The TIGER language.

A Description Language for Syntax Graphs.

Part 1: User's Guidelines

Esther König and Wolfgang Lezius

IMS, University of Stuttgart, Germany
www.ims.uni-stuttgart.de/projekte/TIGER

April 3, 2002

Contents

Chapter 1

Introduction

This paper introduces a decription language for *syntax graphs*, called the TIGER language. Syntax graphs are close relatives to (syntax) trees, to feature structures or dependency graphs. Syntax graphs are syntax trees with two additions: Edges may carry labels, and crossing edges are permitted. On the other hand, syntax graphs differ from feature structures due to the following two properties: The terminal nodes are ordered (like in syntax trees). Albeit edges may cross, two edges must not join in a common node. This means that the structure sharing mechanism of feature structures is ruled out (at least in the kernel of the TIGER language).

Syntax graphs are a formalization of the essence of the so-called NEGRA data structure, which has been used to encode a syntactically annotated corpus for German [17]. Whereas the NEGRA data structure allows only for the description of a (syntactically annotated) corpus, the TIGER language includes wildcards and Boolean operators in order to allow not only for the definition but also for querying of a corpus.

We call the TIGER language a 'description language' since we want to emphasize that, in principle, the language serves to represent *corpus annotations*, as well as *corpus queries*. In the current set-up of the TIGER system, for corpus annotation only a proper sublanguage of the description language may be used[1], whereas, in queries, one can make unrestricted use of all (implemented) features of the query language.

The TIGER language consists of the following descriptive layers of graph descriptions:

1. level of *node descriptions*
   (a) *feature-value pairs*
   (b) *feature constraints*, i.e. Boolean expressions over feature-value pairs
   By adding a node identifier to a feature constraint, we get the whole node description.
2. level of *graph descriptions*

[1] For technical reasons, the corpus annotation has to be encoded in the XML-counterpart of the corpus definition sublanguage [12].

(a) *node relations*
Dominance and precedence relations among nodes.

(b) *graph constraints* or *graph descriptions*
Logical expressions over node relations.

In an orthogonal dimension, there are the following design decisions to be taken:

1. Which *logical operators* (disjunction, negation, logical variables, quantifiers, ...) should be added at which level? What will be the implications on computational complexity?

2. *Type definitions* would help to structure the universe of syntax graphs. Type definitions include the declaration of features with domain and range types and the definition of type hierarchies.

3. A means for macro or *template* definitions should be included in order to keep complex queries tractable for the human reader.

In the subsequent chapters, we introduce the TIGER language in an informal manner, i.e. by the way of examples. A formal definition of the TIGER language is given in a separate document [11]. In the appendix, the reader will find a style guide (Appendix A) with recommendations how to use the language in an optimal manner and a 'reference card' of TIGER reserved symbols (Appendix A.4.1).

1.1 Acknowledgements and References

The design and the formal definition of the TIGER language has been heavily influenced by work on tree description languages, cf. [7], [13], [14], [1], work on unification-based formalisms, cf. [5], [4], [6], [9], [8], [10], [15], and work on corpus query languages, cf. [2], [3], and literature cited in these publications.

We want to thank our colleagues in the TIGER and DEREKO projects for their creative input and their patience in discussing the details of our proposal. This work has been partially funded by the DFG (Deutsche Forschungsgemeinschaft).

Chapter 2

Node Descriptions

The TIGER language admits feature-based node descriptions: Boolean expressions over feature records (flat feature structures).

2.1 Feature-Value Pairs

For the sake of computational simplicity, we do not admit the nested feature structures. The use of feature structures is restricted to so-called *feature records*. Feature values must be constants or more precisely, strings. *Strings* are to be marked by quotation marks `"..."`. Characters which are TIGER reserved symbols (and maybe some others) must be preceded by the \-symbol. TIGER reserved symbols are listed in Appendix A.4.1.

Examples for simple feature-value pairs:

```
case= "acc"
```
(2.1)

```
word= "\."
```
(2.2)

However, the *description* of a feature value may be a non-constant, i.e. a Boolean expression over constants and constant-denoting types, or a regular expression.

The Boolean operators are `!` (negation), `&` (conjunction), and `|` (disjunction). Here is a use of a Boolean expression as a feature value description:

```
case= ("dat" | "acc")
```
(2.3)

Note that Boolean expressions for feature values must be put always into parentheses, even if only negation is involved:

$$\texttt{case= (!"dat")} \tag{2.4}$$

(2.4) can be also written as

$$\texttt{case != "dat"} \tag{2.5}$$

Similarly, constant-denoting types can be used, see Sect. 4.1.

Regular expressions have to be enclosed by slashes `/`. The syntax of regular expressions in TIGER is compatible with the syntax of regular expressions in Perl 5.003, cf. [19]. Note, however, that in our notation e.g. `/x/` means `/^x$/` in the Perl notation. In our implementation, the following operators are available:

`.`	unspecified character
`*`	unrestricted repetition
`+`	repetition with minimum 1
`[...]`	character set
`[^...]`	negated character sets
`(...)`	grouping
`|`	disjunction

Examples:

$$\texttt{word = /spiel.*/} \tag{2.6}$$

"Find words which start with '`spiel`'."

$$\texttt{lemma = /[dD](as|er)/} \tag{2.7}$$

"Find the words '`das`' and '`der`', irrespective of capitalization of the first letter."

$$\texttt{word = /.+([0-9A-Z])+.*/} \tag{2.8}$$

"Find words (e.g. abbreviations, product names) which contain at least an uppercase letter or a figure at a non-initial position."

Note the difference between e.g. `.` and `\.` in the context of regular expressions. Example (2.9) denotes all strings starting with the prefix `trag`, whereas (2.10) means all strings with the prefix `trag` followed by an arbitrary number of full stops.

`word = /trag.*/` (2.9)

`word = /trag\.*/` (2.10)

2.2 Feature Constraints

Feature records are described by feature constraints, Boolean expressions over feature-value pairs. Examples:

`[ word="das" & pos="ART" ]` (2.11)

`[ word = /sp.*/ | pos = "VVFIN" ]` (2.12)

`[ word="das" & !(pos="ART") ]` (2.13)

Basically, (2.13) is equivalent to (2.14) or (2.15). Note that this kind of equivalence is not generally valid in a typed system (cf. Chapt. 4)!

`[ word="das" & pos != "ART" ]` (2.14)

```
[ word="das" & pos = (!"ART") ]
```
(2.15)

The *unspecific constraint* is written as `[ ]`.

2.3 Node Descriptions

A *node description*, or simply a *node*, is a pair of a node identifier and a feature constraint. Example: A node with a constant node identifier `"id_n1"`:

```
"id_n1":[ word="das" & pos="ART" ]
```
(2.16)

Alternatively, one can use a *node variable* `#n1`:

```
#n1:[ word="das" & pos="ART" ]
```
(2.17)

In queries, the variable or identifier can be completely omitted if it does not occur elsewhere in the query. I.e. a plain feature constraint counts as a node description with an unspecified node identifier:

```
[ word="das" & pos="ART" ]
```
(2.18)

On the other hand, a node variable (or identifier) by itself counts as a node description:

```
#n1
```
(2.19)

which means

```
#n1:[ ]
```
(2.20)

From a linguistic point of view, it is important that nodes can be distinguished in some way. For example, the two occurrences of the noun phrase ``the man'' in the sentence ``the man sees the man'' are distinct nodes even if they carry the same feature constraints.

Chapter 3

Graph Descriptions

For the sake of convenience, we admit node descriptions as graph descriptions, e.g. (2.17) can also be considered as a graph description - describing those graphs which contain a node which fits the given feature constraint. But, now, let's turn to more complex kinds of graph descriptions.

3.1 Basic Node Relations

Since syntax graphs are two-dimensional objects, we need two operators to express the spatial relations in each dimension. We simply take the existing operators from linguistics: dominance and precedence for the vertical and the horizontal dimension, respectively.

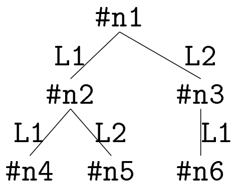
3.1.1 Labeled Direct Dominance

The symbol `>l` is the labeled direct dominance operator which serves to encode *labeled edges*. Edge labels must be strings. The constraint

```
#n1 >head #n2
```
(3.1)

means that an edge which is labeled `head` leads from node `#n1` to node `#n2`. As a more comprehensive example, the (labeled) dominance constraints in (3.2) encode the vertical dimension of the tree (3.3).

```
#n1 >L1 #n2      #n1 >L2 #n3
#n2 >L1 #n4      #n2 >L2 #n5
#n3 >L1 #n6
```
(3.2)



(3.3)

3.1.2 Direct Precedence

The notion of precedence refers to the terminal nodes in a tree. Precedence is represented by the `.`-operator (the concatenation operator in certain programming languages). The horizontal dimension of the tree (3.3) is represented by the two precedence constraints:

```
#n4 . #n5
#n5 . #n6
```

(3.4)

3.2 Derived Node Relations

Labeled direct dominance and precedence relations are good enough to encode fixed syntax graphs, e.g. the corpus annotation. In queries, one wants to refer to larger, indeterminate portions of a graph. Therefore some generalized notions of dominance and precedence ('wildcards') are necessary. Furthermore, certain convenient abbreviations should be introduced like a sibling operator.

3.2.1 Dominance

Dominance in general means that there is a *path* from one node to another one via a connected series of direct dominance relations. The *distance* of a dominance relation is the length of the path between the two nodes, i.e. the number of direct dominance relations to be traversed.

We introduce the following generalizations for the labeled direct dominance relation `>l`.

`>`	unlabeled direct dominance
`>*`	dominance (minimum distance 1)
`>n`	dominance with distance n ($n > 0$)
`>m,n`	dominance with distance between m and n
`>@l`	leftmost terminal successor ('left corner')
`>@r`	rightmost terminal successor ('right corner')
`!>L`	negated labeled direct dominance
`!>`	negated unlabeled direct dominance
`!>*`	negated dominance
`!>n`	negated dominance, distance n
`!>m,n`	negated dominance, distance $m \ldots n$
`!>@l`	negated left corner
`!>@r`	negated right corner

For unlabeled edges, the label is considered to be unspecified.

The notions of leftmost and rightmost terminal successor are required for the definition of precedence of inner nodes. The leftmost resp. rightmost successor of a leaf node is the node itself.

For the tree in (3.3), the following relations hold:

```
#n1 >* #n4        #n1 >2 #n4        #n1 >@l #n4
```
(3.5)

3.2.2 Precedence

For the precedence relation, there are the derived operators listed below. The *distance* n between two non-identical terminal nodes is the number of terminal nodes between these two nodes increased by 1. If the two terminal nodes under consideration are identical, the distance is 0.

`.*`	precedence (minimum distance 1)
`.n`	precedence with distance n ($n > 0$)
`.m,n`	precedence with distance between m and n ($0 < m < n$)
`!.*`	negated precedence
`!.n`	negated precedence, distance n
`!.m,n`	negated precedence, distance $m \ldots n$

For example, in the tree (3.3), the following precedence constraint applies:

```
#n4 .* #n6
```
(3.6)

So far, we have only talked about the precedence of terminal nodes (leaves). There are at least two alternatives to define the precedence of inner nodes:

Proper Precedence

A node `#n1` is said to precede another node `#n2` if the right corner of `#n1` precedes the left corner of `#n2`.

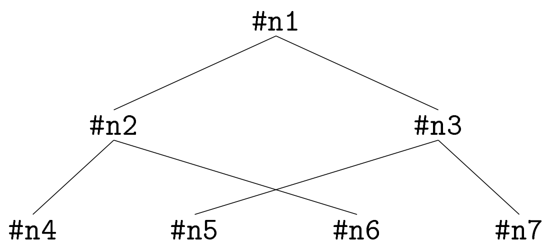
Left Corner Precedence

A node `#n1` is said to precede another node `#n2` if the left corner of `#n1` precedes the left corner of `#n2`.

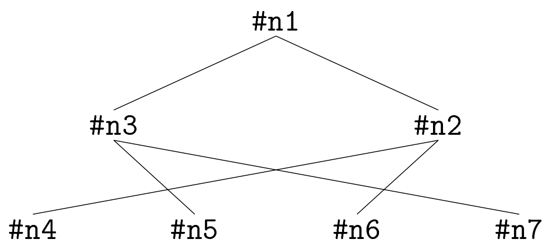
In the case of trees, both definitions coincide. In the example (3.3), for either alternative, it holds that

`#n2 . #n3` (3.7)

However, there are syntax graphs where the two alternative definitions of precedence yield different results. This is the case when the leaves of two nodes overlap, i.e. in the case of crossing edges. Consider the two different layouts (3.8) and (3.9) of the same graph.



(3.8)



(3.9)

In the case of Proper Precedence, the precedence relation between `#n2` and `#n3` is undefined. However, from to the definition of Left Corner Precedence, it follows that `#n2.#n3`. I.e. if based on Left Corner Precedence, the precedence relation among inner nodes is a total order in the sense that for any two nodes `#n`, `#n'` which do not have the same left corner, either `#n` precedes `#n'` or vice versa. For this reason, we choose Left Corner Precedence as the underlying notion for the precedence of inner nodes.

3.2.3 Sibling Relation

Two nodes `#n1` and `#n2` are *siblings* if they are directly dominated by the same node `#n0`, i.e. if they have the same mother node `#n0`. We use the following writing conventions:

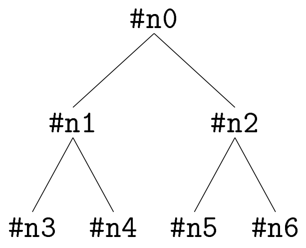
`$`	siblings
`$.*`	siblings with precedence
`!$`	negated siblings

We did not include the negated '`$.*`'-relation since it seems rather counter-intuitive.

For example, in tree (3.3), the following sibling relations hold:

`#n2 $ #n3`   `#n4 $ #n5`   (3.10)

One might wish to state that two inner nodes, e.g. the nodes `#n1` and `#n2` in (3.11), are 'siblings with *direct* precedence'.



(3.11)

In this situation, it seems to be more intuitive to compare the right corner `#n4` of node `#n1` with the left corner `#n5` of node `#n2`, cf. (3.12).

`#n1 >@r #n4`   `#n2 >@l #n5`   `#n4 . #n5`   (3.12)

For this reason, the relations `>@l` and `>@r` have been introduced, so that the user can formulate this constraint when required. The 'built-in' precedence relation which relies on the comparison of leftcorners would lead to the counter-intuitive result that `#n1` does not directly precede `#n2`.

3.3 Secondary edges

Although, in the kernel of the TIGER language, a node must not be dominated by more than one node, for reasons of commodity, we have added an implementation of so-called secondary edges. A secondary edge defines an additional parent node for a given node:

`>~l`	labeled secondary edge
`>~`	secondary edge
`!>~l`	negated labeled secondary edge
`!>~`	negated secondary edge

3.4 Graph Descriptions

Graph descriptions or *graph constraints* are (restricted) Boolean expressions over node relations (and node descriptions). Currently, conjunction `&` and disjunction `|` are available as logical connectives. For example, the set of node relations in (3.2) (which describes the tree (3.3)) is written officially as a graph constraint by using the `&`-operator:

`(#n1>#n2) & (#n1>#n3) & (#n2>#n4) & (#n2>#n5) & (#n3>#n6)` (3.13)

or simpler, by omitting parentheses in an obvious way:

`#n1>#n2 & #n1>#n3 & #n2>#n4 & #n2>#n5 & #n3>#n6` (3.14)

Remark

A *negation operator* has not been included since its (computational) implications in the context of underspecified syntax graphs are unclear to us. Note that underspecified syntax graphs correspond to higher-order terms, i.e.

`n1 >* n2`

$\equiv \lambda P\,.\,\mathtt{n1}(P\ \mathtt{n2}))$.

3.5 Graph Predicates

3.5.1 Root

The root of a graph (for a whole sentence) can be identified by the predicate `root`:

`root(#n1)` (3.15)

3.5.2 Arity Predicates

Note that example (3.16) describes all graphs which contain a certain node `#n1` with *at least* two daughters `#n2` and `#n3`.

```
#n1>#n2 & #n1>#n3
```
(3.16)

However, one would like to state that there must be *exactly* two daughters of a certain kind. For this reason, we introduce a two-place operator `arity` in order to be able to restrict the number of daughters of a node `#n`, e.g. to two daughters:

```
#n1>#n2 & #n1>#n3 & arity(#n1,2)
```
(3.17)

The `arity`-operator can also come with three arguments where it indicates an interval of numbers of daughters, e.g. two or three or four daughters:

```
#n1>#n2 & #n1>#n3 & arity(#n1,2,4)
```
(3.18)

There is also an `tokenarity`-operator which constrains the number of terminal nodes (leaves) which are dominated by a node. For example,

```
tokenarity(#n1,5)
```
(3.19)

means that node `#n1` must dominate exactly 5 terminal nodes.

```
tokenarity(#n1,5,7)
```
(3.20)

constrains the node `#n1` to dominate at least 5 terminal nodes and at most 7 terminal nodes.

3.5.3 Continuity Predicates

It maybe useful to state that the leaves which are dominated by a node must form a continuous string or not. For this purpose, the two unary operators `continuous` and `discontinuous` have been introduced. Examples:

```
continuous(#n1)
```
(3.21)

```
discontinuous(#n1)
```
(3.22)

Chapter 4

Type Definitions

If the user has to declare the symbols which will be used in a corpus and in queries, e.g. inconsistencies in the corpus annotation, can be detected much more easily. TIGER allows for the declaration of type hierarchies and features.

There are the following built-in types in the TIGER language:

- `String` for not listable feature values like the values of the features `word` and `lemma`
- `UserDefConstant` for user-defined ('listable') feature values
- `Constant` comprises both, `String` and `UserDefConstant`
- `NT` for feature constraints of nonterminal nodes
- `T` for feature constraints of terminal nodes
- `FREC` stands for all feature records (feature constraints), i.e. `NT` and `T`
- `Node` stands for node descriptions
- `Graph` means all graphs
- $\top$ means anything (in the world of syntax graphs)

The hierarchy of built-in types is visualized in Fig. 4.1. The expressive means for defining a type hierarchy are introduced below.

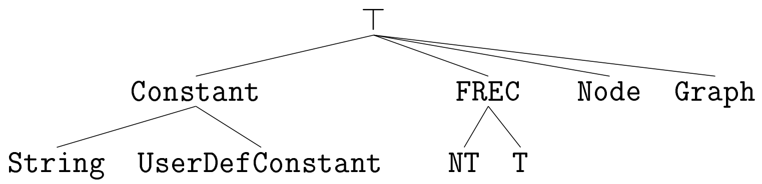


Figure 4.1: Hierarchy of TIGER built-in types

The built-in types `Node` and `Graph` are only required on the conceptual level. In the current implementation, they cannot be referred to in the description language.

4.1 Definition of a type hierarchy

As a TIGER built-in, the types `NT` and `T` are subtypes of the type `FREC`. This is expressed by the following *type definition* - which must not be overwritten by the user.

```
FREC := NT, T;
```

The lefthand side of a type definition is the type `t` which is being defined. The types on the righthand side of a type definition are the (direct) *subtypes* of the type `t`. An occurrence of a type `t'` on the righthand side of a definition is called a *use* of `t'`.

In the current implementation, the user can only add type definitions for feature values, i.e. for the type `UserDefConstant`. For example, the user may want to define the types `Case`, `GrammarSymbol`, and `Person`. The type `GrammarSymbol` is to be the disjoint set union of a type `InnerCat` for inner nodes and a category type `TerminalCat` for terminal nodes, etc.

```
UserDefConstant := Case, GrammarSymbol, Person;

GrammarSymbol := InnerCat, TerminalCat;
TerminalCat := adjectivePOS, adverbPOS, nounPOS, particlePOS, verbPOS;
```

In this way, *type hierarchies* can be defined. The 'terminal nodes' of a type hierarchy (of constant denoting types) can be constants:

```
InnerCat := "AA","AP","AVP","CAC","CAP","CAVP","CCP","CH",
            "CNP","CO","CPP","CS","CVP","CVZ","DL","ISU",
            "NPN","MTA","NM","NP","PP","QL","S","VP","VZ";

nounPOS := "NE","NN";
verbPOS := auxVerbPOS, fullVerbPOS, modalVerbPOS;

auxVerbPOS := "VAFIN","VAIMP","VAINF","VAPP";
fullVerbPOS := "VVFIN","VVIMP","VVINF","VVIZU","VVPP";
modalVerbPOS := "VMFIN","VMINF","VMPP";
```

On the basis of these type definitions, some disjunctions of feature values can be written in a more concise manner, e.g. (4.1) can now be replaced by (4.2).

`[ pos = ("NE" | "NN") ]` (4.1)

```
[ pos = nounPOS ]                                    (4.2)
```

Restrictions for type definitions

1. A type must be defined at most once.
2. A type must be used exactly once in a type definition.

The first restriction means that neither recursion nor cross-classification (alternative definitions of the same type symbol) can be expressed. If you think you need cross-classification, template definitions (see Sect. 9) might be a way out. The second restriction enforces that every type must be hooked up in the *type hierarchy*. In total this means that type definitions define a tree-shaped type hierarchy. Undefined types may only occur as the leaves of a type hierarchy. For example, the built-in types form the hierarchy shown in Fig. 4.1.

4.2 Feature Declarations

A feature declaration states the following information:

- It declares the symbol under consideration as a feature name.
- It restricts the *domain* of the feature, i.e. it states for which type the feature may be used.
- It restricts the feature values (i.e. the *range*).

In the current implementation, features can only be declared for the built-in types `"NT"` and `"T"`. Furthermore, one cannot use a type to restrict the range of a feature, but the possible values for a feature have to be enumerated. The reason is that we want to keep the number of dependencies between corpus definition and type definitions as small as possible. In addition, enumerations of feature values can be constructed automatically, in the case that a corpus does not come with feature declarations.

If the value enumeration is omitted from a feature declaration, the default range of that feature is `String`.

For the type `T` of feature constraints for terminal nodes, the features **`word`**, **`lemma`**, and **`pos`** may be defined as follows.

```
<feature name="word" domain="T" />
<feature name="lemma" domain="T" />
<feature name="pos" domain="T">
  ...
  <value name="VAFIN">...comment...</value>
  <value name="VAIMP"></value>
  <value name="VAINF"></value>
  <value name="VAPP"></value>
  <value name="VMFIN"></value>
  <value name="VMINF"></value>
  ...
</feature>
```

Remark Each feature must be declared exactly once.

The exclusion of multiple declarations for the same feature means that polymorphic overloading a feature symbol is not permitted.

Please note that, the TIGER description language being a typed language, (4.3) is not equivalent to (4.4)!

`[ word="das" & !(pos="ART") ]` (4.3)

`[ word="das" & pos != "ART" ]` (4.4)

The reason is that `!(pos="ART")` equals `!(T & pos="ART")` due to the corresponding feature declaration. The latter formula again is equivalent to `!(T) | (pos != "ART")`, i.e. either the feature `pos` is not defined on a type or it is defined and its value is not equal to `"ART"`.

Chapter 5

Corpus Definition

A *corpus definition*, i.e. an annotated corpus, consists of a list of restricted graph descriptions (*corpus graphs*). The restrictions are as follows:

Declaration of required features

A corpus definition must include the (possibly automatically derived) feature declarations for the type `NT` of nonterminal constraints and the type `T` of terminal constraints.

Single Root Node; Connectedness; No Structure Sharing

Every node except for one distinguished node (*root node*) is directly dominated by exactly one node.

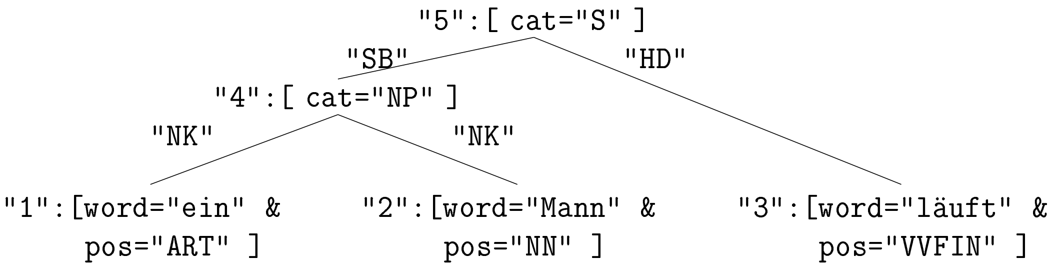
Acyclicity

No node may (indirectly) dominate itself.

Full Disambiguation

- The graph constraints may only include the basic node relations labeled direct dominance (`>l`) and direct precedence (`.`).
- The feature constraint for a node must be a conjunction of feature-value pairs either for all features which have been declared for `NT` or for those which have been defined for `T`.
- For each node, its arity (i.e. the number of its daughters) has to be fixed.
- The precedence relation on terminal nodes (leaf nodes) must be a total order.
- The only logical connective on all structural levels is the conjunction operator `&`.
- Neither types (Sect. 4) nor templates (Sect. 9) nor regular expressions are admitted.

In the terms of the TIGER description language, one can think of a corpus graph (5.1) as being encoded like in (5.2). However, for easier corpus preparation, the actual corpus encoding is XML-based [12].

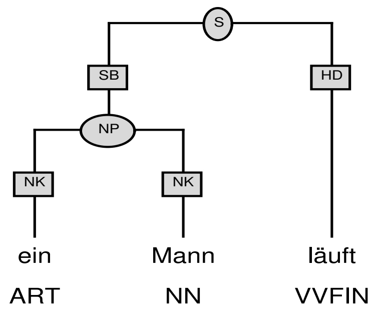


(5.1)

```
root("5") &
"1":[word="ein" & pos="ART"] &
"2":[word="Mann" & pos="NN"] &
"3":[word="läuft" & pos="VVFIN"] &
"4":[cat="NP"] &
arity("4",2) &
"5":[cat="S"] &
arity("5",2) &
"1" . "2" & "2" . "3" &
("5" >"SB" "4")  &  ("5" >"HD" "3") &
("4" >"NK" "1")  &  ("4" >"NK" "2");
```

(5.2)

In the NEGRA-style layout, the above tree corresponds to (5.3).



(5.3)

Remark

The so-called 'secondary edges' of the NEGRA format have been excluded from the *kernel of TIGER language* for reasons of efficiency. However, secondary edges are treated in an additional, less developed layer of the TIGER language.

Chapter 6

Queries and Query Results

A *query* can be any kind of graph description.[1] A *query result* is the set of graphs in the corpus which contain a subgraph which matches the query. For example, if a corpus has been annotated sentence-wise, the graphs in the result will be the graphs of individual sentences. Here is an example. Let's assume that (5.1) is the only syntax graph in our corpus. So, of course, (5.1) will be retrieved as the answer to a query like

```
[ cat="NP" ]>[ word="Mann" ] ;
```
(6.1)

> "Find a corpus graph which contains a subgraph with the following properties:
> Its root node is labeled `"NP"` and one of its daughters is the word `‘‘Mann’’`."

Since queries are restricted to be graph (or node) descriptions, there is a minor inconvenience if you want to express a query which restricts only a feature-value pair or a feature value. For example, if you are looking for all graph with some node which carries the value `"acc"` for the `case`-feature, you will have to write down at least a node description, i.e.

```
[ case="acc" ] ;
```
(6.2)

If you write only `case="acc"` or `"acc"`, this will be rejected as an illegal query syntax.

[1]There is one minor restriction: Constant node identifiers don't make much sense in a query.

Chapter 7

Logical Variables

There are (at least) two ways to use variables:

- *linear use of a variable* (wildcards, “don’t care”)

  If a variable occurs only once in an expression, this means that the value it stands for is uninteresting. For example, the dominance operator without an edge label `>` is actually a dominance operator with a unique edge label variable, say `#L`:

  `>#L.`

- *non-linear use of a variable* (coreference, structure sharing)

  If a variable occurs more than once in an expression, this will enforce the identity of the substructures which are labeled by this variable.

One might think of admitting explicit (hence potentially non-linear) variables for the following types of TIGER expressions:

1. variables for *feature values*
2. variables for *feature constraints*
3. variables for *node identifiers* (i.e. for nodes and the subgraphs rooted in those nodes)
4. variables for *edge labels*

However, since certain types of variables will increase the expressivity (and hence the computational complexity) of the TIGER language drastically, we will not make available all of the above options.

7.1 Variables for Feature Values

Example: agreement of morphosyntactic information

```
[ cat="ART" & case=#Case ] $ [ cat="NN" & case=#Case ]
```
(7.1)

Variables (for feature values) make more sense if they can come with a complex constraint:

```
[ cat="ART" & case= #Case:( "dat" | "acc" ) ]
$ [ cat="NN" & case=#Case ]
```
(7.2)

7.2 Variables for Feature Constraints

Example: Search for a sentence with two nouns which have the same properties.

```
[ #F:cat="NP" ] .* [ #F ]
```
(7.3)

or

```
[ #F:(cat="NP" & case=("nom"|"acc")) ] .* [ #F ]
```
(7.4)

The query 7.3 resp. 7.4 will match e.g. the sentence

“`Die Mutter und die Tochter gehen spazieren.`”

because the noun phrases “`die Mutter`” and “`die Tochter`” both are nominative. However, the sentence

“`Der Vater wickelt seinen Sohn.`”

will not be retrieved by this query because the two nouns “`Vater`” and “`Sohn`” have different case (nominative vs. accusative). The different uses of the same feature constraint variable `#F` in the query mean that the (retrieved) feature constraints in all these places must be pairwise compatible, even for those feature-value pairs which are underspecified or not mentioned in the query!

7.3 Variables for Nodes

Variables for node identifiers have been already introduced in Sect. 2.3.

7.4 Variables for Edge Labels

There are already implicit variables for edge labels, e.g. when you use the unnamed dominance operator $>$ or the dominance wildcard $> *$. If explicit variables for edge labels were available, the user could require co-reference of edge labels. To us it is unclear what kind of computational complexity will be introduced by this additional expressivity. And furthermore, there is a work-around: Since the number of edge labels (grammatical functions) tends to be rather small, it does not seem too inconvenient to enumerate the variants of a constraint wrt. to the edge lables.

Chapter 8

A Note on Quantifiers

Note that the variables which have been introduced in Sect. 7, are existentially bound at the outermost level of a TIGER expression.

Illustration:

You might be tempted to state the following expression to find trees which are rooted in a `VP`, but which do *not* contain *any* `NP`:

$$\texttt{[cat="VP"]} > * \texttt{ [cat=(!"NP")]} \tag{8.1}$$

But in the TIGER language, (8.1) has the interpretation: "Find a graph rooted in a `VP` with *at least one* inner node which is *not* labeled `NP`", i.e.

$$\exists \texttt{\#x ([cat="VP"]} > * \texttt{ \#x:[cat=(!"NP")])} \tag{8.2}$$

The desired constraint of finding a graph which does *not* contain *any* `NP` requires universal quantification and the implication operator (i.e. negation of node relations):

$$\forall \texttt{\#x (([cat="VP"]} > * \texttt{ \#x)} \Rightarrow \texttt{(\#x:[cat=(!"NP")]))} \tag{8.3}$$

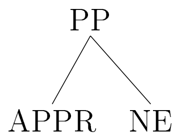
The use of the universal quantifier causes computational overhead in the sense that the concerned constrained has to be *copied* and checked for each descendent of a `VP`-node in each potentially matching graph in the corpus. For the sake of computational simplicity and tractability, the universal quantifier is (currently) not part of the TIGER language.

Chapter 9

Template Definitions (not yet implemented)

When working with a syntactic representation formalism like the TIGER language, certain pieces of code will be used over and over again. Furthermore, it is good programming and grammar writing style to define generalizations. Therefore, there is an urgent need for a means for defining abbreviations, templates (cf. [16]), or macros. We chose TIGER templates to be *non-recursive logical relations*. This means that, although template calls may be embedded into template definitions, there must be neither direct nor indirect self-reference in template definitions. Hence, templates realize the 'database programming' part of a logic programming language like Prolog (cf. [18]).

A simple scheme `PrepPhrase` of a prepositional phrase which consists only of a preposition (`APPR`) and a proper noun (`NE`)



(9.1)

can be defined as follows:

```
PrepPhrase(#n0) <-
   #n0:[cat="PP"] > #n1:[pos="APPR"]
   & #n0 > #n2:[pos="NE"]
   & #n1.#n2
   & arity(#n0,2) ;
```

The arrow `<-` marks a *defining clause* of a template definition. The `<-` operator is a two-place operator which takes a *template head* on its lefthand side and a *template body* on its righthand side. The template head consists of the *template name* (e.g. `PrepPhrase`) and of a list of one or more variables, the *argument parameters* of the template clause. The template body consists of a graph description. The end of a defining clause is signaled by the ;-symbol. A template definition can consist of several defining clauses. If a variable #x is not assigned any constraint in the body of the template definition, i.e. if it cannot be decided which one of the built-in types Constant, FRec, Node, or Graph, the variable #x belongs to, the variable is assumed to be a Graph variable.

The PrepPhrase template can now be *used* or *called* in a query, e.g.

```
#n0 & PrepPhrase(#n0) ;                                   (9.2)
```

The query (9.2) will return all graphs from the given corpus, which contain a subtree (rooted in #n0) with the desired shape.

Warning: The '#n0 &' part is important. A template call by itself corresponds to the empty graph description, i.e. it admits all possible graphs, the whole corpus.

The template PrepPhrase can be used also in the body of some other template definition, e.g. in the definition of a pattern for VerbPhrase's like "geht nach Stuttgart".

```
VerbPhrase(#n0) <-
   #n0:[cat="VP"] > #n1:[pos="VVFIN"]
   & #n0 > #n2
   & #n1.#n2
   & arity(#n0,2)
   & PrepPhrase(#n2) ;
```

Usually, a graph description corresponds to a whole graph from the corpus. However, a template call should rather give access to specific subgraphs. For this reason, when a variable, say #n0 is used as an argument of a template, it denotes only the subgraph (or substructure) which is 'rooted' in #n0.

Note that the scope of a variable is the defining clause it occurs. E.g. the variable #n2 in the definition of PrepPhrase is distinct from the variable #n2 in the definition of VerbPhrase. The TIGER language interpreter will *resolve* the call to PrepPhrase in the following manner, i.e.

1. it will look up the defining clause of PrepPhrase
2. replace all the variable names by new ones (in order to avoid unintended confusion of variables from the 'calling' clause and the 'called' clause)
3. identify the node id variable #n2 in VerbPhrase with the argument #n0 of PrepPhrase's defining clause. Identification of variables means that the constraints for both variables are joined into a single constraint by a logical conjunction.

After the resolution step, you may imagine that the `VerbPhrase`-clause has the following shape:

```
VerbPhrase(#n0) <-
   #n0:[cat="VP"] > #n1:[pos="VVFIN"]
   & #n0 > #n2
   & #n1.#n2
   & #n2:[cat="PP"] > #n21:[pos="APPR"]
   & #n2 > #n22:[pos="NE"]
   & #n21.#n22
   & arity(#n0,2)
   & arity(#n2,2) ;
```

If we want to find out which verbs go with which prepositions, this can be done by adding the appropriate arguments or parameters.

```
PrepPhrase(#n0,#prep) <-
   #n0:[cat="PP"] > #n1:[word=#prep & pos="APPR"]
   & #n0 > #n2:[pos="NE"]
   & #n1.#n2
   & arity(#n0,2) ;

VerbPhrase(#n0,#verblemma,#prep) <-
   #n0:[cat="VP"] > #n1:[pos="VVFIN" & lemma=#verblemma]
   & #n0 > #n2
   & #n1.#n2
   & arity(#n0,2)
   & PrepPhrase(#n2,#prep) ;
```

After a while, we might get bored of the simple notion of prepositional phrases. For that reason, a template definition may consist of several, alternative defining clauses:

```
PrepPhrase(#n0,#prep) <-
   #n0:[cat="PP"] > #n1:[word=#prep & pos="APPR"]
   & #n0 > #n2:[pos="NE"]
   & #n1.#n2
   & arity(#n0,2) ;

PrepPhrase(#n0,#prep) <-
   #n0:[cat="PP"] > #n1:[word=#prep & pos="APPR"]
   & #n0 > #n2:[pos="ART"]
   & #n0 > #n3:[pos="NN"]
   & #n1.#n2
   & #n2.#n3
   & arity(#n0,3) ;
```

The above definition is equivalent to a disjunction of the two defining clauses:

```
PrepPhrase(#n0,#prep) <-
   ( #n0:[cat="PP"] > #n1:[word=#prep & pos="APPR"]
     & #n0 > #n2:[pos="NE"]
     & #n1.#n2
     & arity(#n0,2) )
   |
   ( #n0:[cat="PP"] > #n1:[word=#prep & pos="APPR"]
     & #n0 > #n2:[pos="ART"]
     & #n0 > #n3:[pos="NN"]
     & #n1.#n2
     & #n2.#n3
     & arity(#n0,3) );
```

or, in a more packed manner:

```
// packed definition of PrepPhrase
PrepPhrase(#n0,#prep) <-
   #n0:[cat="PP"]
   & #n1:[word=#prep & pos="APPR"]
   & #n1.#n2
   & ( ( #n0 > #n1
         & #n0 > #n2:[pos="NE"]
         & arity(#n0,2) )
       |
       ( #n0 > #n1
         & #n0 > #n2:[pos="ART"]
         & #n0 > #n3:[pos="NN"]
         & #n2.#n3
         & arity(#n0,3) ) ) ;
```

The '//'-symbol marks the remainder of a line as a comment.

Appendix A

Style guide

This chapter assembles recommendations how to use the TIGER language in a way such that

- the code is more apt to be comprehensible to other readers
- computational traps can be avoided if run-time efficiency counts

In addition, we give also some hints how to handle certain pecularities of the TIGER language and its implementation.

A.1 Feature values

A.1.1 Boolean expressions

Note that the satisfiability problem for Boolean expressions is NP-complete, i.e. for the sake of efficiency, try to avoid overly complex descriptions for feature values. Use types instead.

A.1.2 Regular expressions

Regular expressions for feature values should be reserved for 'open-ended' value types like e.g. a `word` or `lemma` value. For features with a 'listed' range of a values, the use of types and Boolean expressions leads to more readable code.

Note that the TIGER language compiler performs only a rough check of the syntax of regular expressions. The fine-grained syntax check for regular expressions will be carried out when a query is evaluated. Therefore it may involve more effort for you to discover the syntax errors you have made in a regular expression.

A.2 Features Constraints

The features which occur in the corpus annotation, must have been declared in the preamble of the corpus.

When using features in a query, make sure that these features belong to the corpus you want to query.

A.3 Graph descriptions

Two unconstrained nodes `#n1` and `#n2`, may match on the same node in the corpus. If this causes problems, the inequality of the two nodes can be enforced by the following constraint

```
(#n1 .* #n2) | (#n2 .* #n1)
```

A.4 Templates

A.4.1 Templates for feature values

One might think of abbreviating a disjunction of feature values by a template name:

```
gen-dat(#case) <-
    [ case = #case: ( "gen" | "dat" ) ] ;
```

But we recommend to use a type definition instead, if possible, since this does not introduce an explicit disjunction into the structure.

```
gen-dat := "gen", "dat" ;
```

TIGER Quick Reference

Reserved symbols

All the operators and built-in types which are listed in this reference are reserved symbols and may cause problems, when used in edge labels, strings, constants, predicate and type names. In many situations, a preceding \-operator may help. Some examples are given below.

Built-in types

The sample queries for the built-in types are meant to illustrate the context where a built-in type may occur. However, these queries are not really meaningful, since they are too general.

Symbol	*Meaning*	*Sample query*
`Constant`	constants	`[word = Constant]`
`String`	strings	`[word = String]`
`UserDefConstant`	user defined constants	`[pos=UserDefConstant]`
`FREC`	feature records	`[FREC]`
`NT`	feature records for nonterminals	`[NT]`
`T`	feature records for terminals	`[T]`

Constants

`"`	constant mark	`[word = "hallo"]`
\	escape for reserved characters	`[word = "\,"]`

Regular expressions for constants

<table>
<tr><td>/</td><td>regular expression mark</td><td>[word = /hallo/]</td></tr>
<tr><td>.</td><td>unspecified character</td><td>[word = /spiel./]</td></tr>
<tr><td>*</td><td>unrestricted repetition</td><td>[lemma = /spiel.*/]</td></tr>
<tr><td>+</td><td>repetition with minimum 1</td><td>[word = /.+[0-9A-Z]+.*/]</td></tr>
<tr><td>[ ]</td><td>character set</td><td>[word = /.+[0-9A-Z]+.*/]</td></tr>
<tr><td>[^]</td><td>negated character sets</td><td>[word = /[^0-9A-Z].*/]</td></tr>
<tr><td>( )</td><td>grouping</td><td>[word = /([pbkgtd][aeiou])+/]</td></tr>
<tr><td>|</td><td>disjunction</td><td>[word = /[dD](as|er)/]</td></tr>
<tr><td>\</td><td>escape for reserved characters</td><td>[word = /.*\-.*/]</td></tr>
</table>

Feature-value pairs

=	feature-value pair	[case = "acc"]
!=	neg. feat.-value pair	[case != "acc"]

Graph predicates

root( )	root of a graph	root(#n)
arity(,) arity(, ,)	arity of a node	arity(#n,2) arity(#n,2,4)
tokenarity(,) tokenarity(, ,)	# of dominated leaves	tokenarity(#n,5) tokenarity(#n,5,8)
continuous( )	continuous leaves	continuous(#n)
discontinuous( )	discontinuous leaves	discontinuous(#n)

Dominance relation

`>l`	labeled direct dominance	`[cat="NP"] >GR [cat="NP"]` `[cat="NP"] >OA\-MOD [cat="NP"]`
`>`	direct dominance	`[cat="NP"] > [pos="NE"]`
`>*`	dominance	`[cat="NP"] >* [pos="NE"]`
`>n`	dominance, distance n	`[cat="NP"] >2 [pos="NE"]`
`>m,n`	dominance, distance $m \ldots n$	`[cat="NP"] >2,3 [pos="NE"]`
`>@l`	left corner	`[cat="NP"] >@l [word="etwas"]`
`>@r`	right corner	`[cat="NP"] >@r [word="Kohl"]`
`$`	siblings	`[word="etwas"] $ [cat="NP"]`
`$.*`	siblings with precedence	`[word="etwas"] $.* [cat="NP"]`
`!>l`	neg. labeled direct dominance	`[cat="NP"] !>GR [cat="NP"]`
`!>`	neg. direct dominance	`[cat="NP"] !> [pos="NE"]`
`!>*`	neg. dominance	`[cat="NP"] !>* [pos="NE"]`
`!>n`	neg. dominance, distance n	`[cat="NP"] !>2 [pos="NE"]`
`!>m,n`	neg. dominance, distance $m \ldots n$	`[cat="NP"] !>2,3 [pos="NE"]`
`!>@l`	neg. left corner	`[cat="NP"] !>@l [word="etwas"]`
`!>@r`	neg. right corner	`[cat="NP"] !>@r [word="Kohl"]`
`!$`	neg. siblings	`[word="etwas"] !$ [cat="NP"]`
`>~l`	labeled secondary edge	`[cat="VP"] >~HD [cat="NP"]`
`>~`	secondary edge	`[cat="VP"] >~ [cat="NP"]`
`!>~l`	neg. labeled secondary edge	`[cat="VP"] !>~HD [cat="NP"]`
`!>~`	neg. secondary edge	`[cat="VP"] !>~ [cat="NP"]`

Precedence relation

`.`	direct precedence	`[word="etwas"] . [pos=noun]`
`.*`	precedence	`[word="etwas"] .* [pos="NN"]`
`.n`	precedence, distance n	`[word="etwas"] .2 [pos="NN"]`
`.m,n`	precedence, distance $m \ldots n$	`[word="etwas"] .2,4 [pos="NN"]`
`!.`	neg. direct precedence	`[word="etwas"] !. [pos=noun]`
`!.*`	neg. precedence	`[word="etwas"] !.* [pos="NN"]`
`!.n`	neg. precedence, distance n	`[word="etwas"] !.2 [pos="NN"]`
`!.m,n`	neg. precedence, distance $m \ldots n$	`[word="etwas"] !.2,4 [pos="NN"]`

Variables

<table>
<tr><td>#</td><td>variable for a feature value</td><td><code>[case = #Case]</code></td></tr>
<tr><td>:</td><td>constrained variable</td><td><code>[case=#x:("dat"|"akk")] . [case=#x]</code></td></tr>
<tr><td>#</td><td>var. for a feat. description</td><td><code>[#x]</code></td></tr>
<tr><td>:</td><td>constrained variable</td><td><code>[#x:(case = "acc")].[#x]</code></td></tr>
<tr><td>#</td><td>variable for a node identifier</td><td><code>#n1 > #n2</code></td></tr>
<tr><td>:</td><td>node description</td><td><code>(#n:[cat="NP"]>[pos="ADJA"]) & (#n>[pos="NN"])</code></td></tr>
</table>

Boolean expressions

<table>
<tr><td>( )</td><td>bracketing</td><td><code>[pos=(!("NN" | "ART"))]</code></td></tr>
<tr><td>!</td><td>negation, feature values</td><td><code>[pos=(!"NN")]</code></td></tr>
<tr><td>&</td><td>conjunction, feature values</td><td><code>[pos=(!"NN" & !"NE")]</code></td></tr>
<tr><td>|</td><td>disjunction, feature values</td><td><code>[pos=("NN" | "NE")]</code></td></tr>
<tr><td>!</td><td>negation, feature constraints</td><td><code>[!(pos="NN")]</code></td></tr>
<tr><td>&</td><td>conjunction, feature constraints</td><td><code>[pos="NE" & word="Ina"]</code></td></tr>
<tr><td>|</td><td>disjunction, feature constraints</td><td><code>[pos="NE" | word="es"]]</code></td></tr>
<tr><td>&</td><td>conjunction, graph descriptions</td><td><code>#n1>#n2 & #n3.#n2</code></td></tr>
<tr><td>|</td><td>disjunction, graph descriptions</td><td><code>#n1>#n2 | #n1>#n3</code></td></tr>
</table>

Negation `!` must not have scope over variables.

A Boolean expression for a feature value must be put into parentheses.

Comments

//	line comment	`[pos="NE"].[pos="NE"] // 2-word proper nouns`

Note that a line comment must end with a newline or carriage return character (or both), i.e. the end-of-file mark is not enough!

Bibliography

[1] Patrick Blackburn, Claire Gardent, and Wilfried Meyer-Viol. Talking about trees. In *Proceedings of the 6th Conference of the European Chapter of the Association for Computational Linguistics*, pages 21–29, Utrecht, 1993.

[2] Oliver Christ. A modular and flexible architecture for an integrated corpus query system. In *Proceedings of COMPLEX'94: 3rd Conference on Computational Lexicography and Text Research (Budapest, July 7–10 1994)*, pages 23–32, Budapest, Hungary, 1994. CMP-LG archive id 9408005.

[3] Oliver Christ, Bruno M. Schulze, Anja Hofmann, and Esther König. *Corpus Query Processor (CQP). User's Manual.* Institut für Maschinelle Sprachverarbeitung, Universität Stuttgart, Stuttgart, Germany, 1999. `www.ims.uni-stuttgart.de/CorpusWorkbench/`.

[4] Jochen Dörre. *Feature-Logik und Semiunifikation.* Dissertationen zur Künstlichen Intelligenz. infix-Verlag, Sankt Augustin, Germany, 1996.

[5] Jochen Dörre and Michael Dorna. CUF - a formalism for linguistic knowledge representation. Deliverable R.1.2A, DYANA 2, August 1993.

[6] Jochen Dörre, Dov M. Gabbay, and Esther König. Fibred semantics for feature-based grammar logic. *Journal of Logic, Language, and Information. Special Issue on Language and Proof Theory*, 5:387–422, 1996.

[7] Denys Duchier and Joachim Niehren. Solving dominance constraints with finite set constraint programming. Technical report, Universität des Saarlandes, Programming Systems Lab, 1999.

[8] Martin Emele. *Der TFS-Repräsentationsformalismus.* PhD thesis, Institut für Maschinelle Sprachverarbeitung, Universität Stuttgart, Stuttgart, Germany, 1997. AIMS Arbeitspapiere des IMS, vol.3(6).

[9] Martin Emele and Rémi Zajac. A fixed-point semantics for feature type systems. In *Proceedings of the 2nd International Workshop on Conditional and Typed Rewriting Systems*, Montreal, Canada, 1990.

[10] Markus Höhfeld and Gert Smolka. Definite relations over constraint languages. LILOG-Report 53, IBM Deutschland, Stuttgart, Baden-Württemberg, October 1988.

[11] Esther König and Wolfgang Lezius. The TIGER language - a description language for syntax graphs. Part 2: Formal definition. Technical report, IMS, University of Stuttgart, 2001.

[12] Wolfgang Lezius and Hannes Biesinger. *TIGER XML Quick Reference Guide*. Institut für Maschinelle Sprachverarbeitung, Universität Stuttgart, Stuttgart, Baden-Württemberg, 2001.

[13] Mitchell Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 1993.

[14] James Rogers and K. Vijay-Shanker. Reasoning with descriptions of trees. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, Newark, Delaware, 1992.

[15] Helmut Schmid. *YAP: Parsing and Disambiguation With Feature-Based Grammar*. PhD thesis, Institut für Maschinelle Sprachverarbeitung, Universität Stuttgart, Stuttgart, Germany, 1999.

[16] Stuart M. Shieber. *An Introduction to Unification-Based Approaches to Grammar*. Lecture Notes. Center for the Study of Language and Information, Stanford, Ca., 1986.

[17] Wojciech Skut, Brigitte Krenn, Thorsten Brants, and Hans Uszkoreit. An annotation scheme for free word order languages. In *Proceedings of the 5th Conference on Applied Natural Language Processing (ANLP)*, Washington, D.C., March 1997.

[18] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, Cambridge, Mass., 1986.

[19] Larry Wall, Tom Christiansen, Randal L. Schwartz, and Stephen Potter. *Programming Perl*. O'Reilly, Cambrige, 1996.