The XDG Grammar Development Kit

Ralph Debusmann[1]

Denys Duchier[2]

Joachim Niehren[3]

[1] Programming Systems Lab, Saarbrücken, Germany

[2] LORIA, Nancy, France

[3] INRIA Futurs, Lille, France

Overview

1. Introduction
2. Extensible Dependency Grammar (XDG)
3. Lexicon specification language
4. XDG Development Kit (XDK)
5. Mozart implementation
6. Conclusions

Declarative grammar formalisms

- long tradition for modeling natural language (Kay 1979), (Bresnan/Kaplan 1982)
- idea: specify linguistic knowledge in grammars independent from processing
- parsers/generators: can be generically created for all grammars in the formalism
- examples:
  - Lexical Functional Grammar (LFG) (Bresnan/Kaplan 1982)
  - Head-driven Phrase Structure Grammar (HPSG) (Pollard/Sag 1994)
  - Tree Adjoining Grammar (TAG) (Joshi et al. 1975), (Joshi 1985)

Grammar development systems

- tools for grammar creation
- concrete syntax for grammar specification
- parsers
- generators
- debugging facilities
- examples:
  - Grammar Writer’s Workbench (Kaplan/Maxwell 1996) for LFG
  - LKB (Copestake 2002) for HSPG
  - XTAG (XTAG Group 2001) for TAG

Constraint programming

- existing grammar formalisms rely on first-order unification of feature structures
- Smolka (Smolka/Uszkoreit 1996): Could more advanced constraint programming techniques improve linguistic processing?
- motivation: languages with freer word order than English (e.g. German, Czech, Hindi etc.) pose problems for existing formalisms

Axiomatization of dependency trees

- (Duchier 1999): axiomatization of valid dependency trees using finite set constraints
- parsing: reduced to finite set constraint programming
- (Duchier/Debusmann 2001): Topological Dependency Grammar (TDG)
- elegant treatment of German word order

Extensible Dependency Grammar (XDG)

- (Debusmann et al. 2004): generalization of TDG
- graph description language flexible for modeling multiple levels of linguistic structure
- same parsing methods by constraint programming (Duchier 2003)
- allows to extend TDG with a concurrent syntax-semantics interface

XDG Grammar Development Kit (XDK)

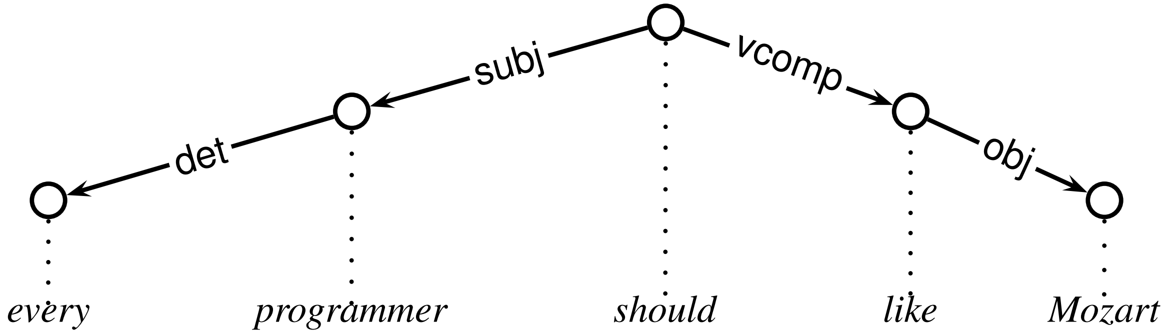
- first grammar development system for XDG
- new lexicon specification language
- implemented in Mozart/Oz, published in MOGUL (Duchier 2004)
- comprehensive suite of facilities:
  - concrete syntaxes (XML, UL, IL)
  - solver for parsing and generation
  - GUI
  - graphical output tools
  - debugging facilities

Overview

1. Introduction
2. Extensible Dependency Grammar (XDG)
3. Lexicon specification language
4. XDG Development Kit (XDK)
5. Mozart implementation
6. Conclusions

Graphs

- XDG describes labeled graphs
- uses the linguistic notion of dependency grammar
- example dependency graph:

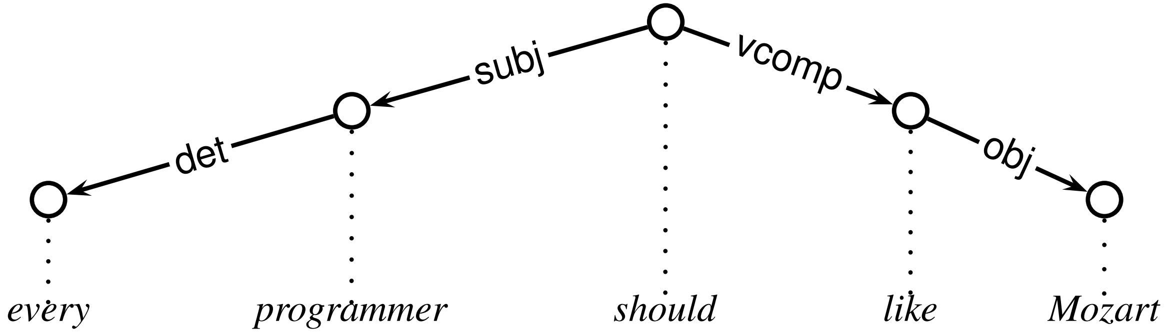


Multiple graphs

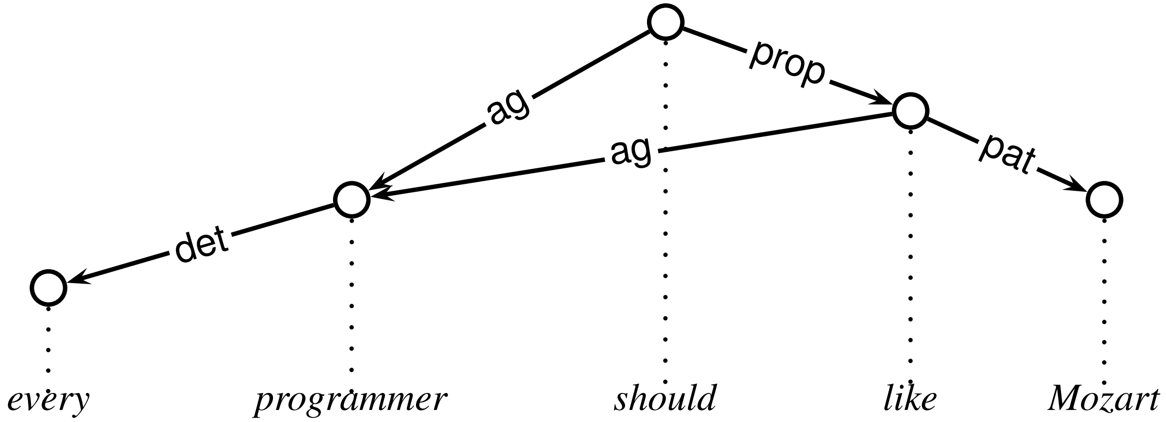
- XDG typically describes an arbitrary number of graphs called dimensions
- same set of nodes, different edges
- elegant treatment of word order (Duchier/Debusmann 2001)
- concurrent syntax-semantics interface (Debusmann et al. 2004)

Example

- Syntax tree:



- Semantic dag:



Graph description language

- well-formedness conditions: interaction of principles and the lexicon
- principles: restrictions on one or more dimensions
- controlled by feature structures, assigned to the nodes by the lexicon
- principles: subset of an extensible principle library
- library covers large fragments of German and English, smaller fragments of Arabic, Czech and Dutch

Example principles

- tree: Dimension i must be a tree
- dag: Dimension i must be a dag
- valency: For each node on dimension i, the incoming edges must be licensed by the in specification, and the outgoing edges by the out specification
- order: Constrains the order of words on dimension i, e.g. subjects precede objects
- linking: Constrains how arguments on dimension i (semantics) must be realized on dimension j (syntax)

Lexical entry

- lexical entry for *like*:

$$\mathit{Like} = \left[\begin{array}{lcl} \mathit{syn} & : & \left[\begin{array}{rcl} \mathsf{in} & : & \{\mathsf{vcomp}?\} \\ \mathsf{out} & : & \{\mathsf{obj}!\} \end{array}\right] \\ \mathit{sem} & : & \left[\begin{array}{rcl} \mathsf{in} & : & \{\mathsf{prop}?\} \\ \mathsf{out} & : & \{\mathsf{ag}!, \mathsf{pat}!\} \\ \mathsf{link} & : & \{\mathsf{ag} \mapsto \{\mathsf{subj}\} \\ & & \mathsf{pat} \mapsto \{\mathsf{obj}\} \end{array}\right] \end{array}\right]$$

Overview

1. Introduction
2. Extensible Dependency Grammar (XDG)
3. Lexicon specification language
4. XDG Development Kit (XDK)
5. Mozart implementation
6. Conclusions

Lexicalization

- XDG: linguistic information mostly specified in the lexicon
- widely accepted in computational linguistics
- lexicon grows huge even for medium-sized grammars
- need facilities for adequate modularization and factorization

Lexical types

- flexible system to define various types of lexical information
- each type: set L and partial function $\sqcap : L \times L \to L$ (combination function of L)
- $\sqcap$: typically greatest lower bound
- domains, records, valencies, sets, tuples, strings

Domain types

- e.g. set of edge labels:

$$\textit{syn.label} \quad = \quad \{\mathsf{det}, \mathsf{subj}, \mathsf{obj}, \mathsf{vcomp}\}$$

- combination function: $a \sqcap a = a$, $a \sqcap b$ undefined for $a \neq b$

Record types

- given set of features $(f_i)_{i=1\dots n}$ and types $T_i = (L_i, \sqcap_i)$:

$$[f_1 : v_1, \dots, f_n : v_n]$$

  where $v_i \in L_i$.

- combination operation defined feature-wise:

$$\begin{array}{l} [f_1 : v_1, \dots, f_n : v_n] \sqcap [f_1 : v'_1, \dots, f_n : v'_n] \ = \\ [f_1 : v_1 \sqcap_1 v'_1, \dots, f_n : v_n \sqcap_n v'_n] \end{array}$$

  when $v_i \sqcap_i v'_i$ defined, undefined otherwise.

Valency types

- e.g. in and out specifications:

$$\textit{syn.valency} \quad = \quad \mathsf{valency}(\textit{syn.label})$$

- defines *syn.valency* to be the record type:

$$[\textit{det} : \textit{mode}, \textit{subj} : \textit{mode}, \textit{obj} : \textit{mode}, \textit{vcomp} : \textit{mode}]$$

- $\textit{mode} = \{0, ?, !, *\}$
- notation:

$$[\textit{det} : 0, \textit{subj} :!, \textit{obj} :?, \textit{vcomp} : 0] \quad = \quad [\mathsf{subj}!, \mathsf{obj}?]$$

- commutative combination operation:

$$0 \sqcap x = x \qquad * \sqcap ! =! \qquad * \sqcap ? =? \qquad ? \sqcap ! =!$$

Meta Grammars

- used for lexicon specification
- CFG-like descriptive device:

$$\begin{array}{rcl} \textit{Clause} & ::= & \textit{Name} \rightarrow \textit{Goal} \\ \textit{Goal} & ::= & \textit{Goal} \wedge \textit{Goal} \mid \textit{Goal} \vee \textit{Goal} \mid \textit{Name} \mid c \end{array}$$

- *Clause* defines a non-terminal *Name*
- *Goal*: non-terminal (lexical class)
- $c \in L$: terminal (feature structure)

Example (1)

- finite verbs can be roots or the root of a relative clause:

$$
\begin{array}{rcl}
\mathsf{finite} & \rightarrow & \mathsf{root} \vee \mathsf{rel} \\
\mathsf{root} & \rightarrow & \left[\begin{array}{lcl} \textsf{\textit{syn}} & : & \left[\begin{array}{lcl} \mathsf{in} & : & \{\} \end{array}\right] \end{array}\right] \\
\mathsf{rel} & \rightarrow & \left[\begin{array}{lcl} \textsf{\textit{syn}} & : & \left[\begin{array}{lcl} \mathsf{in} & : & \{\mathsf{relcl}?\} \end{array}\right] \end{array}\right]
\end{array}
$$

Example (2)

- finite verbs may be either intransitive, transitive or ditransitive:

$$
\begin{array}{rcl}
\mathsf{verb} & \longrightarrow & \mathsf{intr} \vee \mathsf{tr} \vee \mathsf{ditr} \\
\mathsf{intr} & \longrightarrow & \left[\begin{array}{lll} \mathit{syn} & : & \left[\begin{array}{lll} \mathsf{out} & : & \{\mathsf{subj!}\} \end{array}\right] \end{array}\right] \\
\mathsf{tr} & \longrightarrow & \mathsf{intr} \wedge \left[\begin{array}{lll} \mathit{syn} & : & \left[\begin{array}{lll} \mathsf{out} & : & \{\mathsf{obj!}\} \end{array}\right] \end{array}\right] \\
\mathsf{ditr} & \longrightarrow & \mathsf{tr} \wedge \left[\begin{array}{lll} \mathit{syn} & : & \left[\begin{array}{lll} \mathsf{out} & : & \{\mathsf{iobj!}\} \end{array}\right] \end{array}\right]
\end{array}
$$

Example (3)

- finite verb:

$$\mathsf{finite.verb} \quad\longrightarrow\quad \mathsf{finite} \wedge \mathsf{verb}$$

- generative process with start symbol $\mathsf{finite.verb}$:

$$\begin{array}{lll} (\mathsf{root} \wedge \mathsf{intr}) & (\mathsf{root} \wedge \mathsf{tr}) & (\mathsf{root} \wedge \mathsf{ditr}) \\ (\mathsf{rel} \wedge \mathsf{intr}) & (\mathsf{rel} \wedge \mathsf{tr}) & (\mathsf{rel} \wedge \mathsf{ditr}) \end{array}$$

- e.g.:

$$\mathsf{rel} \wedge \mathsf{ditr} \quad\longrightarrow\quad \left[\; \mathit{syn} \;:\; \left[\begin{array}{rcl} \mathsf{in} & : & \{\mathsf{relcl?}\} \\ \mathsf{out} & : & \{\mathsf{subj!}, \mathsf{obj!}, \mathsf{iobj!}\} \end{array} \right] \right]$$

Overview

1. Introduction
2. Extensible Dependency Grammar (XDG)
3. Lexicon specification language
4. XDG Development Kit (XDK)
5. Mozart implementation
6. Conclusions

Concrete syntax

- three concrete syntaxes for different purposes:
  - XML language: automated grammar development
  - User Language (UL): handcrafted grammars
  - Intermediate Language (IL): record-based language, tailored for Mozart/Oz and further processing within the XDK
- UL, XML, IL: conversion into each other

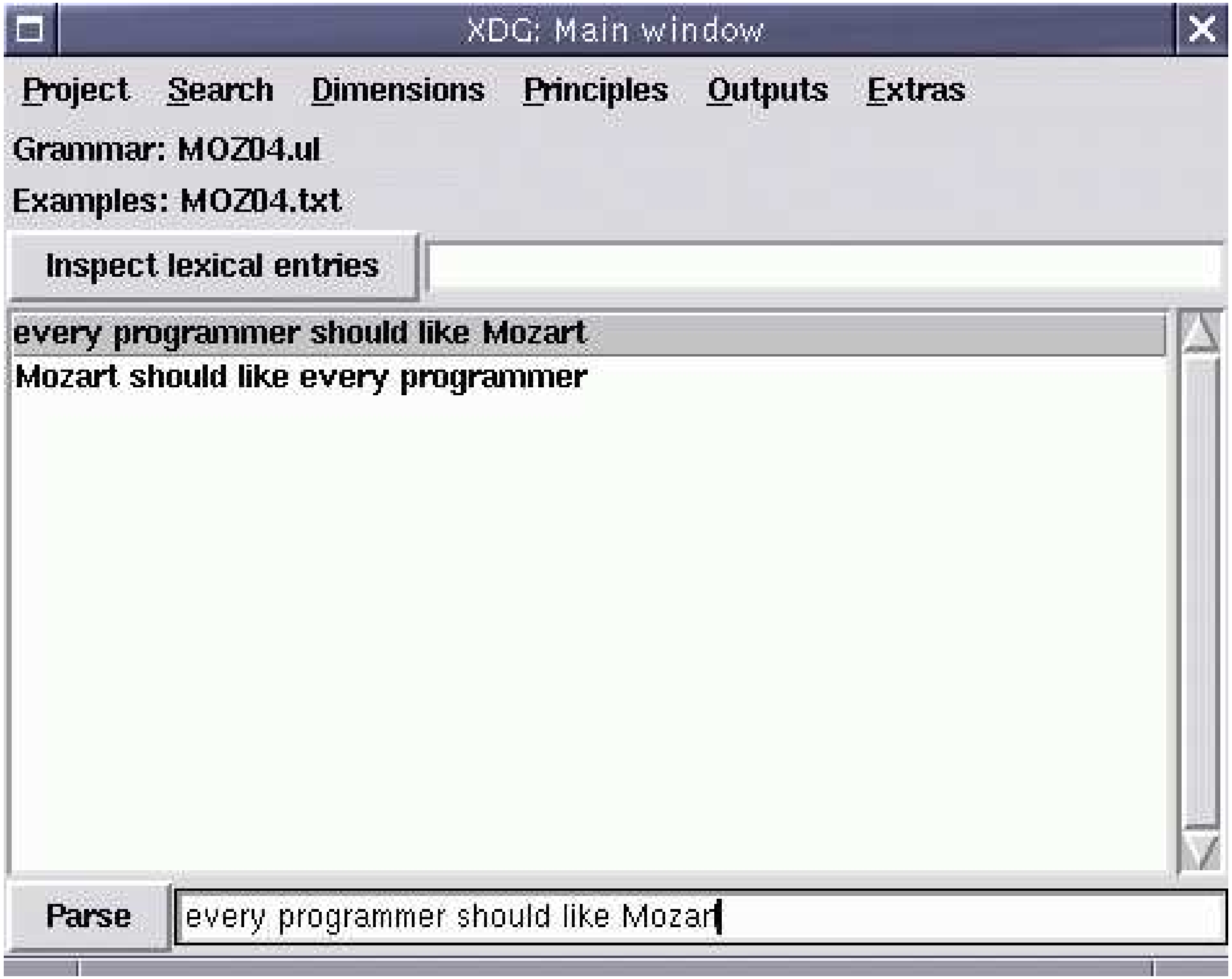
Grammar file compiler

- fast static grammar checker
- fast grammar file compilation
- implemented for IL: i.e. also used for XML language, UL
- compiled grammars stored as pickles (portable) or using Denys Duchier’s GNU GDBM interface (faster)

Graphical interfaces

- comprehensive GUI
- solver search tree visualization: Oz Explorer (Schulte 1997), IOzSeF (Tack 2003)
- visualization of partial/full analyses: output library:
  - Tcl/Tk dag display
  - LaTeX dag output (using Denys Duchier's dtree.sty)
  - internal solver language output using the Oz Inspector (Brunklaus 2000)
- XML output for evaluation, further processing

GUI



Grammar: MOZ04.ul

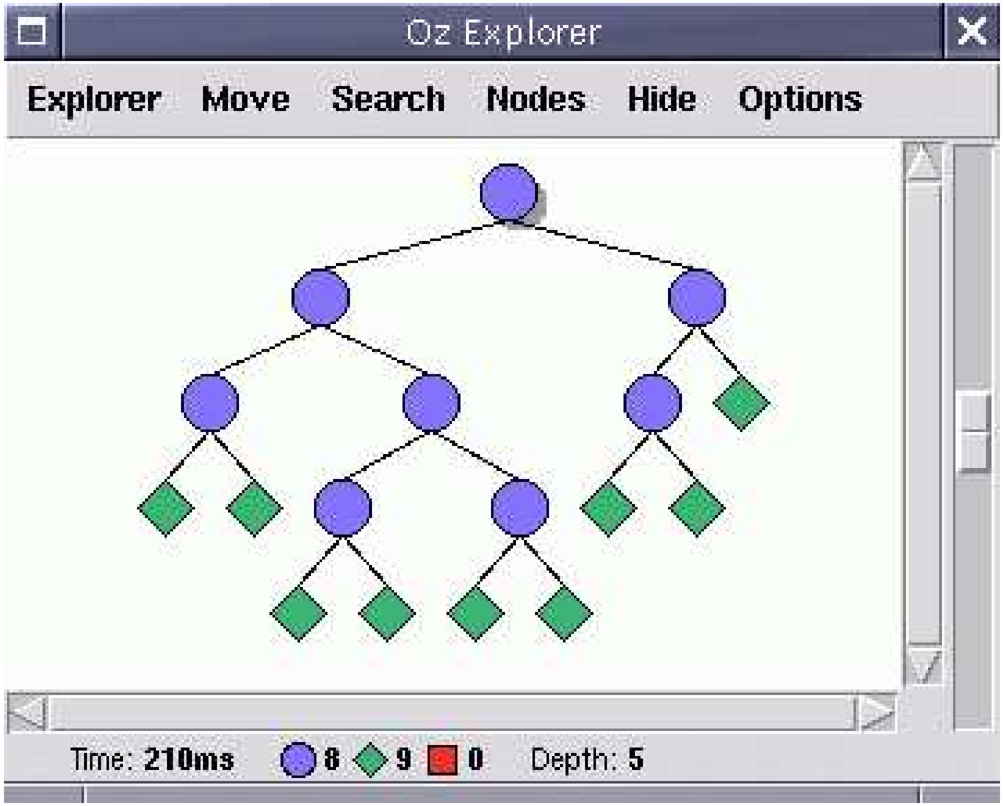
Examples: MOZ04.txt

every programmer should like Mozart

Mozart should like every programmer

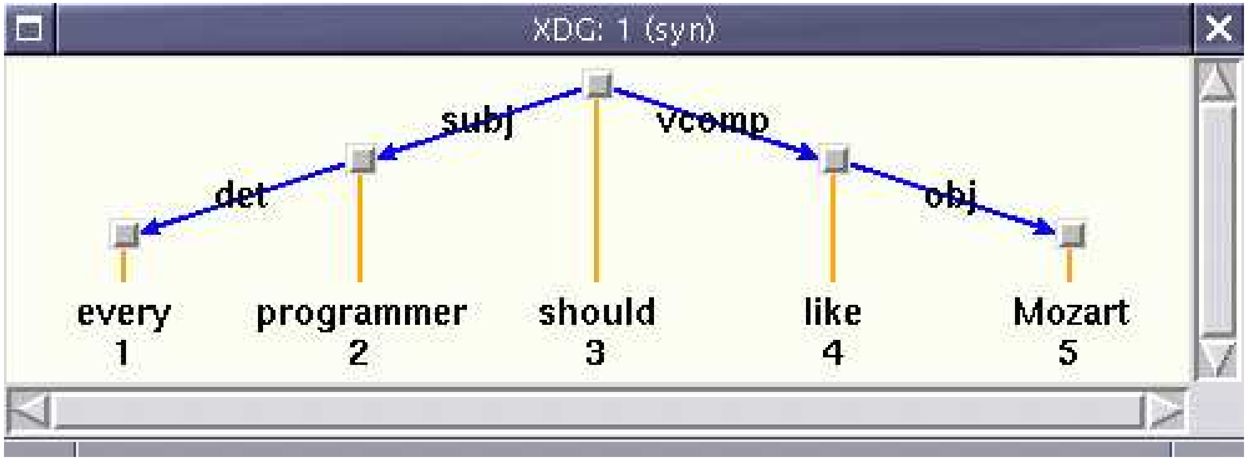
every programmer should like Mozart

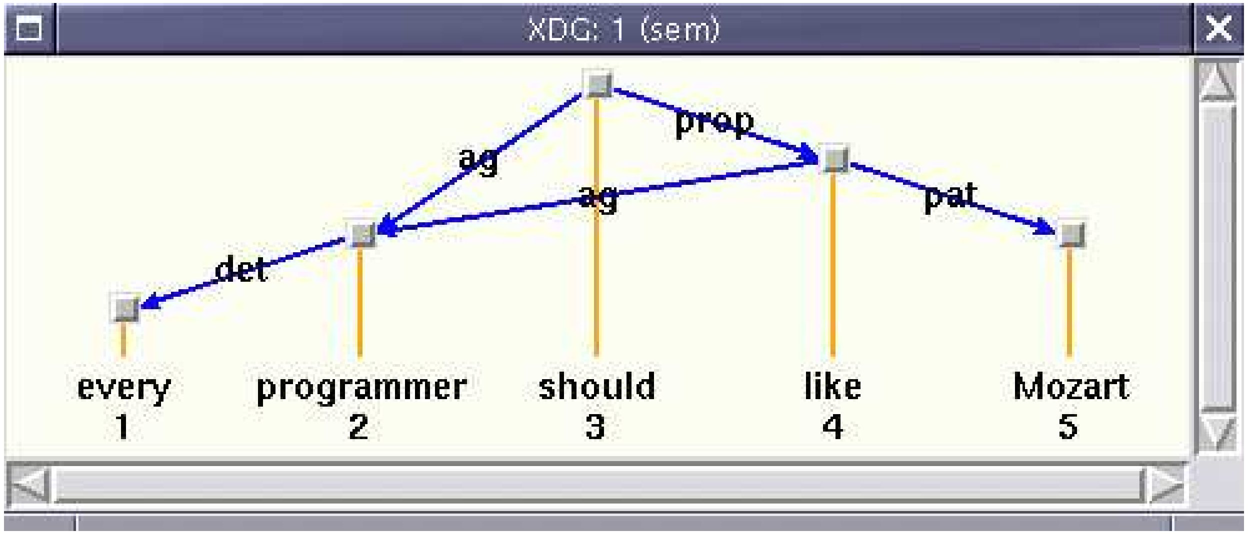
Oz Explorer



Time: 210ms ● 8 ◆ 9 🟥 0 Depth: 5

Dag


XDG: 1 (syn)
subj
vcomp
det
obj
every
1
programmer
2
should
3
like
4
Mozart
5


XDG: 1 (sem)
prop
ag
ag
pat
det
every
1
programmer
2
should
3
like
4
Mozart
5

Solver

- based on axiomatization of dependency parsing in (Duchier 99), (Duchier 2003)
- factorized into modular, extensible principle library
- principles: sets of constraint functors
- e.g. valency principle: in constraint and out constraint
- starting sequence regulated by global constraint priorities to increase efficiency

Preferences and search

- idea: guide the search for solutions by external knowledge sources: oracles
- idea by Thorsten Brants and Denys Duchier, extended in (Dienes et al. 2003)
- oracles interact with solver using sockets
- XDK: supports new standard oracle architecture created by Marco Kuhlmann

Overview

1. Introduction
2. Extensible Dependency Grammar (XDG)
3. Lexicon specification language
4. XDG Development Kit (XDK)
5. Mozart implementation
6. Conclusions

Finite set constraints

- model the graph configuration problem
- e.g. daughters of node w reached by traversing an edge labeled obj represented by set variable $\mathsf{obj}(w)$
- valency specification $\mathsf{obj}?$ corresponds to cardinality constraint $|\mathsf{obj}(w)| \in \{0, 1\}$

Selection constraints

- efficient handling of ambiguity
- typically: word w has multiple lexical entries $L_1, \ldots, L_n$
- variable E_w: ultimately selected entry
- integer variable I_w: index of E_w in the sequence
- selection constraint:

$$E_w = \langle L_1, \ldots, L_n \rangle [I_w]$$

- declarative semantics: $E_w = L_{I_w}$
- can be trivially lifted to record types

Deep guards in disjunctive propagators

- $G_1[]G_2$ enforces complex mutually exclusive well-formedness conditions
- e.g. either a certain edge exists and satisfies additional conditions (G_1) or not (G_2)
- disjunctive propagator for each possible edge

Miscellaneous specialities

- ozmake for convenient compilation and deployment into MOGUL
- principle and output libraries: dynamically linked functors
- two parsers for grammar compilation:
  - flexible LR/LALR parser by Denys Duchier (Gump replacement, fully written in Mozart/Oz)
  - XML parser by Denys Duchier (Mozart Standard Library)
- GUI: Mozart Tcl/Tk interface
- Oz Explorer, IOzSeF, Oz Inspector

Overview

1. Introduction
2. Extensible Dependency Grammar (XDG)
3. Lexicon specification language
4. XDG Development Kit (XDK)
5. Mozart implementation
6. Conclusions

Conclusions

- introduced XDG Development Kit (XDK)
- new lexicon specification language
- large number of development tools implemented in Mozart/Oz
- extensive texinfo documentation (180+ pages)
- no other programming language provides the required expressiveness to combine:
  - set constraints
  - selection constraints
  - deep guards

Future work

- solver: fairly efficient for handcrafted grammars, but not for automatically generated ones
- why? grammar encoding or solver or both?
- theoretical investigation of fragments of XDG
- integration of the new faster GECODE constraint library (Christian Schulte, Gabor Szokoli, Guido Tack)
- super-tagging (lexicon disambiguation before parsing/generation)