\textit{TDL ExtraLight} 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 \textit{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 TDLExtraLight ......................................... 4

3 Starting TDLExtraLight ...................................... 6

4 Syntax and semantics of TDLExtraLight .................... 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 FEMALED .............................. 20
    5.10.3 Printing pretty with TDL2HPrX .................... 22
    5.10.4 Hiding the type field while printing .............. 25

6 Editing and Loading TDL files ................................ 26

7 Displaying the TDL type hierarchy .......................... 26

8 Top level abbreviations .................................... 26

9 Sample session ............................................. 27

10 TDLExtraLight 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 TDL ExtraLight 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 [Döre & 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 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 [Döre & 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. [Knight 89] presents an overview to the different fields in computer science which make use of the notion of unification.

2) Almost every theory/formalism use a different notion when referring to feature structures: *structures* in LFG [Bresnan 82], *feature bundles* or *feature matrices* in GPSG [Gaarder 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 83], *attribute-value matrices* in HPSG [Pollard & Sag 87] or *tags* in PATR-II [Shieber et al. 83].

3) For 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 TDL ExtraLight. 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 & Naas 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 [Daelmans 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 & Mosher 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 & Sag 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, declarateness 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 Kays 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. TDLC 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

---

5Moving 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.

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.
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 $\text{TDLExtraLight}$ is currently concerned with. The DISCO
grammar consists of 650 type specifications written in $\text{TDL}$ and is the largest HPSG grammar for
German [Netter 93]. Input given to $\text{TDL}$ 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 $\text{TDLExtraLight}$ 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 commu-
nicate 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 $\text{TDL}$ which gives back a simplified conjunction of the types.

$\text{TDLExtraLight}$ permits type definitions with multiple inheritance and the inheritance of functional
dependencies. In addition, $\text{TDL}$ 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 $\text{TDLExtraLight}$.

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, $\text{TDL}$ 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,$\text{TDLExtraLight}$ does not support disjunctive or even negated type specifications, although they
can be written on the feature constraint level.\(^7\)

$\text{TDLExtraLight}$ 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 $\text{TDL2}\text{\LaTeX}$ package, transforming typed feature structures into $\text{\LaTeX}$ code

- a number of software switches, which influence the behaviour of the whole system

Grammars and lexicons written in $\text{TDL}$ 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 $\text{TDLExtraLight}$ mentioned above are no longer present in its successor $\text{TDL}$ which will be
available in spring '93. The new system is completely redesigned and re-implemented, includes advanced features, is
fully incremental and has better performance, although its expressive power increases massively. Moreover, the new
$\text{TDL}$ makes a parameterized expansion mechanism available to the user (this is needed by a parser or a generator
to work efficiently) and supports a special form of non-monotonic inheritance (see [Krieger & Schäfer 93] for a full
system overview).
3 Starting \texttt{TDLExtraLight}

1. Start \texttt{COMMON LISP}.

2. \texttt{(load-system "tdl-1")} loads the necessary parts of \texttt{TDLExtraLight} such as the unifier (UDiNE), type definition reader, feature editor (FEGRAME), type hierarchy management and the \texttt{TDL2H} interface. The portable system definition facility \texttt{DEFSYSTEM} is described in [Kantrowitz 91].

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

\begin{verbatim}
Welcome to DISCO's Type Definition Language TDL-el.
USER(1): 
\end{verbatim}

4. To start the \texttt{TDLExtraLight} reader and create a domain for grammar types and symbols, the user should type\footnote{Also \texttt{USER} assertions are possible.}

\begin{verbatim}
(DEFINE-DOMAIN :DISCO) \text{(or abbreviated: \texttt{DEF :disco})}
\end{verbatim}

Any other keyword symbol or string may be chosen instead of \texttt{DISCO} except \texttt{TDL} and the usual \texttt{COMMON LISP} package names like \texttt{COMMON-LISP} or \texttt{USER}. The name \texttt{TDL} is preserved for internal functions and variables. It is possible to define several domains and to change between them by using function \texttt{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 \texttt{LOAD}. Examples:

\begin{verbatim}
DISCO(2): ? my_first_type := [case nom, num 1].
DISCO(3): (LOAD "grammar") \text{(or abbreviated: \texttt{LOAD "grammar"})}
\end{verbatim}

6. \texttt{DISCO(4): (EXIT)} \text{(or abbreviated: \texttt{EX})}

exits Lisp and \texttt{TDLExtraLight}.

The \texttt{EMACS} command \texttt{C-x C-c} kills the Lisp and \texttt{EMACS} process.

4 Syntax and semantics of \texttt{TDLExtraLight}

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

The BNF (Backus-Naur Form) of the \texttt{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 \texttt{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 \texttt{TDL} definitions in a file. Some examples are shown in section 9.

4.1 Type definitions

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

\begin{verbatim}
? (type-name) := (type-def) [(options)].
\end{verbatim}

\texttt{(type-name)} is a symbol, the name of the type to be defined. \texttt{(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). \texttt{(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

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

The definition results in the structure

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

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

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

results in the structure

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

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:

? agr-plural-type := [AGR person-number-type: [NUMBER plural]].

This definition results in the structure

\[
\begin{bmatrix}
\text{agr-plural-type} \\
\text{AGR} \\
\text{person-number-type} \\
\text{PERSON} [ ] \\
\text{NUMBER plural}
\end{bmatrix}
\]

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

This definition results in the structure

\[
\begin{bmatrix}
\text{pl-type} \\
\text{PERSON [ ]} \\
\text{NUMBER plural}
\end{bmatrix}
\]

This feature structure is called the global prototype of \textit{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 \textit{pl-type} is

\[
\begin{bmatrix}
\text{pl-type} \\
\text{NUMBER plural}
\end{bmatrix}
\]

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 \texttt{*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 \textit{number-type}, \textit{person-type} and \textit{gender-type} are defined as follows:

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

Then the \texttt{TDLExtraLight} type definition

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

would result in the following structure:

\[
\begin{bmatrix}
\text{mas-2-type} \\
\text{GENDER mas} \\
\text{PERSON 2} \\
\text{NUMBER [ ]}
\end{bmatrix}
\]

4.1.5 Coreferences

Coreferences indicate information sharing between feature structures. In \texttt{TDLExtraLight}, 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 ].
results in the following structure:

\[
\begin{array}{c}
\text{share-pn} \\
\text{SYN} \left[ \text{person-number-type} \right] \\
\text{SEM} \left[ \text{} \right]
\end{array}
\]

### 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

\[ ^\neg (a_1, a_2, \ldots, a_n) \]

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 \text{TDL ExtraLight} definition

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

would result in the following structure:

\[
\begin{array}{c}
\text{give} \\
\text{RELN give} \\
\text{GIVER } ^\neg (1,2)[[]] \\
\text{GIVEN } #1 \\
\text{GIVEE } #2
\end{array}
\]

### 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 \text{TDL ExtraLight} is that disjunctions are not allowed at the top level of a type definition.

Example for disjunctions in a type definition:

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

The resulting feature structure is

\[
\begin{array}{c}
\text{person-1-or-2} \\
\text{SYN} \left[ \text{person-number-type} \right] \\
\text{PERSON 1} \\
\text{NUMBER } [1] \\
\text{person-number-type} \\
\text{PERSON 2} \\
\text{NUMBER } [1] \\
\end{array}
\]

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
  [person-number-type
    [SYN PERSON [1]
    [NUMBER [2]]]]
```

4.1.8 Distributed disjunctions

A very useful feature of *TDLExtraLight* defined in the underlying unification system UDLfNE 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 *TDLExtraLight* expression:

? season-trigger := [SEASON %1{"spring", "summer", "fall", "winter"},
   NUMBER %1{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

%1{a_i_1, ..., a_i_n},

where i is an integer number, the disjunction index for each group of distributed disjunctions (%1 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_1j (1 ≤ j ≤ n) does not fail, it selects the corresponding alternative b_1j, c_1j, ..., 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 %1{[]}, #1, #2,
           b %1{[c +]}, x:[d #1 g:[m -]], x:[d #2 g:[m +]]].
4.1 Type definitions

\[
\begin{bmatrix}
\text{dis}^2 \\
A \mid 1 \{ \\
\quad \{ g_{M-} \} \\
\quad \{ g_{M+} \} \\
\quad [C +] \\
\quad [x] \\
\quad [x] \\
\quad [D \[ ] \\
\quad [D \[ ] \\
\end{bmatrix}
\]

4.1.9 Negation

The `` sign indicates negation. Example:

? not-mas-type := [GENDER `mas].

The resulting feature structure is

\[
\begin{bmatrix}
\text{not-mas-type} \\
\text{GENDER } ¬\text{mas}
\end{bmatrix}
\]

4.1.10 Lists

In \text{TDLExtraLight}, 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:

? list-it := [LIST < first-element, second, #last >,
\hspace{1em} LAST #last,
\hspace{1em} AN-EMPTY-LIST <>].

The resulting feature structure is

\[
\begin{bmatrix}
\text{list-it} \\
\text{list} \\
\quad *\text{FIRST first-element} \\
\quad \{ \text{list} \} \\
\quad *\text{FIRST second} \\
\quad \{ \text{list} \} \\
\quad *\text{REST} \\
\quad \text{list} \\
\quad *\text{FIRST} \[ ] \\
\quad *\text{REST} \\
\quad *\text{end} \\
\quad \text{LAST} \[ ] \\
\quad \text{AN-EMPTY-LIST} \ *\text{end}
\end{bmatrix}
\]

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 \text{TDL} system.

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

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

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

\[
[ \text{add-prefix} \\
\text{WORD 1} \\
\text{PREFIX 2} \\
\text{WHOLE 3} ]
\]

\text{Functional Constraints:} \\
3 = \text{concatenate(string 2 1)}

The evaluation of functional constraints will be residuated until all parameters are instantiated [Ait-Kaci & Nasr 86b; Smolka 91]. Evaluation can be enforced by using the function \text{EVAL-CONSTRAINTS} of the \text{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 \text{call} simply means syntactic replacement of a template name by its definition and possibly given parameters.

The syntax of template call is

\[
@\text{(template name) } ((\text{template parameter pairs})]
\]

where a \text{(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, TDLC 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 TDLC control system. \text{options} for type definitions are the optional keywords \text{author, doc, date} and \text{status}. When specified, a value must follow the corresponding keyword.

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

The \text{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 \text{status} keyword is given (valid values: see rule \text{statuskey} in the BNF syntax
4.2 Template definitions

Templates in TDLEExtraLight 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 TDLEExtraLight template definition is

\[
\text{? } \langle \text{template-name} \rangle \left( \left[ \langle \text{template parameter pairs} \rangle \right] \right) := \langle \text{template-body} \rangle \left[ \langle \text{options} \rangle \right].
\]

where a \langle \text{template parameter pair} \rangle 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. \langle \text{template-body} \rangle can be a complex description as in type definitions.

Example: The template definition

\[
\text{? } \text{a-template} \left( $\text{inherit } \ast \text{var}, \ $\text{attrib PHON}, \ $\text{value} \right) := \\
$\text{inherit}: \left[ $\text{attrib } \#1 \ $\text{value}, \ 
\text{COPY } \ #1 \right].
\]

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

\[
\text{? } \text{top-level-call} := \text{@a-template}.
\]

is a top-level template call which will result in the feature structure:

\[
\begin{bmatrix}
\text{top-level-call} \\
\text{PHON} \ [ ] \\
\text{COPY} \ [ ]
\end{bmatrix}
\]

while

\[
\text{? } \text{inside-call} := \left[ \text{top-attrib @a-template} \left( $\text{value } \text{"hello"}, \ 
\text{attrib MY-PHON} \right) \right].
\]

is a template call inside a feature type definition which will result in the feature structure:

\[
\begin{bmatrix}
\text{inside-call} \\
\text{TOP-ATTRIB} \left[ \text{MY-PHON } \text{"hello"} \right] \\
\text{COPY } \text{"hello"}
\end{bmatrix}
\]

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

Section 5.7 describes the functions \textbf{DESCRIBE-TEMPLATE} and \textbf{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., 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 TDL extraLight instance definition is

! (type-name) [{instance-body}] [{options}].

or

! (template-call) [{options}].

[{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, PTI and LTI always take the last instance defined with the specified name.

If the :status keyword is given (valid values: see rule 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 use-package. This is done automatically in the function 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 (DEFINE-DOMAIN domain-name [:hide-attributes attribute-list] [:export-symbols symbol-list] [:errorp {T|NIL}])
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:

  ```lisp
  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 redefine will be signaled if a type already exists and is about to be redefined. Example:

  ```lisp
  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 :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, LFP, LLP, FPG, PLP, 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

5.6 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 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:
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-RANGE-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-type#335)

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:
Figure 1: A feature structure type in FEGRAMED

- :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.

- `:filename filename`  
  default value: "type-name-gp.fed", "type-name-lp.fed" or  
  possible values: a string or a LISP path name  
  "instance-name.fed"

Unless `filename` is specified, a filename will be 'computed' from the type name. The file will  
be created by the TDL-FEGRAMED interface in order to communicate the feature structure  
information.

- `:wait flag`  
  default value: NIL
  possible values: {T|NIL}

If `flag` is T, FEGRAMED will wait until the user chooses the return options. If `flag` is NIL,  
FEGRAMED will not wait.

An example screen dump of a feature structure in FEGRAMED is shown in Figure 1.

### 5.10.3 Printing pretty with \texttt{TDL2HFP\texttt{X}}

\texttt{TDL2HFP\texttt{X}} is a tool which generates \texttt{HFP\texttt{X}} compatible high-quality output of \texttt{TDL} feature  
structure types.

- function (\texttt{LLP [type-name [l-options]]})

  LLP starts \texttt{TDL2HFP\texttt{X}} with the \texttt{LOCAL} \texttt{PROTOTYPE} of the feature structure with name `type-name`.  
  If no `type-name` is specified, LLP takes the prototype of the \texttt{last} `type-name` defined before evaluating LLP. The \texttt{LOCAL} \texttt{PROTOTYPE} contains only the \texttt{local} information given in the  
definition of `type-name`. Example:

  \texttt{DISCO(40): ( LLP 'agr-en-type :fontsize "small"  
    :doc-options "a4wide,palatino")}

- function (\texttt{LGP [type-name [l-options]]})

  LGP starts \texttt{TDL2HFP\texttt{X}} with the \texttt{GLOBAL} \texttt{PROTOTYPE} of the feature structure with name  
  `type-name`. If no `type-name` is specified, LGP takes the prototype of the \texttt{last} `type-name` defined  
  before evaluating LGP. The \texttt{GLOBAL} \texttt{PROTOTYPE} contains \texttt{all} information that can be inferred  
  for `type-name` and its supertypes. Example:

  \texttt{DISCO(41): ( LGP 'agr-en-type :mathmode "equation"  
    :doc-options "leqno")}

- function (\texttt{LTI instance-name [l-options]})

  LTI starts \texttt{TDL2HFP\texttt{X}} with the feature structure of instance `instance-name`. Example:

  \texttt{DISCO(42): ( LTI 'agr-en-type4335 )}

An example of a complex feature structure generated by \texttt{TDL2HFP\texttt{X}} is shown in Figure 2.  
\texttt{l-options} are the following optional keywords:

- `:filename filename`  
  default value: "type-name-gp", "type-name-lp" or  
  "instance-name"

  Unless `filename` is specified, a filename will be 'computed' from the type name. The filename  
  will be used to generate the \texttt{HFP\texttt{X}} output file.

- `:filepath pathname`  
  default value: value of variable *FILEPATH*

  possible values: a string or a \texttt{COMMON} LISP path name

  `pathname` sets the directory in which the \texttt{HFP\texttt{X}} output file will be created and the shell  
  command `command` will be executed. The value of *FILEPATH* defaults to the \texttt{tmp} directory  
in the user’s home directory.

- `:hide-types flag`  
  default value: value of variable *HIDE-TYPES* = NIL

  possible values: {T|NIL}

  If `flag` is 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 TDCL2TEX

- :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* = "tdl2latex"
  possible values: {NIL | string}
If command is NIL, only the \inputX file will be created and TDL2\inputX will return. If command is a string, TDL2\inputX will start a shell process and execute command with parameter filename. An example for command is the following shell script with name tdl2ps which starts \inputX with the output file of TDL2\inputX and writes PostScript\TM code to the file filename.ps:
  
  #!/bin/sh
  latex $1
dvips $1 -o $1.ps

• :wait flag  
  default value: value of *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, TDL2\inputX 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 \inputX file with \documentstyle etc. will be generated. If flag is NIL, only the \inputX code of the feature structure enclosed in \begin{featurestruct} and \end{featurestruct} will be written to the output file. This is useful for inserting \inputX feature structures into \inputX documents for papers, books etc.

• :align-attributes-p 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* = "normal size"
  possible values: a string
This parameter sets the size of the \inputX feature structures. It must be a string consisting of a valid \inputX font size name, e.g., "tiny", "scriptsize", "footnotesize", "small", "large", "Large", "LARGE", "huge" or "huge".

• :coresize 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 \inputX font size as in :fontsize.

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

• :corestable a-list  
  default value: value of variable *CORETABLE* = ()
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 coreferences 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 \*\texttt{ARRAYSTRETCH}* = 1.1  
  This parameter sets the vertical distance between attribute names or disjunction alternatives.  
  \texttt{number} is a factor which will be multiplied with the standard character height.

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

- **:doc-opts string**  
  *default value:* value of \*\texttt{DOC-OPTIONS}* = "a4wide"  
  This parameter sets the \LaTeX\ \texttt{documentstyle} options if \texttt{:latex-header-p} is \texttt{T}.  
  \texttt{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\textsuperscript{TM} font styles "avantgarde", "bookman", "chancery", "ncs", "palatino" and "times".

- **:mathmode string**  
  *default value:* value of \*\texttt{MATHMODE}* = "displaymath"  
  This parameter sets the \LaTeX{} 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 \*\texttt{TYPESTYLE}* = :infix  
  possible values: \{ :infix \| :prefix \}  
  If \texttt{style} has value :\texttt{infix}, complex type entries will be printed in infix notation (e.g., \( a \land b \lor c \)).  
  If \texttt{style} has value :\texttt{prefix}, complex type entries will be printed in prefix (\texttt{LISP} like) notation (e.g., \( (a \land b) \lor c \)).

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

### 5.10.4 Hiding the type field while printing

- **global variable \*\texttt{HIDE-TYPES}***  
  *default value:* \texttt{NIL}  
  If \*\texttt{HIDE-TYPES}* is set to \texttt{NIL}, \texttt{LPF}, \texttt{FGP}, \texttt{FTI}, \texttt{PLP}, \texttt{PGP}, \texttt{PTI}, \texttt{LLP}, \texttt{LGP} and \texttt{LTI} print the type names of all feature types.  
  This causes a wider output.  
  If \*\texttt{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}
DISCO(43): (SETQ \*HIDE-TYPES* \texttt{T})
\texttt{T}
DISCO(44): (PGP '\texttt{NOM-SG-AGR})
[CASE  : [GOV : -
           OBL  : -]
  GENDER : []
  NUM   : SG]
DISCO(45): (SETQ \*HIDE-TYPES* \texttt{NIL})
\texttt{NIL}
DISCO(46): (PGP '\texttt{NOM-SG-AGR})
NOM-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 EMACS is used, we recommend running it in \textit{fundamental mode} (which can be switched on with the EMACS command \texttt{M-x fundamental-mode}).

A \textit{TDL} file may contain type definitions, template definitions, instance definitions or LISP code (e.g., LISP function definitions) in arbitrary order.

Before loading a \textit{TDL} file, the \textit{TDL} reader must be switched on using \texttt{(RUN)}. This may also be done within the \textit{TDL} file.

\textbf{COMMON LISP function} \texttt{(LOAD file-name [:verbose \{T\|NIL\}] [:print \{T\|NIL\}]})

loads either \textit{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 ALLEGRO COMPOSER\textsuperscript{TM}. If ALLEGRO COMPOSER\textsuperscript{TM} 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 \textbf{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. \textit{DISCO}. The Composer will show all subclasses (or superclasses) of the specified \textit{TDL} type.

\texttt{DISCO::*var*} is the top type of domain \textit{DISCO}. It is important not to forget the domain name which is internally the \textbf{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 FRANZ ALLEGRO \textit{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:

\begin{verbatim}
Alias  Description
-----  ----------
:composer  start Allegro Composer
:define-domain  define a TDL domain
:fegramed  initialize FeGramed
:fgp  FeGramed global prototype
:flp  FeGramed local prototype
:fti  FeGramed type instance
:lgp  LaTeX global prototype
:llp  LaTeX local prototype
:lti  LaTeX type instance
:pgp  print global prototype
:plp  print local prototype
:pti  print type instance
:roff  switch TDL reader OFF
:ron  switch TDL reader ON
\end{verbatim}

\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 \textit{LISP}, but not in \textit{TDL} or \textit{LISP} source files.
Important Note: Parameters of top level commands should not be quoted. Example:

DISCO(48): (PGP 'agr-en-type :label-hide-list ' (GOV OBL))

but

DISCO(49): :PGP agr-en-type :label-hide-list (GOV OBL)

In addition to these TDL specific commands, the user may define its own abbreviations. Details
are described in the Franz Allegro Common Lisp manual.

9 Sample session

USER(1): (load-system "tdl-el")

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

USER(2): :def :disco
DISCO-TDL-Reader is on.
#<DOMAIN DISCO>
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-date-type := [CASE case-val: [GOV #1,
                OBL #1]].
 #type[NOM-DAT-TYPE]

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

DISCO(8): ; 3. build an instance of type nom-date-type
   ! nom-date-type: [CASE case-val: [GOV #1]].
 #instance[NOM-DAT-TYPE6780]
 #<TDL::FEATURE-STRUCTURE-INFO1 @ #xd70706>

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 sg].
 #type[NUM-SING-TYPE]

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

 #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 : LIST [*REST : *END
                      *FIRST : "hi"]
                      *FIRST : *END]
                      *FIRST : [A : "hi"]]]

DISCO(16): ; 6. distributed disjunction:
? dd-type := [a %I{1,2,3},
             b %I{"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]
   #<TDL::FEATURE-STRUCTURE-IMPLEMENTATION @ x86a8e>

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:=[x @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 (&rest args)
     (APPLY #'CONCATENATE 'STRING args))
   STRCAT
   DISCO(29): ; app:=[a #2 "horn", b #1 "Ein", c %strcat(#1,#2,"haus").
   #type[APP]
10 TDCExtraLight Syntax

The next pages contain the TDCExtraLight syntax in extended BNF (Backus-Naur Form). Terminal characters are printed in bold style. Nonterminal symbols are printed in italic style. There are three grammars, one for type definitions, one for instance definitions and one for template definitions. Each grammar starts with the 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>n successive expressions, where n ∈ {0, 1, ...}</td>
</tr>
<tr>
<td>{ (expression) }+</td>
<td>n successive expressions, where n ∈ {1, 2, ...}</td>
</tr>
</tbody>
</table>

10.1 Type definitions

\[
start ::= ? \ type-name ::= type-def \ {type-opt}^*.
\]

\[
type-def ::= \{ \ complex-def \ | \ template-call \ \}
\]

\[
complex-def ::= [\# \ variable] \ [\{ \ type-name \ | \ \{type-name\}^+ \ type-name \ \} \ : \]
\[
\{ \{feature-descr\}^* \ feature-descr \}\]

\[
feature-descr ::= \attribute-name [value]
\]

\[
value ::= [\# \ variable] \ [\~\#( \ variable \ {, \ variable}^* \ ) \ ] \ val
\]

\[
val ::= [\*] \ \{ \ atom \ | \ conjunction-val \ |
\%
disj-index \ \{ \ value \ , \ }^* \ value \ \} \ |
\< \ \{ \ value \ , \ }^* \ value \ \} \ > \ |
\%
function-name \ ( \ \{ \ value \ , \ }^* \ value \ ) \ |
\}
\]

\[
conjunction-val ::= [ \ type-name : ] \ \{ \{feature-descr\}^* \ feature-descr \} \]
\]

\[
\text{template-call ::= } \@ \text{template-name} \ \{ \{ \ param-spec \}^* \ param-spec \} \]
\]

\[
\text{param-spec ::= } \$ \text{symbol} \ [value]
\]

\[
\text{atom ::= } \{ \ \text{string} \ | \ \text{symbol} \ | \ [-] \text{integer} \}
\]
10.2 Instance definitions

\[
\text{variable} ::= \{ \text{symbol} \mid \text{integer} \} \\
\text{attribute-name} ::= \text{symbol} \\
\text{type-name} ::= \text{symbol} \\
\text{function-name} ::= \text{symbol} \\
\text{template-name} ::= \text{symbol} \\
\text{disj-index} ::= \text{integer} \\
\text{type-opt} ::= \{ \text{author string} \mid \text{date string} \mid \text{doc string} \mid \text{status statuskey} \} \\
\text{statuskey} ::= \{ \text{lex-entry} \mid \text{lex-rule} \mid \text{rule} \mid \text{epsilon} \mid \text{root} \mid \text{unknown} \mid \text{multi-word-lexeme} \mid \text{sat-rule} \mid \text{lex-triggered-rule} \mid \text{morph-template} \mid \text{sat-rule-2nd} \} \\
\text{integer} ::= \{ 0|1|2|3|4|5|6|7|8|9|0 \}^+ \\
\text{symbol} ::= \text{symbol-begin-char} \{ \text{symbol-continue-char} \}^+ \\
\text{symbol-begin-char} ::= \{ \text{a-z|A-Z|_|+|-|*} \} \\
\text{symbol-continue-char} ::= \{ \text{a-z|A-Z|0-9|_|+|-|*|8} \} \\
\text{string} ::= "\{ \text{any character except } \}^{*}" \\
\]

10.3 Template definitions

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


REFERENCES


REFERENCES


REFERENCES


