TDL ExtraLight User's Guide

Hans-Ulrich Krieger, Ulrich Schäfer

June 1993
The German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI) with sites in Kaiserslautern and Saarbrücken is a non-profit organization which was founded in 1988. The shareholder companies are Atlas Elektronik, Daimler-Benz, Fraunhofer Gesellschaft, GMD, IBM, Insiders, Mannesmann-Kienzle, SEMA Group, Siemens and Siemens-Nixdorf. Research projects conducted at the DFKI are funded by the German Ministry for Research and Technology, by the shareholder companies, or by other industrial contracts.

The DFKI conducts application-oriented basic research in the field of artificial intelligence and other related subfields of computer science. The overall goal is to construct systems with technical knowledge and common sense which - by using AI methods - implement a problem solution for a selected application area. Currently, there are the following research areas at the DFKI:

- Intelligent Engineering Systems
- Intelligent User Interfaces
- Computer Linguistics
- Programming Systems
- Deduction and Multiagent Systems
- Document Analysis and Office Automation.

The DFKI strives at making its research results available to the scientific community. There exist many contacts to domestic and foreign research institutions, both in academy and industry. The DFKI hosts technology transfer workshops for shareholders and other interested groups in order to inform about the current state of research.

From its beginning, the DFKI has provided an attractive working environment for AI researchers from Germany and from all over the world. The goal is to have a staff of about 100 researchers at the end of the building-up phase.

Friedrich J. Wendl
Director
TDL ExtraLight User's Guide

Hans-Ulrich Krieger, Ulrich Schäfer

DFKI-D-93-09
This work has been supported by a grant from The Federal Ministry for Research and Technology (FKZ ITW-9002 0).
TDLExtraLight User's Guide*

Hans-Ulrich Krieger, Ulrich Schäfer
{krieger, schaefer}@dfki.uni-sb.de
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3
D-6600 Saarbrücken 11, Germany

Abstract

This paper serves as a user's guide to the first version of the type description language TDL used for the specification of linguistic knowledge in the DISCO project of the DFKI.

*We would like to thank John Nerbonne and Klaus Netter for their helpful comments on an earlier version of this documentation. This work was supported by a research grant (ITW 9002 0) from the German Bundesministerium für Forschung und Technologie to the DFKI DISCO project.
Contents

1 Introduction 3

2 About \textit{TDCExtraLight} 4

3 Starting \textit{TDCExtraLight} 6

4 Syntax and semantics of \textit{TDCExtraLight} 6
   4.1 Type definitions ........................................... 6
      4.1.1 Conjunctive type definitions without inheritance ........................................... 7
      4.1.2 Atoms .................................................. 7
      4.1.3 Type specification and inheritance .................................................. 7
      4.1.4 Multiple inheritance ........................................... 8
      4.1.5 Coreferences ........................................... 8
      4.1.6 Negated coreferences ........................................... 9
      4.1.7 Simple Disjunctions ........................................... 9
      4.1.8 Distributed disjunctions ........................................... 10
      4.1.9 Negation ........................................... 11
      4.1.10 Lists ........................................... 11
      4.1.11 Functional constraints ........................................... 11
      4.1.12 Template calls ........................................... 12
      4.1.13 Type definition options ........................................... 12
   4.2 Template definitions ........................................... 13
   4.3 Instance definitions ........................................... 14
   4.4 Comments ........................................... 14

5 Useful functions, switches and variables 14
   5.1 Creating and changing domains ........................................... 14
   5.2 The reader ........................................... 15
   5.3 Global switches and variables ........................................... 15
   5.4 Hiding attributes at definition time ........................................... 16
   5.5 Collecting parsed identifiers ........................................... 17
   5.6 Getting information about defined types ........................................... 17
   5.7 Getting information about defined templates ........................................... 18
   5.8 Getting information about defined instances ........................................... 18
   5.9 Deleting instances ........................................... 19
   5.10 Printing type prototypes and instances ........................................... 19
      5.10.1 Printing to the interactive screen ........................................... 19
      5.10.2 Printing to \textit{FGRAMED} ........................................... 20
      5.10.3 Printing pretty with \textit{TDC2Iff.TEX} ........................................... 22
      5.10.4 Hiding the type field while printing ........................................... 25

6 Editing and Loading \textit{TDC} files 26

7 Displaying the \textit{TDC} type hierarchy 26

8 Top level abbreviations 26

9 Sample session 27

10 \textit{TDCExtraLight} syntax 30
   10.1 Type definitions ........................................... 30
   10.2 Instance definitions ........................................... 31
   10.3 Template definitions ........................................... 31
1 Introduction

Over the last few years, unification-based grammar formalisms have become the predominant paradigm in natural language processing and computational linguistics. The main idea of representing as much linguistic knowledge as possible via a unique data type called feature structures allows the integration of different description levels, starting with phonology and ending in pragmatics. In this case integration means

1. to represent, process and interpret all linguistic knowledge in one formalism, and
2. to have access to the different description levels and to be able to construct these descriptions in parallel (as syntax and semantics is constructed simultaneously in Montague's framework; cf. [Montague 74])

Here, a feature structure directly serves as an interface between the different description stages, which can be accessed by a parser or a generator at the same time. In this context, unification is concerned with two different tasks: (i) to combine information (unification is a structure-building operation), and (ii) to reject inconsistent knowledge (unification determines the satisfiability of a given structure).

While the first approaches rely on annotated phrase structure rules (for instance GPSG and PATR-II, as well as their successors CLE and ELU [Russell et al. 92]), modern formalisms try to specify grammatical knowledge as well as lexicon entries merely through feature structures. In order to achieve this goal, one must enrich the expressive power of the first unification-based formalisms with disjunctive descriptions. In general, we can distinguish between disjunctions over atoms and disjunctions over complex feature structures. Atomic disjunctions are available in nearly every system. However, they are too weak to represent linguistic ambiguities adequately, motivating the introduction of those ambiguities at higher processing levels. The feature constraint solver UDiNe [Backofen & Weyers 93] of TDLExtraLight allows the use of complex disjunctions and moreover, gives a grammarian the opportunity to formulate distributive disjunctions which are an efficient way to synchronize covarying elements in different attributes through the use of unique disjunction names [Dörre & Eisele 89; Backofen et al. 90]. In addition, this technique obviates the need for expanding to disjunctive normal form, but adds no expressive power to a feature formalism, assuming that it allows for disjunctions.

Later, other operations came into play, viz., (classical) negation or implication. Full negation however can be seen as an input macro facility because it can be expressed through the use of disjunctions, negated coreferences, and negated atoms with the help of existential quantification as shown in [Smolka 88]. UDiNe is currently the only implemented system allowing for general negation. Note that an implication can be easy expressed using negation (although this might not be an efficient way to implement it): \( \phi \rightarrow \psi \equiv \neg \phi \lor \psi \).

Other proposals consider the integration of functional and relational dependencies into the formalism which makes them Turing-complete in general. However the most important extension to formalisms consists of the incorporation of types, for instance in modern systems like TFS [Zajac 92], CUF [Dörre & Eisele 91], or TDL [Krieger & Schäfer 93]. Types are ordered hierarchically (via

---

1[Shieber 86] and [Uszkoreit 88] give an excellent introduction to the field of unification-based grammar theories. Pereira 87] makes the connection explicit between unification-based grammar formalisms and logic programming.

2Almost every theory/formalism use a different notion when referring to feature structures: f-structures in LFG [Bresnan 82], feature bundles or feature matrices in GPSG [Gazdar et al. 85], categories in GPSG, CUG [Uszkoreit 86; Karttunen 86], and CLE [Alshawi 92], functional structures in FUG [Kay 85], terms in DCG [Pereira & Warren 80], attribute-value matrices in HPSG [Pollard & Sag 87] or dags in PATR-II [Shieber et al. 83].

3For instance, Carpenter's ALE system [Carpenter 92] gives a user the opportunity to define definite relations (see [Höhfeld & Smolka 88]), but the underlying constraint system of ALE is even more restricted than the attribute-value logic employed in TDLExtraLight. Definite clauses of ALE can be composed using disjunction, negation, and Prolog cut. However, allowing the user to write Prolog-style relations, e.g., Ait-Kaci's LOGIN [Ait-Kaci & Nasr 86a], gives ALE a flavor more like a general logic programming language than a restricted grammar formalism.
subsumption) as it is known from object-oriented programming languages. This leads to multiple inheritance in the description of linguistic entities (see [Daelemans et al. 92] for a comprehensive introduction). Finally, recursive types are necessary to describe recursion over phrase structure which is inherent in all grammar formalisms relying on a context-free backbone. Other proposals consider the integration of additional data types, for instance sets (cf. [Rounds 88] or [Pollard & Moshier 90]).

Pollard and Sag's Head-Driven Phrase Structure Grammar is currently the most promising grammatical theory which includes all the extensions given above (see [Sag & Pollard 87; Pollard 89; Pollard & Sag 93]). HPSG has been developed further since its first formulation [Pollard & Sag 87], has been applied successfully to the description of tough linguistic phenomena, is interesting from a mathematical viewpoint and is axiomatized to a great extent. HPSG integrates insights from different theories like LFG, GPSG, and GB, but also employs theoretical aspects emerging from situation semantics and DRT. In addition, HPSG covers many ideas from other relating disciplines, like computer science, computational logic and artificial intelligence, especially knowledge representation. HPSG is the ideal representative of the family of unification-based grammar theories which can be characterized roughly by the keywords monotonicity, declarativeness and reversibility.

Martin Kay was the first person who laid out a generalized linguistic framework, called unification-based grammars, by introducing the notions of extension, unification, and generalization into computational linguistics.5 Kay's Functional Grammar [Kay 79] represents the first formalism in the unification paradigm and is the predecessor of strictly lexicalized approaches like FUG, HPSG or UCG [Moens et al. 89]. Pereira and Shieber were the first to give a mathematical reconstruction of PATR-II, in terms of a denotational semantics [Pereira & Shieber 84]. The work of Karttunen led to major extensions of PATR-II, concerning disjunction, atomic negation, and the use of cyclic structures [Karttunen 84]. Kasper and Rounds' seminal work [Kasper & Rounds 86; Rounds & Kasper 86] is important in many respects: they clarified the connection between feature structures and finite automata, gave a logical characterization of the notion of disjunction, and presented for the first time complexity results ([Kasper & Rounds 90] is a good summary of their work). Mark Johnson enriched the descriptive apparatus with classical negation and showed that the feature calculus is a decidable subset of first-order predicate logic [Johnson 88]. Finally, Gert Smolka's work gave a fresh impetus to the whole field: his approach is distinguished from others in that he presents a sorted set-theoretical semantics for feature structures [Smolka 88]. In addition, Smolka gave solutions to problems concerning the complexity and decidability of feature structure descriptions. Further results can be found in [Smolka 89]. Paul King's work aims to reconstruct a special grammar theory, viz. HPSG, in mathematical terms [King 89], whereas Backofen and Smolka's treatment is the most general and complete one, bridging the gap between logic programming and unification-based grammar formalisms [Backofen & Smolka 92]. There exist only a few other proposals to feature structures nowadays which do not use standard first order logic directly, for instance Reape's approach, using a polymodal logic [Reape 91].

2 About TDLExtraLight

TDLExtraLight is a unification-based grammar development environment to support HPSG-like grammars with multiple inheritance. TDL is an acronym for Type Description Language, whereas the suffix ExtraLight should indicate that it is a roughly implemented system with only a few sophisticated features. Work on TDLExtraLight has started at the end of 1988 and is embedded in the DISCO project of the DFKI. The main motivation behind TDLExtraLight was to make a reliable and robust system fast available to the people in the DISCO project: a type system simply

---

5On closer inspection, Kay's proposal was not the first one working with complex features. There have been other approaches in related fields; for instance in linguistics (e.g., [Harman 63]) or compiler construction (e.g., [Knuth 68]), although they made no use of the notion of unification.

---

4Moving from context-free phrase structure rules to ID rule schemata is motivated by the following two facts: (i) there was/is a strong tendency in linguistics to incorporate all kinds of knowledge into feature structures, and (ii) ID schemata are descriptively more adequate than traditional CF rules through the use of underspecification.
belongs to the main ingredients of a modern NLP core machinery. Moreover, a type system can lay the foundations for a grammar development environment because types serve as abbreviations for lexicon entries, categories and principles as is familiar from HPSG (cf. chapter 8 in [Pollard & Sag 87]) and this is exactly the main business TDCExtraLight is currently concerned with. The DISCO grammar consists of 650 type specifications written in TDC and is the largest HPSG grammar for German [Netter 93]. Input given to TDC is parsed by a Zebu-generated parser [Laubsch 93] to allow for a more intuitive input syntax and to abstract from uninteresting details imposed by the unifier and the underlying Lisp system.

The core machinery of DISCO consists of TDCExtraLight and the feature constraint solver UDiNe [Backofen & Weyers 93]. UDiNe is a powerful untyped unification machinery which allows the use of distributed disjunctions, general negation, and functional dependencies. The modules communicate through an interface, and this communication mirrors exactly the way an abstract typed unification algorithm works: two typed feature structures can only be unified if the according types are definitely compatible. This is accomplished by the unifier in that UDiNe handles over two type expressions to TDC which gives back a simplified conjunction of the types.

TDCExtraLight permits type definitions with multiple inheritance and the inheritance of functional dependencies. In addition, TDC allows a grammarian to define and use parameterized templates (macros). Moreover, there exists a special instance definition facility to ease the writing of lexicon entries which differ from normal types in that they are not entered into the type hierarchy.6 However, there are small drawbacks when working with TDCExtraLight.

First of all, every type will be fully expanded at definition time in order to determine the consistency of a feature structure description. Later on, a user is enforced to work with this feature structure, but cannot stick to the old, smaller one. In addition, when using a (complex) type symbol as a part in a description, we have to make sure that this type is already defined, i.e., we are not allowed to refer to an unknown type. As a consequence of this mechanism, TDC rejects recursive type definitions, or to be more precisely, testing the satisfiability of a recursive type leads to an infinite expansion (recursion can only be expressed in the context-free backbone; see below). Second, TDCExtraLight does not support disjunctive or even negated type specifications, although they can be written on the feature constraint level.7

TDCExtraLight comes along with a number of useful tools:

- a type grapher to visualize the underlying type hierarchy (the grapher and also an inspector is supported by the Lisp system)
- a sophisticated interactive feature editor, allowing a user to depict and to edit typed feature structure [Kiefer & Fettig 93]
- a TDC2LaT EX package, transforming typed feature structures into LaT EX code
- a number of software switches, which influence the behaviour of the whole system

Grammars and lexicons written in TDC can be tested by using the chart parser of the DISCO system. The parser is a bidirectional bottom-up chart parser, providing a user with parametrized parsing strategies as well as giving him control over the processing of individual rules (cf. [Kiefer 93] for a general description of the parser module and [Netter 93] for other levels of processing in the DISCO system).

---

6 Strictly speaking, lexicon entries can be seen as the leaves in the type hierarchy which do not admit further subtypes (see also [Pollard & Sag 87], p. 198). Note that this dichotomy is the analogue to the distinction between classes and instances in object-oriented programming languages.

7 The disadvantages of TDCExtraLight mentioned above are no longer present in its successor TDC which will be available in spring '93. The new system is completely redesigned and reimplemented, includes advanced features, is fully incremental and has better performance, although its expressive power increases massively. Moreover, the new TDC makes a parametrized expansion mechanism available to the user (this is needed by a parser or a generator to work efficiently) and support a special form of non-monotonic inheritance (see [Krieger & Schäfer 93] for a full system overview).
3 **Starting **TDLExtraLight

1. Start **Common LISP**.

2. `(load-system "tdl-el")` loads the necessary parts of TDLExtraLight such as the unifier (UDiNE), type definition reader, feature editor (FEGRAMED), type hierarchy management and the TDC2TEX interface. The portable system definition facility DEFSYSTEM is described in [Kantrowitz 91].

3. After loading the Lisp code, the following prompt appears on the screen:

   Welcome to DISCO's Type Definition Language TDL-el.

   USER(1): _

4. To start the TDLExtraLight reader and create a domain for grammar types and symbols, the user should type

   `(DEFINE-DOMAIN :DISCO)`  (or abbreviated :def :disco)

   Any other keyword symbol or string may be chosen instead of DISCO except TDL and the usual Common LISP package names like COMMON-LISP or USER. The name TDL is preserved for internal functions and variables. It is possible to define several domains and to change between them by using function IN-DOMAIN (see Section 5.1).

5. Now it is possible to define types or templates interactively or to load grammar file(s) by simply using the Lisp primitive LOAD. Examples:

   DISCO(2): ? my_first_type := [case nom, num 1].

   DISCO(3): (LOAD "grammar")  (or abbreviated :ld "grammar")

6. DISCO(4): (EXIT)  (or abbreviated :ex)

   exits Lisp and TDLExtraLight.

   The EMACS command C-x C-c kills the Lisp and EMACS process.

4 **Syntax and semantics of TDLExtraLight**

TDLExtraLight can be given a set-theoretical semantics along the lines of [Smolka 88; Smolka 89]. It is easy to translate TDLExtraLight statements into denotation-preserving expressions of Smolka's feature logic, thus viewing TDLExtraLight only as syntactic sugar for a restricted subset of PL1.

The BNF (Backus-Naur Form) of the TDLExtraLight syntax is given in section 10. The syntax is case insensitive. Newline characters, spaces or comments (section 4.4) can be inserted anywhere between the syntax tokens (symbols, braces, parentheses etc.).

All TDLExtraLight definitions must start with a question mark (?) or exclamation mark (!) and end with a period (.). It is important not to forget these delimiters since otherwise the Lisp reader will try to evaluate an expression as Lisp code. It is possible to mix Lisp code and TDL definitions in a file. Some examples are shown in section 9.

4.1 **Type definitions**

The general syntax of a TDLExtraLight type definition is

   ? (type-name) := (type-def) [(options)].

(type-name) is a symbol, the name of the type to be defined. (type-def) is described in the next sections. It is either a conjunctive feature description (sections 4.1.1 and 4.1.3) or a template call (section 4.1.12). (options) will be described in section 4.1.13.
4.1 Type definitions

4.1.1 Conjunctive type definitions without inheritance

All type definitions in *TDLExtraLight* are conjunctive on the top level, i.e., a conjunction of attribute-value pairs. Type definitions using inheritance are described in sections 4.1.3 and 4.1.4. In order to define a feature structure type *person-number-type* with attributes *PERSON* and *NUMBER*, the *TDLExtraLight* syntax is

\[
\text{? person-number-type := [PERSON, NUMBER].}
\]

The definition results in the structure

\[
\begin{align*}
\text{[person-number-type]} \\
\text{PERSON []} \\
\text{NUMBER []}
\end{align*}
\]

If no value is specified for an attribute, the empty feature structure with the top type of the type hierarchy will be assumed. Attribute values can be atoms, conjunctive feature structures, disjunctions, distributed disjunctions, coreferences, lists, functional constraints, template calls, or negated values. The syntax is described in the next sections (BNF on page 30).

4.1.2 Atoms

In *TDLExtraLight*, an atom can be either a number, a string or a symbol. Atoms can be used as values of attributes or as disjunction elements.

Example: The *TDLExtraLight* type definition

\[
\text{? pl-3-phon := [NUMBER plural, PHON "-en", PERSON 3].}
\]

results in the structure

\[
\begin{align*}
\text{[pl-3-phon]} \\
\text{NUMBER plural} \\
\text{PHON "-en"} \\
\text{PERSON 3}
\end{align*}
\]

An example for atoms as disjunctive elements is shown in section 4.1.7.

4.1.3 Type specification and inheritance

All conjunctive feature structures can be given a type specification. Type specification at the top level of a type definition defines inheritance from a supertype. The feature definition of the specified type will be unified with the feature term to which it is attached.

The inheritance relation represents the definitional dependencies of types. Together with multiple inheritance (described in the next section), the inheritance relation can be seen as a directed acyclic graph (DAG).

An example for type specification inside a feature structure definition:

\[
\text{? agr-plural-type := [AGR person-number-type: [NUMBER plural]].}
\]

This definition results in the structure

\[
\begin{align*}
\text{[agr-plural-type]} \\
\text{AGR PERSON []} \\
\text{NUMBER plural}
\end{align*}
\]

Now an example for type inheritance at the top level:
pl-type := person-number-type: [NUMBER plural].

This definition results in the structure

```
[pl-type
 PERSON []
 NUMBER plural]
```

This feature structure is called the **GLOBAL PROTOTYPE** of `pl-type`: a fully expanded feature structure of a defined type which has inherited all information from its supertype(s) is called a **GLOBAL PROTOTYPE**. A feature structure consisting only of the local information given by the type definition is called a **LOCAL PROTOTYPE**. So the **LOCAL PROTOTYPE** of `pl-type` is

```
[pl-type
 NUMBER plural]
```

Section 5.10 explains how the different prototypes of a defined type can be displayed. As mentioned above, type specification is optional. If no type is specified, the top type `*var*` of the type hierarchy will be assumed.

### 4.1.4 Multiple inheritance

On the top level of a feature type definition, multiple inheritance is possible, while inside feature structures only a single type is allowed which might inherit in its definition from multiple types. As an example for multiple inheritance, suppose `number-type`, `person-type` and `gender-type` are defined as follows:

? number-type := [NUMBER].
? person-type := [PERSON].
? gender-type := [GENDER].

Then the **TDCExtraLight** type definition

? mas-2-type := (number-type,
 person-type,
 gender-type): [GENDER mas,
 PERSON 2].

would result in the following structure:

```
[mas-2-type
 GENDER mas
 PERSON 2
 NUMBER []]
```

### 4.1.5 Coreferences

Coreferences indicate information sharing between feature structures. In **TDCExtraLight**, coreference symbols are written **before** the value of an attribute or **instead** of an attribute value. A coreference symbol consists of the hash sign (#), followed by either a number (positive integer) or a symbol. However, in the internal representation and in the printed output of feature structure, the coreference symbols will be normalized to an integer number. Example:

? share-pn := [SYN #pn person-number-type : [ ],
 SEM #pn ].
4.1 Type definitions

results in the following structure:

\[
\begin{align*}
\text{share-pn} & \\
\text{SYN} & \begin{cases} 
\text{PERSON} & \text{[]} \\
\text{NUMBER} & \text{[]} 
\end{cases} \\
\text{SEM} & \text{[]} 
\end{align*}
\]

4.1.6 Negated coreferences

Negated coreferences specify that two attributes must not share the same value, i.e., they may have the same value, but these values must not be linked to each other by coreferences.

The Syntax of negated coreferences is

\[ -#(a_1, a_2, \ldots a_n) \]

where \(a_1, a_2, \ldots a_n\) are coreference symbols, i.e., numbers or symbols, without the hash sign.

Negated coreferences are not allowed at the top level of a type definition.

Example: The \textit{TDCExtraLight} definition

\[ ? \text{ give } : = \text{[RELN give, GIVER } -#(1,2), \text{ GIVEN } #1, \text{ GIVEE } #2]. \]

would result in the following structure:

\[
\begin{align*}
\text{give} & \\
\text{RELN give} & \\
\text{GIVER} & \begin{cases} 
\text{[1.2]} & \text{[]} 
\end{cases} \\
\text{GIVEN} & \text{[]} \\
\text{GIVEE} & \text{[]} 
\end{align*}
\]

4.1.7 Simple Disjunctions

Disjunctive alternatives are enclosed in braces (\{\ldots\}) and separated by commas. Disjunction elements can be atoms, conjunctive feature descriptions, simple disjunctions, distributed disjunctions, lists, template calls or negated values. In simple disjunctions, the alternatives must not contain coreferences to values outside the alternative itself (see [Backofen & Weyers 93] for the reasons).

Distributed disjunctions allow for a restricted way to use coreferences to outside disjunction alternatives (section 4.1.8). Another restriction in \textit{TDCExtraLight} is that disjunctions are not allowed at the top level of a type definition.

Example for disjunctions in a type definition:

\[ ? \text{ person-1-or-2 } : = \text{[SYN } \{ 
\text{ person-number-type: [PERSON } 1, \\
\text{ person-number-type: [PERSON } 2 ] \} \}. \]

The resulting feature structure is

\[
\begin{align*}
\text{person-1-or-2} & \\
\text{SYN} & \begin{cases} 
\text{person-number-type} & \text{[PERSON } 1 \\
\text{NUMBER} & \text{[]} 
\end{cases} \\
& \begin{cases} 
\text{person-number-type} & \text{[PERSON } 2 \\
\text{NUMBER} & \text{[]} 
\end{cases} 
\end{align*}
\]

Another more local specification of the same disjunction would be
? person-1-or-2 := [ SYN person-number-type: [PERSON { 1, 2 }] ].

The resulting feature structure is

```
person-1-or-2

<table>
<thead>
<tr>
<th>person-number-type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYN PERSON { 1, 2 }</td>
</tr>
<tr>
<td>NUMBER []</td>
</tr>
</tbody>
</table>
```

### 4.1.8 Distributed disjunctions

A very useful feature of TDCExtraLight defined in the underlying unification system UDINE are distributed disjunctions. Distributed disjunctions are a special kind of disjunctions which allow to restrict the specification of disjunctions affecting more than one attribute to a local domain, thus avoiding the necessity of constructing a disjunctive normal form in many cases. Consider the following example:

```
season-trigger

| SEASON $1 { "spring", "summer", "fall", "winter" } |
| NUMBER $1 { 1, 2, 3, 4 } |
```

This structure has been generated by the following TDCExtraLight expression:

```
? season-trigger := [SEASON %i{"spring", "summer", "fall", "winter"},
                     NUMBER %i{ 1, 2, 3, 4 }].
```

When a structure of type `season-trigger` will be unified with the structure `[SEASON {"summer", "fall"}]`, then the value of attribute `NUMBER` will become `{2, 3}`, i.e., the value of attribute `SEASON` triggers the value of attribute `NUMBER`, and vice versa.

The syntax of an alternative list in distributed disjunctions is

```
%i{a_1, ..., a_n},
```

where \(i\) is an integer number, the disjunction index for each group of distributed disjunctions (%i in the example). More than two alternative lists per index are allowed. All distributed disjunctions with the same index must have the same number (n) of alternatives. The disjunction index is local in every type definition and is normalized to a unique index when unification of feature structures takes place.

In general, if alternative \(a_{ij} (1 \leq j \leq n)\) does not fail, it selects the corresponding alternative \(b_{ij}, c_{ij}, \ldots\) in all other distributed disjunctions with the same disjunction index \(i\).

As in the case of simple disjunctions, disjunction alternatives must not contain coreferences to values outside the alternative itself. But for distributed disjunctions, there is an exception to this restriction: disjunction alternatives may contain coreferences to values in another distributed disjunction if both disjunctions have the same disjunction index and the alternative containing the coreference has the same position in the disjunction alternative list.

An example for such a distributed disjunctions with coreferences is:

```
? dis2 := [a %i{ [], #1, #2 },
            b %i{ [c +], x:[d #1 g:[m -]], x:[d #2 g:[m +]] }].
```
### 4.1 Type definitions

#### 4.1.9 Negation

The `~` sign indicates negation. Example:

\[
? \text{not-mas-type} := \exists \text{GENDER} \sim \text{mas}.
\]

The resulting feature structure is

\[
\begin{array}{c}
\text{not-mas-type} \\
\text{GENDER} \sim \text{mas}
\end{array}
\]

#### 4.1.10 Lists

In *TDC ExtraLight*, lists are represented as first-rest structures with distinguished attributes *FIRST* and *REST*, where the atomic value *end* indicates the empty list. The input of lists can be abbreviated by using the `< ... >` syntax:

\[
? \text{list-it} := \exists \text{LIST} < \text{first-element}, \text{second}, \#\text{last}>, \text{LAST} \#\text{last}, \text{AN-EMPTY-LIST} < > \).
\]

The resulting feature structure is

\[
\begin{array}{c}
\begin{array}{c}
\text{list-it} \\
\text{LIST} \\
\text{LAST} \\
\text{AN-EMPTY-LIST}
\end{array}
\end{array}
\]

#### 4.1.11 Functional constraints

Functional constraints define the value of an attribute on the basis of a function which has to be defined and computed outside the *TDC* system.

The syntax of functional constraints is
% (function name) ((function parameters))

String concatenation is a nice example for the use of functional constraints:

? add-prefix := [WORD #word, PREFIX #prefix, WHOLE %CONCATENATE (STRING, #prefix, #word)].

where CONCATENATE is the generic LISP function for concatenation of sequences. The usual representation for functional constraints is:

```
[add-prefix]
WORD 1
PREFIX 2
WHOLE 3
```

Functional Constraints:

\[ 3 = \text{concatenate}(\text{string}, 2, 1) \]

The evaluation of functional constraints will be residuated until all parameters are instantiated [Aït-Kaci & Nasr 86b; Smolka 91]. Evaluation can be enforced by using the function EVAL-CONSTRAINTS of the UNIFY package. Further details are described in [Backofen & Weyers 93].

4.1.12 Template calls

Templates are pure textual macros which allow to specify (parts of) type or instance definitions by means of some shorthand. The definition of templates will be explained in section 4.2. Template call simply means syntactic replacement of a template name by its definition and possibly given parameters.

The syntax of template call is

```
CD(template name) «template parameter pairs»
```

where a (template parameter pair) is a pair consisting of a parameter name (starting with the $ character) and a value. All occurrences of the parameter name will be replaced by the value given in the template call or by the default value given in the template definition. See section 4.2 for further details and examples.

4.1.13 Type definition options

For external use, TVC allows a number of optional specifications which give information which is basically irrelevant for the grammar. If the optional keywords are not specified, default values will be assumed by the TVC control system. (options) for type definitions are the optional keywords :author, :doc, :date and :status. When specified, a value must follow the corresponding keyword.

The values of :author, :doc and :date must be strings. The default value of :author is defined in the global variable *AUTHOR*. The default value of :doc is defined in the global variable *DEFAULT-DOCUMENTATION* (see section 5). The default value of :date is a string containing the current time and date.

The :status information is necessary if the grammar should be processed by the DISCO parser. It distinguishes between different categories of types and type instances, e.g., lexical entries, rules or root nodes. If the :status keyword is given (valid values: see rule statuskey in the BNF syntax
on page 30), the status value of the type will become the specified one. If no status option is given, the status will be inherited from the supertype (or be :unknown, if the supertype is the top type of the type hierarchy).

In order to access the :author, :doc, :date and :status values of type, functions with the corresponding names (status etc.) can be used. See section 5.6 for details and examples.

### 4.2 Template definitions

Templates in *TDL ExtraLight* are what parametrized macros are in programming languages: syntactic replacement of a template name by its definition and (possibly) replacement of given parameters in the definition. In addition, the specification of default values for template parameters is possible in the template definition. Templates are very useful for writing grammars that are modular; they can also keep definitions independent (as far as possible) from specific grammar theories.

The general syntax of a *TDL ExtraLight* template definition is

```
? (template-name) ((template parameter pairs)) := (template-body) [(options)].
```

where a (template parameter pair) is a pair consisting of a parameter name (starting with the $ character) and a default value. All occurrences of the parameter name will be replaced by the value given in the template call or by the default value given in the template definition. (template-body) can be a complex description as in type definitions.

Example: The template definition

```
? a-template ($inherit *var*, $attrib PHON, $value) :=
   $inherit:[$attrib #1 $value, COPY #1].
```

makes it possible to generate the following types using template calls:

  - *top-level-call* is a top-level template call which will result in the feature structure:
    ```
    [top-level-call]
    [PHON []]
    [COPY []]
    ```

- `? inside-call := [top-attrib a-template ($value "hello", $attrib MY-PHON)].`
  - *inside-call* is a template call inside a feature type definition which will result in the feature structure:
    ```
    [inside-call]
    [TOP-ATTRIB]
    [MY-PHON "hello"]
    [COPY "hello"]
    ```

(options) in template definitions are the optional keywords :author, :date and :doc. When specified, a keyword must be followed by a string. The default value for the :author string is defined in the global variable *AUTHOR*. The default value for the :doc string is defined in the global variable *DEFAULT-DOCUMENTATION* (see section 5). The default value for :date is a string containing the current time and date.

Section 5.7 describes the functions DESCRIBE-TEMPLATE and RETURN-ALL-TEMPLATE-NAMES which print information about template definitions.
4.3 Instance definitions

An instance of a TDL type is a copy of the GLOBAL PROTOTYPE of the specified type plus (possibly) additional instance-specific information. For instance, each lexical entry will typically be an instance of a more general type, e.g., \textit{intransitive-verb-type} with additional specific graphemic and semantic information. In addition, an instance can also be defined by a template call. Instances will not be inserted into the TDL type hierarchy. In general, instances are objects which will be used by the parser. It is possible to create several instances of the same type with different or the same instance-specific information.

The general syntax of a \texttt{TDLExtraLight} instance definition is

\begin{verbatim}
! (type-name) [(instance-body)] [(options)].
or
! (template-call) [(options)].
\end{verbatim}

[(instance-body)] can be a complex description as in type definitions. (options) in instance definitions are the optional keywords :author, :doc, :date, :name and :status. When specified, a value must follow the corresponding keyword.

If :name is specified, its value must be a symbol which will become the name of the defined instance. If :name is not specified, the instance name will be 'computed' from the symbol (type-name) and a number which always guarantees to create a fresh and unique instance name and allows to distinguish between different instances of the same type. If the same name is given more than once for an instance of the same type, the old entries will not be destroyed and the parser is responsible for the access to all instances. Functions PTI, FTI and LTI always take the last instance defined with the specified name.

If the :status keyword is given (valid values: see rule \texttt{statuskey} in the BNF syntax on page 30), the status value of the instance will become the specified one. If no status option is given, the status will be inherited from (type-name).

The values of :author, :doc and :date must be strings. The default value of :author is defined in the global variable *AUTHOR*. The default value of :doc is defined in the global variable *DEFAULT-DOCUMENTATION* (see section 5). The default of :date is the current time and date.

4.4 Comments

; after an arbitrary token or at the beginning of a line inserts a comment which will be ignored by the TDL reader until end of line. It is also possible to use the COMMON LISP block comment delimiters #|| and ||#. A comment associated with a specific type, template or instance definition should be given in the :doc string at the end of the definition.

5 Useful functions, switches and variables

The following functions and global variables are defined in the package TDL and are made public to all user-defined domains (implemented by COMMON LISP packages) via \texttt{use-package}. This is done automatically in the function \texttt{DEFINE-DOMAIN}.

5.1 Creating and changing domains

Domains are sets of type, instance and template definitions. It is possible to define several domains and to have definitions with the same names in different domains. Domains roughly correspond to packages in COMMON LISP (in fact, they are implemented using the package system).

- function \texttt{(DEFINE-DOMAIN domain-name [:hide-attributes attribute-list] [:export-symbols symbol-list] [:errorp {T|NIL}]})
defines a new domain domain-name (a symbol or a string) and turns the TDL reader on. The global variable *DOMAIN* is set to domain-name. Options: attribute-list is the list of attributes to be hidden (see section 5.4), symbol-list is a list of symbols to be exported from the domain package. If errorp is T, a redefinition of a domain will cause an error, otherwise (NIL) a redefinition of a domain will give a warning; default is NIL. Example:

DISCO(6): (DEFINE-DOMAIN :DISCO :hide-attributes '(SEM))

• function (IN-DOMAIN domain-name [:errorp {T|NIL}])
changes the current domain to domain-name (a symbol or a string) and turns on the TDL reader. The global variable *DOMAIN* is set to domain-name. If errorp (optional) is T, using an undefined domain name will cause an error. If errorp is NIL (default), a warning will be given and the current domain will not be changed. Example:

DISCO(6): (IN-DOMAIN :DISCO)

• global variable *DOMAIN*

  *DOMAIN* contains the name of the current domain (a string). The value of *DOMAIN* should only be changed by DEFINE-DOMAIN or IN-DOMAIN, but not directly by the user. Example:

DISCO(7): *DOMAIN* "DISCO"

5.2 The reader

The reader of TDLExtraLight uses the two macro characters ? and ! in order to detect the beginning of a type, template or instance definition. Before loading complex LISP code, the reader should be switched off temporarily. This can be done by using function ROFF. Example:

DISCO(8): (ROFF)  
(or alternatively :roff)

Some errors cause the reader to be switched off automatically. After this or after loading a LISP file, the reader can be switched on by function RON. Example:

DISCO(9): (RON)  
(or alternatively :ron)

The functions DEFINE-DOMAIN and IN-DOMAIN include an implicit (RON).

5.3 Global switches and variables

The following global LISP variables can be set by the user. Switches are set to T for ON or NIL for OFF.

• global variable *WARN-IF-TYPE DOES NOT EXIST*  
  default value: T

  This variable controls whether a warning will be given if a type definition contains the name of an undefined type in its body. Example:

DISCO(10): (SETQ *WARN-IF-TYPE DOES NOT EXIST* NIL)  
NIL

• global variable *WARN-IF-REDEFINE-TYPE*  
  default value: T

  This variable controls whether a warning will be signaled if a type already exists and is about to be redefined. Example:

DISCO(11): (SETQ *WARN-IF-REDEFINE-TYPE* NIL)  
NIL

• global variable *AUTHOR*  
  default value: ""

  This variable should contain the name of the grammar author or lexicon writer. It will be used as default value for the optional keyword :author in type, template and instance definitions. Example:
5 USEFUL FUNCTIONS, SWITCHES AND VARIABLES

DISCO(12): (SETQ *AUTHOR* "Donald Duck")
"Donald Duck"

- global variable *DEFAULT-DOCUMENTATION*
  
  **default value**: ""

  This parameter specifies the default documentation string for type, template and instance definitions. Example:

  DISCO(13): (SETQ *DEFAULT-DOCUMENTATION* "Version 2.7")
"Version 2.7"

- global variable *VERBOSE-TYPE-DEFINITION-P*
  
  **default value**: NIL

  This parameter specifies the verbosity behavior during processing type definitions. If the value is NIL, only the name of the (successfully) defined type will be printed in brackets, e.g., ¦type[VERB-TYPE]. If an error occurs, the output behavior will be independent of the value of *VERBOSE-TYPE-DEFINITION-P*. Example:

  DISCO(14): (SETQ *VERBOSE-TYPE-DEFINITION-P* T)

  T

- global variable *VERBOSE-TDL2UNIFY-P*
  
  **default value**: NIL

  This parameter increases verbosity in type definitions, especially for debugging purposes. If set to T, the interface function between type system and unifier, TDL2UNIFY, will print the structures which are passed to the unifier. Example:

  DISCO(15): (SETQ *VERBOSE-TDL2UNIFY-P* T)

  T

- global variable *LAST-TYPE*

  This variable contains the name of the last type defined. It is used by the printing functions PGP, PLP, LGP, LLP, FGP, FLP, SUPERTYPES and RETURN-ALL-INSTANCE-NAMES if no parameter is specified. The value of this variable can be changed by the user. Example:

  DISCO(16): *LAST-TYPE*

  AGR-EN-TYPE

  DISCO(17): (SETQ *LAST-TYPE* 'MYTYPE)

  MYTYPE

- global variable *UNIFY-TYPES*
  
  **default value**: T

  If set to T (which is the default), the type field of a feature structure will be reduced to the most specific type(s) using the type hierarchy at definition time or when unification takes place. Otherwise (if *UNIFY-TYPES* is set to NIL), the type field of the resulting feature structure will not be reduced using the type hierarchy. In this case, the type entries become longer and less readable. Function SUPERTYPES returns a list of all supertypes of a type, see section 5.6.

  Important note: changes to *UNIFY-TYPES* will not have an effect on previously defined types or instances.

5.4 Hiding attributes at definition time

It is possible to hide values of attributes at type definition time, so that values will never be used and coreferences out of such structures will never be regarded.

- function (SET-HIDE-ATTRIBUTES attribute-list [domain-name])

  This function sets the list of the attributes to be hidden in the following type definitions. There is one such list for each domain. If no domain is specified, the current domain is taken as the default. The option :hide-attributes in function DEFINE-DOMAIN has the same effect as SET-HIDE-ATTRIBUTES.

  Important note: SET-HIDE-ATTRIBUTES will not have an effect on previously defined types. Example:

  DISCO(18): (SET-HIDE-ATTRIBUTES '(NUM GENDER) :DISCO)

  (NUM GENDER)
5.5 Collecting parsed identifiers

- function (GET-HIDE-ATTRIBUTES [domain-name])
  This function yields the list of the attributes to be hidden (see SET-HIDE-ATTRIBUTES). If no domain is specified, the current domain is taken by default. Example:
  DISCO(19): (GET-HIDE-ATTRIBUTES :DISCO)
  (NUM GENDER)

- global variable *HIDE-COMPLETELY*  
  default value: NIL
  This variable controls whether attributes and values will be hidden (= T) or only the attribute's value (= NIL).
  Important note: changes to *HIDE-COMPLETELY* will not have an effect on previously defined types.
  Example:
  DISCO(20): (SETQ *HIDE-COMPLETELY* T)
  T

5.5 Collecting parsed identifiers

- function (GET-IDENTIFIERS [domain-name])
  yields a list of all identifiers (i.e., type names, attribute names and atomic value names) passed through the TV! reader so far. There is a unique list for each domain. Collecting all identifiers of a domain is useful when working in several domains (i.e., COMMON LISP packages) at the same time. Example:
  DISCO(21): (GET-IDENTIFIERS :DISCO)
  (HUM GEN AGR-TYPE ... )

- function (RESET-IDENTIFIERS [identifier-list] [domain-name])
  resets the list of all identifiers (i.e., type names, attribute names and atomic value names) passed through the TV! reader so far. There is a unique list for each domain. The default value of identifier-list is the empty list. Example:
  DISCO(22): (RESET-IDENTIFIERS)
  NIL

5.6 Getting information about defined types

All functions described in this section (except the last one) take an argument type which must not be quoted.

- function (AUTHOR type)
  returns the author's name (a string) given in the definition of type or in global variable *AUTHOR*.
  Example:
  DISCO(23): (author agr-en-type)
  "Klaus Netter"

- function (DOC type)
  returns the documentation string given in the definition of type or in the global variable *DEFAULT-DOCUMENTATION*.
  Example:
  DISCO(24): (doc agr-en-type)
  "Agreement for -en."

- function (DATE type)
  returns time and date of definition of type.
  Example:
  DISCO(25): (date agr-en-type)
  "The feature type AGR-EN-TYPE was defined on 04/16/1993 at 18:09:40"

- function (STATUS type)
  returns the status symbol given in the definition of type or inherited by its supertype (default). Further details are described in section 4.1.13. Example:
5 USEFUL FUNCTIONS, SWITCHES AND VARIABLES

DISCO(26): (status agr-en-type)
:UNKNOWN

• function (SURFACE type)
  returns the definition string of type. Example:
  DISCO(27): (surface person-number-type)
  "? person-number-type := [PERSON, NUMBER]."

• function (SUPERTYPES [type])
  This function returns a (possibly empty) list of all types type inherits from, i.e., the super-
types of type. The default for type is the name of the last type defined, i.e., the value of the
global variable *LAST-TYPE*. Example:
  DISCO(28): (supertypes agr-en-type)
  (AGR-GRADE-TYPE AGR-TYPE GRADE-TYPE AGR-FEAT)

• function (RETURN-ALL-TYPE-NAMES)
  RETURN-ALL-TYPE-NAMES prints and returns the names of all types defined before. Example:
  DISCO(29): (return-all-type-names)

  The following types are defined:

  PERSON-NUMBER-TYPE
  PL-3-PHON
  AGR-PLURAL-TYPE

Functions for printing prototypes are described in section 5.10.

5.7 Getting information about defined templates

• function (DESCRIBE-TEMPLATE template-name)
  DESCRIBE-TEMPLATE prints a short information text about a template definition. Example:
  DISCO(30): (describe-template 'a-template)

  The template A-TEMPLATE was defined on 04/15/1993 at 17:12:23.
  The author is: tdl-info.
  The following definition is associated with A-TEMPLATE:
  ? a-template ($inherit *var*, $attrib PHON, $value) :=
  $inherit:[$attrib #1 $value,
  COPY #1].

• function (RETURN-ALL-TEMPLATE-NAMES)
  RETURN-ALL-TEMPLATE-NAMES prints and returns the names of all templates defined before. Example:
  DISCO(31): (return-all-template-names)

  The following templates are defined:

  A-TEMPLATE

5.8 Getting information about defined instances

• function (RETURN-ALL-INSTANCE-NAMES [type-name])
  RETURN-ALL-INSTANCE-NAMES prints and returns the names of all instances of type type-
  name. If no type name is specified, RETURN-ALL-INSTANCE-NAMES prints and returns all
instances of the last type defined. If type-name is :all, the function will print and return all instance names of all types defined before. Example:

DISCO(32): (return-all-instance-names 'trans-verb-lex)

The following instances of type TRANS-VERB-LEX are defined:

TRANS-VERB-LEX24068
TRANS-VERB-LEX24118
TRANS-VERB-LEX24098

Functions for printing instances are described in section 5.10.

5.9 Deleting instances

• function (CLEAR-INSTANCES [instance-name])
removes instance instance-name or all instances from the hashtable *FEATURE-TYPES*. If no instance-name is specified, then the default value :all will be taken. In this case, all instances will be removed. Example:
DISCO(33): (CLEAR-INSTANCES)
NIL

5.10 Printing type prototypes and instances

For debugging and documentation purposes, it is possible to print the prototype and instances of a defined feature type. This can be done by using the following functions.

5.10.1 Printing to the interactive screen

• function (PLP [type-name [p-options]])
PLP prints the LOCAL PROTOTYPE of the feature structure with name type-name. If no type name is specified, PLP prints the prototype of the last type defined before evaluating PLP. The LOCAL PROTOTYPE contains only the local information given in the definition of type type-name. Example:
DISCO(34): (PLP 'MAS-SG-AGR :hide-types T :init-pos 12)
[GENDER : [FEM : -
MAS : +]
NUM : SG]

• function (PGP [type-name [p-options]])
PGP prints the GLOBAL PROTOTYPE of the feature structure with name type-name. If no type name is specified, PGP prints the prototype of the last type defined before evaluating PGP. The GLOBAL PROTOTYPE contains all information that can be inferred for type type-name and its supertypes. Example:
DISCO(35): (PGP 'MAS-SG-AGR :hide-types nil)
MAS-SG-AGR [GENDER : GENDER-VAL [FEM : -
MAS : +]
CASE : []
NUM : SG]

• function (PTI instance-name [p-options])
PTI prints the feature structure of instance instance-name. Example:
DISCO(36): (PTI 'agr-en-type4335)

p-options are the following optional keywords:
5 USEFUL FUNCTIONS, SWITCHES AND VARIABLES

- **:hide-types flag**  
  *default value:* the value of global variable *HIDE-TYPES* = NIL  
  *possible values:* {T|NIL}  
  If *flag* is NIL, types will be printed before feature structures (the top type will not be printed). If *flag* is T, types will not be printed. See section 5.10.4.

- **:remove-tops flag**  
  *default value:* NIL  
  *possible values:* {T|NIL}  
  If *flag* is T, attributes with empty values (i.e., values that unify with any value) will not be printed. If *flag* is NIL, all attributes (except those in label-hide-list) will be printed.

- **:label-hide-list list**  
  *default value:* ()  
  *possible values:* a list of symbols (attribute names)  
  Attributes in list and their values will not be printed.

- **:label-sort-list list**  
  *default value:* the value of *LABEL-SORT-LIST*  
  *possible values:* a list of symbols (attribute names)  
  *list* defines an order for attributes to be printed. Attributes of the feature structure will be printed first-to-last according to their left-to-right position in *list*. All remaining attributes which are not member of *list* will be printed at the end.

- **:stream stream**  
  *default value:* T  
  *possible values:* {T | NIL | a LISP stream variable}  
  If *stream* is T, the feature structure will be printed to standard output or to the interactive screen. If *stream* is NIL, the feature structure will be printed to a string. In all other cases the feature structure will be printed to the LISP stream *stream*.

- **:init-pos number**  
  *default value:* 0  
  *possible values:* a positive integer number  
  *number* defines the left margin offset (in space character units) for the feature structure to be printed.

5.10.2 Printing to FEGRAMED

FEGRAMED is DISCO's feature structure editor. Further details are described in [Kiefer & Fettig 93].

- function (FLP [type-name [f-options]])  
  FLP starts FEGRAMED with the LOCAL PROTOTYPE of the feature structure with name *type-name*. If no type name is specified, FLP takes the prototype of the last type defined before evaluating FLP. The LOCAL PROTOTYPE contains only the local information given in the definition of type *type-name*. Example:  
  DISCO(37): (FLP 'MYTYPE)

- function (FGP [type-name [f-options]])  
  FGP starts FEGRAMED with the GLOBAL PROTOTYPE of the feature structure with name *type-name*. If no type name is specified, FGP takes the prototype of the last type defined before evaluating FGP. The GLOBAL PROTOTYPE contains all information that can be inferred for type *type-name* and its supertypes. Example:  
  DISCO(38): (FGP 'MAS-SG-AGR :wait T :hide-types T)

- function (FTI instance-name [f-options])  
  FTI starts FEGRAMED with the feature structure of instance *instance-name*. Example:  
  DISCO(39): (FTI 'agr-en-type4335)

*f-options* are the following optional keywords:
5.10 Printing type prototypes and instances

Figure 1: A feature structure type in FGGRAMED

- **:hide-types flag**
  - **default value**: the value of global variable *HIDE-TYPES* = NIL
  - **possible values**: {T|NIL}

If flag is NIL, types will be printed at the top of feature structures. If flag is T, types will
not be printed. See section 5.10.4.

- \textbf{filename} \texttt{filename} \hspace{1em} \textit{default value: "type-name-gp.fed", "type-name-lp.fed" or possible values: a string or a \textsc{Lisp} path name "instance-name.fed"}
  Unless \texttt{filename} is specified, a filename will be 'computed' from the type name. The file will be created by the \texttt{TDC-FEGRAMED} interface in order to communicate the feature structure information.

- \textbf{wait flag} \hspace{1em} \textit{default value: NIL}
  possible values: \{T\lceil\textsc{NIL}\rfloor\}
  If flag is T, \texttt{FEGRAMED} will wait until the user chooses the return options. If flag is \textsc{NIL}, \texttt{FEGRAMED} will not wait.

An example screen dump of a feature structure in \texttt{FEGRAMED} is shown in Figure 1.

5.10.3 Printing pretty with \texttt{TDC2LATEX}

\texttt{TDC2LATEX} is a tool which generates \texttt{LATEX} compatible high-quality output of \texttt{TDC} feature structure types.

- function \texttt{(LLP \{type-name \{\text{-options}\}\})}
  LLP starts \texttt{TDC2LATEX} with the \textsc{local prototype} of the feature structure with name \texttt{type-name}. If no type name is specified, LLP takes the prototype of the last type defined before evaluating LLP. The \textsc{local prototype} contains only the \textit{local} information given in the definition of type \texttt{type-name}. Example:
  \texttt{DISCO(40): (LLP 'agr-en-type :fontsize "small" :doc-options "a4wide,palatino")}

- function \texttt{(LGP \{type-name \{\text{-options}\}\})}
  LGP starts \texttt{TDC2LATEX} with the \textsc{global prototype} of the feature structure with name \texttt{type-name}. If no type name is specified, LGP takes the prototype of the last type defined before evaluating LGP. The \textsc{global prototype} contains \textit{all} information that can be inferred for type \texttt{type-name} and its supertypes. Example:
  \texttt{DISCO(41): (LGP 'agr-en-type :mathmode "equation" :doc-options "leqno")}

- function \texttt{(LTI \{instance-name \{\text{-options}\}\})}
  LTI starts \texttt{TDC2LATEX} with the feature structure of instance \texttt{instance-name}. Example:
  \texttt{DISCO(42): (LTI 'agr-en-type4335)}

An example of a complex feature structure generated by \texttt{TDC2LATEX} is shown in Figure 2.

\texttt{l-options} are the following optional keywords:

- \textbf{filename} \texttt{filename} \hspace{1em} \textit{default value: "type-name-gp", "type-name-lp" or possible values: string "instance-name"}
  Unless \texttt{filename} is specified, a filename will be 'computed' from the type name. The filename will be used to generate the \texttt{LATEX} output file.

- \textbf{filepath} \texttt{pathname} \hspace{1em} \textit{default value: \texttt{value of variable \texttt{\textbullet\textsc{FILEPATH}}}}
  possible values: a string or a \textsc{common lisp} path name \texttt{pathname} sets the directory in which the \texttt{LATEX} output file will be created and the shell command \texttt{command} will be executed. The value of \texttt{\textbullet\textsc{FILEPATH}} defaults to the \texttt{tmp} directory in the user's home directory.

- \textbf{hide-types flag} \hspace{1em} \textit{default value: \texttt{value of variable \textbullet\textsc{HIDE-TYPES}} = \textsc{NIL}}
  possible values: \{T\lceil\textsc{NIL}\rfloor\}
  If flag is \textsc{NIL}, types will be printed at the top of feature structures (the top type will not be printed). If flag is T, types will not be printed. See section 5.10.4.
Figure 2: A complex feature structure generated by TDC2LaTeX

- `:remove-tops flag`
  - default value: value of *REMOVE-TOPS* = NIL
  - possible values: {T|NIL}
  - If flag is T, attributes with empty values (i.e., values that unify with any value) will not be printed. If flag is NIL, all attributes (except those in LABEL-HIDE-LIST) will be printed.

- `:label-hide-list list`
  - default value: value of *LABEL-HIDE-LIST* = ()
  - possible values: a list of symbols (attribute names)
  - Attributes in list will not be printed.
• :label-sort-list list default value: value of variable *LABEL-SORT-LIST* = ()
possible values: a list of symbols (attribute names)
list defines an order for attributes to be printed. Attributes of the feature structure will be
printed first-to-last according to their left-to-right position in list. All remaining attributes
which are not member of list will be printed at the end.

• :shell-command command default value: value of *SHELL-COMMAND* = "td12latex"
possible values: {NIL | string }
If command is NIL, only the \LaTeX file will be created and TDC2\LaTeX will return. If command is a string, TDC2\LaTeX will start a shell process and execute command with
parameter filename. An example for command is the following shell script with name td12ps
which starts \LaTeX with the output file of TDC2\LaTeX and writes PostScript\textsuperscript{TM} code to the
file filename.ps:

#!/bin/sh
latex $1
dvips $1 -o $1.ps

• :wait flag default value: value of variable *WAIT* = NIL
possible values: {T|NIL}
If flag is NIL and the shell command command is not NIL, command will be started as a
background process. Otherwise, TDC2\LaTeX will wait for command to be terminated.

• :latex-header-p flag default value: value of *LATEX-HEADER-P* = T
possible values: {T|NIL}
If flag is T, a complete \LaTeX file with \texttt{documentstyle} etc. will be generated. If flag is
NIL, only the \LaTeX code of the feature structure enclosed in \texttt{\begin{featurestruct}} and
\texttt{\end{featurestruct}} will be written to the output file. This is useful for inserting \LaTeX feature structures into \LaTeX documents for papers, books etc.

• :align-attributes-p flag default value: value of *ALIGN-ATTRIBUTES-P* = NIL
possible values: {T|NIL}
If flag is T, attribute names and values will be aligned. If flag is NIL, no alignment will take
place.

• :fontsize size default value: value of *FONTSIZE* = "normalsize"
possible values: a string
This parameter sets the size of the \LaTeX feature structures. It must be a string consisting of a valid \LaTeX font size name, e.g., "tiny", "scriptsize", "footnotesize", "small", "normalsize", "large", "Large", "LARGE", "huge" or "Huge".

• :corefsize size default value: value of *COREFSIZE* = NIL
possible values: { string | NIL }
This parameter sets the font size for coreference symbols. If size is NIL, the size for the
coreference symbol font will be computed from the value of the :fontsize keyword. A font
one magnification step smaller than given in :fontsize will be taken. If size is a string, it
must contain a valid \LaTeX font size as in :fontsize.

• :coreffont string default value: value of variable *COREFFONT* = "rm"
This parameter sets the \LaTeX font style for printing coreference symbols. string must
contain a valid \LaTeX font style, e.g., tt, bf, it etc.

• :coreftable a-list default value: value of variable *COREFTABLE* = ()
This parameter defines a translation table for coreferences and corresponding full names
(strings or numbers), e.g., ((1, "subcat") (2, "phon") (3, 1) (4, 2)). All
coreference numbers at the left side of each element in a-list will be replaced by the right
side. All other coreferences will be left unchanged.
5.10 Printing type prototypes and instances

- **arraystretch number**  
  default value: value of *ARRAYSTRETCH* = 1.1  
  This parameter sets the vertical distance between attribute names or disjunction alternatives.  
  *number* is a factor which will be multiplied with the standard character height.

- **arraycolsep string**  
  default value: value of *ARRAYCOLSEP* = "0.3ex"  
  This parameter sets the left and right space between braces or brackets and attribute names or values.  
  *string* must contain a \LaTeX{} length expression.

- **doc-options string**  
  default value: value of *DOC-OPTIONS* = "a4wide"  
  This parameter sets the \LaTeX{} \texttt{documentstyle} options if \texttt{latex-header-p} is \texttt{T}.  
  *string* must be a string consisting of the names of zero, one or more valid \LaTeX{} document styles (separated by commas).  
  Possible document styles are "a4", "a4wide", "11pt", "12pt", "leqno", "fleqn", "twoside", "twocolumn", "titlepage" etc.  
  and PostScript™ font styles "avantgarde", "bookman", "chancery", "ncs", "palatino" and "times".

- **mathmode string**  
  default value: value of *MATHMODE* = "displaymath"  
  This parameter sets the \LaTeX{} \texttt{display mode} for feature structures.  
  It must be a string consisting of the name of a \LaTeX{} or user defined math mode environment name, e.g., "math", "displaymath" or "equation".

- **typestyle style**  
  default value: value of *TYPESTYLE* = :infix  
  possible values: { :infix | :prefix }  
  If *style* has value :infix, complex type entries will be printed in infix notation (e.g., \texttt{a \& b \& c}).  
  If *style* has value :prefix, complex type entries will be printed in prefix (LISP like) notation (e.g., \texttt{(AND \texttt{a b c})}).

- **print-title-p flag**  
  default value: value of variable *PRINT-TITLE-P* = \texttt{T}  
  possible values: {\texttt{T}|\texttt{NIL}}  
  If *flag* is \texttt{T}, a title with \texttt{type-name} will be printed at the bottom of the feature structure.  
  If *flag* is \texttt{NIL}, no title will be printed.

5.10.4 Hiding the type field while printing

- **global variable *HIDE-TYPES***  
  default value: \texttt{NIL}  
  If *HIDE-TYPES* is set to \texttt{NIL}, functions \texttt{FLP}, \texttt{FGP}, \texttt{FTI}, \texttt{PLP}, \texttt{PGP}, \texttt{PTI}, \texttt{LLP}, \texttt{LG}, and \texttt{LT} print the type names of all feature types.  
  This causes a wider output.  
  If *HIDE-TYPES* is set to \texttt{T}, the type names of the feature types are left out.  
  This causes a smaller output.  
  Example:

\begin{verbatim}
DISC0(43): (SETQ *HIDE-TYPES* \texttt{T}) \texttt{T}
DISC0(44): (PGP 'NOM-SG-AGR)
 [CASE : [GOV : -
         OBL : -]]
 GENDER : []
 NUM : SG]

DISC0(45): (SETQ *HIDE-TYPES* \texttt{NIL}) \texttt{NIL}
DISC0(46): (PGP 'NOM-SG-AGR)
 NOME-SG-AGR [CASE : CASE-VAL [GOV : -
            OBL : -]]
 GENDER : []
 NUM : SG]
\end{verbatim}
6 Editing and Loading \textit{TDL} files

\textit{TDLExtraLight} supports loading type definitions from files. \textit{TDL} files can be written using an ordinary text editor. When \textsc{EMACS} is used, we recommend running it in \textit{fundamental mode} (which can be switched on with the \textsc{EMACS} command \texttt{M-x fundamental-mode}).

A \textit{TDL} file may contain type definitions, template definitions, instance definitions or \textsc{LISP} code (e.g., \textsc{LISP} function definitions) in arbitrary order. Before loading a \textit{TDL} file, the \textit{TDL} reader must be switched on using \texttt{(RON)}. This may also be done within the \textit{TDL} file.

\textsc{COMMON LISP} function \texttt{LOAD \texttt{file-name} [:verbose \texttt{T|NIL]} [:print \texttt{T|NIL]}]) loads either \textsc{LISP} files or \textit{TDL} files or mixed files.

7 Displaying the \textit{TDL} type hierarchy

It is possible to display the \textit{TDL} type hierarchy using the \textsc{ALLEGRO Composer™}. If \textsc{ALLEGRO Composer™} isn't active by default, it is necessary to load it explicitly by \texttt{DISCO(47): (COMPOSER:START-COMPOSER)} (or alternatively :com)

The \textit{TDL} type hierarchy is represented via the \textsc{COMMON LISP Object System} (CLOS) [Keene 89; Steele 90]. Select menu ‘CLOS’ and then submenu ‘Show Class Subclasses’ or ‘Show Class Superclasses’ and choose \texttt{DISCO:::*var*} or any other \textit{TDL} type in a domain, e.g. \texttt{DISCO}. The Composer will show all subclasses (or superclasses) of the specified \textit{TDL} type.

\texttt{DISCO:::*var*} is the top type of domain \texttt{DISCO}. It is important not to forget the domain name which is internally the \textsc{COMMON LISP} package name of the domain package.

An example screen dump of a \textit{TDL} type hierarchy in CLOS is shown in Figure 3.

8 Top level abbreviations

In the \textsc{Franz Allegro Common LISP} version of \textit{TDLExtraLight}, some often used commands are also available as top level abbreviations. The top level command \texttt{:alias} prints a list of available abbreviations:

<table>
<thead>
<tr>
<th>Alias</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>composer</td>
<td>start Allegro Composer</td>
</tr>
<tr>
<td>define-domain</td>
<td>define a TDL domain</td>
</tr>
<tr>
<td>fegramed</td>
<td>initialize Fegramed</td>
</tr>
<tr>
<td>fgp</td>
<td>Fegramed global prototype</td>
</tr>
<tr>
<td>flp</td>
<td>Fegramed local prototype</td>
</tr>
<tr>
<td>fti</td>
<td>Fegramed type instance</td>
</tr>
<tr>
<td>lgp</td>
<td>LaTeX global prototype</td>
</tr>
<tr>
<td>llp</td>
<td>LaTeX local prototype</td>
</tr>
<tr>
<td>lti</td>
<td>LaTeX type instance</td>
</tr>
<tr>
<td>pgp</td>
<td>print global prototype</td>
</tr>
<tr>
<td>plp</td>
<td>print local prototype</td>
</tr>
<tr>
<td>pti</td>
<td>print type instance</td>
</tr>
<tr>
<td>roff</td>
<td>switch TDL reader OFF</td>
</tr>
<tr>
<td>ron</td>
<td>switch TDL reader ON</td>
</tr>
</tbody>
</table>

\texttt{:composer}, \texttt{:define-domain} and \texttt{:fegramed} may also be abbreviated by \texttt{:com}, \texttt{:def} and \texttt{:feg}. All top level commands take the same parameters as the corresponding \textit{TDL-LISP} functions described in the sections before. Top level commands can only be used in the interactive mode of \textsc{LISP}, but not in \textit{TDL} or \textsc{LISP} source files.
Figure 3: A \textit{TDC} type hierarchy in CLOS

Important Note: Parameters of top level commands should not be quoted. Example:
\texttt{DISCO(48): (PGP ’agr-en-type :label-hide-list ’(GOV OBL))}
but
\texttt{DISCO(49): :PGP agr-en-type :label-hide-list (GOV OBL) :ron, :roff, :composer and :fegramed don’t take any parameter.}
In addition to these \textit{TDC} specific commands, the user may define its own abbreviations. Details are described in the \texttt{FRANZ ALLEGRO COMMON LISP manual}.

9 Sample session

\texttt{USER(1): (load-system "tdl-el")}

; Fast loading ... ...

\texttt{Welcome to DISCO's Type Definition Language TDL-el.}

\texttt{USER(2): :def :disco}

\texttt{DISCO-TDL-Reader is on.}
\texttt{*<DOMAIN DISCO>
28

DISCO(3): (SETQ *VERBOSE-TYPE-DEFINITION-P* NIL)
NIL

DISCO(4): 1. a simple type definition:
? case-val := [OBL, GOV] :doc "a very simple type"
   :author "trick".
#type[CASE-VAL]

DISCO(5): (PGP)
CASE-VAL [GOV : []
   OBL : []]

DISCO(6): 2. type definition using single inheritance and coreferences:
? nom-dat-type := [CASE case-val:[GOV #1,
   OBL #1]].
#type[NOM-DAT-TYPE]

DISCO(7): (PGP 'nom-dat-type)
NOM-DAT-TYPE [CASE : CASE-VAL [GOV : %1 =[]
   OBL : %1]]

DISCO(8): 3. build an instance of type nom-date-type
! nom-dat-type:[CASE case-val:[GOV +]].
#instance[NOM-DAT-TYPE8780]
#$<TDL::FEATURE-STRUCTURE-INFO> @ #x7f706

DISCO(9): 4. type definition using multiple inheritance (which is only possible
   on toplevel) and disjunction (which is NOT allowed on toplevel):
? num-sing-type := [NUM %1].
#type[NUM-SING-TYPE]

DISCO(10): 5 pers-type := [PERS {1,2,3}] :doc "contains a disjunction".
#type[PERS-TYPE]

DISCO(11): 6 multi-inh:=(num-sing-type,pers-type):[pers 2] :doc "multiple inheritance".
#type[MULTI-INH]

DISCO(12): (PLP)
MULTI-INH [PERS : 2]

DISCO(13): (PGP)
MULTI-INH [NUM : SG
   PERS : 2]

DISCO(14): 5. lists:
? l-type := [LIST-SLOT *VAR*:[A %c "hi"], <>, %c].
#type[L-TYPE]

DISCO(15): (PGP)
L-TYPE [LIST-SLOT : LIST [*REST : LIST [*REST : END]
   *FIRST : "hi"]
 *FIRST : *[END]
  *FIRST : [A : "hi"]]

DISCO(16): 6. distributed disjunction:
? dd-type := [a %1(1,2,3),
   b %1("one", "two", "three")].
#type[DD-TYPE]

DISCO(17): (PGP)
DD-TYPE [B : {#1 "one" "two" "three" }
      A : {#1 1 2 3 }]

DISCO(18): ? dd-type2::dd-type:[a 2]
     :doc "2 at attribute a triggers value 'two' at attribute b."
#type[DD-TYPE2]

DISCO(19): (PGP)
DD-TYPE2 [B : "two"
      A : 2]

DISCO(20): 7. functional constraints:
? f-type := [x #x, y #y, result %+(x,y)].
#type[F-TYPE]

DISCO(21): (PGP)
F-TYPE [RESULT : %1 =[]
     Y : %2 =[]
     X : %3 =[]]

FUNCTIONAL-CONSTRAINTS:
%1 = (+ %3 %2)

DISCO(22): ! f-type:[x 1, y 5].
#instance[F-TYPE861]
#<TLR::FEATURE-STRUCTURE-INFON 0 #xc86a8e>

DISCO(23): 8. template definitions:
? a-b-template($attrib, $value)::=VAR*:[$attrib $value, FLAG +].
#template[A-B-TEMPLATE]

DISCO(24): 9. template expansion:
? a-b-in-type:=b a-b-template($attrib PHON, $value "hi").
#type[A-B-IN-TYPE]

DISCO(25): (PGP)
A-B-IN-TYPE [X : [FLAG : +
              PHON : "hi"]]

DISCO(26): 10. negated coreferences:
? neg-coref-type:=[a #1, b #2, c "#(1,2)]
#type[NEG-COREF-TYPE]

DISCO(27): (PLP)
NEG-COREF-TYPE [C : (-%2 -%1) *[]
     B : %2 =[]
     A : %1 =[]]

DISCO(28): 11. define a LISP function and use it in a FS:
(DEFUN strcat (krest args)
   (APPLY #*CONCATENATE 'STRING args))
STRCAT
DISCO(29): ? app:=[a #2 "horn", b #1 "Ein", c %strcat(#1,#2,"haus")]
#type[APP]
10 TDLExtraLight syntax

The next pages contain the TDLExtraLight syntax in extended BNF (Backus-Naur Form). Terminal characters are printed in \textbf{bold} style. Nonterminal symbols are printed in \textit{italic} style. There are three grammars, one for type definitions, one for instance definitions and one for template definitions. Each grammar starts with the \texttt{start} production. The metasymbols \[ ] , \{ \}, \( \), \* and + in extended BNF have the following meaning:

<table>
<thead>
<tr>
<th>metasymbols</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ (expression) ]</td>
<td>one optional expression</td>
</tr>
<tr>
<td>[ (expression)</td>
<td>(expression)</td>
</tr>
<tr>
<td>{ (expression)</td>
<td>(expression)</td>
</tr>
<tr>
<td>{ (expression) *</td>
<td>{ (expression) } +</td>
</tr>
<tr>
<td>{ (expression) } +</td>
<td>n successive expressions, where ( n \in {1,2,\ldots} )</td>
</tr>
</tbody>
</table>

10.1 Type definitions

\begin{align*}
\texttt{start} & ::= \texttt{? type-name } ::= \texttt{type-def \{type-opt\}*}. \\
\texttt{type-def} & ::= \{ \texttt{complex-def} \mid \texttt{template-call} \} \\
\texttt{complex-def} & ::= \#\texttt{variable} \{ \texttt{type-name} \mid \{ \texttt{type-name} \}^* \texttt{type-name} \} : \} \\
& \quad \left[ \left[ \texttt{feature-descr} \right]^* \texttt{feature-descr} \right] \\
\texttt{feature-descr} & ::= \texttt{attribute-name [value]} \\
\texttt{value} & ::= \#\texttt{variable} \left[ ^\#\left( \texttt{variable} \right)^{, \texttt{variable}} \right] \texttt{val} \\
\texttt{val} & ::= ^* \texttt{atom} \mid \texttt{conjunction-val} \mid \\
& \quad \left[ \texttt{%disj-index} \right] \left\{ \right. \left. \texttt{value} \right)^* \texttt{value} \mid \\
& \quad \left< \left. \left[ \texttt{value} \right)^* \texttt{value} \right> \mid \\
& \quad \texttt{%function-name} \left( \{ \texttt{value} \}^* \texttt{value} \right) \mid \\
& \quad \texttt{template-call} \right] \\
\texttt{conjunction-val} & ::= \left. \left[ \texttt{type-name} \right. \right] \left[ \left[ \texttt{feature-descr} \right]^* \texttt{feature-descr} \right] \\
\texttt{template-call} & ::= \texttt{0template-name} \left[ \left[ \left[ \texttt{param-spec} \right)^* \texttt{param-spec} \right] \right] \\
\texttt{param-spec} & ::= \texttt{symbol [value]} \\
\texttt{atom} & ::= \left. \right\{ \texttt{string} \mid \texttt{symbol} \mid \texttt{[-]integer} \right\} \\
\end{align*}
10.2 Instance definitions

\[
\text{variable ::= } \{ \text{symbol | integer } \} \\
\text{attribute-name ::= symbol} \\
\text{type-name ::= symbol} \\
\text{function-name ::= symbol} \\
\text{template-name ::= symbol} \\
\text{disj-index ::= integer} \\
\text{type-opt ::= } \{ \text{:author string | :date string | :doc string | :status statuskey} \} \\
\text{integer ::= } \{0|1|2|3|4|5|6|7|8|9|0}^+ \\
\text{symbol ::= symbol-begin-char \(\{\text{symbol-continue-char}\}\)^*} \\
\text{symbol-begin-char ::= } \{a-zA-Z\}^+ \text{-}\{a-zA-Z\} \}^* \\
\text{symbol-continue-char ::= } \{a-zA-Z\}^0|9-|+|-|}\}^* \\
\text{string ::= "\{any character except \"\}"}^* \\
\]

10.3 Template definitions

\[
\text{start ::= } ? \text{template-name ( } [\{\text{param-spec}\}^* \text{ param-spec} ] ) ::= \text{conjunction-val } \{\text{template-opt}\}^* \\
\text{template-opt ::= } \{ \text{:author string | :date string | :doc string} \} \\
\]
References


REFERENCES


DFKI Publications

The following DFKI publications or the list of all published papers so far can be ordered from the above address.
The reports are distributed free of charge except if otherwise indicated.

DFKI Research Reports

RR-92-34
Philipp Hanschke: Terminological Reasoning and Partial Inductive Definitions
23 pages

RR-92-35
Manfred Meyer: Using Hierarchical Constraint Satisfaction for Lathe-Tool Selection in a CIM Environment
18 pages

RR-92-36
Franz Baader, Philipp Hanschke: Extensions of Concept Languages for a Mechanical Engineering Application
15 pages

RR-92-37
Philipp Hanschke: Specifying Role Interaction in Concept Languages
26 pages

RR-92-38
Philipp Hanschke, Manfred Meyer: An Alternative to H-Subsumption Based on Terminological Reasoning
9 pages

RR-92-40
Philipp Hanschke, Knut Hinkelmann: Combining Terminological and Rule-based Reasoning for Abstraction Processes
17 pages

RR-92-41
Andreas Lux: A Multi-Agent Approach towards Group Scheduling
32 pages

RR-92-42
John Nerbonne: A Feature-Based Syntax/Semantics Interface
19 pages

RR-92-43
Christoph Klauck, Jakob Mauss: A Heuristic driven Parser for Attributed Node Labeled Graph Grammars and its Application to Feature Recognition in CIM
17 pages

RR-92-44
Thomas Rist, Elisabeth André: Incorporating Graphics Design and Realization into the Multimodal Presentation System WIP
15 pages

RR-92-45
Elisabeth André, Thomas Rist: The Design of Illustrated Documents as a Planning Task
21 pages

RR-92-46
Elisabeth André, Wolfgang Finkler, Winfried Graf, Thomas Rist, Anne Schauder, Wolfgang Wahlster: WIP: The Automatic Synthesis of Multimodal Presentations
19 pages

RR-92-47
Frank Bomarius: A Multi-Agent Approach towards Modeling Urban Traffic Scenarios
24 pages

RR-92-48
Bernhard Nebel, Jana Koehler: Plan Modifications versus Plan Generation A Complexity-Theoretic Perspective
15 pages

RR-92-49
Christoph Klauck, Ralf Legleitner, Ansgar Bernardi: Heuristic Classification for Automated CAPP
15 pages

RR-92-50
Stephan Busemann: Generierung natürlicher Sprache
61 Seiten
RR·92·51
Hans-Jürgen Bückerlert, Werner Nutt:
On Abduction and Answer Generation through
Constrained Resolution
20 pages

RR·92·52
Mathias Bauer, Susanne Biundo, Dietmar Dengler,
Jana Koehler, Gabriele Paul: PHI - A Logic-Based
Tool for Intelligent Help Systems
14 pages

RR·92·53
Werner Stephan, Susanne Biundo:
A New Logical Framework for Deductive Planning
15 pages

RR·92·54
Harold Boley: A Direkter Semantic Characterization
of RELFUN
30 pages

RR·92·55
John Nerbonne, Joachim Laubsch, Abdel Kader
Diagne, Stephan Oepen: Natural Language
Semantics and Compiler Technology
17 pages

RR·92·56
Armin Laux: Integrating a Modal Logic of
Knowledge into Terminological Logics
34 pages

RR·92·58
Franz Baader, Bernhard Hollunder:
How to Prefer More Specific Defaults in
Terminological Default Logic
31 pages

RR·92·59
Karl Schlechta and David Makinson: On Principles
and Problems of Defeasible Inheritance
13 pages

RR·92·60
Karl Schlechta: Defaults, Preorder Semantics and
Circumscription
19 pages

RR·93·02
Wolfgang Wahlster, Elisabeth André, Wolfgang
Finkler, Hans-Jürgen Profillich, Thomas Riss:
Plan-based Integration of Natural Language and
Graphics Generation
50 pages

RR·93·03
Franz Baader, Berhard Hollunder, Bernhard Nebel,
Hans-Jürgen Profillich, Enrico Franconi:
An Empirical Analysis of Optimization Techniques
for Terminological Representation Systems
28 pages

RR·93·04
Christoph Klauck, Johannes Schwagereit:
GGD: Graph Grammar Developer for features in
CAD/CAM
13 pages

RR·93·05
Franz Baader, Klaus Schulte: Combination Tech­
niques and Decision Problems for Disunification
29 pages

RR·93·06
Hans-Jürgen Bückerlert, Bernhard Hollunder, Armin
Laux: On Skolemization in Constrained Logics
40 pages

RR·93·07
Hans-Jürgen Bückerlert, Bernhard Hollunder, Armin
Laux: Concept Logics with Function Symbols
36 pages

RR·93·08
Harold Boley, Philipp Hanschke, Knut Hinkelmann,
Manfred Meyer: COLAB: A Hybrid Knowledge
Representation and Compilation Laboratory
64 pages

RR·93·09
Philipp Hanschke, Jörg Würtz:
Satisfiability of the Smallest Binary Program
8 Seiten

RR·93·10
Martin Buchheit, Francesco M. Donini, Andrea
Schaerf: Decidable Reasoning in Terminological
Knowledge Representation Systems
35 pages

RR·93·11
Bernhard Nebel, Hans-Jürgen Buerckerl:
Reasoning about Temporal Relations:
A Maximal Tractable Subclass of Allen's Interval
Algebra
28 pages

RR·93·12
Pierre Sablayrolles: A Two-Level Semantics for
French Expressions of Motion
51 pages

RR·93·13
Franz Baader, Karl Schlechta:
A Semantics for Open Normal Defaults via a
Modified Preferential Approach
25 pages

RR·93·14
Joachim Niehren, Andreas Podelski, Ralf Treinen:
Equational and Membership Constraints for Infinite
Trees
33 pages
RR-93-15
Frank Berger, Thomas Fehrle, Kristof Klöckner, Volker Schölles, Markus A. Thies, Wolfgang Wahlster: PLUS - Plan-based User Support
Final Project Report
33 pages

RR-93-16
Geri Smolka, Martin Henz, Jörg Würz: Object-Oriented Concurrent Constraint Programming in Oz
17 pages

RR-93-18
Klaus Schild: Terminological Cycles and the Propositional $\mu$-Calculus
32 pages

RR-93-20
Franz Baader, Bernhard Hollunder: Embedding Defaults into Terminological Knowledge Representation Formalisms
34 pages

RR-93-22
17 pages

RR-93-23
Andreas Dengel, Ottmar Lutzy: Comparative Study of Connectionist Simulators
20 pages

RR-93-24
Rainer Hoch, Andreas Dengel: Document Highlighting — Message Classification in Printed Business Letters
17 pages

RR-93-26
Jörg P. Müller, Markus Pischel: The Agent Architecture InteRRaP: Concept and Application
99 pages

RR-93-27
Hans-Ulrich Krieger: Derivation Without Lexical Rules
33 pages

RR-93-28
Hans-Ulrich Krieger, John Nerbonne, Hannes Pirker: Feature-Based Allomorphy
8 pages

RR-93-33
Bernhard Nebel, Jana Koehler: Plan Reuse versus Plan Generation: A Theoretical and Empirical Analysis
33 pages

RR-93-34
Wolfgang Wahlster: Verbmobil Translation of Face-To-Face Dialogs
10 pages

DFKI Technical Memos

TM-91-13
Knut Hinkelmann: Forward Logic Evaluation: Developing a Compiler from a Partially Evaluated Meta Interpreter
16 pages

TM-91-14
Rainer Bleisinger, Rainer Hoch, Andreas Dengel: ODA-based modeling for document analysis
14 pages

TM-91-15
Stefan Busemann: Prototypical Concept Formation An Alternative Approach to Knowledge Representation
28 pages

TM-92-01
Lijuan Zhang: Entwurf und Implementierung eines Compilers zur Transformation von Werkstückrepräsentationen
34 Seiten

TM-92-02
Achim Schupeta: Organizing Communication and Introspection in a Multi-Agent Blocksworld
32 pages

TM-92-03
Mona Singh: A Cognitiv Analysis of Event Structure
21 pages

TM-92-04
Jürgen Müller, Jörg Müller, Markus Pischel, Ralf Scheidhauer: On the Representation of Temporal Knowledge
61 pages

TM-92-05
Franz Schmalhofer, Christoph Globig, Jörg Thoben: The refitting of plans by a human expert
10 pages

TM-92-06
Otto Kühn, Franz Schmalhofer: Hierarchical skeletal plan refinement: Task- and inference structures
14 pages

TM-92-08
Anne Kilger: Realization of Tree Adjoining Grammars with Unification
27 pages

TM-93-01
Otto Kühn, Andreas Birk: Reconstructive Integrated Explanation of Lathe Production Plans
20 pages

TM-93-02
Pierre Sablayrolles, Achim Schupeta: Conflict Resolving Negotiation for COoperative Schedule Management
21 pages
DFKI Documents

D-92-15
DFKI Wissenschaftlich-Technischer Jahresbericht
1991
130 Seiten

D-92-16
Judith Engelkamp (Hrsg.): Verzeichnis von Softwarekomponenten für natürlichsprachliche Systeme
189 Seiten

D-92-17
254 pages
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-92-18
Klaus Becker: Verfahren der automatisierten Diagnose technischer Systeme
109 Seiten

D-92-19
Werner Stein: Indexing Principles for Relational Languages Applied to PROLOG Code Generation
80 pages

D-92-20
Michael Herfert: Parsen und Generieren der Prolog-artigen Syntax von RELFUN
51 Seiten

D-92-21
Anne Schauder: Incremental Syntactic Generation of Natural Language with Tree Adjoining Grammars
57 pages

D-92-22
Werner Stein: Indexing Principles for Relational Languages Applied to PROLOG Code Generation
80 pages

D-92-23
Michael Herfert: Parsen und Generieren der Prolog-artigen Syntax von RELFUN
51 Seiten

D-92-24
Jürgen Müller, Donald Steiner (Hrsg.): Kooperierende Agenten
78 Seiten

D-92-25
Martin Buchheit: Klassische Kommunikations- und Koordinationsmodelle
31 Seiten

D-92-26
Enno Tolzmann: Realisierung eines Werkzeugauswahlmoduls mit Hilfe des Constraint-Systems CONTAX
28 Seiten

D-92-27
Martin Harm, Knut Hinkelmann, Thomas Labisch: Integrating Top-down and Bottom-up Reasoning in COLAB
40 pages

D-92-28
Klaus-Peter Gores, Rainer Bleisinger: Ein Modell zur Repräsentation von Nachrichtentypen
56 Seiten

D-93-01
Hanscharm Frühwirth: Terminologia
Ppilipp Hanschke, Thom Frühwirth: Terminological Reasoning with Constraint Handling Rules
12 pages

D-93-02
Gabriele Schmidt, Frank Peters,
Gernod Laufkötter: User Manual of COKAM+
23 pages

D-93-03
Stephan Busemann, Karin Harbusch (Eds.): DFKI Workshop on Natural Language Systems: Reusability and Modularity - Proceedings
74 pages

D-93-04
Klaus-Peter Gores, Rainer Bleisinger: Ein erwartungsgesteuerter Koordinator zur partiellen Textanalyse
53 Seiten

D-93-05
Hanscharm Frühwirth: Terminologia
Philipp Hanschke, Thom Frühwirth: Terminological Reasoning with Constraint Handling Rules
12 pages

D-93-06
Jürgen Müller (Hrsg.): Beiträge zum Gründungsworkshop der Fachgruppe Verteilte Künstliche Intelligenz Saarbrücken 29.-30. April 1993
235 Seiten
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-93-07
Klaus-Peter Gores, Rainer Bleisinger: Ein erwartungsgesteuerter Koordinator zur partiellen Textanalyse
53 Seiten

D-93-08
Thomas Kieninger, Rainer Hoch: Ein Generator mit Anfragesystem für strukturierte Wörterbücher zur Unterstützung von Texterkennung und Textanalyse
125 Seiten

D-93-09
35 pages