TDL
A Type Description Language for HPSG
Part 2: User Guide

Hans-Ulrich Krieger, Ulrich Schäfer

December 1994
Deutsches Forschungszentrum für Künstliche Intelligenz

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

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

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

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

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

Dr. Dr. D. Ruland
Director
TDL
A Type Description Language for HPSG
Part 2: User Guide

Hans-Ulrich Krieger, Ulrich Schäfer

DFKI-D-94-14
This work has been supported by a grant from The Federal Ministry for Research and Technology (FKZ ITWM-9002 0).
TDL
A Type Description Language for HPSG
Part 2: User Guide

Hans-Ulrich Krieger, Ulrich Schäfer
{krieger, schaefer}@dfki.uni-sb.de
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3
D-66123 Saarbrücken, Germany

Abstract
This documentation serves as a user’s guide to the type description language TDC which is employed in natural language projects at the DFKI. It is intended as a guide for grammar writers rather than as a comprehensive internal documentation. Some familiarity with grammar formalisms/theories such as Head-Driven Phrase Structure Grammar (HPSG) is assumed. The manual describes the syntax of the TDC formalism, the user-accessible control functions and variables, and the various tools such as type grapher, feature editor, TDC2LATEX, Emacs TDC mode, and print interface.

1 We would like to thank Elizabeth Hinkelma for reading a draft version of this manual.
# Contents

1 Introduction  

2 The TDL System  

3 Starting TDL  

4 Syntax of TDL  
   4.1 TDL BNF  
     4.1.1 TDL Main Constructors  
     4.1.2 Type Definitions  
     4.1.3 Instance Definitions  
     4.1.4 Template Definitions  
     4.1.5 Declarations  
     4.1.6 Statements  
     4.2 Creating and Changing Domains  
     4.3 The Structure of TDL Grammars  
     4.4 Domain Environment  
     4.5 Declare Environment and Declarations  
     4.6 Type Environment and Type Definitions  
       4.6.1 Feature Structure Definitions  
       4.6.2 Atoms  
       4.6.3 Paths  
       4.6.4 Logical Operators  
       4.6.5 Type Specification and Inheritance  
       4.6.6 Multiple Inheritance  
       4.6.7 Coreferences  
       4.6.8 Negated Coreferences  
       4.6.9 Simple Disjunctions  
       4.6.10 Distributed Disjunctions  
       4.6.11 Lists  
       4.6.12 Difference Lists  
       4.6.13 Negation  
       4.6.14 External Coreference Constraints  
       4.6.15 Functional Constraints  
       4.6.16 Template Calls  
       4.6.17 Nonmonotonicity and Value Restrictions  
       4.6.18 Rules  
     4.7 Optional Keywords in Definitions  
     4.8 Template Environment and Template Definitions  
     4.9 Instance Environment and Instance Definitions  
     4.10 Control Environment  
     4.11 Lisp Environment  
     4.12 Comments  

5 Useful Functions, Switches, and Variables  
   5.1 Global Switches and Variables  
   5.2 Setting Switches and Global Variables  
   5.3 Including Grammar Files  
   5.4 Expanding Types and Instances  
     5.4.1 Defining control information: defcontrol  
     5.4.2 Expanding Types and Instances: expand-type and expand-instance  
     5.4.3 The Syntax of expand-control
### CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4.4</td>
<td>Printing Control Information</td>
<td>35</td>
</tr>
<tr>
<td>5.4.5</td>
<td>How to Stop Recursion</td>
<td>35</td>
</tr>
<tr>
<td>5.5</td>
<td>Checking Welltypedness/Appropriateness</td>
<td>37</td>
</tr>
<tr>
<td>5.6</td>
<td>Deleting Types and Instance Definitions</td>
<td>38</td>
</tr>
<tr>
<td>5.7</td>
<td>Resetting Prototypes of Types and Instances</td>
<td>38</td>
</tr>
<tr>
<td>5.8</td>
<td>Accessing Internal Information (Infons)</td>
<td>38</td>
</tr>
<tr>
<td>5.9</td>
<td>Collecting and Printing Statistical Information</td>
<td>39</td>
</tr>
<tr>
<td>5.10</td>
<td>Memoization</td>
<td>40</td>
</tr>
<tr>
<td>5.11</td>
<td>Tuning up Unification: Training Sessions</td>
<td>41</td>
</tr>
<tr>
<td>5.12</td>
<td>Defining Reader Macros</td>
<td>42</td>
</tr>
<tr>
<td>5.13</td>
<td>Printing Messages</td>
<td>42</td>
</tr>
<tr>
<td>5.14</td>
<td>Help</td>
<td>42</td>
</tr>
<tr>
<td>5.15</td>
<td>Wait</td>
<td>42</td>
</tr>
<tr>
<td>5.16</td>
<td>Exit \texttt{TDC}</td>
<td>42</td>
</tr>
<tr>
<td>5.17</td>
<td>Getting Information about Defined Templates</td>
<td>42</td>
</tr>
<tr>
<td>5.18</td>
<td>Printing Feature Structures</td>
<td>42</td>
</tr>
<tr>
<td>5.18.1</td>
<td>Printing to the Interactive Screen or to Streams (ASCII)</td>
<td>43</td>
</tr>
<tr>
<td>5.18.2</td>
<td>\texttt{FERGAMED}</td>
<td>44</td>
</tr>
<tr>
<td>5.18.3</td>
<td>\texttt{TDC2LATEX}</td>
<td>46</td>
</tr>
<tr>
<td>6</td>
<td>\texttt{TDC} Grapher</td>
<td>54</td>
</tr>
<tr>
<td>7</td>
<td>Print/Read Syntax for \texttt{TDC} Type Entries</td>
<td>56</td>
</tr>
<tr>
<td>7.1</td>
<td>Print Modes</td>
<td>56</td>
</tr>
<tr>
<td>7.2</td>
<td>Global Variables</td>
<td>57</td>
</tr>
<tr>
<td>7.3</td>
<td>BNF</td>
<td>58</td>
</tr>
<tr>
<td>8</td>
<td>Emacs \texttt{TDC} Mode</td>
<td>59</td>
</tr>
<tr>
<td>8.1</td>
<td>Installation</td>
<td>59</td>
</tr>
<tr>
<td>8.2</td>
<td>Key Bindings</td>
<td>59</td>
</tr>
<tr>
<td>9</td>
<td>Top Level Abbreviations (\texttt{ALLEGRO COMMON LISP Only})</td>
<td>60</td>
</tr>
<tr>
<td>10</td>
<td>Sample Session</td>
<td>61</td>
</tr>
</tbody>
</table>
1 Introduction

This is the second part of TDC—A type description language for HPSG. This documentation serves as a user guide to TDC. It is intended as a guide for grammar writers rather than as a comprehensive internal documentation. Some familiarity with grammar formalisms theories such as Head-Driven Phrase Structure Grammar [Pollard & Sag 87; Pollard & Sag 94] is assumed. The manual describes the syntax of the TDC formalism, the user accessible control functions and variables, and the various tools such as type grapher, feature editor, TDC2LaTeX, Emacs TDC mode, and print interface.

For motivation, architecture, properties of the type hierarchy, implementational issues and comparison to related systems, refer to [Krieger & Schäfer 93a], [Krieger & Schäfer 94a], [Krieger & Schäfer 94b], [Krieger 95], and [Schäfer 95].

The TDC system is integrated into various natural language systems such as DISCO [Uszkoreit et al. 94], and PRACMA [Jameson et al. 94].

Corrections and other information can be ftp'd from ftp://cl-ftp.dfki.uni-sb.de/pub/tdc.
World Wide Web: (publications, software, etc.): http://cl-www.dfki.uni-sb.de/
Email: tdl@dfki.uni-sb.de

2 The TDC System

The TDC distribution includes COMMON LISP source files in the following directories, which correspond to the modules of the system definition.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Module</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>tdl/compile/</td>
<td>compile</td>
<td>TDL-COMPILE</td>
</tr>
<tr>
<td>tdl/control/</td>
<td>control</td>
<td>TDL</td>
</tr>
<tr>
<td>tdl/elisp/</td>
<td>Emacs lisp files</td>
<td>~</td>
</tr>
<tr>
<td>tdl/encode/</td>
<td>hierarchy</td>
<td>TDL-HIERARCHY</td>
</tr>
<tr>
<td>tdl/expand/</td>
<td>expand</td>
<td>TDL</td>
</tr>
<tr>
<td>tdl/grapher/</td>
<td>TDC Grapher system</td>
<td>CLIM-USER</td>
</tr>
<tr>
<td>tdl/packages/</td>
<td>package definitions</td>
<td>TDL-USER</td>
</tr>
<tr>
<td>tdl/parse/</td>
<td>parse</td>
<td>TDL-PARSE</td>
</tr>
<tr>
<td>tdl/recycler/</td>
<td>TDC Recycler system</td>
<td>TDL-RECYCLER</td>
</tr>
<tr>
<td>tdl/simplify/</td>
<td>simplify</td>
<td>TDL-SIMPLIFY</td>
</tr>
<tr>
<td>tdl/statistics/</td>
<td>statistics</td>
<td>TDL-STATISTICS</td>
</tr>
</tbody>
</table>

The TDC system depends on the systems ZEBU (LALR(1) parser), UDNe (unifier) and FEGRAMED (feature editor). The TDC Recycler is a tool which translates grammar files from TDCExtraLight [Krieger & Schäfer 93b] into the new TDC syntax. TDC2LaTeX and TDC Grapher are part of the TDC system.

The system is implemented in portable COMMON LISP [Steele 90] and has been tested with Franz Allegro Common Lisp, Macintosh Common Lisp, Lucid Common Lisp, and CLISP2.

3 Starting TDC

To start TDC,

1. Start COMMON LISP.
2. (load-system "tdl") loads necessary parts of TDC such as the unifier UDNe, type definition reader, feature editor (FEGRAMED), type hierarchy management and the TDC2LaTeX interface. Alternatively, (load-system "tdl-grapher") can be used to start system tdl and the type grapher.

The portable system definition facility DESYSTEM is described in [Kantrowitz 91].

2 Thanks to Stephan Oepen and Bernd Kiefer for checking and improving portability.
3 The availability of this function presupposes that the DISCO loadup environment (file loadup.lisp) has been successfully loaded into the COMMON LISP system. Refer to the DISCO installation and operation guide for details.
3. After loading the LISP code, the following prompt appears on the screen:

Welcome to the Type Description Language TDL.

USER(2):  _

4. The TDC reader is invoked by simply typing (tdl). You can either work interactively (e.g., create a domain, define types, etc.) or load TDC grammar files by using the include command. If an error has occurred, e.g., a syntax error, (tdl) restarts the TDC reader.

5. ldt exits the TDC reader and returns to COMMON LISP. The COMMON LISP function (EXIT) quits the interpreter. If you are in an Emacs environment, C-x C-c kills the Emacs process.

It is also possible to define one’s own portable system definitions in the [Kantrowitz 91] paradigm which could then automatically start TDC and include grammar definitions, etc.

4 Syntax of TDC

The TDC syntax provides type definitions, instance definitions (for roles and lexicon entries), templates (parameterized macros), specification of declarative control information, as well as statements (calls to built-in functions) that are especially useful for the interactive development of NL grammars. There is no difference between the syntax of TDC grammar files and the syntax for the interactive mode. All syntactic constructs can be used in either mode.

Note that the arguments of statements need to be quoted if they are symbols or lists containing symbols. This is necessary if the statement is defined as a COMMON LISP function, but not if the statement is defined as a COMMON LISP macro. Almost all statements are defined as functions. The only macros defined by the TDC system are defcontrol, leval, alias, print-control, print-switch, set-switch, and with-print-mode. Examples:

\[
\text{pgp 'agr-type :label-hide-list '(x y z).}
\]

- but:

\[
\text{defcontrol agr-type ((:delay (u *) (v x.y))).}
\]

It is important not to forget the dot delimiter at the end of TDC expressions since the TDC reader will wait for it. It is possible to mix LISP code and TDC definitions in a file. Some examples are shown in Section 4.3. Newline characters, spaces or comments (Section 4.12) can be inserted anywhere between tokens (symbols, braces, parentheses, etc.).

4.1 TDC BNF

The TDC syntax is given in extended BNF (Backus-Naur Form). Terminal symbols (characters to be typed in) are printed in typewriter style. Nonterminal symbols are printed in italic style. The grammar starts with the start production. The following table explains the meanings of the metasymbols used in extended BNF.

<table>
<thead>
<tr>
<th>metasymbols</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\ldots</td>
<td>alternative expressions</td>
</tr>
<tr>
<td>\ldots</td>
<td>one optional expression</td>
</tr>
<tr>
<td>\ldots\ldots</td>
<td>one or none of the expressions</td>
</tr>
<tr>
<td>\ldots\ldots</td>
<td>exactly one of the expressions</td>
</tr>
<tr>
<td>\ldots\ldots</td>
<td>(n ) successive expressions, where (n \in {0,1,\ldots})</td>
</tr>
<tr>
<td>\ldots\ldots</td>
<td>(n ) successive expressions, where (n \in {1,2,\ldots})</td>
</tr>
</tbody>
</table>
4.1 TDL Main Constructors

\[
\begin{align*}
\text{start} &\rightarrow \{\text{block} \mid \text{statement}\}^* \\
\text{block} &\rightarrow \begin{cases}
\begin{align*}
&\text{begin} : \text{control} \{\text{type-def} \mid \text{instance-def} \mid \text{start}\}^* \text{end} : \text{control}. \\
&\text{begin} : \text{declare} \{\text{declare} \mid \text{start}\}^* \text{end} : \text{declare}. \\
&\text{begin} : \text{domain} \{\text{start}\}^* \text{end} : \text{domain domain}. \\
&\text{begin} : \text{instance} \{\text{instance-def} \mid \text{start}\}^* \text{end} : \text{instance}. \\
&\text{begin} : \text{lisp} \{\text{Common-Lisp-Expression}\}^* \text{end} : \text{lisp}. \\
&\text{begin} : \text{template} \{\text{template-def} \mid \text{start}\}^* \text{end} : \text{template}. \\
&\text{begin} : \text{type} \{\text{type-def} \mid \text{start}\}^* \text{end} : \text{type}.
\end{align*}
\end{cases}
\end{align*}
\]

4.2 Type Definitions

\[
\begin{align*}
\text{type-def} &\rightarrow \text{type} \{\text{avm-def} \mid \text{subtype-def}\}. \\
\text{type} &\rightarrow \text{identifier} \\
\text{avm-def} &\rightarrow \begin{cases}
\begin{align*}
&\text{body}\{\text{option}\}^* \\
&\text{= nonmonotonic}\{\text{constraint}\{\text{constraint}\}^*\}\{\text{option}\}^*
\end{align*}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{body} &\rightarrow \text{disjunction}\{\text{disjunction}\}\{\text{constraint}\{\text{constraint}\}^*\}
\end{align*}
\]

\[
\begin{align*}
\text{disjunction} &\rightarrow \text{conjunction}\{\{1 \mid^*\}\text{conjunction}\}^*
\end{align*}
\]

\[
\begin{align*}
\text{conjunction} &\rightarrow \text{term}\{\& \text{term}\}^*
\end{align*}
\]

\[
\begin{align*}
\text{term} &\rightarrow \text{atom} \mid \text{feature-term} \mid \text{diff-list} \mid \text{list} \mid \text{coreference} \\
\text{distributed-disj} \mid \text{templ-par} \mid \text{templ-call} \mid \text{term}\{\text{disjunction}\}
\end{align*}
\]

\[
\begin{align*}
\text{atom} &\rightarrow \text{string} \mid \text{integer} \mid \text{identifier}
\end{align*}
\]

\[
\begin{align*}
\text{feature-term} &\rightarrow \begin{cases}
\begin{align*}
&\text{attr-val}\{\text{option}\}^* \\
&\text{attribute}\{\text{restriction}\} \mid \text{attribute}\{\text{restriction}\}\text{disjunction}
\end{align*}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{attr-val} &\rightarrow \text{identifier} \mid \text{templ-par}
\end{align*}
\]

\[
\begin{align*}
\text{restriction} &\rightarrow \text{conj-restriction}\{\{1 \mid^*\}\text{conj-restriction}\}^*
\end{align*}
\]

\[
\begin{align*}
\text{conj-restriction} &\rightarrow \text{basic-restriction}\{\& \text{basic-restriction}\}^*
\end{align*}
\]

\[
\begin{align*}
\text{basic-restriction} &\rightarrow \text{type} \mid \text{term} \mid \text{diff-list} \mid \text{list} \mid \text{coreference} \\
\text{distributed-disj} \mid \text{templ-par} \mid \text{templ-call} \mid \text{term}\{\text{disjunction}\}
\end{align*}
\]

\[
\begin{align*}
\text{diff-list} &\rightarrow \begin{cases}
\begin{align*}
&\text{< disjunction \{ disjunction \}^* } \text{>} [ ]\{ \text{type}\}
\end{align*}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{list} &\rightarrow \begin{cases}
\begin{align*}
&\text{< nonempty-list } [ \text{list-restriction}]
\end{align*}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{nonempty-list} &\rightarrow \begin{cases}
\begin{align*}
&\text{disjunction}\{\text{disjunction}\}^*, \ldots \mid \text{disjunction}\{\text{disjunction}\}^*\{\text{disjunction}\}
\end{align*}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{list-restriction} &\rightarrow \begin{cases}
\begin{align*}
&\text{type}\{\text{integer}, \text{integer}\}\{\text{integer}\}
\end{align*}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{coreference} &\rightarrow \text{#coref-name}\{\text{coref-name}\}^* \\
\text{coref-name} &\rightarrow \begin{cases}
\begin{align*}
&\text{identifier} \mid \text{integer}
\end{align*}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{distributed-disj} &\rightarrow \text{#disj-name}\{\text{disjunction}\{\text{disjunction}\}^+\}
\end{align*}
\]

\[
\begin{align*}
\text{disj-name} &\rightarrow \text{identifier} \mid \text{integer}
\end{align*}
\]

\[
\begin{align*}
\text{templ-name} &\rightarrow \text{identifier} \\
\text{templ-par} &\rightarrow \text{#templ-name}\{\text{templ-par}\{\text{templ-par}\}^*\}
\end{align*}
\]

\[
\begin{align*}
\text{constraint} &\rightarrow \text{#coref-name}\{\text{function-call}\{\text{disjunction}\}^*\}
\end{align*}
\]

\[
\begin{align*}
\text{function-name} &\rightarrow \text{identifier}
\end{align*}
\]

\[
\begin{align*}
\text{nonmonotonic} &\rightarrow \text{type}\{\text{overwrite-path}\{\text{overwrite-path}\}^*\}
\end{align*}
\]

\[
\begin{align*}
\text{overwrite-path} &\rightarrow \text{identifier}\{\text{identifier}\}\{\text{disjunction}\}
\end{align*}
\]

\[
\begin{align*}
\text{subtype-def} &\rightarrow \begin{cases}
\begin{align*}
&\text{< type}\{\text{option}\}^* \\
&\text{option}\rightarrow \text{status}\{\text{identifier}\} \mid \text{author}\{\text{string}\} \mid \text{date}\{\text{string}\} \mid \text{doc}\{\text{string}\}
\end{align*}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{expand-control} &\rightarrow \text{expand-control}
\end{align*}
\]
4.1 TDC BNF

\[ \text{expand-control} \rightarrow \left( \left( \text{expand} \left( \left( \text{type} \mid \text{type index [pred]} \right) \mid \text{path}^+ \right) \right)^* \right) \mid \left( \text{expand-only} \left( \left( \text{type} \mid \text{type index [pred]} \right) \mid \text{path}^+ \right) \right)^* \mid \left( \text{delay} \left( \left( \text{type} \mid \text{type [pred]} \right) \mid \text{path}^+ \right) \right)^* \mid \left( \text{maxdepth} \text{ integer} \right) \mid \left( \text{ask-disj}\text{-preference} \left( \text{t} \mid \text{nil} \right) \right) \mid \left( \text{attribute-preference} \left( \text{identifier} \right)^* \right) \mid \left( \text{use-conj}\text{-heuristics} \left( \text{t} \mid \text{nil} \right) \right) \mid \left( \text{use-disj}\text{-heuristics} \left( \text{t} \mid \text{nil} \right) \right) \mid \left( \text{expand-function} \left( \text{depth} \mid \text{types} \right) \mid \text{-first-expand} \right) \mid \left( \text{resolved-predicate} \left( \text{resolved-p} \mid \text{always-false} \mid \ldots \right) \right) \mid \left( \text{ignore-global-control} \left( \text{t} \mid \text{nil} \right) \right) \right) \]

\[ \text{path} \rightarrow \left( \text{identifier} \mid \text{pattern} \right) \left( \left( \text{identifier} \mid \text{pattern} \right)^* \right) \]

\[ \text{pattern} \rightarrow \text{eq} \mid \text{subsumes} \mid \text{extends} \mid \ldots \]

4.1.3 Instance Definitions

\[ \text{instance-def} \rightarrow \text{instance avm-def} . \]
\[ \text{instance} \rightarrow \text{identifier} \]

4.1.4 Template Definitions

\[ \text{template-def} \rightarrow \text{templ-name} \left( \left[ \text{templ-par} \left( \left( \text{ templ-par} \right)^* \right) \right] \right) : = \text{body} \left( \left( \text{option} \right)^* \right) . \]

4.1.5 Declarations

\[ \text{declaration} \rightarrow \text{partition} \mid \text{incompatible} \mid \text{sort-def} \mid \text{built-in-def} \mid \text{hide-attributes} \mid \text{hide-values} \mid \text{export-symbols} \]
\[ \text{partition} \rightarrow \text{type} = \text{type} \left( \left[ \text{type} \right]^* \right) . \]
\[ \text{incompatible} \rightarrow \text{nil} = \text{type} \left( \text{& type} \right)^+ . \]
\[ \text{sort-def} \rightarrow \text{sort}[s] : \text{type} \left( \right)^+ . \]
\[ \text{built-in-def} \rightarrow \text{built-in}[s] : \text{type} \left( \right)^+ . \]
\[ \text{hide-attributes} \rightarrow \text{hide-attributes}[s] : \text{identifier} \left( \right)^* . \]
\[ \text{hide-values} \rightarrow \text{hide-values}[s] : \text{identifier} \left( \right)^* . \]
\[ \text{export-symbols} \rightarrow \text{export-symbols}[s] : \text{identifier} \left( \right)^* . \]
4.1.6 Statements

\[
\text{statement} \rightarrow \text{check-well-typedness} \left[ \{ \text{type} \mid \text{instance} \mid :\text{all} \} \right] \left[ \{ :\text{instances} \mid :\text{avms} \} \right] \\
\quad \left[ :\text{domain} \mid :\text{index} \mid :\text{verbose} \left( \{ \text{t} \mid \text{nil} \} \right) \right] \\
\text{clear-simplify-memo-tables} \left[ :\text{domain} \mid :\text{threshold} \mid \text{integer} \right] \\
\text{compute-appro\p{p} \left[ :\text{domain} \mid :\text{warn-if-not-unique} \left( \{ \text{t} \mid \text{nil} \} \right) \right] \\
\text{defcontrol} \left[ :\text{global} \mid :\text{instance} \right] \text{expand-control} \left[ :\text{index} \mid \text{index} \right] \\
\text{defdomain} \text{domain} \left\{ \text{defdomain-option} \right\} \\
\text{deletedomain} \text{domain} \\
\text{defreadermacro} \text{identifier} \left( \{ \text{string} \mid \text{nil} \} \{ \text{string} \} \right) \\
\text{delete-all-instances} \left[ :\text{domain} \right] \\
\text{delete-instance} \left[ \text{instance} \mid :\text{domain} \mid :\text{index} \mid \text{integer} \right] \\
\text{delete-type} \left[ \{ \text{type} \mid :\text{domain} \mid \text{domain} \right] \\
\text{describe-template} \text{template-name} \left[ :\text{domain} \right] \\
\text{do-all-infons} \left[ \text{infon-keys} \right] \\
\text{do-infon} \left[ \text{infon-keys} \right] \\
\text{expand-all-instances} \left[ \text{expand-option} \right] \\
\text{expand-all-types} \left[ \text{expand-option} \right] \\
\text{expand-instance} \left[ \text{instance} \mid :\text{index} \mid \text{integer} \right] \left[ \text{expand-option} \right] \\
\text{expand-type} \left[ \text{type} \mid :\text{index} \mid \text{integer} \right] \left[ \text{expand-option} \right] \\
\text{fegramed} \\
\quad \{ \{ \text{fgi} \mid \text{fl} \} \left[ \text{instance} \mid \text{fegramed-option} \right] \} \\
\quad \{ \{ \text{fgp} \mid \text{flp} \} \left[ \text{type} \mid \text{fegramed-option} \right] \} \\
\text{get-switch} \text{identifier} \\
\text{grapher} \\
\text{help} \left[ \text{identifier} \right] \\
\text{include} \text{filename} \\
\text{lct} \\
\text{leval} \left[ \text{Common-Lisp-Expression} \right] \\
\quad \{ \{ \text{lg} \mid \text{l} \} \left[ \text{instance} \mid \text{latex-option} \right] \} \\
\quad \{ \{ \text{lgp} \mid \text{llp} \} \left[ \text{type} \mid \text{latex-option} \right] \} \\
\text{message} \text{string} \left[ \text{Common-Lisp-Expression} \right] \\
\text{load-simplify-memo-table} \left[ :\text{domain} \mid :\text{filename} \mid \text{filename} \right] \\
\quad \{ \{ \text{gpi} \mid \text{pl} \} \left[ \text{instance} \mid \text{print-option} \right] \} \\
\quad \{ \{ \text{gpp} \mid \text{plp} \} \left[ \text{type} \mid \text{print-option} \right] \} \\
\text{print-all-names} \left[ \text{infon-keys} \right] \\
\text{print-all-statistics} \left[ :\text{domain} \mid :\text{stream} \mid \text{stream} \right] \\
\text{print-appro\p{p}} \left[ :\text{domain} \mid \text{domain} \right] \\
\text{print-control} \left[ \{ \text{type} \mid \text{instance} \mid :\text{global} \right] \\
\text{print-domain-statistics} \left[ :\text{domain} \mid \text{domain} \mid :\text{stream} \mid \text{stream} \right] \\
\text{print-expand-statistics} \left[ :\text{domain} \mid \text{domain} \mid :\text{stream} \mid \text{stream} \right] \\
\text{print-global-statistics} \left[ :\text{stream} \mid \text{stream} \right] \\
\text{print-recursive-sscs} \left[ :\text{domain} \mid \text{domain} \right] \\
\text{print-simplify-memo-tables} \left[ :\text{domain} \mid :\text{stream} \mid :\text{stream} \right] \\
\quad \left[ :\text{threshold} \mid \text{integer} \right] \\
\text{print-simplify-statistics} \left[ :\text{domain} \mid :\text{stream} \mid :\text{stream} \right] \\
\text{print-switch} \text{identifier} \\
\text{print-unified-types} \left[ :\text{filename} \mid \text{string} \mid :\text{domain} \mid \text{domain} \right] \\
\text{recompute} \\
\text{reset-all-instances} \left[ :\text{domain} \right] \\
\text{reset-all-protos} \left[ :\text{domain} \right] \\
\text{reset-all-statistics} \left[ :\text{domain} \mid \text{domain} \right] \\
\text{reset-domain-statistics} \left[ :\text{domain} \mid \text{domain} \right] \\
\]
reset-expand-statistics [:domain domain]. |
reset-global-statistics. |
reset-instance [instance [:domain domain] [:index integer]]. |
reset-print-mode. |
reset-proto [type [:domain domain] [:index index]]. |
reset-simplify-statistics [:domain domain]. |
restore-print-mode. |
save-print-mode. |
set-print-mode [print-mode]. |
set-print-mode [print-mode]. |
set-switch identifier Common-Lisp-Expression. |
start-collect-unified-types [:domain domain]. |
tune-types [:domain domain] [:threshold integer] [...]. |
wait. |
with-print-mode print-mode Common-Lisp-Expression.

infon-keys → see Section 5.8
print-option → :domain domain | :index index | :stream {t | nil | stream} |
| :label-sort-list ({identifier}*) | :label-hide-list ({identifier}*) |
| :remove-tops {t | nil} | :init-pos integer |
laxt-option → :domain domain | :index index | :hide-types {t | nil} |
| :filename filename | ... (see Section 5.18.3)
| fegramed-option → :domain domain | :index index |
| :filename filename | :wait {t | nil} | :file-only {t | nil}
defdomain-option → :hide-attributes ({identifier}*) |
| :hide-values ({identifier}*) |
| :export-symbols ({identifier}*) |
| :documentation string |
| :top type |
| :bottom type |
| :load-built-ins-p {t | nil} |
| :errorp {t | nil}
expand-option → :domain domain |
| :expand-control expand-control |
domain → :identifier | :identifier | "identifier"
| :hide-types | :read-in | :tdll2asll | :x2morpf
filename → string
index → integer for instances
| integer | identifier string for avm types
integer → {0|1|2|3|4|5|6|7|8|9}+
identifier → {a-z|A-Z|0-9|_+\|\*?}+
string → "{any character}"*

4.2 Creating and Changing Domains

Domains are sets of type, instance and template definitions. It is possible to define and use several domains
at the same time 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).
An arbitrary keyword symbol or string may be chosen for domain except those which are names of existing
COMMON LISP packages, e.g. TDL, COMMON-LISP or COMMON-LISP-USER. All domain names will be
normalized into strings containing only uppercase characters.
The name TDL is reserved for internal functions and variables. It is possible to specify the domain of the
current definition by using the begin :domain domain and end :domain domain block delimiters (see
Sections 4.3 and 4.4).
• function defdomain domain [:hide-attributes attribute-list]
  [:hide-values values-list]
  [:export-symbols symbol-list]
  [:documentation doc-string]
  [:top top-symbol]
  [:bottom bottom-symbol]
  [:load-built-ins-p {t | nil}]
  [:errorp {t | nil}]

creates a new domain domain (a symbol or a string). Options:
values-list is a list of attributes whose values will be hidden at definition time. attribute-list is a list of attributes that will be hidden (including their values) at definition time. symbol-list is a list of symbols to be exported from the domain package. These three symbol lists can also be declared in the :declare part (cf. Section 4.5). If :load-built-ins-p t is specified, the standard file of TDL built-in types will be included, otherwise, nothing will be loaded. The default is t.
:top and :bottom define the name of the top and bottom symbol; the default is the value of the global variables *TOP-SYMBOL* and *BOTTOM-SYMBOL*. A domain can be given documentation using the :documentation keyword; its value must be a string. If :errorp t is specified, it will cause error to redefine a domain. If the value of :errorp is nil (default), only a warning will be given. Example:
#<DOMAIN MY-DOMAIN>
<TDL>

• function deldomain domain [:errorp {t | nil}][:delete-package-p {t | nil}]
deletes domain domain (symbol or string), which includes all type, instance, and template definition as well as the corresponding package with package name domain, including all symbols defined in this package. If :errorp t is specified, using an undefined domain name will cause an error. If :errorp nil is specified (default), a warning will be given and the current domain will not be changed. If :delete-package-p t is specified, the corresponding package will be deleted. Otherwise (default), only all domain-specific information (type definitions, global hashtables etc.) will be deleted. Example:
< TDL > deldomain :MY-DOMAIN.
<TDL>

Definitions from the standard tdl-built-ins.tdl file are shown below. They will be included automatically if :load-built-ins-p t (which is also the default) is specified in defdomain.

begin :declare.
  built-in: fixnum, bignum, integer. ;; only COMMON LISP types can be
  built-in: atom, string, symbol. ;; declared as built-ins
  sort: *null*. ;; is the type of the empty list <>
  sort: *built-in*. ;; the top built-in type
  sort: *sort*. ;; the top sort type
  sort: *undef*. ;; make *undef* a sort in order to
                  ;; exclude features defined on it
end :declare.

begin :type.
  *avm* := [ ]. ;; the top avm type
  atom := *built-in*. ;; now specify the subtype
  integer := atom. ;; relation for the built-ins
  fixnum := integer.
  bignum := integer.
  symbol := atom.

4 SYNTAX OF TDL
4.3 The Structure of TDC Grammars

A TDC grammar may consist of type definitions, sort declarations, template definitions, instance specifications and control information. In addition, it is possible to mix different domains and LISP code. The structuring feature in TDC grammars is the begin and end statement, which is comparable to BEGIN/END blocks in PASCAL, or the \texttt{Latex} environments. Environments determine syntactic as well as semantic contexts for different purposes.

Some environments provide the necessary preconditions for enclosed definitions, e.g., the domain environment supports definitions of entire type lattices, the necessary context for type definitions. Others such as the control environment are completely optional.

Another constraint is that template definitions precede the type or instance definitions that contain template calls (cf. macros in programming languages).

Environments can be nested freely (the so-called environment stack keeps track of this) and distributed over different grammar files. The loading of files may also be nested (cf. the include documentation).

The TDC prompt indicates the current domain and the current environment. Its general output format is \texttt{<domain:environment>}, e.g.,

\texttt{<MY-DOMAIN:TEMPLATE>}

If no domain is active, the prompt is \texttt{<TDL>}. Typically, a TDC grammar starts with the definition of a domain (\texttt{defdomain}), changes to this domain (begin : domain), declares sorts (begin/end : declare), defines templates (begin/end : template), feature types (begin/end : type), and, finally, instances (begin/end : instance).

Example:

\begin{verbatim}
;;; The structure of tdl grammars. An example.
defdomain :grammar. ;; this includes a default initial type hierarchy
defdomain :junk-domain :load-built-ins-p NIL.
set-switch *verbose-p* nil. ;; set verbosity
\end{verbatim}
begin :domain :grammar.

;; we start with semantics
begin :declare.
  include "grammar/semantic-sorts".
  sorts : organism, moveable, nonmoveable, physical, selfmoving,
        nonselfmoving, animated, non-animated, vehicle.
  sign = word ^ phrase.
end :declare.

begin :template. ;; load template definitions
  include "grammar/semantic-templates".
end :template.

begin :type. ;; type definitions
  selfmoving :- moveable :- physical.
  nonmoveable :- physical.
  nonselfmoving :- moveable.
  animated :- moveable.
  non-animated :- moveable.
  organism := selfmoving & animated.
  vehicle := selfmoving & non-animated.

Figure 1: The initial type hierarchy (with :load-built-ins-p t)
4.4 Domain Environment

A domain environment starts with

```lisp
begin :domain domain.
```

and ends with

```lisp
end :domain domain.
```

*domain* must be the name of a previously defined domain, i.e., a quoted symbol, a string or a keyword symbol. *begin :domain* pushes *domain* to the global stack *DOMAIN*, while *end :domain* pops it.

Arbitrary TDC statements or other environments may be enclosed by the domain environment.

All symbols (sort, type, template, instance names) occurring between *begin :domain* and *end :domain* are defined or looked up in *domain*.

All other environments except the :lisp environment must be enclosed by at least one domain environment. See Section 4.2 for more details.

4.5 Declare Environment and Declarations

A declare environment starts with

```lisp
begin :declare.
```

and ends with

```lisp
end :declare.
```
It may appear anywhere within a domain environment, and may contain arbitrary declarations, other environments and TDC statements. Declare environments declare

- built-in sort symbols: built-in[s] : type{, type}*. built-in declares a COMMON LISP type to be a sort in TDC, e.g., integer, string, etc. A built-in sort can be unified only with its subsorts or with an atom that has the corresponding COMMON LISP type. An example is shown in section 4.6.2. Note that the standard TDC type *built-in* itself is an ordinary sort but is not declared to be a built-in type. Its only purpose is to be the super-sort of all predefined sorts in TDC such as *null* and the atomic sorts, e.g., atom, string, etc.

- sort symbols: sort[s] : type{, type}*. sort declares types to be sorts (singleton sorts in Smolka's terminology). Sorts always live in a closed world, i.e., their unification fails unless they subsume each other or have a greatest lower bound. This declaration is optional. It is equivalent to the definition type :< *sort*, in the :type environment which may be faster at definition time in the current implementation.

- incompatible types: nil = type{ & type}+. declares the types to be incompatible. This declaration is useful for avm types in an open world (i.e., if *AND-OPEN-WORLD-REASONING-P* has value t).

- exhaustive partitions: supertype = type{ | type}*. declares an exhaustive partition (cf. [Krieger & Schäfer 94a1]).

- disjoint exhaustive partitions: supertype = type{ ~ type}*. declares an exhaustive disjoint partition (cf. [Krieger & Schäfer 94a1]).

All other relations between types (conjunctions, disjunctions) must be defined in the :type environment. The type hierarchy for avm types will be inferred from the avm type definitions. In addition, the following domain specific symbol lists can also be declared in this environment (cf. Section 4.2).

- hide-value[s] : identifier{, identifier}*. specifies the attributes whose values are to be hidden at definition time,

- hide-attribute[s] : identifier{, identifier}*. specifies the attributes (including their values) to be hidden at definition time,

- export-symbol[s] : identifier{, identifier}*. specifies the symbols to be exported from the domain package.

All declarations must be specified directly in a declare environment, and nowhere else.

4.6 Type Environment and Type Definitions

A type environment starts with

```
begin :type.
```

and ends with

```
end :type.
```

It may appear anywhere within a domain environment, and may contain arbitrary type definitions, other environments and TDC statements. The syntax for subsort and subtype definitions is

```
type{ :< type}+{, option}*. 
```

Subsort or subtype definitions define the order of sorts or types without feature constraints. They are not necessary if feature constraints are specified for a subtype. Example:

```
*avm* :< *top*.
```
4.6 Type Environment and Type Definitions

but
*cons*- := *avm* & [ first, rest ].
In both cases, the left hand side is a subtype of the right hand side type.
The general syntax of type definitions¹ is

type := body {, option}*

type is a symbol, the name of the type to be defined. The complex BNF syntax of body is given in
Section 4.1.2. Some examples are presented on the following pages.

4.6.1 Feature Structure Definitions

The TDL syntax for a feature structure type person-number-type with attributes PERSON and NUMBER is

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

The definition results in the structure

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

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, feature structures, disjunctions, distributed disjunctions,
coreferences, lists, functional constraints, template calls, or negated values.

4.6.2 Atoms

In TDL, an atom can be a number, a string or a symbol. Symbols must be quoted with a single quote ‘
(otherwise they will be interpreted as sorts or avm types). Atoms can be used as values of attributes or as
disjunction elements.

Example: The TDL type definition

pl-3-phon := [ NUMBER 'plural,
PHON "-en",
PERSON 3 ].
results in the structure

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

An example of atoms as disjunctive elements is shown in Section 4.6.9.
Atoms are not ordered hierarchically (as is the case for sorts). An atom only unifies with itself, the top type
of the type hierarchy or, if defined, with a built-in sort of an appropriate data type, i.e., integer atoms unify
with the built-in sorts integer and number, the string atoms unify with the built-in sort string, symbolic
atoms unify with the built-in sort symbol. These three sorts are defined in the standard built-in file tdl-
built-ins.tdl. Built-in sorts may also be user-defined. One need only define an appropriate COMMON
LISP type, e.g.

begin :lisp.
(DEFTYPE Even-Integer () '(AND INTEGER (SATISFIES EVENP)))
end :lisp.
begin :declare.
built-in : Even-Integer.
end :declare.

These lines define a COMMON LISP type Even-Integer and declare it as a built-in sort in TDL. This sort
unifies only with even integer atoms.

¹For nonmonotonic definitions see Section 4.6.17.
4.6.3 Paths

Paths may be abbreviated using dots between attribute names, e.g.

\[ P1 := [\text{DTRS.HEAD-DTR.CAT.SYN.LOCAL.SUBCAT } \text{ hi }] \]

yields structure

\[
[ P1 \text{ DTRS [HEAD-DTR [CAT [SYN [LOCAL [SUBCAT hi]]]]]}]
\]

4.6.4 Logical Operators

In TDL, values (atoms, types, feature structures) may be combined by the logical operators & (conjunction, unification, inheritance), | (disjunction), ^ (exclusive or), and ~ (negation). These operators may be freely nested, where ~ has highest priority, and & binds stronger than | and ^. Parentheses may be used to break this binding or to group operands. Example:

\[ l := [a \land (~b \lor c [r 1])] \]

Important note for Emacs users: If you type in the \ symbol in the emacs lisp mode, you must quote it with a backslash, i.e., \|. Otherwise, the Emacs lisp mode will wait for another ‘closing’ \l. When you are writing grammar file (preferably in TDL mode, of course), you can omit the backslash.

4.6.5 Type Specification and Inheritance

All conjunctive feature structures can be given a type specification. Type specification at the top level (empty feature path) of a type definition means inheritance from a supertype. The feature definition of the specified type will be unified with the feature term to which it is attached when type expansion takes place. The inheritance relation represents the definitional dependencies of types. Together with multiple inheritance (described in the following section), the inheritance relation can be seen as a directed acyclic graph (DAG).

An example of type specification inside a feature structure definition follows:

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

Expanding this definition results in the structure

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

Now an example of type inheritance at the top level:

\[ \text{pl-type} := \text{person-number-type} \land \text{NUMBER 'plural} \]

Expanding this definition results in the structure

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

This feature structure is called the global prototype of pl-type: an expanded feature structure of a defined type which has (possibly) inherited 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 or skeleton. So the local prototype of pl-type is

\[
\begin{array}{c}
\text{person-number-type} \\
\text{NUMBER plural}
\end{array}
\]

Section 5.18 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 structure being defined will be assumed to have the top type of the hierarchy.

### 4.6.6 Multiple Inheritance

Multiple inheritance is possible at any level. A glb (greatest lower bound) type is not required to exist if the global variable \*AND-OPEN-WORLD-REASONING-P\* has value t.

Suppose number-type, person-type and gender-type are defined as follows:

\[
\begin{align*}
\text{number-type} & := [\text{NUMBER}]. \\
\text{person-type} & := [\text{PERSON}]. \\
\text{gender-type} & := [\text{GENDER}].
\end{align*}
\]

Then the TDC type definition

\[
\text{mas-2-type} := \text{number-type} & \text{person-type} & \text{gender-type} & [\text{GENDER 'mas, PERSON 2}].
\]

would result in the following structure (after type expansion):

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

### 4.6.7 Coreferences

Coreferences indicate information sharing between feature structures. In TDC, coreference symbols may occur anywhere in the value of an attribute. If values are specified, they are attached to the coreference tag by the & connector. The order of the elements of such a conjunction does not matter.

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 structures, the coreference symbols will be normalized to an integer number. Example:

\[
\text{share-pn} := [\text{SYN #pn} & \text{person-number-type}, \text{SEM #pn}].
\]

results in the following structure (after type expansion):

\[
\begin{array}{c}
[\text{share-pn}] \\
[\text{person-number-type}] \\
[\text{SYN [] PERSON [] NUMBER []}] \\
[\text{SEM []}]
\end{array}
\]
4.6.8 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 (they may be type identical but must not be token identical).

The syntax of negated coreferences is

```
¬(#(a₁, a₂, ... aₙ))
```

where a₁, a₂, ... aₙ are coreference symbols, i.e., numbers or symbols, without the hash sign. If n = 1, the parentheses can be omitted.

Example: The TDC definition

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

would result in the following structure:

```
give
  RELN give
  GIVER ¬(1,2)
  GIVEN #1
  GIVEE #2
```

4.6.9 Simple Disjunctions

Disjunction elements are separated by | (or \ in the Emacs interaction mode, cf. Section 4.6.4). 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 95] for the reasons).

Distributed disjunctions provide a restricted way to use coreferences to outside disjunction alternatives (Section 4.6.10).

An example of 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

```
person-1-or-2
  SYN { [person-number-type] PERSON 1 }
  SYN { [person-number-type] PERSON 2 }
```

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
  SYN [person-number-type]
```

Disjunctions at the top level of a type definition introduce disjunctive types (depicted as bold edges in the TDC grapher). Arbitrary combinations of sorts, atoms, and feature structure types are allowed. Example:
*list* := *null* | *cons*. ;; where *null* is a sort, *cons* an avm

The resulting feature structure (after type expansion) is:

```
{ "*cons* 
  FIRST [ ]
  REST [ ]
  *null* }
```

The only case where no disjunctive edges are introduced in the type hierarchy is a disjunction of pure atoms, e.g.,

```
  num := 1 | 2 | 3.
```

* instead of | means exclusive-or. Disjoint type partitions can be declared in the declare environment (Section 4.5).

### 4.6.10 Distributed Disjunctions

A very useful feature of TDC, implemented in the underlying unification system UDiNe, is distributed (or named) disjunction [Eisele & Dörre 90]. This kind of disjunction has a specification restricted to a local domain, although it may affect more than one attribute. The main advantage of distributed disjunction is a more succinct representation. 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 TDC expression:

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

When a structure of type `season-trigger` is unified with the structure `[SEASON "summer" | "fall"]`, the value of attribute `NUMBER` become 2 | 3, i.e., the value of attribute `SEASON` triggers the value of attribute `NUMBER`, and vice versa. The syntax of the alternative list in distributed disjunctions is

```
%i(a_1, \ldots, a_n)
```

where `i` is an integer number or a symbol, the disjunction index for each group of distributed disjunctions (`%1` in the example). More than two alternative lists per index are allowed. All alternative lists with the same index must have the same number (`n`) of alternatives. The disjunction index is local in every type definition and is normalized to a unique index when unification of feature structures takes place.
In general, if alternative $a_{ij}$ ($1 \leq j \leq n$) does not fail, it selects the corresponding alternative $b_{ij}, c_{ij}, \ldots$ in all other alternative lists with the same disjunction index $i$.

As in the case of simple disjunctions, disjunction alternatives must not contain coreferences referring 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 of such a distributed disjunctions with coreferences is:

```plaintext
dis2 := [ a %name( [], #1 , #2 ),
b %name( [c +], x&[d #1 g&[m -]], x&[d #2 g&[m +]] ) ].
```

4.6.11 Lists

In $TDC$, lists are represented as first-rest structures with distinguished attributes FIRST and REST, where the sort $^*null*$ at the last REST attribute indicates the end of a list (and, of course, the empty list). The input of lists can be abbreviated by using the <> syntax which is only syntactic sugar.

```plaintext
list-it := [ MYLIST < first-element, #second, [] >,
            SECOND #second,
            EMPTY <> ].
```

The resulting feature structure is

```plaintext
list-it
  list
    FIRST first-element
      rest
        MYLIST
          REST
            FIRST [ ]
              rest
                list
                  FIRST [ ]
                    REST *null*
                      EMPTY *null*
```

Dotted pair lists can be defined à la LISP, e.g.

```plaintext
dot-list := [ DOTLIST < first-element, second . #rest >,
              DOTREST #rest ].
```
The resulting feature structure is

```
    dot-list
    DOTLIST
    DOTREST[]
```

Lists with open end can be defined by using `...` as the last element. The value of the last REST attribute will be `[]` (top).

In addition, the `<...>` notation can be combined with type and length restrictions. The general syntax is

```
<...> [ : (restriction) ] : type [ : (minlength, maxlength) ] : length ]
```

The following definition produces a list with element type `x` of length 0, 1 or 2 at the top level attribute `a`.

```
list012-of-x := [a <...> : x : (0, 2)].
```

```
    [ *cons* ]
    FIRST x
    REST *

    A { [ *cons* ]
        FIRST x
        REST *
        REST *null*
        REST *null*
    }

    [ *null* ]
```

### 4.6.12 Difference Lists

Difference lists are first-rest structures with distinguished attributes FIRST and REST, and a special LAST attribute at the top level, which shares the value with the last REST attribute.

Difference lists permit appending of list structures without requiring an append type or an append relation.

In TDC, the elements of difference lists may be enclosed in `<! !>` as an abbreviation. Example:

```
difflist3 := <! [], [], [] !>.
```

```
    [ *cons* ]
    FIRST []

    LIST
    REST *

    [ *cons* ]
    FIRST []

    REST *

    LAST []
```
4.6.13 Negation

The ~ sign indicates negation. Example:

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

The resulting feature structure is

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

Negation of types will be pushed down to atoms according the schema of [Smolka 88; Smolka 89]. If *list* list is defined as in the tdl-built-ins.tdl file (page 4.2), the definition

\[
\text{notlist} := \sim *\text{list} .
\]

will result in the following (expanded) structure:

\[
\begin{array}{c}
\sim *\text{cons} \sim *\text{null} \\
\sim *\text{null} \\
\sim *\text{null} \\
\sim *\text{undef} \\
\sim *\text{null} \\
\sim *\text{undef}
\end{array}
\]

Here *undef* indicates undefined attributes. It is an atom that unifies with nothing else.

4.6.14 External Coreference Constraints

Instead of specifying the values of coreferences within a feature structure, it is also possible to add a list of such constraints at the end of a feature type definition. The syntax is

\[
\ldots \text{where } \{ \text{constraint} | \ldots \text{constraint} \} .
\]

where constraint \( \rightarrow^{\#\text{coref-name}} \{ \text{function-call} | \text{disjunction} \} \)

Here, ‘\( \ldots \)’ mean the body of a type, instance or template definition. External coreference constraints are pure syntactic sugar, but may be useful, e.g. for expressing the identity of coreferences in very complex definitions, or as variables, e.g. where \( (#8 = #9, \text{undef} = *\text{undef}) . \)

function-call is explained in the following section.

4.6.15 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 TDLC system.

The syntax of functional constraints is similar to that of external coreference constraints, i.e., functional constraints must be specified outside a feature structure, but are connected with it through a coreference tag, cf. last section.

\[
\text{function-call} \rightarrow \text{function-name} \{ \text{disjunction} | \ldots \text{disjunction} \} .
\]

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

\[
\text{add-prefix} := [ \text{WORD } \#\text{word}, \text{PREFIX } \#\text{prefix}, \text{WHOLE } \#\text{whole} ] .
\]

where \( (\#\text{whole} = \text{String-Append}(\#\text{prefix}, \#\text{word})) .\)

The definition of the LISP function String-Append is shown in the example in Section 4.11. The usual representation for functional constraints is:

\[
\begin{array}{c}
\text{add-prefix} \\
\text{WORD } [1] \\
\text{PREFIX } [2] \\
\text{WHOLE } [3]
\end{array}
\]
4.6 Type Environment and Type Definitions

**Functional Constraints:**

\[ S = \text{String-Append}(2, \{1\}) \]

The evaluation of functional constraints will be postponed until all parameters are instantiated (residuation; cf. [Ait-Kaci & Nasr 86; Smolka 91] for theoretical backgrounds). The evaluation can be enforced by using the function EVAL-CONSTRAINTS of the UNIFY package. Further details are described in [Backofen & Weyers 95].

### 4.6.16 Template Calls

Templates are pure textual macros which allow specification of (parts of) type or instance definitions by means of some shorthand. The definition of templates will be explained in Section 4.8. A template call simply means the syntactic replacement of a template name by its definition and possibly given parameters. Thus we restrict templates to be non-recursive.

The syntax of template calls is

\[ \text{@templ-name ( [ templ-par \{, templ-par}]* )} \]

where a templ-par is a pair consisting of a parameter name (starting with the $ character), followed by =, 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 4.8 for further details and examples.

### 4.6.17 Nonmonotonicity and Value Restrictions

TDC supports nonmonotonic definitions for avm types and instances, called single link overwriting (SLO). A type can be overwritten by a set of paths with associated overwrite values. The general syntax for nonmonotonic definitions is

\[ \text{identifier} = \text{nonmonotonic} \{ \text{where} \{ \text{constraint} \{, constraint\}*\} \{, \text{option}\}*\}. \]

where

- **nonmonotonic** \rightarrow type \& [ overwrite-path \{, overwrite-path\}*\]

and

- **overwrite-path** \rightarrow identifier \{. identifier\}* disjunction

This feature of TDC can be used to model defaults. A special extension of typed unification will handle nonmonotonic unification in a future version of TDC [Krieger 93]. Currently, one has to be careful when using this feature. A suitable application would be lexical types that normally will not be unified with a nonmonotonically defined lexicon entry.

Note that the \[ ... \] syntax denotes a set of overwrite paths with associated overwrite values. This is different from the \[ ... \] notation known from avm type or instance definitions, because everything following a path specification identifier \{. identifier\}* is the overwrite value and will replace all inherited information at this path.

begin :type.

\[ a := [ \text{person}_x : \text{integer}, \text{person}_y : \text{integer} ]. \]
\[ b := a \& [ \text{person}_x 1 \mid 2, \text{person}_y 1 \mid 2 ]. \]
\[ c != b \& [ \text{person}_x 3 ]. \]
\[ d != b \& [ \text{person}_x "three" ]. \]
\[ ;;; \text{error: restriction violated} \]
end :type.

The expanded prototype of \( c \) is

\[
\begin{bmatrix}
\text{PERSON}_X \\
\text{PERSON}_Y
\end{bmatrix}
\]
If TDC has been compiled with the #+TDL-Restriction compiler option, the :restriction type specification at attributes will be checked before paths are overwritten. If an overwrite value is not equal to or more specialized than the type specified in the definition of the structure to be overwritten, an error will be signaled:

Error: Restriction INTEGER is inconsistent with overwrite value
(ATOM "three") under path PERSON_X in D

Restart actions (select using :continue):
0: Continue; overwrite restriction anyway.

4.6.18 Rules

Different parsers use different representations for rules. The --> syntax allows abstraction from the internal representation of rules as feature structures. A user-definable function is responsible for translating the abstract representation into the desired format. Rules can be defined as types and as instances. A sophisticated representation of rules, e.g., as used in the DISCO system, even allows inheritance from rule types. Rule definition syntax is

\[
\text{identifier} \ := \ \text{disjunction -->list } \{ \text{where (constraint \{, constraint\}*)} \}
\]

where disjunction represents the left hand side of the rule and list contains the right hand side in the TDC list syntax. Examples:

\[
\star \text{rule}^* \ := \ [\ ] \ \text{-->} \ < \ldots >.
\]

\text{binary-rule} := \ [\ ] \ \text{-->} \ < [\ ], [\ ] >.

\text{np} := \ \text{binary-rule} \ \text{-->} \ < \text{Det}, \text{Noun} >.

\text{xp} := \ \text{yp} \ & \ \text{zp} \ \text{-->} \ < \text{Cat1}, \text{Cat2}, \ldots >.

\text{ap} := \ (\text{bp} \ & \ \text{cp}) \ \text{I} \ \text{dp} \ \text{-->} \ < \#1 \ & \text{Cat1}, \text{Cat2}, \#1 >.

\text{tp} := \ \text{sp} \ & \ [\text{attr} \#a \ & \text{foo}] \ \text{-->} \ < \text{fee}, \text{fum} \ & \ [\text{rtta} \#a \ & \#b], \text{fuu} \ & \#b >.

We show two kinds of representation of the rules here, taking the np type from above.

\[
\text{np} \ \Rightarrow \ \left[ \begin{array}{c}
\text{binary-rule} \\
\star \text{cons}^* \text{FIRST det} \\
\text{ARGS} \\
\text{REST} \ \star \text{cons}^* \\
\text{FIRST noun} \\
\text{REST} \ *,\text{null}^* \\
\end{array} \right]
\]

\text{internal representation of the DISCO parser}

\[
\text{np} \ \Rightarrow \ \left[ \begin{array}{c}
\star \text{cons}^* \\
\text{FIRST binary-rule} \\
\star \text{cons}^* \text{FIRST det} \\
\text{REST} \\
\text{REST} \ *,\text{null}^* \\
\end{array} \right]
\]

\text{internal representation of another parser}
The name of the function that generates feature structures from the rule syntax can be set in the global variable *WHICH-PARSER*, its default value is the symbol TDL-PARSE::CONSTRUCT-BERNIE for the DISCO parser.

### 4.7 Optional Keywords in Definitions

For external use, TDC provides a number of optional specifications which are basically irrelevant to the grammar (except for controlled expansion). If the optional keywords are not specified, default values will be assumed by the TDC control system. Optional keywords are author:, doc:, date:, status:, and expand-control:. If a keyword is given, it must be followed by a value.

The values of author:, doc: and date: must be strings. The default value of author: is defined in the global variable *DEFAULT-AUTHOR*. The default value of doc: is defined in the global variable *DEFAULT-DOCUMENTATION* (see Section 5). The value of date: is a string containing the current time and date. If not specified, this string will be automatically generated by the system.

The status: information is necessary if the grammar is to 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, 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.

The expand-control: keyword can be used to specify control information for type expansion. It has the same effects as the defcontrol statement for type with index nil, see Section 4.10.

### 4.8 Template Environment and Template Definitions

A template environment starts with

```
begin :template.
```

and ends with

```
end :template.
```

It may appear anywhere within a domain environment, and may contain arbitrary template definitions, environments and TDC statements.

Templates in TDC are what parametrized macros in programming languages are: 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 in writing grammars that are modular; they can also keep definitions independent (as far as possible) from specific grammar theories.

Note that template definitions must not be recursive. Recursive definitions are only allowed for avm types.

The general syntax of a TDC template definition is

```
templ-name ([templ-par {, templ-par}]*) := body {, option}.
```

where templ-par is a pair consisting of a parameter name (starting with the $ character), followed by =, 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. body can be a complex description as in type definitions.

**Example:** The template definition

```
a-template ($inherit = *avm*, $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:

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

while

\[
\text{inside-call} := \begin{bmatrix}
  \text{top-attr} \textbf{a-template} (\$\text{value} = \text{"hello"}, \\
  \$\text{attrib} = \text{MY-PHON})
\end{bmatrix}.
\]

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

\[
\begin{bmatrix}
  \text{inside-call} \\
  \text{TOP-ATTRIB} \textbf{[]}
  \begin{bmatrix}
    \text{*avm*} \\
    \text{MY-PHON "hello"} \\
    \text{COPY "hello"}
  \end{bmatrix}
\end{bmatrix}
\]

Disjunction and coreference names in template definitions are local to each template expansion. In this sense, templates are similar to COMMON LISP macros.

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

Section 5.17 describes the function describe-template which prints information about template definitions.

### 4.9 Instance Environment and Instance Definitions

A type environment starts with

```
begin :instance.
```

and ends with

```
end :instance.
```

It may appear anywhere within a domain environment, and may contain arbitrary instance definitions, environments and \(\text{TDC}\) statements.

An instance definition is similar to a type definition, but instances are not part of the type hierarchy although they can inherit from types. For instance, each lexical entry will typically be an instance of a more general type, e.g., \textit{intransitive-verb-type} with additional specific graphemic and semantic information. The idea is to keep the type lattice as small as possible. The distinction between types and instances is similar to that of classes and instances in object oriented programming languages, e.g., CLOS.

Instances are not inserted into the \(\text{TDC}\) type hierarchy. In general, instances are objects (feature structures) which can 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 \(\text{TDC}\) instance definition\(^5\) is

```
\text{instance-name} := \text{body} \{, \text{option}\}^*.
```

\textit{body} can be a complex description as in type definitions. \textit{options} in instance definitions are the optional keywords author:, doc:, date:, and status:.

If the same name is given more than once for an instance of the same type, the old instances will not be destroyed and the parser is responsible for the access to all instances. This behavior can be controlled by the global variable \(\text{*ACCUMULATE-INSTANCE-DEFINITIONS*}\).

\(^5\) For nonmonotonic definitions see section 4.6.17.
If the status: keyword is given, the status value of the instance will become the specified one. If no status option is given, the status will be inherited from the top level types.

The values of author:, doc: and date: must be strings. The default value of author: is defined in the global variable *DEFAULT-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.

### 4.10 Control Environment

A control environment starts with

```lisp
begin :control.
```

and ends with

```lisp
end :control.
```

It may appear anywhere within a domain environment, and must be enclosed directly by either an instance or a type environment. It may contain arbitrary control definitions, other environments and TDL statements. The control environment is supported only for the sake of clarity in order to structure a TDL grammar. The environment itself is completely syntactic sugar. Control information can be given either in a type or instance definition by the optional :expand-control keyword, or through the defcontrol statement in a type or instance environment. The control environment additionally allows specification of control information separately from the type or instance definitions, e.g., in a separate file.

Note that the control environment needs to be enclosed by a domain environment and either a type or an instance environment, depending on what kind of definitions are to be expanded.

The macro

```lisp
defcontrol { type | instance | :global } expand-control[: index number].
```

defines control information for the expansion of a type or an instance. For further details see section 5.4.

### 4.11 Lisp Environment

A LISP environment starts with

```lisp
begin :lisp.
```

and ends with

```lisp
end :lisp.
```

The LISP environment allows insertion arbitrary LISP code into TDL grammars. Example:

```lisp
begin :lisp.
(DEFUN String-Append (&rest args)
  (APPLY #'CONCATENATE 'STRING args))
end :lisp.
```

This DEFUN defines the function String-Append used in the example of Section 4.6.15.

There is also a short notation for evaluating LISP expressions from TDL: The macro

```lisp
leval Common Lisp Expression.
```

evaluates Common Lisp Expression in any environment. For the sake of clarity, we recommend using this statement only in the interactive mode. Example:

```lisp
leval (LOAD-SYSTEM "my-parser").
```

### 4.12 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. A comment associated with a specific type, template or instance definition should be given in the doc: string at the end of the definition.

#| | and | | can be used to comment regions (as in COMMON LISP).
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 defdomain.

5.1 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 *AND-OPEN-WORLD-REASONING-P***
  
  *possible values: t or nil*
  
  This variable controls whether avm types live in an open or in closed world. Cf. [Krieger & Schäfer 94a].

- **Global variable *SIGNAL-BOTTOM-P***
  
  *possible values: t or nil*
  
  If t, an error is signaled if the conjunction of two types is bottom.

- **Global variable *IGNORE-BOTTOM-P***
  
  *possible values: t or nil*
  
  Default value: nil
  
  If t, typed unification skips over bottom. The result of an inconsistent type conjunction will be the bottom type, and feature unification will be continued as if the conjunction would be consistent. This is useful to debug a grammar.

- **Global variable *WARN-IF-TYPE-DIDES-NOT-EXIST***
  
  *possible values: t or nil*
  
  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.

- **Global variable *WARN-IF-REDEFINE-TYPE***
  
  *possible values: t or nil*
  
  Default value: t
  
  This variable controls whether a warning will be signaled if a type already exists and is about to be redefined.

- **Global variable *ACCUMULATE-INSTANCE-DEFINITIDNS***
  
  *possible values: t or nil*
  
  Default value: nil
  
  If t, redefining an instance will push the new definition onto a stack. Otherwise, new definitions will replace older ones.

- **Global variable *DEFAULT-AUTHOR***
  
  *possible values: string*
  
  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.

- **Global variable *DEFAULT-DOCUMENTATION***
  
  *possible values: string*
  
  Default value: ""
  
  This parameter specifies the default documentation string for type, template and instance definitions. It will be used as default value for the optional keyword doc: in type, template and instance definitions.

- **Global variable *VERBOSE-P***
  
  *possible values: t or nil*
  
  Default value: nil
  
  This parameter specifies the verbosity behavior during processing of type definitions. If the value is t, a verbose output will be generated. If the value is nil, only the name of the (successfully) defined type will be printed in brackets, e.g., #Avm<VERB-TYPE>. 


5.1 Global Switches and Variables

- **Global variable *VERBOSE-READER-P***
  default value: nil
  possible values: t or nil
  This parameter specifies the verbosity behavior of the TDL reader. If the value is nil, the TDL reader does not print values that are returned from function calls and type, instance, template definitions. Otherwise, the first return value will be printed.

- **Global variable *VERBOSE-EXPANSION-P***
  default value: nil
  possible values: t or nil
  This parameter specifies the verbosity behavior when type expansion takes place. If the value is nil, TDL type expansion will only print messages when types are not defined or inconsistent. Otherwise, a verbose trace of the expansion will be printed.

- **Global variable *TRACE-P***
  default value: nil
  possible values: t or nil
  If t, verbose trace information is printed by the TDL parser, the definition functions, and the expansion algorithm.

- **Global variable *LAST-TYPE***
  default value: undefined
  possible values: a type-symbol
  This variable contains the name of the last type defined. It is used by the print functions pgp, plp, lgp, llp, fgp, flp, and by expand-type, delete-type, and reset,proto when no type name is specified. The value of this variable can be changed by the user.

- **Global variable *LAST-INSTANCE***
  default value: undefined
  possible values: an instance-symbol
  This variable contains the name of the last instance defined. It is used by the print functions pg1, pl1, lgi, lli, fgi, fli, and by expand-instance, delete-instance, and reset-instance when no instance name is specified. The value of this variable can be changed by the user.

- **Global variable *USE-MEMOIZATION-P***
  default value: t
  possible values: t or nil
  If nil, no memoization of simplified type expression will take place. Otherwise, the domain-specific hash tables will be used.

- **Global variable *EXPAND-TYPE-P***
  default value: nil
  possible values: t or nil
  If nil, feature structures, i.e., avm types and instances, will not be expanded at definition time. This saves time and space at definition time. If t, expansion will run on the prototypes with the control information known so far.

- **Global variable *SIMPLIFY-FS-P***
  default value: t
  possible values: t or nil
  If not nil, feature structure simplification will be performed at the end of type or instance expansion. Feature structure simplification may remove unnecessary fails in disjunctions, and hence may speed up subsequent unifications.

- **Global variable *BUILD-INTERMEDIATE-TYPES-P***
  default value: nil
  possible values: t or nil
  This global variable controls whether TDL introduces intermediate types for certain complex formulae recognized during the parsing of type definitions which, however, are not specified by the user, i.e., these types will not occur on the left side of a type definition. If the value is t, intermediate types will always be created (at any level of a feature structure). If nil, intermediate types are not created, except for the top level of a type definition in order to classify the new type correctly.

- **Global variable *USE-INTERMEDIATE-TYPES-P***
  default value: t
  possible values: t or nil
  This global variable controls whether intermediate types generated at definition or at run time will be
used as abbreviations during typing. If nil, intermediate type will not be used. This variable will be
used in type definitions as well as in instance definitions.

- Global variable *CREATE-LEXICAL-TYPES-P*
  default value: t
  possible values: t or nil
  If t, TDc will be forced to introduce an intermediate type at the top level of an instance definition
  (if necessary). At all other levels of an instance definition, the variables *BUILD-INTERMEDIATE-
  TYPES-P* and *USE-INTERMEDIATE-TYPES-P* control the behavior of intermediate type genera-
  tion.

- Global variable *WARN-UNIFICATION-P*
  default value: nil
  possible values: t or nil
  Normally, no unification takes place at definition time. But there are some infrequent cases, e.g., if
  the grammar writer specifies a conjunction of two feature structures ([a [b x]] & [b y]), which
  make unification necessary. If *WARN-UNIFICATION-P* is nil, no warning will be given when
  unification is performed.

- Global variable *SOURCE-GRAMMAR*
  default value: user's home directory pathname
  possible values: a pathname
  This variable contains the default prefix for grammar files, if no absolute pathname is specified for
  filename in the include statement.

- Global variable *LOAD-BUILT-INS-P*
  default value: t
  possible values: t or nil
  This variable contains the default value for the :load-built-Ins-p keyword in the include
  statement.

- Global variable *WARN-NORMAL-FORM-P*
  default value: nil
  possible values: t or nil
  This variable determines whether a warning is given if a TDc expression is not in normal form (only
  at definition time). If nil, no such warning will be given.

- Global variable *UPDATE-GRAPHER-OUTPUT-P*
  default value: nil
  possible values: t or nil
  This variable controls whether an automatic redraw is performed on the grapher when a type is
  (re)defined. If nil, no automatic redraw will be done.

- Global variable *NORMALFORM-OPERATOR-SYMBOL*
  default value: :and or :or
  possible values: :and or :or
  This variable contains the operator for the normal form of type expressions. Either disjunctive or
  conjunctive normal form is possible.

- Global variable *TOP-SYMBOL*
  default value: *TOP*
  possible values: a symbol
  This variable contains the name for the top type of type hierarchies. This is the default for the :top
  keyword in the defdomain function.

- Global variable *BOTTOM-SYMBOL*
  default value: *BOTTOM*
  possible values: a symbol
  This variable contains the name for the bottom type of type hierarchies. This is the default for the
  :top keyword in the defdomain function. It is also used for the generation of symbol names for
  bottom types.

- Global variable *TOP-SORT*
  default value: *SORT*
  possible values: a symbol
  This variable contains the name for the top sort of type hierarchies. This is the default for the
  declaration of sorts if no super-sort is specified.
5.2 Setting Switches and Global Variables

- Global variable *LIST-TYPE-SYMBOL*  
  possible values: a symbol  
  default value: *LIST*  
  This variable contains the name for the first/rest list type.

- Global variable *CONS-TYPE-SYMBOL*  
  possible values: a symbol  
  default value: *CONS*  
  This variable contains the name for the first/rest cons type.

- Global variable *DIFF-LIST-TYPE-SYMBOL*  
  possible values: a symbol  
  default value: *DIFF-LIST*  
  This variable contains the name for the difference list type.

- Global variable *END-OF-LIST*  
  possible values: a symbol  
  default value: *NULL*  
  This variable contains the name for the end-of-list type (sort).

- Global variable *FIRST-IN-LIST*  
  possible values: a symbol  
  default value: FIRST  
  This variable contains the name for the FIRST attribute in first/rest lists.

- Global variable *REST-IN-LIST*  
  possible values: a symbol  
  default value: REST  
  This variable contains the name for the REST attribute in first/rest lists.

- Global variable *LAST-IN-LIST*  
  possible values: a symbol  
  default value: LAST  
  This variable contains the name for the LAST attribute in difference lists.

- Global variable *LIST-IN-LIST*  
  possible values: a symbol  
  default value: LIST  
  This variable contains the name for the LIST attribute in difference lists.

5.2 Setting Switches and Global Variables

- macro set-switch identifier Common-Lisp-Expression.  
  This macro sets the value of a global value with name identifier (cf. Section 5.1). Example:  
  <MY-DOMAIN:TYPE> set-switch *WARN-IF-TYPE-DOES-NOT-EXIST* NIL.

- function print-switch identifier.  
  This function prints the value of a global variable. Example:  
  <MY-DOMAIN:TYPE> print-switch *AUTHOR*.

5.3 Including Grammar Files

The function  
include filename.  
includes TDL grammar files. If no begin:domain is specified at the beginning of the file, the definitions are loaded in the current domain. include files may be arbitrarily nested.  
filename should be a string containing a filename or a path. If no absolute filename path is specified, the default path of variable *SOURCE-GRAMMAR* is taken.  
TDL filenames must have the .tdl extension. If no such extension is specified, it will be appended automatically.  
Example:  
<MY-DOMAIN:DECLARE> include "my-declarations".  
is equivalent to  
<MY-DOMAIN:DECLARE> include "my-declarations.tdl".
5.4 Expanding Types and Instances

Type expansion means replacing the type names in typed feature structures by their definitions. Partial expansion can be done according to the control information given by the defcontrol statements or the expand-control keywords. Default is full expansion of all types.

All expand functions store the expanded structures in the global prototype slot of the type or instance infon (cf. Section 5.8). The local prototype (skeleton) always remains unchanged.

5.4.1 Defining control information: defcontrol

The control information for the expansion algorithm may be specified globally and/or locally, either in a separate file or mixed with the type and instance definitions.

The begin :control, ... end :control, environment can be used to indicate control information, but this is not necessary.

The syntax of macro defcontrol is:

```
defcontrol { type | instance | :global } expand-control[:index index].
```

The first parameter in the defcontrol statement is a symbol, either the name of a type or the name of an instance (this depends on the surrounding environment), or :global which indicates global control information. A defcontrol can be anywhere in a grammar file, even before the corresponding type definitions. A newer defcontrol declaration (with the same type and index) will replace an older one (this is also true for global control).

5.4.2 Expanding Types and Instances: expand-type and expand-instance

The syntax is given by:

```
expand-type [[:index index] [:expand-control expand-control]].
expand-instance [[:index index] [:expand-control expand-control]].
expand-all-types [[:index index] [:expand-control expand-control]].
expand-all-instances [[:index index] [:expand-control expand-control]].
```

Additional internal functions such as expand-fs exist in order to destructively expand anonymous feature structure or parts of it, e.g., from within a parser, etc.

The index parameter may be used to define different prototypes of the same type that are (possibly) only partially expanded. Each prototype needs its own defcontrol.

The indexed prototypes of a type can be 'spliced' into a feature structure through type expansion using the :expand-only or :expand slot of the control information.

Instance indices (only integers are allowed) can be used to define different levels of expanded lexicon entries, etc.

The default index is nil which is the standard prototype. If no special control information is given (locally or globally), the nil index specifies a fully expanded prototype.

The :skeleton index may be useful at the :expand-only or :expand slot. It denotes the unexpanded definition of a type (its skeleton). :skeleton cannot be used as an argument for the defcontrol keyword :index, because a skeleton is always unexpanded and expansion is permitted.

expand-type will generate a new prototype with index index from a copy of the :skeleton of type type if this index does not exist. If it exists, and is not already fully expanded, it will be expanded again.

5.4.3 The Syntax of expand-control

If expand-control is specified for expand-instance or expand-type, the values of slots that are omitted will be inherited from global control. Control information which has been defined for the type or instance with the same index will be ignored.

If some slots in defcontrol :global are omitted, they will be taken from global variables with corresponding names: *MAXDEPTH*, *ATTRIBUTE-PREFERENCE*, *EXPAND-FUNCTION*, *RESOLVED-PREDICATE*, *IGNORE-GLOBAL-CONTROL*, *ASK-DISJ-PREFERENCE*, or *USE-DISJ-HEURISTICS*, or
5.4 Expanding Types and Instances

**Figure 2:** Types, skeletons, prototypes and indices: Type `xyz`'s prototype is either unexpanded or contains no AVM types. Thus, its prototypical feature structure is identical with its definition (skeleton). Type `uvw` has a (fully) expanded prototype and a user-defined prototype which are both (possibly partially) expanded copies of `uvw`'s skeleton feature structure.

*USE-CONJ-HEURISTICS*. Their value can be set with `set-switch`. The global switch `VERBOSE-EXPANSION-P` can be set to `t` for verbose trace of type expansion. Default value is `nil` (quiet).

If some slots in local `expand-control` are omitted, they will be inferred from global `expand-control`. The syntax of `expand-control` is as follows.

```
expand-control → ( [ (:expand ( {type | (type [index ?pred])} {path} ) ) ] | (:expand-only ( {type | (type [index ?pred])} {path} ) ) ) | (:delay ( {type | (type [pred])} {path} ) ) | (:maxdepth integer) | (:ask-disj-preference {t | nil}) | (:attribute-preference {identifier} ) | (:use-conj-heuristics {t | nil}) | (:use-disj-heuristics {t | nil}) | (:expand-function {depth | types} -first-expand) | (:resolved-predicate {resolved-p | always-false} ) | (:ignore-global-control {t | nil}) )
```

```
path → {identifier | pattern} . {identifier | pattern} *
```

```
pattern → ? | * | + | ![identifier] | ![+]
```

```
pred → eq | subsumes | extends | ...
```

```
index → integer for instances
```

```
integer | identifier | string for AVM types
```
Now we describe the expand-control slots.

- :expand-function
  This slot specifies the basic expansion algorithm. The default expansion algorithm is depth-first-expand with prototype memoization and a special treatment of recursive types (a combination of breadth first for recursive and depth first for non-recursive types).
  The alternative is a combined 'types-first' algorithm for non-recursive types and breadth first for recursive types. This 'types-first' algorithm is advantageous only if feature structures with many delayed types are to be fully expanded (e.g., at run time). The behavior for recursive types is the same as with the proper depth first algorithm. (:expand-function types-first-expand) selects the 'types-first' algorithm.

- :delay
  The delay slot specifies a list of types whose expansion is deferred. For each type, a comparison predicate pred (eq, subsumes, extends, or user-defined, default is eq) and a list of paths or path patterns can be defined.
  Path patterns can facilitate path specifications. * denotes zero or more features, + one or more features, ? exactly one feature. In each case, the prefix ?identifier can be used to define variables for features or path segments. The variables are local to each path pattern. ?identifier without a * or ? suffix is the same as ?identifier?, i.e., one feature variable. Example:

  ```lisp
  defcontrol mytype ((:delay (vtype syn.loc.* sem.?x.head.?x)
                      ((ntype subsumes) *)))
  ```
  Here, expansion of the type vtype will be delayed under all paths which start with syn.loc and all paths which start with sem, then an arbitrary feature (bound to variable x), then head, then the second feature again (constraint by variable x). Expansion of the type ntype and all its subtypes will be delayed under all paths.

- :expand-only and :expand
  There are two mutually exclusive type expansion modes. If the :expand-only list is specified, only types in this list will be expanded, all others will be delayed. If the :expand list is specified, all types will be expanded. Types not mentioned in the list will be expanded using the default prototype index nH, i.e., fully, if not specified otherwise.
  In both cases, the types in the :delay list will be delayed anyway.
  index specifies the index of the prototype to be spliced in. pred is as described in the paragraph before (:delay).

- :maxdepth
  If (:maxdepth integer) is specified, all types at paths longer than integer will be delayed anyway.
  This feature may be used as a brute force method to stop infinite expansion.

- :attribute-preference
  This slot defines a partial order on attributes. The sub-feature structures at the leftmost attributes will be expanded first. This may speed up expansion if no numerical preference data is available.
  Example:

  ```lisp
  defcontrol :global ((:attribute-preference first rest last head-dtr
                      comp-dtrs front back))
  ```

- :ask-disj-preference
  If this flag is set to t, the expansion algorithm interactively asks for the order in which disjunction alternatives should be expanded. Example:
5.4 Expanding Types and Instances

Ask-Disj-Preference in G under path X
The following disjunctions are unexpanded:
Alternative 1:
   (:Type A :Expanded NIL) []
Alternative 2:
   (:Type B :Expanded NIL) []

Which alternative in G under path X should be expanded next (1, 2, or 0 to leave them unexpanded, or :all to expand all alternatives in this order, or :quiet to continue without asking again in G)?

- :use-conj-heuristics and :use-disj-heuristics
  [Uszkoreit 91] suggested that exploitation of numerical preference information for features and disjunctions would speed up unification. These slots control the use of this information in conjunctive and disjunctive structures respectively.

- :resolved-predicate
  This slot specifies a user definable predicate that may be used to stop recursion. The description of such predicates is be rather complex and is omitted here. The default predicate is always-false which will make the expansion algorithm complete (if no delay or maxdepth restriction is given, of course).

- :ignore-global-control
  If this flag has value t, the values of the three globally specified lists :expand-only, :expand, :delay will be ignored. If nil, locally and globally specified lists will be taken into account.

5.4.4 Printing Control Information
The TDC statement (macro)
   print-control { type | instance | :global }. 
prints the control information in an internal format with path patterns replaced by a special syntax.

   function print-recursive-sccs [:domain domain].
prints the strongly connected components of the recursive dependency graph computed so far. It contains recursive types recognized so far (by type expansion). Example:
   print-recursive-sccs.
   ((*CONS* *LIST*) (APPEND APPEND1))

5.4.5 How to Stop Recursion
Type expansion with recursive type definition is undecidable in general, i.e., there is no complete algorithm that halts on arbitrary input (type definitions) and decides whether a description is satisfiable or not. However, there are several ways to stop infinite expansion.

- The first method is part of the expansion algorithm. If a recursive type occurs in a typed feature structure that is to be expanded, and this type has been expanded under a subpath, and no features or other types are specified at this node, then this type will be delayed, since it would expand forever (this is called lazy expansion). An example of a recursion that stops like this is the recursive version of the list type (see below). A counter example, i.e., a type that will not stop without a finite input (using the default resolved predicate always-false and no delay pattern), is Alt-Kaci's append type [Alt-Kaci 86]. That's life.

Expanding append with finite input will stop, of course; an example of this is the last type definition in the code below.
defdomain :append :load-built-ins-p NIL.
begin :domain :append.
begin :declare.
  sort: *null*. ;; the empty list
end :declare.
begin :type.
  *avm* := [ ]. ;; the top avm type
  *list* := *null* | *cons*.
  *cons* := *avm* & [FIRST,REST *list*].

;;; Ait-Kaci's version of APPEND
append0 := *avm* & [FRONT < >,
  BACK #1 & *list*,
  WHOLE #1].
append1 := *avm* & [FRONT <#first . #rest1>,
  BACK #back & *list*,
  WHOLE <#first . #rest2>,
  PATCH append & [FRONT #rest1,
     BACK #back,
     WHOLE #rest2]].
append := append0 | append1.
r:=append & [FRONT <'a,'b>,
  BACK <'c,'d>].
expand-type 'r.

Full expansion of r results in the following structure.

```
[ r
  WHOLE (λa. λb. (c. (d. ())))]
  [append1
    PATCH [append0
      FRONT λ()
      BACK λ
      WHOLE λ]
    FRONT λ
    BACK λ]
  FRONT λ
  BACK λ]
```

- The second way is brute force: use the :maxdepth slot to cut expansion at a suitable path depth.
- The third method is to define :delay patterns or to select the :expand-only mode.
- The fourth method may work in some cases (Prolog hackers may like it): Use the :attribute-preference list to define the 'right' order for expansion.
- The last method is to define a suitable :resolved-predicate for a class of recursive types. For further details, see [Schäfer 95].
5.5 Checking Welltypedness/Appropriateness

TDL supports optional welltypedness checks at run time as well as at definition time. The appropriateness specification for a feature is inferred by the type definition of the most general type that introduces this feature. This is done by the function

\[
\text{compute-approp}[\text{domain \ domain}][\text{warn-if-not-unique \ \{t \ | \ nil\}].}
\]

Its optional keyword \text{warn-if-not-unique} determines whether a warning is given if there is more than one most general type that introduces a feature\(^6\). The function is called by the two functions described below if necessary.

The function

\[
\text{print-approp}[\text{domain \ domain}].
\]

prints the current appropriateness table of a domain. This table is comparable to the \text{Approp} function in [Carpenter 93, Chapter 6]. But there, it is defined \text{Approp} : \mathcal{F} \times \mathcal{T} \mapsto \mathcal{T}, i.e., for all features and types, while TDL stores \text{Approp} : \mathcal{F} \mapsto \mathcal{T} \times \mathcal{T} only once for each feature and infers the admissible value types by a (cheap) lookup at the prototypical feature structure of the (sub)type of the type which introduces the feature.

A feature structure is welltyped if each feature has an appropriate type and if the type of its \text{value} is equal to or more specific than the value type of its appropriateness specification.

The welltypedness check can be done

1. at definition time. The global variable \text{*CHECK-WELLLYPEDNESS-P*} (values: t or nil) controls whether this check is done (t) or not (nil). This check enforces expansion at definition time.

2. at run time. The global variable \text{*CHECK-UNIFICATION-WELLLYPEDNESS-P*} (values: t or nil) controls whether this check is done (t) or not (nil). The global variable \text{*RETURN-FAIL-IF-NOT-WELLLYPED-P*} (values: t or nil) controls whether a unification failure is triggered if the unified nodes are not welltyped (t).

3. for a specific type or instance. The function

\[
\text{check-welltypedness}[\text{type \ instance \ :all \ :instances \ :avms \ [\text{domain \ domain}][\text{index \ index}][\text{verbose \ \{t \ | \ nil\}]].}
\]

provides such a check for a single type or instance as well as for all types or instances with the specified index.

The global variable \text{*VERBOSE-WELLLYPEDNESS-CHECK-P*} controls whether a warning is given if a welltypedness check is negative (t).

Below we show a brief example output of \text{print-approp}.

<table>
<thead>
<tr>
<th>Feature</th>
<th>((Intro-Type . Value-Type)*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUE</td>
<td>((NON-LOCAL-TYPE . <em>TOP</em>))</td>
</tr>
<tr>
<td>SLADJ</td>
<td>((NON-LOCAL-TYPE . <em>TOP</em>))</td>
</tr>
<tr>
<td>SLASH</td>
<td>((NON-LOCAL-TYPE . <em>TOP</em>))</td>
</tr>
<tr>
<td>HOUR</td>
<td>((TIME-VALUE . <em>TOP</em>))</td>
</tr>
<tr>
<td>NON-LOC</td>
<td>((NON-LOCAL . NON-LOCAL-TYPE))</td>
</tr>
<tr>
<td>SUBJ-SC</td>
<td>((SUBJ-SUBCAT-TYPE . <em>TOP</em>))</td>
</tr>
<tr>
<td>LIST</td>
<td>((<em>DIFF-LIST</em> . <em>TOP</em>))</td>
</tr>
<tr>
<td>SEM-MOOD</td>
<td>((QUESTION-SEMANTICS . SYMBOL))</td>
</tr>
<tr>
<td>SUBCAT</td>
<td>((SUBCAT-TYPE . <em>TOP</em>))</td>
</tr>
<tr>
<td>FILLER-DTR</td>
<td>((FILLER-DTR-TYPE . MAX-SIGN-TYPE))</td>
</tr>
</tbody>
</table>

\(^6\)Carpenter 93, Chapter 6 calls such an appropriateness condition unacceptable and stipulates that there exists exactly one most general type which introduces a feature. TDL is not so restrictive, but the warnings can be employed to write grammars that do not make use of such ‘unacceptable’ appropriateness conditions. Our treatment is comparable to \text{polyfeatures} in CUF [Dörre & Dorna 81].
USEFUL FUNCTIONS, SWITCHES, AND VARIABLES

5.6 Deleting Types and Instance Definitions

- function delete-type [type [:domain domain]].
  deletes a type (avm or sort). It removes type from the avm/sort hashtable in domain domain and from the type hierarchy. Example:
  \textless MY-DOMAIN:INSTANCE\textgreater delete-type 'my-type.

- function delete-instance [instance [:domain domain] [:index number]].
  removes instance with index number (default: 0) from the instance hashtable in domain domain. Example:
  \textless MY-DOMAIN:INSTANCE\textgreater delete-instance 'root-node :index 0.

- function delete-all-instances [domain].
  removes all instances from the instance hashtable in domain domain (default: current domain). Example:
  \textless MY-DOMAIN:INSTANCE\textgreater delete-all-instances.

5.7 Resetting Prototypes of Types and Instances

A global prototype is a (possibly partially) expanded feature structure of an avm type or of an instance. If a type or an instance is not expanded at all, or if its definition is already fully expanded, then the global prototype is the same as its local one (its definition or skeleton), i.e., they are identical.

- function reset-proto [type [:domain domain] [:index index]].
  resets the prototype of an avm type to its skeleton.

- function reset-all-protos [domain].
  resets all prototypes of all avm types in domain.

- function reset-instance [instance [:domain domain] [:index number]].
  resets the prototype of an instance to its skeleton.

- function reset-all-instances [domain].
  resets the prototypes of all instances in domain.

5.8 Accessing Internal Information (Infons)

The following functions apply a functional argument function, a COMMON LISP function, e.g., a print function, collector, etc., to the slots of the internal representation (infon structures) of avms, sorts, instances and templates.

domain specifies the domain (default: current domain).
name must be the name of a sort, avm, template or instance.
table may be one of :avms (the default) :sorts, :templates, :instances.
accessor may be one of the following slot accessor functions: name (the default), surface, domain, intermediate, comment, author, date, value-types, restriction-types, atomic-symbols, attributes, expand-control, skeleton, prototype, creation-index, monotonic, overwrite-values, overwrite-paths.

There is an additional accessor function parameters which can be applied only to template infons.
The additional accessors class-info and mixed? can only be applied to type infons.
5.9 Collecting and Printing Statistical Information

The TDC system can be compiled with or without the statistics module. If the system is compiled with statistics, the following functions are defined:

  applies a function to infon of name in table and domain. Example:

  `<MY-DOMAIN:TYPE> do-infon :name 'my-instance
    :table :instances
    :accessor #'author.

  applies a function to all infons in one table (sorts, avms, templates, instances) in one domain. Example:


- print-all-names [table [domain]] .
  prints all names in one table in one domain, it is just a special case of do-all-infons. Example:

  `<MY-DOMAIN:TEMPLATE> print-all-names :templates.

5.9 Collecting and Printing Statistical Information

The TDC system can be compiled with or without the statistics module. If the system is compiled with statistics, the following functions are defined:

  prints all statistical information that is available. If domain is not specified, this will be done for all domains.

  prints all statistical information that are domain specific. If domain is not specified, the current domain is assumed.

  prints expansion statistics. If domain is not specified, the current domain is assumed.

  prints all statistical information that is domain independent.

  prints type simplification statistics. If domain is not specified, the current domain is assumed.

- function reset-all-statistics [:domain domain] .
  resets all statistical information. If domain is not specified, this will be done for all domains.

- function reset-domain-statistics [:domain domain] .
  resets all statistical information that is domain specific. If domain is not specified, the current domain is assumed.

- function reset-expand-statistics [:domain domain] .
  reset expansion statistics. If domain is not specified, the current domain is assumed.

- function reset-global-statistics .
  resets all statistical information that is domain independent.

- function reset-simplify-statistics [:domain domain] .
  resets type simplification statistics.
function count-nodes {type | instance | :all}
[:table {:avms | :instances}]
[:expand-p {t | nil}]
[:verbose {t | nil}]
[:domain domain]
[:index index]
[:stream stream]
[:filename {nil | filename}].

counts the number of nodes in an avm type or instance with the specified index (default is nil for
types and 0 for instances). Instead of a name, the :all keyword can be specified to count all nodes
in all instances or types with index. In this case, :verbose t will output the number of nodes for
each type or instance. Otherwise, only the total will be printed.

The :filename or :stream argument can be used to redirect the output to a file or a stream (default:
standard output). :expand-p t will expand structures before counting if necessary (default is nil).
When called from Lisp, the function returns 9 values (integers) in the order as below. Here is an
example output:

Total number of nodes in all instances:
# of conj avm nodes: 13868
# of atomic nodes: 5564
# of sortal nodes: 3697
# of attributes: 24717
# of disj nodes: 644
# of disj elements: 1606
# of fail nodes: 0
# of undef nodes: 0
# of shared nodes: 2763
total # of nodes: 23773

If domain is not specified, the current domain is assumed.

5.10 Memoization

At definition time as well as at run time, type expressions are simplified (syntactically and semantically)
and stored in memoization hashtables. In each domain, there are four memo tables: for conjunctive and
disjunctive normal form and with and without exploiting information from the type hierarchy.

function clear-simplify-memo-tables [:domain domain] [:threshold integer].
clears the simplification memo tables. If the :threshold number is specified, only entries that have
been used less than or equal to integer times will be removed from the memo tables. If integer is nil,
all entries will be removed (default). The :threshold keyword is only supplied with the statistics
module.

function print-simplify-memo-tables [:domain domain] [:threshold integer].
prints the contents of the simplification memo tables. If the :threshold number is specified, only
entries that have been used more than integer times will be printed. If integer is nil, all entries will
be printed (default). The :threshold keyword is only supplied with the statistics module.

function save-simplify-memo-table [:domain domain] [:filename string]
[:threshold integer].
saves type simplification memoization table to a file. The :threshold keyword is only supplied
with the statistics module. All entries occurring less than integer times will not be saved.

function load-simplify-memo-table [:domain domain] [:filename string].
loads type simplification memoization table from a file (written with tune-types or save-simplify-
memomemo-table). This may speed up subsequent unifications.
5.11 Tuning up Unification: Training Sessions

In this section, some tools will be described that may be used to speed up unification at run time. A training session is necessary

1. to extract type definitions for GLB types if the result of the unification of their expanded definitions is consistent,

2. to generate a table of sets of types that are inconsistent (or consistent) with other types (as a result of their feature constraints).

Such a training session consists of the following steps

1. load (and expand) a grammar
2. call Start-Collect-Unified-Types.
3. do train parses
4. call tune-types.

To work with the tuned types later on, simply type

1. include "glb-types". ;;; load the additional GLB type definitions (optional)
2. load-simplify-memo-table [:domain domain] [:filename "cnf-memo-table"].

before run-time. Of course, the user is responsible for updates of the files if the type hierarchy as changed.

- function start-collect-unified-types [:domain domain] .
  This function enables a training session, resets some variables, and clears some tables.

  loads types simplification memoization table from a file (written with tune-types or save-simplify-memo-table).

- function print-unified-types [:filename string] [:domain domain] .
  prints contents of the global hashtable :unified-types to screen (or to file if filename string is given).

- function tune-types [:domain domain]
  [:threshold integer]
  [:unify-input-file {nil | string}]
  [:create-glbs {nil | t}]
  [:hashtable hashtable]
  [:assume-consistency {nil | t}]
  [:memo-output-file string]
  [:glb-output-file string] .

Main function. Either takes the current unify table (default) or loads one from file :unify-input-file that has been written with print-unified-types.

Tune-Types creates a :memo-output-file (default name "cnf-memo-table") as well as a :glb-output-file (default name "glb-types.tdl"). The first file can be loaded at run time with load-simplify-memo-table, the second one with include.

If :create-glbs nil is specified, no glb types will be introduced (default: t) and no glb output file will be created.

:threshold integer specifies a threshold for the entries in the table of unified types to be considered. Only entries that have occurred at least integer times will be considered (default: 0=all entries).

:assume-consistency t is a sensible default because it assumes that all type expressions that occur as an argument of an entry in the unify table are consistent. Otherwise it would unify (expand) all arguments.

:hashtable is one of :simplify-cnf-hierarchy (default), :simplify-cnf, :simplify-dnf-hierarchy, or :simplify-dnf and specifies the corresponding type memoization hashtable.
5.12 Defining Reader Macros

The alias facility allows extension of the TDL syntax by adding new macros that may abbreviate TDL syntax or integrate other modules like parsers or other user shells.

```lisp
macro alias identifier {definition-string | nil} [help-string].
```

defines a user macro with name `name` (string or symbol) and definition `definition-string`. `definition-string` must start with a COMMON LISP function or macro name (without surrounding parentheses), followed by arbitrary arguments.

Arguments specified with a TDL reader macro call will be passed to the COMMON LISP function or macro by simply appending them at the end of the `definition-string`. If `definition-string` is nil, then `name` will be defined to call a function or macro with the same name. In this case, the corresponding symbol must be exported from the TDL or COMMON-LISP package.

`help-string` should contain a string with a brief description of the reader macro. It will be printed with the help command (see below). Example: The `message` command described in the following paragraph is defined as a reader macro as follows.

```lisp
alias "MESSAGE" "FORMAT T" "print a message (Lisp's FORMAT syntax)".
```

5.13 Printing Messages

During parsing the grammar files, the function

```lisp
message string {Common Lisp Expression}•.
```

can be used to print messages. The args may be variable names or LISP function calls as in the COMMON LISP FORMAT function. Example:

```lisp
message "Default author is "*A" *DEFAULT-AUTHOR*."
```

5.14 Help

`help [statement | :all]`.  
`help :all` (default) prints a list of all statements (readermacros) that are defined. If a `statement` name is specified, a brief description associated with the readermacro will be printed. Example:

```lisp
<DISCO:TYPE> help begin.
Help for begin: begin a TDL definition block.
```

5.15 Wait

`wait`.  
waits until the return key is pressed on COMMON LISP'S *TERMINAL-IO* (useful for demos etc.).

5.16 Exit TDL

`ldt`.  
quits the TDL syntax reader and returns to COMMON LISP.

5.17 Getting Information about Defined Templates

`function describe-template template-name`.  
prints a short information text about a template definition. Example:
5.18 Printing Feature Structures

For debugging and documentation purposes, it is possible to print prototypes of the defined feature types and instances. This can be done by using the following functions. For all kinds of representation (ASCII, \textsc{pegramed} or \LaTeX{}), the print modes described in section 7 will be considered.

5.18.1 Printing to the Interactive Screen or to Streams (ASCII)

The following four functions call the print function \textsc{print-fs} of the \textsc{UDWNe} system which is defined in package \textsc{unify}. It prints feature structures either to the standard output (default) or to streams, e.g., text files. For internal details we refer to the \textsc{UDWNe} documentation. The output format of the \textsc{TDC} type entries is described in this manual in section 7.

- \textbf{function plp [ type \{print-option\}* ]}.
  plp prints the \textit{local prototype} of the feature structure with name \textit{type}. If no type name is specified, plp prints the prototype of the \textit{last} type defined before evaluating plp. The \textit{local prototype} (or skeleton) contains only the \textit{local} information given in the definition of \textit{type}. Example:
  
  \begin{verbatim}
  <MY-DOMAIN:TYPE> plp 'mas-sg-agr : init-pos 12 : hide-types T.
  [ GENDER : [FEM : -
  MAS : +]
  NUM : SG]
  \end{verbatim}

- \textbf{function pgp [ type \{print-option\}* ]}.
  pgp prints the \textit{global prototype} of the feature structure with name \textit{type}. If no type name is specified, pgp prints the prototype of the \textit{last} type defined before evaluating pgp. The \textit{global prototype} contains all information that has been inferred for \textit{type} by type expansion so far. Example:
  
  \begin{verbatim}
  <MY-DOMAIN:TYPE> pgp 'mas-sg-agr.
  (:TYPE MAS-SG-AGR) [GENDER : GENDER-VAL [FEM : -
  MAS : +]
  CASE : []
  NUM : SG]
  \end{verbatim}

- \textbf{function pli [ instance \{print-option\}* ]}.
  pli prints the \textit{local prototype} of the instance with name \textit{instance}. If no instance name is specified, pli prints the prototype of the \textit{last} instance defined before evaluating pli. The \textit{local prototype} (or skeleton) contains only the \textit{local} information given in the definition of \textit{instance}.

- \textbf{function pgi [ instance \{print-option\}* ]}.
  pgi prints the \textit{global prototype} of the instance with name \textit{instance}. If no instance name is specified, pgi prints the prototype of the \textit{last} instance defined before evaluating pgi. The \textit{global prototype} contains all information that has been inferred for \textit{instance} by expansion so far.

\textbf{print-options} are the following optional keywords:

- \textbf{:remove-topsflag}
  \textit{default value: nil}
  possible values: \{t|nil\}
  If \textit{flag} is \textit{t}, attributes with empty values (i.e., values that unify with everything, i.e., the top type of the
hierarchy ([]) will not be printed. If flag is nil, all attributes (except those in label-hide-list) will be printed.

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

- :label-sort-list (identifier)*
  
  default value: the value of *LABEL-SORT-LIST*

  possible values: a list of symbols (attribute names)

  the 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 the list. All remaining attributes which are not member of the list will be printed at the end.

- :stream stream
  
  default value: t

  possible values: {t | nil | a COMMON LISP stream}

  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 (space characters) for the feature structure to be printed.

- :read-in-mode flag
  
  default value: nil

  possible values: {t | nil}

  If t, the feature structure is printed in a way such that the output could be used by UDNe's input function build-fs. Otherwise a pretty print is done. To be read in, feature structures have to be printed with print mode :read-in (see section 7). Otherwise, type information may be incomplete.

5.18.2 FEGRAMED

FEGRAMED is a feature structure editor [Kiefer & Fettig 94]. It can be started from TDC through the function fegramed.

Feature structures from TDC can be passed to FEGRAMED using the following commands.

- function flp [type {fegramed-option}]*
  
  flp starts FEGRAMED with the local prototype of the feature structure with name type. If no type name is specified, flp takes the prototype of the last type defined before evaluating flp. The local prototype (or skeleton) contains only the local information given in the definition of type type.

  Example:
  
  `<MY-DOMAIN:TYPE> flp 'mytype.'

- function fgp [type {fegramed-option}]*
  
  fgp starts FEGRAMED with the global prototype of the feature structure with name type. 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 has been inferred for type type by type expansion so far.

  Example:
  
  `<MY-DOMAIN:TYPE> fgp 'mas-sg-agr :wait t :hide-types t.

- function fli [instance {fegramed-option}]*
  
  fli starts FEGRAMED with the local prototype of instance instance. If no instance name is specified, fli takes the prototype of the last instance defined before evaluating fli. The local prototype (or skeleton) contains only the local information given in the definition of instance.

  Example:
  
  `<MY-DOMAIN:INSTANCE> fli 'agr-en-type.`
function \( \texttt{fgi} \{ \text{instance \{fegramed-option\}} \} \).

\( \texttt{fgi} \) starts \textsc{fegramed} with the global prototype of instance \textit{instance}. If no instance name is specified, \( \texttt{fgi} \) takes the prototype of the last instance defined before evaluating \( \texttt{fgi} \). The global prototype contains all information that has been inferred for instance \textit{instance} by expansion so far.

\[\begin{array}{c}
\text{VERB-AGR-TYPE} \\
\text{SYNTAX-TYPE} \\
\text{LOCAL-TYPE} \\
\text{HEAD-TYPE} \\
\text{MAJ: [VCAT-TYPE} \\
\text{FIN-MAJOR-VAL} \\
\text{FIN: [FIN-VAL} \\
\text{AGR: [VAGR-VAL} \\
\text{NUM: [TOP} \\
\text{PERS: [TOP} \\
\text{TENSE: [TOP} \\
\text{INF: [NIL} \\
\text{NI: +} \\
\text{V: +} \\
\text{MIN: [TOP} \\
\text{MOD: [TOP} \\
\text{SUBJ: [TOP} \\
\text{DP-NOM-TYPE} \\
\text{SYNTAX-TYPE} \\
\text{LOCAL-TYPE} \\
\text{HEAD: [DHEAD-TYPE} \\
\text{INFL: [S-GRADE} \\
\text{NOM: [NOM-VAL} \\
\text{OBL: [OBL} \\
\text{GENDER: [TOP} \\
\text{NUM: [TOP} \\
\text{GRADE: ST} \\
\text{PERS: [TOP} \\
\text{MAJ: [NCAT-TYPE} \\
\text{N: +} \\
\text{V: +} \\
\text{MIN: [TOP} \\
\text{MOD: [TOP} \\
\text{SUBJ: [TOP} \\
\text{SUBCAT: [TOP} \\
\text{SUBJ-SC: [TOP} \\
\text{Y-SUBCAT: [TOP} \\
\text{LPE: -} \\
\text{SUBCAT: [NULL} \\
\text{SUBJ-SC: [NIL} \\
\text{V-SUBCAT: [NIL} \\
\end{array}\]

Figure 3: A feature structure type in \textsc{fegramed}

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

- \texttt{:filename \textit{filename}}
  
  \textit{default value: "type-gpi.fed", "type-lp.fed", "instance-lii.fed", or "instance-gii.fed"}
  
  \textit{possible values:} a string or a LISP path name

Unless \texttt{filename} is specified, a filename will be 'computed' from the type or instance name and the index \( i \) of the instance or the global prototype. The file will be created by the \textit{TDC-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.

• **file-only flag**
  
  *default value: nil*
  
  possible values: {t|nil}

  If flag is t, the FEGRAMED interface function will only generate an output file, but not execute the FEGRAMED program on it. If flag is nil, the file will be generated and FEGRAMED will be called.

Further details are described in [Kiefer & Fettig 94]. An example screen dump of a feature structure in FEGRAMED is shown in Figure 3.

### 5.18.3 TDL::2ft.TPC

TDL::2ft.TPC is a tool which generates \[\text{TLP}\] compatible high-quality output of TDL feature structure types [Lamport 86; Goossens et al. 94].

**TDL Interface Functions to TDL::2ft.TPC**

• function `llp` [type `{latex-option}`*].
  
  `llp` starts TDL::2ft.TPC with the *local* prototype (skeleton) of the feature structure with name *type*. If no type name is specified, `llp` takes the prototype of the *last* type used before evaluating `llp`. The *local* prototype (LP) contains only the *local* information given in the definition of type *type*. Example:

  ```latex
  <MY-DOMAIN:TYPE> llp 'agr-en-type :fontsize "small"
    :doc-header \"\documentstyle[a4,times]{article}\".
  </MY-DOMAIN:TYPE>
  ```

• function `lgp` [type `{latex-option}`*].
  
  `lgp` starts TDL::2ft.TPC with the *global* prototype of the feature structure with name *type*. If no type name is specified, `lgp` takes the prototype of the *last* type used before evaluating `lgp`. The *global* prototype (GP) contains all information that has been inferred for type *type* by type expansion so far. Example:

  ```latex
  <MY-DOMAIN:TYPE> lgp 'agr-en-type :mathmode "equation".
  </MY-DOMAIN:TYPE>
  ```

• function `lli` [instance `{latex-option}`*].
  
  function `lgi` [instance `{latex-option}`*].

  `lli` and `lgi` start TDL::2ft.TPC with the feature structure of instance *instance*. The *local* instances (LI) contain only the *local* information given in the definition of instance (skeleton). The *global* instances (GI) contain all information that has been inferred for instance *instance* by expansion so far. Example:

  ```latex
  <MY-DOMAIN:INSTANCE> lli 'head-initial-rule :index 0.
  </MY-DOMAIN:INSTANCE>
  <MY-DOMAIN:INSTANCE> lgi 'head-initial-rule :index 0.
  ```

The optional keywords *latex-options* are described in section 5.18.3.

There is also a function `latex-fs` which operates on feature structures analogously to UD\[\text{Ve}\]'s `print-fs`. It roughly takes the same arguments as `lgp` etc.

An example of a complex feature structure generated by TDL::2ft.TPC is shown in figure 4.

```latex
<MY-DOMAIN:TYPE> lgp 'count-noun-sem-type :label-sort-list '(first rest)
   :align-attributes-p nil
   :coreftable '((1. "Sem"))
</MY-DOMAIN:TYPE>
```
Optional Keyword Arguments

`latex-options` are the following optional keywords:

- **filename** `filename`
  - default value: "type-gpi", "type-lp", "instance-gii", or "instance-lii"
  - possible values: string
    - Unless `filename` is specified, a filename will be 'computed' from the type or instance name and the index of the instance or global prototype. 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 will be executed. The default for `pathname` is 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. The print mode options are described in section 7.

- **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** `({identifier}* )`
  - default value: value of `*LABEL-HIDE-LIST* = ()`
  - possible values: a list of symbols (attribute names)
    - Attributes in the list will not be printed.

- **label-sort-list** `({identifier}* )`
  - default value: value of `*LABEL-SORT-LIST* = ()`
  - possible values: a list of symbols (attribute names)
    - The 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 the list. All remaining attributes which are not member of the list will be printed at the end.

- **shell-command** `command`
  - default value: value of `*SHELL-COMMAND* = "td12latex"`
  - possible values: {nil|string}
    - If `command` is nil, only the \LaTeX{} file will be created and \TeX{} will return. If `command` is a string, \TeX{} will start a shell process and execute `command` with parameter `filename`. An example for `command` is the following shell script with name `td12ps` which starts `\LaTeX` with the output file of `TDC2iTEx` and generates a PostScript file using [Rokicki 93]'s DVIPS.

```
#!/bin/sh
#td12ps generates PostScript file
latex $1
dvips $1 -o $1.ps
```

The following script `td12epsf` generates an encapsulated PostScript file (EPSF). When generated with a PostScript font (such as option times in the document header), the EPSF file can be used to scale a feature structure in order to fit into an arbitrary box (e.g., in \TeX{} documents using \verb|\epsfbox|, see [Rokicki 93]). To achieve this, the output file of `TDC2iTEx` must consist of exactly one page. Large feature structure may lead to 2 or 3 pages of output. In this case, add \verb|\textheight80cm \textwidth40cm| or so to the file header generated by \TeX{}. Then \TeX{} should generate one-page output that can be scaled arbitrarily. If \TeX{} stack size is too small to process large feature structures, recompilation of \TeX{} with increased stack size will help.
USEFUL FUNCTIONS, SWITCHES, AND VARIABLES

```bash
#!/bin/sh
#tdl2epsf generates encapsulated PostScript file (EPSF)
latex $1
dvips $1 -E -o $1.epsf

tdl2x generates a dvi file and runs xdvi on it.

#!/bin/sh
#tdl2x generates a dvi file and starts xdvi
latex $1
xdvi $1
```

- **:wait flag**
  - *default value*: value of variable *WAIT* = nil
  - *possible values*: {t|nil}
    - If flag is nil and command is not nil, the shell command command will be started as a background process. Otherwise, TDL2LATEX 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 variable *COREFSIZE* = nil
  - *possible values*: {nil|string}
    - 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 name for printing coreference symbols. string must contain a valid LATEX or user defined font name, 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 variable *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.
5.18 Printing Feature Structures

- **:doc-header** string  
  *default value:* value of \*DOC-HEADER*  
  This parameter sets the \LaTeX\ documentstyle or \documentclass header if \latex-header-p is \texttt{t}. Default value is "\documentstyle{article}". It could be replaced by a document style with additional options such as "a4", "times", etc., or, for new \LaTeX\ [Goossens et al. 94], by "\documentclass[article] \usepackage{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 variable \*TYPESTYLE* = \texttt{:infix}  
  possible values: { \texttt{:infix} | \texttt{:prefix} }  
  If \texttt{style} has value \texttt{:infix}, complex type entries will be printed in infix notation (e.g., \texttt{a \& b \& c}). If \texttt{style} has value \texttt{:prefix}, complex type entries will be printed in prefix (LISP like) notation (e.g., \texttt{( AND a b c )}).

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

- **:domain** domain  
  *default value:* value of variable \*DOMAIN*  
  possible values: name of a valid domain, only in \TDLC.

- **:poster** \texttt{flag}  
  *default value:* value of variable \*POSTER* = \texttt{nil}  
  If \texttt{t}, [van Zandt 93]'s poster macros are used to print large feature structures on as many sheets as are needed. This variable only inserts \input poster and \begin{Poster} ... \end{Poster} and forces "math" math mode.

- **:pprint-lists** \texttt{flag}  
  *default value:* value of variable \*PPRINT-LISTS* = \texttt{t}  
  possible values: \{ \texttt{t|nil} \}  
  If \texttt{flag} is \texttt{t}, lists will be printed using the () notation. If \texttt{nil}, the internal FIRST/REST encoding will be used.

- **:title** title  
  *possible values:* \{ \texttt{nil} | \texttt{string} \}  
  prints a title at the bottom of a feature-structure.

- **:index number**  
  *default value:* \texttt{0}  
  This keyword is valid only for \TDLC statements \texttt{llp} and \texttt{lgp}. Its purpose is to select the index of a \TDLC instance.

Example: Modifying the Output Style

The following settings can be used for an output style as it is used in [Carpenter 93].

\begin{verbatim}
<MY-DOMAIN:TYPE> set-switch \*ATOM-COMMAND* "\newcommand{\atom}[1] \{\mbox{[\bf #1]}\}" .
<MY-DOMAIN:TYPE> set-switch \*ATTRIB-COMMAND* "\newcommand{\attrib}[1] \{\mbox{\sc\lowercase{#1}:\ \}}" .
<MY-DOMAIN:TYPE> set-switch \*TYPE-COMMAND* "\newcommand{\type}[1] \{\mbox{\bf #1/\}}" .
\end{verbatim}

Example: Printing Huge Structures

A simple way to get huge feature structures on one page is to use a small font and to hide unimportant attributes e.g.
Example: Printing Structures Hugely

For slides, posters, etc. one may choose big fonts:

```
!set-switch *doc-header* "\documentstyle[a4wide,times]{article}
\textwidth30cm\textheight60cm".

!lgp 'speaker-sem :fontsize "LARGE".
```

Global Variables for \LaTeX\text{\textsc{p}}

Most of the following global variables serve as default for the keywords in the \LaTeX\text{\textsc{p}} print functions. Others are definitions for \LaTeX\text{\textsc{p}} macros for printing attribute names, types, etc. They may be changed for user purposes.

- Global variable *\textsc{filepath}*

  default value: "/tmp/

  possible values: a pathname or pathname string

  specifies the path where .tex, .dvi and other files go.
5.18 Printing Feature Structures

- Global variable *SHELL-COMMAND*  
  default value: "td12latex"  
  possible values: nil or a string containing a shell command name  
  specifies a shell command to run on output file, e.g., 'latex'. If nil, no shell process will be started.

- Global variable *LATEX-HEADER-P*  
  default value: t  
  possible values: t or nil  
  If nil, no \LaTeX{} header (documentstyle..., etc.) will be written to output file.

- Global variable *POSTER*  
  default value: nil  
  possible values: t or nil  
  If t, [van Zandt 93]'s poster macros are used to print feature structures on as many sheets as are needed. This variable only inserts \input poster and \begin{Poster}...\end{Poster} and forces "math" math mode.

- Global variable *PPRINT-LISTS*  
  default value: t  
  possible values: t or nil  
  If t, lists will be printed using the () notation. If nil, the internal FIRST:/REST: encoding will be used.

- Global variable *PRINT-TITLE-P*  
  default value: nil  
  possible values: t or nil  
  If nil, no title (default: type or instance) will be printed.

- Global variable *ALIGN-ATTRIBUTES-P*  
  default value: nil  
  possible values: t or nil  
  If nil, attributes and values will not be aligned.

- Global variable *FONTSIZE*  
  default value: "normalsize"  
  possible values: font size string  
  \LaTeX{} size for feature structures, i.e., one of tiny, scriptsize, footnotesize, normalsize, large, Large, LARGE, huge, Huge.

- Global variable *COREFSIZE*  
  default value: nil  
  possible values: nil or a string  
  if nil, the size for the coreference symbol font will be computed from *FONTSIZE* or the : font size keyword. If it is a string, it must be a valid \LaTeX{} font size, e.g., tiny, scriptsize, footnotesize, normalsize, large, Large, LARGE, huge, Huge.

- Global variable *COREFFONT*  
  default value: "rm"  
  possible values: string  
  \LaTeX{} font name for printing coreferences.

- Global variable *COREFTABLE*  
  default value: ()  
  possible values: assoc list  
  Translation table for coreference numbers and corresponding full names (strings/numbers), e.g.  
  ((1 . "subcat") (2 . "phon") (3 . 1) (4 . 2)).

- Global variable *DOC-HEADER*  
  default value: "  
  possible values: string  
  \LaTeX{} document style header.

- Global variable *MATHMODE*  
  default value: "displaymath"  
  possible values: string  
  \LaTeX{} math mode for feature structures, one of math, displaymath, equation.

- Global variable *TYPESTYLE*  
  default value: :infix  
  possible values: :infix or :prefix  
  style for complex types, infix or Lisp-like prefix.
• Global variable **REMOVE-TOPS**
  default value: nil
  possible values: t or nil
  If t, attributes with empty values and top type will be removed. If nil, top type attributes will not be removed.

• Global variable **TITLE**
  default value: nil
  possible values: nil or string
  If string: title of feature structure. If nil, no title will be printed.

• Global variable **WAIT**
  default value: nil
  possible values: t or nil
  If t, \texttt{LATEX} will wait for shell-command to be terminated, if nil, \texttt{LATEX} will not wait.

• Global variable **LABEL-HIDE-LIST**
  default value: ()
  possible values: list of symbols
  List of attribute symbols to be hidden.

• Global variable **HIDE-TYPES**
  default value: nil
  possible values: t or nil
  If nil, types will be printed, if t, types will be hidden.

• Global variable **ARRAYCOLSEP**
  default value: "0.3ex"
  possible values: string
  Distance (a \texttt{LATEX} length value) between braces resp. brackets and attribute names and their values.

• Global variable **ARRAYSTRETCH**
  default value: 1.1
  possible values: number
  factor for distance between attributes in conjunctions or values in disjunctions.

• Global variable **ATOM-COMMAND**
  default value: "\newcommand{\atom}[1]{\mbox{\tt #1}}"
  possible values: string
  \texttt{LATEX} command for printing atoms.

• Global variable **ATTRIB-COMMAND**
  default value: "\newcommand{\attrib}[1]{\mbox{\tt #1 \ }}"
  possible values: string
  \texttt{LATEX} command for printing attribute names.

• Global variable **TYPE-COMMAND**
  default value: "\newcommand{\type}[1]{\mbox{\it #1 \ }}"
  possible values: string
  \texttt{LATEX} command for printing types.

• Global variable **COREF-COMMAND**
  default value: "\newcommand{\coref}[1]{\setlength{\fboxsep}{0.1ex} % \fbox{\coresize\coreffont #1}}"
  possible values: string
  \texttt{LATEX} command for printing coreferences.

• Global variable **EMPTYNODE-COMMAND**
  default value: "\newcommand{\emptynode}[0]{\mbox{$\{\\}$$}}"
  possible values: string
  \texttt{LATEX} command for printing an empty feature structure [].

• Global variable **TITLE-COMMAND**
  default value: "\newcommand{\featuretitle}[1]{\centerline{\it #1/\}}\smallskip}"
possible values: string
\LaTeX command for printing title.

- Global variable *\texttt{\texttt{LATEX-SIZE-TRANSLATION}}*
  default value:

\[
\left( \begin{array}{ll}
\text{"tiny"} & 0.5 \\
\text{"footnotesize"} & 0.8 \\
\text{"normalsize"} & 1.0 \\
\text{"Large"} & 1.44 \\
\text{"huge"} & 2.074
\end{array} \right) \left( \begin{array}{ll}
\text{"scriptsize"} & 0.7 \\
\text{"small"} & 0.9 \\
\text{"large"} & 1.2 \\
\text{"LARGE"} & 1.728 \\
\text{"Huge"} & 2.488
\end{array} \right)
\]

An assoc list for magnification factors fontsize $\mapsto$ magstep.
6 TDL Grapher

It is possible to display the TDL type hierarchy using the TDL grapher. The TDL grapher has been implemented first in CLIM 1.2 and is now ported to CLIM 2.0 [McKay et al. 92]. Start: either type (load-system "tdl-grapher") instead of (load-system "tdl") at the beginning or with the function

grapher.

from the TDL reader.

An example screen dump of a TDL grapher session is shown in Figure 5. The grapher layout consists of three screen areas: the menu bar with the buttons described below, the grapher display, and an output window with command history.

![Grapher Application](image)

- **Domain**: Select a domain out of a list of domain names.
- **Choose**: Choose a new node (type) out of a list of alphabetically sorted types. This is the same as clicking the second mouse button at a highlighted type in the graph display. The chosen node is called 'current node' in the following lines.
- **Exit**: Exit the Grapher process.
- **Fegramed**: Call FEGRAMED with the global prototype of the current node.
- **Latex**: Call TDL2LaTeX with the global prototype of the current node.

![Figure 5: TDL grapher](image)
• Expand: Call expand-type on the global prototype of the current node.

• Inspect Type: Call the Inspector with the type info structure of the current node.

• Top Node: Redraw the graph with the current node as the new top node. This is the same as pressing the Shift key and clicking the second mouse button.

• Backup: Redraw the graph with one of the super types of the current node as the new top node.

• Reset: Redraw the graph with the top node of the type hierarchy as the new top node.

• Print Graph: Print the graph to a PostScript™ file tdl-graph.ps.

• Recompute: Recompute and redraw the graph, e.g. after definition of new types. This can also be done while processing a grammar file by simply inserting the recompute statement into the file.

• Options: Menu for setting the output style, e.g., horizontal and vertical space between nodes, maximal depth, etc.

• Info: Print information about the current node.

• Other Links: Toggle between showing/hiding dependency arcs in the type hierarchy.

• Toggle Autoredraw: Toggle the autoredraw mode. If this mode is on, the graph will be redrawn automatically when a type is (re)defined. If autoredraw mode is off (which is the default), the user must press the recompute button for an update of the graph. This button toggles the value of the global variable *UPDATE-GRAPHER-OUTPUT-\(\star\)* (see documentation on page 30).

• External: Print information about lbs and glbs of the current node.

• Encoding: Print information about the encoding of the current node.

• Tdl: Set useful TDL switches.
7 Print/Read Syntax for TDC Type Entries

The output style of the print functions for TDC's typed feature structures as described in section 5.18 can be controlled in a way such that different flags are printed, feature are hidden, or types are omitted. In this section, we describe the syntax of type entries (mainly for the ascii printing) and how the output behavior can be changed.

7.1 Print Modes

The type entry printer/reader is configurable and supports different modes ('print modes') for dumping/transforming typed feature structures to files or other modules of a NL system. The easiest way to change the print mode is to use the following functions/macros.

- **Macro with-print-mode mode lisp-body.**
  Temporarily sets print mode to mode and executes lisp-body.

- **Function save-print-mode.**
  Saves print mode to stack *PRINT-VAR-STACK*.

- **Function restore-print-mode.**
  Restores print mode from stack *PRINT-VAR-STACK*.

- **Function set-print-mode [mode].**
  Sets print mode mode. Default for mode is :default.

The default global print mode is :default, it may be changed by the user for debugging purposes etc. Possible modes are :debug, :default, :exhaustive, :fs-nll, :hide-all, :hide-types, :read-in, :td12asl, :x2morf.

Additional user modes can be defined by extending the global variable *PRINT-PROFILE-LIST*.

Examples:

- Save-Print-Mode. ;; saves print mode
- Set-Print-Mode :debug. ;; changes print mode
- <debugging>
- Restore-Print-Mode. ;; restores print mode

- With-Print-Mode :X2MORF ...(print-fs-calls)...
  ;; can be used in the X2MORF grammar dumping function
- With-Print-Mode :FS-NLL ...(print-fs-calls)...
  ;; for FS-to-NLL translations.

The print mode functions/macro change the following four global variables:

- *PRINT-SLOT-LIST*
- *PRINT-CATEGORY-LIST*
- *PRINT-SORTS-AS-ATOMS*, † in the table below (column 4)
- *PRINT-ONLY-NON-DEFAULTS*, * in the table below (column 5)
7.2 Global Variables

The following global variables are defined in package TDL:

- \*PRINT-SLOT-LIST* 
  default value: (:sort-p :expanded) 
  possible values: list of :complete, :delta, :expanded, :restriction, :sort-p 
  used in: ascii printing, TDC2ISP, FEGRAMED 
  type slots to be printed. The :type slot is always printed.

- \*PRINT-CATEGORY-LIST* 
  default value: (:atoms :avms :sorts) 
  possible values: list of :atoms, :avms, :sorts 
  used in: ascii printing, FEGRAMED 
  List of tdl type categories to be printed.

- \*PRINT-ONLY-NON-DEFAULTS* 
  default value: t 
  possible values: t, nil 
  used in: ascii printing, TDC2ISP, FEGRAMED 
  If nil, all slots in \*PRINT-SLOT-LIST* are printed. If not nil, only slots with non-default values 
  that are member of \*PRINT-SLOT-LIST* are printed. The default values are: 
  :complete t, :delta nil, :expanded t, :restriction *TOP*, :sort-p nil 
  If \*PRINT-ONLY-NON-DEFAULTS* is t and these 4 slots have default value and :type value is the 
  top type of the current domain, then no type entry is printed at all. In all other cases, the value of the 
  :type slot will be printed anyway.

- \*HIDE-TYPES* 
  default value: nil 
  possible values: t, nil 
  used in: ascii printing, TDC2ISP, FEGRAMED 
  If not nil, only true UDNNE atoms (like *fail* and *undef*) will be printed. TDC atoms, sorts 
  and types will not be printed.

- \*PRINT-SORTS-AS-ATOMS* 
  default value: nil 
  possible values: t, nil 
  used in: ascii printing, TDC2ISP, FEGRAMED 
  If not nil, sort symbols will be printed the same way atoms are printed. If nil, sort symbols will be 
  printed like type symbols (:type sort). Conjunctions or Disjunctions of Sorts are always printed 
  with (:type (:and/:or ... )).

- \*PRINT-SLOT-LENGTH* 
  default value: 16 
  possible values: nil or a number 
  used in: ascii printing only 
  If a slot is longer than 16 characters, a newline character will be printed (nil = no limit).

- \*PRINT-NEWLINE* 
  default value: nil 
  possible values: t, nil 
  used in: ascii printing only 
  Prints a newline after each type entry (except before empty label lists).
7.3 BNF

The BNF for typed feature structures (input and output for ascii) is:

\[
\begin{align*}
\text{node} & \rightarrow \text{atom} | \\
& \quad \left[ \text{type-info} \left[ \{ (\text{identifier node}) \}^* \right] \right] | \\
& \quad \left[ \text{type-info} \left[ \text{node} \{ ^\text{node} \}^+ \right] \right] | \\
& \quad \ldots
\end{align*}
\]

\[
\text{type-info} \rightarrow \left( \text{:type type-expr} \\
\quad \{ \text{:complete} \{ t | \text{nil} \} \} \\
\quad \{ \text{:delta} \{ \text{nil} | \{ \{ \text{type-expr} \}^+ \} \} \} \\
\quad \{ \text{:expanded} \{ t | \text{nil} \} \} \\
\quad \{ \text{:restriction type-expr} \} \\
\quad \{ \text{:sort-p} \{ t | \text{nil} \} \} \right)
\]

\[
\text{type-expr} \rightarrow \text{identifier} | \\
\quad \left( \text{:and} \{ \text{type-expr} \}^+ \right) | \\
\quad \left( \text{:or} \{ \text{type-expr} \}^+ \right) | \\
\quad \left( \text{:not} \text{type-expr} \right) | \\
\quad \left( \text{:atom} \text{atom} \right)
\]

\[
\text{atom} \rightarrow \text{identifier} | \text{integer} | \text{string}
\]
8 Emacs TDL Mode

TDL mode for Emacs supports comfortable editing facilities for TDL grammar files. It indicates matching parentheses (([]}{<>), as in Emacs LISP or Tex mode), performs indentation of label lists, and, important for grammar development and debugging, establishes a connection to the TDL system and COMMON LISP. Currently, the TDL mode is implemented for ALLEGRO COMMON LISP.

8.1 Installation

The following installation steps let Emacs know about TDL mode.

1. copy the file tdl-mode.el from the TDL system distribution into your Emacs load path.
2. if it doesn’t already exist, create a directory for auto-include files, e.g. ~/autoinclude
3. copy the file header.tdl from the TDL system distribution into this directory. You can modify this file, but the first line should be ;* Mode: TDL -*.
4. add the following lines to your Emacs init file (~/.emacs by default)

   (load "tdl-mode" nil t)
   (push '("\.tdl$" . tdl-mode) auto-mode-alist)
   (load "autoinclude" nil t)
   (push '("\.tdl$" . "header.tdl") auto-include-alist)
   (setq auto-include-directory "~/autoinclude")

After this, the header file will be included when a new file with extension .tdl is created in Emacs and TDL mode will be switched on when a file's first line is ;* Mode: TDL -*.

8.2 Key Bindings

The following key bindings are defined for the TDL mode:

- key TAB is bound to function tdl-indent-command
  indents one line
- key ESC C-\ is bound to function tdl-indent-region
  indents a whole marked region, e.g. one or more type definitions at once, or the whole buffer
- key ESC C-x is bound to function eval-tdl-expression
  evaluates the whole definition where the cursor is in (up to a terminating dot at the end of a line)
- key C-c C-s is bound to function eval-current-tdl-expression
  is currently the same as ESC C-x
- key C-c C-r is bound to function eval-tdl-region
  evaluates the whole marked region, e.g. one or more type definitions at once, or the whole buffer
- key C-c r is bound to function eval-tdl-region-and-go
  evaluates the marked region and switches to the inferior COMMON LISP buffer
- key C-c C-b is bound to function eval-tdl-file
  performs a TDL include of the whole file associated with the current buffer
- key C-c C-e is bound to function goto-end-of-tdl-expression
  moves the cursor to the end of a TDL definition or statement
- key C-c C-a is bound to function goto-begin-of-tdl-expression
  moves the cursor to the beginning of TDL definition or statement

TDL mode can also be switched on 'by hand' with M-x tdl-mode.
9 Top Level Abbreviations (ALLEGRO COMMON LISP Only)

In the ALLEGRO COMMON LISP [Fra 92] version of TDL, some often used commands are also available as top level abbreviations. The top level command :alias prints a list of available abbreviations:

<table>
<thead>
<tr>
<th>Alias</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>composer</td>
<td>start allegro composer</td>
</tr>
<tr>
<td>fegramed</td>
<td>initialize fegramed</td>
</tr>
<tr>
<td>fgi</td>
<td>fegramed global instance</td>
</tr>
<tr>
<td>fgp</td>
<td>fegramed global prototype</td>
</tr>
<tr>
<td>fli</td>
<td>fegramed local instance</td>
</tr>
<tr>
<td>flap</td>
<td>fegramed local prototype</td>
</tr>
<tr>
<td>grapher</td>
<td>start grapher [system]</td>
</tr>
<tr>
<td>lgi</td>
<td>LaTeX global instance</td>
</tr>
<tr>
<td>lgp</td>
<td>LaTeX global prototype</td>
</tr>
<tr>
<td>lli</td>
<td>LaTeX local instance</td>
</tr>
<tr>
<td>llp</td>
<td>LaTeX local prototype</td>
</tr>
<tr>
<td>pgi</td>
<td>print global instance</td>
</tr>
<tr>
<td>pgp</td>
<td>print global prototype</td>
</tr>
<tr>
<td>pli</td>
<td>print local instance</td>
</tr>
<tr>
<td>plp</td>
<td>print local prototype</td>
</tr>
<tr>
<td>recompute</td>
<td>recompute grapher</td>
</tr>
<tr>
<td>tdl</td>
<td>start tdl reader</td>
</tr>
</tbody>
</table>

:composer, :recompute and :fegramed may also be abbreviated by :com, :rec and :feg.

All top level commands take the same parameters as the corresponding TDL-LISP functions described in the sections before. For compatibility reasons, we recommend that top level commands be used in the interactive mode only but not in the TDL grammar files.

Important Note: Parameters of top level commands should not be quoted. Example:

```
<MY-DOMAIN:TYPE> pgp 'agr-en-type :label-hide-list '(GOV OBL).
```

but

```
MY-DOMAIN(49): :PGP agr-en-type :label-hide-list (GOV OBL)
```

:tdl, :composer and :fegramed don't take any parameter.

In addition to these TDL specific commands, the user may define its own abbreviations. Details are described in the ALLEGRO COMMON LISP manual [Fra 92].
10 Sample Session

;;; -* Mode: TDL -*-
;;; Parametrized Type Expansion in TDL. Demo file

```
defdomain "DEMO". ;;; built-in types will be loaded automatically
begin :domain "DEMO".

set-switch *WARN-IF-REDEFINE-TYPE* NIL. ;;; switch off warnings
set-switch *WARN-IF-TYPE-DOES-NOT-EXIST* NIL. ;;; do
set-switch *PRINT-SORTS-AS-ATOMS* T. ;;; for fegramed/pgp
set-switch *VERBOSE-EXPANSION-P* T. ;;; verbose expansion
set-switch *PRINT-SLOT-LIST* (CONS :DELTA *PRINT-SLOT-LIST*). ;;; show :delta
set-switch *LABEL-SORT-LIST* '(FIRST REST LAST INPUT EDGE NEXT ;;; for output
WHOLE FRONT BACK A B C D X Y Z). ;;; only

fegramed. ;;; start Feature Editor
set-switch FEGRAMED:*DEF-FILENAME* "/tmp/".
grapheer. ;;; start Type Grapher
begin :type.

;;; Parametrized Expansion: expand-only for type d
```
;;; Interactively ask for disjunct order
;;;------------------------------------------------------------------

inter := [disj a1 | b2 | c,
          disj2 b2 | d | 42].

defcontrol inter (:ask-disj-preference t).
expand-type 'inter.
fgp 'inter.
wait.

;;;------------------------------------------------------------------

;;; Negation 'a la [Smolka 89]
;;;------------------------------------------------------------------

xn := [a 1, b 2, c 3].
xn := [n ~xn].
expand-type 'xn.
fgp 'xn.
wait.

;;;------------------------------------------------------------------

;;; Nonmonotonicity (single link overwriting)
;;;------------------------------------------------------------------

a := [person_x:INTEGER,
      person_y:INTEGER].
b := a & [person_x 1 | 2 ].
px3 != b & [person_x 3 ].
expand-type 'px3.
fgp 'px3.
wait.

pxs != b & [person_y "string" ].
expand-type 'pxs.
fgp 'pxs.
wait.

;;;------------------------------------------------------------------

;;; Welltypedness Check for an instance at definition time:
;;;------------------------------------------------------------------

set-switch *VERBOSE-EXPANSION-P* NIL.
set-switch *CHECK-WELLTYPEDNESS* T.
;;; now expand-instance will be done automatically!
begin :instance.
  zi := zz & [z c & [c 3],
          x a1 ].
  fgi 'zi.
wait.
end :instance.
set-switch *CHECK-WELLTYPEDNESS-P* NIL.

;;;;-------------------------------------------
;;;; Automata - Basic Configurations
;;;;-------------------------------------------

proto-config := *avm* & [EDGE, NEXT, INPUT].

non-final-config := proto-config & [EDGE #first,
NEXT.INPUT #rest,
INPUT <#first . #rest>].

final-config := proto-config & [INPUT < >,
EDGE *undef*,
NEXT *undef*].

cconfig := non-final-config | final-config.

;;;;-------------------------------------------
;;;; consider the two regular expressions U=(a+b)^c and X=a(b^)+c^-):
;;;;-------------------------------------------

U := non-final-config & [EDGE %covary('a | 'b, 'c),
NEXT %covary( U , V)].

V :=< final-config.

X := non-final-config & [EDGE 'a,
NEXT Y].

Y := non-final-config & [EDGE 'b,
NEXT Y | Z].

Z := config & [EDGE %covary( 'c, *undef*),
NEXT %covary( Z, *undef*)].

;;;;-------------------------------------------
;;;; now we intersect the two automata U and X --> a(b^)+c
;;;;-------------------------------------------

UX := U & X.

test1 := UX & [INPUT '<a,'b,'c>]. ;; accepted
test2 := UX & [INPUT '<a,'b,'b,'c>]. ;; accepted
test3 := UX & [INPUT '<b,'c>]. ;; is inconsistent
test4 := UX & [INPUT <'a,'b,'c,'d>]. ;; is inconsistent

set-switch *VERBOSE-EXPANSION-P* NIL. ;; silent expansion
set-switch *PRINT-SLOT-LIST* (REMOVE :DELTA *print-slot-list*).
    ;; don't print delta list

expand-type 'test1.
fgp 'test1.
wait.
expand-type 'test2.
fgp 'test2.
wait.
expand-type 'test3.
expand-type 'test4.
wait.

;;; Alt-Kaci's version of APPEND

set-switch *VERBOSE-EXPANSION-P* T.
*cons* := *avm* & [FIRST,REST *list*]. ;; redefine *LIST* recursively

append0 := *avm* & [FRONT <>,
    BACK #1 & *list*,
    WHOLE #1].
append1 := *avm* & [FRONT <#first . #rest1>,
    BACK #back & *list*,
    WHOLE <#first . #rest2>,
    PATCH append & [FRONT #rest1,
        BACK #back,
        WHOLE #rest2]].

append := append0 | append1.

r:=append & [FRONT <'a,'b>,
    BACK <'c,'d>]. ;; result will be in WHOLE

expand-type 'r.
wait.
lgp 'r. ;; generate LaTeX code
wait.

set-switch *VERBOSE-EXPANSION-P* NIL.

q:=append & [WHOLE <'a,'b,'c>]. ;; compute possible inputs
expand-type 'q.
fgp 'q.
wait.

;;; Print Recursive Types (SCCs)
message "List of recursive sccs:"
print-recursive-sccs.

;;; -----------------------------------------------
;;; Print Appropriateness table
;;; -----------------------------------------------

message "Computing appropriateness table"
compute-approp :warn-if-not-unique T.
print-approp.
Index

\[
\begin{align*}
\text{!} &= 23 \\
\& &= 14, 16 \\
\' &= 15 \\
\rightarrow &= 24 \\
: &= 14 \\
+= &= 15 \\
; &= 27 \\
\langle &= 20 \\
\langle \text{!} \rangle &= 21 \\
\text{=} &= 22, 23 \\
\{, \} &= 15 \\
\# &= 17 \\
\$ &= 23 \\
\, &= 14, 16 \\
\mid, \| &= 14, 16, 18 \\
\text{#} &= 16, 22 \\
\text{"#} &= 18
\end{align*}
\]

accessing internal information (infons), 38
:accessor, 39
*ACCUMULATE-INSTANCE-DEFINITIONS*, 26, 28
:alias, 60
alias, 5, 42
*ALIGN-ATTRIBUTES-P*, 48, 51
:align-attributes-p, 48
:and, 30
:and, 30
*AND-OPEN-WORLD-REASONING-P*, 14, 17, 28
append, 35
appropriateness, 37
*ARRAYCOLSEP*, 48, 52
:arraycolsep, 48
*ARRAYSTRETCH*, 48, 52
:arraystretch, 48
ASCII printing, 43
*ASK-DISJ-PREFERENCE*, 32
:ask-disj-preference, 33
:assume-consistency, 41
atom, 15
*ATOM-COMMAND*, 52
atomic-symbols, 38
:atoms, 57
*ATTRIB-COMMAND*, 52
attribute restriction, 23
*ATTRIBUTE-PREFERENCE*, 32
:attribute-preference, 33
attributes, 38
author, 38
author; 25, 28
:avms, 57
:avms, 38
backslash, 16
backup, grapher button, 55
begin, 11
*BOTTOM*, 30
:bottom, 10
*BOTTOM-SYMBOL*, 10, 30
*BUILD-INTERMEDIATE-TYPES-P*, 29
built-in, 14
built-in sorts, 15
buttons, 54
*CHECK-UNIFICATION-WELLTYPEDNESS-P*, 37
check-welltypedness, 37
*CHECK-WELLTYPEDNESS-P*, 37
choose, grapher button, 54
class-info, 38
clear-simplify-memo-tables, 40
comment, 38
comments, 27
COMMON LISP, 27
COMMON LISP packages, 9
:complete, 57
compute-approp, 37
*CONS*, 31
*CONS-TYPE-SYMBOL*, 31
constraints
functional, 22
control environment, 27
*COREF-COMMAND*, 52
coreferences, 17
e external, 22
negated, 18
*COREFFEFONT*, 48, 51
:coreffont, 48
*COREFSIZE*, 48, 51
:corefsize, 48
*COREFTABLE*, 48, 51
:coreftable, 48
count-nodes, 40
close-open-world-reasoning-p, 14, 17, 28
date, 38
date; 25
debug, 56
declare environment, 13
:default, 56
*DEFAULT-AUTHOR*, 25–28
*DEFAULT-DOCUMENTATION*, 25–28
defaults, 23

66
INDEX

defcontrol, 5, 25, 27
defdomain, 10, 30
defsystem, 4
delay, 33
deldomain, 10
delete-all-instances, 38
delete-instance, 38
:delete-package-p, 10
delete-type, 38
deleting instances, 38
deleting types, 38
:delta, 57
describe-template, 42
*DIFF-LIST*, 31
*DIFF-LIST-TYPE-SYMBOL*, 31
difference list, 21
disjoint exhaustive partitions, 14
disjunctions
distributed, 19
with coreferences, 20
simple, 18
distributed disjunctions, 19
with coreferences, 20
do-all-infons, 39
do-infon, 39
*DOC-HEADER*, 49, 51
doc: , 25, 27, 28
documentation, 10
*DOMAIN*, 49
domain, 9
define, 10
delete, 10
:domain, 39, 41, 49
domain, 38
domain environment, 13
domain, grapher button, 54
dotted pair list, 20

Emacs, 59
*EMPTYNODE-COMMAND*, 52
encoding, grapher button, 55
do, 11
*END-OF-LIST*, 31
environment, 11
control, 27
declare, 13
instance, 26
lisp, 27
template, 25
type, 14
EPSF, 47
:errorp, 10
EVAL-CONSTRAINTS, 23
eval-current-tdl-expression, 59
eval-tdl-expression, 59
eval-tdl-file, 59
eval-tdl-region, 59
eval-tdl-region-and-go, 59
:exhaustive, 56
exhaustive partitions, 14
exit
TDC, 5, 42
exit, grapher button, 54
:expand, 33
expand, grapher button, 55
expand-all-instances, 32
expand-all-types, 32
:expand-control, 27
expand-control, 38
expand-control:, 25
*EXPAND-FUNCTION*, 32
:expand-function, 33
expand-instance, 32
:expand-only, 33
expand-type, 32, 55
*EXPAND-TYPE-P*, 29
:expanded, 57
expanding types and instances, 32
expansion, 32
export-symbol, 14
:export-symbols, 10
external, grapher button, 55

efragmented, 44
FEGRAMED, 44, 54, 56
Fegramed, grapher button, 54
fgl, 45
fgp, 44
:file-only, 46
filename, 31
:filename, 45, 47
*FILEPATH*, 47, 50
:filepath, 47
FIRST, 20, 21, 31
*FIRST-IN-LIST*, 31
fli, 44
flp, 44
*FONTSIZE*, 48, 51
:fontsize, 48
:fs-nll, 56
:function, 39
function call, 22
functional constraints, 22
:glob-output-file, 41
global prototype, 16
global variables, 28, 31
goto-begin-of-tdl-expression, 59
goto-end-of-tdl-expression, 59
grammar files, 31
grapher, 54
   buttons, 54
grapher, 54
: hashtable, 41
header.tdl, 59
help, 42
: hide-all, 56
hide-attribute, 14
: hide-attributes, 10
* HIDE-TYPES*, 47, 52, 57
: hide-types, 56
: hide-types, 47
hide-value, 14
: hide-values, 10
* IGNORE-BOTTOM-P*, 28
* IGNORE-GLOBAL-CONTROL*, 32
: ignore-global-control, 33
include, 30, 31, 41
incompatible types, 14
indent
   line, 59
   region, 59
: index, 49
: infix, 49
info, grapher button, 55
infs, 38
information sharing, 17
inheritance, 16
   multiple, 17
: init-pos, 44
inspect, grapher button, 55
Inspector, 55
instance
   definition, 26
   delete, 38
   environment, 26
   expand, 32
   reset prototype, 38
: instances, 38
intermediate, 38
key bindings for Emacs TDC mode, 59
keywords
   in definitions, 25
* LABEL-HIDE-LIST*, 47, 52
: label-hide-list, 44, 47
* LABEL-SORT-LIST*, 44, 47
: label-sort-list, 44, 47
LAST, 21, 31
* LAST-IN-LIST*, 31
* LAST-INSTANCE*, 29
* LAST-TYPE*, 29
\LaTeX, printing feature structures with, 46
\LaTeX, grapher button, 54
latex-fs, 46
* LATEX-HEADER-P*, 48, 51
: latex-header-p, 48, 49
* LATEX-SIZE-TRANSLATION*, 53
ldt, 42
leval, 5, 27
lexical rules, 24
lg, 46
Lisp, 27
lisp environment, 27
* LIST*, 31
LIST, 31
list, 20
   difference, 21
dotted pair, 20
empty, 20
   end of, 20
   with open end, 21
   with restrictions, 21
* LIST-IN-LIST*, 31
* LIST-TYPE-SYMBOL*, 31
li, 46
lisp, 46
* LOAD-BUILT-INS-P*, 30
: load-built-ins-p, 10, 30
load-simplify-memo-table, 40, 41
local prototype, 16
logical operators, 16
* MATHMODE*, 49, 51
: mathmode, 49
* MAXDEPTH*, 32
: maxdepth, 33
: memo-output-file, 41
memoization, 40
message, 42
metasymbols in BNF, 5
mixed?, 38
monotonic, 38
   multiple inheritance, 17
: name, 39
name, 38
negated coreferences, 18
negation, 22
nil, 14
nonmonotonicity, 23
* NORMALFORM-OPERATOR-SYMBOL*, 30
INDEX

not, 22
*NULL*, 31
number, 15

operators
  logical, 16
optional keywords, 25
options, grapher button, 55
: or, 30
other links, grapher button, 55
overwrite-paths, 38
overwrite-values, 38
overwriting, 23

parameters, 38
pathname, 31
paths, 16
pg4, 43
pgp, 43
pl1, 43
plp, 43
*POSTER*, 49, 51
: poster, 49
PostScript™, 47, 50, 55
*PPRINT-LISTS*, 49, 51
: print-lists, 49
: prefix, 49
print feature structures
  to text files, 43
  to the screen, 43
  with PGEMBED, 44
  with \LaTeX, 46
print graph, grapher button, 55
print messages, 42
print modes, 56
print statistics, 39
print-all-names, 39
print-all-statistics, 39
print-appropr, 37
*PRINT-CATEGORY-LIST*, 56, 57
print-control, 5, 35
print-domain-statistics, 39
print-expand-statistics, 39
print-global-statistics, 39
*PRINT-NEWLINE*, 57
*PRINT-ONLY-NON-DEFAULTS*, 56, 57
*PRINT-PROFILE-LIST*, 56
print-recursive-sccs, 35
print-simplify-memo-tables, 40
print-simplify-statistics, 39
*PRINT-SLOT-LENGTH*, 57
*PRINT-SLOT-LIST*, 56, 57
*PRINT-SORTS-AS-ATOMS*, 56, 57
print-switch, 5, 31
*PRINT-TITLE-P*, 49, 51
: print-title-p, 49
print-unified-types, 41
*PRINT-VAR-STACK*, 56
prototype
  global, 16
  local, 16
prototype, 38
: read-in, 56
: read-in, 44
: read-in-mode, 44
recompute, 55
recompute, grapher button, 55
*REMOVE-TOPS*, 47, 52
: remove-tops, 43, 47
reset prototypes, 38
reset, grapher button, 55
reset-all-instances, 38
reset-all-protos, 38
reset-all-statistics, 39
reset-domain-statistics, 39
reset-expand-statistics, 39
reset-global-statistics, 39
reset-instance, 38
reset-proto, 38
reset-simplify-statistics, 39
*RESOLVED-PREDICATE*, 32
: resolved-predicate, 33
REST, 20, 21, 31
*REST-IN-LIST*, 31
restore-print-mode, 56
: restriction, 57
restriction, 23
restriction-types, 38
*RETURN-FAIL-IF-NOT-WELL-TYPED-P*, 37
rule inheritance, 24
rules, 24

sample session, 61
save-print-mode, 56
save-simplify-memo-table, 40
semicolon, 27
set-print-mode, 56
set-switch, 5, 31
setting switches, 31
*SHELL-COMMAND*, 47, 51
: shell-command, 47
*SIGNAL-BOTTOM-P*, 28
simple disjunctions, 18
*SIMPLIFY-PS-P*, 29
skeleton, 16
skeleton, 38
*SORT*, 30
References


[Rokicki 93] Tomas Rokicki. *DVIPS: A \TeX{} driver*, 1993. Documentation included in the DVIPS distribution, ftpable from most \TeX{} sites.


[van Zandt 93] Timothy van Zandt. *Documentation for poster.tex/poster.sty, Posters and banners with generic \TeX{}*, May 1993. Documentation included in the poster distribution, ftpable from most \TeX{} sites.
DFKI Publikationen

Die folgenden DFKI Veröffentlichungen sowie die aktuelle Liste von allen bisher erschienenen Publikationen können von der oben angegebenen Adresse oder per anonymem ftp von ftp.dfki.uni-kl.de (131.246.241.100) unter pub/Publications bezogen werden.

Die Berichte werden, wenn nicht anders gekennzeichnet, kostenlos abgegeben.

DFKI Research Reports

RR-93-14
Joachim Niehren, Andreas Podelski, Ralf Treinen: Equational and Membership Constraints for Infinite Trees
33 pages

RR-93-15
Frank Berger, Thomas Fehrle, Kristof Klöckner, Volker Schölles, Markus A. Thies, Wolfgang Wahlster: PLUS - Plan-based User Support Final Project Report 33 pages

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

RR-93-17
Rolf Backofen:
Regular Path Expressions in Feature Logic
37 pages

RR-93-18
Klaus Schild: Terminological Cycles and the Propositional μ-Calculus
32 pages

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

RR-93-22
Manfred Meyer, Jörg Müller:
Weak Looking-Ahead and its Application in Computer-Aided Process Planning
17 pages

DDKI Publications

The following DFKI publications or the list of all published papers so far are obtainable from the above address or via anonymous ftp from ftp.dfki.uni-kl.de (131.246.241.100) under pub/Publications.

The reports are distributed free of charge except if otherwise indicated.

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

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

RR-93-25
Klaus Fischer, Norbert Kuhn: A DAI Approach to Modeling the Transportation Domain
93 pages

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

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

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

RR-93-29
Armin Laux: Representing Belief in Multi-Agent Worlds via Terminological Logics
35 pages

RR-93-30
Stephen P. Spackman, Elizabeth A. Hinkelman:
Corporate Agents
14 pages
RR-93-31
Elizabeth A. Hinkelmann, Stephen P. Spackman: Abductive Speech Act Recognition, Corporate Agents and the COSMA System
34 pages

RR-93-32
28 pages

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

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

RR-93-35
Harold Boley, François Bry, Ulrich Geske (Eds.): Neuere Entwicklungen der deklarativen KI-Programmierung — Proceedings
150 Seiten
Note: This document is available only for a nominal charge of 25 DM (or 15 US$).

RR-93-36
Michael M. Richter, Bernd Bachmann, Ansgar Bernardi, Christoph Klauck, Ralf Legleiner, Gabriele Schmidt: Von !DA bis IMCOD: Expertensysteme im CIM-Umfeld
13 Seiten

RR-93-38
Stephan Baumann: Document Recognition of Printed Scores and Transformation into MIDI
24 pages

RR-93-40
Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi, Werner Nutt, Andrea Schaefer: Queries, Rules and Definitions as Epistemic Statements in Concept Languages
23 pages

RR-93-41
Winfried H. Graf: LAYLAB: A Constraint-Based Layout Manager for Multimedia Presentations
9 pages

RR-93-42
Hubert Comon, Ralf Treinen: The First-Order Theory of Lexicographic Path Orderings is Undecidable
9 pages

RR-93-43
M. Bauer, G. Paul: Logic-based Plan Recognition for Intelligent Help Systems
15 pages

RR-93-44
Martin Buchheit, Manfred A. Jeusfeld, Werner Nutt, Martin Staudt: Subsumption between Queries to Object-Oriented Databases
36 pages

RR-93-45
Rainer Hoch: On Virtual Partitioning of Large Dictionaries for Contextual Post-Processing to Improve Character Recognition
21 pages

RR-93-46
Philipp Hanschke: A Declarative Integration of Terminological, Constraint-based, Data-driven, and Goal-directed Reasoning.
81 pages

RR-93-48
Franz Baader, Martin Buchheit, Bernhard Hollunder: Cardinality Restrictions on Concepts
20 pages

RR-94-01
Elisabeth André, Thomas Rist: Multimedia Presentations: The Support of Passive and Active Viewing
15 pages

RR-94-02
Elisabeth André, Thomas Rist: Von Textgeneratoren zu Intellimedia-Präsentationssystemen
22 Seiten

RR-94-03
Gert Smolka: A Calculus for Higher-Order Concurrent Constraint Programming with Deep Guards
34 pages

RR-94-05
Franz Schmalhofer, J. Stuart Aitken, Lyle E. Bourne jr.: Beyond the Knowledge Level: Descriptions of Rational Behavior for Sharing and Reuse
81 pages

RR-94-06
Dietmar Dengler: An Adaptive Deductive Planning System
17 pages

RR-94-07
Harold Boley: Finite Domains and Exclusions as First-Class Citizens
25 pages

RR-94-08
Otto Kühn, Björn Höfing: Conserving Corporate Knowledge for Crankshaft Design
17 pages
DFKI Documents

D-93-14
264 pages
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-93-15
Robert Laux:
Untersuchung maschineller Lernverfahren und heuristischer Methoden im Hinblick auf deren Kombination zur Unterstützung eines Chart-Parsers
86 Seiten.

D-93-16
Bernd Bachmann, Ansgar Bernardi, Christoph Klauck, Gabriele Schmidt: Design & Kl
74 Seiten

D-93-20
Bernhard Herbig:
Eine homogene Implementierungsebene für einen hybriden Wissensrepräsentationsformalismus
97 Seiten.

D-93-21
Dennis Drollinger:
Intelligentes Backtracking in Inferenzsystemen am Beispiel Terminologischer Logiken
53 Seiten.

D-93-22
Andreas Abecker:
Implementierung graphischer Benutzeroberflächen mit Tcl/Tk und Common Lisp
44 Seiten

D-93-24
Brigitte Krenn, Martin Volk:
DiTo-Datenbank: Datendokumentation zu Funktionsverbürgungen und Relativsätzen
66 Seiten

D-93-25
Hans-Jürgen Bürckert, Werner Nutt (Eds.): Modeling Epistemic Propositions
118 pages
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-93-26
Frank Peters: Unterstützung des Experten bei der Formalisierung von Textwissen
INFOCOM: Eine interaktive Formalisierungskomponente
58 Seiten

D-93-27
Rolf Backofen, Hans-Ulrich Krieger, Stephen P. Spackman, Hans Uszkoreit (Eds.): Report of the EAGLES Workshop on Implemented Formalisms at DFKI, Saarbrücken
110 pages

D-94-01
Josua Boon (Ed.):
DFKI-Publications: The First Four Years 1990 - 1993
93 pages

D-94-02
Markus Steffens: Wissenserhebung und Analyse zum Entwicklungsprozeß eines Druckbehälters aus Faserverbundstoff
90 pages

D-94-03
Franz Schmalhofer: Maschinelles Lernen:
Eine kognitionswissenschaftliche Betrachtung
54 pages

D-94-04
Franz Schmalhofer, Ludger van Elst:
Entwicklung von Expertensystemen: Prototypen, Tiefenmodellierung und kooperative Wissensevolution
22 pages

D-94-06
Ulrich Buhrmann:
Erstellung einer deklarativen Wissensbasis über recyclingrelevante Materialien
117 pages

D-94-07
Claudia Wenzel, Rainer Hoch:
Eine Übersicht über Information Retrieval (IR) und NLP-Verfahren zur Klassifikation von Texten
25 Seiten

D-94-08
Harald Feibel: IGLOO 1.0 - Eine grafikunterstützte Beweisentwicklungsumgebung
58 Seiten

D-94-09
DFKI Wissenschaftlich-Technischer Jahresbericht 1993
145 Seiten

D-94-10
F. Baader, M. Lenzerini, W. Nutt, P. F. Patel-Schneider (Eds.): Working Notes of the 1994 International Workshop on Description Logics
118 pages
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-94-11
F. Baader, M. Buchheit, M. A. Jusfeld, W. Nutt (Eds.):
Working Notes of the KI'94 Workshop: KRDB'94 - Reasoning about Structured Objects: Knowledge Representation Meets Databases
65 Seiten

D-94-12
69 Seiten
RR-94-10
Knut Hinkelmann, Helge Hintze:
Computing Cost Estimates for Proof Strategies
22 pages

RR-94-11
Knut Hinkelmann: A Consequence Finding Approach for Feature Recognition in CAPP
18 pages

RR-94-12
Hubert Comon, Ralf Treinen:
Ordering Constraints on Trees
34 pages

RR-94-13
Jana Koehler: Planning from Second Principles
—A Logic-based Approach
49 pages

RR-94-14
Harold Boley, Ulrich Buhrmann, Christof Kremer:
Towards a Sharable Knowledge Base on Recyclable Plastics
14 pages

RR-94-15
Winfried H. Graf, Stefan Neurohr: Using Graphical Style and Visibility Constraints for a Meaningful Layout in Visual Programming Interfaces
20 pages

RR-94-16
Gert Smolka: A Foundation for Higher-order Concurrent Constraint Programming
26 pages

RR-94-17
Georg Siruth:
Philosophical Logics—A Survey and a Bibliography
58 pages

RR-94-18
Rolf Backofen, Ralf Treinen:
How to Win a Game with Features
18 pages

RR-94-20
Christian Schulte, Gert Smolka, Jörg Würtz:
Encapsulated Search and Constraint Programming in Oz
21 pages

RR-94-31
Otto Kühn, Volker Becker, Georg Lohse, Philipp Neumann:
Integrated Knowledge Utilization and Evolution for the Conservation of Corporate Know-How
17 pages

RR-94-33
Franz Baader, Armin Laux:
Terminological Logics with Modal Operators
29 pages

DFKI Technical Memos

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

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

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

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

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

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

TM-93-03
Harold Boley, Ulrich Buhrmann, Christof Kremer:
Konzeption einer deklarativen Wissensbasis über recyclingrelevante Materialien
11 pages

TM-93-04
Hans-Günther Hein:
Propagation Techniques in WAM-based Architectures — The FIDO-III Approach
105 pages

TM-93-05
Michael Siniek: Indexing PROLOG Procedures into DAGs by Heuristic Classification
64 pages

TM-94-01
Rainer Bleisinger, Klaus-Peter Gores:
Text Skimming as a Part in Paper Document Understanding
14 pages

TM-94-02
Rainer Bleisinger, Berthold Kröll:
Representation of Non-Convex Time Intervals and Propagation of Non-Convex Relations
11 pages