

cu-Prolog 第三版ユーザーズマニュアル

津田 宏

(財) 新世代コンピュータ技術開発機構 第二研究部

〒 108 東京都港区三田 1-4-28 三田国際ビル 21 階

Tel : 03 (3456) 3069 Fax : 03 (3456) 1618 E-mail : tsuda@icot.or.jp

1994 年 7 月 7 日

目次

1	概要	3
1.1	cu-Prolog 実行モジュールの作成法	3
1.2	他システムへの移植	5
1.3	起動法・終了法	5
1.4	第二版からの改良点	5
2	cu-Prolog プログラムの記述	6
2.1	用語の説明	6
2.2	制約付ホーン節 (Constrained Horn Clause: CHC)	6
2.3	部分項 (PST) の記述	6
2.4	制約の記述	7
2.5	BNF 記法による cu-Prolog の構文	8
3	システムコマンド一覧	8
3.1	Prolog に関するコマンド	9
3.2	ファイル入出力についてのコマンド	9
3.3	デバッグコマンド	9
3.4	制約変換に関するコマンド	9
3.5	その他のコマンド	9
3.6	Macintosh 版 cu-Prolog メニュー	9
3.6.1	File メニュー	10
3.6.2	Command メニュー	10
3.6.3	Trace メニュー	10
3.6.4	MODE メニュー	10
4	cu-Prolog 組込み述語・ファンクタ	10
4.1	一般的な組込み述語	10
4.1.1	関数的な組み込み述語	10
4.1.2	述語的な組み込み述語	12
4.2	制約変換に関する組込み述語	13
4.3	JPSG パーザに関する組み込み述語・ファンクタ	13

5 ファイルの入出力	14
5.1 プログラムの読み込み	14
5.2 プログラムの保存	14
5.3 ログファイルの設定、解除	14
6 制約変換機構	14
6.1 制約変換のみの実行	14
6.1.1 @ モード	15
6.1.2 unify(C,NC)	15
6.2 制約変換の操作	15
6.3 制約変換のヒューリスティック	16
6.3.1 DEFINITION からの節の選択	16
6.3.2 NON-MODULAR からの節の選択	16
6.3.3 unfold するリテラルの選択	16
6.4 実行例	16
6.4.1 記号・組合せ制約	16
6.4.2 選言的素性構造の単一化	17
7 トレース機能	18
7.1 スパイフラグの設定	18
7.2 トレースモード	18
7.3 制約変換のトレース	18
7.3.1 トレースの表示	18
7.3.2 ステップトレースのユーザ入力	19
8 JPSG パーザ	19
8.1 多義語の辞書記述	19
8.2 JPSG の制約の記述	20
8.3 JPSG の素性	20
8.4 実行例	21
9 おわりに	22

1 概要

制約論理型言語 cu-Prolog は、自然言語処理の単一化文法の特徴である宣言的な文法記述の実装を目的とした、記号的・組合せ的な制約を解くのに適した Prolog 処理系である。ここでいう制約とは、変数共有や変数束縛などによる依存関係をいう。従来の CLP の多くが代数方程式・不等式による制約を扱うのに対して、cu-Prolog は自然言語処理並びに AI 全般への応用に適するプログラム言語であるといえる。

cu-Prolog は ICOT の PSG-WG にて、JPSG (Japanese Phrase Structure Grammar: 日本語句構造文法) のパーザを実現する目的で考案された。JPSG では、情報を素性 (ラベル) とその値の対の不定個の並びによる素性構造により格納し、それらをノードとする句構造によって自然言語の構造を表現する。JPSG のような制約ベースの単一化文法では、自然言語の文法はすべて制約で宣言的に記述される。JPSG における制約とは、二分木の各ノードにおける素性値の記号的・組合せの関係である。

cu-Prolog は Prolog のユーザ定義述語による項を制約として扱うことができるので、JPSG などの自然言語処理における制約を自然に記述し、そのまま制約として実行することが可能である。cu-Prolog の制約解消系は、論理プログラムの unfold/fold 変換を基本操作とし、変数共有や束縛などによるヒューリスティックスを加えたところに特徴がある。

cu-Prolog 第三版では、制約ベースの単一化文法の記述により適したものとするため、データ構造として部分項 (Partially Specified Term) をとり入れた。これにより、素性構造をそのまま扱うことができる。さらに、制約解消系も部分項の導入とともに自然に拡張を行ない、単一化文法で重要となる選言的素性構造に対しても対応できるようになった。選言的素性構造の単一化は本来計算量の大きな問題であり、より実際的なアルゴリズムが研究されている。cu-Prolog 第三版では、それがヒューリスティックスも含めて制約変換という共通した枠組で解決されている。

cu-Prolog は DEC10 ライクな構文の Prolog インタプリタに、制約解消系を加えた処理系であり、アルゴリズムおよびソースは全て公開されている。cu-Prolog 第三版は C 言語により記述され、UNIX (4.2/3 BSD) 上にて開発された。¹本マニュアルは ICOT Free Software として公開されている UNIX 版 cu-Prolog 第三版のユーザーズマニュアルである。

1.1 cu-Prolog 実行モジュールの作成法

cu-Prolog 第三版は、C 言語による以下のモジュールからなる。

- ヘッダファイル

`include.h` 構造体・マクロの定義

`funclist.h` 他モジュールで使われる関数の外部参照

`varset.h` 大域変数の初期化

`globalv.h` 大域変数の外部参照

`defsyp.h` 組み込み述語変数の初期化

`syp.h` 組み込み述語変数の外部参照

- システムの基本モジュール

`main.c` 処理系トップレベル

`mainsub.c` `main.c` のサブルーチン

`new.c` メモリ管理

¹他に中京大学の白井英俊助教授 (sirai@scs.chukyo-u.ac.jp) により開発された MS-DOS 版、Macintosh 版がある。詳細は README.j を参照のこと

read.c プログラム・項の読み込み

print.c 項の表示ルーチン

- Prolog 処理系モジュール

refute.c 処理系本体

unify.c 単一化

- 組み込み述語に関するモジュール

defsysp.c 組み込み述語の初期化

syspred1.c 組み込み述語の定義.1

syspred2.c 組み込み述語の定義.2

jpsgsub.c JPSG パーザ用組み込み述語の定義

- 制約解消系に関するモジュール

modular.c 制約解消系エントリ、共通ツール

trans.c 制約解消系メイン

tr_sub.c 制約解消系サブ

tr_split.c 素式の分割ルーチン

これらのモジュールをコンパイル、リンクして実行ファイルを得る。例えば UNIX の場合は上記のファイルを一つのディレクトリに置き、

```
cc -o cuprolog *.c <cr>
```

とする。また添付の makefile を用いて、

```
make <cr>
```

としてもよい。

コンパイルの注意。使用する機種、コンパイラによっては、コンパイル前に include.h 中の最初の `#define` 文を書き換える必要がある。

- SUN4 の場合、`#define SUN4 1`
- UNIX.4.2BSD のライブラリ `times()` がサポートされている場合、`#define CPUTIME 60`
- それ以外の場合 (CPU タイムは表示されない)、`#define CPUTIME 0`

ver3.9(94/11/4) より、ストリング関数で EUC 漢字コード (2 バイトコード) を認識することが可能となった。include.h にて

```
#define KANJI 1
```

とすると、`strlen`, `divstr`, `substring`, `concat2`, `concat` の組み込み述語にて、全角カナや漢字 (2 バイト) を 1 文字と認識した処理を行なう。KANJI を 1 以外の値に設定すると、従来と同じく 1 バイトを 1 文字とした処理を行なう。

また、cu-Prolog には以下のデータ領域がある。

システムヒープ: プログラム節、新述語定義等。デフォルト値は `SHEAP_SIZE` (デフォルトは 600Kbytes)

ユーザヒープ: Prolog 反駁時の一時的な構造等。大きさは `HEAP_SIZE` (デフォルトは 500Kbytes)

制約ヒープ: 制約・PST の一時的な構造等。大きさは `CHEAP_SIZE` (デフォルトは 500Kbytes)

環境スタック: Prolog 反駁時の環境。大きさは `ESP_SIZE` (デフォルトは 80Kbytes)

ユーザスタック: Prolog 反駁時のポインタのつけかえ等。大きさは `USTACK_SIZE` (デフォルトは 30Kbytes)

文字列ヒープ: 文字列の格納領域。大きさは `NAME_SIZE` (デフォルトは 50Kbytes)

cu-Prolog 実行中に、これらヒープ、スタックのオーバーフローが頻繁に起こるならば、起動時の引数にてサイズを変更することができる。

Macintosh 版では、MacCUP アイコンのインフォメーションで使用するメモリの大きさを設定することで、上記の領域は適当に確保される。

1.2 他システムへの移植

cu-Prolog 第三版は UNIX4.2/3BSD(特に Sun3,Sun4,Symmetry) の上での実行を確認している。ただし、OS に依存するライブラリの使用は CUP 消費時間計測のみ²なので他システムの移植は容易であろう。

1.3 起動法・終了法

以降では、実行ファイルの名称は `cuprolog` とする。以下のような引数を起動時に指定することができる。

`cuprolog [-Hxxx] [-Sxxx] [-Exxx] [-Cxxx] [-Uxxx] [-Nxxx] [ファイル名]`

それぞれの意味は次の通りである。

- `-Hxxx` : `xxx` には数値を指定する。ユーザーヒープの大きさ (Kbyte 単位) を変更することができる。
- `-Sxxx` : `xxx` には数値を指定する。システムヒープの大きさ (Kbyte 単位) を変更することができる。
- `-Cxxx` : `xxx` には数値を指定する。制約ヒープの大きさ (Kbyte 単位) を変更することができる。
- `-Exxx` : `xxx` には数値を指定する。環境スタックの大きさ (Kbyte 単位) を変更することができる。
- `-Uxxx` : `xxx` には数値を指定する。ユーザースタックの大きさ (Kbyte 単位) を変更することができる。
- `-Nxxx` : `xxx` には数値を指定する。文字列ヒープの大きさ (Kbyte 単位) を変更することができる。
- `ファイル名` : 起動と同時に読み込むファイルを指定する。

例えば、システムヒープを 800Kbytes, 制約ヒープを 600Kbytes, で初期ファイル `foo.p` を読み込んで cu-Prolog を立ち上げるには、

```
cuprolog -S800 -C600 foo.p
```

とする。

cu-Prolog を終了するにはトップレベルから、`%Q` または `:-halt.` とする。

1.4 第二版からの改良点

cu-Prolog 第三版は、第二版と比べ主として以下のような点で改良がなされている。

- 制約変換のモジュール化・マニュアル制約変換導入
- PST(部分項) データ構造の導入, それに伴う制約解消系の拡張
- 組み込み述語の拡充 (含オペレータ)

²`mainsub.c` の `settimer()` および `printtime()` で、SUN-4 の `clock()` または BSD の `times()` を使用している。

2 cu-Prolog プログラムの記述

2.1 用語の説明

項: アトム、変数、複合項、部分項

アトム: 定数、ストリング、数値

定数: 英小文字で始まる文字列、' で囲まれた任意の文字列

ストリング: " で囲まれた任意の文字列

数値: 整数、浮動小数点

変数: 英大文字または _ で始まる文字列。_ のみは無名変数と呼ばれ、任意の二つの無名変数は異なった変数として扱う。

複合項: p を文字列、 $t_1, t_2, \dots, t_n$ を項としたとき、 $p(t_1, t_2, \dots, t_n)$ を複合項と言う。他の項の引数に現れる複合項を**関数**と呼び、そのときの p を**ファンクタ**、それ以外の複合項を**述語**と呼び、そのときの p を**述語名**と言う。なお、第三版より単項演算子 (not)、二項演算子 ($=$, $<$, $>$ 等) もサポートされた。

部分項 (PST): 素性名 / 素性値の並びを $\{ \}$ で囲んだもの。素性名は英小文字で始まる文字列、素性値は項を取る。

コメント (行): %% で始まる行、または * と * \ で囲まれた文字列 (この場合コメントの入れ子は許さない)

2.2 制約付ホーン節 (Constrained Horn Clause: CHC)

cu-Prolog のプログラム節は、Constrained Horn Clause (CHC: 制約付ホーン節) と呼ばれ、通常のホーン節に制約を加えたものになっている。CHC は以下の 3 種類の節からなる。

1. $H; C_1, \dots, C_n.$ (事実節)

2. $H : -B_1, \dots, B_m; C_1, \dots, C_n.$ (ルール節)

3. $: -B_1, \dots, B_n; C_1, \dots, C_n.$ (質問節)

H は頭部 (ヘッド)、 $B_1, \dots, B_n$ は本体、 $C_1, \dots, C_n$ は制約部と呼ぶ。制約部が空の制約付きホーン節が通常のホーン節である。

なお、プログラムの本体で素式を表す変数も許している。これにより、

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

と、call/1, not/1 が定義できる。

2.3 部分項 (PST) の記述

cu-Prolog 第三版では部分項 (Partially Specified Term: PST) を項として使用することができる。部分項は次のように、ラベルと値を / をデリミタとした対の不定個の並びを中括弧で囲んだものである。ラベルはアトム、値は項をとる。

$\{11/a, 12/X, 13/\{f/b, g/c\}\}$

部分項の導入により単一化が拡張される。部分項 X, Y の単一化結果が部分項 Z の場合、以下が成り立つ。

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

例えば、 $\{l/a, m/X\}$ と $\{m/b, n/c\}$ の単一化結果は $\{l/a, m/b, n/c\}$ となる。

同一の部分項がプログラム中に複数出現する場合には、媒介変数を用いて制約により表示される。例えば

$f(X) :- g1_p1(X), g2_p2(X); _p1=\{f/a, g/c\}.$

ようになる。`%w` コマンドで書き出されたファイル中では、部分項を表す変数は上のように `_p1, _p2, ...` と表される。このような媒介変数は `%R` コマンド (制約のプリプロセス) によって、部分項へのポインタへと書き換えられる。

2.4 制約の記述

制約は以下に定義される「モジュラー」という標準形 (に書き換えられるもの) でなければならない。プログラムの制約部分を制約を標準形に書き換えるには、プリコンパイル機構 (システムの `%P` コマンド) が用意されている。

[定義] 1 (モジュラー) 素式列 $C_1, C_2, \dots, C_m$ は以下の全ての条件を満たす時、モジュラーであるという。

1. C_i の全ての引数は変数 ($1 \leq i \leq m$) である。
2. どの変数も二個以上の空虚でない引数位置には現れない。
3. C_i はモジュラー定義述語から成る ($1 \leq i \leq m$)。 □

ただし、空虚な引数、およびモジュラー述語は以下に定義される。

[定義] 2 (空虚な引数位置) 述語 p の全ての定義節において p の第 i 引数が変数である時、 p の第 i 引数は空虚であるという。述語 p による *unfolding* によって、第 i 引数はそれ自体で何かに束縛しないことを表している。 □

[定義] 3 (モジュラー述語) 述語 p の全ての定義節の本体が、モジュラーまたは *nil* の場合、 p はモジュラー述語であるという。 □

PST の導入にともない、制約の標準形を一部拡張した。

[定義] 4 (引数の成分) 述語の各引数位置の成分 (*component*) とは、その引数が具体化し得る部分項のパスの集合である。通常項 (アトム, リスト, 複合項) はパス `[]` と見なす。

述語 p の第 n 引数の成分を $Cmp(p, n)$ と書き、 $Cmp(T)$ により項 T のパスの集合を表す。また、特殊な述語として $X=t$ なる形がある。変数 X は成分が $Cmp(t)$ の引数位置にあるとみなす。 □

引数の成分はプログラムを静的に解析することにより得られる。述語の引数成分は、`%d` コマンド (述語定義を表示する) により見ることができる。以下の例では述語 `p3/2` の第 1 引数の成分は $\{f, h\}$, 第 2 引数の成分は $\{g, h\}$ である。

```
_%d p3
%d +----- ( p3/2 )----- [f.h|g.h] --2/2--+
p3({f/a},{g/b}).
p3({h/c},{h/d}).
```

[定義] 5 (依存関係) 制約 (素式列) において

1. 同一変数が共通の成分を持つ複数の引数位置に出現している、または
2. 同一変数が成分 $\{[]\}$ の引数と、成分が部分項のパスのみの引数位置に現れている、または
3. 成分が ϕ でない引数位置が具体化されている場合

は**依存関係**があるという。 □

例えば、 $\text{Cmp}(p, 1) = \{f, g\}$, $\text{Cmp}(q, 1) = \{h\}$ の場合、制約 $p(\{f/b\})$ は依存関係があり、 $p(X), q(X)$ や $p(\{1/a, m/b\})$ は依存関係を持たない。

[定義] 6 (モジュラー (制約の標準形)) 依存関係の存在しない制約をモジュラーという。 □

2.5 BNF 記法による cu-Prolog の構文

以下は cu-Prolog の構文の BNF による記述である。漢字コードは EUC に対応している。

```

< char > ::= < capital > | < small > | < digit > | < 漢字 >
< charwhite > ::= < char > | < space >
< capital > ::= A|B|C|...|X|Y|Z
< small > ::= a|b|c|...|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 > ::= < charwhite > | < charwhite > < 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_list > ::= < 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 > ::= < => | = | = .. | > | > = | < | < = | ==

```

3 システムコマンド一覧

cu-Prolog のトップレベルのコマンドには以下のものが用意されている。ここで、*predicate* は、*predicate_name* または *predicate_name/arity* を表すとする。

3.1 Prolog に関するコマンド

%h	ヘルプ
# <i>OS_command</i>	OS のコマンドを実行する
%d <i>predicate</i>	述語の定義を表示する
%d.	全述語の定義を表示する
%d/	述語名を表示する (簡約された述語も含む)
%d?	述語名を表示する (簡約された述語は含まない) システム組み込み述語には +:recursive, ^:functor ユーザ述語には *:spied, -:reduced, +:recursive, #: 制約変換による新述語
%f	システムヒープの使用状況を表示
%G	(静的) ガーベジコレクション
%Q	cu-Prolog を終了する
%R	システムリセット
%S	直前の質問の処理に対する統計情報の表示
%c 数	推論の深さの上限を設定する。これより深いゴールは失敗とする
%u	未定義述語をエラー (TURE) にするか、失敗 (FALSE) にするかのスイッチ

3.2 ファイル入出力についてのコマンド

"filename"	プログラムファイルをエコーバックなしで読み込む
"filename?"	プログラムファイルをエコーバックつきで読み込む
%l filename	ログファイルを設定する
%l no	ログファイルへの記録を中止する
%w filename	プログラムをファイルに書き出す

3.3 デバッグコマンド

%p <i>predicate</i>	述語のスパイフラグを設定 / 解除するスイッチ
%p *	全述語のスパイフラグを設定する
%p .	全述語のスパイフラグを解消する
%p >	制約変換にスパイフラグを設定 / 解除するスイッチ
%p ?	スパイフラグが設定されている述語名を表示する
%t	ノーマルトレースモード on/off のスイッチ
%s	ステップトレースモード on/off のスイッチ

3.4 制約変換に関するコマンド

%L	制約変換で作られた新述語の導入定義節を表示する
%n <i>predicate</i>	制約変換で作られる新述語名を設定する (デフォルトは c)
%a	制約変換を全モジュラーモードにする
%o	制約変換を M-solvable モード ³ にする
%P <i>predicate</i>	述語の定義節の制約部を前処理する (標準形に書き換える)。
%P *	全ての述語の定義節の制約部を前処理する
%P ?	標準形でない制約を持つ定義節を表示する

3.5 その他のコマンド

%C JPSG パーザ用組込みファンクタ cat() を再定義する。

3.6 Macintosh 版 cu-Prolog メニュー

Macintosh 版では、システムコマンドの多くがメニューから実行できる。

3.6.1 File メニュー

Open	cu-Prolog のプログラムファイルを読み込む
Save Predicates	プログラムをファイルに書き出す (%w)
Quit	cu-Prolog を終了する (%Q)

3.6.2 Command メニュー

Help	ヘルプを表示する (%h)
Show Predicate	述語定義を表示する (%d)
Set Log File	ログファイルを設定・解除する (%l)
Preprocess Constraint	制約を前処理する (%P)
Max # of Refutation	Prolog 推論の段数の最大を設定する (%c)
List New Predicates	制約変換で作られた新述語の導入節を表示する (%L)
Reset	cu-Prolog のリセット (%R)
Handling Undef	未定義述語をエラー (true), 失敗 (false) にするかを切り替える (%u)
Print Level	表示のレベルを設定する
Garbage Collection	静的ガーベジコレクションを行う (未サポート %G)
New Predicate Name	制約変換でできる新述語の名前を再定義する (%n)
Max # of Vars in Trans	制約変換で一制約中の変数の最大個数を設定する。これを越える制約は処理しない。

3.6.3 Trace メニュー

Step Trace	ステップトレースモードにする (%s)
Normal Trace	ノーマルトレースモードにする (%t)
Trace Off	ノートレースにする
Marking Spy	述語のスパイフラグを設定する (%p)

3.6.4 MODE メニュー

All Modular mode	制約を全部モジュラーに変換する (%a)
M-solvable mode	制約を M-solvable モードで変換する (定義節の少なくとも一つがモジュラーになる)

4 cu-Prolog 組込み述語・ファンクタ

4.1 一般的な組込み述語

cu-Prolog 第三版では、以下の組込み述語が用意されている。引数のモードは、+ は具体化された項、- は自由変数を表す。なお、strlen, divstr, substring, concat2, concat のストリング関係の組み込み述語では EUC 漢字コードを認識した処理もできる。詳しくは本稿最初のコンパイルの注意を参照のこと。

4.1.1 関数的な組み込み述語

解が一つ (バックトラックしない) の述語である。

述語	動作
!/0	カット
abolish/2	abolish(P+,A+) 述語 P/A の定義を消去する
arg/3	arg(Pos+,T+,Arg-) T の Pos 番目の引数を Arg と単一化する 例:arg(2,test(a,b,c),X) では X=b
assert/1,2,3	assert(Head+),assert(Head+,Body+),assert(Head+,Body+,Cstr+) 節 Head:-Body;Cstr をプログラムの最後に付け加える
asserta/1,2,3	節をプログラムの最初に付け加える
assertz/1,2,3	節をプログラムの最後に付け加える
attach_constraint/1	attach_constraint(Cstr+) Cstr を新たな制約として付け加える Cstr は素式または素式のリスト
atom_to_str/2	atom_to_str(ATOM+,STR-) アトムを文字列に変換する。

close/1	close(FP+) open/3 で作られたファイルポインタ FP を閉じる
compare/3	compare(X+,Y+,C-) X と Y が共に数値または文字列の時にそれらの大小関係を返す。 C の値は>,,=<,のいずれかである。 例: compare("abc","xyz",'<') compare(123,456,'<')
concat2/2	concat2(Str+,List-) 文字列 Str を一文字毎に分解したリストを List と単一化する 例: concat2("ab",["a","b"])
default	default(X?,Y+,Z+) PST の X が Y の属性値を一つでも含まなければ fail, そうでなければ X に Z が追加される。 例: :- X={pos/p,sem/Y},default(X,{pos/p},{aj/n,[],refl/[]}). なら X={pos/p,aj/n,[],refl/[],sem/Y} :- X={pos/n,sem/Y},default(X,{pos/p},{aj/n,[],refl/[]}). は fail
divstr/4	divstr(X+,N+,Y-,Z-) 文字列 X の N 番目から前を Y に後を Z にバインドする (N が負の場合は文字列の末尾から数える) 例: divstr("abcdefg",3,"abc","defg"). divstr("abcdefg",-2,"abcde","fg").
equal/2	equal(X,Y) X と Y を単一化する
neq/2	neq(X,Y) X と Y が単一化可能な場合成功する
eq/2	eq(X,Y) X が Y と等しいかチェックする
fail/0	常に失敗する
functor/3	functor(T+,F,A) T の述語名が F/A 例: functor(p(a,b),p,2)
geq/2	geq(X+,Y+) 数値比較 (X>=Y)
greater/2	greater(X+,Y+) 数値比較 (X > Y)
halt/0	cu-Prolog を終了する
leq/2	leq(X+,Y+) 数値比較 (X<=Y)
less/2	less(X+,Y+) 数値比較 (X < Y)
ml/2	univ. 述語 ml(T+,L-) は T=..L と等価 L は項 T の述語名と引数からなるリストである 例: ml(f(a,b,c),X) --> X=[f,a,b,c]
multiply/3	multiply(X+,Y+,Z-) 数値乗算 X * Y = Z
name/2	name(A+,L-) 文字列 A を文字のリスト L に変換 例: name(abc,[97,98,99]).
nl/0	'\n' を表示する
op/3	op(P+,T+,Op+) 演算子またはそのリスト Op を, 優先度を P、タイプを T として定義する 優先度は 0 から 1000 まで。タイプは xf,yf,fx,fy,xfx,xfy,yfx の何れかである。 例: :-op(700,xfx,'=').
open/3	open(FileName+,Type+,FP-) ストリームを開く FileName はファイル名, Type は r または w. FP は新しく作られるファイルポインタである.
pnames/2	pnames(PST+,FL) PST の素性のリストを FL と単一化する 例: pnames({1/a,m/b},{1,m}).
pvalue/3	pvalue(PST+,F+,V) PST の素性 F の値を V と単一化する PST に相当する素性がない場合には fail する
read/1	read(X) 現ストリームから項を読み込み引数と単一化する read(X) に対してプロンプト (?) が出力待ちとなる。 キーボードから入力された文字列と X とが単一化する。
read/2	read(X-,FP+) はファイルポインタ FP から読み込む。 FP がファイル末尾ならば、X は end_of_file と単一化する
reset_timer/0	CPU タイマをリセットする (timer/2 参照)
retract/1,2,3	retract(Head+), retract(Head+,[Body+]),retract(Head+,[Body+],[Cstr+]) Head:-Body;Cstr. と単一化する節を一つ消去する 例: retract(p1(a)) : 節 p1(a). が消去される

	<code>retract(f(X),[m(X,Y)])</code> : 節 <code>f(X):-m(X,Y)</code> . が消去される
	<code>retract(f(X),[m(X,Y)],[c(Y)])</code> : 節 <code>f(X):-m(X,Y);c(Y)</code> . が消去される
<code>see/1</code>	<code>see(F-)</code> 名前 <code>F</code> のファイルが現入力ストリームになる
<code>seen/0</code>	<code>see/1</code> で作られた現入力ストリームを閉じる
<code>strcmp/3</code>	<code>strcmp(X+,Y+,C-)</code> 文字列 <code>X,Y</code> の大小関係を <code>C</code> にバインドする <code>C</code> の値は <code>>`,``,`<</code> のいずれか 例: <code>strcmp("abc","xyz",'<')</code> .
<code>strlen/2</code>	<code>strlen(S+,L)</code> <code>L</code> は文字列 <code>S</code> の長さ 例: <code>strlen("abcdef",6)</code> .
<code>substring/3</code>	<code>substring(X+,N+,Y-)</code> 文字列 <code>X</code> の <code>N</code> 番目以降を <code>Y</code> にバインドする (<code>N</code> が負の場合は文字列の末尾から数える) 例: <code>substring("abcdefg",3,"defg")</code> <code>substring("abcdefg",-3,"efg")</code> .
<code>substring/4</code>	<code>substring(X+,N+,L+,Y-)</code> 文字列 <code>X</code> の <code>N</code> 番目以降 <code>L</code> 個の文字列を <code>Y</code> にバインドする (<code>N</code> が負の場合は文字列の末尾から数える) 例: <code>substring("abcdefg",3,2,"de")</code> <code>substring("abcdefg",-3,2,"ef")</code> .
<code>sum/3</code>	<code>sum(X,Y,Z)</code> 数値加算 <code>X + Y = Z</code> (注: 少なくとも 2 つの引数は数値にバインドしていればよい)
<code>tab/0</code>	タブを表示する
<code>tab/1</code>	<code>tab(FP+)</code> ファイルポインタ <code>FP</code> にタブを出力する
<code>tell/1</code>	<code>tell(T+)</code> 名前 <code>T</code> のファイルを現出力ストリームにする
<code>timer/2</code>	<code>timer(X-,Y-)</code> <code>X</code> は <code>reset_timer</code> 以後の CPU 時間、 <code>Y</code> にはその内制約変換で使われた CPU 時間をバインドする
<code>told/0</code>	<code>tell/1</code> で作られた現出力ストリームを閉じる
<code>true/0</code>	常に成功する
<code>unbreak/0</code>	ステップトレースで <code>break</code> した後、 <code>:-unbreak.</code> でその時点に戻る
<code>var/1</code>	<code>var(T)</code> <code>T</code> が自由変数であるとき成功
<code>write/1</code>	<code>write(T)</code> 項 <code>T</code> を表示する <code>T</code> がストリングの場合、引用記号は表示されない

4.1.2 述語的な組み込み述語

解が複数ありうる (バックトラックする) 組み込み述語である。

述語	動作
<code>apnd/3</code>	<code>apnd(List,List,List+)</code> または <code>apnd(List+,List,List)</code> (第 1 もしくは第 3 引数がバインドされていなければ fail) <code>append</code> 述語の組み込み版
<code>clause/3</code>	<code>clause(T+,Body-,Cstr-)</code> <code>T</code> と単一化するヘッドを持つ節の 本体を <code>Body</code> , 制約を <code>Cstr</code> にバインドする
<code>concat/3</code>	<code>concat(S1,S2,S)</code> 文字列 <code>S1</code> と <code>S2</code> を連結したものを <code>S</code> とバインドする <code>concat(S1+,S2+,S-)</code> または <code>concat(S1-,S2-,S+)</code> 例: <code>concat("ab","cd",S) --> S="abcd"</code> <code>concat(X,Y,"abc") --></code> <code>X="",Y="abc" or X="a",Y="bc" or X="ab",Y="c" or X="abc",Y=""</code>
<code>count/1</code>	<code>count(X-)</code> 呼び出し毎に異なる数 (<code>X=0,1,2,...</code>) を返す
<code>execute/1</code>	<code>execute(L+)</code> リスト <code>L</code> で与えられた素式列を順次実行する <code>execute([memb(X,[a,b]),memb(X,[b,c])])</code>
<code>gensym/1</code>	<code>gensym(X-)</code> 呼び出し毎に異なる文字列を返す。 初期値はコマンド <code>%n</code> で変えられる (デフォルトは <code>c</code>)
<code>isop/3</code>	<code>isop(Prec,Type,OP)</code> <code>Prec</code> と <code>Type</code> は演算子 <code>OP</code> の優先度とタイプ 例: <code>:-isop(X,Y,Z)</code> . に対し <code>X=900</code> , <code>Y=xfy</code> , <code>Z='/'</code> 等
<code>memb/2</code>	<code>member(Atom,List+)</code> <code>member</code> 述語の組み込み版
<code>or/2,3,4,5</code>	複数のゴールを実行

```
例: :-or(memb(X,[a,b]),memb(X,[j,k])).
--> X=a,b,j,k
```

4.2 制約変換に関する組み込み述語

述語	動作
condname/2	condname(Cstr+,PL-) 制約 Cstr の述語名のリストを PL にバインドする 例: condname([f(a,b),g(c,d)], [g,f])
pcon/0	述語実行時点での制約を表示する
project_cstr/1	project_cstr(Term+) 述語実行時点での制約のうち、 項 Term の中に含まれる変数に関連するものを表示する
project_cstr/2	project_cstr(Term+,Var-) 述語実行時点での制約のうち、 項 Term の中に含まれる変数に関連するものを変数 Var にバインドする
unify/2	unify(C+,NC-) C を制約変換したものを NC にバインドする C,NC はのような素式のリスト 例: unify([member(X,[a,b,c]),f(X,Y)], [c0(X,Y)])

4.3 JPSG パーザに関する組み込み述語・ファンクタ

cu-Prolog によりユニフィケーション文法の属性構造を記述する場合、PST を用いるのが適当であるが、次に示す cat という特別なファンクタも用意されている。

述語	動作
cat/6	JPSG の文法範疇を表すファンクタ
t(M,L,R)	履歴を表すファンクタ
tree(H)	履歴 H を木構造で表示する

JPSG 等の文法範疇の実装に便利な cat() という特別なファンクタが用意されている。tree() により適当にプリティプリントされる。cat のアリティは次の %C コマンドでトップレベルから変更可能である。ただし %C コマンドは一度しか実行できない。

```
%C [素性名 1, 素性型 1, 素性名 2, 素性型 2,... ]
```

と素性名と素性型をリストで囲んで指定する。

ただし、素性名は大文字から始まり 5 文字以内でなければならない。また、素性型は、1,2,3 の数値で指定する。ここで、

素性型	素性
1	2,3 以外
2	値として文法範疇の一つ取る素性 (Adjacent 等)
3	値として文法範疇の集合を取る素性 (Subcat 等)

デフォルトでは、[POS,1,FORM,1,AJA,2,AJN,2,SC,3,SEM,1] つまり

素性名	対応する JPSG の素性
POS	pos
FORM	gr,vform,pform, and so on
AJA	adjacent
AJN	adjunct
SC	subcat
SEM	sem

となっている。

また tree コマンドにより cat ファンクタは

```
第 1 素性値 [第 2 素性値, 第 3 素性名: 第 3 素性値, ... ]: 最終素性値
```

の順に、さらに素性値が [] の素性は省いてプリントされる。

[定義] 7 (履歴) 履歴は次のように定義される

- 文法範疇は履歴
- C と W が文法範疇の時 $t(C, W, [\])$ は履歴
- L と R が履歴、 M が文法範疇の時、 $t(M, L, R)$ は履歴
- L と R が履歴、 C と W が文法範疇の時、 $t(t(C, W, [\]), L, R)$ は履歴

□

$tree(t(C, W, [\]))$ は

$C \dashrightarrow W$

を出力し、 $tree(t(M, L, cat(n, ga, [\], [\], [\], ken)))$ は

```
---M
|
|-L
|
|-n[ga]:ken
```

を出力する。

5 ファイルの入出力

5.1 プログラムの読み込み

- システム起動と同時に読み込むには、OS のプロンプトから

```
cuprolog filename
```

とする。エコーバックはない。

- トップレベルからエコーバックなしで読み込む

```
"filename"
```

- トップレベルからエコーバックありで読み込む (デバッグ時など)

```
"filename?"
```

5.2 プログラムの保存

プログラムをファイルに保存するには、トップレベルから

```
%w filename
```

とする。

5.3 ログファイルの設定、解除

- ログファイル名を設定する。これ以後の画面はすべて設定されたファイルに格納される。

```
%l filename
```

- ログの中止

```
%l no
```

6 制約変換機構

6.1 制約変換のみの実行

次のように、制約変換機構のみを単独で実行することができる。

6.1.1 @ モード

トップレベルから制約変換機構を単独で使うことができる (主にデバッグ用)。

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

とすると、cu-Prolog はこれと等価でモジュラーな制約 (例えば $c0(X)$) と、制約変換の際に作られた新述語の定義を返す。

6.1.2 unify(C, NC)

組み込み述語 unify により Prolog 上で変換ルーチンを陽に使うこともできる。この場合、制約は次のように素式を要素とするリストで記述する。

[$c0(X, [a, b, c])$, $c1(P, Q, R)$, $c2(Q, S)$]

unify(OldCond, NewCond) は、OldCond がリスト形式の制約に具体化されていて、NewCond が自由変数の場合にのみ制約変換を実行し、NewCond には、OldCond と等価でモジュラーな制約が格納される。

6.2 制約変換の操作

制約変換でできる新しい節の集合 (以下、「節集合」と呼ぶ) として、以下の 3 つを用意する。

DEFINITION 新述語の導入節

NON-MODULAR 新定義節のうちモジュラーでない節

MODULAR 新定義節のうちモジュラーな節

モジュラーでない制約を C 、 C に含まれる変数を $X_1, \dots, X_n$ 、 p を n 引数の新述語名とする。制約変換ルーチンは

$$p(X_1, \dots, X_n) == C.$$

を DEFINITION に付加し、以下の三つの基本操作を繰り返すことで、DEFINITION および NON-MODULAR を空にすることができた場合に成功する。このとき、MODULAR の中の節が新定義節で、 C は $p(X_1, \dots, X_n)$ に書き換えられたことになる。

なお現実装では、制約変換の前後で reduction 操作 (定義節が一つの述語は強制的に展開する) を行っている。制約変換は次の三つの基本操作からなる。

1. unfolding

NON-MODULAR または DEFINITION から一節 $H : -L, R$. を取り除く。 L はリテラル、 R は残りの本体である。 L と単一化する頭部を持つプログラムを $P_i : -B_i$ ($P_i \theta_i = L \theta_i$, $i = 1, \dots, n$) とすると、 $H \theta : -B_i \theta, R \theta$ ($i = 1, \dots, n$) を、本体がモジュラーの場合は MODULAR に、そうでない場合は NON-MODULAR に付け加える。

2. folding

NON-MODULAR より、次の条件を満たす一節 $H : -B, D$. を取り除く。

- B と D は変数の依存関係を持たない
- DEFINITION に節 $P == Q$ があって、 $Q \theta = B$ を満たす

NON-MODULAR に $H : -P \theta, D$. を加える。

3. modularize

NON-MODULAR から一つの節 $H : -B_1, \dots, B_n, R$ を取り除く。ただし、

- B_i ($i = 1, \dots, n$), R は素式列
- B_i と B_j ($i \neq j$) は変数の依存関係を持たない
- R はモジュラー

とする。各 B_i は変数 $X_{i,1}, \dots, X_{i,m_i}$ を含み、 p_i を m_i 引数の新述語として、 $P_i = p_i(X_{i,1}, \dots, X_{i,m_i})$ とする ($i = 1, \dots, n$)。このとき、 $P_i == B_i$. を DEFINITION に追加し ($i = 1, \dots, n$)、MODULAR に $H : -P_1, \dots, P_n, R$. を追加する。

6.3 制約変換のヒューリスティック

現行の cu-Prolog では制約変換を次の手順で行っている。

1. DEFINITION に節がある場合には、DEFINITION より一つの節 C を選び unfolding する。節 C は DEFINITION より取り除く。
2. DEFINITION が空の場合には、NON-MODULAR より節 N を選び、N の頭部において全ての引数が相異なる変数の場合には、unfolding を行う。
3. そうでない場合は NON-MODULAR の任意の節 N の本体を folding または modularize する。
4. 以上の操作を、DEFINITION および NON-MODULAR が空になるまで繰り返す。

したがって、ヒューリスティックとしては

- DEFINITION からの節の選び方
- NON-MODULAR からの節の選び方
- unfolding において展開するリテラルの選び方

が考えられる。

6.3.1 DEFINITION からの節の選択

DEFINITION はスタックで実装しており、もっとも最新のものを選択している。

6.3.2 NON-MODULAR からの節の選択

現在 DEFINITION および NON-MODULAR はスタックで実装しており、もっとも最新のものを選択している。

6.3.3 unfold するリテラルの選択

unfolding におけるリテラルの選択に関するヒューリスティックは、現行では以下の要素により各リテラルの活性係数を計算し、活性係数の最も高いものから展開している。

<i>Arity</i>	=	述語の引数の個数
<i>Const</i>	=	引数のうち定数に具体化しているものの個数
<i>Vnum</i>	=	引数に現れる自由変数が制約全体の中で何回出現しているかの個数
<i>Funct</i>	=	引数のうち (変数を含む) 複合項に具体化しているものの個数
<i>Rec</i>	=	述語が再帰的なら 1、そうでなければ 0
<i>Defs</i>	=	述語の定義節の個数
<i>Units</i>	=	述語の定義節のうち単位節 (本体が空) の個数
<i>Facts</i>	=	述語定義がすべて単位節からなる場合に 1、そうでなければ 0
活性係数	=	$3 * Const + 2 * Funct + Vnum - Defs + Units - 2 * Rec + 3 * Facts$

活性係数は、cu-Prolog の制約変換に関して今まで経験的に得られていたヒューリスティックを含むように決定した。さらに要素を増やして実験を行うことでより有効なヒューリスティックも得られると思われる。

6.4 実行例

ここでは、cu-Prolog の制約変換系の実行例をいくつか示す。いずれも、前述の @ コマンドによる実行である。

6.4.1 記号・組合せ制約

述語 member/2 および append/3 は記号的・組合せ的な制約において良く使われる。これらによる制約で非モジュラーなものをモジュラーに変換する例である。

```
tsuda@icot21[5] cup3 % cu-Prolog の起動
***** 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]).                % member/2 の定義
_member(X,[Y|Z]):-member(X,Z).
_append([],X,X).                  % append/3 の定義
_append([A|X],Y,[A|Z]):-append(X,Y,Z).

_@ member(X,[a,b,c]),member(X,[b,c,d]).    % ユーザ制約入力 1

solution = c0(X_0)                  % 等価でモジュラーな制約
c0(b).                              % 制約変換中に作られた新述語の定義
c0(c).
CPU time = 0.017 sec (Constraints Handling = 0.000 sec)

_@ member(X,[a,b,c]),member(X,[j,k,l]).    % ユーザ制約入力 2

solution = fail.                    % 制約変換に失敗
CPU time = 0.017 sec (Constraints Handling = 0.000 sec)

_@ member(A,X),append(X,Y,Z).        % ユーザ制約入力 3

solution = c2(X_1, Y_2, Z_3, A_0)      % 等価でモジュラーな制約とその定義
c4(V0_0, V1_1, V2_2, [V0_0 | V3_3]) :- append(V1_1, V2_2, V3_3).
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)

```

6.4.2 選言的素性構造の単一化

cu-PrologIII は PST を導入することで、言語学の単一化文法で重要なデータ構造である、選言的素性構造に対応することができた。例えば次のような選言的素性構造 [1]:

$$\left[a = \left\{ \begin{bmatrix} b = + \\ c = - \\ b = - \\ c = + \end{bmatrix} \right\} \right] \quad \text{および} \quad \left[\begin{array}{l} a = \left[\begin{array}{l} b = < d > \end{array} \right] \\ d = \left[\begin{array}{l} \end{array} \right] \end{array} \right]$$

は、PST と制約を用いて、 $X=\{a/U\}, s(U)$ および $Y=\{a/\{b/V\}, d/V\}$ と表すことができる。ただし $s/1$ は次のように定義されている。

```

s({b/+,c/-}).    % s/1 の定義
s({b/-,c/+}).

```

X と Y の単一化は制約 $X=Y=\{a/U, d/V\}, U=\{b/V\}, s(U)$ をモジュラーに変換することで行なわれる。以下は cu-PrologIII による実行例である。

```

_s({b/'+' ,c/'-' }).    % 選言の定義
_s({b/'-' ,c/'+' }).

_@ X={a/U},s(U),X={a/{b/V},d/V}.    % 選言的素性構造単一化

solution = c0(X_0, V_2, {a/{b/V_2}, d/V_2}, U_1, {a/U_1})    % 変換後
c0(_p1, '+' , _p1, {b/'+' , c/'-' }, _p1);_p1={a/{b/'+' , c/'-' }, d/'+' }. % 新述語
c0(_p1, '-' , _p1, {b/'-' , c/'+' }, _p1);_p1={a/{b/'-' , c/'+' }, d/'-' }.

```

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

```
_: -c0(X,_,_,_,_). % 新制約を解く
X = {a/{b/'+'}, c/'-'}, d/'+'}; % 解1
X = {a/{b/'-'}, c/'+'}, d/'-'}; % 解2
no.
CPU time = 0.000 sec (Constraints Handling = 0.000 sec)
```

選言的素性構造単一化の結果、X には新たに $c0(X, _, _, _, _)$ なる制約がかかっている。実際に X がどのような値を取り得るかを調べるには、上のように新制約を解いてやれば良い。この例では、単一化後の X は 2 つの PST $\{a/\{b/+, c/-\}, d/+\}$ および $\{a/\{b/-, c/+\}, d/-\}$ の選言にて表される。

7 トレース機能

cu-Prolog には、デバッグ用にトレース機能がある。最初に述語にスパイフラグを設定してから、次のいずれかのトレースモードを選ぶ。

ノーマルトレース: (プロンプトは \$ になる) 途中でストップしない

ステップトレース: (プロンプトは > になる) 実行を一時停止し、入力待ちになる。

7.1 スパイフラグの設定

%p *	全述語にスパイフラグを設定する
%p .	全述語のスパイフラグを解除する
%p predicate	述語のスパイフラグをスイッチする
%p >	制約変換のスパイフラグをスイッチする
%p ?	スパイされている述語名を表示する

7.2 トレースモード

トレースモードに入るスイッチは以下が用意されている。トレースモードから以下のコマンドを再び行くとノーマルトレースモードに戻ることができる。

%s ステップトレースのスイッチ。プロンプトは '>' になる

%t ノーマルトレースのスイッチ。プロンプトは '\$' になる

ステップトレースモードでは、ゴールを表示してからユーザの入力待ちになる。以下によりコマンドを指定する。

入力	動作
リターン	実行を継続する
s	最左ゴールが終了するまでトレースを省略する
a	親ゴールを表示する
b	ブレーク。一時的にトップレベルに移る。:-unbreak によりこの時点に戻ることができる。
f	現在のゴールを fail させる。
l	リープ。このゴールの終了まで飛ぶ。
z	実行を中止する

7.3 制約変換のトレース

制約変換のトレースは、まず %p > によって制約変換にスパイをかけた後に、%t または %s により、ノーマルトレースかステップトレースかを指定する。

7.3.1 トレースの表示

DEFINITION に含まれる節 (新述語の導入節) は、以下の書式で表示される。

[節番号 (節状態, 述語定義数)] 導入節

[1(d,0)] c0(X) <=> member(X,[a,b,c]). (例)

ここで、節状態は以下の記号で表す。

r removed: unfolding により消去され (以下の f, g でない) 節
d derivation: 導入節で unfolding されていないもの
g registered: 新述語の定義に一つ以上の単位節が得られたもの
f false_registered: 新述語の定義が空となったもの

NON-MODULAR および MODULAR に含まれる節 (新しく定義された節) は、以下の書式で表示される。

<節番号 (節状態, 述語定義数)> 新定義節
<2(u)> c0(X):-member(X,[b,c]). (例)

ここで、節状態は以下の記号で表す。

r removed: 簡約化操作、unfolding により消去された節
u untouched: NON-MODULAR の節
m modular: 本体に変数の共有関係がない節
i unit: 単位節

7.3.2 ステップトレースのユーザ入力

ステップトレースでは、トレース表示に次いでユーザの入力待ちになる。以下によりコントロールを指定する。

入力	動作
リターン	継続: デフォルトのヒューリスティックスで節とリテラルを選択し継続する
u CN LN	マニュアル unfolding: CN で節番号、LN で本体の何番目を展開するかを指定する。
s	トレースの中止: 以後の変換過程は表示されない。
n	ステップトレースの中止: 以後ノーマルトレースとなる。
q	制約変換の中断: その時点で新定義節のうち消去されていないものをアサートして制約変換を終了する。
z	制約変換の中止: 新定義節を消去後、制約変換を終了する。

8 JPSG パーザ

JPSG(Japanese Phrase Structure Grammar) 等の単一化文法のパーザを、cu-Prolog の CHC で記述することを考えよう。JPSG における制約は、局所的な枝分かれにおける文法範疇 (親、左右娘) の各素性の関係として宣言的に記述されている。cu-Prolog では辞書や句構造規則を表すプログラム節に、ユーザ定義述語による制約を直接付加できるので、JPSG の制約を自然に記述することが可能である。パーザの処理効率を考えると、効率の良いアルゴリズムの分かっている構文木作成部分はプログラムの本体で記述し、曖昧性に関わる部分を制約で記述するのが望ましい。また、PST により選言を含まない素性構造を表し、選言的な部分は制約で付加することで、選言的素性構造も cu-Prolog で容易に記述できる。

このように文法記述, パーザ記述, 選言的素性構造の扱いがすべて統一的に制約変換という枠組で処理できることが cu-Prolog の大きな特徴である。

ここでは JPSG に基づく簡単な日本語パーザにおいて、CHC の二種類の効果的な使い方を紹介する。

8.1 多義語の辞書記述

同音異義語を別々の辞書エントリに実装しては、バックトラックの度に辞書引きを行い、非常に効率が悪くなる。制約により辞書エントリをまとめるのが制約ベースの自然言語処理の常套手段である。次に挙げるのは、助動詞「れる」の辞書の一部である。「れる」は五段、サ変動詞の未然形に接続し、動詞の subcat の値が二つ (主格、目的格) ならば通常の受身になり、一つ (主格) ならば被害の受身になる。

```
% 「れる」の辞書 (本体が空の CHC)
lex(reru,{sc/SC, sem/Sem, adjacent/{pos/v,infl/I,sc/VSC,sem/Sem}});
    reru_form(I),          % 活用語 (制約)
    reru_sem(VSC,Vsem,SC,Sem). % subcat と sem の組合せ (制約)
```

```
%%%%%% 制約を構成する述語の定義 %%%%%%
```

```

reru_form(vs). % 直前の動詞の活用型 (5 段)
refu_form(vs1). % (サ変)

reru_sem([form/ga,sem/Sbj],Sem, % 被害の受身に対応
        [{form/ga,sem/A},{form/ni,sem/Sbj}],
        affected(A,Sem)).
reru_sem([form/ga,sem/Sbj],[form/wo,sem/Obj], Sem, % 通常の受身に対応
        [{form/ga,sem/Obj},{form/ni,sem/Sbj}], Sem).

```

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

辞書の段階ではこのように曖昧でも、処理が進んでいく間に変数に別の制約が課せられ、それらの制約を解消することで曖昧性が減少していく。先の「はし」の例では、文の別の部分の処理において変数 OBJ が tool に具体化したとすると、その時点で制約が解消され、変数 TYPE も chopsticks に具体化することになる。従来の Prolog 等では、同音語の辞書はそれぞれの意味により別々の辞書エントリとして記述され、一つの可能性が失敗する度にバックトラックを行うことになり効率が悪い。同様に、動詞の活用語尾や、後置詞の接続の記述も扱っている。

8.2 JPSG の制約の記述

もう一つは、JPSG の句構造規則における構造原理を制約として埋め込むことができる。前述のように、局所的な枝分かれにおける文法範疇の各素性の関係を制約として記述できればよい。

JPSG では [2]、FFP(束縛素性原理) は次のようになっている。

局所的な枝分かれにおいて、親の束縛素性の値は、子の束縛素性の値の和と単一化する。

CHC を用いると、この原理は次のように句構造規則に埋め込むことができる。

```
psr([foot(MS)], [foot(LDS)], [foot(RDS)]); union(LDS, RDS, MS).
```

しかし、Prolog でこの種の制約を記述した場合に、実行は手続き的 (psr か union のどちらかが先に実行される) になってしまい、変数の具体化の状況によっては効率が悪くなる。

8.3 JPSG の素性

JPSG パーザでは以下の素性を取り扱っている。

(素性名)	(素性型)	(取り得る値)	(用途)
pos	主辞素性	n, v, p	品詞
infl	主辞素性	vc, vv, ...	品詞の活用の種類
form	主辞素性	root, sbj, obj, ...	品詞の活用形
gr	主辞素性	ga, wo, ...	名詞の文法関係
mod	主辞素性	+, -	修飾可能性

dep	主辞素性	カテゴリの主辞素性	修飾するカテゴリ
view	主辞素性	asp(B,E,D)	視野・アスペクト・参照時間
adjacent	Subcat 素性	カテゴリ	直前に来るカテゴリ
sc	Subcat 素性	カテゴリ	補語
slash	束縛素性	カテゴリの主辞素性	文のギャップに相当
pslash	束縛素性	カテゴリの主辞素性	resumptive pronoun にて生じる
refl	束縛素性	カテゴリの主辞素性	「自分」にて生じる
sem			意味

8.4 実行例

以下は、cu-Prolog による JPSG パーザの実行例である。「健が奈緒美を愛する」には曖昧性がないので対応する文法範疇に付随する制約は空である。「健が愛する」は「健が(tを)愛する」というギャップのある文、または「健が愛する(t)」という関係節、の二通りの読みがある。これらの曖昧性はトップレベルに付随する制約の複数の解として表されている。

tsuda#icot21[5]% cup3 % cu-Prolog の立ち上げ

```

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

_"jpsg/j4.p" % JPSG パーザファイルを読み込む
=== open 'jpsg/j4.p'

***** end of file *****
CPU time = 2.050 sec
_:-p([ken,ga,naomi,wo,ai,suru]). % 「健が奈緒美を愛する」の解析
% 構文木が返る
%% 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/[]}---[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/[]}---[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/[]}---[suff_p]
| | core/{form/vs2, pos/v}, sc/[], refl/Ref1AC_3702}}---[suff_p]
category= {sem/[love,ken,naomi], core/{form/Form_3670, pos/v},

```

```

sc/[], refl/[], slash/[], psl/[], ajn/[], ajc/[]}
% トップレベルのカテゴリ
constraint= syu_ren(Form_3670) % それに付随する制約 (終止形または連体
      % 形に対応する)
true.
CPU time = 0.750 sec
19%(program) 6%(pst/const) 12%(string)

_:-p([ken,ga,ai,suru]). % 「健が愛する」の構文解析

{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/Ref1AC_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/Ref1AC_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)
% トップレベルの制約を解く
_:-c31(_ ,Refl,_ ,Slash,_ ,SC,Obj,Sbj,_ ,_ ).
  Refl = [] Slash = [] SC = [{sem/V0_36, core/{form/wo, pos /p}}]
Obj = V0_36 Sbj = ken; % 解1:subcat が残っている
  Refl = [] Slash = [{sem/V3_58}] SC = [] Obj = V3_58 Sbj = ken;
% 解2:subcat から slash に要素が移っている
no.
CPU time = 0.050 sec
20%(program) 0%(pst/const) 13%(string)
-

```

9 おわりに

UNIX 版 cu-Prolog の実装は 1989 年から ICOT のサポートにより津田 宏・橋田 浩一により行なわれました。UNIX 版の著作権は ICOT にあります。Macintosh 版 (MacCUP) および MS-DOS 版 cu-Prolog は、UNIX 版の仕様を元に中京大学の白井英俊助教授により開発されました。JPSG パーザについては、大阪大学の郡司隆男主査をはじめとする ICOT PSG ワーキンググループ委員諸氏のコメント・協力をいただきましたここに感謝致します。

cu-Prolog および JPSG パーザについての参考文献を以下に紹介します。

cu-Prolog の元となった条件付単一化については [11, 3]。初期の論文としては [9, 13] が制約論理型言語の観点から、[10, 8] が JPSG パーザの観点からの論文です。

橋田浩一氏はさらに制約変換を一般的な計算原理とする DP(Dependency Propagation) 理論 [4] を提案していますが、それと cu とをからめた論文には [8, 5] があります。cu-Prolog に PST をデータ構造として導入することで、選言的素性構造を扱ったものには [12, 7] があります。

JPSG については [2]、単一化文法全般については [6] を参考にしました。

参考文献

- [1] Andreas Eisele and Jochen Dörre. Unification of Disjunctive Feature Descriptions. In *Proc. of 26th Annual Meeting of ACL*, pp. 286–294, June 1988.
- [2] Takao GUNJI. *Japanese Phrase Structure Grammar*. Reidel, Dordrecht, 1986.
- [3] Kôiti HASIDA. Conditioned Unification for Natural Language Processing. In *Proceedings of the 11th International Conference on Computational Linguistics*, pp. 85–87, 1986.
- [4] Kôiti HASIDA. Common Heuristics for Parsing, Generation, and Whatever. In *Workshop on Reversible Grammar in Natural Language Processing*, Berkeley, 1991.
- [5] Kôiti HASIDA and Hiroshi TSUDA. Parsing without Parser. In *Proc. of Second International Workshop on Parsing Technologies*, pp. 1–10. Sigparse ACL, February 1991.
- [6] Stuart M. Shieber. *An Introduction to Unification-Based Approach to Grammar*. CSLI Lecture Notes Series No.4. Stanford:CSLI, 1986.
- [7] Hiroshi TSUDA. cu-Prolog for Constraint-Based Grammar. In *Proceedings of Fifth Generation Computer System*, 1992.
- [8] Hiroshi TSUDA and Kôiti HASIDA. Parsing as Constraint Transformation – an Extension of cu-Prolog. In *Proceedings of the Seoul International Conference on Natural Language Processing*, pp. 325–331, 1990.
- [9] 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*, pp. 134–143. Springer-Verlag LNAI-485, 1989.
- [10] Hiroshi TSUDA, Kôiti HASIDA, and Hidetosi SIRAI. JPSG Parser on Constraint Logic Programming. In *Proc. of 4th ACL European Chapter*, pp. 95–102, 1989.
- [11] 橋田浩一, 白井英俊. 条件付単一化. コンピュータソフトウェア, Vol. 3, No. 4, pp. 28–38, 1986.
- [12] 津田宏. cu-prolog による選言的素性構造. 日本ソフトウェア科学会第 8 回大会論文集, pp. 505–508, 1991.
- [13] 津田宏, 橋田浩一, 白井英俊. 制約充足としての構文解析 — 制約論理型言語 cu-prolog の応用. 日本ソフトウェア科学会第 6 回大会論文集, pp. 257–260, 1989.