Studies on Logic Programming Language for Constraint-based Natural Language Analysis

論理プログラムに基づく制約ベースの自然言語解析

津田　宏

Studies on Logic Programming Language for Constraint-based Natural Language Analysis

by

Hiroshi Tsuda

A Thesis

Submitted to

The Graduate School of
The University of Tokyo
in Partial Fulfillment of the Requirements
for the Degree of
Doctor of Science
in Information Science

1997

Acknowledgments

My special thanks are due to Professors Masami Hagiya, Jun-ichi Tsujii, Toshihisa Takagi, Kazumasa Yokota, and Takao Gunji for reviewing this thesis and making a number of helpful suggestions.

The research on cu-Prolog owes much to the thoughtful and helpful comments of Kôiti Hasida and Hidetosi Sirai about natural language processing and logic programming. I gratefully acknowledge helpful discussions about JPSG with members of the JPSG working group especially Takao Gunji, Yasunari Harada, Ivan Sag, Peter Sells, and Yutaka Tomioka. I wish to thank Hideki Yasukawa, Satoshi Tojo, Mark Johnson, Andre Wlodarczyk, and Kuniaki Mukai for enumerable discussions on natural language processing. I would like to thank Akira Aiba for helpful suggestions about constraint logic programming.

For the research on Quixote, discussions with Kazumasa Yokota about DOOD and knowledge representation languages were always helpful. I would like to thank Ryo Ojima, Yutaka Niibe, Chie Takahashi, Toshihiro Nishioka and other members of the Quixote group.

Most of the research topics in this thesis were performed while the author was a researcher of the Institute for the New Generation Computer Technology (ICOT). Thanks are due to Professor Kazuhiro Fuchi, Shun-ichi Uchida, Koichi Furukawa, and Katsumi Nitta for encouraging me to perform research on natural language processing and logic programming.

I am also indebted to Professor Hisao Yamada and former members of his laboratory at the University of Tokyo: Yosihiko Ono, Kôiti Hasida, Nobuo Satake, Yuka Tateishi, and Mikio Nakano, who had arouse my interest to natural language processing.

I wish to express my thanks to Akinori Yonezawa, Ken Satoh, Hirotaka Hara, Kunio Matsui, Haruo Akimoto, Hiromu Hayashi, and Shigeru Sato for their encouragements. I would like to thank my colleagues at Fujitsu Laboratories Ltd. and ICOT. I thank anonymous referees of my previous papers.

Finally, I wish to express my gratitude to my family.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation – Constraint-based Natural Language Analysis

The purpose of this thesis is to give a logic programming framework for "constraint-based natural language analysis". Natural language analysis is a computational process to convert a surface string into an internal structure in a computer. In the process, it is said that there are linguistically different kinds of analysis such as lexical analysis, parsing, semantic analysis, pragmatic analysis and so on.

Traditional natural language analysis connects processing modules linearly, for example parsing (syntax processing), semantics processing, and pragmatic processing as in Figure 1.1. Such linear systems have difficulty in treating inter-module constraints because the processing rule is fixed in advance. Those modules are not always independent. Sometimes, semantic or pragmatic processing helps reducing the ambiguity in the syntactic processing. In such a case, the linear model can be backtrack-based, or needs a special internal structure to pack the ambiguity in each module.

On the other hand, a processing model of constraint-based natural language analysis is shown in Figure 1.2. In the model, all the linguistic information is stored as a set of constraints. Lexical entries and grammatical rules are also provided as a set of constraints. Constraints are represented as a set of formulas. Each formula defines a range of a variable, or a relation among variables and objects. Those constraints are solved with several interacting processing modules, which are constraint solvers. Constraint representation has two significant characteristics: information partiality and decralativeness. Because of the information partiality, ambiguity, vagueness, and situated-dependent representation of natural language can be described using constraints. Constraints are also declarative, namely they only define static relations and nothing about their processing. Declarative

grammars and lexical entries can be used in various directions, for example, parsing and generation in natural language processing.

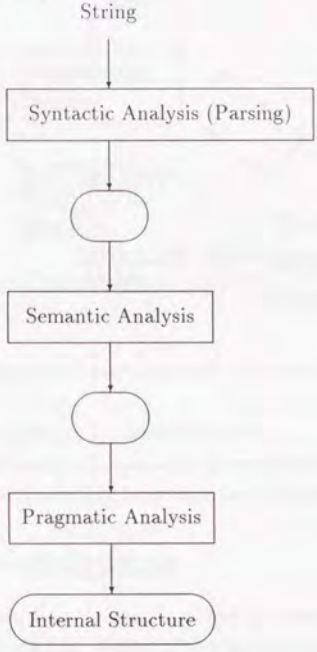


Figure 1.1: Traditional model of natural language analysis

How the constraint-based natural language analysis can be realized? The points are both a constraint-based description framework of linguistic information and a computational framework to solve constraints.

As a framework to describe grammars based on constraints, various *constraint-based grammar* formalisms have been studies since 1980s, such as GPSG (Generalized Phrase Structure Grammar)[GKPS85], LFG (Lexical Functional Grammar)[Bre82], HPSG (Head-driven Phrase Structure Grammar)[PS87b, PS94], JPSG (Japanese Phrase Structure Grammar)[Gun87, GUN96], and so on. Most of them are based on phrase structure grammars whose nodes are *feature structures*. A feature structure is a set of label-value pairs. Most of the grammar description is declaratively given as local con-

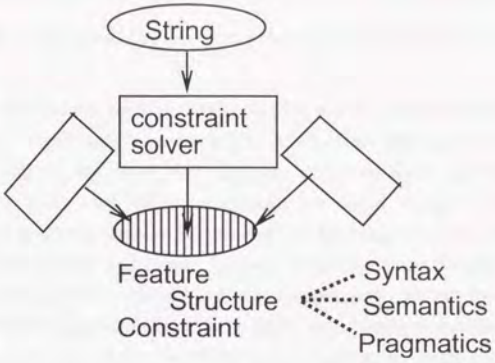


Figure 1.2: Constraint-based model of natural language analysis

straints among feature structures in a phrase structure.

As a computational framework of the declarative constraint-based grammar, this thesis gives two constraint logic programming languages *cu-Prolog* and *Quixote*.

1.2 cu-Prolog and Quixote

Main topics of this thesis are two "constraint-based" programming languages, *cu-Prolog* [THS89, Tsu94] and Quixote[TY94, Yok94]. Their language features and their successful applications to constraint-based natural language analysis are discussed. Here, the term "constraint-based" bears a double meaning.

First, cu-Prolog and Quixote are exemplifications of CLP (Constraint Logic Programming) languages[JL87]. cu-Prolog extends Prolog to treat constraints described in user-defined Prolog predicates. Quixote is designed as a knowledge representation language with CLP and Object-Oriented features. Quixote handles constraints described as subsumption relations (type hierarchy) among objects and their attributes.

Second, these two languages were born as processing frameworks of *constraint-based grammar* formalisms[Shi92]. In computational linguistics, which is a mixed research area between computer science and linguistics, a new grammar description formalism called constraint-based grammar has been studied. Generally, they consist of

- feature structure: data structure to store partial information,

- phrase structure rule: skeleton to associate feature structures, and
- structure principle: local constraint among associated feature structures in a phrase structure.

Their processing framework must be equipped with a data structure to treat partial information, an inference mechanism to construct phrase structures, and a device to represent and process constraints. cu-Prolog and Quixote have data structures that correspond to feature structures. Both have inference mechanisms based on logic programming. Constraints in cu-Prolog are represented as user-defined Prolog predicates and processed using a constraint transformation technique. Quixote constraints are represented as subsumption among objects with modularized (situation-dependent) knowledge description.

Most of the research topics in this thesis have been performed while the author was a researcher of the Institute for the New Generation Computer Technology (ICOT). ICOT is the central research center of both the Japanese Fifth Generation Computer System (FGCS) project and FGCS Follow-on project. One of the purpose of these projects is to develop technologies for knowledge information processing based on logic programming and parallel processing.

While the author was working at ICOT, the author engaged in natural language processing (NLP) and knowledge representation research based on the constraint logic programming paradigm. The author contributed to the following research projects.

1. A CLP language cu-Prolog: the author designed and implemented cu-Prolog and applied it to JPSG parsing,
2. A Constraint-based knowledge representation language Quixote: the author formalized and implemented its constraint solving mechanism and applied Quixote to feature structures and semantic treatment of JPSG,
3. An heterogeneous, distributed, and cooperative problem solving system Helios: the author designed its basic negotiation handler and applied it to constraint-based natural language processing.

This thesis compiles these research contributions with particular emphasis on the first two topics from the view point of processing constraint-based grammar.

1.3 Organization of this Thesis

Chapter 2 illustrates an introduction to constraint-based grammar formalisms. Here, varieties of feature structures and JPSG – a constraint-based grammar formalism especially

for Japanese – are mainly explained.

Chapter 3 describes cu-Prolog. Compared with other CLP languages, cu-Prolog has several unique features. Most CLP languages take algebraic equations or inequations as constraints. cu-Prolog, on the other hand, takes Prolog formulas with user-defined predicates which can implement linguistic constraints. As a constraint solver, cu-Prolog uses the unfold/fold transformation, which is a program transformation technique, dynamically with heuristics. cu-Prolog is equipped with PSTs (Partially Specified Term) as one of its data structure.

Chapter 4 presents applications of cu-Prolog in computational linguistics. It includes disjunctive feature structure (DFS) unification, parsing constraint-based grammar formalisms such as HPSG and JPSG, and CFG parsing based on constraint transformation. In these applications, DFS unification, disambiguation, and parsing are uniformly processed as constraint transformation.

Chapter 5 introduces another constraint-based language Quixote. Quixote is a kind of DOOD (Deductive and Object-Oriented Database)[KNN89, DKM91] language in the sense that it has Object-Oriented features such as object identity and property inheritance, and an inference mechanism. Quixote is also a kind of CLP language with subsumption constraints. Quixote also has a data structure called *attribute term* to treat information partiality.

Chapter 6 illustrates applications of Quixote to constraint-based grammar formalisms. Here, the author takes typed feature structures[TTY+93, TTY+94] and situation-dependent semantic representation in JPSG[TH96]. Subsumption constraints and the module mechanism of Quixote play important role in describing situation-dependent semantic and pragmatic information.

Chapter 2

Constraint-based grammar formalism

In computational linguistics, which is a mixed research area between computer science and linguistics, a new grammar description framework called "constraint-based grammar"[Shi92] has been studies since 1980s, such as GPSG (Generalized Phrase Structure Grammar)[GKPS85], LFG (Lexical Functional Grammar)[Bre82], HPSG (Head-driven Phrase Structure Grammar) [PS87b, PS94], JPSG (Japanese Phrase Structure Grammar)[Gun87, GUN96], and so on. There are several features in the constraint-based grammar framework, compared with traditional grammar description and Chomsky's transformational grammar (TG)[Cho81]. Here, the author introduces an equation to characterize constraint-based grammar formalisms:

$$\begin{aligned} \text{constraint based grammar} \;=\; & \text{feature structure} + \\ & \text{phrase structure} + \\ & \text{structural constraint} \end{aligned}$$

First, the constraint-based grammar formalism is based on computationally tractable data structures and their algorithms. The data structure on which most of the framework are based on is called *feature structure*. At first, feature structure was a set of label-value pairs and only unification is considered as its operation. That is why constraint-based grammar was called "unification-based grammar" in the early 1980s[Shi86]. Further, disjunctive feature structure was provided in FUG[Kay85] to treat ambiguities in natural language. To allow efficient dictionary representation, type hierarchy has been derived as typed feature structure (or sorted feature structure). Typed feature structure is a

central data structure in the latest formalism of HPSG. Apart from their linguistic applications, solvability and efficiently algorithms of such feature structure formalisms have been studied in computational linguistics.

Second, constraint-based grammar formalisms postulate only one internal structure; they analyze natural language sentences as phrase structure whose nodes are feature structures. From processing point of view, phrase structure is easy to treat. Chomsky's series of work on generative grammar, on the other hand, has long been committed to the *transformational grammar* (TG). In TG, he postulated multiple representation levels and several transformation operations among those levels. For example, in GB theory[Cho81], three levels S-structure, D-structure, and logical form (LF) were considered, and the transformation was abstracted to be a very general operation, hence a computationally hard operation, called move-α. In his latest theory *minimalist program*[Cho95], however, he considers only an internal structural level corresponding to LF and studies a computationally economical procedure to map a surface string into an internal tree. Chomsky's approach seems to become similar to the constraint-based grammar.

Third, as its name suggests, constraint-based grammar describes a grammar as a set of constraints. HPSG and JPSG, for example, are based on very few phrase structures and most of the grammatical information is stored as local constraints among nodes in local phrase structures. Constraint-based approach makes grammar formalisms more general and rich, because morphology, syntax, semantics, and pragmatics are uniformly treated as constraints. Also, declarative grammar description, one of the most important features of constraints, allows various flows of information in processing.

Above three features characterize constraint-based grammar formalisms; feature structure as its main data structure, phrase structure as its analysis framework, and constraints as its declarative grammar description. In the rest of the chapter, the author will explain their details based on HPSG and JPSG formalisms especially.

2.1 Feature Structure – data structure

2.1.1 Feature structure

The central data structure to treat information partiality in the constraint-based (unification-based) grammar formalism is *feature structure*. A feature structure is a set of label-value pairs. The labels are called *features*. In constraint-based grammars, morphological, syntactic, semantic, and pragmatic information is uniformly stored in the feature structure.

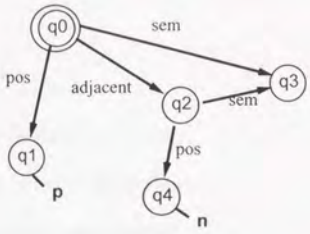


Figure 2.1: Example of a feature graph

First, feature structure can be seen as a rooted, directed, (acyclic,) labeled graph structure called *feature graph*[Shi86]. Each arc is labeled with a feature name, and points to another feature graph or an atomic symbol. Figure 2.1 is an example of a feature graph representing a JPSG-like lexical entry. Let *pos* represent a part of speech, *adjacent* the category that the word follows, and *sem* a semantic representation.

Let A and L be a non-empty set of atomic symbols and feature names. A feature graph is an automaton defined as a tuple $< Q, q_0, \delta, \pi >$ [Kel93], where

- Q is a non-empty, finite set of *states*,
- $q_0 \in Q$ is the start state,
- $\delta : (Q \times L) \rightarrow Q$ is a partial transition function, and
- $\pi : Q \rightarrow A$ is a partial assignment function. If $\pi(q)$ is defined then $\delta(q, l)$ is not defined for any q and l.

The above example of a feature graph is represented over $A = \{p, n\}$ and $L = \{pos, adjacent, sem\}$ with the following function definitions.

- $Q = \{q_0, q_1, q_2, q_3, q_4\}$
- $\delta(q_0, pos) = q_1$, $\delta(q_0, adjacent) = q_2$, $\delta(q_0, sem) = q_3$, $\delta(q_2, pos) = q_4$, $\delta(q_2, sem) = q_3$
- $\pi(q_1) = p$, $\pi(q_4) = n$

Feature structure is formally treated as so-called *feature logics* by Kasper[Kas88, KR90], Rounds, Smolka[Smo88, Smo92], and so on. For example, Smolka's *feature term*[Smo88] has following syntax, where A and L are a non-empty set of atomic symbols and feature names.

- $\top \mid \bot$,
- a for $a \in A$,
- $l : S$ for $l \in L$ and S: feature term,
- $(S \cup T)$ for S, T: feature terms (disjunction),
- $(S \cap T)$ for S, T: feature terms (conjunction),
- $p1 \downarrow p2$ for $p1, p2$: paths (path equivalence)

Here, a path is a finite sequence of labels and the last term denotes the feature terms designated by two paths agree. Using the notation of the feature terms the feature graph in Figure 2.1 is described as follows.

$$pos : p \cup adjacent.pos : n \cup adjacent.sem \downarrow sem$$

Feature terms have been extended to treat negation, path inequality, quantification, and so on.

Generally, a feature structure is represented as an AVM (attribute-value matrix) notation. Above example of a feature graph is represented as follows.

$$\begin{bmatrix} pos : p \\ adjacent : \begin{bmatrix} pos : n \\ sem : Y \end{bmatrix} \\ sem : Y \end{bmatrix} \tag{2.1}$$

Multiple occurrence of Y means the structure is shared.

2.1.2 Disjunctive Feature Structure (DFS)

Natural language includes ambiguities such as polysemic words, homonyms, and so on. A *disjunctive feature structure (DFS)* introduced in FUG[Kay85] is commonly used to handle disjunction in a feature structure. There are several kinds of structures of DFSs.

Value disjunction A value disjunction specifies alternative values for a single feature. (2.2) is an example of a feature structure with value disjunctions. It illustrates that the value of the *pos* feature is n or v, and the value of the *sc* feature is $<>$ (empty set) or $< [pos : p] >$. The DFS is, thus, four-way ambiguous.

$$\begin{bmatrix} pos : \{n, v\} \\ sc : \left\{ \begin{array}{l} \langle\rangle \\ \langle \begin{bmatrix} pos : p \end{bmatrix} \rangle \end{array} \right\} \end{bmatrix} \tag{2.2}$$

General disjunction A general disjunction specifies alternative groups of multiple features. (2.3) is an example of a feature structure with a general disjunction. Feature *sem* is common, the rest being two-way ambiguous.

$$\left[\begin{array}{l} \left\{\begin{array}{l} \left[\begin{array}{l} pos : n \end{array}\right] \\ \left[\begin{array}{l} pos : v \\ vform : vs \\ sc : \left\langle \left[\begin{array}{l} pos : p \end{array}\right] \right\rangle \end{array}\right] \end{array}\right\} \\ sem : love \end{array}\right] \tag{2.3}$$

Disjunction Name A disjunction name was introduced by Dörre [DE90]. It is a special case of general disjunction where more than one value are mutually dependent. For example, consider a German preposition "in" (into). It expresses the static reading when the following noun phrase is dative case, and directional reading when accusative case. The disjunction can be expressed as a general disjunction in (2.4).

$$\left[\begin{array}{l} \left\{\begin{array}{l} \left[\begin{array}{l} syn : \left[\begin{array}{l} arg : \left[\begin{array}{l} case : dat \end{array}\right] \end{array}\right] \\ sem : \left[\begin{array}{l} rel : stat\_in \end{array}\right] \end{array}\right] \\ \left[\begin{array}{l} syn : \left[\begin{array}{l} arg : \left[\begin{array}{l} case : acc \end{array}\right] \end{array}\right] \\ sem : \left[\begin{array}{l} rel : dir\_in \end{array}\right] \end{array}\right] \end{array}\right\} \end{array}\right] \tag{2.4}$$

(2.4) is represented as a feature structure with a disjunction name as (2.5). A name d_1 attached to curly brackets is a label to specify the mutual disjunction. The combinations of the values of *case* and *rel* features are (dat, dir_in) or $(acc, stat\_in)$. (2.5) is, thus, two-way ambiguous.

$$\left[\begin{array}{l} syn : \left[\begin{array}{l} arg : \left[\begin{array}{l} case : d_1 \left\{\begin{array}{l} dat \\ acc \end{array}\right\} \end{array}\right] \end{array}\right] \\ sem : \left[\begin{array}{l} rel : d_1 \left\{\begin{array}{l} dir\_in \\ stat\_in \end{array}\right\} \end{array}\right] \end{array}\right] \tag{2.5}$$

One serious problem in treating DFSs is that the computational complexity of their unification is essentially NP-complete[KR86]. Some practical, efficient algorithms have been studied by [Kas87, ED88]. About DFS unification, see Section 4.2 in detail.

2.1.3 Typed feature structure

In the latest framework of HPSG [PS94], typed feature structure[Car92b] (sorted feature structure[Smo88]) is used as its basic data structure. As shown in Figure 2.2, a typed

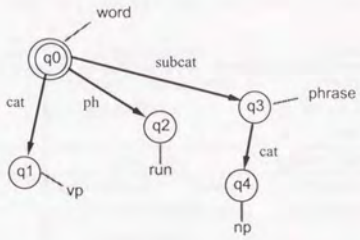


Figure 2.2: Example of a typed feature structure

feature structure is described as a feature graph whose nodes are sorted such as *word*, *vp*, *phrase*, and so on.

A typed feature structure over a set of type symbols $Type$ and a set of labels L is a tuple $< Q, q_0, \delta, \theta >$ [Car92b] where

- Q : a finite set of nodes,
- $q_0 \in Q$: the root node,
- $\delta : (Q \times L) \rightarrow Q$: partial feature value function, and
- $\theta : Q \rightarrow Type$: total node typing function

Figure 2.2 describes a typed feature structure defined as follows.

- $Q = \{q_0, q_1, q_2, q_3, q_4\}$
- $L = \{subcat, cat, ph\}$
- $Type = \{word, vp, run, phrase, np\}$
- $\delta(q_0, cat) = q_1$, $\delta(q_0, ph) = q_2$, $\delta(q_0, subcat) = q_3$, $\delta(q_3, cat) = q_4$
- $\theta(q_0) = word$, $\theta(q_1) = vp$, $\theta(q_2) = run$, $\theta(q_3) = phrase$, $\theta(q_4) = np$

The example can be described in the AVM notation as (2.6).

$$\begin{bmatrix} \textbf{word} \\ \texttt{cat} : [\textbf{vp}] \\ \texttt{ph} : [\textbf{run}] \\ \texttt{subcat} : \begin{bmatrix} \textbf{phrase} \\ \texttt{cat} : [\textbf{np}] \end{bmatrix} \end{bmatrix} \tag{2.6}$$

Table 2.1: Various linguistic constraints

Linguistic category	Case study	Constraint type
syntax	efficient encoding coordination	term equation, term inequations, subsumption
semantics	ellipsis	higher-order term equation
pragmatics	collaborative dialog pronoun reference, translation	temporal constraint statistical constraint

There are several advantages to adopt typed feature structure for describing linguistic information [Kel93]:

- Error detection arising for mis-spellings or misunderstandings of the type system,
- Multiple inheritance allows eliminating much redundancy, and
- Pre-defined types allow for compact representations of feature structures.

2.2 Linguistic Constraints

This section illustrates the variety of the constraint domain and their processing in natural language.

2.2.1 Variety of linguistic constraint

One kind of heterogeneity in NLP is the variety of constraint domains. In [Shi91], Shieber describes several constraints in natural language with various linguistic categories and case studies as Table 2.1. Most of the constraints are represented as sets of formulas containing variables. Constraints are characterized as a set of variable substitutions, called *models*, that satisfy them. In this thesis, (disjunctive) term equation constraints and situation-based subsumption constraints are mainly considered.

2.2.2 Processing linguistic constraint

Processing of constraints is roughly classified as constraint solving and constraint relaxation. Constraint solving is to compute the intersection of models of given constraints. In many cases, including cu-Prolog and Quixote explained in this thesis, constraint solving

is a process to convert a given constraint into a normal form. [1] Constraint relaxation is to relax given over-constrained constraints into satisfiable constraints.

This thesis mainly concerned about constraint solving in two logic programming languages and their applications to natural language applications. Examples of linguistic constraint solving will be investigated in Chapter 4 and Chapter 6. However, constraint relaxation is also important in natural language analysis. The rest of the subsection introduces several linguistic phenomena which are well explained using constraint relaxation. Part of them will be discussed with an extension of cu-Prolog in Section 7.2.

First, sentences (1) are called *garden-path sentences*, because many readers backtrack to understand them[Mar80]. In (1a), "cotton clothing" is read as a noun phrase at first, although the parse fails when one comes to "grows." In these cases, not all the constraints are solved; only partial constraints are solved depending on the processing cost.

(1) a. The cotton clothing is made of grows in Mississippi.
    b. Have all the eggs broken {? or !}

Secondly, sentences (2) are examples of semantic/syntactic interaction[Mar80]. They are all syntactically correct sentences. However, the acceptability differs in terms of syntactic and semantic preferences in constraints. (2a) is recognized as syntactically and semantically good, (2b) semantically bad, and (2c) mildly good.

(2) a. Which dragon did the knight give the boy?
    b. * Which boy did the knight give the dragon?
    c. (?) Which boy did the knight give the sword?

[Mar80] explains those phenomena using preference in syntactic and semantic constraints.

Lastly, sentences (3) are examples of a parse preference (disambiguation). Although they are all syntactically ambiguous, each sentence has a preferred reading. The preference comes from syntactic processing principles, constraint priorities, constraint interaction, and so on. In (3a), the reading "John knows (that) the best man wins." is preferred to "John knows the best (thing that) man wins." The preference comes from a syntactic principle: disfavor of headless structure[HB90]. In the next two Japanese sentences, the first adjective phrase (ADJ) tends to modify the nearest following noun (left association principle) as shown in (3b). In (3c), however, "tsumeeri" (a stand-up collar uniform) and "joseito" (a girl student) are semantically inconsistent. Consequently the reading to modify the last noun ("gakusei") is preferred.

[1] CSP (Constraint Satisfaction Problem) usually searches variable bindings that satisfy given constraints. On the other hand, constraint solving lays emphasis on constraint transformation or rewriting.

(3) a. John knows the best man wins. [HB90]

b. Se no taka-i joseito-wo suki-na gakusei.
tall-ADJ girl students-OBJ like boy student
"a boy student who likes tall girls"

c. Tsumeeri-no joseito-wo suki-na gakusei.
a stand-up collar uniform-ADJ girl students-OBJ like boy student
"a boy student in a stand-up collar uniform who likes girls."

Besides the above phenomena, ill-formed sentences, which are syntactically bad although semantically understandable, can be considered as inter-module constraints between semantic and syntactic processing.

2.3 JPSG (Japanese Phrase Structure Grammar)

JPSG is a constraint-based grammar formalism advocated by Takao Gunji[MGS+86, Gun87, Gun95, GUN96], especially designed to treat Japanese. Until 1992, JPSG had been discussed in the PSG-working group at ICOT.

Constraint-based grammar formalisms are characterized as phrase structures whose nodes are feature structures. Their grammar descriptions consist of a phrase structure and local constraints in a phrase structure called *structural principles*. Current constraint-based grammars such as HPSG and JPSG have a few and general phrase structures, and most of the grammatical information is mainly described with structural principles.

2.3.1 Phrase Structure

JPSG has only one binary phrase structure (Figure 2.3). This phrase structure is appli-

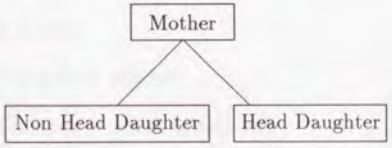


Figure 2.3: Phrase Structure of JPSG

Table 2.2: Non head features of JPSG

feature name	meaning	value range
morph	morphology	a sequence of morphological representations
subcat	subcategorization	a set of local category
adjacent	adjacent category	a local category
slash	nonlocal dependency	a local category
pslash	nonlocal dependency (zero pronoun)	a local category
refl	nonlocal dependency ("zibun")	a local category
sem	a semantic representation	

Table 2.3: Head features of JPSG

feature name	meaning	value range
pos	part of speech	n, v, p, adv, adn
gr	grammatical relation	sbj, obj
case		ga, wo, ni
infl	inflection type	vc, vv,vk,vs,adj,na,nil
vform	verbal form	root, conj, euph, rel, inf, imp
dep	dependent	a core category
mod	modifier (adjunct)	+, -

cable to both a *complementation structure* and an *adjunction structure* of Japanese[2]. In the complementation structure, *Non Head Daughter* works as a complement. It also acts as a modifier in the adjunction structure.

2.3.2 Features

Table 2.2 and Table 2.3 are examples of feature in JPSG[Gun95]. Features in Table 2.2 are called *non-head features* and features in Table 2.3 are called *head features*. Head features are represented as a value of *core* feature. There are other kinds of features as below.

- core features: head features + sem
- local features: core features + subcat, adjacent
- constituency features: local features + slash, pslash, refl

[2]For example, "Ken-ga aisuru (Ken loves)" is a complementation structure, and "ooki-na yama (big mountain)" is an adjunction structure.

2.3.3 Structural Principle

Most of the grammatical information is given as constraints among nodes in a phrase structure. The constraints are called *structural principles*. In [Gun87], head feature principle, subcat feature principle, and foot feature principle are explained as follows.

head feature principle: Head features conform to the *head feature principle*:

> The value of a head feature of the mother unifies with that of the head.

subcat feature principle: Features *sc* (subcategorization) and *adjacent* are called *subcat features*. They take a set of feature structures that specify complement categories and conform to the *subcat feature principle*:

> In the complementation structure, the value of a subcat feature of the mother unifies with that of the head minus left daughter.

foot feature principle: Features *slash* and *pslash* are called *foot features*. They take a list of feature structures and conform to the foot feature principle:

> The value of foot feature of the mother unifies with the union of those of her daughters.

Figure 2.4 is a simple analysis of the Japanese sentence "Ken-ga aruku (Ken walks)." The subcat feature of "aru-ku" takes an singleton set of feature structure. According to the subcat feature principle, the element of *sc* in "aru-ku" unifies with the feature structure corresponds to "Ken ga" to bind variable X into *ken*. Similarly, Y binds to *ken* from the subcat feature principle of *adja* (adjacent) feature in "Ken" and "ga."

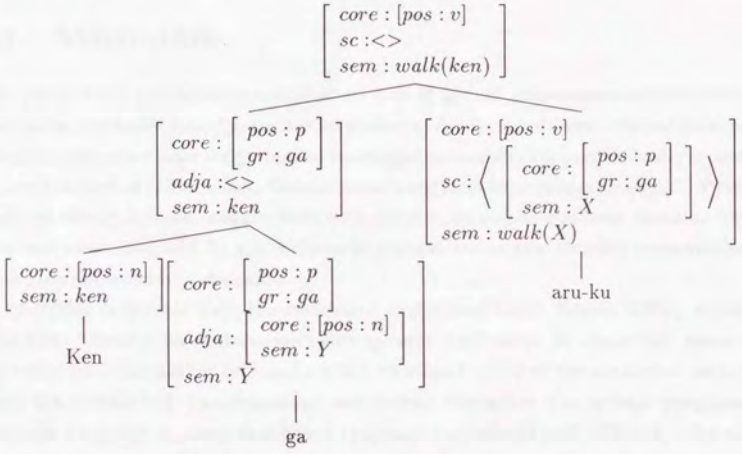


Figure 2.4: JPSG treatment of "Ken-ga aruku."

Chapter 3

cu-Prolog

3.1 Motivation

The research on cu-Prolog has started to aim at giving a fundamental framework for processing constraint-based grammar formalisms. As discussed later, conventional procedural languages and logic programming languages such as Prolog have difficulty in treating varieties of flows of information. Conventional constraint logic programming (CLP) framework, on the other hand, mainly deals with constraints on the algebraic domain. Natural language processing and AI applications in general are largely founded on symbolic and combinatorial constraint domains.

cu-Prolog originates from the *constraint unification*[Has85, Has86, HS86], which is a unification between two patterns with Prolog-term constraints. Its name "cu" comes from the technique. The author reformalizes the constraint solver of the constraint unification using the unfold/fold transformation and embed the solver into a logic programming language to design a constraint-based language cu-Prolog[Tsu89, THS89]. As a data structure to store partial information in constraint-based grammar, cu-Prolog is equipped with PSTs (Partially Specified Term) as its data structure. For procedural processing such as parsing algorithms, cu-Prolog leaves the fixed processing mechanism of Prolog in its part. For symbolic and combinatorial constraints, cu-Prolog can deal with constraints with user-defined Prolog predicates[Tsu94, Tsu92].

Remaining sections of this chapter explain cu-Prolog from a logic-programming viewpoint. cu-Prolog has been implemented on UNIX as cu-PrologIII. Implementation issues are illustrated in Section 3.6. Several natural language applications of cu-Prolog are explained in the next chapter.

3.2 Conventional Approaches

In the beginning of the research on constraint-based grammar (unification-based grammar), Prolog has been often used as the implementation language[Shi86, PS87a], mainly because Prolog is equipped with the unification mechanism between terms. To treat the variety of linguistic constraints mentioned in the previous chapter, however, Prolog itself has the following several defects.

Firstly, Prolog does not have a data structure to represent a feature structure that partially represents information. The arity of complex terms and the order of arguments of lists are both fixed. Even if feature structures are encoded in certain Prolog terms, special procedures for their unification are necessary.

Secondly, its processing order is fixed and procedural. Prolog is often considered as a "declarative" programming language, because declarative predicate definitions instantly become a Prolog program. Seen from its processing mechanism, however, Prolog is not a declarative, but rather a procedural programming language. Atomic formulas of goals are selected from left to right and rules are applied from top to bottom. Prolog programmers intentionally have to align goals such that they are solved properly. In the processing of not only linguistic constraints but constraints in general, it is impossible to stipulate in what order those constraints are processed in advance. When there are both syntactic and semantic constraints, sometimes the semantic constraint is processed at first, but in other situations, the other constraint may be processed earlier.

How about other logic programming languages? Consider Prolog-like systems such as Prolog-II and CIL[Muk88, Muk91]. They extend Prolog with special data structures and new processing mechanisms. The rational tree in Prolog-II removes the restriction of the occur-check from Prolog unification. CIL employs a PST to represent attribute/value pairs. A PST corresponds to a feature structure. In processing, their *bind-hook* mechanism can delay some goals (constraints) until certain variables bind to ground terms. For example, a formula $freeze(X, print(X))$ or $print(X?)$ in CIL delays the evaluation of the goal $print(X)$ until the variable X binds to something. The bind-hook mechanism, however, only checks the satisfiability of the delayed goals only by evaluating them. Even if the same constraint is imposed on more than one place, they are checked independently. So it is not always efficient in the following situations for example:

- When constraints are imposed on more than one variable, the bind-hook description becomes ad-hock.
- Constraints are always solved independently. Even if some of them bind to an

equivalent constraint, they are executed independently.

To extend the declarative feature of Prolog, CLP (Constraint Logic Programming) languages[JL87] have been considered. Most CLP languages, such as CLP(R)[JL87], Prolog-III[Col87], and CAL[ASS+88], take constraints of the algebraic domain with equations or inequations. Their constraint solvers are based on algebraic algorithms such as deriving Gröbner bases, solving algebraic equations, and so on. However, for AI applications and natural language processing systems especially, symbolic constraints are far more desirable than algebraic ones.

There are several implementation frameworks specialized for constraint-based grammar formalisms such as PATR-II[Shi88] and ALE (Attribute Logic Engine)[Car92a]. PATR-II is a grammar representation framework for the earlier constraint-based grammar formalisms (namely, unification-based grammar). As opposed to those frameworks, cu-Prolog is designed as a general computational framework to treat symbolic constraints and a partial data structure. cu-Prolog can handle constraints with user-defined Prolog predicates. ALE is an engine to process the typed feature structure. cu-Prolog cannot handle typed feature structure. However, typed feature structure is discussed in the other language Quixote in Chapters 5 and 6 in this thesis.

Compared with above conventional approaches, advantages of cu-Prolog are summarized as follows.

- cu-Prolog can handle a PST (Partially Specified Term) that has close relation with the feature structure.
- cu-Prolog takes Prolog atomic formulas with user-defined predicates as constraints. Its constraint domain is the Herbrand universe that is suitable for processing symbolic and combinatorial constraints.
- In the CHC (Constrained Horn Clause) of cu-Prolog, both procedural programming and declarative programming are integrated.
- Constraints are processed using the unfold/fold transformation technique[TS83] with certain heuristics. The fold transformation can eliminate redundant constraint processing.

3.3 Syntax

3.3.1 Syntax of Terms

Variables, constants, function names in cu-Prolog are defined as Prolog. cu-Prolog borrows the notation of PSTs (Partially Specified Term) introduced in the CIL language[Muk88].

First, terms of cu-Prolog are defined as follows.

[Def] 1 (Term) *A* term *of cu-Prolog is defined recursively as follows.*

- *A variable is a term.*
- *A constant is a term.*
- *When f is an n-ary function name and $t_1, \cdots, t_n$ are terms, then $f(t_1, \cdots, t_n)$ is a term.*
- *A PST is a term.*

□

[Def] 2 (Partially Specified Term (PST)) *A* PST*(Partially Specified Term) is a term which is a set of label and value pairs having the following form:*

$$\{l_1/t_1, l_2/t_2, \ldots\} .$$

l_i is a constant called a label *where $l_i \neq l_j (i \neq j)$. t_i is a term called a* value. *"/" is the delimiter between a label and value.* □

The order of labels of PSTs is not significant. An infinite PST structure such as $X = \{l/X\}$ is not allowed mainly because of the implementation restriction. A non-recursive PST is semantically equivalent to a complex term. Let $l_1, l_2, \cdots, l_n$ are all the PST labels of a program ordered by a certain total ordering such as a dictionary ordering. A PST $\{p_1/v_1, \cdots, p_m/v_m\}$ is equivalent to a complex term of a certain function name (such as 'pst') whose i-th argument is:

- v_k, if $l_i = p_k$ $(k \in [1, m])$
- _, otherwise.

For example, when (l, m, n) are all the labels, a PST $\{m/a, l/X\}$ is equivalent to a term $pst(X, a, \_)$.

In the rest of this thesis, $Lab(p)$ represents the set of labels of PST p, and the value of the label l of a PST p is written as $p.l$. Variables contained in a (sequence of) term t is denoted as $Var(t)$.

3.3.2 Syntax of CHC and Program

This subsection introduces a basic component of cu-Prolog, called CHC (Constrained Horn Clause).

There are two kinds of predicate names, *constraint predicate names* and *program predicates*. A constraint predicate is a predicate that is used as a constraint of CHC. Constraint predicates are defined by a set of normal Horn clauses, which are especially called *constraint definition clauses*. Constraint definition clauses must be modularly defined as explained later in Subsection 3.5.1.

Predicates except for constraint predicates are called program predicates. Program predicates are defined by a set of CHCs.

Atomic formulas of cu-Prolog is defined as follows.

[Def] 3 (Atomic formula) *An* atomic formula *is a term of the form* $p(t_1, \cdots, t_n)$, *where* p *is a n-ary predicate name and* $t_1, \cdots, t_n$ *are terms.* □

Here comes the definition of *Constrained Horn Clause (CHC).* [1]

[Def] 4 (CHC) *A* constrained Horn clause (CHC) *is either a program clause or a question clause.* □

[Def] 5 (Program clause) *A* program clause *has one of the following forms.*

$$H \quad ; \quad C_1, C_2, \ldots, C_m.$$
$$H \quad :- \quad B_1, B_2, \ldots, B_n; C_1, C_2, \ldots, C_m.$$

H, B_i, *and* C_j *are atomic formulas. The predicate of* H *is a program predicate and each* C_j *consists of a constraint predicate.* H *is called* head, $B_1, B_2, \ldots, B_n$ body, *and* $C_1, C_2, \ldots, C_m$ constraint. *Each* C_i *is called a* constraint formula. □

When constraints are null, above program clauses are equivalent to the following Horn clauses.

$$H.$$
$$H \quad :- \quad B_1, B_2, \ldots, B_n.$$

[1] A CHC was called a *Constraint Added Horn Clause (CAHC)* in the earlier research of cu-Prolog[THS89].

The declarative semantics of the CHC is equivalent to that of Horn clause in the sense that the two kinds of formulas of the program clause are equivalent to the following Horn clauses. A CHC is an extension of a Horn clause where part of its body is specified as a constraint.

$$H \ :- \ C_1, C_2, \ldots, C_m.$$
$$H \ :- \ B_1, B_2, \ldots, B_n, C_1, C_2, \ldots, C_m.$$

[Def] 6 (Question clause) *A* question clause *has one of the following forms.*

$$?- \ G_1, G_2, \ldots, G_n; C_1, C_2, \ldots, C_m.$$

Each G_i is an atomic formula whose predicate is a program predicate and C_i a constraint formula. $G_1, G_2, \ldots, G_n$ is called a goal. *$C_1, C_2, \ldots, C_m$ is called a* goal constraint. □

When constraints are null, above question clause is equivalent to the following question clause of Horn clause.

$$?- \ G_1, G_2, \ldots, G_n.$$

The above question clause of CHC has the same declarative semantics as the following question clause of a Horn clause.

$$?- \ G_1, G_2, \ldots, G_n, C_1, C_2, \ldots, C_m.$$

Clauses listed below are examples of CHCs.

$lexicon([yomu|X], X) : -dictionary(yomu, Y); lexical\_rule(Y, X).$
$psr(H, M, D); head\_feature\_principle(H, M),$
$subcat\_feature\_principle(H, M, D).$

Here, $lexical\_rule/2$, $head\_feature\_principle/2$, and $subcat\_feature\_principle/2$ are user-defined constraint predicates. $lexicon/2$ and $psr/3$ are program predicates.

In the surface syntax, PSTs can appear in the heads and bodies of CHCs. In the internal structure of CHC, however, PSTs always appear in the constraint part. A PST occurring in the head or body is labeled with a new variable and the equivalent equal (unification) constraint of the form $X = PST$ is added to the constraint part. It is called a *constrained PST* defined as follows.

[Def] 7 (Constrained PST) *Let p be a PST, V be a variable, and $t_1, \cdots, t_n$ be related constraints. A* constrained PST *is represented as following constraints in the constraint part of CHC.*

$$V = p,\ t_1, \cdots, t_n$$

where $Var(p) \cap Var(t_i) \neq nil$ *or* $V \in Var(t_i)$ $(i = 1 \cdots n)$. □

For example, a CHC program clause:

$$f(\{pos/v, head/X\}) : -g(X), h(Y); c1(X).$$

is equivalent to

$$f(V) : -g(X), h(Y); V = \{pos/v, head/X\}, c1(X).$$

where $V = \{pos/v, head/X\}, c1(X)$ is a constrained PST. In the following discussion, PSTs in CHCs are supposed to be in the constrained PSTs.

A *program* of cu-Prolog is a tuple (CDC, CHC), where CDC is a set of constraint definition Horn clauses and CHC is a set of CHCs.

Syntax of cu-Prolog is summarized as Table 3.1. Here, $<$ *constant* $>$ is a string of alphabet symbols, numbers, and Kanjis that does not begins with a capital alphabet nor '_'. Especially, the following notations are used.

$[< A >] = < A >$ can be omitted.
$< A >$-list = $< A >$ [',' $< A >$-list] (a sequence of A with a delimiter ',')

3.4 Operational Semantics

This section explains topics in operational semantics of cu-Prolog. Unification between terms including PSTs and derivation rules of cu-Prolog are discussed.

3.4.1 Unification between terms

The unification operation of cu-Prolog is extended to treat PSTs. In Prolog, a unifier between two terms is a variable substitution that makes the terms equal. A PST unifies with a variable or another PST. A unifier between PSTs is given as a PST that is more informative than the two terms. Before going into PST unification, the *subsumption relation* between cu-Prolog terms is introduced.

Table 3.1: Syntax of cu-Prolog in BNF

$$
\begin{array}{rcl}
< capital > & ::= & A|B|C|...|X|Y|Z \\
< term > & ::= & < constant > | < variable > | < function\_term > | < pst > \\
< variable > & ::= & < capital > [< constant >]|\_[< constant >] \\
< function\_term > & ::= & < constant > (< term > -list) \\
< pst > & ::= & < pair > -list \\
< pair > & ::= & < constant > / < term > \\
< aformula > & ::= & < constant > | < constant > (< term > -list) \\
< chc > & ::= & < pclause > | < qclause > \\
< pclause > & ::= & < aformula > [; < aformula > -list].| \\
 & & < aformula >: - < aformula > -list[; < aformula > -list]. \\
< qclause > & ::= & ?- < aformula > -list[; < aformula > -list]. \\
< constraint\_def > & ::= & < aformula > .| \\
 & & < aformula >: - < aformula > -list.
\end{array}
$$

[Def] 8 (Subsumption relation) *A term t subsumes a term u (written as $t \sqsupseteq_p u$) when either of the following condition holds.*

1. *$t \sqsupseteq_p t\theta$ where θ is a variable substitution.*
2. *$p \sqsupseteq_p q$ (p, q: PST) when $\forall l \in Lab(p)$, $\exists l \in Lab(q)$ and $p.l \sqsupseteq_p q.l$.*

□

Here $Lab(p)$ is a set of labels of p, and $p.l$ is the value of label l of p as defined in 3.3.1. Subsumption relation is a partial relation between terms. When $p \sqsupseteq_p q$, q is more informative than p, or p is more general than q.

Example 1 Term subsumption
For example,

$f(X, Y) \sqsupseteq_p f(a, b)$
$\{l/a, m/b\} \sqsupseteq_p \{l/a, m/b, n/c\}$
$\{l/X, m/X\} \sqsupseteq_p \{l/a, m/a\} \sqsupseteq_p \{l/a, m/a, n/c\}$.

□

[Def] 9 (Unification between PSTs) *A* unifier *between two PSTs X and Y is a PST Z which satisfies the following relations.*

$$X \sqsupseteq_p Z,\ Y \sqsupseteq_p Z$$

When Z exists, X and Y are called unifiable. *There exists the most general unifier (mgu) between two unifiable PSTs in terms of $\sqsupseteq_p$.* □

Example 2 PST unification

The most general unifier between $\{l/a, m/b\}$ and $\{l/a, n/c\}$ is $\{l/a, m/b, n/c\}$. The unification between $\{l/a, m/X\}$ and $\{m/b, n/c\}$ yields the mgu $\{l/a, m/b, n/c\}$ and a variable binding $X = b$. □

Algorithm 1 (PST Unification algorithm) *$punify$ in Table 3.2 computes the most general unifier between two PST terms. Let X,Y,Z be pointers to PSTs, and T be the variable substitution of the unification. If unification fails, it returns $fail$.*

The procedure $unify(t_1, t_2, Theta)$ is defined over two Prolog terms (pterm) to compute the most general unifier $Theta$ between two terms t_1 and t_2.

$instance(X, Theta)$ is the instance of X by applying the substitution $Theta$, and $T*T0$ is a composition of two variable substitutions T and $T0$.

The algorithm is a subset of the extended unification of CIL proposed by Mukai[Muk91] because his PST includes a recursive structure. Mukai formalizes PST unification as a constraint satisfaction between two record terms.

3.4.2 Derivation of CHC

The following is the derivation rule of cu-Prolog. The rule derives a new pair of a goal and constraint (resolvent) from an old goal-constraint pair. It is a SLD derivation with the leftmost selection rule as Prolog, followed by constraint solving.

[Def] 10 (Derivation rule) *A* derivation rule *derives a new goal (resolvent) from an initial goal and a program clause.*

Let $? - A, \mathbf{K}; \mathbf{C}$ be an initial goal where A is the left most literal, $A' :- \mathbf{L}; \mathbf{D}$ be a program clause whose head A' unifies with A, and θ be the most general unifier between A and A'. Consequently, a new goal is derived with the following rule.

$$\frac{\overbrace{? - A, \mathbf{K}; \mathbf{C}.}^{goal} \quad \overbrace{A' :- \mathbf{L}; \mathbf{D}.}^{program} \quad \overbrace{\theta = mgu(A, A')}^{substitution} \quad \overbrace{(\mathbf{C}', \sigma) = modular(\mathbf{C}\theta \cup \mathbf{D}\theta)}^{constraint\ transformation}}{\underbrace{? - \mathbf{L}\theta\sigma, \mathbf{K}\theta\sigma; \mathbf{C}'.}_{new\ goal}}$$

Table 3.2: PST Unification algorithm

```
bool punify(X,Y,Z,T)   % (Z,T) is a unifier between X and Y
                       % return true/fail
  pst X, Y, Z; % pointer to PST
  substitution T = nil; % variable substitution
{
  label : l
  substitution: T0
  for l ∈ (Lab(X) ∪ Lab(Y)) {
    if (l ∉ Lab(Y)) then Z.l = X.l
    else if (l ∉ Lab(X)) then Z.l = Y.l
    else {
      if (X.l:pst ∧ Y.l:pst) then {
        if (punify(X.l,Y.l,U,T0) == true) then {
          Z.l = U;
          X = instance(X,T0);
          Y = instance(Y,T0);
          Z = instance(Z,T0);
          T = T * T0;
        }
        else return(fail);
      }
      else if (X.l:pterm ∧ Y.l:pterm) then {
        if (unify(X.l,Y.l,T0) == true) then {
          Z.l = X.l;
          X = instance(X,T0);
          Y = instance(Y,T0);
          Z = instance(Z,T0);
          T = T * T0;
        }
        else return(fail);
      }
      else return(fail);
    }
  }
  return(true);
}
```

modular($Cstr$) *represents an equivalent and simplified form of constraint* $Cstr$ *called* modular. □

Computation of *modular*() is discussed in Subsection 3.5.1. In the above definition, a unifier σ and a set of constraint C' are equivalent to $\mathbf{C}\theta \cup \mathbf{D}\theta$.

Example 3 Derivation
Let

$$? - p(X); member(X, [a, b, c])$$

be an initial goal, and

$$p(Y) : -q(Y); member(Y, [b, c, d]).$$

be a program clause. The head unification between $p(X)$ and $p(Y)$ yields the mgu $\theta = \{X/Y\}$. The constraint becomes

$$member(X, [a, b, c]), member(X, [b, c, d]).$$

$modular(\{member(X, [a, b, c]), member(X, [b, c, d])\})$ returns $c0(X)$ where $c0/1$ is a new predicate and defined as:

$c0(b)$.
$c0(c)$.

□

With the above derivation rule, the refutation in cu-Prolog can be seen as an extension of the SLD refutation[Llo84]. It is a sequence of goal-constraint pairs as Figure 3.1. Let $G_0, G_1, \cdots, G_n$ be goals where $G_n = \phi$, and each C_i be a constraint. (G_{i+1}, C_{i+1}) is derived from (G_i, C_i) with P_i, θ_i, and σ_i. Here, P_i is a program clause whose head unifies with the leftmost atomic formula of G_i. θ_i is the mgu of the head unification. σ_i comes from constraint transformation from constraints of G_i and P_i under a substitution θ_i.

When a goal G_n becomes null (ϕ), the derivation path stops and the result becomes both the products of unifiers

$$\theta_1\theta_2 \cdots \theta_n\sigma_1\sigma_2 \cdots \sigma_n$$

and the remaining constraint C_n.

Note that the heads, goals, and bodies of CHCs are processed procedurally just as Prolog, that is the SLD refutation with the leftmost selection rule and the depth-first

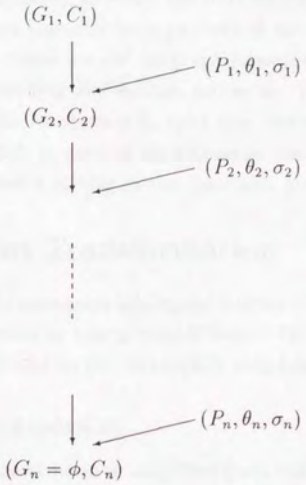


Figure 3.1: Refutation of cu-Prolog

search strategy. The constraint parts of CHCs, however, are solved by constraint transformation with the unfold/fold transformation and the heuristic selection rule as shown in the next subsection.

The soundness of the refutation of cu-Prolog depends on the soundness of SLD refutation and the constraint transformation. The former is guaranteed by Lloyd[Llo84], and the latter is proved by Tamaki and Sato[TS83] in that unfold/fold transformations keep semantics of a given program (in this case, constraints). About the completeness of the refutation, cu-Prolog has the same defect as Prolog because the search strategy is fixed for the depth first strategy[Llo84], although the SLD refutation itself is a complete process.

Most of the problem contains both procedural and declarative computations. For example, in constraint-based natural language processing, the procedural part contains parsing algorithms, consulting dictionaries, and so on. The declarative part, on the other hand, contains linguistic constraints in rules and dictionaries. It is efficient to realize procedural processes such as parsing algorithms in the body, and unspecified processes such as linguistic constraint solving in the constraint part.

3.5 Constraint Transformation

This section explains the constraint solving mechanism of cu-Prolog. The constraint solver transforms a given constraint into a normal form. The normal form is called *modular*. The transformation is based on the unfold/fold transformation.

3.5.1 Modular constraint

A constraint in CHC must be in a simplified form called *modular*. The modularity of constraints is checked syntactically and used in the constraint transformer. Intuitively, constraints are modular when all the arguments are different variables[Tsu89, THS89]. For example,

- `member(X,Y),member(U,V)` is modular,
- `member(X,Y),member(Y,Z)` is not modular because the variable Y occurs in the two distinct places, and
- `append(X,Y,[a,b,c,d])` is not modular because the third argument is not a variable.

Modular constraints are satisfiable if each atomic formula is satisfiable [2]. In the following, We extend the modularity to treat PSTs.

[2]Note that a modular constraint is not the canonical form of constraints

[Def] 11 (Component) *A* component *of an argument of a constraint predicate is a set of PST labels to which the argument can bind. Constants and complex terms are considered as PSTs of nil label.* □

The component of the nth argument of a predicate p is represented as $Cmp(p, n)$.

When there is an atomic formula of the form $X = t$ in a body, it is equivalent to $c0(X, V_1, \ldots, V_n)$ where $\{V_1, \ldots, V_n\} = Var(t)$ and $c0$ is a new $n+1$-ary constraint predicate defined as below.

$c0(t, V_1, \ldots, V_n)$.

The components generalize *vacuous argument places*[HS86, TH90]. In [TH90], the vacuous argument place is defined as follows.

> When an argument place of a predicate is a variable in all of its definition clauses, the argument place is called a *vacuous argument place*. For example, the first argument place of `member` defined below is vacuous.
>
> ```
> member(E,[E|_]).
> member(E,[_|S]) :- member(E,S).
> ```

Vacuous argument places are restated as arguments whose components are ϕ.

Components of a program are computed by static analysis of the program [Tsu91] as follows.

Algorithm 2 (Computing Component) *Components of each constraint predicate are computed from the constraint definition clauses.*

1. *Initially, each $Cmp(p, n)$ is empty for each constraint predicate p of a give program.*
2. *Repeat the following procedure to the constraint definition clauses until there are no changes.*
   (a) *If a definition clause of p has a PST T in the n-th argument of its head, $Cmp(p, n) = Cmp(p, n) \cup Lab(T)$.*
   (b) *If a definition clause of p has a constant or complex term t in the n-th argument of its head, $Cmp(p, n) = Cmp(p, n) \cup \{nil\}$.*
   (c) *If a definition clause of p has a common variable occurring both in the n-th argument of its head and m-th argument of predicate q in its body, $Cmp(p, n) = Cmp(p, n) \cup Cmp(q, m)$.*

□

The process always stops because the length of every component does not exceed the number of PST labels in the program.

Example 4 Components

Consider components of the program :

```
c0({f/b},X,Y):-c1(Y,X).
c0(X,b,_):-X={g/c},c2(X).
c1(X,X).
c1(X,[X|_]).
c2({h/a}).
c2({f/c}).
```

1. Step (a) yields $Cmp(c0,1) = \{f\}$ and $Cmp(c2,1) = \{h, f\}$.
2. After step (b), we get $Cmp(c0,2) = Cmp(c1,2) = \{[\,]\}$.
3. After step (c), we get $Cmp(c0,1) = \{f, g, h\}$.
4. Repeating either rule does not change components. Thus, the computing process stops. The rest of the components is $\{\}$.

Finally, we get following components.

$Cmp(c0,1) = \{f, g, h\}$
$Cmp(c0,2) = Cmp(c1,2) = \{[\,]\}$
$Cmp(c2,1) = \{h, f\}$
$Cmp(c0,3) = Cmp(c1,1) = \{\}$

□

Dependency of more than one constraint formula are defined as follows.

[Def] 12 (Dependency of component) *Two components are dependent when*

- *they have a common element, or*
- *one component is $\{nil\}$ and the other component is not empty.*

For example, $\{a, b, c\}$ and $\{b, c, d\}$ are dependent because b and c are common element. $\{nil\}$ and $\{a, b\}$ are also dependent.

[Def] **13 (Dependency of component)** *A sequence of atomic formulas is* dependent *when at least one of the following conditions holds:*

1. *a variable occurs in two distinct places where their components are dependent, or*
2. *the binding of an argument which has a dependent component.*

□

Example 5 Dependency of constraint
Let $Cmp(f,1) = \{a,b,c\}$, $Cmp(g,1) = \{b,c,d\}$,and $Cmp(h,1) = \{[\,], e\}$.

- $f(X), g(X)$ is dependent because $Cmp(f,1)$ and $Cmp(g,1)$ have a common element and a variable X multiply occurs.
- $f(X), h(X)$ is dependent because $Cmp(f,1)$ and $Cmp(h,1)$ are dependent and a variable X multiply occurs.
- $f(\{a/xx, c/yy\})$ is (internally) dependent because the first argument binds to a PST which has a common element in the component.

□

[Def] **14 (Modular)** *A sequence of atomic formulas is* modular *when it contains no dependency.* □

[Def] **15 (Modularly defined)** *A constraint predicate is* modularly defined *when the bodies of its definition clauses are modular.* □

In cu-Prolog, constraint predicates must be modularly defined. For example, *member*/2, *append*/3, and finite predicates are modularly defined. For natural language processing applications, the description power is sufficient. [3]

3.5.2 Constraint Transformation

The constraint solver of cu-Prolog transforms non-modular constraints into modular ones. The constraint solver is called the *constraint transformer.*

The constraint transformer dynamically utilizes *unfold/fold transformation*[TS83] that preserves the semantics of constraints. One of the idea of the constraint transformer is to introduce heuristic selection rule in unfolding. The heuristic is mainly explained in Section 3.6.

[3]To make cu-Prolog constraint description more powerful, [Sir91] defines *M-solvable* that is weaker than modular. A constraint predicate is *M-solvable* when at least one of the body of its definition has no dependency.

Mechanism of constraint transformation

The unfold/fold transformation[TS83], which is a partial evaluation technique, transforms a program into another preserving its semantics.

Let the original constraints be $\Sigma = C_1, \ldots, C_n$ where any two constraint formulas have (transitively) dependency. Let $\mathcal{T}$ be a set of program Horn clauses, $\{x_1, \ldots, x_m\} = Var(\Sigma)$, and p be a new m-ary predicate. Let $\mathcal{P}_i$ and $\mathcal{D}_i$ be sequences of sets of clauses that are initially defined as:

$$\begin{aligned} \mathcal{D}_0 &= \{\, p(x_1, \ldots, x_m) :\!- C_1, \ldots, C_n. \,\} \\ \mathcal{P}_0 &= \mathcal{T} \cup \mathcal{D}_0. \end{aligned}$$

The *constraint transformer* transforms original constraints Σ into a new constraint $p(x_1, \ldots, x_m)$, if and only if there exists a sequence $\mathcal{P}_0, \ldots, \mathcal{P}_l$ such that every clause in $\mathcal{P}_l$ is modular. $\mathcal{P}_{i+1}$ and $\mathcal{D}_{i+1}$ are derived from $\mathcal{P}_i$ and $\mathcal{D}_i$ by either *unfolding*, *folding*, or *definition* operation $(i = 0 \ldots l)$.

Three operations of unfold/fold transformation is defined as follows.

[Def] 16 (Unfolding) *$\mathcal{P}_{i+1}$ and $\mathcal{D}_{i+1}$ are derived from $\mathcal{P}_i$ and $\mathcal{D}_i$ by* unfolding *as follows.*

$$\frac{\mathcal{P}_i = \{H :\!- A \cdot \mathbf{R}\} \cup \mathcal{P}_i', \quad \{A_j :\!- \mathbf{B}_j\} \subset \mathcal{P}_i, \quad A_j\theta_j = A\theta_j \ (j = 1 \ldots m)}{\mathcal{P}_{i+1} = \mathcal{P}_i' \cup \bigcup_{j=1}^{m} \{H\theta_j :\!- \mathbf{B}_j\theta_j, \mathbf{R}\theta_j\} \quad \mathcal{D}_{i+1} = \mathcal{D}_i}$$

Here, A is a selected atomic formula, A_j are atomic formulas , and $\mathbf{R}$ and $\mathbf{B}_j$ are sequences of atomic formulas. If there are no program clause whose head unifies with H, the unfolding fails. □

[Def] 17 (Folding) *$\mathcal{P}_{i+1}$ and $\mathcal{D}_{i+1}$ are derived from $\mathcal{P}_i$ and $\mathcal{D}_i$ by* folding *as follows.*

$$\frac{\mathcal{P}_i = \{H :\!- \mathbf{C} \cdot \mathbf{R}\} \cup \mathcal{P}_i' \quad \{A :\!- \mathbf{B}\} \subset \mathcal{D}_i, \quad \mathbf{B}\theta = \mathbf{C}}{\mathcal{P}_{i+1} = \mathcal{P}_i' \cup \{H :\!- A\theta, \mathbf{R}\} \quad \mathcal{D}_{i+1} = \mathcal{D}_i}$$

Here, $\mathbf{C}$ and $\mathbf{R}$ are selected such that they have no common variables. □

[Def] 18 (Definition) *Let $\mathbf{B}$ be a sequence of non-modular atomic formulas, $x_1, \ldots, x_n = Var(\mathbf{B})$, and q be a new n-ary predicate. The clause $\{q(x_1, \ldots, x_n) :\!- \mathbf{B}.\}$ is called the* derivation clause *of predicate q/n.*

$$\frac{\mathcal{D}_i, \mathcal{P}_i}{\begin{array}{c}\mathcal{D}_{i+1} = \mathcal{D}_i \cup \{q(x_1, \ldots, x_n) :\!- \mathbf{B}.\} \\ \mathcal{P}_{i+1} = \mathcal{P}_i\end{array}}$$

□

Unfolding selects a target formula (A) and apply (Prolog) derivation for all the program clauses whose heads unify with A. Folding uses a transformation history to eliminate redundant transformation. Definition derives a new predicate name.

The strategy of the constraint solver to apply those three operations is discussed in Section 3.6.

Reduction

New predicates with only one definition clause can be reduced into variable substitutions.

For example, consider $\Sigma =$ `member(X,[ga])`. Σ is transformed into $c0(X)$ where $c0/1$ is defined with the following single definition clause.

$c0(ga)$.

Here, $c0(X)$ can be unfolded to become a term unifier $X = ga$. This operation is called *reduction*. The unifier σ in the [Def] 10(Derivation rule) in 3.4.2 is derived from the reduction operation.

Example of Constraint Transformation

The following example demonstrates a transformation of $\Sigma =$ `member(A,Z),append(X,Y,Z)`.

Firstly, by introducing a new predicate `p1/4` as $D1$, we have:

$$
\begin{aligned}
\mathcal{T} &= \{T1, T2, T3, T4\} \\
T1 &= \texttt{member(X,[X|Y])}. \\
T2 &= \texttt{member(X,[Y|Z])} :\texttt{-}\ \texttt{member(X,Z)}. \\
T3 &= \texttt{append([\,],X,X)}. \\
T4 &= \texttt{append([A|X],Y,[A|Z])} :\texttt{-}\ \texttt{append(X,Y,Z)}. \\
D1 &= \texttt{p1(A,X,Y,Z)} :\texttt{-}\ \texttt{member(A,Z),append(X,Y,Z)}. \\
\mathcal{D}_0 &= \{D1\} \\
\mathcal{P}_0 &= \mathcal{T} \cup \{D1\}.
\end{aligned}
$$

Step 1: By unfolding of the first formula of D1's body (`member(A,Z)`), we get $T5$,$T6$, and $\mathcal{P}_1$.

$$
\begin{aligned}
T5 &= \texttt{p1(A,X,Y,[A|Z])} :\texttt{-}\ \texttt{append(X,Y,[A|Z])}. \\
T6 &= \texttt{p1(A,X,Y,[B|Z])} :\texttt{-}\ \texttt{member(A,Z),append(X,Y,[B|Z])}. \\
\mathcal{P}_1 &= \mathcal{T} \cup \{T5, T6\}
\end{aligned}
$$

Step 2: By defining new predicates `p2/4` and `p3/5` as $D2$ and $D3$, we get the following clauses.

$$
\begin{aligned}
D2 &= \texttt{p2(X,Y,A,Z) :- append(X,Y,[A|Z]).} \\
D3 &= \texttt{p3(A,Z,X,Y,B) :- member(A,Z), append(X,Y,[B|Z]).} \\
T5' &= \texttt{p1(A,X,Y,[A|Z]) :- p2(X,Y,A,Z).} \\
T6' &= \texttt{p1(A,X,Y,[B|Z]) :- p3(A,Z,X,Y,B).} \\
\mathcal{D}_2 &= \{D1, D2, D3\} \\
\mathcal{P}_2 &= \mathcal{T} \cup \{T5', T6', D2, D3\}
\end{aligned}
$$

Step 3: Unfolding $D2$ gives the following clauses.

$$
\begin{aligned}
T7 &= \texttt{p2([ ],[A|Z],A,Z).} \\
T8 &= \texttt{p2([B|X],Y,A,Z) :- append(X,Y,Z).} \\
\mathcal{P}_3 &= \mathcal{T} \cup \{T5', T6', T7, T8, D3\}
\end{aligned}
$$

Step 4: Unfolding the second formula of $D3$'s body (`append(X,Y,[B|Z])`) gives $T9$, $T10$, and $\mathcal{P}_4$.

$$
\begin{aligned}
T9 &= \texttt{p3(A,Z,[ ],[B|Z],B) :- member(A,Z).} \\
T10 &= \texttt{p3(A,Z,[B|X],Y,B) :- member(A,Z), append(X,Y,Z).} \\
\mathcal{P}_4 &= \mathcal{T} \cup \{T5', T6', T7, T8, T9, T10\}.
\end{aligned}
$$

Step 5: Folding $T10$ by $D1$ generates $T10'$ and finally we get the following clauses.

$$
\begin{aligned}
T10' &= \texttt{p3(A,Z,[B|X],Y,B) :- p1(A,X,Y,Z).} \\
\mathcal{P}_5 &= \mathcal{T} \cup \{T5', T6', T7, T8, T9, T10'\}.
\end{aligned}
$$

Every clause in $\mathcal{P}_5$ is modular. As a result, `member(A,Z),append(X,Y,Z)` has been transformed into `p1(A,X,Y,Z)`, preserving equivalence, and new predicates `p1/4`, `p2/4`, and `p3/5` have been defined by $T5'$,$T6'$,$T7$,$T8$,$T9$, and $T10'$.

3.6 Implementation

This section presents several implementation issues about cu-Prolog. cu-PrologIII is an implementation of cu-Prolog.

3.6.1 cu-PrologIII

cu-Prolog has been implemented in the C language on UNIX4.2/3 BSD. The latest version, called *cu-PrologIII*, is registered as ICOT Free Software (IFS). IFS is a byproduct of the Japanese FGCS project and available by anonymous FTP from `http://www.icot.or.jp`.

cu-Prolog is also implemented on MS-DOS and Apple's Macintosh by Hidetosi Sirai[Sir91]. They are also available from the above URL.

Syntax of cu-PrologIII

term: atom, variable, complex term, or PST

atom: constant, string, or number

constant: sequence of characters that begins with a small letter or sequence of any characters with single quotations.

string: sequence of any characters with double quotes.

number: integer and floating number.

variable: sequence of characters that begins with a capital letter or '_'. '_' is called an anonymous variable and any two anonymous variables are different.

complex term: let f be a constant and $t_1, t_2, \cdots, t_n$ be terms, then $f(t_1, t_2, \cdots, t_n)$ be a complex term. f is called a (n-ary)*functor*. A list is a special functor.

atomic formula: let p be a constant and $t_1, t_2, \cdots, t_n$ be terms, then $p(t_1, t_2, \cdots, t_n)$ be an atomic formula. p is called a (n-ary)*predicate symbol*.

PST: sequence of feature/value pairs quoted by '{' and '}'. A feature is a constant and a value is a term.

Constrained Horn Clause (CHC)

A program of cu-PrologIII is a set of Constrained Horn Clauses (CHCs). A CHC has one of the following forms:

1. $H; C_1, \cdots, C_n.$ (Fact)
2. $H :- B_1, \cdots, B_m; C_1, \cdots, C_n.$ (Rule)
3. $:- B_1, \cdots, B_n; C_1, \cdots, C_n.$ (Question)

Each H, B, and C is an atomic formula. H is called a *head*, $B_1, \cdots, B_m$ a *body*, and $C_1, \cdots, C_n$ a *constraint*. A Horn clause is a special case of a CHC whose constraint is null.

cu-PrologIII allows a variable as an atomic formula. By the following programs, `call/1` and `not/1` are defined.

```
call(X) :- X.
not(X)  :- X, !, fail.
not(_).
```

PST (Partially Specified Term)

cu-PrologIII supports PSTs (Partially Specified Term) as a data structure to implement feature structures of constraint-based grammar formalisms. A PST is a term of the following form:

$$\{l_1/t_1, l_2/t_2, \ldots, l_n/t_n\}.$$

l_i, called **label**, is an atom and $l_i \neq l_j (i \neq j)$. t_i, called **value**, is a term. Recursive PSTs are not allowed.

Unification between PSTs X and Y produces Z when:

- $\forall l, l/v \in X, l \notin Y \rightarrow l/v \in Z$
- $\forall l, l/v \in Y, l \notin X \rightarrow l/v \in Z$
- $\forall l, l/v \in X, l/u \in Y \rightarrow l/unify(u, v) \in Z$

For example, the unification between $\{l/a, m/X\}$ and $\{m/b, n/c\}$ produces $\{l/a, m/b, n/c\}$ and X is bound to b.

When a PST occurs in multiple places, it is printed with a new variable in the constraint part of CHC. For example,

```
f(X) :- g1(_p1,X),g2(_p2,X); _p1={f/a,g/c}.
```

Syntax and various features of cu-PrologIII is described in Appendix A of this thesis.

3.6.2 Implementation of Constraint Transformer

Clause Pool

The constraint transformer is implemented with the three clause pools as illustrated in Figure 3.2.

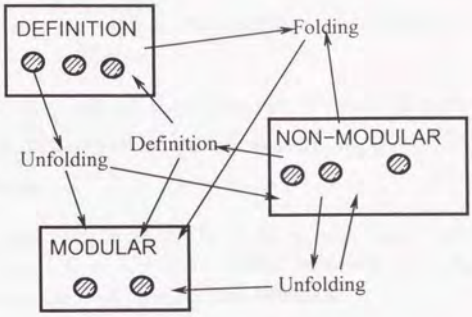


Figure 3.2: Implementation of cu-Prolog constraint solver

- `DEFINITION` stores the derivation clauses of new predicates,
- `NON-MODULAR` stores non-modular constraint definition clauses, and
- `MODULAR` stores modular constraint definition clauses.

`DEFINITION` realizes $\mathcal{D}_i$ and `NON-MODULAR` and `MODULAR` correspond to $\mathcal{P}_i$ in Section 3.

Implementation of Unfold/fold transformation

The unfold/fold transformations presented in Section 3.5.2 are restated using three clause pools as follows.

1. *unfolding*
   Remove one clause $H : -B.$ from `NON-MODULAR` or `DEFINITION`. Select an atomic formula L from B, and the rest formulas are R, namely, $B = L \wedge R$. Let $P_i : -B_i.$ $(i = 1, \ldots, n)$ be all the constraint definition clauses whose heads unify with L $(P_i\theta_i = L\theta_i)$. Add $H\theta : -B_i\theta, R\theta$ $(i = 1, \ldots, n)$ to `NON-MODULAR` or `MODULAR` according to the modularity of their bodies.

2. *folding*
   Remove one clause $H : -B, D.$ from `NON-MODULAR`, where

   - B and D have no dependency and
   - there is a clause $P : -Q$ in `DEFINITION` and $Q\theta = B$.

   Then, add $H : -P\theta, D.$ to `NON-MODULAR`.

3. *definition*

   Remove one clause $H : -B$ from NON-MODULAR, and B is divided into several clusters $B = B_1 \wedge \cdots \wedge B_n \wedge R$, where

   - all $B_i (i = 1, \cdots, n)$ and R are sequences of atomic formulas,
   - B_i and B_j $(i \neq j)$ have no variable dependency, and
   - R is modular.

   Let $X_{i,1}, \cdots, X_{i,m_i} = Var(B_i)$ [4], p_i be a new m_i-ary predicate and $P_i = p_i(X_{i,1}, \cdots, X_{i,m_i})$ $(i = 1, \cdots, n)$. Then, add each $P_i :- B_i$. to DEFINITION $(i = 1, \cdots, n)$ and $H : -P_1, \cdots, P_n, R$ to MODULAR.

Transformation Strategy

Procedure 1 (Constraint Transformation Strategy) *Let C be a non-modular constraint, and $X_1, \cdots, X_n = Var(C)$, p be a new n-ary predicate. The constraint transformer adds*

$$p(X_1, \cdots, X_n) : -SC.$$

to DEFINITION*, and repeat following procedures until* DEFINITION *and* NON-MODULAR *become empties or one of the unfolding fails.*

1. *If* DEFINITION *is not empty, remove one clause from* DEFINITION *and try* **unfolding**.
2. *If* DEFINITION *is empty but* NON-MODULAR *is not empty, remove one clause N from* NON-MODULAR*. If N's head is modular, try* **unfolding**. *If not, attempt* **folding** *or* **definition** *on N's body.*

The head modularity checking of the second procedure reduces the number of new predicates. Consider transformation of $member(X, [a, b, c]), member(X, [b, c, d])$. Without this checking, the constraint is transformed into $c0(X)$ where $c0/1$ and $c1/1$ are defined as:

$c0(b).$
$c0(X) : -c1(X).$
$c1(c).$
$c1(X) : -c2(X).$

[4] A set of variables included in B_i.

However, with this checking, it is transformed into $c0(X)$ *with the following definition of* $c0/1$.

$c0(b)$.
$c0(c)$.

When the procedure successfully end, the clauses in MODULAR *are newly defined constraint definition clauses and* C *is transformed into* $p(X_1, \cdots, X_n)$ *which has same semantics with* C. *When the procedure fails during the unfolding transformation,* C *cannot be transformed into a modular form.* □

By fixing the transformation strategy as above, some constraints cannot be transformed into modular ones, although such a situation is rare for actual linguistic constraints[Tom92]. To avoid the situation, there are several kinds of choices:

- to set the maximum number of unfolding (cu-PrologIII),
- to adjust heuristics (cu-PrologIII),
- to confine user predicates in finite or linear[Tom92] predicates, or
- to relax the definition of modularly-defined such as M-solvable [Sir91].

Heuristics

The constraint transformer includes heuristics such as

- how to select a clause from DEFINITION,
- how to select a clause from NON-MODULAR, and
- how to select a literal in unfolding.

As DEFINITION and NON-MODULAR are implemented as stacks, that is, cu-PrologIII always selects the latest clause.

An unfolding literal can be selected arbitrarily. The constraint transformer computes the *activation value* ε of each atomic formula from following factors, and apply unfolding to the atomic formula of the highest value.

$Const$ = Number of arguments that bind to constants
$Funct$ = Number of arguments that bind to complex terms
$Vnum$ = Total number of dependent variables in the formula

Rec = 1 for recursive predicate and 0 for finite predicate

$Defs$ = Number of definition clauses of the predicate

$Units$ = Number of unit clauses in the predicate definition

$Facts$ = If the predicate is defined only by unit clauses then 1, otherwise 0

From above factors, an activation value is computed as:

$$\text{activation\_value}\ \varepsilon = 3 * Const + 2 * Funct + Vnum - 2 * Rec - Defs + Units + 3 * Facts$$

The formula is defined so as to include some empirical heuristics used in [THS90]. Examples of the heuristics is illustrated as follows.

- For $p(X, Y), q(a, X, b)$, select the second formula from $Const$ factor.
- $m(X, Y), p(X)$ where m and p are defined as

  $m(X, [\,]).$
  $m([A|B], X) : -m(B, X).$
  $p([\,]).$
  $p([a]).$

  Select the second formula from Rec factor. Otherwise, the transformation requires more new predicates.

- $fuse(X, Y, Z), f(Z, T, U), f(U, V, W)$ where $fuse$ is defined as

  $fuse([\,], [\,], [\,]).$
  $fuse([A|X], Y, [A|Z]) : -fuse(X, Y, Z).$
  $fuse(X, [A|Y], [A|Z]) : -fuse(X, Y, Z).$
  $fuse([A|X], [A|Y], Z) : -fuse(X, Y, Z).$

  From $Vnum$ factor, select the second formula that has two dependencies in terms of Z and U. Otherwise, the transformer goes into an infinite loop.

- $ab(X), bcd(X)$ where ab and abc are defined as

  $ab(a).$
  $ab(b).$
  $bcd(b).$
  $bcd(c).$
  $bcd(d).$

Select the first formula from $Defs$ factor. It is because the number of the new definition clauses become less.

- $p(X,Y), q(X,Y)$ where $p/2$ is defined with 20 unit clauses and $q/2$ 3 unit clauses: select the latter from $Units$ factor. It is because the case becomes a "generate and test" computation. Smaller generated candidates are better.

Discussions

Above heuristics in selection rules are well discussed in the research on loop-checking and termination in Prolog[Llo84, Kle84, MS89, Bes89, ABK89], static analysis of Prolog programs[Bru82, Deb89].

Another extension of Prolog is to improve the SLD resolution. OLDT (OLD resolution with Tabulation)[TS86] extends Prolog using histories of previously derived goals. OLDT can solve a program that cannot be solved finitely in Prolog even if heuristic selection rules are used, such as:

$reach(X,Y) : -reach(X,Z), edge(Z,Y).$
$reach(X,X).$
$edge(a,b).$
$edge(a,c).$
$edge(b,a).$
$edge(b,d).$
$? - reach(a,X).$

In cu-Prolog, $reach(a,X)$ is transformed into $c0(X)$ using the previously mentioned strategy and heuristics, where

$c0(a).$
$c0(b).$
$c0(c).$
$c0(d).$

In the transformation, a clause $c0(Y) : -c0(Z), e(Z,Y).$ is created using folding. A tabulation mechanism of OLDT corresponds to folding in the unfold/fold transformation.

There is a kind of Prolog problem that cannot be solved efficiently in any resolution methods (top-down approaches) such as:[5]

[5]The problem was called an "Overbeek's Problem" by Hasegawa at ICOT.

$p(Y) : -p(X), p(i(X, X))$.
$p(i(i(i(X, Y), Z), i(i(Z, X), i(U, X))))$.
$? - p(i(i(i(a, b), a), a))$.

A theorem prover MGTP (Model Generation Theorem Prover)[Has94] can solve above problem in a bottom-up approach.

The author is not concerned here with more effective heuristics of cu-Prolog with more factors and with a non-linear formula. Such heuristics are mainly discussed by Hasida's DP[Has91].

Chapter 4

Applications of cu-Prolog to Natural Language Analysis

4.1 Introduction

This chapter explains three applications of cu-Prolog to constraint-based natural language analysis. The first is the treatment of disjunctive feature structures (DFS) that are fundamental devices to store ambiguities in feature structures. A DFS is treated as a PST followed by a set of constraint. The DFS unification is achieved with the PST unification followed by constraint solving.

The next application is the most successful application of cu-Prolog: a constraint-based JPSG (Japanese Phrase Structure Grammar) parser. Lexical ambiguities and structured principles are equally processed using CHC.

The last application is an experimental application of the constraint transformer of cu-Prolog to process CFG parsing. Structured ambiguities are dealt with the constraint transformation based on the unfold/fold transformation.

4.2 Disjunctive Feature Structure unification in cu-Prolog

In this section, the author applies cu-Prolog to processing disjunctive linguistic information. Natural language is inherently ambiguous. The ambiguity can be seen in the lexical level such as polysemic words, structure level such as PP-attachment [1], discourse level, and so on.

In the constraint-based frameworks, feature structures has been extended to treat the

[1]Such as "John saw a man with a telescope."

ambiguity as disjunctive feature structures (DFS)[Kay85]. This section surveys several works about DFSs and explains how cu-Prolog can treat DFSs and their unification.

4.2.1 DFS

As described in Chapter 2, there are three kinds of disjunctive feature structures : value disjunctions, general disjunctions, and disjunction names.

A serious problem in treating DFSs is the computational complexity of their unification. Kasper[KR86] examined the complexity of unification between DFSs showing that any unification algorithms for DFS have a non-polynomial worst-case complexity (if P≠NP). After that, some practical algorithms with better average performance have been studied by [Kas87] and [ED88]. When every disjunction is expanded as DNF (disjunctive normal form), the number of formulas becomes exponential and hence the unification will be a hard problem. These practical approaches delay the expansion of disjunctions as late as possible.

4.2.2 DFSs as constrained PSTs

cu-Prolog requires no special device to represent DFSs. Every type of DFS can be expressed with a constrained PST as shown in Table 4.1.

4.2.3 DFS unification

Every kind of DFS has a corresponding constrained PST. Thus, the unification between DFSs falls into the unification between constrained PSTs. The unification between constrained PSTs is performed by the unification between PSTs followed by the transformation of dependent constraints.

Consider the following example to unify two DFSs[ED88]:

$$\left[\mathtt{a} : \left\{ \left[\begin{array}{l} \mathtt{b} : + \\ \mathtt{c} : - \end{array} \right], \left[\begin{array}{l} \mathtt{b} : - \\ \mathtt{c} : + \end{array} \right] \right\} \right] \tag{4.1}$$

and

$$\left[\begin{array}{l} \mathtt{a} : \left[\mathtt{b} : \mathtt{V} \right] \\ \mathtt{d} : \mathtt{V} \end{array} \right]. \tag{4.2}$$

These DFSs correspond to two constrained PST as

$X = \{a/U\}, s(U)$ and
$Y = \{a/\{b/V\}, d/V\}$

Table 4.1: DFS as constrained PST

DFS in AVM notation	constrained PST

value disjunction:
$\left[\begin{array}{l} pos : \{n, v\} \\ sc : \left\{ \begin{array}{l} \langle\rangle \\ \left\langle \left[\; pos : p \; \right] \right\rangle \end{array} \right\} \end{array} \right]$
	`V={pos/X, sc/Y},`
	`  c0(X), c1(Y)`
	`  where`
	`c0(n).`
	`c0(v).`
	`c1([]).`
	`c1([{pos/p}]).`

general disjunction:
$\left[\left\{ \begin{array}{l} \left[\; pos : n \; \right] \\ \left[\begin{array}{l} pos : v \\ vform : vs \\ sc : \left\langle \left[\; pos : p \; \right] \right\rangle \end{array} \right] \end{array} \right\} \right.$
$\left. sem : love \right]$
	`U={sem/love}, c2(U).`
	`  where`
	`c2({pos/n}).`
	`c2({pos/v, vform/vs,`
	`  sc/[{pos/p}]}).`

disjunction name:
$\left[\begin{array}{l} syn : \left[arg : \left[\left[case : d_1 \left\{ \begin{array}{l} dat \\ acc \end{array} \right\} \right] \right] \right] \\ sem : \left[rel : d_1 \left\{ \begin{array}{l} dir\_in \\ stat\_in \end{array} \right\} \right] \end{array} \right]$
	`V={syn/{arg/{case/ X}},`
	`  sem/{rel/Y}},c3(X,Y).`
	`  where`
	`c3(dat, stat_in).`
	`c3(acc, dir_in).`

where a constraint predicate $s/1$ is defined by:

$$s(\{b/+, c/-\}).$$
$$s(\{b/-, c/+\}).$$

The component of the first argument of s is $Cmp(s, 1) = \{b, c\}$.

The unification between X and Y gives

$$X = Y = \{a/U, d/V\}, U = \{b/V\}, s(U). \tag{4.3}$$

There is a dependency by U of the second and third formula with a label b. Note that $U = \{b/V\}$ is equivalent to $t0(U, V)$, where $t0/2$ is defined as a constraint definition clause:

$$t0(\{b/V\}, V).$$

A new predicate $c1/2$ is defined as

$$c1(U, V) :- t0(U, V), s(U).$$

By means of the unfold transformation, the definition clauses of $c1/2$ becomes

$$c1(\{b/+, c/-\}, +).$$
$$c1(\{b/-, c/+\}, -).$$

Finally, (4.3) is transformed into

$$X = Y = \{a/U, d/V\}, c1(U, V). \tag{4.4}$$

Note that the result contains no dependency.

4.2.4 Comparison with Kasper's approach

Compare the constrained PST with Kasper's treatment of DFS[Kas87]. Kasper postulates DFSs are of the form:

$$uconj \wedge disj_1 \wedge \cdots \wedge disj_m$$

uconj is called an *unconditional conjunct* that contains no disjunction. Each $disj_i (i = 1, \cdots, m)$ is a disjunction of two or more alternatives. In the definition of a constrained PST ([Def] 7 of Section 3.3), $V = p$ corresponds to the unconditional conjunct and $c_1(X), c_2(X), \ldots, c_n(X)$ the disjunctions.

In the Kasper's approach, DFS unification consists of three procedures:

1. definite component unification

2. compatibility checking, and

3. exhaustive consistency checking.

Each computational order is $O(nlogn)$, $O(d^2nlogn)$, and $O(2^{(d/2)})$ for the total number of symbols n and number of disjunctions d. Kasper's algorithm requires exponential computational time in the worst case. In practical cases, however, it's complexity is $O(n^3)$. That is because the third procedure is seldom required, and $d < n$.

PST unification corresponds to the first procedure, and the following constraint transformation corresponds to the second and third procedures. In the worst case where all constraints interact each other, we have to unfold all the constraints, which requires exponential time of the number of disjunctions (that is, the product of the number of each definition clause), but in reality our approach requires polynomial time, as Kasper's does.

The cu-Prolog approach is superior to Kasper's in the following points:

- checking is performed by unfolding only dependent PSTs,
- an unfolding formula is selected by applying heuristics as shown in Section 3.6, and
- constrained PSTs can treat disjunction names [DE90] and disjunction among more than one feature structures[Tsu91] in the same way.

Figure 4.1 is an example of processing a DFS unification example of [Kas87] in cu-PrologIII. It demonstrates the unification between

$$\left[subj : \left[\begin{array}{l} lex : yall \\ person : 2 \\ number : pl \end{array} \right] \right] \tag{4.5}$$

and

$$\begin{array}{l} \left[\begin{array}{l} rank : clause \\ subj : [case : nom] \end{array} \right] \\ \wedge \left(\left[\begin{array}{l} voice : passive \\ transitivity : trans \\ [< subj >, < goal >] \end{array} \right] \vee \left[\begin{array}{l} voice : active \\ [< subj >, < actor >] \end{array} \right] \right) \\ \wedge \left(\left[\begin{array}{l} transitivity : intrans \\ actor : [person : 3] \end{array} \right] \vee \left[\begin{array}{l} transitivity : trans \\ goal : [person : 3] \end{array} \right] \right) \\ \wedge \left(\left[\begin{array}{l} number : sing \\ subj : [number : sing] \end{array} \right] \vee \left[\begin{array}{l} number : pl \\ subj : [number : pl] \end{array} \right] \right). \end{array} \tag{4.6}$$

Here, $[< subj >, < goal >]$ indicates an path equivalence, that is, the value of feature $subj$ is equal to the value of $goal$.

```
%% definition of the unconditional conjuncts (user's input)
cc1({voice/passive,trans/trans,subj/X,goal/X}).
cc1({voice/active, subj/X,actor/X}).
cc2({trans/intrans, actor/{person/third}}).
cc2({trans/trans, goal/{person/third}}).
cc3({numb/sing, subj/{numb/sing}}).
cc3({numb/pl, subj/{numb/pl}}).

%% Disjunctive Feature Structure unification (user's input)
@ U={rank/clause, subj/{case/nom}},cc1(U),cc2(U),cc3(U),
    U={subj/{lex/,person/second,numb/pl}}.

%% answer: equivalent constraint
solution = c0(U_0, {subj/{case/nom}, rank/clause},
               {subj/{person/second, numb/pl, lex/yall}})

%% definitions of a new predicate (c0)
c0(_p1, _p1, _p1) :- cc2(_p1), cc1(_p1);
   _p1={subj/{person/second, numb/pl, case/nom, lex/yall},
               numb/pl, rank/clause}.

CPU time = 0.150 sec (Constraints Handling = 0.000 sec)

>:-c0(X,_,_).          % solve the new constraint
success.                 % X is the final answer of the unification.
  X = {voice/active, trans/trans, subj/{person/second,
     numb/pl, case/nom, lex/yall}, goal/{person/third},
     actor/{person/second, numb/pl, case/nom, lex/yall},
     numb/pl, rank/clause};
```

This is a demonstration of a DFS unification using the constraint transformer. The first 7 lines define disjunctions in (4.6) with user-defined predicates. In cu-PrologIII, a constraint that follows "@" at the top level is transformed into modular one. In this case, it specifies the unification between (4.6) and (4.5). The constraint transformer returns an equivalent modular constraint and definition clauses of newly defined predicates. The result of the unification, which is a non-disjunctive FS in this case, is given as the binding of X in the last 3 lines.

Figure 4.1: DFS unification

4.3 Processing JPSG in cu-Prolog

This section demonstrates another application of cu-Prolog: processing JPSG (Japanese Phrase Structure Grammar).

4.3.1 Constraint-based NL analysis

This section will show both lexical constraints and structural principles are uniformly treated as constraints in CHCs of cu-Prolog. Moreover, constraints are accumulated to reduce the value range of variables. In other words, a disambiguation process is automatically realized by constraint transformation. This gives a picture of *constraint-based natural language analysis.*

Most traditional approaches, on the other hand, are procedural and backtrack-based. That is, a parser returns an answer, then backtracks to search another answer. Also, phonological, syntactic, semantic, and pragmatic processes are applied, one by one.

4.3.2 JPSG Parsing in cu-Prolog

Using DCG [PW80], CFG parsing is essentially realized as a following simple Prolog program.

```
parse(S0,S1,Cat) :- lexicon(S0,S1,Cat).
parse(S0,S1,Cat) :- parse(S0,S2,Cat1),parse(S2,S1,Cat2),
                    psr(Cat1,Cat2,Cat).
```

Here, `S0`, `S1`, and `S2` represent strings. `parse(S0,S1,Cat)` means that `S1` is a last part of `S0` and the string which subtracts `S1` from `S0` is parsed to be a category `Cat`. The coding style is called a *difference list* because a parsed string is represented as a difference of two strings.

Let `Cat`, `Cat1`, and `Cat2` be categories. The first clause looks up a dictionary entry that is defined as `lexicon/3`. The second clause means that the string `S0 - S1` comprises a category `Cat`, when its substring `S0 - S2` is a category `Cat1`, and the other substring `S2 - S1` comprises `Cat2`, and there is a phrase structure `psr/3` whose mother is `Cat` and their daughters are `Cat1` and `Cat2`.

In constraint-based grammar formalisms such as JPSG and HPSG, constraints are stored in various places. For example, lexical entries are equipped with lexical constraints, and the relations among categories of local phrase structure are given as structural constraints. These constraints are implemented in Prolog as follows.

```
parse(S0,S1,Cat) :- lexicon(S0,S1,Cat),
                        lexical_constraint(S0,S1,Cat).
parse(S0,S1,Cat) :- parse(S0,S2,Cat1),parse(S2,S1,Cat2),
                    psr(Cat1,Cat2,Cat).
psr(Mother,Left,Right) :- structure_constraint(Mother,Left,Right).
```

Above program, however, has several defects mainly because the processing order of Prolog is fixed as mentioned in Section 3.2.

In cu-Prolog, on the other hand, those constraints are encoded as user-defined Prolog predicates. Here, CFG parsing is encoded in the Prolog part of a CHC and only constraints are encoded in the constraint part.

```
parse(S0,S1,Cat) :- lexicon(S0,S1,Cat);
                        lexical_constraint(S0,S1,Cat).
parse(S0,S1,Cat) :- parse(S0,S2,Cat1),parse(S2,S1,Cat2),
        psr(Cat1,Cat2,Cat);
        structure_constraint(Cat1,Cat2,Cat).
```

`lexical_constraint/3` and `structure_constraint/3` are constraint predicates processed with the constraint solver.

Appendix B illustrates a HPSG parser and a JPSG parser in cu-Prolog. In either program, constraints are embedded in a left corner CFG parser program. A left corner parser parses a string from left to right. It is implemented as a Prolog program listed below. (Figure 4.2)

```
parse(S0,S2,Cat) :- lexicon(S0,S1,LC),
             parse1(LC,S1,S2,Cat).
    % lookup leftmost category
parse1(C,S,S,C).
parse1(LC,S0,S2,Cat) :-
     psr(LC,RC,MC), parse(S0,S1,RC),
     parse1(MC,S1,S2,Cat).
    % LC followed by (S0,S2) is parsed as Cat
```

`parse1(LC,S1,S2,Cat)` means the substring S1-S2 that follows a category LC is parsed to be a category Cat.

Besides the above parser program, lexical constraints and structural principles can be embedded in `psr/3` and `lexicon/3` as follows.

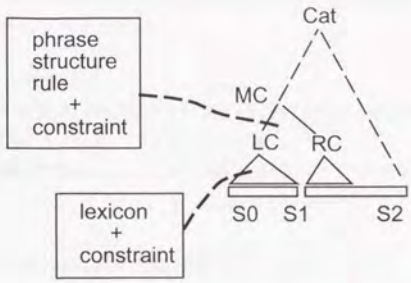


Figure 4.2: Left corner parser

```
lexicon(S0,S1,Cat) :- dictionary(S0,S1,C);
                      lexical_constraint(C,Cat).
psr(D,Head,Mother);
   sc_p(D,Head,Mother),
   head_p(Head,Mother),
   ph_p(D,Head,Mother).
```

Although constraints are stored in various places, they are uniformly solved with the constraint solver in a derivation path of CHC. Constraints imposed on a variable are accumulated and solved to decrease the value range of the variable. Namely, disambiguation process is automatically realized as a constraint transformation of cu-Prolog.

Following two subsections discuss two topics of the JPSG parser in cu-Prolog.

4.3.3 Encoding Lexical Ambiguity

As an example of the usage of DFSs, consider dictionary entries of homonyms or polysemic words. If an ambiguous word is stored into multiple lexical entries, the parsing process may be inefficient in that it sometimes backtracks to consult the lexicon. In constraint-based NLP, such ambiguity is packed as a lexical constraint.

Below is a sample lexical entry of a Japanese auxiliary verb "reru." "reru" follows a verb whose inflection type is `vs` or `vs1`. If the adjacent verb is transitive, "reru" indicates plain passive. [2] If the adjacent verb is intransitive, "reru" indicates affective passive [3].

[2]For example, "Ken ga Naomi ni ai-sa-reru" (Ken is loved by Naomi.)
[3]For example, "Ken ga ame ni fu-ra-reru" (Ken is affected by the rain.)

These combinations are represented by adding constraints `reru_form/1` and `reru_sem/4` to a lexical entry as follows.

```
%% lexical entry of ''reru''
seclex(reru,{sc/SC, sem/Sem, adjacent/{pos/v,infl/Inf,sc/VSC,sem/VSem}});
    reru_form(Inf),                 % inflection
    reru_sem(VSC,VSem,SC,Sem).      % combination of subcat and sem
                                    %  (constraints)

%%%%%% definition of constraints  %%%%%%
reru_form(vs).  % inflection type of the adjacent verb
refu_form(vs1).

% constraint for intransitive (affective) passive
reru_sem([{form/ga,sem/Sbj}],Sem,[{form/ga,sem/A},
          {form/ni,sem/Sbj}],affected(A,Sem)).
% constraint for transitive (normal) passive
reru_sem([{form/ga,sem/Sbj},{form/wo,sem/Obj}],Sem,
          [{form/ga,sem/Obj},{form/ni,sem/Sbj}],Sem).
```

`reru_form(Inf)` defines the value range of inflection type of the adjacent verb. `reru_sem(VSC,VSem,SC,Sem)` defines relations among subcategorization of the adjacent verb `VSC`, semantics of the adjacent verb `VSem`, subcategorization of "reru" `SC`, and semantics of "reru" `Sem`.

This lexical entry corresponds to the following DFS.

$$\left[\begin{array}{l} \left\{\begin{array}{l} \left[\begin{array}{l} adjc : \left[\begin{array}{l} sc : \left\langle\ [pos : ga, sem : S1]\ \right\rangle \\ sem : Sem1 \end{array}\right] \\ sc : \left\langle \left[\begin{array}{l} form : ga \\ sem : A \end{array}\right], \left[\begin{array}{l} form : ni \\ sem : S1 \end{array}\right] \right\rangle \\ sem : affected(A, Sem1) \end{array}\right] \\ \left[\begin{array}{l} adjc : \left[\begin{array}{l} sc : \left\langle\ [pos : ga, sem : S2], [pos : wo, sem : O2]\ \right\rangle \\ sem : Sem2 \end{array}\right] \\ sc : \left\langle \left[\begin{array}{l} pos : ga \\ sem : O2 \end{array}\right], \left[\begin{array}{l} pos : ni \\ sem : S2 \end{array}\right] \right\rangle \\ sem : Sem2 \end{array}\right] \end{array}\right\} \\ adjc : \left[\begin{array}{l} pos : v \\ infl : \{vs1, vs2\} \end{array}\right] \end{array}\right]$$

Although the lexical entry is ambiguous, many kinds of constraints are automatically accumulated to be solved during parsing.

4.3.4 Encoding Structural Principle

As mentioned in Section 2.2, structural principles of JPSG and HPSG are relations among feature structures in a local phrase structure. As shown in Subsection 4.3.1, structure principles are encoded as constraints in a phrase structure rule as:

$$psr(M, D, H); sp_1(M, D, H), \ldots, sp_n(M, D, H).$$

Here, $psr/3$ is a phrase structure rule and each sp_i $(i = 1 \ldots n)$ indicates a structure principle.

In cu-Prolog, these structural principles are evaluated flexibly with heuristics. A Prolog program equivalent to the above phrase structure rule is represented as:

$$psr(M, D, H) :- sp_1(M, D, H), \ldots, sp_n(M, D, H).$$

Each principle is, however, always evaluated sequentially (from left to right). Prolog, therefore, is not well suited for processing constraint-based grammars because it is impossible to stipulate in advance which linguistic constraint is processed, and in what order.

Consider structural principles of JPSG explained in Subsection 2.3.3. The head feature principle is represented as constraint $hfp(M, D, H)$ defined as below.

$$hfp(\{core/H\}, \_, \{core/H\}). \tag{4.7}$$

The subcat feature principle is realized as constraint $sfp(M, D, H)$ with the following definition.

$$sfp(\{subcat/MS\}, D, \{subcat/HS\}) : -union(MS, D, HS). \tag{4.8}$$

The foot feature principle is encoded as constraint $ffp(M, D, H)$ defined as follows.

$$ffp(\{slash/MSL\}, \{slash/DSL\}, \{slash/HSL\}) : -union(DSL, HSL, MSL). \tag{4.9}$$

4.3.5 Example

The author illustrates an example of the JPSG parser implemented on cu-PrologIII that parses ambiguous Japanese sentences.

For an ambiguous sentence, the parser returns the corresponding feature structure with remaining constraints. The ambiguity of the sentence is stored in the definition clauses

of remaining constraints. By evaluating the constraints, we can see how ambiguous the sentence is.

Figure 4.3 shows a parse tree of a sentence (4). The parser returns a parse tree followed by a top feature structure and remaining constraints. Sentence (4) is not ambiguous, so the top category does not have constraints on the `sem` feature in the Figure. [4]

(4) Ken-ga Naomi-wo ai-suru.
Ken-NOM Noami- love-PRES
'Ken loves Naomi.'

On the other hand, sentence (5) is ambiguous such as "Ken-ga (someone wo) ai-suru" and "Ken ga ai-suru (someone)" (relative clause).

(5) Ken-ga ai-suru.
Ken-NOM love-PRES
'Ken loves (someone).' or '(Someone) whom Ken loves'

Figure 4.4 shows a parsing result of (5). The top feature structure still has remaining constraint `c31/10` after parsing. The ambiguity of the sentence is automatically packed as a definition of `c31`. By solving `c31`, we can see the content. Here, there are two results. The ambiguity comes from a constraint to share elements between a subcat feature and a slash feature, attached as a part of a lexical rule. In the JPSG parser program (Appendix B), the constraint is implemented as `sc_sl_move/3`.

4.4 CFG parsing as constraint transformation

In the JPSG parser explained in Section 4.3, CFG parsing was implemented in the Prolog part of CHCs for efficiency. Consequently, the parser can not handle ambiguity on syntactic parse trees [5] because the parsing algorithm is written procedurally in the Prolog part of CHCs.

This section introduces an experimental extension of the constraint transformation of cu-Prolog, called Dependency Propagation (DP) [TH90, Has91, Has90] which regards constraint transformation as computation. As an example of DP, a simple CFG is parsed only with constraint transformation. The content of this section is a cooperative work with Kôiti Hasida[TH90].

[4] A small lexical constraint about the form of the sentence remains as `syu_ren(Form)`.

[5] For example, the ambiguity in "I saw a man with a telescope". The ambiguity is called *structural ambiguity*[Cry97].

```
tsuda#icot21[5]% cup3 j4.p %% Start cu-Prolog with JPSG parser

*****  cu - Prolog III  *****
Copyright: Institute for New Generation Computer Technology,  Japan 1989-91
in Cooperation with SIRAI@sccs.chukyo-u.ac.jp
M-solvable mode (help ->  %h)

%%%%% Example1. ``Ken-ga Naomi-wo ai-suru''(Ken loves Naomi.) %%%%%%
_:-p([ken,ga,naomi,wo,ai,suru]). %% Input

%% The parser returns the parse tree.
{sem/[love,ken,naomi], core/{form/Form_3670, pos/v}, sc/[], refl/[],
    slash/[], psl/[], ajn/[], ajc/[]}---[suff_p]
 |
 |--{sem/[love,ken,naomi], core/{form/vs2, pos/v}, sc/[], refl/[],
       slash/[], psl/[], ajn/[], ajc/[]}---[subcat_p]
 |   |
 |   |--{sem/ken, core/{form/ga, pos/p}, sc/[], refl/[], slash/[],
 |          psl/[], ajn/[], ajc/[]}---[adjacent_p]
 |   |   |
 |   |   |--{sem/ken, core/{form/n, pos/n}, sc/[], refl/[], slash/[],
 |   |          psl/[], ajn/[], ajc/[]}---[ken]
 |   |   |
 |   |   |__{sem/ken, core/{form/ga, pos/p}, sc/[], refl/[], slash/[],
 |   |          psl/[], ajn/[], ajc/[{sem/ken, core/{pos/n}, sc/[],
 |   |          refl/ReflAC_140}]}---[ga]
 |   |
 |   |__{sem/[love,ken,naomi], core/{form/vs2, pos/v},
 |      sc/[{sem/ken, core/{form/ga, pos/p}}],
 |      refl/[], slash/[], psl/[], ajn/[], ajc/[]}---[subcat_p]
 |       |
 |       |--{sem/naomi, core/{form/wo, pos/p}, sc/[], refl/[],
 |       |      slash/[], psl/[], ajn/[], ajc/[]}---[adjacent_p]
 |       |   |
 |       |   |--{sem/naomi, core/{form/n, pos/n}, sc/[], refl/[],
 |       |   |      slash/[], psl/[], ajn/[], ajc/[]}---[naomi]
 |       |   |
 |       |   |__{sem/naomi, core/{form/wo, pos/p}, sc/[], refl/[], slash/[],
 |       |          psl/[], ajn/[], ajc/[{sem/naomi, core/{pos/n},
 |       |          sc/[], refl/ReflAC_960}]}---[wo]
 |       |
 |       |__{sem/[love,ken,naomi], core/{form/vs2, pos/v}}---[ai]
 |
 |__{sem/[love,ken,naomi], core/{form/Form_3670, pos/v}, sc/[],
    refl/[], slash/[], psl/[], ajn/[], ajc/[{sem/[love,ken,naomi],
    core/{form/vs2, pos/v}, sc/[], refl/ReflAC_3702}]}---[suru]

category= {sem/[love,ken,naomi], core/{form/Form_3670, pos/v},
sc/[], refl/[], slash/[], psl/[], ajn/[], ajc/[]} %% Top feature structure

constraint= syu_ren(Form_3670) %% Remaining constraints

true.
CPU time = 0.750 sec
19%(program) 6%(pst/const) 12%(string)
```

Figure 4.3: The parsing of "Ken ga Naomi-wo ai-suru."

```
%%%%% Example2. ''Ken-ga ai-suru''(Ken loves). %%%%%
_:-p([ken,ga,ai,suru]). %% Input

{sem/[love,Sbj_394,Obj_396], core/{form/Form_1056, pos/v},sc/Msc_1128,
    refl/Mref_1130, slash/Msl_1132, psl/[], ajn/[], ajc/[]}---[suff_p]
 |
 |--{sem/[love,Sbj_394,Obj_396], core/{form/vs2, pos/v}, sc/Csc_1102,
    refl/Cref_1104, slash/Csl_1106, psl/[], ajn/[], ajc/[]}---[subcat_p]
 |   |
 |   |--{sem/ken, core/{form/ga, pos/p}, sc/[], refl/[], slash/[],
          psl/[], ajn/[], ajc/[]}---[adjacent_p]
 |   |   |
 |   |   |--{sem/ken, core/{form/n, pos/n}, sc/[], refl/[], slash/[],
              psl/[], ajn/[], ajc/[]}---[ken]
 |   |   |
 |   |   |__{sem/ken, core/{form/ga, pos/p}, sc/[], refl/[], slash/[],
              psl/[], ajn/[], ajc/[{sem/ken, core/{pos/n}, sc/[],
              refl/Refl_AC_140}]}---[ga]
 |   |
 |   |__{sem/[love,Sbj_394,Obj_396], core/{form/vs2, pos/v}}---[ai]
 |
 |__{sem/[love,Sbj_394,Obj_396], core/{form/Form_1056, pos/v}, sc/[],
    refl/[], slash/[], psl/[], ajn/[], ajc/[{sem/[love,Sbj_394,Obj_396],
    core/{form/vs2, pos/v}, sc/[], refl/Refl_AC_1116}]}---[suru]

category= {sem/[love,Sbj_394,Obj_396], core/{form/Form_1056, pos/v},
         sc/Msc_1128, refl/Mref_1130, slash/Msl_1132, psl/[], ajn/[], ajc/[]}
constraint= c31(Cref_1104, Mref_1130, Csl_1106, Msl_1132, Csc_1102,
        Msc_1128, Obj_396, Sbj_394, HC_222, Hsl_226),syu_ren(Form_1056)
true.
CPU time = 0.367 sec
20%(program) 2%(pst/const) 13%(string)
%% solve remaining constraints of the top structure (c31)
_:-c31(_,Refl,_,Slash,_,SC,Obj,Sbj,_,_).
  Refl = []  Slash = []  SC = [{sem/V0_36, core/{form/wo, pos /p}}]
Obj = V0_36  Sbj = ken;     %% First solution: subcat remains.
  Refl = []  Slash = [{sem/V3_58}]  SC = []  Obj = V3_58  Sbj = ken;
%% Second solution: One element moves from subcat to slash.
no.
CPU time = 0.050 sec
20%(program) 0%(pst/const) 13%(string)
-
```

Figure 4.4: The parsing of "Ken ga ai-suru."

4.4.1 Dependency

A trigger of constraint transformation is a *dependency* among literals. A variable occurring in more than one distinct non-vacuous places in a clause has *dependency*. When an argument place of a predicate is a variable in all of its definition clauses, the argument place is called a *vacuous argument place* explained in Subsection 3.5.1. For example, the first argument place of `member` defined below is vacuous.

```
member(E,[E|_]).
member(E,[_|S]) :- member(E,S).
```

Variables in vacuous argument places are represented with `#` as follows.

```
member(X#,Y),c0(X,Z)
```

In the above, though variable `X` occurs in two places, there is no dependency because the first `X` is vacuous.

4.4.2 Trans-clausal variable

Trans-clausal variables correspond to global variables of many programming languages. They are treated as if they were constants in some context. We put `*` in front of a trans-clausal variable as follows.

```
:-vp(*V0,B),*V0=[see|*V1],*v1=[a,man|*V2].
```

Constraint transformation is executed so as to eliminate dependency of goal clauses or a body of program clauses, therefore is more general than Earley deduction [PW83] which executes the body of each clause in the fixed left-to-right order.

4.4.3 Penetration

To process vacuous variables and trans-clausal variables, we introduce two *penetration* operation in addition to the unfold/fold transformation.

Downward penetration is to replace a literal that contains trans-clausal variables with a new literal that contains no trans-clausal variable. The operation is a combination of definition and unfolding. For example,

```
:-p(*V0,B),*V0=[a|*V1].
p([a|X],X).
p(X,Z):-p(X,Y),p(Y,Z).
```

Table 4.2: Simple ambiguous CFG grammar

$$
\begin{array}{rcl}
VP & \rightarrow & V\ NP \\
VP & \rightarrow & VP\ PP \\
NP & \rightarrow & NP\ PP \\
V & \rightarrow & \text{'see'} \\
NP & \rightarrow & \text{'a man'} \\
PP & \rightarrow & \text{'with a telescope'}
\end{array}
$$

are transformed to the following clauses. `p0/1` is a new predicate and `p0(B)` is equivalent to `p(*V0,B)`. The dependency concerning `*V0` in the first goal is dissolved.

```
:-p0(B),*V0=[a|*V1].
p0(*V1).
p0(Z):-p(*V0,Y),p(Y,Z).
```

Upward penetration is to reduce a unit clause containing trans-clausal variables so as to change some argument places to begin vacuous. For example, let

```
p0(*V1).
p0(Z):-p0(Y),p(Y,Z).
```

be all the clauses that define `p0/1`. The argument place of `p0` is not vacuous because the trans-clausal variable `*V1` in the first clause is considered as a constant.

By replacing `p0(*V1)` with a new predicate `p1/0`, they are transformed to following clauses.

```
p1.
p0(Z):-p1,p(*V1,Z).
p0(Z):-p0(Y#),p(Y,Z).
```

Then the argument place of `p0` becomes vacuous.

4.4.4 Parsing an ambiguous CFG

Let us consider a simple ambiguous context-free grammar in Table 4.2. This grammar produces structural ambiguity of the PP attachment in "I see a man with a telescope."

Parsing program in terms of this grammar can be formulated as follows.

```
(C0) *V0=[see|*V1],
     *V1=[a,man|*V2],
     *V2=[with,a,telescope|*V3],
     *V3=NIL
(C1) :-vp(*V0,B).
(C2) v([see|W],W).
(C3) np([a,man|W],W).
(C4) pp([with,a,telescope|W],W).
(C5) vp(X,Z):-v(X,Y#),np(Y,Z).
(C6) vp(X,Z):-vp(X,Y#),pp(Y,Z).
(C7) np(X,Z):-np(X,Y#),pp(Y,Z).
```

There is only one dependency to be eliminated: `*V0` in (C1). Here, we introduce a new predicate `vp0` as `vp0(V) = vp(*V0,V)`. By downward penetration, (C1) is replaced with the following clauses.

```
(C1') :-vp0(B).
(C8) vp0(V):-vp(*V0,Y),pp(Y,V).
(C9) vp0(V):-v(*V0,Y),np(Y,V).
```

By folding the first literal in the body of (C8), we have

```
(C8') vp0(V):-vp0(Y),pp(Y,V).
```

Next, we process the dependency concerning `*V0` in the body of (C9). Let `v0(V)=v(*V0,Y)` and by downward penetration, we get

```
(C9') vp0(V):-v0(Y),np(Y,V).
(C10) v0(*V1).
```

As `v0` has only one definition clause (C10), then it is reduced.

```
(C9') vp0(V):-np(*V1,V).
```

Let `np1(V)=np(*V1,V)` and by downward penetration,

```
(C9") vp0(V):-np1(V).
(C11) np1(*V2).
(C12) np1(V):-np(*V1,Y),pp(Y,V).
```

Fold the first literal of the body of (C12), then

```
(C12') np1(V):-np1(Y),pp(Y,V).
```

The argument place of `np1` is not vacuous, then we apply downward penetration to `np1`. Here, `np12=np1(*V2)`.

```
(C11') np12.
(C13) np1(V):-np12,pp(*V2,V).
(C12') np1(V):-np1(Y),pp(Y,V).
```

We have to consider the dependency of the second literal of the body of (C13). Here, let pp2(V)=pp(*V2,V).

```
(C13') np1(V):-np12,pp2(V).
(C14) pp2(*V3).
```

pp2 has only one definition, then is reduced.

```
(C13") np1(*V3).
```

The remaining definition of np0 is (C11') and (C13"). Then (C9") is reduced.

```
(C9-1) vp0(*V2).
(C9-2) vp0(*V3).
```

Apply upward penetration to (C8'), (C9-1), and (C9-2) introducing a new predicate as vp02=vp0(*V2) and vp03=vp0(*V3), then the definition of vp0 becomes as follows:

```
vp02.
vp03.
vp0(V):-vp02,pp(*V2,V).
vp0(V):-vp03,pp(*V3,V).
vp0(V):-vp0(Y),pp(Y,V).
```

Finally, it is transformed to

```
vp02.
vp03.
vp03.
vp0(V):-vp0(Y),pp(Y,V).
```

The two occurrences of vp03 correspond to the two meanings of "I see a man with a telescope".

4.4.5 Complexity

This subsection reviews the complexity of parsing on constraint transformation. (4.10–4.11) is a simple CFG example mentioned in [Has90].

$$P \rightarrow a \tag{4.10}$$

$$P \rightarrow PP \tag{4.11}$$

Parsing the string "aa ... a"(length is n) under this grammar may be formulated in terms of a set of constraints (4.12–4.14).

$$: -p(A^0, B), A^0 = [a|A^1], \cdots, A^{n-1} = [a|A^n]. \tag{4.12}$$

$$p([a|X], X). \tag{4.13}$$

$$p(X, Z) : -p(X, Y), p(Y, Z). \tag{4.14}$$

After some transformation steps, (4.15–4.21) is finally obtained.

$$: -q, A^0 = [a|A^1], \cdots, A^{n-1} = [a|A^n]. \tag{4.15}$$

$$q : -p_0(B^0). \tag{4.16}$$

$$q : -p_{0,i}, B^0 = A^i.(0 < i \leq n) \tag{4.17}$$

$$p_i(Z) : -p_{i,j}, p_j(Z).(0 \leq i < j < n) \tag{4.18}$$

$$p_i(Z) : -p_i(Y), p(Y, Z).(0 \leq i < n) \tag{4.19}$$

$$p_{i,i+1}.(0 \leq i < n) \tag{4.20}$$

$$p_{i,k} : -p_{i,j}, p_{j,k}.(0 \leq i < j < k < n) \tag{4.21}$$

Clauses in (4.21) forms a well-formed substring table, as in CYK algorithm, Earley's algorithm [Ear70], chart parser, tabulation technique [TS86], and so on. For instance, the existence of clause $\mathtt{p}_{i,k}$ `:-` $\mathtt{p}_{i,j}$, $\mathtt{p}_{j,k}$. means that the part of the given string from position i to position k has been parsed as having category P and is subdivided at position j into two parts, each having category P. Note that the computational complexity of the above process is $O(n^3)$ in terms of both space and time.

Moreover, the space complexity is reduced to $O(n^2)$ if we delete the literals irrelevant to instantiation of variables, which preserves the semantics of the constraints in the case of Horn programs. That is, the resulting structure would be:

$$: -q, A^0 = [a|A^1], \cdots, A^{n-1} = [a|A^n]. \tag{4.22}$$

$$q : -p_0(B^0). \tag{4.23}$$

$$q : -B^0 = A^i.(0 < i \leq n) \tag{4.24}$$

$$p_i(Z) : -p_j(Z).(0 \leq i < j < n) \tag{4.25}$$

$$p_i(Z) : -p_i(Y), p(Y, Z).(0 \leq i < n) \tag{4.26}$$

$$p_{i,j}.(0 \leq i < j < n) \tag{4.27}$$

The process illustrated above corresponds best to Earley's algorithm. Our procedure may be generalized to employ bottom-up control, so that the resulting process should be regarded as chart parsing, left-corner parsing, and so on.

4.4.6 Parsing CFG with feature structure as constraint transformation

This section tries to handle various types of constraints such as the constraints on feature structure or on phrase structures mentioned in the previous two sections. We have to investigate some heuristics to determine which constraint is processed earlier than the others.

Heuristics

In the following discussion, we consider only Horn clause constraint, and two types of linguistic constraint: constraint on feature structure and on phrase structure.

Following is a heuristic used in this section. This heuristic guarantees that the computation takes place in such a way that it may be looked upon as phrase-structure computation annotated with constraints on feature structures as in the approach of Section 2, just as people would like to regard parsing processes to be:

- A variable occurring in both types of constraint does not have dependency.
- Dependencies concerning feature structures should be eliminated earlier than those concerning phrase structures.
- Literals concerning phrase structures should be unfolded first when you attempt to eliminate dependency between literals about phrase structures and literals about feature structures.

Example

The program below is another formulation of the simple CFG (Table 4.2). We consider only one feature called `pos` [6] that takes a part of speech such as `np`,`vp`,`pp`,and so on. `pos` feature follows the constraint that corresponds to the first three rules of Table 4.2:

> The combination of the value of `pos` feature of mother, left daughter, and right daughter category is (`vp`,`n`,`np`),(`np`,`np`,`pp`), or (`vp`,`vp`,`pp`).

[6] It is different from `pos` feature of JPSG.

In the following, constraints concerning phrase structure (predicate cst) and those concerning feature structure (predicate p) are separated by '|'. Let phrase structure constraints and feature structure constraints have no dependency.

```
(P0) *V0=[see|*V1],
     *V1=[a,man|*V2],
     *V2=[with,a,telescope|*V3],
     *V3=NIL
(P1) :-p(*V0,B,C).
(P2) p(X,Z,Cat):-p(X,Y#,LC),p(Y,Z,RC) |
                  cst(LC,RC,Cat).
(P3) p([see|W],W,v).
(P4) p([a,man|W],W,np).
(P5) p([with,a,telescope|W],W,pp).
(P6) cst(v,np,vp).
(P7) cst(np,pp,np).
(P8) cst(vp,pp,vp).
```

The dependency to be processed is in terms of *V0 in (P1) because LC and RC in (P2) do not have dependencies. Process downward penetration. Here, p0(B,C)=p(*V0,B,C).

```
(P1') :-p0(B,C).
(P9)  p0(*V1,v).
(P10) p0(B,C):-p(*V0,Y#,LC),p(Y,B,RC) |
                cst(LC,RC,C).
```

Fold the first literal of (P10).

```
(P10') p0(B,Cat):-p0(Y,LC),p(Y,B,RC) |
                   cst(LC,RC,Cat).
```

Upward penetration. Let p01=p0(*V1,v).

```
(P9) p01.
(P11) p0(B,Cat):-p01,p(*V1,B,RC) |
                  cst(v,RC,Cat).
(P10'') p0(B,Cat):-p0(Y#,LC),p(Y,B,RC) |
                    cst(LC,RC,Cat).
```

Unfold the feature constraint of (P11).

```
(P11') p0(B,vp):-p01,p(*V1,B,np).
```

Downward penetration. Here, p1(B)=p(*V1,B,np).

```
(P11") p0(B,vp):-p01,p1(B).
(P12) p1(*V2).
(P13) p1(Z):-p(*V1,Y,np),p(Y,Z,RC) |
              cst(np,RC,np).
```

Unfold the feature constraint of (P13) and fold p1.

```
(P13') p1(Z):-p1(Y),p(Y,Z,pp).
```

Upward penetration. Let p12=p1(*V2).

```
(P12') p12.
(P14) p1(Z):-p12,p(*V2,Z,pp).
(P13') p1(Z):-p1(Y#),p(Y,Z,pp).
(P15) p0(*V2,vp):-p01,p12.
```

Downward penetration p2(B)=p(*V2,Z,pp).

```
(P14') p1(Z):-p12,p2(Z).
(P16) p2(*V3).
(P17) p2(Z):-p(*V2,Y,LC),p(Y,Z,RC) |
              cst(LC,RC,pp).
```

Unfold the feature constraint of (P17), however it fails because there is no clause matching cst(LC,RC,pp). p2/1 has only one definition (P16). p1/1 is reduced to be:

```
(P14") p1(*V3).
```

By upward penetration introducing p13=p1(*V3), (P11') and (P15) become

```
(P11-1) p0(*V3,vp).
(P15') p0(*V2,vp).
```

(P11-1) corresponds to one interpretation : "see (a man with a telescope."
From (P15') let p02=p0(*V2,vp) and apply upward penetration to (P10'').

```
(P10-1) p0(B,Cat):-p02, p(*V2,B,RC) | cst(vp,RC,Cat).
```

Unfold the feature constraint of (P10-1).

```
(P10-2) p0(B,vp):-p02, p(*V2,B,pp).
```

It is finally becomes

```
(P10-3) p0(*V3,vp).
```

and corresponds to "(see a man) with a telescope."

Chapter 5

Quixote

5.1 Motivation

Natural language processing involves a very complex flow of information that cannot be stipulated in terms of procedural programming languages.n This requires some sort of tool that enables us to represent and to process this complexity.

cu-Prolog, explained in the previous chapters, gives a framework to embody this picture. cu-Prolog, however, does not have enough power to realize the context-based semantic structure of natural language, because it is declaratively equal to Horn-clause logic. What is required then is a transaction mechanism of *typed feature structures*[Car92b] and *situation-based* representation of semantics[BP83], as well as constraint-based logical inference.

In the FGCS project, the author contributed to the Quixote project in its design and implementation of constraint solving and its applications to natural language processing. Quixote is a general knowledge representation language based on logic programming. It has also several useful features for natural language processing. Corresponding to typed feature structures, Quixote is equipped with sort hierarchy and attribute terms that stores partial information. For a situation-based semantic description, the module mechanism of Quixote enables classified and context-dependent knowledge representation.

Remaining sections of this chapter explain Quixote from a logic programming viewpoint. Natural language applications of Quixote are explained in the next chapter.

5.2 Introduction

The aim of this chapter is to explain another knowledge representation language Quixote[TY94, Yok94]. Quixote is designed as a hybrid language for a deductive object-

oriented database (DOOD[KNN89, DKM91]) language and a constraint logic programming (CLP) language based on subsumption constraints. Quixote combines object-orientation concepts such as object identity and property inheritance, constraint representation and processing, and a mechanism called *module* to classify a large knowledge base[YTY92, YY92, YTM93]. In addition, its logical inference system is extended to make hypothetical reasoning and restricted abduction. Such features play important roles in applications such as legal reasoning, biological databases, and natural language processing.

Section 5.3 introduced basic language features of Quixote. Section 5.4 shows several implementation issues mainly concerning its constraint solver.

5.3 Quixote language

This section introduces the syntax and several basic knowledge representation features of Quixote from a logic-programming point of view.

5.3.1 Object Term

Quixote terms consist of :

- atom (basic object),
- variable,
- (complex) object term, and
- set of object terms (set term).

Basic object term

[Def] 19 (Basic object term) *A* basic object term *or* basic object *is an atom. Let B represent a set of basic objects.* □

Below is examples of basic objects.

$mozart, person, piano, violin, instrument$

Variable and Label

[Def] 20 (Variable) *A* variable *is either a* single value variable *or a* set value variable. *A single value variable ($X \in V_i$) takes an object term as its value. On the other hand, a set value variable ($X^* \in V_s$) takes a set term as its value. Let V be a set of variables, where $V = V_i \cup V_s, V_i \cap V_s = \emptyset$* □

[Def] 21 (Label) *A* label *is either a* single value label *or a* set value label. *A single value label ($l \in L_i$) takes a non-set value, whereas a set value label ($l^* \in L_s$) takes a set term as its value ($L_i \cap L_s = \emptyset$).* [1] □

Object term

Complex concepts composed of multiple basic objects, such as "opus 73 of Beethoven" are represented with *complex object terms* as:

$$op73[composer = beethoven]$$

where *op*73 is a basic object, *composer* is a label, and *beethoven* is an object term as the value.

An object term is defined as follows.

[Def] 22 (Object term) *An* object term *is a term of the following form.*

$$o[l_1 = t_1, \cdots, l_n = t_n] \quad (0 \leq n)$$

Here, let $o \in B$, $l_1, \cdots, l_n \in L_i$ where $l_i \neq l_j$ $(i \neq j)$, and $t_1, \cdots, t_n$ be object terms or variables ($\in V_i$). Let O represent a set of object terms. □

In the above definition, when $n = 0$, the object term is written as o, namely an basic object term, instead of $o[\,]$.

For example,

$$mozart$$
$$male[occupation = pianist]$$

are object terms. When $n > 0$, object terms are called *complex object terms*. An object term with variables is called a *parametric* object term. A variable-free object term is called a *ground* object term.

The label-value pairs of a complex object term are called *intrinsic properties* of the object. Unlike attribute terms (explained later in 5.3.3), only = is allowed as operators of complex object terms.

[1] In Quixote system, a set label is an atom followed by *, such as `instruments*`.

[Def] **23 (Set term)** *A* set term *is a set of ground (variable-free) object terms.* □

For example following terms are set terms.

$\{piano, violin\}$
$\{mozart, beethoven\}$

[Def] **24 (Canonical form of set term)** *A set term S is* canonical *when S satisfies the following condition.*

- $\forall e1, e2 \in S(e1 \neq e2), e1 \not\sqsubseteq e2, e2 \not\sqsubseteq e1$

□

Here, $\sqsubseteq$ is a subsumption relation between terms explained later in the next subsection. For example, a set term $\{int, 1\}$ is not canonical because $1 \sqsubseteq int$. The canonical form of the set is $\{int\}$.

Computing the canonical form of given set term S is to repeat the following procedure until there is no change.

> If there are two different element $e1$ and $e2$ in S and $e1 \sqsubseteq e2$, remove $e1$ from S.

5.3.2 Subsumption Relation

Partial relation between basic objects

Initially Quixote programmers give partial relations $\preceq$ between basic object terms in a program. The relation corresponds to "ISA" relation or "A-KIND-OF (AKO)" relation.

For example,

$mozart \preceq person$,
$piano \preceq instrument$, $violin \preceq instrument$

illustrate relations between concepts such as "Mozart *is a* person,", "violin is *a kind of* instrument" and "piano is *a kind of* instrument." For simplicity, we assume that $\preceq$ is a strict order without circularity. A complete lattice can be constructed from $\preceq$ relations among basic objects. [2]

[2] In constructing the lattice, Quixote sometimes assumes intermediate nodes. For example, when the user defines $c \preceq a, c \preceq b, d \preceq a, d \preceq b$, Quixote assumes a new atom tmp that meets the relation : $c \preceq tmp, d \preceq tmp, tmp \preceq a, tmp \preceq b$.

Subsumption relation between object terms

A partial relation $\preceq$ among basic object terms is extended to a *subsumption relation* $\sqsubseteq$ among object terms as follows:

[Def] 25 (Subsumption Relation) *Given two ground complex object terms* $t1 = o[l_1 = t_1, \cdots, l_n = t_n]$ *and* $t2 = o'[l'_1 = t'_1, \cdots, l'_m = t'_m]$.

$t2$ subsumes $t1$ *(written as* $t1 \sqsubseteq t2$*) when the following condition holds.*

$$o \preceq o' \text{ and } \forall l'_j, \exists l_i \quad l_i = l'_j \wedge t_i \sqsubseteq t'_j,$$

where $1 \leq j \leq m$ *and* $1 \leq i \leq n$. □

When $t_1 \sqsubseteq t_2 \wedge t_1 \sqsupseteq t_2$, we denote $t_1 \cong t_2$.

Example 6 Subsumption Relations

Following subsumption relations hold, if $male \sqsubseteq person$ and $pianist \sqsubseteq musician$ are initially defined.

$$apple[color = green] \sqsubseteq apple$$
$$male[age = 30, occupation = pianist] \sqsubseteq person[occupation = musician]$$

□

Let $\top$ and $\bot$ be special basic objects defined as:

$$\forall o \in O, o \sqsubseteq \top, \bot \sqsubseteq o.$$

Subsumption relation between set terms

The subsumption relation is extended to be a relation among set terms.

[Def] 26 (Subsumption relation between sets) *Given two set terms,* $S_1 = \{o_1, \cdots, o_n\}$ *and* $S_2 = \{o'_1, \cdots, o'_m\}$.

S_2 *subsumes* S_1 *(written as* $S_1 \sqsubseteq_H S_2$*), when the following condition holds, which is called Hoare ordering.*

$$S_1 \sqsubseteq_H S_2 \stackrel{def}{=} \forall o_i \in S_1, \exists o'_j \in S_2 \; o_i \sqsubseteq o'_j$$

□

When $1 \sqsubseteq int$, $2 \sqsubseteq int$, and so on are defined, the following subsumption relations hold.

$\{1,2,3\} \sqsubseteq_H \{1,2,3,4,5\}$
$\{1,2,3\} \sqsubseteq_H \{int\}$

Generally, Hoare ordering is not a partial order between set terms. For example, both $\{1, int\} \sqsubseteq_H \{2, int\}$ and $\{2, int\} \sqsubseteq_H \{1, int\}$ holds although $\{1, int\}$ and $\{2, int\}$ are different. However, the ordering becomes a partial order for canonical set terms. $\{int\}$ is the canonical form of both $\{1, int\}$ and $\{2, int\}$. So we assume without loss of generality that $\sqsubseteq_H$ is a partial order for Quixote set terms.

When $t_1^* \sqsubseteq_H t_2^* \wedge t_1^* \sqsupseteq_H t_2^*$, we denote $t_1^* \cong_H t_2^*$.

Comments on set ordering

In the current design of Quixote language, we use Hoare ordering ($\sqsubseteq_H$) between set terms. As a subsumption ordering between set terms, however, Smyth ordering can also be considered. Smyth ordering $\sqsubseteq_{sm}$ between set terms is defined as:

$$S_1 \sqsubseteq_{sm} S_2 \stackrel{def}{=} \forall o_i \in S_2, \exists o_j' \in S_1 \; o_j' \sqsubseteq o_i.$$

For Smyth ordering, however, the procedure to calculate the canonical set must be:

If there are two different element $e1, e2 \in S_0$ and $e1 \sqsubseteq e2$, remove $e2$ from S_0.

For example, the canonical form of $\{1, int\}$ is $\{1\}$ for Smyth ordering. [3]

Lattice, meet, and join

Since lattice construction procedure from a partially ordered set is a well known process, we assume that a set of object terms O (without variables) with $\top$ and $\bot$ can become a lattice $(O, \sqsubseteq, \top, \bot)$ without loss of generality.

The meet and join operations of object terms o_1 and o_2 are denoted by $o_1 \downarrow o_2$ (meet) and $o_1 \uparrow o_2$ (join), respectively:

$$\begin{aligned} o[l_1 = t_1, \cdots, m_1 = u_1, \cdots] \downarrow p[m_1 = v_1, \cdots, n_1 = w_1, \cdots] \; &= \\ q[l_1 = t_1, \cdots, m_1 = u_1 \downarrow v_1, \cdots, n_1 = w_1, \cdots]. & \\ o[l_1 = t_1, \cdots, m_1 = u_1, \cdots] \uparrow p[m_1 = v_1, \cdots, n_1 = w_1, \cdots,] \; &= \\ r[m_1 = u_1 \uparrow v_1, \cdots]. & \end{aligned}$$

where $q = o \downarrow p$ and $r = o \uparrow p$.

[3] From personal discussions with Keiji Hirata and Ko Sakai.

Set terms constitute another lattice. Given two set terms, S_1 and S_2, we can define meet and join operations ($\Downarrow$ and $\Uparrow$, respectively) under Hoare order as follows:

$$S_1 \Downarrow S_2 \stackrel{def}{=} \text{canonical form of } \{e_1 \downarrow e_2 |\ e_1 \in S_1, e_2 \in S_2\}$$
$$S_1 \Uparrow S_2 \stackrel{def}{=} \text{canonical form of } S_1 \cup S_2$$

where $\{\top\}$ is the top of the lattice and $\{\ \}$ is the bottom.

Example 7 Meet and Join

When $male \sqsubseteq person$, $1 \sqsubseteq int$, and so on are defined, the following meet and join operations hold.

$person[age = 30] \downarrow male[occupation = pianist] =$
$\quad male[age = 30, occupation = pianist],$
$person[age = 30] \uparrow male[occupation = pianist] =$
$\quad person$
$\{1,2,3\} \Downarrow \{2,3,4\} = \{2,3\}$
$\{6,7,8\} \Downarrow \{int\} = \{6,7,8\}$
$\{1,2,3\} \Uparrow \{2,3,4\} = \{1,2,3,4\}$
$\{6,7,8\} \Uparrow \{int\} = \{int\}$

□

5.3.3 Subsumption Constraint and Attribute Term

Objects (object terms) in Quixote can have various attributes. The attribute specification is represented in the form of attribute terms or subsumption constraints.

Attribute term

Attribute terms define a way of attribute specification of object terms.

[Def] 27 (Attribute Term) *An* attribute term *is a term of the following form*

$$o/[l_1\ op_1\ t_1, \cdots, l_n\ op_n\ t_n] \quad (0 \leq n).$$

Here, let $o \in B$, $l_1, \cdots, l_n \in L_i \cup L_s$ *where* $l_i \neq l_j$ $(i \neq j)$, *and* $t_1, \cdots, t_n$ *be object terms or variables ,* $op_i \in \{\rightarrow, \leftarrow, =\}$*(for single value)* $\cup \{\rightarrow_H, \leftarrow_H, =_H\}$*(for set value).* □

The attribute-value pairs specified in the right side of / are called *extrinsic properties.* Do not confuse extrinsic properties with intrinsic properties which are attribute-value pairs within complex object terms (See 5.3.1).

For an attribute term of a complex object term, intrinsic properties are also extrinsic properties. That is,

$$o[\cdots, l{=}t_1, \cdots]/[p{\rightarrow}t_2]$$

is equivalent to

$$o[\cdots, l{=}t_1, \cdots]/[l{=}t_1, p{\rightarrow}t_2].$$

When there is a same label for intrinsic and extrinsic properties, intrinsic properties override extrinsic properties. That is, below attribute term

$$o[\cdots, l{=}t_1, \cdots]/[l{\rightarrow}t_2]$$

is equivalent to

$$o[\cdots, l{=}t_1, \cdots]/[l{=}t_1].$$

An attribute term is equivalent to an object term with subsumption constraints. The transformation is explained in 5.3.3.

Example 8 The first line represents an object *mozart* which has two extrinsic properties about *birth* and *dead*. The second line shows a complex object whose *type* is a kind of *symphony* and *no* is 9. In the latter case, *composer* is an intrinsic attribute.

$$mozart/[birth = 1756, dead = 1791]$$
$$op125[composer = beethoven]/[type \rightarrow symphony, no = 9]$$

□

Dotted term

[Def] 28 (Dotted Term) *A dotted term is a term of the form o.l or o.l* which specifies the value of a label of an object.*

The value of a single value label l of an object term o is represented as o.l, and the value of a set value label l is o.l*.* □

For example, a dotted term *mozart.first_name* corresponds to "Mozart's first name."

Subsumption constraint

The knowledge such as "the first name of Mozart is Amadeus", can be represented as a constraint between a dotted term (such as $mozart.first\_name$) and an object term (*amadeus*). The constraint is a *subsumption constraint* defined as follows.

[Def] 29 (Subsumption Constraint) *Let t_1, t_2 be object terms, single value variables, or dotted terms with single value labels, then $t_1 \sqsubseteq t_2$ is a* subsumption constraint.

In the case of a set, if t_1^* and t_2^* are set terms, set value variables, or dotted terms with set value labels, then $t_1^* \sqsubseteq_H t_2^*$ is also a (set) subsumption constraint. □

For example, "the first name of Mozart is Amadeus" is a subsumption constraint $mozart.first\_name \cong amadeus$, which is equivalent to

$$mozart.first\_name \sqsubseteq amadeus \wedge mozart.first\_name \sqsupseteq amadeus.$$

The semantics of Quixote is outlined in the following three parts (see [YY90] for details). :

(1) An object term is mapped into a *labeled graph* as a subclass of a hyperset[Acz88].
(2) The subsumption relation among object terms corresponds to a *bisimulation relation* among labeled graphs.
(3) A label or an object term used as a label corresponds to a *function* on a set of labeled graphs. Here the subsumption relation among labels is not considered.

Solving subsumption constraints

Solving a set of subsumption constraints is to apply the following rules until the set is saturated. Constraints of the right hand side are added from those of the left hand side.

$$\begin{array}{rcl}
x \sqsupseteq y & \Rightarrow & y \sqsubseteq x \\
x \sqsubseteq y,\ y \sqsubseteq z & \Rightarrow & x \sqsubseteq z \\
x \sqsubseteq y,\ x \sqsubseteq z & \Rightarrow & x \sqsubseteq (y \downarrow z) \\
y \sqsubseteq x,\ z \sqsubseteq x & \Rightarrow & (y \uparrow z) \sqsubseteq x \\
x \cong y,\ y \cong z & \Rightarrow & x \cong z \\
x \sqsubseteq y,\ y \sqsubseteq x & \Rightarrow & x \cong y \\
o[\cdots] \sqsubseteq o' & \Rightarrow & o \sqsubseteq o' \\
o[\cdots, l{=}x, \cdots] \sqsubseteq o'[\cdots, l{=}y, \cdots] & \Rightarrow & o \sqsubseteq o', x \sqsubseteq y \\
o[\cdots, l{=}x, \cdots] \cong o'[\cdots, l{=}y, \cdots] & \Rightarrow & o \cong o', x \cong y
\end{array}$$

where $x \sqsubseteq x$ and $x \cong x$ are removed in the procedure. When $a \cong b$ occurs for different basic objects a and b in the process, the constraint solving fails. The termination and confluency of the above rules are proved in [Muk90]. Similar rules are also defined for set constraints. See Section 5.4.1 for detail.

Example 9 Subsumption constraint solving
When $c \sqsubseteq a$ and $c \sqsubseteq b$ are defined, subsumption constraints $\{x \sqsubseteq a, x \sqsubseteq b, y \sqsubseteq x, c \sqsubseteq y\}$ derive $\{x \cong c, y \cong c\}$. □

Attribute term and subsumption constraint

The following rules transform an attribute term into an object term with subsumption constraints. $o|C$ represents an object term with constraints where o is an object term and C is a set of subsumption constraints.

$$
\begin{aligned}
o/[l \to t]|C &\iff o|\{o.l \sqsubseteq t\} \cup C \\
o/[l \leftarrow t]|C &\iff o|\{o.l \sqsupseteq t\} \cup C \\
o/[l{=}t]|C &\iff o|\{o.l \cong t\} \cup C \\
o/[l^* \to_H s]|C &\iff o|\{o.l^* \sqsubseteq_H s\} \cup C \\
o/[l^* \leftarrow_H s]|C &\iff o|\{o.l^* \sqsupseteq_H s\} \cup C \\
o/[l^* {=}_H s]|C &\iff o|\{o.l^* \cong_H s\} \cup C
\end{aligned}
$$

Example 10 Attribute Term and Constraints $mozart/[birth = 1756, dead = 1791]$ is equivalent to

$$mozart|\{mozart.birth = 1756, mozart.dead = 1791\}.$$

$$op125[composer = beethoven]/[type \to symphony, no = 9]$$

is equivalent to

$$
\begin{aligned}
&op125[composer = beethoven]| \\
&\{op125[composer = beethoven].type \sqsubseteq symphony, \\
&\quad op125[composer = beethoven].no = 9\}
\end{aligned}
$$

□

Property inheritance

Here we comes the property inheritance mechanism of Quixote. Extrinsic properties of an object inherit from another object in a subsumption relation.

[Def] 30 (Property Inheritance) *If $o \sqsubseteq p$ and o does not have an intrinsic property of a label l (l^*), then $o.l \sqsubseteq p.l$ ($o.l^* \sqsubseteq_H p.l^*$).* □

If $o \sqsubseteq p$, $o.l = a$, and $p.l$ is not defined then $a \sqsubseteq p.l$. It is called an *upward inheritance*. On the other hand, if $o \sqsubseteq p$, $p.l = b$, and $o.l$ is not defined then $o.l \sqsubseteq b$. The property inheritance is called a *downward inheritance*.

Properties of labels in intrinsic properties do not inherit. Using this feature, we can represent an exception of the property inheritance.

Example 11 Property Inheritance

(1) If $apple/[color \rightarrow red]$,
then $apple[weight = heavy]/[color \rightarrow red]$, but
$apple[color = green]$ does not inherit $color \rightarrow red$.

(2) If $apple[weight = heavy]/[area^* \leftarrow_H \{aomori\}]$
and $apple[color = green]/[area^* \leftarrow_H \{nagano\}]$,
then $apple/[area^* \leftarrow_H \{aomori, nagano\}]$
(by the join operation between sets).

□

Note that, in the cases of $\leftarrow$ and $\leftarrow_H$, extrinsic properties are inherited upward by the above rule, while intrinsic properties are not, even though $apple[color = green]$ is $apple[color = green]/[color = green]$.

Properties are equivalent to a set of constraints in Quixote. A *multiple inheritance*, thus, corresponds to merging constraints without a preference. For example, when $o \sqsubseteq p$, $o \sqsubseteq q$, $p/[l = a]$, and $q/[l = b]$ are defined, o inherits two constraints $o.l \sqsubseteq a$ and $o.l \sqsubseteq b$ from properties of p and q respectively. The constraints are merged to be $o.l \sqsubseteq a \downarrow b$.

5.3.4 Rule and Module

Module

Quixote objects are defined by rules that are modularized into several *modules*:

$$m : \{r_1, \cdots, r_n\},$$

where m is an object term called a *module identifier* (mid) , and $r_1, \cdots, r_n$ are rules defined later. For simplicity, we use the notation 'a module m' instead of 'a module with a mid m.'

When a mid contains a variable, it is called a *parametric module*. Variables in a mid are global in the module, that is, variables in a mid can be shared by rules in the module. A rule in a module can explicitly refer an object in another module.

Module mechanism is introduced with the following objectives:

- modularization and classification of knowledge,
- co-existence or localization of inconsistent knowledge,
- temporal storage of tentative knowledge, and
- introduction of a modular programming style.

Rule

A rule of Quixote specifies an existence of objects in a module, property specifications of objects, and a implication relation among objects.

[Def] 31 (Rule) *Let* a_0 *(*$=o_0|C_0$*),* a_1 *(*$=o_1|C_1$*),* $\cdots$*,* a_n *(*$=o_n|C_n$*) be attribute terms,* $m_0, m_1, \cdots, m_n$ *be mids, and* D *a set of subsumption constraints. A* rule *is defined as follows:*

$$m_0 :: a_0 \Leftarrow m_1 : a_1, \cdots, m_n : a_n \parallel D;;$$

a_0 *is called a* head *and* $m_1 : a_1, \cdots, m_n : a_n \parallel D$ *is called a* body. C_0 *(extrinsic properties of* a_0*) must not contain subsumption relations between basic object terms.* □

The rule in the above definition intuitively means that a module m_0 has a rule such that if a_1 is satisfied in a module m_1, $\cdots$, and a_n is satisfied in a module m_n under constraint D, then a_0 is satisfied in a module m_0.

The rule can be transformed into:

$$m_0 :: o_0|C_0 \Leftarrow m_1 : o_1, \cdots, m_n : o_n \parallel A \cup C;;$$

where C_0 is called a *head constraint* and $A \cup C = C_1 \cup \cdots \cup C_n \cup D$ is called a *body constraint*. Further, a body constraint can be divided into a set of constraints containing dotted terms (A) and a set of the rest of the constraints (C). The restriction of C_0 in the above definition is to avoid destruction of the lattice by assertion of a subsumption relation during derivation. A rule with empty body is called a *fact*.

As shown in 5.3.5, Quixote goals are not always processed from left to right. However, it is possible to specify the processing order from left to right. A Rule whose delimiter of goal is ";" instead of "," is called a *serialized rule*. The goal is processed from left to right like Prolog.

From an object-oriented point of view, a rule gives an *intensional* definition of Quixote objects. An object in Quixote consists of an object term and a set of methods. An object term without variables plays the role of an *object identifier* (oid)[AK89, MHY90], while each extrinsic property plays the role of a method. That is, a label corresponds to a message and the value corresponds to the result.

For the case in which there is no head constraint, Quixote can be considered as an instance of CLP(X) [JL87], where a constraint domain is a set of labeled graphs – as a subclass of hypersets[Acz88] and (extended) subsumption relations. Without set subsumption constraints, Quixote becomes a subclass of CLP(AFA), with a hyperset constraint domain [Muk90].

Submodule relation and Rule Inheritance

A module can inherit rules from another module. The relation between modules in terms of the *rule inheritance* is a *submodule relation*. When a module m_1 inherits rules in a module m_2, m_1 is called a *submodule* of m_2, and m_2 is called a *supermodule* of m_1. The relation is represented as $m_1 \sqsupseteq_S m_2$.

Note that, submodule relations are different from subsumption relations defined in 5.3.2 although they are both relations between object terms. The submodule relation specifies *rule inheritance*, while the subsumption relation specifies property inheritance.

Each rule can have an inheritance flag o, l, or ol to control the rule inheritance between modules. A rule with o *overrides* inherited rules from the supermodule that have the same head as shown in Example 12. A rule with l is a *local* rule, which is not inherited by the submodules. A rule with ol is inherited as a combination of o and l.

Example 12 Rule Inheritance Consider representing "In Europe, cars usually drive on the right. But cars drive on the left in England."

$$england \sqsupseteq_S europe;; france \sqsupseteq_S europe;;$$
$$europe :: car/[drive = right];;$$
$$england :: (o)car/[drive = left];;$$

As the last rule has an inheritance flag o, a rule with the same head of the supermodule (the second line) is overridden. □

Storing inconsistent information into several modules

In the current framework of Quixote, names of object terms, subsumption relations, and submodule relations are global in a database, while the existence of objects and extrinsic properties are local. That is, if there is no submodule relation between two modules, their extrinsic properties do not contradict, that is, inconsistent knowledge can co-exist separately in independent modules.

For example, the following program becomes inconsistent because *john* has a different extrinsic *age* property in the module *year_1994*.

$$year\_1994 :: john/[age = 20];;$$
$$year\_1994 :: john/[age = 30];;$$

However, the following is not inconsistent, when there is no submodule relation between *year_1982* and *year_1994*.

$$year\_1982 :: john/[age = 20];;$$
$$year\_1994 :: john/[age = 30];;$$

5.3.5 Query processing

Program

A *program* is defined as a triple (S, M, R), where S, M, R correspond to definitions of subsumption relations [4], submodule relations, and rules. Definitions of rules can be considered as definitions of objects or definitions of contents of modules.

In reality, a database tends to be partial, that is, some definitions might be missing, rules might be ambiguous or indefinite, and a Quixote object might be incompletely defined[YNT+94]. To treat the partiality of information, Quixote has features such as hypothetical reasoning and answers with assumption.

Query and answer

A query is defined as follows.

[Def] 32 (Query and Answer) *Let* $m_0, \cdots, m_n$ *be mids,* $a_0, \cdots, a_n$ *attribute terms, and* C *a set of subsumption constraints,* H *a set of additional rules called* hypotheses. *A query is defined as follows:*

$$? - m_0 : a_0, \cdots, m_n : a_n \parallel C\ [\ ;;\ H\ ].$$

An answer is in the form of

if [Assumption] *then* [Result] *because* [Explanation]

where Assumptions *corresponds to information that does not included in the program,* Result *is a set of subsumption constraints, and* Explanation *shows what knowledge is used to derive the answer.* □

Answer with Assumption

Example 13 Answer with Assumption

Consider a piece of knowledge about classical music:

"C major is a kind of major"
"K551 is a symphony named Jupiter"
"K467 is a piano concert in C major", and
"Major key pieces are preferable when gloomy."

[4] Only the $\preceq$-relation is defined in S.

The knowledge is encoded in the following Quixote program.

$major \sqsupseteq c\_major;;$
$music :: k551/[type = symphony, name = jupiter];;$
$music :: k467/[type = piano\_concert, key = c\_major];;$
$m :: listen[mood = gloom, music = X] \Leftarrow$
$\quad music : X/[key \rightarrow major];;$
(When gloomy, a piece with a major key is preferable.)

For a query such as:

$$? - listen[mood = gloom, music = k467],$$

the answer is simply **yes**.

How about the answer to the following question:

$$? - listening : listen[mood = gloom, music = k551]$$

which asks whether K551 is preferable? Although there is an object $k551$ in *music* module, it does not specify *key* (extrinsic) property. Without making any assumptions, the query fails. However, as we focus on the partiality of the information, the lack of information suggests an assumption to be taken. Hence, the answer is

$$IF\ music : k551.key \sqsubseteq major\ THEN\ yes$$

where unsatisfied constraints of other objects' extrinsic properties in bodies are assumed. □

In logic programming, finding a lack of information or unsatisfiable subgoals corresponds to *abduction*, that is, hypothesis or explanation generation [CM85, HSME88]. Remember that a rule in Quixote can be represented as follows (see Subsection 5.3.4):

$$m_0 :: o_0|C_0 \Leftarrow m_1 : o_1, \cdots, m_n : o_n \mid\mid A \cup C;;$$

In Quixote, only dotted term constraints can become assumptions, that is, when body constraints about dotted terms are not satisfied, they are taken as a conditional part of an answer. Although A and C are disjoint, when variables in C are bound by dotted terms during query processing, constraints with the variables in C are moved into A. If the subsumption relation between object terms is taken as assumption, it might destroy the soundness of the derivation because it affects property inheritance and does not guarantee results in the former derivation.

Derivation

An abduction is closely related to procedural semantics. Here we will only briefly explain the relation [NOTY93].

In general, a derivation by query processing in Prolog is a finite sequence of a pair (G, θ) where G is goals and θ is a substitution[Llo84]:

$$(G_0, \theta_0) \Rightarrow (G_1, \theta_1) \Rightarrow \cdots \Rightarrow (G_{n-1}, \theta_{n-1}) \Rightarrow (\emptyset, \theta_n).$$

In CLP, substitutions of Prolog are extended as constraints (See discussions of cu-Prolog in Subsection 3.4.2). Thus derivation by query processing in CLP is the finite sequence of a pair (G, C) of a set G of goals and a set C of constraints:

$$(G_0, C_0) \Rightarrow (G_1, C_1) \Rightarrow \cdots \Rightarrow (G_{n-1}, C_{n-1}) \Rightarrow (\emptyset, C_n).$$

On the other hand, derivation in Quixote is a finite directed acyclic graph of the triple (G, A, C) of the set G of goals, the set A of assumptions, and the set C of constraints.

A query is also transformed in the form of

$$?\text{-}o_1, \cdots, o_n \parallel A_0 \cup C_0,$$

It is a triple $(\{o_1, \cdots, o_n\}, A_0, C_0)$. For a node $(\{G\} \cup G_i, A_i, C_i)$, a rule $G'|C' \Leftarrow B \parallel A \cup C$, and $\exists\theta\ G\theta = G'\theta$, where B is a set of object terms and θ is a substitution, the transformed node is:

$$((G_i \cup B)\theta,\ (A_i\theta \setminus C'\theta) \cup A\theta,\ (C_i \cup C \cup C')\theta).^5$$

The derivation image is illustrated in the network in Figure 5.1.

If there are two nodes, (G, A, C) and (G, A', C), where $A \subseteq A'$, then the derivation path of (G, A', C) is thrown away. That is, only the minimal assumption is made. If there are two nodes, (G, A, C) and (G, A, C'), then they are merged into $(G, A, C \cup C')$.

5.4 Implementation

This section explains implementation issues of Quixote, especially about the constraint solver to which the author mainly contributes. There are two kinds of Quixote implementation; big-Quixote that is a full client-server style implementation in KLIC (KL1) and micro-Quixote that is small restricted implementation in C. Both system are registered as ICOT Free Software (IFS) [6].

[6]ICOT free software is available from `http://www.icot.or.jp`

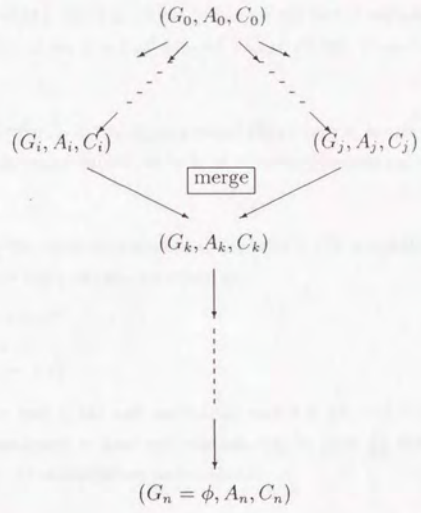


Figure 5.1: Derivation Network of Quixote

5.4.1 Implementation of Constraint Solving

Element_of and Disequation constraints

In big-Quixote, the domain of constraint is extended to treat element-of and disequation constraints. For actual applications, there occurs constraints like "a variable can be bound to Mozart, Beethoven, or Bach" or "a variable does not have the same value of another variable." To treat such disjunctive information, Quixote can process element_of and disequation constraints.

[Def] 33 (Element-of Constraint) *Let t be object terms, single value variables, or dotted terms with single value labels, s be a set of ground object terms, then $t \in s$ is an* element_of constraint. □

[Def] 34 (Disequation Constraint) *Let t_1, t_2 be ground object terms, single value variables, or dotted terms with single value labels, then $t_1 \neq t_2$ is a* disequation constraint. □

In addition to rewriting rules for subsumption constraints in 5.3.3, the following rules are added to treat element_of and disequation constraints.

$$\begin{aligned} x \in s1,\ x \in s2 &\Rightarrow x \in s1 \cap s2 \\ x \in \{a\} &\Rightarrow x \cong a \\ x \neq a,\ x \in s &\Rightarrow x \in s - \{a\} \end{aligned}$$

Constraints such as $x \neq y$ (x and y are not unifiable) and $o \in \{o, \cdots\}$ are removed. When $x \neq x$, $o \in s$ (o is ground and s does not contain o), or $x \in \{\}$ occurs in the constraint rewriting process, the constraints are unsatisfiable.

Constraint solving procedure

Let

- C be a set of constraint,
- A, B be object terms,
- A_g, B_g be ground object terms,
- T, U be single-value variables or object terms,
- x, y be single-value variables,

- x^*, y^* be set-value variables,
- S be a set-value variable or a set, and
- s be a set.

The constraint solving procedure of big-Quixote is to repeat the following rewriting rules to C until C is saturated.

$$
\begin{array}{rcl}
\{T \sqsubseteq T\} \cap C & \Rightarrow & C \\
\{x \sqsubseteq T, T \sqsubseteq x\} \cap C & \Rightarrow & \{x \cong T\} \cap C \\
\{U \sqsubseteq y, y \sqsubseteq T\} \cap C & \Rightarrow & \{U \sqsubseteq y, y \sqsubseteq T, U \sqsubseteq T\} \cap C \\
\{x \sqsubseteq A_g, x \sqsubseteq B_g\} \cap C & \Rightarrow & \{x \sqsubseteq A_g \downarrow B_g\} \cap C \\
\{A_g \sqsubseteq x, B_g \sqsubseteq x\} \cap C & \Rightarrow & \{A_g \uparrow B_g \sqsubseteq x\} \cap C \\
\{o[\cdots] \sqsubseteq p\} \cap C & \Rightarrow & \{o \sqsubseteq p\} \cap C \\
\{o[\cdots l_i = T_i, \cdots] \sqsubseteq p[\cdots, l_i = U_i, \cdots]\} \cap C & \Rightarrow & \{o \sqsubseteq p, \forall l_i \in p[\cdots], T_i \sqsubseteq U_i\} \cap C \\
\{T \cong T\} \cap C & \Rightarrow & C \\
\{A \cong x\} \cap C & \Rightarrow & \{x \cong A\} \cap C \\
\{x \cong T\} \cap C(\ni x) & \Rightarrow & \{x \cong T\} \cap C[x/T] \\
\{o[l_1 = T1, \cdots, l_n = Tn] \cong & & \\
p[l_1 = U1, \cdots, l_n = U_n]\} \cap C & \Rightarrow & \{o \cong p, \forall i \in [1,n], T_i \cong U_i\} \cap C \\
\{x^* \sqsubseteq_H x^*\} \cap C & \Rightarrow & C \\
\{x^* \sqsubseteq_H S, S \sqsubseteq_H x^*\} \cap C & \Rightarrow & \{x^* \cong_H S\} \cap C \\
\{S1 \sqsubseteq_H y^*, y^* \sqsubseteq_H S2\} \cap C & \Rightarrow & \{S1 \sqsubseteq_H y^*, y^* \sqsubseteq_H S2, S1 \sqsubseteq_H S2\} \cap C \\
\{x^* \sqsubseteq_H s1, x \sqsubseteq_H s2\} \cap C & \Rightarrow & \{x^* \sqsubseteq_H s1 \Downarrow s2\} \cap C \\
\{s1 \sqsubseteq_H x^*, s2 \sqsubseteq_H x^*\} \cap C & \Rightarrow & \{s1 \Uparrow s2 \sqsubseteq_H x^*\} \cap C \\
\{s \cong_H s\} \cap C & \Rightarrow & C \\
\{x^* \cong_H s\} \cap C & \Rightarrow & C[x^*/s] \\
\{x \in s1, x \in s2\} \cap C & \Rightarrow & \{x \in s1 \cup s2\} \cap C \\
\{x \in \{A\}\} \cap C & \Rightarrow & \{x \cong A\} \cap C \\
\{t1 \ncong t2, t1[x/T] = t2[x/T]\} \cap C & \Rightarrow & \{x \ncong T\} \cap C \\
\{t1 \ncong t2, \neg unifiable(t1, t2)\} \cap C & \Rightarrow & C \\
\{x \ncong A, x \in s1\} \cap C & \Rightarrow & \{x \in s1 - \{A\}\} \cap C
\end{array}
$$

Here, $T[x/t]$ stands for replacing all the occurrence of x in T with t.

When one of the following constraints occurs during the saturation process, the constraint solver fails.

1. $o \cong p$ where o and p are literally different atoms.
2. $s1 \cong s2$ where $s1 \cup s2 \neq s1 \cap s2$
3. $T \ncong T$
4. $x \in \phi$

5.4.2 Big-Quixote

Big-Quixote is a full implementation of all the features of Quixote language described in Section 5.3. It consists of Quixote-client and Quixote-server modules as shown in Figure 5.2.

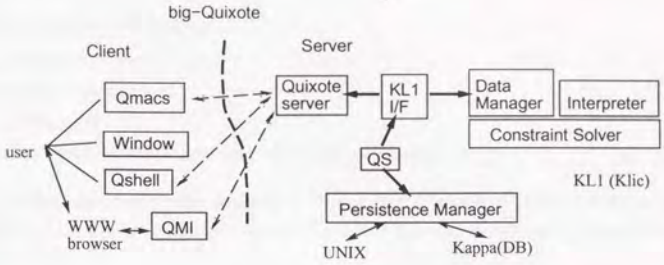


Figure 5.2: System configuration of big-Quixote

Quixote-server consists of following modules. They are mainly implemented in the KL1 language on PIMOS and KLIC systems. KL1 is a parallel logic programming language developed at ICOT. KLIC is a system to translate programs in KL1 into programs in C language, which is also developed at ICOT .

1. Quixote server: TCP/IP communication interface.
2. KL1 IF: data transformation, etc.
3. QS: manages external DBs.
4. Persistence Manager: interface to external DBs.
5. Data Manager: manages internal representation of Quixote objects.
6. Interpreter: makes inference.
7. Constraint Solver: solves subsumption, set, and disequation constraints.

In the latest version of big-Quixote (ver.4), there are three kinds of client interface: Qmacs, Qshell, and X-Windows interface with a WWW browser such as Netscape.

1. Qmacs: interactive user interface on top of GNU-Emacs.
2. Qshell: batch user interface with QIF libraries.
3. QMI: CGI interface to Quixote from WWW browser.
4. window: window interface to display lattice structures, module hierarchies, and derivation trees.

5.4.3 micro-Quixote

Full implementation of rich Quixote features tends to be too heavy for small workstations and PCs. Micro-Quixote is designed to extract central features of Quixote as a programming language and offers a small system for knowledge information processing[NTY94]. Micro-Quixote supports the following feature of Quixote.

- object terms (without set),
- subsumption constraints,
- property inheritance,
- module, and
- answer with assumptions, hypothetical reasoning.

For simplicity, micro-Quixote utilizes a Prolog-like depth-first search without merging derivations. Consequently, some results of micro-Quixote are different from those of big-Quixote.

Micro-Quixote is implemented in the C language independent of big-Quixote and has the following features.

- Everything is implemented in C and has high portability,
- small system size (199KB of source code), and
- has an external call mechanism.

System configuration of micro-Quixote is shown in Figure 5.3.

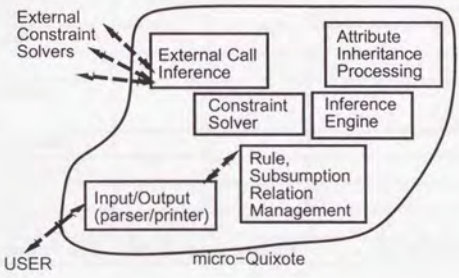


Figure 5.3: System configuration of micro-Quixote

A unique feature of micro-Quixote is an external call mechanism. Micro-Quixote allows representing external constraints which are solved by an external constraint solver. An operation of external constraints begins and ends with "#", such as

```
?-X/[name=A] ||
    {X=<male, A #regexp# ''on''}.
```

(search a man whose name contains "on.")

When an external constraint binds to be ground, micro-Quixote throws it out and waits for the result (true or false). The trigger is like the bind-hook explained in Section 3.2. This is the external call mechanism. In micro-Quixote, external call messages are dispatched from GNU-Emacs to the X-Windows interface or bc (binary calculator), and so on. The external call mechanism plays an important role when micro-Quixote is embedded into a heterogeneous cooperative problem solving system like Helios[AYT94], which will be discussed in Section 7.5.

Table 5.1 summarizes the difference between big-Quixote and micro-Quixote in terms of various features.

Table 5.1: Comparison between big-Quixote and micro-Quixote

	big-Quixote	micro-Quixote
OS/machine Environment	UNIX, PIMOS	UNIX, MS-DOS, Macintosh
Development Language	KL1 (KLIC), C Emacs-Lisp, etc.	C
Code Size	6Mbyte	199Kbyte
Method for Derivation	OLDT	SLD with Prolog-like search strategy
Subsumption Constraint	√	√
Property Inheritance	√	√
Module	√	×
Answer with Hypothesis Conditional Query	√	√
Set, Inequality Constraint	√	×
NAF (Negation as Failure)	√	×
Solution Composition	√	×
Database Functionality	√	×
External Problem Solver Call	Arithmetic module	External Constraint Call

Chapter 6

Applications of Quixote to Natural Language Analysis

6.1 Introduction

Several features of Quixote introduced in the previous chapter make it applicable to various domains: for example, legal reasoning, genetic information processing, and natural language processing.

Of these, this chapter focuses on two natural language applications of Quixote: *typed feature structure* [TTY+93, TTY+94] and a semantic representation in constraint-based grammar[TH96]. As the latter example, semantic representation of a phrase of the form "A *no* B" in Japanese is discussed. The phrase is a famous example that various interpretations are possible according to the situation the phrase is uttered. The latter example is a cooperative work with Yasunari Harada.

6.2 Attribute Term and Feature Structure

This subsection mentions the relation between an attribute term of Quixote and a feature structure.[TTY+93, TTY+94]

A feature structure is a partial function from features to their values and represented with a set of feature-value pairs as introduced in Section 2.1. For example, an AVM form (6.1) shows a function mapping the feature `number` onto the value *singular* and `person` *third* [Shi86].

Similar to a feature structure, an attribute term can describe information partially.

For example, a feature structure (6.1) corresponds to an attribute term (6.2).

$$\begin{bmatrix} \texttt{number} : \mathit{singular} \\ \texttt{person} : \mathit{third} \end{bmatrix} \tag{6.1}$$

$$X/[number = singular, person = third] \tag{6.2}$$

Here, X stands for the feature structure itself. The unification between feature structures is performed by unifying head object terms followed by related dotted term constraint solving. (6.2) is equivalent to the following dotted term constraints.

$X.number \cong singular$
$X.person \cong third.$

Next, consider a typed feature structure[Car92b], where the latest framework of HPSG is constructed upon[PS94]. It is the feature structure whose nodes are labeled with sort (type) symbols as introduced in Section 2.1. Using inheritance among supersorts and subsorts of typed feature structures, efficient representation of lexicon[DSG92] and grammar is possible.

For example, (6.3) is a simple HPSG-like typed feature structure representing the word "run." **word**, **vp**, **run**, or **np** specifies the sort of each structure.

$$\begin{bmatrix} \textbf{word} \\ \texttt{CAT} : [\textbf{vp}] \\ \texttt{PH} : [\textbf{run}] \\ \texttt{SUBCAT} : \begin{bmatrix} \textbf{phrase} \\ \texttt{CAT} : [\textbf{np}] \end{bmatrix} \end{bmatrix} \tag{6.3}$$

The basic object term and a subsumption relation (partial relation) $< Bobj, \sqsubseteq >$ in Quixote naturally comprises types and their inheritance hierarchy. To treat the inheritance and information partiality, it is natural to describe a typed feature structure with an attribute term whose head is subsumed by a basic object term. Property inheritance mechanism corresponds to the inheritance between typed feature structures. (6.3) is represented as the following subsumption relations and two attribute terms.

```
Y=<word, Z=<phrase
Y/[cat->vp,ph->run,subcat=Z]
Z/[cat->np]
```

In comparison with related KR languages, PST in CIL[MY85] and ψ-terms in LOGIN[AKN86] have close relation with feature structures. However, attribute terms in Quixote are more powerful because CIL does not have a property inheritance feature and LOGIN cannot handle constraints.

6.3 JPSG treatment of "A no B" in Quixote

This section discusses a semantic and pragmatic representation of a Japanese phrase of the form "A no B" in Quixote.

6.3.1 Introduction to the variety of "A no B" phrase

Japanese adnominal particle *no* combines two noun phrases A and B to form a compound noun phrase of the form "A *no* B". The kinds of relations that obtain between the referents of A and B are quite varied, which is somewhat reminiscent of the case of English compound nominal. Thus, the first linguistic problem concerning the adnominal particle *no* is how to handle its semantics. The "meaning" of *no* is "underspeicified" and "situation dependent." Also, there is a very interesting pragmatic problem to be solved. In certain cases, the noun phrase A has to have a complex internal structure in order for the entire NP to be meaningful.

Toward the end of this section, we propose how inference process resolves referents of noun phrase expressions within contexts. Various aspects of Quixote offer a natural explanation of the difference of acceptability among expressions of the form "A *no* B."

Semantic aspect of *no*

Let us briefly review the kinds of relations that can hold between the referents of the noun phrases combined by the adnominal particle *no*, taking examples that will be discussed later.

(6) a. 1987 nen no "Information-based Syntax and Semantics"
(the book) "Information-based Syntax and Semantics" published in (the year) 1987

b. 20 sai no otoko
a/the man aged 20

c. nagai kami no zyosei
a/the woman with long hair

d. bin no kuti
the mouth/opening of the/a bottle

e. otoko no nenrei
the age of the man

f. 1992 nen 8 gatu no COLING
(the conference of) COLING in August of 1992

g. honyurui no ningen
human beings, which is a kind of mammals
h. gakki no piano
the piano, which is a kind of musical instrument

These examples and their translations would suffice to indicate that the "meaning" of *no* is "underspecified" and "context-dependent." Intuitively speaking, what *no* does is to combine two noun phrases, regardless of the meaning, and the rest is up to the context, for the most part. We would not want to say that *no* is 25-way ambiguous or 100-way ambiguous. On the other hand, we will give a small representation model of a semantic framework of "A no B" and the processing mechanism how the ambiguity is generated according to the situation using Quixote.

However, traditional approaches to the semantics of *no* was either to enumerate the kinds of relations *no* can "mean" or to disregard the meaning entirely and let the examples take over. For example, as will shown in Subsection 6.3.2, [SNN86] categorized these semantic relations into 5 groups, with 86 finer sub-categories and gave their semantic representations in the form of Prolog terms. This approach, however, has the following problems.

- Anything outside this categorization has to be added.
- The representations are too artificial and complex because they are represented as combination of many Prolog terms.
- There is no way metaphorical expressions can be handled in a simple way.
- A given expression has to have only one meaning.
- It cannot treat situation dependencies.

Example based translation offers a good performance, especially with this kind of phenomena. However, interpretation of noun phrases with *no* is in fact context and situation dependent, and is not lexically driven. In most cases, the relation that holds between the referents of the two noun phrases combined by the adnominal particle *no* can be determined only after the referents of these two noun phrases are determined. Since statistical approach works only by chance, there is no way to remedy the system when it fails. Also, there is no way in which statistical approaches can begin to explain the kind of pragmatic constraints mentioned below.

On the other hand, we conceive of the semantics of *no* as something very much like anaphora. Semantic representations of noun phrases will correspond to object terms and dotted terms in Quixote. Also, modules in Quixote is employed to express situation of

utterance and context for interpretation. Referents of noun phrases are determined against the background of those situations. Given the description of the context of utterance and designation of object referred to by the noun phrases, the relation between the two noun phrases combined can be settled in a fairly straightforward manner. Most of the cases fall into only 2 different ways of interpreting the phrase "A *no* B," although we are more than well aware that these does not exhaust the use of *no*.

Pragmatic aspect of *no*

Although *no* can combine almost any two noun phrases to form a noun phrase that can be interpreted in one way or another, there is an interesting pragmatic constraint on what kinds of combinations form more or less natural expressions. Take the following example, for instance. Without special contextualization, the (7b-b) sentence is much more difficult to interpret than (7b-a). Note, however, that it is not the case that it is completely impossible to imagine a situation where such expressions can be meaningful. For instance, if we are talking about people working in a beauty parlor, it may refer to the person in charge of doing the hair. [1]

(7) a. nagai kami no zyosei
b. (*) kami nó zyosei

Compare this with the following English case. [2]

The examples above, along with other similar cases, suggests that the noun phrase preceding *no* must be "complex" when it refers to something that is either a part of, inalieably possessed by, or a property of the referent of the noun phrase following *no*.

(9) a. hutoi ude-no dansei
b. (*) ude-no dansei

[1] This problem is discussed in [Har91].

[2] The whole distribution seems to suggest something similar to the English cases, although we cannot go further into this comparison here.

(8) a. long-haired girl
b. (*) haired girl

1. The murdered man had thrown a bomb into the Police Station.
2. * The killed man had thrown a bomb into the Police Station.

(a) cut diamond
(b) cut glass
(c) * cut bread
(d) badly cut bread
(e) sliced bread

(10) a. otituita seikaku-no dansei
b. (*) seikaku-no gakusei

Note that this cannot be described simply by syntactic terms. Although examples like those above might seem to suggest that the noun phrase preceding *no* must itself be syntactically "complex" or modified by other prenominal element, such is not the case as can be seen from the examples below.

(11) a. geta-no otoko
b. hige-no otoko

Taking into consideration the examples above, we can observe that noun phrase of the form "A *no* B" is rather odd or meaningless if the referent of A is part or property of the referent of B, whereas if the relation involved is optional part-whole relation or something other than object-property, the whole noun phrase is much easier to make sense of.

(12) a. otoko-no geta
b. geta-no otoko
c. otoko-no asi
d. * asi-no otoko

(13) a. kuruma-no sanruuhu
b. sanruuhu-no kuruma
c. kuruma-no taiya
d. * taiya-no kuruma

In [Har91], Harada pointed out this interesting phenomenon, and asked whether this should be considered a syntactic, semantic or pragmatic issue. In this section, we will argue that this is in fact a pragmatic matter, and show how this falls out from our treatment of the semantics of *no* within Quixote.

6.3.2 An analysis of *no* in Quixote

In this subsection, we classify semantic contents of "A *no* B" phrases and give their representations in Quixote, with objects, constraints, and modules. First, in 6.3.2, we show limited interpretation of noun phrases represented in Quixote. Second in 6.3.2, we classify the reading of "A *no* B" from an object-oriented, constraint-based, and situation-dependent view. 6.3.2 combines semantic aspects with syntactic treatment of *no* in JPSG. 6.3.2 discusses the recognition of "A *no* B" and 6.3.3 compares our approach with another one.

Limited semantic interpretation of nouns in Quixote

Before discussing about "A *no* B", consider how the interpretations of following nouns are represented in Quixote.

(a) proper noun,

(b-1) non-relational common noun, and

(b-2) relational common noun

First, (a) proper nouns designate some objects in a situation. So they are represented as object terms in some modules. For example, interpretations of *Ken* (a person's name) and *"Information Based Semantics"* (a book name) at CSLI in July 23, 1995 are

$csli\_1995\_7\_23 : ken$

$csli\_1995\_7\_23 : book[title = information\_based\_semantics]$

Secondly, consider common nouns such as *tukue* (desk), *kami* (hair) , *hahaoya* (mother) , *mae* (front) , and so on. Common nouns give semantic types of objects. According to the kinds of constraint between semantic types and objects that the nouns refer to, they are classified into non-relational nouns and relational nouns.

tukue, *kami*, and so on are classified as (b-1) non-relational common nouns. The constraint between objects and semantic types that the nouns refer to, is the subsumption (is_a or a_kind_of) constraint. *tukue* specifies an object X that is subsumed by *desk* that is the semantic type of *tukue*. Its representation in Quixote is:

$sit : X \mid X \sqsubseteq desk$

where *sit* is a situation.

hahaoya, *mae*, and so on are classified as (b-2) relational nouns. They have hidden arguments in their semantic type representation. The constraint between semantic types and objects depends on their utterance situations. *mae* specifies an object $mae[base = X]$ that is defined by the following rule in a situation *sit*.

$sit : mae[base = X] \Leftarrow X;;$

The rule intuitively means that if an object X is defined in a module *sit*, then the object $mae[base = X]$ exists in the module *sit*, which gives definition of *"(X no) mae"*.

In any cases, interpretations of common nouns are given as object identifiers (variables) in a module, equipped with constraints or rule definitions.

Note that some nouns such as *sensei*, *hahaoya*, and so on work as both non-relation and relational nouns, which give rise to some ambiguities in "A *no* B". Consider "*sensei no tukue*" (a kind of desk for teachers, or a desk of a teacher).

Classification of "A *no* B" and their representation in Quixote

In this subsection, we give a categorization of "A *no* B" and their representation in Quixote. Seen from objects, constraints, and situations, interpretations of "A *no* B" are classified into the following three cases:

(R1) B is one of the attributes of object A,

(R2) A is the value of one of the attributes of object B, and

(R3) A and B are equivalent in a super situation between S_A and S_B. (S_X is a situation where X exists.)

(R1)

Examples of (R1) are *otoko no nenrei* (age of a/the man), *bin no kuti* (opening of a/the bottle), and so on. The reading is possible when A specifies an object and B is an attribute of A. Let O_A be an object that A specifies, L_B be an attribute that B specifies, and M_A be a situation where O_A exists. The total semantic representation of reading (R1) of "A *no* B" is a dotted term in Quixote:

$$M_A : O_A.L_B \tag{6.4}$$

Consider *otoko no nenrei* for example. An interpretation of *otoko* is represented as $X \mid X \sqsubseteq man$ where *man* is a semantic type of *otoko*. Let *age* be an attribute specified by *nenrei*. *otoko no nenrei* falls into $X.age$ with the constraint $X \sqsubseteq man$.

(R2)

Examples of (R2) are *hatati no otoko* (a/the man whose age is 20), *taro no kekkon* (Taro's marriage), and so on. The reading is possible when B is an object and A is the value of one of the attributes of B. Let O_A and O_B be the objects specified by A and B, M_B be the situation where O_B resides. Then the total semantic representation of "A *no* B" is an attribute term in a module:

$$\exists l, M_B : O_B/[l = O_A]. \tag{6.5}$$

Consider *hatati no otoko* for example. An interpretation of *otoko* is $X \mid X \sqsubseteq man$. *hatati* is the value to specify one's *age*. So, *hatati no otoko* falls into the following semantic representation: $X/[age = 20]$, where $X \sqsubseteq man$.

(R3)

owan no fune (a/the bowl as a boat), and *hasi no kai* (a/the chopstick as an oar), and so on are examples of (R3). This is the reading where an object is referred to as different objects in different situations. In the object-oriented analysis such as with Quixote, using the same object identifier in the same module may lead to constraint contradiction or database inconsistency. For example, suppose X is also a *bowl* and a *boat* in a situation. Then two sets of constraints imposed on X: $X \sqsubseteq bowl$, $X \sqsubseteq boat$, and they are merged into $X == \bot$ (X is the bottom object).

To avoid the contradiction, such information must be stored in different modules. Consider three modules M_A, M_B, and M_{NO}, and each of them has an object with the same object identifier X. M_A is the module where X is referred as A, M_B is the module where X is referred as B, and M_{NO} is a supermodule of M_A and M_B where X is referred as "A *no* B". Then the semantic interpretation of "A *no* B" is the object in a module M_{NO}:

$$M_{NO} : X \ where \ M_{NO} \sqsubseteq_S \{M_A, M_B\} \tag{6.6}$$

Consider "*owan no fune*." Let o be an object identifier, $m\_owan, m\_fune, m\_owan\_fune$ be three modules where m_owan_fune is a super-module of the other two modules. The semantics interpretation of *owan no fune* represented as a set of rules in Quixote is: [3]

$m\_owan\_fune \sqsubseteq_S m\_owan;; m\_owan\_fune \sqsubseteq_S m\_fune;;$

$m\_owan :: o \mid o.kind \sqsubseteq bowl;;$

$m\_fune :: o \mid o.kind \sqsubseteq boat;;$

$m\_owan\_fune :: o;;$

[3]This description, actually, has a little problem because of several limitations of Quixote. It may be more natural to describe the example as the following rules, using variable X as an object.

$m\_owan\_fune \sqsubseteq_S m\_owan;; m\_owan\_fune \sqsubseteq_S m\_fune;;$
$m\_owan :: X \mid X \sqsubseteq bowl;;$
$m\_fune :: X \mid X \sqsubseteq boat;;$
$m\_owan\_fune :: X;;$

This description, however, is not valid because the variable scope is within a rule in Quixote. So, Xs in different modules can be bind to different objects.

Instead, how about the following representation?

$m\_owan\_fune \sqsubseteq_S m\_owan;; m\_owan\_fune \sqsubseteq_S m\_fune;;$
$m\_owan :: o \mid o \sqsubseteq bowl;;$
$m\_fune :: o \mid o \sqsubseteq boat;;$
$m\_owan\_fune :: o;;$

This leads to contradiction because the subsumption relations among basic objects are defined globally (over every module) in the current framework of Quixote. At least one of the constraints, $o \sqsubseteq bowl$ and $o \sqsubseteq boat$, always fails.

Syntactic treatment of *no* in JPSG

As can be seen in the previous subsections, an interpretation of "A *no* B" is constructed from interpretations of A and B under its utterance situation. Here, we combine the semantic analysis with syntactic treatment of *no* in JPSG. Syntactically, the lexical entry of *no* has *adjacent* and *mod* features to combine an adjacent structure and a following structure. Figure 6.1 illustrates a rough syntactic feature structure representation of "A *no* B".

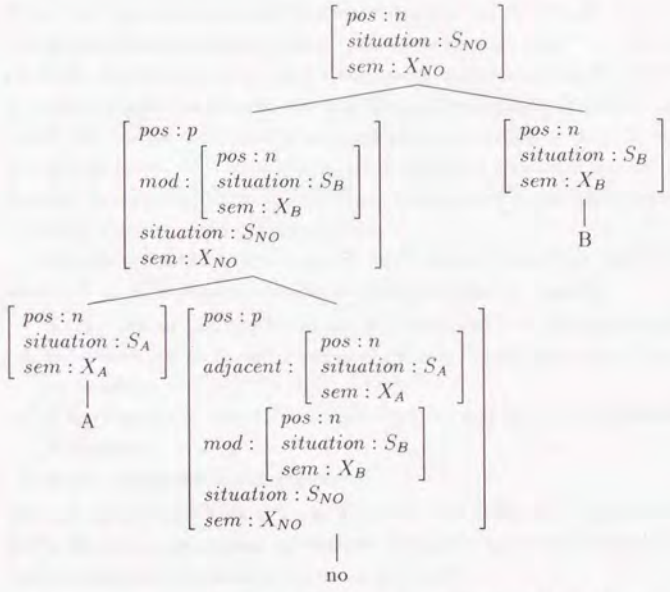


Figure 6.1: Syntactic treatment of "A no B"

The value of *sem* (semantic) feature of "A *no* B" unifies with X_{NO} that comes from the value of *sem* feature of the lexical entry of *no*. So, we have only to give the value of *sem* of *no*. Let *situation* be a feature that takes a situation identifier. According to three readings of "A *no* B" in 6.3.2, X_{NO} is defined as follows.

$$X_{NO} = S_A : X_A.X_B, \textit{ where } S_A : X_A,\ S_A = S_B = S_{NO}\ (R1)$$
$$\vee$$

$S_B : X_B/[l = X_A]$, *where* $S_B : X_B$, $S_A : X_A$, $S_A = S_B = S_{NO}$
$(R2)$

$\vee$

$S_{NO} : X$,
where $S_{NO} \sqsubseteq_S \{S_A, S_B\}$, $S_A : X \mid X \sqsubseteq X_A$, $S_B : X \mid X \sqsubseteq X_B$ $(R3)$

Recognition of "A *no* B"

We started from the standpoint that the interpretation of "A *no* B" is situation dependent, and gave a constraint-based, object-oriented representation of "A *no* B" using Quixote in 6.3.2. In this subsection, we give a processing model of "A *no* B" that explains several pragmatic aspects introduced in 6, and discuss about some ambiguities in interpreting "A *no* B". We assume the situation is stored as a set of object with some attributes, namely a database (DB). All the object in every situation and their attributes are defined in the DB. Interpreting "A *no* B" corresponds to queries to the DB that match previously explained three semantic representations.

A simple procedure to recognize "A *no* B" is listed as follows. Let Sit be the situation where "A *no* B" is uttered, O_A, O_B be interpretations of A and B.

1. If O_A exists in Sit, and O_B is one of the attributes of A, the reading is (R1):$O_A.O_B$.
2. Otherwise, search the attribute of O_B whose value can be O_A. If the attribute exists $(= l)$, the reading is (R2) $O_B/[l = O_A]$.
3. If there are submodules S_A and S_B where O_A and O_B exist respectively, reading (R3) is available.
4. If not, the phrase is unacceptable.

The ambiguities of "A *no* B" arise from the variations of interpretation of both nouns (twice for each), disjunction of semantic feature of the lexical entry of *no*(three times), and the selection of a label in the reading of (R2).

Reconsider sentences 7b in refss:pragmatic. Why *kami no zyosei* sounds strange? For the phrase, interpretation (R2) is most likely because *zyosei* cannot be an attribute of *kami* semantically, and it is hardly to imagine an object that is both *kami* and *zyosei* pragmatically. Pragmatically, nouns that specify attributes or parts of an object such as *kami,nenrei, ude* seems unlikely to be values of some other attributes. However, when such situation is possible, for instance, if we are talking about people working in a beauty parlor, *kami no zyosei* refers a certain object.

6.3.3 Discussion

Comparison with Shimazu *et. al.*'s Analysis

Shimazu[SNN86] analyzes the usage of "A *no* B" semantically into the following 5 categories and into minute 86 sub-categories.

1. B is predicative, and A is one of the cases of B.
2. B specifies the role from basis A.
3. B is an attribute of A.
4. A is predicative, and B is one of the cases of A.
5. A is one of the properties of B.

The relation between Shimazu's categorization and ours in 6.3.2 is summarized in Table 6.1. The occurrence rates are counted from 5,950 cases abstracted from essays of newspapers[SNN86].

In [SNN86] (occurrence)	Example	Our analysis, Quixote term
1 (21.0%)	Taro no kekkon	(R2) $X/[agent = taro] \mid X \sqsubseteq marriage$
2 (12.2%)	biru no mae	(R2) $front[base = building1]$ [4]
3 (6.3%)	bara no iro	(R1) $rose.color$
4 (4.6%)	sanpo no hito	(R2) $X/[action \rightarrow walk] \mid X \sqsubseteq man$
5 (56.0%)	23sai no seinen	(R2) $X/[age = 23] \mid X \sqsubseteq man$

Table 6.1: Shimazu's Analysis and Quixote term

Seen from the table, over 75% of the "A *no* B" in [SNN86] falls into our category (R2) in 6.3.2 with Quixote attribute terms. Simazu's minute categorization can be seen as default sets of labels of each object. Such lexical static analysis, however, cannot explain the variety of the interpretation of "A *no* B", especially, context dependency.

Further study

This section applies Quixote to the semantic and pragmatic representation of "A *no* B" phrases in Japanese. First, interpretations of various kinds of nouns are encoded in Quixote by using its object and constraint representation. Second, we give a categorization of the semantic interpretation of "A *no* B" into three different Quixote terms: a dotted term, an attribute term, and objects in modules. Lastly, pragmatic aspects in recognizing "A *no* B" phrases are given as a database query, where databases correspond to current situation and queries fall into semantic representation of "A *no* B".

The following aspects are remaining.

- Minute categorization of (R2). The constraints are relatively too weak, compared with (R1) and (R3).
- Implementation on top of Quixote system.

Chapter 7

Conclusion

7.1 Summary

This thesis gives a logic programming framework for constraint-based natural language analysis. Two constraint-based logic programming languages cu-Prolog and Quixote are explained and their effectiveness to constraint-based natural language analysis is illustrated through various applications.

Most of the previous works on constraint-based natural language analysis are specific to a certain theory[Car92a, Shi86] or based on traditional logic programming[PS87a]. Little studies have been made as to tackle the problem from a programming language point of view. This thesis gives a fundamental CLP framework to constraint-based natural language analysis. Two logic programming languages can be also applicable to other constraint-based problems than natural language processing.

cu-Prolog and Quixote contribute to computation linguistics as giving workable computational devices to represent and process constraints. Using cu-Prolog, for example, researchers can actually write down linguistic constraints in the form of Prolog predicates, and simulate their behavior as JPSG parser in Section 4.3. cu-Prolog thus can become a debugging tool for describing linguistic constraints. Linguistic phenomena have been discussed in various theories using various formats and logical frameworks. These programming languages can represent such knowledge and clarify the difference among theories, as shown in a treatment of "A no B" in Quixote (Section 6.3).

Reconsider the first equation in Chapter 2 that explains constraint-based grammar formalisms.

$$\begin{aligned} \text{constraint-based grammar} \;=\; & \text{feature structure} + \\ & \text{phrase structure} + \end{aligned}$$

Several features of cu-Prolog and Quixote described in this thesis are summarized as Table 7.1.

Table 7.1: Comparison among constraint-based grammar, cu-Prolog, and Quixote

	Constraint-based Grammar	cu-Prolog	Quixote
Data Structure	Feature Structure	constrained-PST	Attribute Term
Procedural	Phrase Structure	CHC (Prolog Part)	Rule
Declarative	Structural Constraint	Prolog term constraint	Subsumption constraint

Information in natural language should be described partially in syntax, semantics, pragmatics, and so on. To tackle the information partiality from the viewpoint of computer science, constraint-based programming formalisms are desirable up to now, because constraints only specify the value range or relations between variables.

Contributions of the author concerning this thesis are summarized as follows.

- The author plays a central role in designing and implementing cu-Prolog. (Chapter 3)
- The author gives various applications of cu-Prolog to natural language processing illustrated in Chapter 4 including implementation of JPSG parser (Section 4.3) which is a killer application of cu-Prolog. The example on CFG parsing (Section 4.4) was a cooperative work with Kôiti Hasida.
- Quixote itself is designed and implemented in the Quixote group at ICOT headed by Kazumasa Yokota. As a member of the group, the author contributed to design and implement Quixote especially concerning its constraint solving (Chapter 5).
- The author gives example applications of Quixote to natural language processing as explained in Chapter 6. The analysis of "A no B" is a cooperative work with Yasunari Harada.

Figure 7.1 is a map of the research around constraint-based grammar processing.

7.2 Discussion about cu-Prolog

The author would like to stress that every feature mentioned in this thesis was uniformly processed in the same framework as constraint transformation.

DP (Dependency Propagation or Dynamic constraint Processing) [Has91, HI87, HNM93, TH90], which is an extension framework of the constraint unification[Has85,

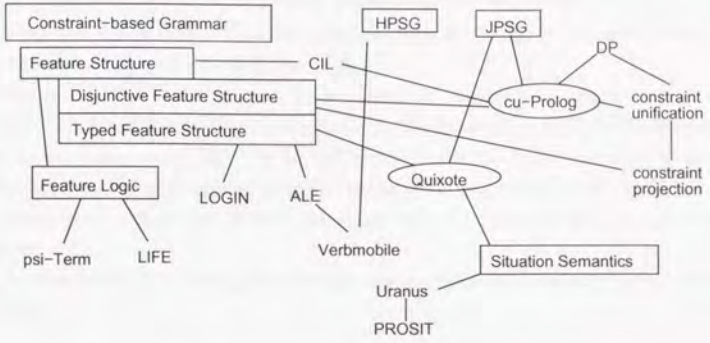


Figure 7.1: Technology Map around Constraint-based Grammar Processing

HS86], treats clausal-form logic programs by constraint transformation. DP adopts idea of the dynamics; potential energy is defined to programs and inferences are controlled to minimize the energy[HNM93]. Nakano proposes another extension of constraint unification called *constraint projection*(CP) that can treat DFS unification efficiently[Nak91]. Compared with DP and CP, cu-Prolog mixes procedural programming and constraints by CHC, and can be seen as being a more practical approach to processing constraint-based grammar.

In the current framework, every constraint is equally satisfied, such that if the constraint is over-constrained, the transformation fails. However, constraints occurring in a grammar description sometimes contradict each other and have preferences or hierarchies as shown in Section 2.2. Such cases would easily occur if we were to consider various heterogeneous linguistic constraints.

For example, Marcus postulates following two constraints[Mar80], semantic and syntactic preferences, to explain the acceptability of Wh-clauses such as sentences (2) introduced in Section 2.2.

- (Semantic preference): The preference of indirect object (IO) taken by the verb "give" is

  $$higher\_animate(people) > animate > inanimate.$$

- (Syntactic preference):
  - prefer: NEXT-as-IO: The noun next to the verb is IO.

– not-prefer: WH-comp-as-IO: The complement of the WH-clause is IO.

Cost-based abduction[HSME88] adds numerical costs and weights to literals to derive the least cost abduction as the best explanation.

What is the framework to treat such constraint hierarchy or relaxation? A cue in the field of CLP is a hierarchical constraint logic programming (HCLP) [BMMW89] proposed as an extension of CLP. In HCLP, every constraint is labeled with a strength, with constraints being processed from the stronger to the weaker ones. HCLP also provides comparators, that may differ in the application, to compare the appropriateness of solutions.

A further work of cu-Prolog is to attach such a constraint hierarchy in the constraint part of CHC.

7.3 Discussion about Quixote for NLP framework

Compare Quixote with related works. It resembles F-logic [Kif90] in the sense that it is a DOOD language and introduces object-orientation concepts into logic programming. As a knowledge representation language, however, Quixote has additional convenient features such as a module mechanism, abductive inference, and so on. Unlike conventional CLP languages[JL87], Quixote can treat constraints on symbolic constraint domain, which is suitable for natural language description. As a predecessor of the situated inference system, PROSIT is proposed [NPS91, NSHP88]. In PROSIT, each rule is asserted in one hierarchical situation and those rules are inherited. Quixote offers the same ability and the simple infons in PROSIT are extended to complex object terms in Quixote. The concept of complex object terms and attribute terms of Quixote inherits PST of CIL [MY85], that is an ancestor language developed in ICOT. The new mechanism of Quixote is summarized as follows:

- an object-orientation concept such as *object identity* is introduced into the logic programming as the fundamental philosophy,
- the concept of *module* enables us local definition in a large knowledge-base, and
- its logical inference system is extended to be able to restricted abduction.

7.4 Constraint-based NLA and Disambiguation

In the programming languages mentioned in this thesis, a natural language grammar is represented as both procedurally (in rules in cu-Prolog and Quixote) and declaratively

(in the constraints part of both languages.) The author considers the cooperation of both kinds of processing is especially important for natural language analysis. In the JPSG parser in Section 4.3, phrase structural processing in the body part of CHC constructs a skeleton phrase structure tree, and most of the feature structures are determined through constraint processing in the constraint part of CHC.

Maxwell discusses parsing constraint-based grammar as a combination of processing both phrasal constraints and attribute-value/function constraints[MK92]. He shows various strategies such as:

- interleaved strategy: functional constraint processing is interleaved in phrase structure constraint processing.
- non-interleaved strategy: first process phrase structure constraint and then process functional constraint processing.

and illustrates that the former is preferable for efficient parsing. The JPSG parser in Section 4.3 can be seen as an exemplification of the interleaved strategy, because phrase structure constraint processing is performed in the Prolog part and functional constraint processing is performed in the constraint part of CHC and they interact each other in the derivation.

Examine two kinds of processing from the disambiguation point of view. In [Cry97], "ambiguity" is explained as having following types.

- grammatical (structural) ambiguity:
- phrase-structure ambiguity: "new houses and shops" [1]
- lexical ambiguity: "I found the table fascinating." [2]
- vagueness: "He didn't hit the dog." [3]

In NLP research, the structural ambiguity has been especially studied for disambiguation to yield various algorithms such as [TOM86]. Lexical ambiguity and vagueness, however, are matters of constraints (disjunctive and negative constraints). They are represented declaratively and processed as constraint solving and constraint relaxation.

Those kinds of disambiguations have been studied independently in NLP. However, cu-Prolog and Quixote can process them totally as constraint solving. The JPSG parser in Section 4.3 states nothing about new structural disambiguation algorithms. Any existing algorithms are available in the body part of cu-Prolog. It suggests, however, a method to cooperate two kinds of – structural/procedural and constraint/declarative – processing.

[1] "new (houses and shops)" or "(new houses) and shops."
[2] table = 'object of furniture' or 'table of figures.'
[3] The sentence has unspecifiable range of meaning except for hitting the dog.

7.5 Comments about Heterogeneous Constraints

cu-Prolog and Quixote are independent programming languages. They have different constraint domains and different kinds of applications to natural language processing.

As shown in Section 2.2, the constraint domain and processing in natural language processing are inherently heterogeneous. Seen from the heterogeneity in natural language processing, the following three aspects are important.

1. To explain various natural language phenomena, the constraint domain must be diversified, such as symbolic, temporal, and term unification.
2. Seen from view of constraint processing, natural language processing involves various kinds of processes: not only constraint satisfaction but constraint relaxation.
3. the data size utilized in natural language processing is becoming larger and larger as shown in recent electronic dictionaries and large corpora. There also already exist various natural language processing resources such as dictionaries, parsers, and constraint solvers. From an engineering point of view, it is preferable to combine those existing natural language resources.

One of the further step of the constraint-based NLP research is to combine heterogeneous languages such as cu-Prolog and Quixote. In [TA94, TA96], the author gives an idea to realize the combination in MAS (Multi-agent system) called *Helios*[AYT94] that meets above three requirements. In Helios, existing problem solvers including constraint solvers are wrapped by *capsules* to become *agents*. Agents can communicate each other by sending messages in an *environment*.

Figure 7.2 is an example of a JPSG parser with heterogeneous constraint solvers.

A CFG parser, a feature structure unifier, and dictionaries are independent problem solvers to become agents. The CFG parser agent is the leader of parsing process. When the agent detects a set of constraint that cannot be solved by itself, it throws the constraint to its outside environment. The environment determines suitable destination agents to solve the constraint. The constraint is sent to the destination agents and solved to be returned to the original agent (CFG parser) through environment.

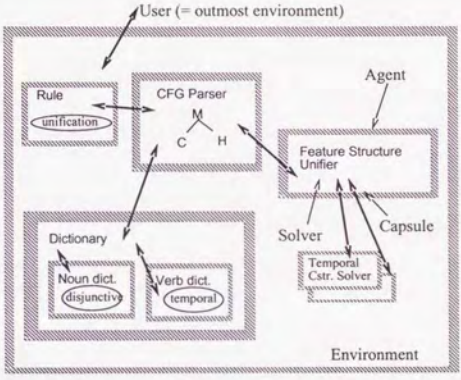


Figure 7.2: JPSG parser with heterogeneous constraints

Appendix A

Appendix I: cu-PrologIII user's manual (Abstract)

This chapter is abstracted from cu-PrologIII User's manual. The original manual is available from IFS (ICOT Free Software) whose URL is `http://www.icot.or.jp/`.

A.1 Introduction

cu-PrologIII is an implementation of cu-Prolog in the C language of UNIX BSD. cu-PrologIII is registered as IFS (ICOT Free Software) that is available from `http://www.icot.or.jp/`. Later, Hidetosi Sirai (sirai@sccs.chukyo-ac.jp) implemented cu-Prolog in Apple Macintosh and DJ's GPP (80386/486 MS-DOS machine with the DOS extender). They are also available from the above URL.

A.1.1 How to Compile cu-PrologIII

The source codes of cu-PrologIII consist of the following header files and program files in the C language.

- header files:
  `include.h funclist.h varset.h globalv.h sysp.h`
- program files concerning Prolog interpreter:
  `main.c mainsub.c new.c read.c print.c refute.c unify.c`
- program files concerning built-in predicates:
  `defsysp.c syspred1.c syspred2.c jpsgsub.c`
- program files concerning constraint transformation:
  `modular.c trans.c tr_sub.c tr_split.c`

To get the execution code of cu-PrologIII, you have only to compile and link all the modules (`*.c` files). In UNIX for example,

```
cc -o cuprolog *.c [CR]
```

or

```
make [CR]
```

A.1.2 Customize

Before compiling, you may have to rewrite some statements in `include.h` according to your system.

CPU time

cu-PrologIII uses system dependent functions to count process times.

1. If your system has `times()`[1] function,
   `#define CPUTIME 60`
   If `times()` in your system returns CPU time in N-th of a second,
   `#define CPUTIME N`
2. In Sun-4 system, `clock()` is supported. Then, please define `SUN4` as follows instead of `CPUTIME`.
   `#define SUN4 1`
3. Otherwise, the CPU time is not printed.
   `#define CPUTIME 0`

[1] `times()` is the UNIX 4.2/3 BSD Library that returns CPU time in 60th of a second.

Heap size

cu-PrologIII has the following data areas.

system heap: stores program clauses. Its size is `SHEAP_SIZE` (the default value is 20000)

user heap: stores temporal data in the Prolog interpreter. Its size is `HEAP_SIZE` (the default value is 600000)

constraint heap: stores constraints and PSTs. Its size is `CHEAP_SIZE` (the default value is 25000)

environment stack: stores temporal environments in the Prolog interpreter. Its size is `ESP_SIZE` (the default value is 500000)

user stack: is a trail stack in the Prolog interpreter. Its size is `USTACK_SIZE` (the default value is 10000) CT

string heap: stores strings. Its size is `NAME_SIZE` (the default value is 50000)

When there are frequent overflows of above areas, increase their area sizes and compile all the modules.

A.1.3 How to start and quit cu-PrologIII

To start cu-PrologIII, type the following in OS.

`cuprolog [CR]`

To start cu-PrologIII with reading an initial program, type

`cuprolog` *filename* `[CR]`

To quit cu-PrologIII, type

`%Q [CR]` or `:-halt. [CR]`

at the top level of cu-PrologIII.

A.2 Syntax of cu-PrologIII

term : atom, variable, complex term, or PST

atom : constant, string, or number

constant : sequence of characters that begins with a small letter or sequence of any characters with single quotations.

string : sequence of any characters with double quotes.

number : integer, floating number.

variable : sequence of characters that begins with a capital letter or _. _ is called an anonymous variable and any two anonymous variables are different.

complex term : let p be a string and $t_1, t_2, \cdots, t_n$ be terms, then $p(t_1, t_2, \cdots, t_n)$ be a complex term. p is called a *functor* or a *predicate symbol*. List is a special functor.

PST (Partially Specified Term) : sequence of feature/value quoted by '{' and '}'. A feature is a constant and a value is a term.

A.2.1 Constrained Horn Clause (CHC)

The program clause of cu-PrologIII is called Constrained Horn Clause(CHC) and has the following forms:

1. $H; C_1, \cdots, C_n.$ (Fact)

2. $H : -B_1, \cdots, B_m; C_1, \cdots, C_n$. (Rule)

3. $: -B_1, \cdots, B_n; C_1, \cdots, C_n$. (Question)

$H, B_1, \cdots, B_n$, and $C_1, \cdots, C_n$ are called *Head*, *Body*, and *Constraint* respectively. Horn clause is a special case (null constraint) of CHC.

cu-PrologIII allows a variable as an atomic formula. By the following programs, `call/1` and `not/1` are defined.

```
call(X) :- X.
not(X)  :- X, !, fail.
not(_).
```

A.2.2 PST (Partially Specified Term)

cu-PrologIII supports PSTs (Partially Specified Term) as a data structure to implement feature structures of constraint-based grammar formalisms. A PST is a term of the following syntax:

$\{l_1/t_1, l_2/t_2, \ldots, l_n/t_n\}$.

l_i, called **label**, is an atom and $l_i \neq l_j (i \neq j)$. t_i, called **value**, is a term. Recursive PST structures are not allowed.

For example, the unification between `{1/a,m/X}` and `{m/b,n/c}` produces `{1/a,m/b,n/c}`.

When a PST occurs in multiple places, it is printed with a new variable in the constraint part of CHC. For example,

```
f(X) :- g1(_p1,X),g2(_p2,X); _p1={f/a,g/c}.
```

A.2.3 Simplified form of Constraint

In a CHC, atomic formulas of the constraint part must be a simplified form called **modular**.

[Def] 35 (modular) *A sequence of atomic formulas $C_1, C_2, \ldots, C_m$ is* modular *when*

1. *every argument of C_i is a variable ($1 \leq i \leq m$), and*
2. *no variable occurs in two distinct places, and*
3. *the predicates occurring in C_i are modularly defined ($1 \leq i \leq m$).* □

The predicate occurring in the constraint of CHC is an ordinary Prolog predicate of the following form.

[Def] 36 (modularly defined) *A predicate* `p` *is* modularly defined, *when every body of its definition clause is modular or empty.* □

With PST, modular is naturally extended as follows.

[Def] 37 (component) *A* component *of an argument of a predicate is a set of labels to which the argument can bind. Here, an atom or a complex term is regarded as a PST of the label* □. □

`Cmp(p,n)` stands for the component of the nth argument of a predicate `p`. `Cmp(T)` represents a set of labels of a PST `T`. In a constraint of the form `X=t`, variable `X` is regarded as taking `Cmp(t)`.

Components can be computed by static analysis of the program [Tsu91]. *Vacuous argument places*[TH90] are arguments whose components are ϕ.

Consider the following example.

```
c0({f/b},X,Y):-c1(Y,X).
c0(X,b,_):-X={g/c},c2(X).
c1(X,X).
c1(X,[X|_]).
c2({h/a}).
c2({f/c}).
```

The components are computed as follows.

```
Cmp(c0,1)={f,g,h}
Cmp(c0,2)=Cmp(c1,2)={[]}
Cmp(c0,3)=Cmp(c1,1)={}
Cmp(c2,1)={f,h}
```

You can see each component with `%d` command of cu-Prolog. In the following example, `Cmp(p3,1)={f,h}` and `Cmp(p3,2)={g,h}`.

```
_%d p3
%d +---------( p3/2 )-----[f.h|g.h]--2/2--+
p3({f/a},{g/b}).
p3({h/c},{h/d}).
```

[Def] 38 (dependency) *A constraint is* dependent *when*

1. *a variable occurs in two distinct places where their components have common labels,*
2. *a variable occurs in two distinct places where one component is* $\{[]\}$ *and another component does not contain* $[]$*, or*
3. *the binding of an argument whose component is not* ϕ*.* □

For example, when `Cmp(p,1) = {f,g}`, `Cmp(q,1) = {h}`, constraint `p({f/b})` has a dependency, and `p(X),q(X)` and `p({1/a,m/b})` do not have dependencies.

[Def] 39 (modular (with PST)) *A constraint is* modular *when it contains no dependency. A Horn clause is* modular *when its body has no dependency.* □

User-defined predicates in a constraint must be defined with modular Horn clauses [2].

A.2.4 BNF description of cu-PrologIII syntax

The following is a BNF description of the syntax of cu-PrologIII.

[2] For example, `member/2`, `append/3`, and finite predicates are defined with modular Horn clauses.

$$
\begin{array}{rcl}
< char > & ::= & < capital > | < small > | < digit > \\
< whitechar > & ::= & < char > | < space > \\
< capital > & ::= & A|B|C|\ldots|X|Y|Z \\
< small > & ::= & a|b|c|\ldots|x|y|z \\
< digit > & ::= & 0|1|2|3|4|5|6|7|8|9 \\
< series > & ::= & < digit > | < digit >< series > \\
< number > & ::= & < series > | < series > . < series > \\
< charseq > & ::= & < char > | < char >< charseq > \\
< cwseq > & ::= & < whitechar > | < whitechar >< cwseq > \\
< string > & ::= & " < cwseq > " \\
< smallseq > & ::= & < small > | < small >< charseq > \\
< capitalseq > & ::= & < capital > | < capital >< charseq > \\
< term > & ::= & < var > | < atom > | < smallseq > (< term\_list >)| < PST > \\
< term\_list > & ::= & < term > | < term >, < term\_list > \\
< var > & ::= & < capitalseq > |\_ < charseq > |\_ \\
< atom > & ::= & < constant > | < string > | < number > \\
< constant > & ::= & < smallseq > |' < cwseq >' \\
< PST > & ::= & < pair\_list > \\
< pair\_list > & ::= & < pair > | < pair >, < pair\_list > \\
< pair > & ::= & < name > / < term > \\
< af > & ::= & < smallseq > (< term\_list >)| < smallseq > | \\
 & & < op\_term > | < var > \\
< af\_ist > & ::= & < af > | < af >, < af\_list > \\
< HORN > & ::= & < af > | < af >: - < af\_list > |?- < af\_list > \\
< CHC > & ::= & < horn > .| < horn >; < af\_list > . \\
< op\_term > & ::= & < op1 >< term > | < term >< op2 >< term > \\
< op1 > & ::= & not \\
< op2 > & ::= & <=> | = | = ..| > | >= | < | <= | ==
\end{array}
$$

A.3 Summary of system commands

This section lists all the system commands from the top level of cu-PrologIII. *predicate* represents *predicate_name* or *predicate_name/arity*.

A.3.1 Prolog commands

%h	help
# *OS_command*	execute OS command.
%d *predicate*	list definition clauses of a predicate
%d*	list all the program
%d/	list all predicate names include reduced predicates
%d?	list all predicate names without reduced predicates
	for system predicates, +:recursive, ^:functor
	for user predicates, *:spied, -:reduced,+:recursive,
	#: newly defined predicates during constraint transformation
%f	show free heap size
%Q	quit cu-PrologIII
%R	reset cu-PrologIII
%G	static garbage collection
%c *number*	set maximum depth of resolution
%u	toggle switch to handle undefined predicates (ERROR/TRUE)

A.3.2 File I/O commands

"*filename*"	consult file without echo back
"*filename*?	consult file with echo back
%l *filename*	set log file name
%l no	reset log file
%w *filename*	write the current program to file

A.3.3 Debug commands

%p *predicate*	switch (on/off) spy points on the predicate
%p *	set spy points on all the predicates
%p .	remove spy points of all the predicates
%p >	switch (set/remove) spy points on constraint transformation
%p ?	list spied predicates
%t	normal trace mode on/off switch
%s	step trace mode on/off switch

A.3.4 Constraint Transformation commands

%L	list derivation clauses of new predicates (in)
%a	all modular mode (in CT)
%o	M-Solvable mode (in CT)
%n *predicate*	set new predicates name (in CT) (c0,c1,...)
%P *predicate*	preprocess constraint parts
%P *	preprocess constraint parts of all the CHCs
%P ?	list predicates with non-canonical constraints

A.3.5 Other commands

%C redefine cat functor for JPSG parser

A.4 Built-in predicates, functors

This section lists built-in predicates and functors of cu-PrologIII. In the following table, + represents the input argument, - the output argument, PST the argument to take a PST.

A.4.1 Functional built-in predicates

These predicates work as functions.

Predicate	Meaning
!/0	cut
abolish/2	abolish(P+,A+) Delete definition clauses of predicate P/A.
arg/3	arg(Pos+,T+,Arg-) Unify the Posth argument of T with Arg. Eg: arg(2,test(a,b,c),X) --> X=b.
assert/1,2,3	assert(H+),assert(H+,B+),assert(H+,B+,Cstr+) Add H:-B;Cstr. in the end of the program.
asserta/1,2,3	add the clause in the beginning of the program.
assertz/1,2,3	add the clause in the beginning of the program.
attach_constraint/1	attach_constraint(Cstr+) Add new constraint Cstr. Cstr is a formula or list of formulas.
close/1	close(F) close file pointer F
compare/3	compare(X+,Y+,C-) When X and Y are numbers or strings, C unifies with their relation. The relation is >,==, or < Eg: compare("abc","xyz",'<') compare(123,456,'<')
concat2/2	concat2(Str+,List-) List is a list of characters of string Str Eg: concat2("ab",["a","b"])
default/3	default(X?,Y+,Z+) If PST X unifies with Y, unify X with Z,

otherwise, fails.
Eg: :- X={pos/p,sem/Y},default(X,{pos/p},{ajn/□})
makes X={pos/p,ajn/□,sem/Y}.
:- X={pos/n,sem/Y},default(X,{pos/p},{ajn/□}). fails.

divstr/4 — divstr(X+,N+,Y-,Z-)
Y unifies with the first N characters of string X and Z the rest.
When N < 0, Y unifies with the last -N characters.
Eg: divstr("abcdefg",3,"abc","defg").
divstr("abcdefg",-2,"abcde","fg").

equal/2 — equal(X,Y) unify X and Y

eq/2 — eq(X,Y) check X is equal to Y

fail/0 — always fails

functor/3 — functor(T+,F,A) predicate name of term T is F/A.

geq/2 — geq(X+,Y+) check X>=Y

greater/2 — greater(X+,Y+) check X > Y

halt/0 — quit cu-PrologIII

leq/2 — leq(X+,Y+) check X=<Y

less/2 — less(X+,Y+) check X < Y

ml/2 — univ. ml(T,L) is equivalent to T=..L.
L is a list of predicate name and arguments of term T.
Eg: ml(f(a,b,c),X) --> X=[f,a,b,c]

multiply/3 — multiply(X+,Y+,Z-) is X * Y = Z

name/2(A,L) — name(A+,L-) the name of the atom A is the list L
Eg: name(abc,[97,98,99]).

nl/0 — write '\n'

op/3 — op(P+,T+,Op+) Define operator Op as precedence P and type T.
P $in[0,1000]$ and T is xf,yf,fx,fy,xfx,xfy, or yfx.
Eg: :-op(700,xfx,'=').

open/3 — open(FileName+,Type+,FP-) Open a stream.
FileName is a file name. Type is r (read) or w (write).
FP is a corresponding file pointer.

pnames/2 — pnames(PST+,FL) Unify the list of features of PST with FL.
Eg: pnames({1/a,m/b},[1,m]).

pvalue/3 — pvalue(PST+,F+,V) Unify the value of feature F of PST with V.
When PST does not have feature F, it fails.

read/1 — read(X-), read a term for the keyboard and unify it with X.

read/2 — read(X-,FP+) read a term form FP and unify it with X.
When FP is the end of files, X unifies with end_of_file

reset_timer/0 — reset CPU timer(cf. timer/2)

see/1 — see(F-) file F becomes the current input stream

seen/0 — close the current input stream

strcmp/3 — strcmp(X+,Y+,C-)
Unify C with the relation between strings X and Y.

	(cf. compare/3)
	C is >,==,or <
	Eg: strcmp("abc","xyz",'<').
strlen/2	strlen(S+,L) L is the length of string S
	Eg: strlen("abcdef",6).
substring/3	substring(X+,N+,Y-)
	Unify Y with more than Nth characters of string X
	(When N < 0, Y unifies with the last -N characters of X.)
	Eg: substring("abcdefg",3,"defg")
	substring("abcdefg",-3,"efg").
substring/4	substring(X+,N+,L+,Y-)
	Unify Y with L characters from Nth element of string X.
	Eg: substring("abcdefg",3,2,"de")
	substring("abcdefg",-3,2,"ef").
sum/3	sum(X,Y,Z) compute X + Y = Z
	(More than one argument must be bound.)
tab/0	print tab
tab/1	tab(FP+) print tab to file pointer FP
tell/1	tell(T+) file T becomes the current output stream
timer/2	timer(X-,Y-) X unifies with CPU time after reset_timer.
	Y unifies with CUP time used in the constraint transformation.
told/0	close the current output stream made by tell/1
true/0	always succeeds
unbreak/0	return to the breakpoint in the step trace.
var/1	var(T) T is a free variable
write/1	write(T) write term T

A.4.2 Predicative built-in predicates

These predicates may have many solutions by backtracking.

PREDICATE	MEANING
clause/3	clause(T+,Body-,Cstr-)
	There is a program clause Head:-Body;Cstr.
	where Head unifies with T.
concat/3	concat(S1+,S2+,S-) or concat(S1-,S2-,S+)
	string S is a concatenation of S1 and S2.
	Eg: concat("ab","cd",S) --> S="abcd"
	concat(X,Y,"abc") -->
	X="",Y="abc" or X="a",Y="bc" or
count/1	count(X-) Produce an integer to be unified with X (X=0,1,2,$\cdots$).
execute/1	execute(L+) Execute goals given as a list L
	Eg: execute([memb(X,[a,b]),memb(X,[b,c])])
gensym/1	gensym(X-) Create a new string to X.

	%n changes the first half of the string (default is c).
isop/3	isop(Prec,Type,OP)
	Prec and Type is a precedence and type of the operator OP.
	Eg: :-isop(X,Y,Z). --> X=900, Y=xfy, Z='/' etc.
memb/2	member(Atom,List+) built-in member predicate
or/2,3,4,5	Execute more than one goals.
	Eg: :-or(memb(X,[a,b]),memb(X,[j,k])).
	--> X=a,b,j,k
retract/1,2,3	retract(Head+), retract(Head+,Body+),
	retract(Head+,Body+,Cstr+)
	Delete a program clause that unifies with Head:-Body;Cstr.

A.4.3 Built-in predicates for constraint transformation

The following are special built-in predicates for constraint transformation.

Predicate	Meaning
condname/2	condname(Cstr+,PL-) Unify predicate names in Cstr with PL.
	Eg: condname([f(a,b),g(c,d)],[g,f])
pcon/0	Print current constraints.
unify/2	unify(C+,NC-) Transform constraint C into NC.
	Here, C and NC are list of atomic formulas.
	Eg: unify([member(X,[a,b,c]),f(X,Y)],[c0(X,Y)])

A.4.4 Built-in predicates for JPSG parser

The following are special built-in predicates for JPSG parser.

Predicate/Functor	Meaning
cat/6	Functor for the feature structure of JPSG
t(M,L,R)	Functor for history
tree(H)	Print history H in tree format

cat/6 functor is set by %C command from the top level of cu-PrologIII as follows.

%C [$Feature_1$,$FType_1$,$Feature_2$,$FType_2$,...]

Here, $Feature_i$ is a feature name that begins with a capital letter and be shorter than five letters. $FType_i$ is a feature type defined as follows.

FType	Meaning
2	takes one category (adjacent,slash, etc.)
3	takes set of categories (subcat, etc.)
1	otherwise

Default value is `[POS,1,FORM,1,AJA,2,AJN,2,SC,3,SEM,1]`, that is,

Feature	PSG features
POS	pos
FORM	gr,vform,pform, and so on
AJA	adjacent
AJN	adjunct
SC	subcat
SEM	sem

By `tree/3`, functor `cat`($Arg_1, \cdots, Arg_n$) is printed as
Arg_1`[`Arg_2`,` $Feature_3$`:`Arg_3`,` $\cdots$`,` $Feature_{n-1}$`:`Arg_{n-1}`]:`Arg_n
Features of null values (`[]`) are not printed.

[Def] 40 (History) *A* history *is defined recursively as follows.*

- *A category is a history*
- *If* `C` *and* `W` *are categories, then* `t(C,W,[ ])` *is a history.*
- *If* `L` *and* `R` *are histories and* `M` *is category, then* `t(M,L,R)` *is a history.*
- *If* `L` *and* `R` *are histories, and* `C` *and* `W` *are categories, then* `t(t(C,W,[ ]),L,R)` *is a history.* □

`tree(t(C,W,[ ]))` writes

```
C--W
```

and `tree(t(M,L,R))` writes

```
---M
   |
   |-L
   |
   |-R
```

A.5 File I/O

A.5.1 Read a program

- Start cu-PrologIII from OS with reading an initial file:
  `cuprolog` *filename* `[CR]`
- Read a file in the top level of cu-PrologIII without echo back:
  "*filename*" `[CR]`
- Read a file in the top level of cu-PrologIII with echo back:
  "*filename*? `[CR]`

A.5.2 Save a program

To save current program clauses to a file in the top level of cu-PrologIII,
`%w` *filename* `[CR]`

A.5.3 Log file

- Set log file :
  %1 *filename* [CR]
- End log file :
  %1 no [CR]

A.6 Constraint Transformation

A.6.1 Use constraint transformer alone

You can use the constraint solver (transformer) of cu-PrologIII alone in the following two ways.

'@' command

First, from the top level of cu-PrologIII, type '@' followed by a sequence of atomic formulas and a period. For example, by typing as follows,

```
@ member(X,[a,b,c]),member(X,[b,c,d]).  [CR]
```

Then, cu-PrologIII returns the equivalent modularly defined constraint and its definitions.

unify/2 predicate

Second, you can use the constraint transformation routine as a Prolog procedure. cu-PrologIII has the predicate `unify(OldCond, NewCond)`. `unify/2` takes constraints as a list of literals as follows.

```
[c0(X,Y), c1(P,Q,R), c2(Q,S)]
```

`unify/2` succeeds iff `OldCond` is instantiated to a constraint and `NewCond` is a free variable. `NewCond` is instantiated to the modularly defined constraint that is equivalent to `OldCond`.

A.6.2 Transformation operations

See Section 3.6.2.

A.6.3 Example

This subsection shows some example in the constraint transformer of cu-PrologIII.

Symbolic and combinatorial constraint

Predicates `member/2` and `append/3` are frequently used as symbolic and combinatorial constraints. The following example transforms non-modular constraints into modular ones.

```
tsuda@icot21[5] cup3              % start cu-Prolog from UNIX
*****  cu - Prolog   Ver. III  *****
Copyright: Institute for New Generation Computer Technology,  Japan 1989-91
in Cooperation with SIRAI@sccs.chukyo-u.ac.jp
```

```
All Modular mode (help ->  %h)

_member(X,[X|Y]).                    % definition of member/2
_member(X,[Y|Z]):-member(X,Z).
_append([],X,X).                     % definition of append/3
_append([A|X],Y,[A|Z]):-append(X,Y,Z).

_@ member(X,[a,b,c]),member(X,[b,c,d]).    % User's constraint input 1

solution = c0(X_0)              % Equivalent and modular constraint
c0(b).                          % New predicates made in the transformation
c0(c).
CPU time = 0.017 sec (Constraints Handling = 0.000 sec)

_@ member(X,[a,b,c]),member(X,[j,k,l]).    % User's constraint input 2

solution = fail.                % Transformation fails.
CPU time = 0.017 sec (Constraints Handling = 0.000 sec)

_@ member(A,X),append(X,Y,Z).            % User's constraint input 3

solution = c2(X_1, Y_2, Z_3, A_0)     % Equivalent and modular constraint
c4(V0_0, V1_1, V2_2, [V0_0 | V3_3]) :- append(V1_1, V2_2, V3_3).
                                      % New predicates
c3(V0_0, V1_1, V2_2, [V0_0 | V3_3], V4_4) :- c2(V1_1, V2_2, V3_3, V4_4).
c2([V0_0 | V1_1], V2_2, V3_3, V0_0) :- c4(V0_0, V1_1, V2_2, V3_3).
c2([V0_0 | V1_1], V2_2, V3_3, V4_4) :- c3(V0_0, V1_1, V2_2, V3_3, V4_4).
CPU time = 0.050 sec (Constraints Handling = 0.000 sec)
```

A.7 Program trace

A.7.1 Set spy points

%p *	set spy points on all the predicates.
%p .	remove spy points on all the predicates.
%p *predicate*	switch (set/remove) the spy point on the predicate
%p >	switch (set/remove) the spy point on the unfold/fold transformation.
%p ?	list spied predicates.

A.7.2 Set trace flag

There are two trace modes of spied predicates.

%s switch step (interactive) trace (on/off). In this mode, the prompt is '>'.

%t switch normal trace (on/off). In this mode, the prompt is '$'.

In the step trace mode, the user suggests the next action at spy points.

INPUT	ACTION
[CR]	continue
s	skip tracing

a	print ancestor goals.
b	break. Go to cu-PrologIII temporally. By :-unbreak, user can return to this point.
f	fail this goal.
l	leap. skip until the current goal ends.
z	quit refutation

A.7.3 Trace of constraint transformation

To trace constraint transformation, first spy constraint transformation by %p >, then type %t for the normal trace and %s for the step trace.

Print traces

Clauses in DEFINITION (derivation clauses of new predicates) are printed as follows.

[Clause_Number(Status,Number_of_Definitions)] Derivation_Clause
`[1(d,0)] c0(X) <=> member(X,[a,b,c]).` (Example)

Here, Status is:

Status	Function
r	removed: clause (other than f and g) removed in unfolding
d	derivation: derivation clauses not unfolded
g	registered: unfolding of this clause yields at least one unit clause.
f	false_registered: unfolding of this clause fails

Clauses in NON-MODULAR or MODULAR are printed as follows.

<Clause_Number(Status,Number_of_Definitions)> Clause
`<2(u)> c0(X):-member(X,[b,c]).` (Example)

Here, Status is defined as follows.

Status	Function
r	removed: clause removed by reduction or unfolding
u	untouched: clause in NON-MODULAR that is not unfolded
m	modular: clause whose body is modular
i	unit: unit clause

Commands in the step trace mode

In the step trace mode, the user have to suggest the next action after spy points.

Input	Function
[CR]	continue: continue with default heuristics.
u CN LN	manual unfolding: CN is a clause number, its LNth formula is selected for unfolding.
s	stop tracing: notrace after this

n	stop step tracing: normal trace after this
q	quit transformation (asserting the current clauses).
z	abort transformation (deleteing all the new clauses).

Appendix B

Appendix II: JPSG/HPSG parser in cu-Prolog

This appendix shows a sample HPSG parser and JPSG parser program in cu-Prolog.

B.1 Simple HPSG parser in cu-Prolog

This program is a simple HPSG parser. The grammar was given by G. Smolka during our private communication in 1994.

The program implements the following three features.

feature name	meaning	value
ph	phonological feature	string
head	head feature	{noun, adjective, determiner, verb}
sc	subcat feature	list of head features

The grammar contains two kinds of phrase structure rules.

1. $Mother \leftarrow HeadDaughter + NonHeadDaughter$
2. $Mother \leftarrow NonHeadDaughter + HeadDaughter$

Each feature follows the following set of principles.

Phonological Feature Principle The value of phonological feature of the mother unifies with the concatenation of the phonological values of her daughters.

Head Feature Principle The value of head feature of the mother unifies with that of her Head daughter.

Subcat Feature Principle The value of subcat feature of the head unifies with the append of the head value of the non-head and the subcat value of the mother.

```
%%%%%%%%%%%%%%%%%%%%%%%%% hpsg.p %%%%%%%%%%%%%%%%%%
%%       Simple  HPSG parser
%%                          1994.5.20
%%       H.Tsuda
%%       (grammar by G.Smolka)
%%     {head/_, sc/_, ph/_}
%% ---------------------------------------------
%%       head: head feature
%%       sc:   subcat feature
%%       ph:   phonological feature
%% ---------------------------------------------
%%  Example.
%%  ?-p([mary,meets,john]).
%%  ?-p([the,girl,is,mary]).
%%  ?-p([mary,is,embarrassed]).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%  Left Corner Parser
p(Sentence):-
        parse0(Cat,H,Sentence,[]),nl, tree(H),nl,
```

```
21:         write("category= "),write(Cat),nl,
22:         write("constraint= "),pcon,nl.
23:
24: parse0(MCat,MHist,Str,Rest):-
25:         lookup(Str,SubStr,Cat,Hist),!,
26:         parse1(Cat,Hist,MCat,MHist,SubStr,Rest).
27:
28: parse1(Cat,H,Cat,H,Str,Str).
29:
30: dictparse1(LCat,LHist,GCat,GHist,Str,Rest):-
31:         psr(LCat,RCat,MCat,RN),
32:         parse0(RCat,RHist,Str,SubStr),
33:         parse1(MCat,t(t(MCat,RN,[]),LHist,RHist),
34:                GCat,GHist,SubStr,Rest).
35:
36:
37: %%  phrase structure rules
38: %%      psr(LeftCat,RightCat,MotherCat)
39: psr(Head,D,P,1);              %  Right head
40:    sc_p(Head,D,P),
41:    head_p(Head,P),
42:    ph_p(Head,D,P).
43: psr(D,Head,P,2);              %  Left head
44:    sc_p(Head,D,P),
45:    head_p(Head,P),
46:    ph_p(D,Head,P).
47:
48: %% head feature principle
49: %% head_p(HeadDaughter, Mother)
50: head_p({head/H},{head/H}).
51:
52: %% phonology feature principle
53: %% ph_p(LeftDaughter, RightDaughter, Mother)
54: ph_p({ph/LP},{ph/RP},{ph/PP}) :- append(LP,RP,PP).
55:
56: %% subcat feature principle
57: %% sc_p(Head,Daughter,Mother)
58: sc_p({sc/[RH|PSC]},{head/RH,sc/[]},{sc/PSC}).
59:
60: %%      dictionary
61: %%      lookup(Str,RestStr,Cat,History)
62: lookup([Word|X],X,{ph/[Word],head/Cat,sc/SC},t(Cat,[Word],[]))
63:          :-(Word,Cat,SC).
64:
65: dict(mary, noun, []).
66: dict(john, noun, []).
67: dict(girl, noun, [determiner]).
68: dict(nice,adjective,[]).
69: dict(pretty,adjective,[]).
70: dict(the,determiner,[]).
71: dict(laughs,verb,[noun]).
72: dict(meets,verb,[noun,noun]).
73: dict(kisses,verb,[noun,noun]).
```

```
74: dict(embarrasses,verb,[noun,noun]).
75: dict(thinks,verb,[verb,noun]).
76: dict(is,verb,[adjective,noun]).
77: dict(met,adjective,[]).
78: dict(kissed,adjective,[]).
79: dict(embarrassed,adjective,[]).
80:
81: %%% constraints definition
82: append([],X,X).
83: append([A|X],Y,[A|Z]):-append(X,Y,Z).
84: member(X,[X|Y]).
85: member(X,[Y|Z]):-member(X,Z).
```

B.2 JPSG parser in cu-Prolog

The program implements following features.

feature name	meaning	feature type	value
core	core syntactic features	head feature	core category
ajc	adjacent category	adjacent feature	FS (feature structure)
sc	subcategorization	subcat feature	list of FS(s)
adj	adjunction	head feature	feature structure
psl	slash for relative clause	slash feature	list of FS(s)
slash	slash	slash feature	list of FS(s)
refl	reflective	slash feature	list of FS(s)
sem	semantics	head feature	Prolog term
temp	temporal feature in the proto lexicon	head feature	Prolog term

core feature takes a core category that is a feature structure with the following syntactic features.

feature name	meaning	value
pos	part of speech	p, n, v, vs, adn, adv
form	verb form, noun form	n, ns, v, vv, vk, vcw, adj, na
view	aspect (derived from temp)	Prolog term

```
 1: %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 2: %%              JPSG parser ver1.3
 3: %%              1992.6.6  Hiroshi Tsuda
 4: %%              1995.12   commented by John Fry (ETL)
 5: %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 6: %%  Category :
 7: %%       {core/{pos/Pos, form/Form, view/View},
 8: %%         ajc/Adjacent, sc/Subcat, ajn/Adjoin,
 9: %%         psl/PSlash, slash/Slash, refl/Refl, sem/Sem (,temp/Temp)}
10: %% Pos: part of speech   p,n,v,vs(sahen-dousi),adn(rentai-si),adv(fuku-si)
```

```
11: %% Form: when Pos=n, n,ns(sahen-meisi)
12: %%        when Pos=v, vv,vk,vcw,... (verb form),
13: %%                adj(adjective), na(adjective-verb)
14: %% Temp: Temp feature for proto-lexicon
15:
16: %%%  Left Corner Parser
17: p2(S):-parse0(A,B,S,[],[idx(s,speaker)]),write(A),nl,pcon,nl.
18: p(Sentence):-
19:         parse0(Cat,H,Sentence,[],[idx(s,speaker)]),nl, tree(H),nl,
20:         write("category= "),write(Cat),nl,
21:         write("constraint= "),project_cstr(Cat),nl.
22:
23: nil_or_speaker([]).
24: nil_or_speaker([{sem/speaker}]).
25:
26: parse0(MCat,MHist,Str,Rest,Idxlist):-
27:         lookup(Str,SubStr,Cat,Hist,Idxlist,Nidx),!,
28:         parse1(Cat,Hist,MCat,MHist,SubStr,Rest,Nidx).
29:
30: parse1(LCat,LHist,GCat,GHist,[Word|SubStr],Rest,Idxlist):-
31:         lookup_post(LCat,Word,RCat,RHist,RuleName),
32:         psr_adj(LCat,RCat,MCat),
33:         parse1(MCat,
34:                 t(t(MCat,RuleName,[]),LHist,RHist),
35:                 GCat,GHist,SubStr,Rest,Idxlist).
36:
37: parse1(Cat,H,Cat,H,Str,Str,N).
38:
39: parse1(LCat,LHist,GCat,GHist,Str,Rest,Idxlist):-
40:         psr(LCat,RCat,MCat,RN),
41:         parse0(RCat,RHist,Str,SubStr,Idxlist),
42:         parse1(MCat,t(t(MCat,RN,[]),LHist,RHist),
43:                 GCat,GHist,SubStr,Rest,Idxlist).
44:
45: %%  phrase structure rules
46: %%     psr(LeftCat,RightCat,MotherCat)
47:
48: %% 1. Adjacent Structure:  psr_adj(Left,Head,Mother)
49: %%                          (CHC)
50: psr_adj({core/Cc,sc/Csc,refl/Cref,slash/Csl,psl/Cpsl,sem/SEM0,ajn/[]},
51:     {core/Hc,ajc/[{core/Cc,sc/Asc,refl/ReflAC,sem/SEM0}],
52:       ajn/Adj, sc/Hsc, refl/Href, slash/Hsl, sem/SEM},
53:     {core/Hc,ajc/[],ajn/Adj,sc/Msc,refl/Mref,slash/Msl,psl/Cpsl,sem/SEM});
54:         adjacent_sc_p(Csc,Asc,Hsc,Msc),
55:         slash_p(Csl,Hsl,Msl),
56:         refl_cond(Cref,Href,Mref,Hsc).
57:
58: %% slash feature principle:
59: %%     slash_p(LeftS,RightS,MotherS)
60: %%     LeftS=C.slash, RightS=H.slash, MotherS=M.slash
61: slash_p([],[],[]).
62: slash_p([S],[],[S]).
63: slash_p([],[S],[S]).
64: slash_p([S],[RS],[RS]):-sem_unify(S,RS).
65: sem_unify({sem/X},{sem/X}).
```

```
66:
67: %%  adjacent-subcat principle for adjacent structure
68: %% adjacent_sc_p(CSC,ASC,HSC,MSC).
69: %%       CSC=C.sc, ASC=H.ajc.sc, HSC=H.sc, MSC=M.sc
70: %% adjacent_sc_p(CSC,[],HSC,MSC):-merge(CSC,HSC,MSC).
71: adjacent_sc_p([],[],SC,SC).
72: adjacent_sc_p([SC|R],[],[],[SC|R]).
73: adjacent_sc_p(CSC,[AS],HSC,SC):-one_of(CSC,AS,Rest),append(HSC,Rest,SC).
74: adjacent_sc_p([A1,A2|R],[A1,A2],SC,MSC):-append(SC,R,MSC). /* passive */
75:
76: %%      2. relative clause structure : psr(R,H,M)
77: %%                          (CHC)
78: psr({core/{form/rel},sc/Rsc,slash/Rsl,psl/Rps,sem/Rs,ajc/[],ajn/[]},
79:     {core/Hc,ajc/Ha,slash/Hsl,sem/Hs},
80:     {core/Hc,sc/[],slash/Msl,psl/[],sem/[Hs,Rs]},[relative_s]);
81:         {pos/n}=Hc,sc_sl(Rsc,Rsl,Rps,Ha,Hsl,Msl,Hs).
82:
83: sc_sl([],[{sem/Hsm}],Rps,[],Hsl,Msl,Hsm):-slash_p(Rps,Hsl,Msl).
84: sc_sl([],Rsl,Rps,[{core/{form/rel}}],Hsl,Msl,Hsm):-
85:      sl_psl_p(Rsl,Hsl,Msl,Rps).
86: sc_sl([{sem/Hsm}],[Rsl],Rps,[],Hsl,Msl,Hsm):-sl_psl_p([Rsl],Hsl,Msl,Rps).
87: %% sl_psl_p(Csl,Hsl,Msl,Psl)
88: sl_psl_p([],[],[P],[P]).
89: sl_psl_p(Csl,Hsl,Msl,[]):-slash_p(Csl,Hsl,Msl).
90: sl_psl_p(Csl,Hsl,Msl,[{sem/X}]):-slash_p(Csl,Hsl,[{sem/X}]).
91:
92: %%      3. Subcategorization Structure : psr(C,H,M)
93: %%                              (CHC)
94: psr(Comp,
95:   {core/Hc, ajn/Hn, ajc/Hac, sc/HC, refl/Hr, slash/Hsl, sem/Hs},
96:   {core/Hc, ajn/Hn, ajc/Hac, sc/Rest, refl/Mr, slash/Msl,psl/Cps,sem/Hs},
97:       [subcat_p] );
98:       {core/Cc, ajc/[], refl/Cr,slash/Csl,psl/Cps, ajn/[]}=Comp,
99:       one_of(HC,Comp,Rest),
100:      slash_p(Csl,Hsl,Msl),
101:      refl_cond(Cr,Hr,Mr,Rest),
102:      sc_cond(Cc,Hc).
103:
104: sc_cond({pos/p},{pos/v}).
105: sc_cond({pos/adn},{pos/n}).
106:
107: refl_cond([],[],[],Rt).
108: refl_cond([],[Cat],[Cat],Rt).
109: refl_cond([Cat],[],[Cat],Rt).
110: refl_cond([Cat],[],[],SC) :- memb3(Cat,SC).
111:
112: %%%     4. Adjunction Structure: psr(An,H,M)
113: psr({core/_,ajn/[Head], slash/Asl,refl/ReflA,psl/Apsl,sem/As},
114:     Head,
115:     {core/C,ajn/A,ajc/[],sc/Hsc,slash/Msl,refl/ReflM,psl/Apsl,sem/As},
116:         [adjunct_p] );
117:         {core/C,ajn/A,ajc/[],sc/Hsc,slash/Hsl,refl/ReflH,sem/Hs}=Head,
118:         refl_cond(ReflA,ReflH,ReflM,Hsc),
119:         slash_p(Asl,Hsl,Msl).
120:
```

```
%%      lexical rule (for general dictionary)
%%   lex_rule(OrigCat,NewCat)
lex_rule(C,Cat):-
        same_feature(C,Cat),
        sc_to_sl(C,Cat),default(Cat,{},{ajc/[],ajn/[],refl/[]}).

%% core,sem
same_feature({core/C,sem/S},{core/C,sem/S}).

%% sc,slash (subcat to slash movement, sc,slash,psl default)
%%  (CHC)
sc_to_sl({sc/[]},{sc/[],slash/[],psl/[]}):-!.
sc_to_sl({sc/S},{sc/Nsc,slash/Nsl,psl/[]});sc_sl_move(S,Nsc,Nsl).
sc_sl_move(Sc,Sc,[]).
sc_sl_move(Sc,Nsc,[{sem/Sl}]):-one_of(Sc,{sem/Sl},Nsc).

%% temp,asp (temp to aspect conversion, asp default)
%% temp_to_asp({temp/[]},{core/{asp/[]}}):-!.
%% temp_to_asp({temp/t(S,F,R,DSF,DRF)},{core/{view/asp(AB,AE,AD,AT)}});
%%      temp_cstr(S,F,R,DSF,DRF,AB,AE,AD,AT).
temp_cstr(S,F,T0,DSF,T1,S,F,DSF,basic).
temp_cstr(T0,F,R,T1,f,F,R,f,result).
temp_cstr(T0,F,T1,T2,T3,F,i,i,exp).
temp_cstr(T0,T1,R,T2,f,R,i,i,exp).

%%      general dictionary
%%      (CHC)
lookup([idx(Word,I)|X],X,Cat,t(Cat,[Word,I],[]),OldIdx,OldIdx)
        :-member(idx(I,Sem),OldIdx),!,
        dict1(Word,C),lex_rule(C,Cat);{sem/Sem}=Cat.
lookup([idx(Word,I)|X],X,Cat,t(Cat,[Word,I],[]),OldIdx,
        [idx(I,Sem)|OldIdx]) :-!,
        dict1(Word,C),lex_rule(C,Cat);{sem/Sem}=C.
lookup([[Sent|Cont]|X],X,Cat,Hist,Idx,Idx):-
        parse0(Cat,Hist,[Sent|Cont],[],Idx).
lookup([Word|X],X,Cat,t(Cat,[Word],[]),Idx,Idx)
        :-dict1(Word,C),lex_rule(C,Cat).

%%      pp, suffix, auxiliary verb dictionary  (CHC)
%%      lookup_post(PreCat, Word, PostCat, PostHist,RuleName)
lookup_post({core/{pos/v,form/Form}},Word, Cat,t(Cat,[Word],[]),[suff_p])
        :-search_suffix(Form,Word,Cat);
        {slash/[],psl/[],refl/[]}=Cat.
lookup_post(_,Word,Cat,t(Cat,[Word],[]),[adjacent_p])
        :-dict_pos(Word,Cat);
        {slash/[],psl/[],refl/[]}=Cat.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%   Various constraints
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%      general constraint
%%  constraint  (finite predicate)

%% one_of(List, One, Rest)     |List|<=3
one_of([X|Y],X,Y).
```

```
177: one_of([X,Y|Z],Y,[X|Z]).
178: one_of([X,Y,Z],Z,[X,Y]).
179:
180: %% member(X,List)        |List|<=3
181: memb3(X,[X|Y]).
182: memb3(X,[Y,X|Z]).
183: memb3(X,[Y,Z,X]).
184:
185: %% finite permutation
186: perm2([A,B],[A,B]).
187: perm2([A,B],[B,A]).
188: perm3([A,B,C],[A|X]):-perm2([B,C],X).
189: perm3([A,B,C],[B|X]):-perm2([A,C],X).
190: perm3([A,B,C],[C|X]):-perm2([A,B],X).
191:
192: %%  constraint (recursive predicate)
193: merge([],[],[]).
194: merge([],[A|X],[A|X]).
195: merge([A|X],[],[A|X]).
196: merge([A|X],[B|Y],[A|Z]):-merge(X,[B|Y],Z).
197: merge([B|X],[A|Y],[A|Z]):-merge([B|X],Y,Z).
198:
199: append([],X,X).
200: append([A|X],Y,[A|Z]):-append(X,Y,Z).
201:
202: member(X,[X|Y]).
203: member(X,[Y|Z]):-member(X,Z).
204:
205: %%%%%%%%%%%%% SA-HEN constraint %%%%%%%%%%%%%%%%%%%
206: su_handler(Adjacent,v_su,[],SC,Sem) :-
207:       suru(Adjacent,SC,Sem).
208: su_handler([{core/{pos/v,form/vcs},ajc/[],ajn/[],sc/[],sem/Sem}],
209:                 Form,[],[],Sem).
210:
211: suru([{core/{pos/n,form/ns},sc/Subc,sem/[Pred|SRest]}],
212:         SC,[Pred|SRest]):- suru_correspond(SRest,Subc,SC).
213: suru([],[{core/{pos/p,form/wo},
214:                 sc/[{core/{pos/adn,form/ga},refl/RF,sem/Sbj}|Sc],
215:         refl/RF,sem/[Pred,Sbj,Obj|SRest]}|SC],
216:         [Pred,Sbj,Obj|SRest] ):-
217:        suru_correspond([Sbj|SRest],
218:         [{core/{pos/adn,form/ga},refl/RF,sem/Sbj}|Sc],SC).
219: suru([],
220:         [{core/{pos/p,form/ga},refl/RF,sem/Sbj},
221:         {core/{pos/p,form/wo},
222:                 sc/[{core/{pos/adn,form/ga},refl/RF,sem/Sbj}|SC],
223:                 sem/[Pred,Sbj,Obj|SRest]}|SC],
224:      BR,RF,[Pred,Sbj,Obj|SRest]) :- suru_correspond(SRest,Sc,SC).
225:
226: suru_correspond([],[],[]).
227: suru_correspond([Sbj], [{core/{form/F},sem/Sbj}],
228:         [{core/{pos/p,form/F},sem/Sbj}]).
229: suru_correspond([Sbj,Obj],
230:       [{core/{form/FO},sem/Obj},{core/{form/FS},sem/Sbj}],
231:       [{core/{pos/p,form/FS},sem/Sbj},{core/{pos/p,form/FO},sem/Obj}]).
```

```
232: suru_correspond([Sbj,Obj,Iob],
233:         [{core/{form/FO},sem/Obj},
234:         {core/{form/FI},sem/Iob},{core/{form/FS},sem/Sbj}],
235:         [{core/{pos/p,form/FS},sem/Sbj},
236:         {core/{pos/p,form/FI},sem/Iob},
237:         {core/{pos/p,form/FO},sem/Obj}]).
238:
239: adn_2([{core/{pos/adn,form/First},sem/Obj},
240:         {core/{pos/adn,form/Second},sem/Sbj}],
241:         First,Second,Obj,Sbj).
242: adn_2([{core/{pos/adn,form/First},sem/Obj},
243:         {core/{pos/adn,form/Second},sem/Sbj}],
244:         Second,First,Obj,Sbj).
245:
246: adn_wo_ga(SC,Obj,Sbj):-adn_2(SC,wo,ga,Obj,Sbj).
247:
248: adn_1([{core/{pos/adn,form/First},sem/Sem}],Form,Sem).
249:
250: %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
251: %%      PP (postposition), aux , etc.  (have ADJACENT feature)
252: %%      dict_pos( 1word, Cat)           : pp, aux, etc.
253: %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
254:
255: %%      Post Positions
256: dict_pos(nado,
257:         {core/{pos/n,form/Form},ajn/[],sc/[],
258:         ajc/[{core/{pos/n,form/Form},sc/[],sem/SEM}],sem/etc(SEM)} ).
259:
260: %%   Postpositional particles (Zyosi)
261: %%  --wo,ga,ni,de,to,no,ha,ba
262: general_pp(Form,
263:         {core/{pos/p,form/Form},ajn/[],sc/[],
264:         ajc/[{core/{pos/n},sc/[],sem/SEM}],sem/SEM} ).
265: dict_pos(wo, Cat):-general_pp(wo, Cat).
266: dict_pos(ga, Cat):-general_pp(ga, Cat).
267: dict_pos(ni, Cat):-general_pp(ni, Cat).
268:
269: dict_pos(de,
270:         {core/{pos/adn,form/de},sc/[],
271:         ajc/[{core/{pos/n},sc/[],sem/SEM1}],
272:         ajn/[{core/{pos/v},sem/SEM2}],sem/de(SEM1,SEM2)} ).
273:
274: dict_pos(to,
275:         {core/{pos/Pos,form/to},sc/[],
276:         ajc/[{core/{pos/Cat1,form/Form},sc/[],sem/SEM1}],
277:         ajn/[{core/{pos/Cat2},sem/SEM2}], sem/SEM} );
278:         to_compl(Pos,Cat1,Form,SEM1,Cat2,SEM2,SEM).
279:
280: to_compl(adj,n,_,S1,v,S2,[with(S1)|S2]).
281: to_compl(adn,n,_,S1,n,S2,[S1,and|S2]).
282: to_compl(adj,v,Inf,S1,v,S2, [with,S1,S2] ):-sentence_end(Inf).
283: sentence_end(inf).
284: sentence_end(a_inf).
285: to_compl(adj,v,imp,S1,nil,S2,S1).
286:
```

```
287: dict_pos(no,
288:        {core/{pos/adn,form/Form},sc/[],
289:         ajc/[{core/{pos/CP,form/CF},sc/[],sem/CS}],ajn/Adjoin,sem/SEM} );
290:         no_handler(CP,CF,Form,Adjoin,CS,SEM).
291:
292: no_handler(p,Form,Form,[],Sem,Sem):-
293:     with_case(Form).
294: no_handler(n,n,no,
295:         [{core/{pos/n,form/n},ajc/[],ajn/[],sc/[],sem/inst(Obj,Type)}],
296:            Atr,inst(Obj,[rel,Atr,Type])).
297:
298: dict_pos(ha,
299:        {core/{pos/p,form/Form},ajn/[],sc/[],
300:         ajc/[{core/{pos/Cat,form/F},sc/[],sem/SEM}],sem/SEM} );
301:        wa_compl(Cat,F,Form).
302:
303: wa_compl(p,Form,Form):-with_case(Form).
304: wa_compl(n,_,Form) :- without_case(Form).
305:
306: with_case(to).
307: with_case(he).
308: with_case(ni).
309: with_case(no).
310:
311: without_case(ga).
312: without_case(wo).
313:
314: dict_pos(mo,  {core/{pos/p,form/Form},ajn/[],sc/[],
315:         ajc/[{core/{pos/Cat,form/F},sc/[],sem/SEM}],sem/SEM} );
316:                mo_compl(Cat,F,Form).
317:
318: mo_compl(p,wo,wo).
319: mo_compl(n,F,ga).
320: mo_compl(p,Form,Form):-with_case(Form).
321:
322: %% general constraint
323: v_renyou(conj).
324: v_renyou(vv).
325: v_renyou(v_y).
326: v_renyou(v_si).
327: inf_form(inf).
328: inf_form(a_inf).
329:
330: %%  Fuku Zyosi
331: %%---ba,temo,demo,te,de,tari,dari,shi
332:
333: dict_pos(ba,  {core/{pos/adv,form/katei},sc/[],
334:         ajc/[{core/{pos/v,form/katei},sc/[],sem/SEM1}],
335:         ajn/[{core/{pos/v},ajc/[],ajn/[],sem/SEM2}],
336:         sem/if(SEM1,SEM2) } ).
337:
338: %%       --temo,demo  (even if)
339: temo_demo(Form,
340:         {core/{pos/adj,form/temo},sc/[],
341:         ajc/[{core/{pos/v,form/Form},sc/[],sem/SEM1}],
342:         ajn/[{core/{pos/v},ajc/[],ajn/[],sem/SEM2}],
```

```
343:         sem/even_if(SEM1,SEM2) } ).
344: dict_pos(temo,Cat):-temo_demo(Form,Cat);te_form(Form).
345: dict_pos(demo,Cat):-temo_demo(Form,Cat);demo_form(Form).
346: demo_form(conj_de).
347: demo_form(na).
348:
349: %%   --te, de      + iru,miru, etc.
350: te_de(Form,
351:         {core/{pos/v,form/conj2},sc/[],
352:         ajc/[{core/{pos/v,form/Form},sc/[],sem/SEM}],sem/SEM} ).
353: dict_pos(te,Cat):-te_de(Form,Cat);te_form(Form).
354: dict_pos(de,Cat):-te_de(conj_de,Cat).
355: te_form(vv).
356: te_form(v_y).
357: te_form(conj_te).
358: te_form(v_si).
359:
360: %%%%    --tari,dari
361: tari_dari(Form,
362:         {core/{pos/adj,form/tari},sc/[],
363:         ajc/[{core/{pos/v,form/Form},sc/[],sem/SEM1}],
364:         ajn/[{core/{pos/vs},ajc/[],ajn/[],sem/SEM2}],
365:         sem/[SEM1|SEM2]} ).
366:
367: dict_pos(tari,Cat):-tari_dari(Form,Cat);te_form(Form).
368: dict_pos(dari,Cat):-tari_dari(conj_de,Cat).
369:
370: %%%%    --shi    (--shi,--shi)
371: dict_pos(shi,
372:        {core/{pos/adj,form/adn},sc/[],
373:         ajc/[{core/{pos/v,form/Form},sc/[],sem/SEM1}],
374:         ajn/[{core/{pos/v,form/Form},ajc/[],ajn/[],sc/[],sem/SEM2}],
375:         sem/[SEM1|SEM2] } );
376:         inf_form(Form).
377:
378: %% weak verbs
379: %%      --(te/de)iru,iku,kuru,miru,shimawu: stative verbs
380: stative_verb(F,
381:         {core/{pos/v,form/F},ajn/[],sc/[],
382:         ajc/[{core/{pos/v,form/conj2},sc/[],sem/Sem}],sem/[stative|Sem]}).
383: dict_pos(i,Cat):-stative_verb(vv,Cat).
384: dict_pos(ik,Cat):-stative_verb(vck,Cat).
385: dict_pos(k,Cat):-stative_verb(vk,Cat).
386: dict_pos(mi,Cat):-stative_verb(vv,Cat).
387: dict_pos(shimaw,Cat):-stative_verb(vcw,Cat).
388:
389: %%%%    --dasu,kakeru,hajimeru,owaru,tuzukeru :v(renyou) + v  sub-verb
390: sub_verb(F,A,
391:         {core/{pos/v,form/F},ajn/[],sc/[],
392:         ajc/[{core/{pos/v,form/Form},sc/[],sem/Sem}],sem/[A|Sem]} );
393:                 v_renyou(Form).
394: dict_pos(das,Cat):-sub_verb(vcs,begin,Cat).
395: dict_pos(kake,Cat):-sub_verb(vv,nearly,Cat).
396: dict_pos(hajime,Cat):-sub_verb(vv,begin,Cat).
397: dict_pos(owar,Cat):-sub_verb(vcr,end,Cat).
```

```
dict_pos(tuzuke,Cat):-sub_verb(vv,continue,Cat).

%%%%%%%%%%%%%   Auxiliary verbs (Zyo-doushi)    %%%%%%%%%%%%%%%%%%
%%%%%   --seru, --saseru : Causative verb (shieki)
shieki_verb(Form,
        {core/{pos/v,form/vv},ajn/[],
        ajc/[{core/{pos/v,form/Form},sc/[{core/{pos/p,form/ga},sem/Iob}],
                sem/Act}],
        sc/[{core/{pos/p,form/ga},sem/Sbj},
            {core/{pos/p,form/ni},sem/Iob}],
        sem/[cause,Sbj,Iob,Act]} ).

dict_pos(sase,Cat):-shieki_verb(Form,Cat);sara_set(Form).
dict_pos(se,Cat):-shieki_verb(Form,Cat);sere_set(Form).
sara_set(vv).
sara_set(vk).
sere_set(vc_m).
sere_set(v_sa).

%%%%%   --re(ru), --rare(ru) : Passive verb (ukemi)
dict_pos(rare,Cat):-ukemi_verb(Form,Cat);sara_set(Form).
dict_pos(re,Cat):-ukemi_verb(Form,Cat);sere_set(Form).

ukemi_verb(Form,
     {core/{pos/v,form/vv},ajn/[],
      ajc/[{core/{pos/v,form/Form},
              sc/[{core/{pos/p,form/ga},sem/Agt},
                  {core/{pos/p,form/F},sem/Pat}],
              sem/Act}],
      sc/[{core/{pos/p,form/ga},sem/Pat},{core/{pos/p,form/ni},sem/Agt}],
      sem/[passive,Pat,Agt,Act]} );
      obj_case(F).

control_passive([],Sbj,[]).
control_passive([{core/{pos/p,form/Form},ajc/[],ajn/[],sc/[],sem/Sbj}|
        Rest], Sbj,Rest):-obj_case(Form).

obj_case(ni).
obj_case(wo).

%%% general auxirialy verb (syusi,rentai form & others)
%%%   nu, ta,u,you,mai
aux_syu_ren(Form,A,
       {core/{pos/v,form/Fm},ajn/X,sc/[],sem/Z,
        ajc/[{core/{pos/v,form/Form},sc/Y,sem/Sem}]} );
                syu_ren(Fm,X,Y,Z,[A,Sem]).

aux_verb(Fm,Form,A,
        {core/{pos/v,form/Fm},ajn/[],sc/[],sem/[A,Sem],
        ajc/[{core/{pos/v,form/Form},sc/[],sem/Sem}]}).

%%% rel_clause(Form,HSC,CSC,HAdjoin,Csem,Hsem)
syu_ren(inf,[],[],Z,Z).
syu_ren(rel,[{core/{pos/n},sem/inst(A,B)}],[{core/{pos/p},sem/A}],
                inst(A,[B,Z]),Z).
```

```
%%%%%   --na(i),--nu  : Not
dict_pos(na,Cat):-aux_verb(adj,Form,no,Cat); nai_set(Form).
nai_set(vc_m).
nai_set(vv).
nai_set(vk).
nai_set(v_si).
nai_set(mizen).

dict_pos(nu,Cat):-aux_syu_ren(Form,no,Cat);nu_set(Form).
dict_pos(n,Cat):-aux_syu_ren(Form,no,Cat);nu_set(Form).
dict_pos(zu,Cat):-aux_verb(renyou,Form,no,Cat);nu_set(Form).
dict_pos(ne,Cat):-aux_verb(katei,Form,no,Cat);nu_set(Form).
nu_set(vc_m).
nu_set(vv).
nu_set(vk).
nu_set(v_se).

%%%%%   --ta,da : Past
dict_pos(ta,Cat):-aux_syu_ren(Form,past,Cat);ta_form(Form).
dict_pos(da,Cat):-aux_syu_ren(conj_de,past,Cat).
dict_pos(tara,Cat):-aux_verb(katei,Form,past,Cat);ta_form(Form).
dict_pos(dara,Cat):-aux_verb(katei,conj_de,past,Cat).

ta_form(adj_tt).
ta_form(na_tt).
ta_form(X):-te_form(X).

%%%%%   --u, --you  : suiryou (guess) or ishi(will)
dict_pos(u,Cat):-aux_syu_ren(Form,may,Cat);u_set(Form).
u_set(vc_o).
u_set(mizen).

dict_pos(you,Cat):-aux_syu_ren(Form,may,Cat);you_set(Form).
you_set(vv).
you_set(vk).
you_set(v_si).

%%%%% --rashii : suitei
dict_pos(rashi,Cat):-aux_verb(adj,Form,polite,Cat);rashii_set(Form).
rashii_set(na).
rashii_set(F):-inf_form(F).

%%%%%%  --mai  : not+guess, not+will
dict_pos(mai,Cat):-aux_syu_ren(Form,no,Cat);mai_set(Form).
mai_set(inf).
mai_set(vv).
mai_set(vk).
mai_set(v_si).

%%%%%%  --ta(i)  : hope
dict_pos(ta,Cat):-aux_verb(adj,Form,hope,Cat);v_renyou(Form).

%%%%%  --sou(da). : may & I hear
dict_pos(sou,Cat):-aux_verb(na,Form,A,Cat); souda_set(Form,A).
souda_set(F,may):-souda_may_set(F).
souda_may_set(F):-v_renyou(F).
souda_may_set(adj).
```

```
souda_may_set(na).
souda_set(inf,hear).
souda_set(a_inf,hear).

%%%%%  --desu, masu.    : teinei
desu(Inf,Cat):-aux_verb(Inf,Form,polite,Cat);desu_set(Form).
desu_set(rel).
desu_set(adj).
desu_set(na).
dict_pos(desho,Cat):-desu(vc_o,Cat).
dict_pos(deshi,Cat):-desu(v_y,Cat).
dict_pos(desu,Cat):-aux_syu_ren(Form,polite,Cat);desu_set(Form).
masu(Inf,Cat):-aux_verb(Inf,Form,polite,Cat);v_renyou(Form).
dict_pos(mase,Cat):-masu(v_se,Cat).
dict_pos(masho,Cat):-masu(vc_o,Cat).
dict_pos(mashi,Cat):-masu(v_y,Cat).
dict_pos(masu,Cat):-aux_syu_ren(Form,polite,Cat);v_renyou(Form).
dict_pos(masure,Cat):-masu(katei,Cat).
dict_pos(mase,Cat):-masu(imp,Cat).

%%%%%  --nagara, --tsutsu : cont. adverb
dict_pos(nagara,
       {core/{pos/adv,form/adv},sc/[],
        ajc/[{core/{pos/v,form/Form},sc/[],sem/SEM}],
        ajn/[{core/{pos/v1},ajc/[],ajn/[],sem/SEM1}],
        sem/[with,SEM,SEM1]} );
       nagara_set(Form).
nagara_set(a_inf).
nagara_set(F):-v_renyou(F).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      verb suffix
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%   search_suffix(Form,Word,Cat)    %%%%%%%%%%%%%%%
%%      << Example >>
%%                          mizen/  renyou/ syusi/  rentai/ katei/  imp/
%%
%% vc?  yom-mu              ma      mi      mu      mu      me      me
%%                          mo(+u)  n
%% vv   iki-ru              _       _       ru      ru      re      ro
%% vk   kuru                ko      ki      kuru    kuru    kure    koi
%% vs1  tanjyou-suru        si      si      suru    suru    sure    seyo
%%                          sa                                      siro
%%                          se
%% vs2  ai-suru             sa      si      suru    suru    sure    seyo
%% adj  haya-i              karo    katt    i       i       kere    _
%%                                  ku
%% na   kirei-da            daro    datt    da      na      nara    _
%%                                  de
%%                                  ni
%%      << Form Table >>
%% vc?:                     vc_m    conj    inf     rel     katei   imp
%% (vck,vcs,vct,vcn,        vc_o    conj_te (k,t,r,w,ik)
%%  vcr,vcw,vcg,vcb,                conj_de (g,n,m,b)
%%  vcik)
```

```
%% vv:                    vv      vv      inf     rel     katei   imp
%% vk:                    mizen   v_y     inf     rel     katei   imp
%% vs1:                   v_si    v_si    inf     rel     katei   imp
%%                        v_se
%%                        v_sa
%% vs2:                   vc_m    v_y     inf     rel     katei   imp
%% adj:                   mizen   adj_tt  a_inf    rel     katei   --
%%                                adj_ku
%% na:                    mizen   na_tt   a_inf    rel     katei   --
%%                                na_de
%%                                na_ni
search_suffix(adj,i,
      {core/{pos/v,form/Form},ajn/X,sc/[],sem/Z,
        ajc/[{core/{pos/v,form/adj},sc/Y,sem/Sem}]} );
               syu_ren(Form,X,Y,Z,[neg,Sem]).

search_suffix(na,na,
     {core/{pos/v,form/rel},sc/[],
      ajc/[{core/{pos/v,form/na},
              sc/[{core/{pos/p},ajc/[],ajn/[],sc/[],sem/Obj}],
              sem/Sem}],
      ajn/[{core/{pos/n,form/n},ajc/[],ajn/[],sc/[],sem/inst(Obj,Type)}],
        sem/inst(Obj,[and,Type,Sem])} ).

%%%     suffix search
search_suffix(Fm,Word,
        {core/{pos/v,form/Form},ajn/X,sc/[],sem/Z,
        ajc/[{core/{pos/v,form/Fm},sc/Y,sem/Sem}]} )
        :-suff_s(Word,Fm);syu_ren(Form,X,Y,Z,Sem).

search_suffix(Form2,Word,
        {core/{pos/v,form/Form1},ajn/[],sc/[],
        ajc/[{core/{pos/v,form/Form2},sc/[],sem/Sem}],sem/Sem} )
        :-suff(Word,Form1,Form2).

%%%%%   suffix  -  syusi, rentai
suff_s(su,vcs).
suff_s(nu,vcn).
suff_s(mu,vcm).
suff_s(bu,vcb).
suff_s(ku,vck).
suff_s(ku,vcik).
suff_s(tu,vct).
suff_s(u,vcw).
suff_s(ru,vcr).
suff_s(ru,vv).
suff_s(kuru,vk).
suff_s(suru,vs1).
suff_s(suru,vs2).

%%%%%    Suffix ( mizen, renyou, meirei )

%%%     adj, na
suff(karo,mizen,adj).
suff(katt,adj_tt,adj).
suff(ku,adj_ku,adj).
suff(daro,mizen,na).
```

```
624: suff(datt,na_tt,na).
625: suff(de,na_de,na).
626: suff(ni,na_ni,na).
627: suff(da,a_inf,na).
628:
629: %%%     vs, vk  - mizen, renyou
630: suff(se,v_se,vs1).
631: suff(si,v_si,vs1).
632: suff(sa,v_sa,vs1).
633: suff(si,v_y,vs2).
634: suff(ko,mizen,vk).
635: suff(ki,v_y,vk).
636:
637: %%%     v5 mizen
638: suff(Suf,vc_m,Inf):-vc_m_suff(Suf,Inf).
639: vc_m_suff(sa,vcs).
640: vc_m_suff(sa,vs2).
641: vc_m_suff(na,vcn).
642: vc_m_suff(ma,vcm).
643: vc_m_suff(ba,vcb).
644: vc_m_suff(ka,vck).
645: vc_m_suff(ka,vcik).
646: vc_m_suff(ta,vct).
647: vc_m_suff(wa,vcw).
648: vc_m_suff(ra,vcr).
649:
650: suff(Suf,vc_o,Inf):-vc_o_suff(Suf,Inf).
651: vc_o_suff(so,vcs).
652: vc_o_suff(so,vs2).
653: vc_o_suff(no,vcn).
654: vc_o_suff(mo,vcm).
655: vc_o_suff(bo,vcb).
656: vc_o_suff(ko,vck).
657: vc_o_suff(ko,vcik).
658: vc_o_suff(to,vct).
659: vc_o_suff(wo,vcw).
660: vc_o_suff(ro,vcr).
661:
662: %%%     v5 renyou
663: suff(si,v_y,vcs).
664:
665: suff(Suf,conj,Inf):-vc_conj_suff(Suf,Inf).
666: vc_conj_suff(ni,vcn).
667: vc_conj_suff(mi,vcm).
668: vc_conj_suff(bi,vcb).
669: vc_conj_suff(wi,vcw).
670: vc_conj_suff(gi,vcg).
671: vc_conj_suff(ki,vck).
672: vc_conj_suff(ki,vcik).
673: vc_conj_suff(ti,vct).
674: vc_conj_suff(ri,vcr).
675:
676: %%%     v5 renyou - onbin
677: suff(i,conj_te,vck).
678: suff(i,conj_de,vcg).
679: suff(t,conj_te,vct).
680: suff(t,conj_te,vcw).
681: suff(n,conj_de,vcb).
```

```
682: suff(n,conj_de,vcm).
683: suff(t,conj_te,vcr).
684: suff(n,conj_de,vcn).
685: suff(t,conj_te,vcik).
686:
687: %%%     katei  ( only  -ba)
688: suff(Suf,katei,Inf):-suff_ba(Suf,Inf).
689: suff_ba(se,vcs).
690: suff_ba(ne,vcn).
691: suff_ba(me,vcm).
692: suff_ba(be,vcb).
693: suff_ba(ke,vck).
694: suff_ba(ke,vcik).
695: suff_ba(te,vct).
696: suff_ba(we,vcw).
697: suff_ba(re,vcr).
698: suff_ba(re,vv).
699: suff_ba(kure,vk).
700: suff_ba(sure,vs1).
701: suff_ba(sure,vs2).
702: suff_ba(kere,adj).
703: suff_ba(nara,na).
704:
705:
706: %%%     meirei
707: suff(Suf,imp,Inf):-imp_suff(Suf,Inf).
708: imp_suff(se,vcs).
709: imp_suff(ne,vcn).
710: imp_suff(me,vcm).
711: imp_suff(be,vcb).
712: imp_suff(ke,vck).
713: imp_suff(ke,vcik).
714: imp_suff(te,vct).
715: imp_suff(we,vcw).
716: imp_suff(re,vcr).
717: imp_suff(ro,vv).
718: imp_suff(koi,vk).
719: imp_suff(siro,vs1).
720: imp_suff(sero,vs1).
721: imp_suff(seyo,vs2).
722: %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
723: %% noun.dic
724: %% common noun, proper nouns
725: %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
726:
727: %%%%%%% common nouns.
728: c_noun(Sem,{core/{pos/n,form/n},sem/inst(Obj,Sem)}).
729:
730: dict1(ari,Cat):-c_noun(ant,Cat).
731: dict1(esa,Cat):-c_noun(food,Cat).
732: dict1(gakusha,Cat):-c_noun(scholar,Cat).
733: dict1(gyouretsu,Cat):-c_noun(row,Cat).
734: dict1(hana,Cat):-c_noun(flower,Cat).
735: dict1(hatarakiari,Cat):-c_noun(worker_ant,Cat).
736: dict1(hito,Cat):-c_noun(person,Cat).
737: dict1(hon,Cat):-c_noun(book,Cat).
738: dict1(ishi,Cat):-c_noun(stone,Cat).
```

```
739: dict1(jimen,Cat):-c_noun(ground,Cat).
740: dict1(katamari,Cat):-c_noun(block,Cat).
741: dict1(michi,Cat):-c_noun(road,Cat).
742: dict1(michishirube,Cat):-c_noun(row,Cat).
743: dict1(michisuji,Cat):-c_noun(road,Cat).
744: dict1(mokutekichi,Cat):-c_noun(goal,Cat).
745: dict1(natsu,Cat):-c_noun(summer,Cat).
746: dict1(niwa,Cat):-c_noun(garden,Cat).
747: dict1(satou,Cat):-c_noun(sugar,Cat).
748: dict1(soto,Cat):-c_noun(out,Cat).
749: dict1(sumi,Cat):-c_noun(corner,Cat).
750: dict1(tsubu,Cat):-c_noun(grain,Cat).
751: dict1(yousu,Cat):-c_noun(yousu,Cat).
752: dict1(yukute,Cat):-c_noun(way,Cat).
753:
754: %%%%%%% proper nouns.
755: p_noun(Sem,{core/{pos/n,form/n},sem/Sem}).
756:
757: dict1(amerika,Cat):-p_noun(america,Cat).
758: dict1(hiroshi,Cat):-p_noun(hiroshi,Cat).
759: dict1(ken,Cat):-p_noun(ken,Cat).
760: dict1(naomi,Cat):-p_noun(naomi,Cat).
761: dict1(wilson,Cat):-p_noun(wilson,Cat).
762:
763: %%%%% jibun (self)
764: dict1(jibun, {core/{pos/n,form/n},
765:         refl/[{core/{pos/p,form/ga},sem/Sem}],
766:         sem/Sem}).
767: dict1(jken, {core/{pos/n,form/n},
768:         refl/[{core/{pos/p,form/ga},sem/ken}],
769:         sem/ken}).
770:
771: %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
772: %%  verb.dic
773: %%  Verbs except sahen-v
774: %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
775: %% vi (|subcat|==1: --ga)
776: ga_verb(F,Act,
777:         {core/{pos/v,form/F},
778:         sc/[{core/{pos/p,form/ga},sem/Sbj}],
779:         sem/[Act,Sbj]}).
780:
781: %%%%% vt (|subcat|==2:)
782: %%  --ga --wo
783: ga_wo_verb(F,Act,
784:         {core/{pos/v,form/F},
785:         sc/[{core/{pos/p,form/ga},sem/Sbj},
786:             {core/{pos/p,form/wo},sem/Obj}],
787:         sem/[Act,Sbj,Obj]}).
788: %%  --ga --ni
789: ga_ni_verb(F,Act,
790:         {core/{pos/v,form/F},
791:         sc/[{core/{pos/p,form/ga},sem/Sbj},
792:             {core/{pos/p,form/ni},sem/Obj}],
793:         sem/[Act,Sbj,Obj]}).
```

```
794:
795: %%  --ga --wo --ni
796: ga_wo_ni_verb(F,Act,
797:         {core/{pos/v,form/F},
798:         sc/[{core/{pos/p,form/ga},sem/Sbj},
799:             {core/{pos/p,form/wo},sem/Iob},
800:             {core/{pos/p,form/ni},sem/Dob}],
801:         sem/[Act,Sbj,Iob,Dob]}).
802:
803: %% temp. feature
804: %% kiru,akeru
805: temp1({core/{view/asp(AB,AE,AD,AT)}});
806:         temp_cstr(3,2,2,f,f,AB,AE,AD,AT).
807: %% anki-suru
808: temp2({core/{view/asp(AB,AE,AD,AT)}});
809:         temp_cstr(3,2,0,f,u,AB,AE,AD,AT).
810: %% aruku,yomu
811: temp3({core/{view/asp(AB,AE,AD,AT)}});
812:         temp_cstr(3,2,2,f,0,AB,AE,AD,AT).
813: %% matu,damaru
814: temp4({core/{view/asp(AB,AE,AD,AT)}});
815:         temp_cstr(3,1,1,f,0,AB,AE,AD,AT).
816: %% suwaru,kekkon-suru
817: temp5({core/{view/asp(AB,AE,AD,AT)}});
818:         temp_cstr(3,2,2,0,f,AB,AE,AD,AT).
819: %% suwaru,sinu
820: temp6({core/{view/asp(AB,AE,AD,AT)}});
821:         temp_cstr(3,2,0,0,f,AB,AE,AD,AT).
822: %% niru,tadayou
823: temp7({core/{view/asp(AB,AE,AD,AT)}});
824:         temp_cstr(0,0,0,0,u,AB,AE,AD,AT).
825: %% odoroku,tumaduku
826: temp8({core/{view/asp(AB,AE,AD,AT)}});
827:         temp_cstr(3,2,2,0,0,AB,AE,AD,AT).
828:
829: %%% lexical entry
830: dict1(age,Cat):-ga_wo_ni_verb(vv,give,Cat).
831: dict1(ai,Cat):-ga_wo_verb(vs2,love,Cat).
832: dict1(ake,Cat):-ga_wo_verb(vv,open,Cat),temp1(Cat).
833: dict1(aruk,Cat):-ga_verb(vck,walk,Cat).
834: dict1(chigaw,Cat):-ga_verb(vcw,differ,Cat).
835: dict1(chirijirininar,Cat):-ga_verb(vcr,scatter,Cat).
836: dict1(deki,Cat):-ga_verb(vv,can,Cat).
837: dict1(de,Cat):-ga_ni_verb(vv,go_out,Cat).
838: dict1(hashir,Cat):-ga_verb(vcr,run,Cat).
839: dict1(hazure,Cat):-ga_wo_verb(vv,be_off,Cat).
840: dict1(i,Cat):-ga_verb(vv,be,Cat).
841: dict1(ik,Cat):-ga_ni_verb(vck,go_to,Cat).
842: dict1(isog,Cat):-ga_verb(vcg,hurry,Cat).
843: dict1(kaer,Cat):-ga_ni_verb(vcr,return,Cat).
844: dict1(kag,Cat):-ga_wo_verb(vcg,smell_of,Cat).
845: dict1(kak,Cat):-ga_wo_verb(vck,write,Cat).
846: dict1(kaw,Cat):-ga_wo_verb(vcw,buy,Cat).
```

```
847: dict1(kawar,Cat):-ga_verb(vcr,lose,Cat).
848: dict1(kat,Cat):-ga_ni_verb(vct,win,Cat).
849: dict1(ke,Cat):-ga_wo_verb(vv,kick,Cat).
850: dict1(ki,Cat):-ga_wo(vv,wear,Cat),temp1(Cat).
851: dict1(majiwar,Cat):-ga_verb(vcr,cross,Cat).
852: dict1(manab,Cat):-ga_wo_verb(vcb,learn,Cat).
853: dict1(mayow,Cat):-ga_verb(vcw,be_lost,Cat).
854: dict1(mi,Cat):-ga_wo_verb(vv,see,Cat).
855: dict1(midare,Cat):-ga_verb(vv,be_confused,Cat).
856: dict1(mitsuke,Cat):-ga_wo_verb(vv,find,Cat).
857: dict1(mot,Cat):-ga_wo_verb(vct,have,Cat).
858: dict1(na,Cat):-ga_ni_verb(vv,become,Cat).
859: dict1(nor,Cat):-ga_ni_verb(vcr,get_on,Cat).
860: dict1(ok,Cat):-ga_wo_ni_verb(vck,put,Cat).
861: dict1(omow,Cat):-ga_wo_verb(vcw,think,Cat).
862: dict1(os,Cat):-ga_wo_verb(vcs,push,Cat).
863: dict1(saegir,Cat):-ga_wo_verb(vcr,interrupt,Cat).
864: dict1(sagas,Cat):-ga_wo_verb(vcs,seek,Cat).
865: dict1(shir,Cat):-ga_wo_verb(vcr,know,Cat).
866: dict1(shin,Cat):-ga_verb(vcn,die,Cat).
867: dict1(susum,Cat):-ga_ni_verb(vcm,advance,Cat).
868: dict1(tador,Cat):-ga_wo_verb(vcr,follow,Cat).
869: dict1(tasuke,Cat):-ga_wo_verb(vv,help,Cat).
870: dict1(toor,Cat):-ga_ni_verb(vcr,pass,Cat).
871: dict1(tsuge,Cat):-ga_wo_ni_verb(vv,tell,Cat).
872: dict1(tsuk,Cat):-ga_ni_verb(vck,reach,Cat).
873: dict1(tsuzuku,Cat):-ga_verb(vck,continue,Cat).
874: dict1(wakar,Cat):-ga_wo_verb(vcr,understand,Cat).
875: dict1(yom,Cat):-ga_wo_verb(vcm,read,Cat).
876: %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
877: %%   sahen.dic
878: %%   sahen-n,v dictionary
879: %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
880:
881: %%%%%   verb : SURU
882: suru_verb(F, {core/{pos/vs,form/F},ajc/Adj,sc/SC,sem/Sem});
883:         suru(Adj,SC,Sem).
884:
885: dict1(shi,Cat):-suru_verb(v_si,Cat).
886: dict1(se,Cat):-suru_verb(v_se,Cat).
887: dict1(sa,Cat):-suru_verb(v_sa,Cat).
888: dict1(sure,Cat):-suru_verb(katei,Cat).
889: dict1(shiro,Cat):-suru_verb(imp,Cat).
890: dict1(sero,Cat):-suru_verb(imp,Cat).
891:
892: %%%%%   sa-hen verbs (do)
893: sahen_verb(F,Act,
894:         {core/{pos/v,form/F},sc/[{core/{pos/p,form/ga},sem/Sbj}],
895:         sem/[Act,Sbj]} ).
896: dict1(tanjou,Cat):-sahen_verb(vs1,be_born,Cat).
897:
898: %%%%%   sa-hen nouns.
899: dict1(chousa,
900:         {core/{pos/n,form/ns},sc/SC,sem/[investigate,Sbj,Obj]} );
901:         adn_wo_ga(SC,Obj,Sbj).
```

```
902:
903: %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
904: %%   adject.dic
905: %%   adjective, adjective-verb
906: %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
907:
908: %%%%% Adjectives
909: adjective(A,
910:         {core/{pos/v,form/adj}, sc/[{core/{pos/p,form/ga},sem/Obj}],
911:         sem/[A,Obj]} ).
912: dict1(aka,Cat):-adjective(red,Cat).
913: dict1(siro,Cat):-adjective(white,Cat).
914: dict1(kuro,Cat):-adjective(black,Cat).
915: dict1(ooki,Cat):-adjective(big,Cat).
916: dict1(yo,Cat):-adjective(good,Cat).
917:
918: %%%%%   na (adjective-verb)
919: ajverb(A,
920:       {core/{pos/v,form/na}, sc/[{core/{pos/p,form/ga},sem/Obj}],
921:       sem/[A,Obj]} ).
922: dict1(kirei,Cat):-ajverb(beautifle,Cat).
923: dict1(kaiteki,Cat):-ajverb(comfortable,Cat).
924: %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
925: %%     dictionary of misc. words
926: %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
927:
928: %%%%%%%% rentai-shi
929: rentaishi(A,
930:         {core/{pos/adn, form/adn},
931:         ajn/[{core/{pos/n,form/n},ajc/[],ajn/[],sc/[],sem/SEM}],
932:         sem/[A|[SEM]]} ).
933: dict1(sono,Cat):-rentaishi(the,Cat).
934: dict1(kono,Cat):-rentaishi(this,Cat).
935: dict1(ano,Cat):-rentaishi(that,Cat).
936: dict1(ippikino,Cat):-rentaishi(one,Cat).
937:
938: %%%%%%% fuku-shi (adverb)
939:
940: adverb(MODIFY,
941:         {core/{pos/adv, form/adv}, ajn/[{core/{pos/v},sem/SEM}],
942:         sem/[MODIFY|SEM]}).
943: dict1(yoku,Cat):-adverb(often,Cat).
944: dict1(zutto,Cat):-adverb(continue,Cat).
945: dict1(hajimeni,Cat):-adverb(first,Cat).
946: dict1(sukoshi,Cat):-adverb(slightly,Cat).
947: dict1(yagate,Cat):-adverb(in_the_end,Cat).
948: dict1(tsugitsugito,Cat):-adverb(continue,Cat).
949: dict1(dandanni,Cat):-adverb(gradually,Cat).
950: dict1(komakani,Cat):-adverb(minutely,Cat).
951: dict1(kesshite,
952:      {core/{pos/adv,form/adv}, ajn/[{core/{pos/v},sem/[no|SEM]}],
953:       sem/[never|SEM]}).
954: %%%%%  Examples  %%%%%%%%%%%%%%%%
955: %% ?-p([ken,ga,naomi,wo,ai,suru]).
956: %% ?-p([ken,ga,naomi,ni,ai,sa,re,ta]).
```

Bibliography

[ABK89] Krzysztof R. Apt, Roland N. Bol, and Jan Willem Klop. On the safe termination of PROLOG programs. In *Proc. of 6th International Conference of Logic Programming*, pages 353–368, 1989.

[Acz88] Peter Aczel. *Non-Well Founded Set Theory.* Lecture Notes No. 14, Stanford:CSLI, 1988.

[AK89] S. Abiteboul and P. Kanellakis. Object Identity as a Query Language Primitive. In *Proc. ACM SIGMOD International Conference on Management of Data*, Portland, June 1989.

[AKN86] Hassan Aït-Kaci and Roger Nasr. LOGIN: A Logic Programming Language with Built-In Inheritance. *Journal of Logic Programming*, 3:185–215, 1986.

[ASS+88] Akira Aiba, K. Sakai, Y. Sato, D. J. Hawley, and R. Hasegawa. Constraint Logic Programming Language CAL. In *Proc. FGCS88*, 1988.

[AYT94] Akira Aiba, Kazumasa Yokota, and Hiroshi Tsuda. Heterogeneous Distributed Cooperative Problem Solving System HELIOS. In *Proc. FGCS94*, 1994.

[Bes89] Philippe Besnard. On Infinite Loops in Logic Programming. Rapports de Recherche N 1096, INRIA, September 1989.

[BMMW89] Alan Borning, Michael Maher, Amy Martindale, and Molly Wilson. Constraint Hierarchies and Logic Programming. In *Proc. 6th International Conference of Logic Programming*, pages 149–164, 1989.

[BP83] Jon Barwise and John Perry. *Situation and Attitudes.* MIT Press, Cambridge, Mass, 1983.

[Bre82] Joan W. Bresnan, editor. *The Mental Representation of Grammatical Relations.* MIT Press, Cambridge, Mass, 1982.

[Bru82] Maurice Bruynooghe. Analysis of Dependencies to Improve the Behaviour of Logic Programming. In D. W. Loveland, editor, *6th Conference on Automated Deduction*, pages 293–305, New York, June 1982. Springer-Verlag.

[Car92a] Bob Carpenter. ALE - The Attribute Logic Engine User's Guide. Anonymous FTP (CMU), December 1992.

[Car92b] Bob Carpenter. *The Logic of Typed Feature Structure.* Cambridge University Press, 1992.

[Cho81] Norm Chomsky. *Lectures on Government and Binding.* Foris, Dordrecht, 1981.

[Cho95] Norm Chomsky. *The Minimalist Program.* MIT Press, 1995.

[CM85] E. Charniak and D. McDermott. *Introduction to Artificial Intelligence.* Addison-Wesley, 1985.

[Col87] A. Colmerauer. Prolog III. *BYTE*, August 1987.

[Cry97] David Crystal. *A Dictionary of Linguistics and Phonetics.* Blackwell, 4th edition, 1997.

[DE90] Jochen Dörre and Andreas Eisele. Feature Logic with Disjunctive Unification. In *COLING-90 Vol.2*, pages 100–105, August 1990.

[Deb89] Saumya K. Debray. Static Inference of Modes and Data Dependencies in Logic Programs. *ACM Transaction on Programming Language and Systems*, 11(3):418–450, July 1989.

[DKM91] C. Delobel, M. Kifer, and Y. Masunaga, editors. *Deductive and Object-Oriented Databases (Proc. Second International Conference on Deductive and Object-Oriented Databases (DOOD'91))*, volume 566 of *LNCS*. Springer, 1991.

[DSG92] Walter Daelemans, Koenraad De Smedt, and Gerald Gazdar. Inheritance in Natural Language Processing. *Computational Linguistics*, 18(2):205–218, 1992.

[Ear70] J Earley. An Efficient Context-Free Parsing Algorithm. *Communications of ACM*, 13:94–102, 1970.

[ED88] Andreas Eisele and Jochen Dörre. Unification of Disjunctive Feature Descriptions. In *26th ACL Annual Meeting*, pages 286–294, June 1988.

[GKPS85] Gerald Gazdar, Ewan Klein, Geoffrey K. Pullum, and Ivan A. Sag. *Generalized Phrase Structure Grammar.* Basil Blackwell, England:Oxford, 1985.

[Gun87] Takao Gunji. *Japanese Phrase Structure Grammar.* Reidel, Dordrecht, 1987.

[Gun95] Takao Gunji. An Overview of JPSG: A Constraint-Based Grammar for Japanese. In R. Mazuka and N. Nagai, editors, *Japanese Sentence Processing, Proc. International Symposium on Japanese Syntactic Processing*, chapter 5. Lawrence Erlbaum, 1995.

[GUN96] Takao GUNJI, editor. *Studies on the Universality of Constraint-Based Phrase Structure Grammar.* Report of the International Scientific Research Program, Joint Research, Project No. 06044133, Supported by the Ministry of Education, Science, and Culture, Japan. Osaka University, March 1996.

[Har91] Yasunari Harada. "No" ni tuite no zyakkan no kansatu. *Waseda Daigaku, Gogaku Kenkyusyo Kiyo 42*, 1991. (in Japanese).

[Has94] Ryuzo Hasegawa, Parallel Theorem-Provins System: MGTP. In *Proc. FGCS94*, 1994.

[Has85] Kôiti Hasida. *Bounded Parallelism: A Theory of Linguistic Performance.* PhD thesis, Department of Information Science, University of Tokyo, 1985.

[Has86] Kôiti Hasida. Conditioned Unification for Natural Language Processing. In *11th International Conference on Computational Linguistics*, pages 85–87, 1986.

[Has90] Kôiti Hasida. Sentence Processing as Constraint Transformation. In *Proc. ECAI'90*, 1990.

[Has91] Kôiti Hasida. Common Heuristics for Parsing, Generation, and Whatever. In Tomek Strzalkowski, editor, *Reversible Grammar in Natural Language Processing*. Kluwer Academic Publishers, 1991.

[HB90] Jerry R. Hobbs and John Bear. Two Principles of Parse Preference. In *COLING Vol.3*, pages 162–167, August 1990.

[HI87] Kôiti Hasida and Shun ISIZAKI. Dependency Propagation: A Unified Theory of Sentence Comprihension and Generation. In *IJCAI*, 1987.

[HNM93] Kôiti Hasida, Katashi Nagao, and Takashi Miyata. Joint Utterance: Intrasentential Speaker/Hearer Switch as an Emergent Phenomenon. In *IJCAI93*, pages 1193–1199, Chambery, 1993.

[HS86] Kôiti Hasida and Hidetosi Sirai. Zyoukentuki Tan'itu-ka (Conditioned Unification). *Computer Software*, 3(4):28–38, 1986. (in Japanese).

[HSME88] Jerry R. Hobbs, Mark Stickel, Paul Martin, and Douglas Edwards. Interpretation as Abduction. In *26th ACL Annual Meeting*, pages 95–103, 1988. (also in Artificial Intelligence, Vol.63, No.1-2, 1993).

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint Logic Programming. In *Proc. 14th ACM POPL Conference*, pages 111–119, Munich, 1987.

[Kas87] Robert T. Kasper. A Unification Method for Disjunctive Feature Descriptions. In *25th ACL Annual Meeting*, pages 235–242, July 1987.

[Kas88] Robert T. Kasper. Conditional Descriptions in Functional Unification Grammar. In *26th ACL Annual Meeting*, pages 233–240, June 1988.

[Kay85] Martin Kay. Parsing in Functional Unification Grammar. In David R. Dowty, Lauri Karttunen, and Arnold M. Zwicky, editors, *Natural language parsing*, chapter 7, pages 251–278. Cambridge university press, 1985.

[Kel93] Bill Keller. *Feature Logics, Infinitary Descriptions and Grammar*. Lecture Notes No. 44, Stanford:CSLI, 1993.

[Kif90] Michael Kifer. Logical Foundation of Object-Oriented and Frame-Based Language. Technical Report 90/14, State University of New York at Stony Brook, June 1990.

[Kle84] Johan de Kleer. Choices Without Backtracking. In *Proc. of AAAI*, pages 79–85, 1984.

[KNN89] W. Kim, J.-M. Nicolas, and S. Nishio, editors. *Deductive and Object-Oriented Databases (Proc. 1st Int. Conf. on Deductive and Object-Oriented Data-bases (DOOD89))*. North-Holland, 1989.

[KR86] Robert T. Kasper and William C. Rounds. A Logical Semantics for Feature Structure. In *Proc. 24th ACL Annual Meeting*, pages 257–266, 1986.

[KR90] Robert T. Kasper and William C. Rounds. The Logic of Unification in Grammar. *Linguistics and Philosophy*, 13(1):35–58, 1990.

[Llo84] John W. Lloyd, Foundations of Logic Programming. Springer-Verlag, 1984.

[MK92] John T. Maxwell and Ronald M. Kaplan. The Interface between Phrasal and Functional Constraint. *Proc. Workshop of ECAI92*, 1992.

[Mar80] Mitchell P. Marcus. *A Theory of Syntactic Recognition for Natural Language*. MIT Press, Cambridge:Mass, 1980.

[MS89] Shaul Markovitch and Paul D. Scott. Automatic Ordering of Subgoals – a Machine Learning Approach. In *Proc. of the North American Conference of Logic Programming*, pages 3–19. MIT, 1989.

[MGS+86] Hideo Miyoshi, Takao Gunji, Hidetosi Sirai, Kôiti Hasida, and Yasunari Harada. Nihongo Kukouzou Bunpou : JPSG(Japanese Phrase Structure Grammar:JPSG). *Computer Software*, 3(4):39–45, 1986. (in Japanese).

[MHY90] Y. Morita, H. Haniuda, and K. Yokota. Object Identity in Quixote. In *Proc. SIGDBS and SIGAI of IPSJ*, Oct 1990. (in Japanese).

[Muk88] Kuniaki Mukai. Partially Specified Term in Logic Programming for Linguistic Analysis. In *Proc. International Conference of Fifth Generation Computer Systems*, pages 479–488. ICOT, OHMSHA, Springer-Verlag, 1988.

[Muk90] K. Mukai. CLP(AFA): Coinductive Semantics of Horn Clauses with Compact Constraint. In *2nd Conf. on Situation Theory and Its Applications*. Kinloch Rannoch Scotland, Sep 1990.

[Muk91] Kuniaki Mukai. *Constraint Logic Programming and the Unification of Information*. PhD thesis, Tokyo Institute of Technology, 1991.

[MY85] Kuniaki Mukai and Hideki Yasukawa. Complex Indeterminates in Prolog and its Application to Discourse Models. *New Generation Computing*, 3(4):441–466, 1985.

[NOTY93] Toshihiro Nishioka, Ryo Ojima, Hiroshi Tsuda, and Kazumasa Yokota. Procedural Semantics of a DOOD Programming Language Quixote. In *SIG-DBS No.94*, pages 1–10, Nagasaki, 1993. Inf. Proc. Soc. Japan. (in Japanese).

[Nak91] Mikio Nakano, Constraint Projection: An Efficient Treatment of Disjunctive Feature Descriptions In *Proc. of 29th ACL Annual Meeting*, pages 307–314, 1991.

[NPS91] H. Nakashima, S. Peters, and H. Schutze. Communication and inference through situations. *Proc. IJCAI'91*, pages 76–81, 1991.

[NSHP88] H. Nakashima, H. Suzuki, P. Halvorsen, and S. Peters. Towards a computational interpretation of situation theory. In *Proc. FGCS88*, pages 489–498, 1988.

[NTY94] Y. Niibe, C. Takahashi, and K. Yokota. Design and Implementation of micro-Quixote and Its Extension Function. In *Proc. Joint Workshop of SIGDBS of IPSJ and SIGDE of IEICE*, pages 139–146, July 1994. (in Japanese).

[PS87a] Fernando C. N. Pereira and Stuart M. Shieber. *Prolog and Natural-Language Analysis*. Lecture Notes No. 10, Stanford:CSLI, 1987.

[PS87b] Carl Pollard and Ivan A. Sag. *Information-Based Syntax and Semantics, Vol.1 Fundamentals*. Lecture Notes No. 13, Stanford:CSLI, 1987.

[PS94] Carl Pollard and Ivan A. Sag. *Head-Driven Phrase Structure Grammar*. University of Chicago Press and CSLI Publications, 1994.

[PW80] Fernando C. N. Pereira and David H. D. Warren. Definite Clause Grammar for Language Analysis. *Artificial Intelligence*, 13:231–278, 1980.

[PW83] Fernando C. N. Pereira and David H. D. Warren. Parsing as Deduction. *Proc. ACL'83*, pages 137–144, 1983.

[Shi86] Stuart M. Shieber. *An Introduction to Unification-Based Approach to Grammar*. Lecture Notes No. 4, Stanford:CSLI, 1986.

[Shi88] Stuart M. Shieber. A Uniform Architecture for Parsing and Generation. In *12th International Conference on Computational Linguistics*, pages 614–619, 1988.

[Shi91] Stuart M. Shieber. Constraints and natural-language analysis. Tutorial in International Logic Programming Symposium, October 1991.

[Shi92] Stuart M. Shieber. *Constraint-Based Grammar Formalisms*. MIT Press, A Bradford Book, 1992.

[Sir91] Hidetosi Sirai. A Guide to MacCUP. unpublished, 1991. (available by anonymous FTP from csli.stanford.edu (pub/MacCup)).

[Smo88] Gert Smolka. A Feature Logic with Subsorts. LILOG Report 33, IBM Deutschland, Stuttgart, West Germany, May 1988.

[Smo92] Gert Smolka. Feature Constraint Logics for Unification Grammars. *Journal of Logic Programming*, 12(1 and 2):51–87, 1992.

[SNN86] Akira Shimazu, Shozo Naito, and Hirosato Nomura. Analysis of semantic relations between noun connected by a Japanese particle "no". *Mathematical Linguistics*, 15(7):247–266, 1986. (in Japanese).

[TA94] Hiroshi Tsuda and Akira Aiba. Heterogeneous Natural Language Understanding in Helios. In *FGCS94 Workshop on Heterogeneous Cooperative Knowledge-Bases*. ICOT, Dec. 1994.

[TA96] Hiroshi Tsuda and Akira Aiba. Heterogeneous Natural Language Understanding in Helios. *Computer Software*, 13(6):43-52, 1996. (In Japanese).

[TH90] Hiroshi Tsuda and Kôiti Hasida. Parsing as Constraint Transformation – an Extension of cu-Prolog. In *Seoul International Conference on Natural Language Processing*, pages 325–331, 1990.

[TH96] Hiroshi Tsuda and Yasunari Harada. Semantics and Pragmatics of Adnominal Particle NO in Quixote. In Takao GUNJI, editor, *Studies on the Universality of Constraint-Based Phrase Structure Grammar*, pages 191–201. Osaka University, March 1996. Report of the International Scientific Research Pro-

gram, Joint Research, Project No. 06044133, Supported by the Ministry of Education, Science, and Culture, Japan.

[THS89] Hiroshi Tsuda, Kôiti Hasida, and Hidetosi Sirai. JPSG Parser on Constraint Logic Programming. In *4th ACL European Chapter*, pages 95–102, 1989.

[THS90] Hiroshi Tsuda, Kôiti Hasida, and Hidetosi Sirai. cu-Prolog and its application to a JPSG parser. In K.Furukawa, H.Tanaka, and T.Fujisaki, editors, *Logic Programming '89*, pages 134–143. Springer-Verlag LNAI-485, 1990.

[TOM86] Masaru TOMITA. *Efficient Parsing for Natural Language*. Kluwer Academic Press, 1986.

[Tom92] Yutaka Tomioka. Computability of Modularization of Constraints. *Computer Software*, 9(6):58–68, 1992. (in Japanese).

[TS83] Hisao Tamaki and Taisuke Sato. Unfold/Fold Transformation of Logic Programs. In *Proc. Second International Conference on Logic Programming*, pages 127–137, 1983.

[TS86] Hisao Tamaki and Taisuke Sato. OLD Resolution with Tabulation. In *Proc. Third International Conference on Logic Programming*, pages 84–98, 1986.

[Tsu89] Hirosi Tsuda. A JPSG Parser in Constraint Logic Programming. Master's thesis, Department of Information Science, University of Tokyo, 1989.

[Tsu91] Hiroshi Tsuda. Disjunctive Feature Structure in cu-Prolog. In *8th Conf. Proc. Japan Soc. Softw. Sc. Japan.*, pages 505–508, 1991. (in Japanese).

[Tsu92] Hiroshi Tsuda. cu-Prolog for Constraint-Based Grammar. In *Proc. FGCS92*, pages 347–356, June 1992.

[Tsu94] Hiroshi Tsuda. cu-Prolog for Constraint-Based Natural Language Processing. *IEICE Transactions on Information and Systems*, E77-D(2):171–180, February 1994.

[TTY+93] Satoshi Tojo, Hiroshi Tsuda, Hideki Yasukawa, Kazumasa Yokota, and Yukihiro Morita. Quixote as a Tool for Natural Language Processing. In *TAI93*, pages 266–270, Boston, 1993. IEEE.

[TTY+94] Satoshi Tojo, Hiroshi Tsuda, Hideki Yasukawa, Kazumasa Yokota, and Yukihiro Morita. Quixote: A Framework for Linguistic Information Processing. *Journal of Japan Society of Artificial Intelligence*, 9(6):863–874, 1994. (in Japanese).

[TY94] Hiroshi Tsuda and Kazumasa Yokota. Knowledge Representation Language Quixote. In *Proc. FGCS94*, June 1994.

[YNT+94] Kazumasa Yokota, Toshihiro Nishioka, Hiroshi Tsuda, , and Satoshi Tojo. Query Processing for Partial Information Databases in Quixote. In *6th IEEE International Conference on Tools with Artificial Intelligence*, New Orleans, Nov. 6-9 1994.

[Yok94] Kazumasa Yokota. *Quixote: A Constraint Based Approach to a Deductive Object-Oriented Database*. PhD thesis, Kyoto University, 1994.

[YTM93] Kazumasa Yokota, Hiroshi Tsuda, and Yukihiro Morita. Specific Features of a Deductive Object-Oriented Database Language Quixote. In *Proc. ACM SIGMOD Workshop on Combining Declarative and Object-Oriented Databases*, Washington DC, USA, May 29 1993.

[YTY92] Hideki Yasukawa, Hiroshi Tsuda, and Kazumasa Yokota. Objects, Properties, and Modules in Quixote. In *Proc. FGCS92*, pages 257–268, 1992.

[YY90] Hideki Yasukawa and Kazumasa Yokota. Labeled Graphs as Semantics of Objects. In *Proc. SIGDBS and SIGAI of IPSJ*, October 1990.

[YY92] K. Yokota and H. Yasukawa. Towards an Integrated Knowledge-Base Management System. In *Proc. FGCS92*, volume 1, pages 89–112. Institute for New Generation Computer Technology, 1992.

Index

Publication List

(Papers with * are included in this thesis.)

- (*) Hiroshi Tsuda, Kôiti Hasida, and Hidetosi Sirai. JPSG Parser on Constraint Logic Programming, 4th ACL European Chapter, pages 95–102, 1989.
- Hiroshi Tsuda, Kôiti Hasida, and Hidetosi Sirai. Parsing as Constraint Satisfaction — an Application of cu-Prolog 6th Conf. Proc. Japan Soc. Softw. Sc. Japan., pages 257–260, 1989.(in Japanese)
- (*) Hiroshi Tsuda, Kôiti Hasida, and Hidetosi Sirai. cu-Prolog and its application to a JPSG parser. In K.Furukawa, H.Tanaka, and T.Fujisaki, editors, *Logic Programming '89*, pages 134–143. Springer-Verlag LNAI-485, 1990.
- (*) Hiroshi Tsuda and Kôiti Hasida., Parsing as Constraint Transformation – an Extension of cu-Prolog, Seoul International Conference on Natural Language Processing, pages 325–331, 1990.
- Hiroshi Tsuda. Disjunctive Feature Structure in cu-Prolog, 8th Conf. Proc. Japan Soc. Softw. Sc. Japan., pages 505–508, 1991.(in Japanese)
- (*) Hiroshi Tsuda. cu-Prolog for Constraint-Based Grammar, Proc. FGCS92, pages 347–356, June 1992.
- Hideki Yasukawa, Hiroshi Tsuda, and Kazumasa Yokota. Objects, Properties, and Modules in Quixote, Proc. FGCS92, pages 257–268, 1992.
- Hiroshi Tsuda and Kazumasa Yokota., A DOOD Approach to Constraint-Based Grammar In *SIG-DBS No.94*, pages 21–28, Nagasaki, 1993. Inf. Proc. Soc. Japan. (in Japanese).
- (*) Hiroshi Tsuda. cu-Prolog III User's manual, ICOT Free Software (http://www.icot.or.jp/), 1992.
- Satoshi Tojo, Hiroshi Tsuda, Hideki Yasukawa, Kazumasa Yokota, and Yukihiro Morita. Quixote as a Tool for Natural Language Processing. *TAI93*, pages 266–270, Boston, 1993. IEEE.
- Kazumasa Yokota, Hiroshi Tsuda, and Yukihiro Morita. Specific Features of a Deductive Object-Oriented Database Language Quixote. Proc. ACM SIGMOD Workshop on Combining Declarative and Object-Oriented Databases, Washington DC, USA, May 29 1993.

- Satoshi Tojo, Hiroshi Tsuda, Hideki Yasukawa, Kazumasa Yokota, and Yukihiro Morita. Quixote: A Framework for Linguistic Information Processing. *Journal of Japan Society of Artificial Intelligence*, 9(6):863–874, 1994. (in Japanese).

- (*) Hiroshi Tsuda and Kazumasa Yokota., Knowledge Representation Language Quixote. *Proc. FGCS94*, June 1994.

- (*) Hiroshi Tsuda. cu-Prolog for Constraint-Based Natural Language Processing. *IEICE Transactions on Information and Systems*, E77-D(2):171–180, February 1994.

- Akira Aiba, Kazumasa Yokota, and Hiroshi Tsuda., Heterogeneous Distributed Cooperative Problem Solving System HELIOS, Proc. FGCS94, 1994.

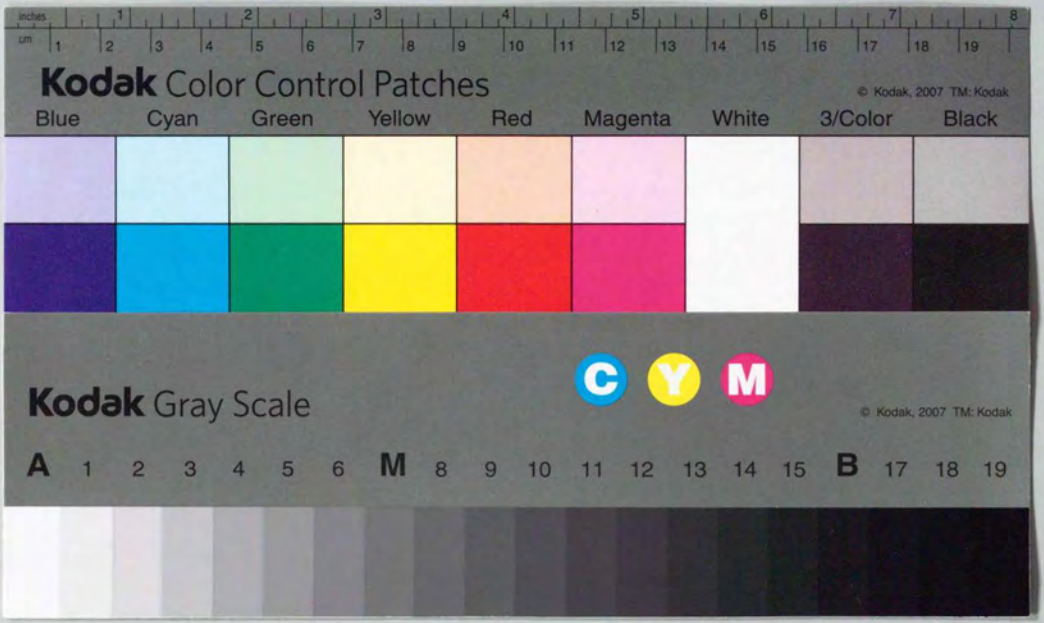
- Hiroshi Tsuda and Akira Aiba., Heterogeneous Natural Language Understanding in Helios. FGCS94 Workshop on Heterogeneous Cooperative Knowledge-Bases, ICOT, Dec. 1994.

- (*) Hiroshi Tsuda. Big-Quixote User's manual, ICOT Free Software (http://www.icot.or.jp/), 1995.

- (*) Hiroshi Tsuda and Yasunari Harada., Semantics and Pragmatics of Adnominal Particle NO in Quixote. In Takao GUNJI, editor, *Studies on the Universality of Constraint-Based Phrase Structure Grammar*, pages 191–201. Osaka University, March 1996. Report of the International Scientific Research Program, Joint Research, Project No. 06044133, Supported by the Ministry of Education, Science, and Culture, Japan.

- Sigeichirou Yamasaki and Hiroshi Tsuda (Eds. Trans.) , "Telescript Gengo Nyuumon" (Introduction to Telescript Language), ASCII, Japan, 1996 (in Japanese).

- Hiroshi Tsuda and Akira Aiba., Heterogeneous Natural Language Understanding in Helios. Computer Software, 13(6), pp.43–52, 1996. (In Japanese).


inches
cm
Kodak Color Control Patches
© Kodak, 2007 TM: Kodak
Blue
Cyan
Green
Yellow
Red
Magenta
White
3/Color
Black
Kodak Gray Scale
C
Y
M
© Kodak, 2007 TM: Kodak
A 1 2 3 4 5 6 M 8 9 10 11 12 13 14 15 B 17 18 19