technique. For each grammar rule that contains the left-hand-side of the rule on its right-hand-side, a copy of the rule is produced where the variables are unified with the left-hand-side of the nonchain rule and the corresponding right-hand-side element is replaced with the right-hand-side of the nonchain rule. E.g. insertion of the rule for *is* into the vk-rule above leads to

```
vk(SC)/Sem ---> [is], vk([Subj|SC])/Sem.
```

which is a normal chain rule.

A final compilation transforms the rules to executable PROLOG code and sorts the right hand side to achieve a proper semantics information flow. Suppose that, in the following nonchain rule the first argument of a category is its semantics argument.

```
node(a(f(Sem)),S,S0) --->
     node(b(BSem),S,S1),
     node(c(CSem,BSem),S1,S2),
     node(d(Sem,CSem),S2,S0).
```

The righthand side has to be ordered in such a way that all semantics arguments have a chance to be instantiated when the corresponding category is expanded, as in the following rule:

```
node(a(f(Sem)),S,S0) --->
     node(d(Sem,CSem),S2,S0),
     node(c(CSem,BSem),S1,S2),
     node(b(BSem),S,S1).
```

This ordering is achieved by a bubble-sort like mechanism. Elements of the right-hand-side are sorted into the new right-hand-side from right to left. To insert a new element e_{new} into an (already sorted) list $e_1 \ldots e_i$, e_{new} is inserted into $e_1 \ldots e_{i-1}$ if the semantics argument of e_{new} is not equal to some argument of e_i, otherwise it is sorted after e_i.

In the final PROLOG code nonchain rules are indexed by the functor of their lefthand side's semantics as in the following example.

```
f(f(Sem),Exp) :-
     generate(Sem,node(d(Sem,CSem),
                       S2,S0)),
     generate(CSem,node(c(CSem,BSem),
                        S1,S2)),
     generate(BSem,node(b(BSem),S,S1)),
     a(node(a(f(Sem)),S,S0),Exp).
```

Chain rules (like e.g. the one that results by replacing node(a(f(Sem)),S,S0) by node(a(Sem),S,S0) in the rule above) are indexed by their syntactic category:

```
d(node(d(Sem),S2,S0),Exp) :-
     link(a(Sem),Exp),
     generate(CSem,node(c(CSem,BSem),
```

```
                    S1,S2)),
    generate(BSem,node(b(BSem),S,S1)),
    a(node(a(Sem),S,S0),Exp).
```

The auxiliary predicates needed for the generator then can be reduced to bottom-up termination rules `C(X,X)` for all syntactic category symbols C and the predicate for `generate/2`:

```
generate(Sem,Exp) :-
    functor(Sem,F,A),
    functor(Goal,F,2),
    arg(1,Goal,Sem),
    arg(2,Goal,Exp),
    call(Goal).
```

4 CONCLUSION

We have distinguished three levels of reversibility: runtime reversibility, interpretation reversibility and compilation reversibility. We then have presented Trace & Unification Grammar, a grammar formalism that tries to bridge the gap between UG and GB theory in an undogmatic way and have presented a parser generator and a generator generator that lead to effient runtime code of the grammar both for parsing and for generation. No special effort has been invested to optimize the compilers themselves, so the compilation takes about 1.5 secs. per rule or lexicon entry. Due to space limitations many details of the compilation phase could not be discussed.

The presented grammar formalism has been used to describe a relevant subset of German language and a smaller subset of Chinese. The grammars describe a mapping between German and Chinese and QLF expressions.

ACKNOWLEDGEMENTS

I would like to thank Ms. Ping Peng and my collegues Manfred Gehrke, Rudi Hunze, Steffi Schachtl and Ludwig Schmid for many discussions on the TUG-formalism.

REFERENCES

Aho, A.V., R. Sethi and J.D. Ullman (1986) *Compilers. Principles, Techniques and Tools.* Addison-Wesley Publishing Company.

Alshawi, H. (1990) "Resolving Quasi Logical Forms", *Computational Linguistics*, Vol. 16, pp. 133-144.

Block, H. U. (forthcoming) "Two optimizations for Semantic-Head-Driven Generators".

Block, H. U. and L. A. Schmid (forthcoming) "Using Disjunctive Constraints in a Bottom-Up Parser".

Chen, H.-H., I-P. Lin and C.-P. Wu (1988) "A new design of Prolog-based bottom-up Parsing System with Government-Binding Theory", *Proc. 12th International Conference on Computational Linguistics* (COLING-88), pp. 112-116.

Chen, H.-H. (1990) "A Logic-Based Government-Binding Parser for Mandarin Chinese", *Proc. 13th International Conference on Computational Linguistics* (COLING-90), pp. 1-6.

Neumann, G. (1991) "A Bidirectional Model for Natural Language Processing" *Proc. 5th Conf. of the European Chapter of the ACL* (EACL-91) pp. 245-250.

Pereira, F. (1981) "Extraposition Grammar" *Computational Linguistics* Vol. 7, pp. 243-256.

Shieber, S.M. (1984) "The design of a Computer Language for Linguistic Information" *Proc. 10th International Conference on Computational Linguistics* (COLING-84), pp. 362-366.

Shieber, S.M. (1988) "A Uniform Architecture for Parsing and Generation", *Proc. 12th International Conference on Computational Linguistics* (COLING-88), pp. 614-619.

Shieber, S.M., G. van Noord, F.C.N. Pereira and R.C. Moore (1990). "Semantic-Head-Driven Generation". *Computational Linguistics*, Vol. 16, pp. 30-43.

Strzalkowsi, T. and Ping Peng (1990). "Automated Inversion of Logic Grammars for Generation" *Proc. Conf. of the 28th Annual Meeting of the ACL*, (ACL-90) pp. 212-219.

Strzalkowsiki, T. (1990). "How to Invert a Natural Language Parser into an Efficient Generator: An Algorithm for Logic Grammars" *Proc. 13th International Conference on Computational Linguistics* (COLING-90), pp. 347-352.

Tomita, M. (1986). *Efficient Parsing for Natural Language: A fast Algorithm for Practical Systems.* Boston: Kluwer Academic Publishers.