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 *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

Over the last few years, unification-based grammar formalisms have become the predominant paradigm in natural language processing and computational linguistics.[1] 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.[2] 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): $\phi \rightarrow \psi \equiv \neg\phi \vee \psi$.

Other proposals consider the integration of *functional* and *relational dependencies* into the formalism which makes them Turing-complete in general.[3] 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. [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 refering 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].

[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 *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., Aït-Kaci's LOGIN [Aït-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*.[4] 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 & 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, 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] 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 $\mathcal{TDL}$*ExtraLight*

$\mathcal{TDL}$*ExtraLight* is a unification-based grammar development environment to support HPSG-like grammars with multiple inheritance. $\mathcal{TDL}$ 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 $\mathcal{TDL}$*ExtraLight* has started at the end of 1988 and is embedded in the DISCO project of the DFKI. The main motivation behind $\mathcal{TDL}$*ExtraLight* was to make a reliable and robust system fast available to the people in the DISCO project: a type system simply

[4] Moving 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.

[5] On 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 *TDL ExtraLight* is currently concerned with. The DISCO grammar consists of 650 type specifications written in *TDL* and is the largest HPSG grammar for German [Netter 93]. Input given to *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 *TDL ExtraLight* 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 *TDL* which gives back a simplified conjunction of the types.
TDL ExtraLight permits *type definitions with multiple inheritance* and the *inheritance of functional dependencies*. In addition, *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 *TDL ExtraLight*.
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, *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, *TDL ExtraLight* does not support disjunctive or even negated type specifications, although they can be written on the feature constraint level.[7]

TDL ExtraLight 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 *TDL*2LaTeX package, transforming typed feature structures into LaTeX code
- a number of software switches, which influence the behaviour of the whole system

Grammars and lexicons written in *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 *TDL ExtraLight* mentioned above are no longer present in its successor *TDL* 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 *TDL* 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 *TDL ExtraLight*

1. Start COMMON LISP.

2. `(load-system "tdl-el")` loads the necessary parts of *TDL ExtraLight* such as the unifier (UDINE), type definition reader, feature editor (FEGRAMED), type hierarchy management and the *TDL*2LaTeX 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 *TDL ExtraLight* 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 *TDL ExtraLight*.
The EMACS command `C-x C-c` kills the LISP and EMACS process.

4 Syntax and semantics of *TDL ExtraLight*

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

4.1.1 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

$$\begin{bmatrix} \textit{person-number-type} \\ \texttt{PERSON}\ [\,] \\ \texttt{NUMBER}\ [\,] \end{bmatrix}$$

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

$$\begin{bmatrix} \textit{pl-3-phon} \\ \texttt{NUMBER plural} \\ \texttt{PHON}\ \ \texttt{"-en"} \\ \texttt{PERSON 3} \end{bmatrix}$$

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

$$\begin{bmatrix} \textit{agr-plural-type} & \\ \texttt{AGR} & \begin{bmatrix} \textit{person-number-type} \\ \texttt{PERSON}\ [\,] \\ \texttt{NUMBER plural} \end{bmatrix} \end{bmatrix}$$

Now an example for type inheritance at the top level:

```
? pl-type := person-number-type:[NUMBER plural].
```

This definition results in the structure

$$\begin{bmatrix} \textit{pl -type} \\ \texttt{PERSON []} \\ \texttt{NUMBER plural} \end{bmatrix}$$

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

$$\begin{bmatrix} \textit{pl-type} \\ \texttt{NUMBER plural} \end{bmatrix}$$

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 `*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 *TDLExtraLight* type definition

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

would result in the following structure:

$$\begin{bmatrix} \textit{mas-2-type} \\ \texttt{GENDER mas} \\ \texttt{PERSON 2} \\ \texttt{NUMBER []} \end{bmatrix}$$

4.1.5 Coreferences

Coreferences indicate information sharing between feature structures. In *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:

$$\begin{bmatrix} \textit{share-pn} & \\ \texttt{SYN}\ \boxed{1} & \begin{bmatrix} \textit{person-number-type} \\ \texttt{PERSON}\ [\,] \\ \texttt{NUMBER}\ [\,] \end{bmatrix} \\ \texttt{SEM}\ \boxed{1} & \end{bmatrix}$$

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

$$\texttt{\~{}\#(}a_1\texttt{,}\ a_2\texttt{,}\ \ldots\ a_n\texttt{)},$$

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 *TDLExtraLight* definition

```
? give := [RELN give, GIVER ~#(1,2), GIVEN #1, GIVEE #2].
```

would result in the following structure:

$$\begin{bmatrix} \textit{give} \\ \texttt{RELN}\ \ \texttt{give} \\ \texttt{GIVER}\ \neg(\boxed{1},\boxed{2})[\,] \\ \texttt{GIVEN}\ \boxed{1} \\ \texttt{GIVEE}\ \boxed{2} \end{bmatrix}$$

4.1.7 Simple Disjunctions

Disjunctive alternatives are enclosed in braces ({...}) and separated by commata. 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 *TDLExtraLight* is that disjunctions are not allowed at the top level of a type definition.
Example for disjunctions in a type definition:

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

The resulting feature structure is

$$\begin{bmatrix} \textit{person-1-or-2} & \\ \texttt{SYN} & \left\{ \begin{bmatrix} \textit{person-number-type} \\ \texttt{PERSON}\ 1 \\ \texttt{NUMBER}\ [\,] \end{bmatrix} \begin{bmatrix} \textit{person-number-type} \\ \texttt{PERSON}\ 2 \\ \texttt{NUMBER}\ [\,] \end{bmatrix} \right\} \end{bmatrix}$$

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

$$\left[\begin{array}{ll} \textit{person-1-or-2} & \\ \texttt{SYN} & \left[\begin{array}{ll} \textit{person-number-type} & \\ \texttt{PERSON} & \left\{\begin{array}{l}\texttt{1}\\ \texttt{2}\end{array}\right\} \\ \texttt{NUMBER} & [\,] \end{array}\right] \end{array}\right]$$

4.1.8 Distributed disjunctions

A very useful feature of *TDL ExtraLight* 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:

$$\left[\begin{array}{ll} \textit{season-trigger} & \\ \texttt{SEASON} & \$1 \left\{\begin{array}{l}\texttt{"spring"}\\ \texttt{"summer"}\\ \texttt{"fall"}\\ \texttt{"winter"}\end{array}\right\} \\ \texttt{NUMBER} & \$1 \left\{\begin{array}{l}\texttt{1}\\ \texttt{2}\\ \texttt{3}\\ \texttt{4}\end{array}\right\} \end{array}\right]$$

This structure has been generated by the following *TDL ExtraLight* expression:

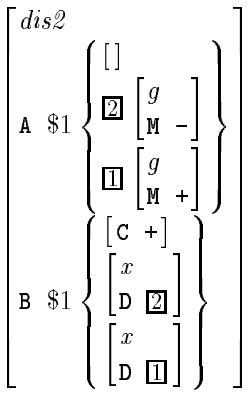
```
? season-trigger := [SEASON %1{"spring", "summer", "fall", "winter"},
                     NUMBER %1{   1    ,    2    ,   3   ,    4    }].
```

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

$$\texttt{\%}i\texttt{\{}a_{i_1}, \ldots, a_{i_n}\texttt{\}},$$

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_{i_j} ($1 \leq j \leq n$) does not fail, it selects the corresponding alternative b_{i_j}, c_{i_j}, ... 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 +]]}].
```

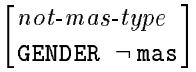


4.1.9 Negation

The ~ sign indicates negation. Example:

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

The resulting feature structure is

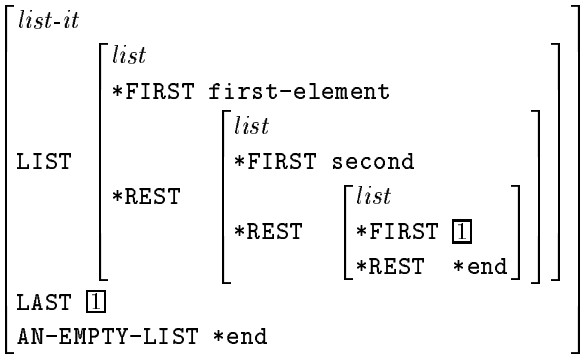


4.1.10 Lists

In *TDL 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:

```
? list-it := [LIST < first-element, second, #last >,
              LAST #last,
              AN-EMPTY-LIST <> ].
```

The resulting feature structure is



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 *TDL* 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:

$$\begin{bmatrix} \textit{add-prefix} \\ \texttt{WORD} \ \boxed{1} \\ \texttt{PREFIX} \ \boxed{2} \\ \texttt{WHOLE} \ \boxed{3} \end{bmatrix}$$

Functional Constraints:

$$\boxed{3} = \texttt{concatenate}(\texttt{string}, \boxed{2}, \boxed{1})$$

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

@⟨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, TDL 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 TDL 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 *TDLExtraLight* 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 *TDLExtraLight* 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 := @a-template.
```

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

$$\begin{bmatrix} \textit{top-level-call} \\ \texttt{PHON}\ \boxed{1} \\ \texttt{COPY}\ \boxed{1} \end{bmatrix}$$

while

```
? inside-call := [top-attrib @a-template ($value "hello",
                                           $attrib MY-PHON)].
```

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

$$\begin{bmatrix} \textit{inside-call} & \\ \texttt{TOP-ATTRIB} & \begin{bmatrix} \texttt{MY-PHON} & \texttt{"hello"} \\ \texttt{COPY} & \texttt{"hello"} \end{bmatrix} \end{bmatrix}$$

⟨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., *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`, `FTI` 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`}])

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(5): (DEFINE-DOMAIN :DISCO :hide-attributes '(SEM))
#<DOMAIN :DISCO>
:DISCO
```

- 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:

```
DISCO2(6): (IN-DOMAIN :DISCO)
#<DOMAIN :DISCO>
: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 *TDL ExtraLight* 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:

```
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)
  ```

- 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 $\mathcal{TDL}$ 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)
  (NUM 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 $\mathcal{TDL}$ 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 *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 supertypes 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:

- `: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:

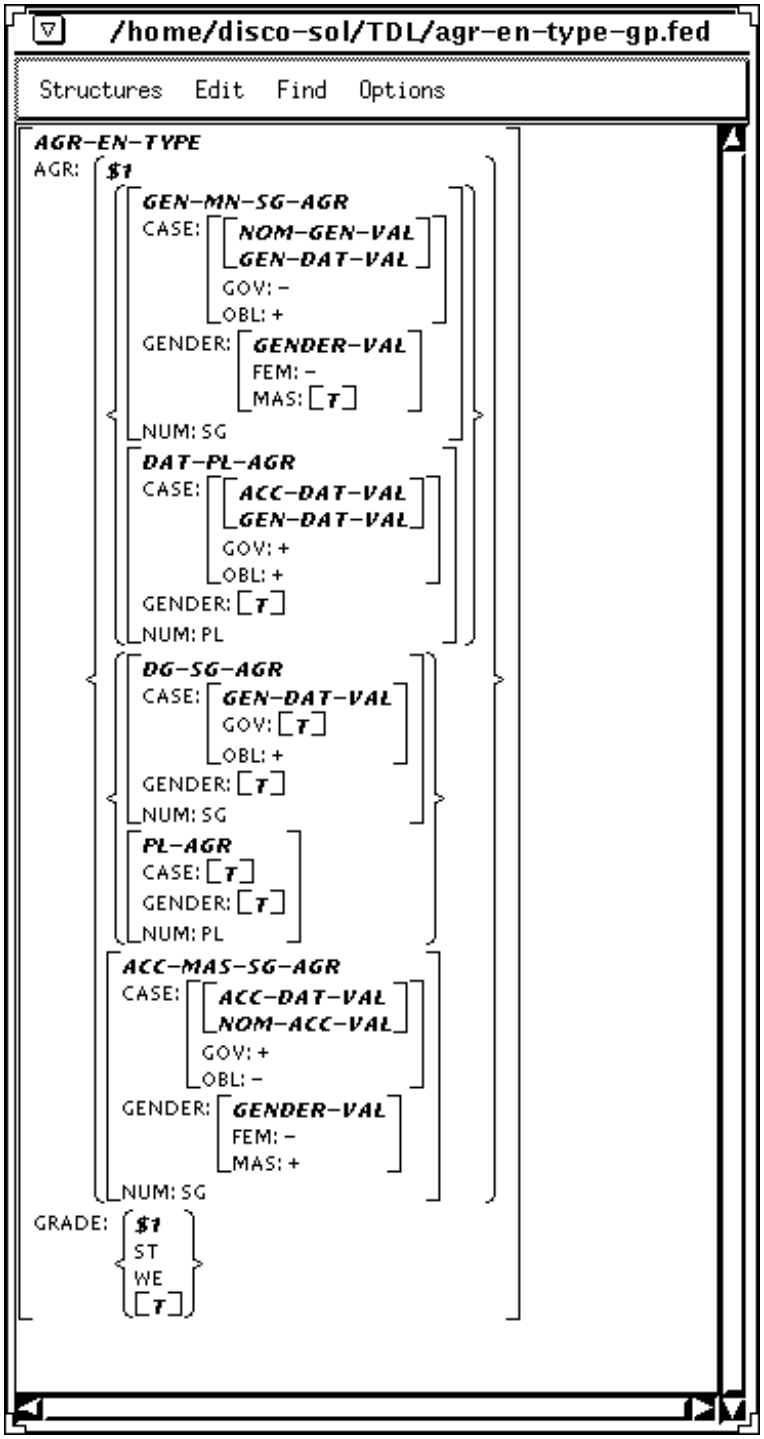


Figure 1: A feature structure type in FEGRAMED

- `: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 "*instance-name*`.fed`"
  possible values: a string or a LISP path name
  Unless *filename* is specified, a filename will be 'computed' from the type name. The file will be created by the $\mathcal{TDL}$-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 $\mathcal{TDL}$2LaTeX

$\mathcal{TDL}$2LaTeX is a tool which generates LaTeX compatible high-quality output of $\mathcal{TDL}$ feature structure types.

- function (`LLP` [*type-name* [*l-options*]])
  `LLP` starts $\mathcal{TDL}$2LaTeX with the LOCAL PROTOTYPE of the feature structure with name *type-name*. If no type name is specified, `LLP` takes the prototype of the *last* type defined before evaluating `LLP`. The LOCAL PROTOTYPE contains only the *local* information given in the definition of type *type-name*. Example:
```
DISCO(40): (LLP 'agr-en-type :fontsize "small"
                :doc-options "a4wide,palatino")
```

- function (`LGP` [*type-name* [*l-options*]])
  `LGP` starts $\mathcal{TDL}$2LaTeX with the GLOBAL PROTOTYPE of the feature structure with name *type-name*. If no type name is specified, `LGP` takes the prototype of the *last* type defined before evaluating `LGP`. The GLOBAL PROTOTYPE contains *all* information that can be inferred for type *type-name* and its supertypes. Example:
```
DISCO(41): (LGP 'agr-en-type :mathmode "equation"
                              :doc-options "leqno")
```

- function (`LTI` *instance-name* [*l-options*])
  `LTI` starts $\mathcal{TDL}$2LaTeX with the feature structure of instance *instance-name*. Example:
```
DISCO(42): (LTI 'agr-en-type4335)
```

An example of a complex feature structure generated by $\mathcal{TDL}$2LaTeX is shown in Figure 2.
l-options are the following optional keywords:

- `:filename` *filename* *default value:* "*type-name*`-gp`", "*type-name*`-lp`" or "*instance-name*"
  possible values: *string*
  Unless *filename* is specified, a filename will be 'computed' from the type name. The filename will be used to generate the LaTeX output file.

- `:filepath` *pathname* *default value:* value of variable `*FILEPATH*`
  possible values: a string or a COMMON LISP path name
  pathname sets the directory in which the LaTeX output file will be created and the shell command *command* will be executed. The value of `*FILEPATH*` defaults to the `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.

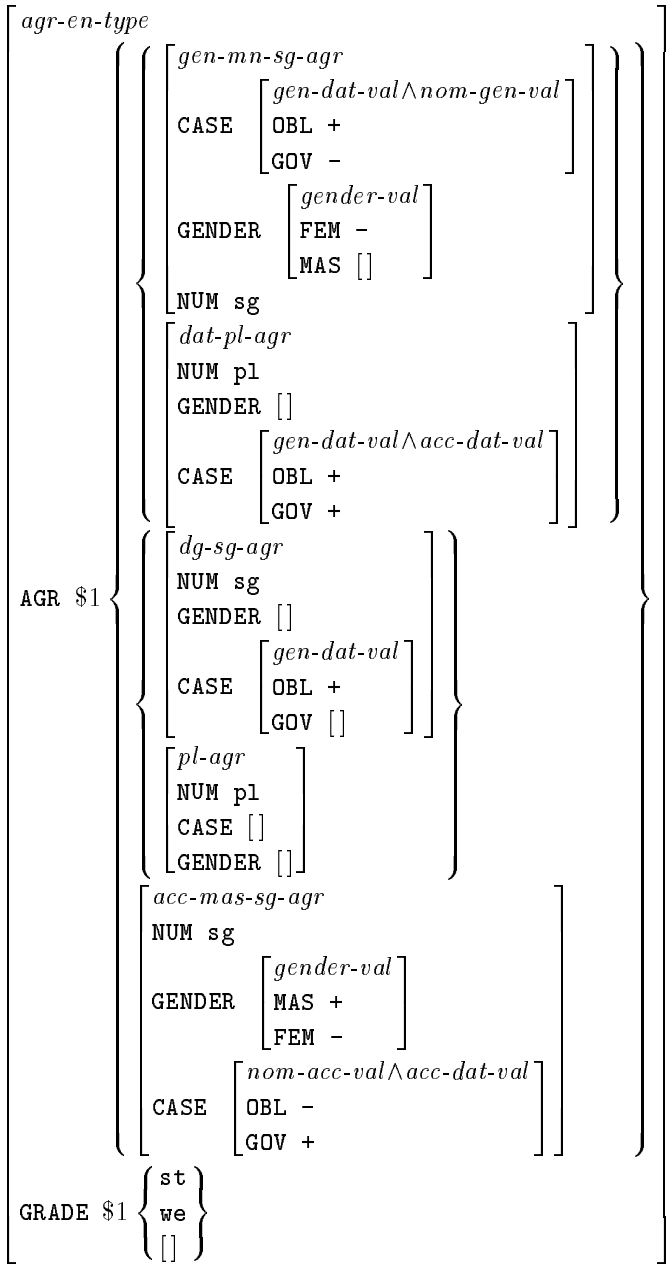


Figure 2: A complex feature structure generated by $\mathcal{TDL}$2LaTeX

- `: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* it `NIL`, only the LaTeX file will be created and $\mathcal{TDL}$2LaTeX will return. If *command* is a string, $\mathcal{TDL}$2LaTeX 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 LaTeX with the output file of $\mathcal{TDL}$2LaTeX and writes PostScript™ 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, $\mathcal{TDL}$2LaTeX 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 `\documentstyle` etc. will be generated. If *flag* is `NIL`, only the LaTeX code of the feature structure enclosed in `\begin{featurestruct}` and `\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.

- `: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 `\documentstyle` options if `:latex-header-p` is `T`. *string* must be a string consisting of the names of zero, one or more valid LaTeX document styles (separated by commata). 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 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., $a \wedge b \wedge c$).
  If *style* has value `:prefix`, complex type entries will be printed in prefix (LISP like) notation (e.g., (AND *a b c*)).

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

5.10.4 Hiding the type field while printing

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

```
DISCO(43): (SETQ *HIDE-TYPES* T)
T
DISCO(44): (PGP 'NOM-SG-AGR)
[CASE   : [GOV : -
           OBL : -]
 GENDER : []
 NUM    : SG]

DISCO(45): (SETQ *HIDE-TYPES* NIL)
NIL
DISCO(46): (PGP 'NOM-SG-AGR)
NOM-SG-AGR [CASE   : CASE-VAL [GOV : -
                               OBL : -]
            GENDER : []
            NUM    : SG]
```

6 Editing and Loading $\mathcal{TDL}$ files

$\mathcal{TDL}$ExtraLight supports loading type definitions from files. $\mathcal{TDL}$ files can be written using an ordinary text editor. When EMACS is used, we recommend running it in *fundamental mode* (which can be switched on with the EMACS command `M-x fundamental-mode`).
A $\mathcal{TDL}$ file may contain type definitions, template definitions, instance definitions or LISP code (e.g., LISP function definitions) in arbitrary order.
Before loading a $\mathcal{TDL}$ file, the $\mathcal{TDL}$ reader must be switched on using `(RON)`. This may also be done within the $\mathcal{TDL}$ file.
COMMON LISP function `(LOAD` *file-name* [`:verbose {T|NIL}`] [`:print {T|NIL}`])
loads either LISP files or $\mathcal{TDL}$ files or mixed files.

7 Displaying the $\mathcal{TDL}$ type hierarchy

It is possible to display the $\mathcal{TDL}$ type hierarchy using the ALLEGRO COMPOSER™. If ALLEGRO COMPOSER™ isn't active by default, it is necessary to load it explicitly by
`DISCO(47): (COMPOSER:START-COMPOSER)` (or alternatively `:com`)
The $\mathcal{TDL}$ type hierarchy is represented via the COMMON LISP OBJECT SYSTEM (CLOS) [Keene 89; Steele 90].
Select menu 'CLOS' and then submenu 'Show Class Subclasses' or 'Show Class Superclasses' and choose `DISCO::*var*` or any other $\mathcal{TDL}$ type in a domain, e.g. `DISCO`. The Composer will show all subclasses (or superclasses) of the specified $\mathcal{TDL}$ type.
`DISCO::*var*` is the top type of domain `DISCO`. It is important not to forget the domain name which is internally the COMMON LISP package name of the domain package.
An example screen dump of a $\mathcal{TDL}$ type hierarchy in CLOS is shown in Figure 3.

8 Top level abbreviations

In the FRANZ ALLEGRO COMMON LISP version of *$\mathcal{TDL}$ExtraLight*, some often used commands are also available as top level abbreviations. The top level command `:alias` prints a list of available abbreviations:

```
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
```

`:composer`, `:define-domain` and `:fegramed` may also be abbreviated by `:com`, `:def` and `:feg`.
All top level commands take the same parameters as the corresponding $\mathcal{TDL}$-LISP functions described in the sections before. Top level commands can only be used in the interactive mode of LISP, but not in $\mathcal{TDL}$ or LISP source files.

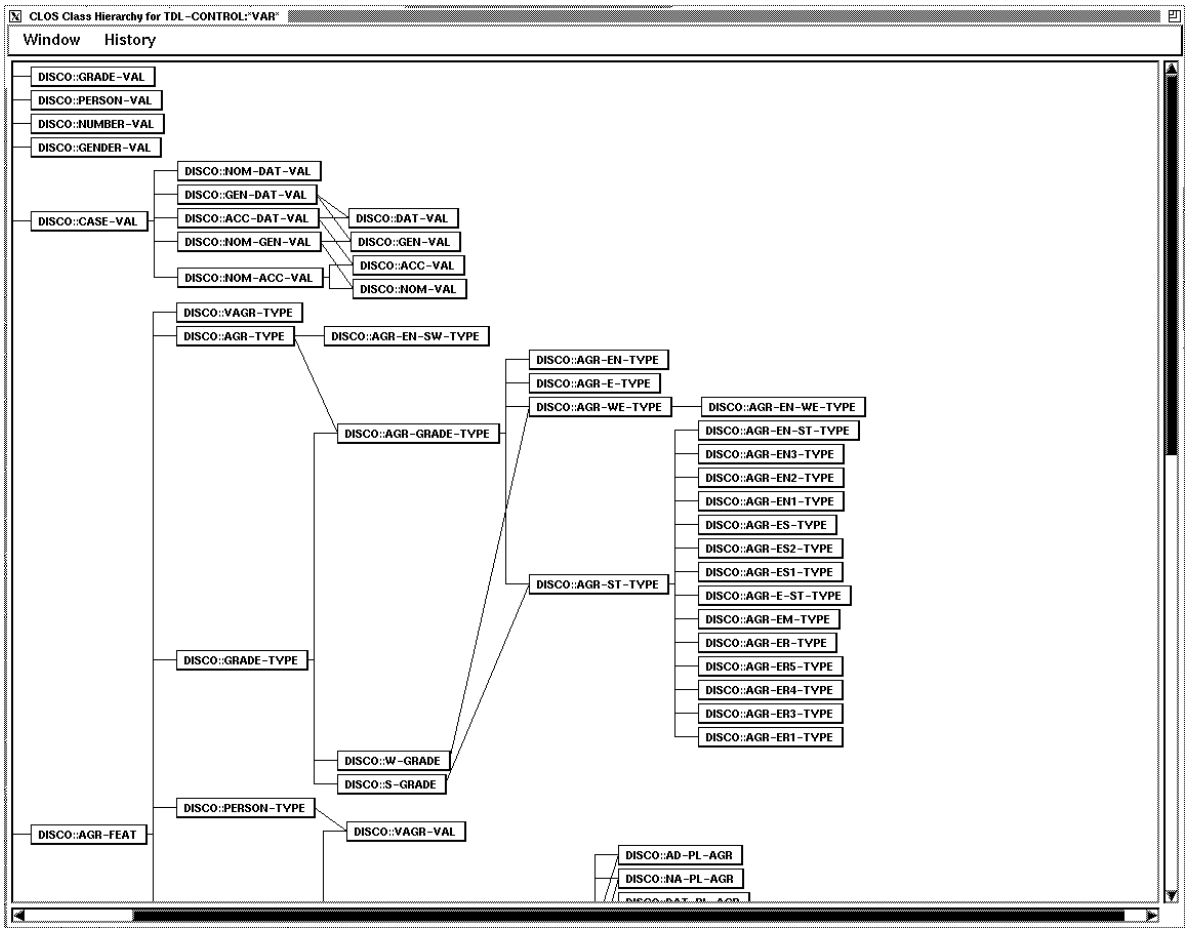


Figure 3: A $\mathcal{TDL}$ type hierarchy in CLOS

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)`
`:ron`, `:roff`, `:composer` and `:fegramed` don't take any parameter.
In addition to these $\mathcal{TDL}$ 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-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-TYPE6780]
#<TDL::FEATURE-STRUCTURE-INFON @ #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]

DISCO(11): ? 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  : 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]
#<TDL::FEATURE-STRUCTURE-INFON @ #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:=[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]
```

```
DISCO(30): (PLP)
APP [C : "Einhornhaus"
     B : "Ein"
     A : "horn"]

DISCO(31): (PPRINT (Get-Identifiers))  ; PPRINT prints *all* identifiers

(APP C NEG-COREF-TYPE PHON A-B-IN-TYPE FLAG A-B-TEMPLATE RESULT Y X
 F-TYPE DD-TYPE2 B DD-TYPE A LIST-SLOT L-TYPE MULTI-INH PERS PERS-TYPE SG NUM
 NUM-SING-TYPE + CASE NOM-DAT-TYPE GOV OBL CASE-VAL $ATTRIB $VALUE)

DISCO(32): _
```

10 *TDLExtraLight* syntax

The next pages contain the *TDLExtraLight* 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:

metasymbols	**meaning**
[⟨expression⟩]	one optional expression
[⟨expression⟩ | ⟨expression⟩ | . . .]	one or none of the expressions
{ ⟨expression⟩ | ⟨expression⟩ | . . .}	exactly one of the expressions
{ ⟨expression⟩ }*	n successive expressions, where $n \in \{0, 1, \ldots\}$
{ ⟨expression⟩ }$^+$	n successive expressions, where $n \in \{1, 2, \ldots\}$

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*] **)** |
template-call }

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

template-call ::= **@***template-name* [**(** [{*param-spec* ,}* *param-spec*] **)**]

param-spec ::= $*symbol* [*value*]

atom ::= { *string* | *symbol* | [**–**]*integer* }

variable ::= { *symbol* | *integer* }

attribute-name ::= *symbol*

type-name ::= *symbol*

function-name ::= *symbol*

template-name ::= *symbol*

disj-index ::= *integer*

type-opt ::= { **:author** *string* |
:date *string* |
:doc *string* |
:status *statuskey* }

statuskey ::= { **:lex-entry** | **:lex-rule** | **:rule** | **:epsilon** | **:root** | **:unknown** |
:multi-word-lexeme | **:sar-rule** | **:lex-triggered-rule** |
:morph-template | **:sar-rule-2nd** }

integer ::= {**0**|**1**|**2**|**3**|**4**|**5**|**6**|**7**|**8**|**9**|**0**}$^+$

symbol ::= *symbol-begin-char*{*symbol-continue-char*}*

symbol-begin-char ::= {**a**–**z**|**A**–**Z**|_|**+**|–|***}

symbol-continue-char ::= {**a**–**z**|**A**–**Z**|**0**–**9**|_|**+**|–|***|$}

string ::= **"**{*any character except* **"**}***"**

10.2 Instance definitions

start ::= { **!** *type-name* {*instance-opt*}*. |
! *conjunction-val* {*instance-opt*}*. |
! *template-call* {*instance-opt*}* . }

instance-opt ::= { **:author** *string* |
:date *string* |
:doc *string* |
:status *statuskey* |
:name *symbol* }

10.3 Template definitions

start ::= **?** *template-name* **(** [{*param-spec* ,}* *param-spec*] **)** **:=** *conjunction-val* {*template-opt*}*.

template-opt ::= { **:author** *string* |
:date *string* |
:doc *string* }

References

[Aït-Kaci & Nasr 86a] Hassan Aït-Kaci and Roger Nasr. *LOGIN: A Logic Programming Language with Built-In Inheritance.* Journal of Logic Programming, 3:185–215, 1986.

[Aït-Kaci & Nasr 86b] Hassan Aït-Kaci and Roger Nasr. *Residuation: A Paradigm for Integrating Logic and Functional Programming.* Technical Report AI-359-86, MCC, Austin, TX, 1986.

[Alshawi 92] Hiyan Alshawi (ed.). *The Core Language Engine.* ACL-MIT Press Series in Natural Language Processing. MIT Press, 1992.

[Backofen & Smolka 92] Rolf Backofen and Gert Smolka. *A Complete and Recursive Feature Theory.* Technical Report RR-92-30, Deutsches Forschungszentrum für Künstliche Intelligenz, Saarbrücken, Germany, 1992.

[Backofen & Weyers 93] Rolf Backofen and Christoph Weyers. *UDiNe—A Feature Constraint Solver with Distributed Disjunction and Classical Negation.* Technical report, Deutsches Forschungszentrum für Künstliche Intelligenz, Saarbrücken, Germany, 1993. Forthcoming.

[Backofen et al. 90] Rolf Backofen, Lutz Euler, and Günter Görz. *Towards the Integration of Functions, Relations and Types in an AI Programming Language.* In: Proceedings of GWAI-90, Berlin, 1990. Springer.

[Bresnan 82] Joan Bresnan (ed.). *The Mental Representation of Grammatical Relations.* Cambridge, Mass.: MIT Press, 1982.

[Carpenter 92] Bob Carpenter. *ALE—The Attribute Logic Engine User's Guide. Version β.* Technical report, Laboratory for Computational Linguistics. Philosophy Department, Carnegie Mellon University, Pittsburgh, PA, December 1992.

[Daelemans et al. 92] Walter Daelemans, Koenraad De Smedt, and Gerald Gazdar. *Inheritance in Natural Language Processing.* Computational Linguistics, 18(2):205–218, 1992.

[Dörre & Eisele 89] Jochen Dörre and Andreas Eisele. *Determining Consistency of Feature Terms with Distributed Disjunctions.* In: Dieter Metzing (ed.), Proceedings of GWAI-89 (15th German Workshop on AI), pp. 270–279, Berlin, 1989. Springer-Verlag.

[Dörre & Eisele 91] Jochen Dörre and Andreas Eisele. *A Comprehensive Unification-Based Grammar Formalism.* Technical Report Deliverable R3.1.B, DYANA, Centre for Cognitive Science, University of Edinburgh, January 1991.

[Gazdar et al. 85] Gerald Gazdar, Ewan Klein, Geoffrey Pullum, and Ivan Sag. *Generalized Phrase Structure Grammar.* Harvard University Press, 1985.

[Harman 63] Gilbert Harman. *Generative Grammars Without Transformation Rules: A Defence of Phrase Structure.* Language, 39:597–616, 1963.

[Höhfeld & Smolka 88] Markus Höhfeld and Gert Smolka. *Definite Relations over Constraint Languages.* LILOG Report 53, WT LILOG–IBM Germany, Stuttgart, October 1988.

[Johnson 88] Mark Johnson. *Attribute Value Logic and the Theory of Grammar.* CSLI Lecture Notes, Number 16. Stanford: Center for the Study of Language and Information, 1988.

[Kantrowitz 91] Mark Kantrowitz. *Portable Utilities for Common Lisp.* Technical Report CMU-CS-91-143, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1991.

[Karttunen 84] Lauri Karttunen. *Features and Values.* In: Proceedings of the 10th International Conference on Computational Linguistics, COLING-84, pp. 28–33, 1984.

[Karttunen 86] Lauri Karttunen. *Radical Lexicalism.* Technical Report CSLI-86-68, Center for the Study of Language and Information, Stanford University, 1986.

[Kasper & Rounds 86] Robert T. Kasper and William C. Rounds. *A Logical Semantics for Feature Structures.* In: Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics, pp. 257–266, 1986.

[Kasper & Rounds 90] Robert T. Kasper and William C. Rounds. *The Logic of Unification in Grammar.* Linguistics and Philosophy, 13:35–58, 1990.

[Kay 79] Martin Kay. *Functional Grammar.* In: C. Chiarello et al. (ed.), Proceedings of the 5th Annual Meeting of the Berkeley Linguistics Society, pp. 142–158, Berkeley, Cal, 1979.

[Kay 85] Martin Kay. *Parsing in Functional Unification Grammar.* In: David R. Dowty, Lauri Karttunen, and Arnold M. Zwicky (eds.), Natural Language Parsing. Psychological, Computational, and Theoretical Perspectives, chapter 7, pp. 251–278. Cambridge: Cambridge University Press, 1985.

[Keene 89] Sonya E. Keene. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS.* Reading, Massachusetts: Addison-Wesley, 1989.

[Kiefer & Fettig 93] Bernd Kiefer and Thomas Fettig. *FEGRAMED—An Interactive Graphics Editor for Feature Structures.* Technical report, Deutsches Forschungszentrum für Künstliche Intelligenz, Saarbrücken, Germany, 1993.

[Kiefer 93] Bernd Kiefer. *Gimmie more HQ Parsers.* Technical report, Deutsches Forschungszentrum für Künstliche Intelligenz, Saarbrücken, Germany, 1993. Forthcoming.

[King 89] Paul J. King. *A Logical Formalism for Head-Driven Phrase Structure Grammar.* PhD thesis, University of Manchester, Department of Mathematics, 1989.

[Knight 89] Kevin Knight. *Unification: A Multidisciplinary Survey.* ACM Computing Surveys, 21(1):93–124, March 1989.

[Knuth 68] Donald E. Knuth. *Semantics of Context-Free Languages.* Mathematical Systems Theory, 2(2):127–145, 1968.

[Krieger & Schäfer 93] Hans-Ulrich Krieger and Ulrich Schäfer. *$\mathcal{TDL}$—A Type Description Language for HPSG. Part 1: Overview.* Technical report, Deutsches Forschungszentrum für Künstliche Intelligenz, Saarbrücken, Germany, 1993. Forthcoming.

[Laubsch 93] Joachim Laubsch. *Zebu: A Tool for Specifying Reversible LALR(1) Parsers.* Technical report, Hewlett-Packard, 1993.

[Moens et al. 89] Marc Moens, Jo Calder, Ewan Klein, Mike Reape, and Henk Zeevat. *Expressing generalizations in unification-based grammar formalisms.* In: Proceedings of the 4th EACL, pp. 174–181, 1989.

[Montague 74] Richard Montague. *Formal Philosophy. Selected Papers of Richard Montague.* New Haven: Yale University Press, 1974. Edited by Richmond H. Thomason.

[Netter 93] Klaus Netter. *Architecture and Coverage of the DISCO Grammar.* In: S. Busemann and Karin Harbusch (eds.), Proceedings of the DFKI Workshop on Natural Language Systems: Modularity and Re-Usability, 1993.

[Pereira & Shieber 84] Fernando C.N. Pereira and Stuart M. Shieber. *The Semantics of Grammar Formalisms Seen as Computer Languages.* In: Proceedings of the 10th International Conference on Computational Linguistics, pp. 123–129, 1984.

[Pereira & Warren 80] Fernando C.N. Pereira and David H.D. Warren. *Definite Clause Grammars for Language Analysis—A Survey of the Formalism and a Comparison with Augmented Transition Networks.* Artificial Intelligence, 13:231–278, 1980.

[Pereira 87] Fernando C.N. Pereira. *Grammars and Logics of Partial Information.* In: J.-L. Lassez (ed.), Proceedings of the 4th International Conference on Logic Programming, Vol. 2, pp. 989–1013, 1987.

[Pollard & Moshier 90] Carl J. Pollard and M. Drew Moshier. *Unifying Partial Descriptions of Sets.* In: P. Hanson (ed.), Information, Language, and Cognition. Vol. 1 of Vancouver Studies in Cognitive Science, pp. xxx–yyy. University of British Columbia Press, 1990.

[Pollard & Sag 87] Carl Pollard and Ivan Sag. *Information-Based Syntax and Semantics. Vol. I: Fundamentals.* CSLI Lecture Notes, Number 13. Stanford: Center for the Study of Language and Information, 1987.

[Pollard & Sag 93] Carl Pollard and Ivan Sag. *Head-Driven Phrase Structure Grammar.* CSLI Lecture Notes. Stanford: Center for the Study of Language and Information, 1993.

[Pollard 89] Carl Pollard. *The Syntax-Semantics Interface in a Unification-Based Phrase Structure Grammar.* In: Stephan Busemann, Christa Hauenschild, and Carla Umbach (eds.), Views of the Syntax-Semantics Interface: Proceedings of the Workshop on "GPSG and Semantics", Technische Universität Berlin, 22-24.Feb 1989, pp. 167–184. Technische Universität Berlin: KIT FAST, 1989.

[Reape 91] Mike Reape. *An Introduction to the Semantics of Unification-Based Grammar Formalisms.* Technical Report Deliverable R3.2.A, DYANA, Centre for Cognitive Science, University of Edinburgh, January 1991.

[Rounds & Kasper 86] William C. Rounds and Robert T. Kasper. *A Complete Logical Calculus for Record Structures Representing Linguistic Information.* In: Proceedings of the 15th Annual Symposium of the IEEE on Logic in Computer Science, 1986.

[Rounds 88] William C. Rounds. *Set Values for Unification-Based Grammar Formalisms and Logic Programming.* Technical Report CSLI-88-129, Center for the Study of Language and Information, 1988.

[Russell et al. 92] Graham Russell, Afzal Ballim, John Carroll, and Susan Warwick-Armstrong. *A Practical Approach to Multiple Default Inheritance for Unification-Based Lexicons.* Computational Linguistics, 18(3):311–337, 1992.

[Sag & Pollard 87] Ivan A. Sag and Carl Pollard. *Head-Driven Phrase Structure Grammar: An Informal Synopsis.* Technical Report CSLI-87-89, Center for the Study of Language and Information, Stanford University, 1987.

[Shieber et al. 83] Stuart Shieber, Hans Uszkoreit, Fernando Pereira, Jane Robinson, and Mabry Tyson. *The Formalism and Implementation of PATR-II.* In: Barbara J. Grosz and Mark E. Stickel (eds.), Research on Interactive Acquisition and Use of Knowledge, pp. 39–79. Menlo Park, Cal.: AI Center, SRI International, 1983.

[Shieber 86] Stuart M. Shieber. *An Introduction to Unification-Based Approaches to Grammar.* CSLI Lecture Notes, Number 4. Stanford: Center for the Study of Language and Information, 1986.

[Smolka 88] Gert Smolka. *A Feature Logic with Subsorts.* LILOG Report 33, WT LILOG–IBM Germany, Stuttgart, Mai 1988.

[Smolka 89] Gert Smolka. *Feature Constraint Logic for Unification Grammars.* IWBS Report 93, IWBS–IBM Germany, Stuttgart, November 1989.

[Smolka 91] Gert Smolka. *Residuation and Guarded Rules for Constraint-Logic Programming.* Research Report RR-91-13, DFKI, Saarbrücken, 1991.

[Steele 90] Guy L. Steele. *Common Lisp: The Language.* Bedford, MA: Digital Press, 2nd edition, 1990.

[Uszkoreit 86] Hans Uszkoreit. *Categorial Unification Grammars.* In: Proceedings of the 11th International Conference on Computational Linguistics, pp. 187–194, 1986.

[Uszkoreit 88] Hans Uszkoreit. *From Feature Bundles to Abstract Data Types: New Directions in the Representation and Processing of Linguistic Knowledge.* In: A. Blaser (ed.), Natural Language at the Computer—Contributions to Syntax and Semantics for Text Processing and Man-Machine Translation, pp. 31–64. Berlin: Springer, 1988.

[Zajac 92] Rémi Zajac. *Inheritance and Constraint-Based Grammar Formalisms.* Computational Linguistics, 18(2):159–182, 1992.