TR-I-0323

# プラン認識プログラム LAYLA

## (LAYered pLAn recognition system)

## - 利用マニュアル -

## User's Manual

大井 耕三　　西村 仁志*　　飯田 仁

Kozo OI　　Hitoshi NISHIMURA　　Hitoshi IIDA

1993.3

概要

本報告は、階層型プラン認識モデルの実装を行なったシステム LAYLA の利用方法について書いたものである。

LAYLA は従来のプラン認識システムに比べ、1) プラン探索のための制御機構を備え、2) 複数入力・複数目標を対象にしたプラン認識を行なうことができ、3) 概念シソーラスを利用した単一化を行なうことにより、より柔軟な認識を可能とする、といった特徴を持つ。本システムの実現により、プラン認識がより現実的な速度で行なうことが可能となる。

ATR 自動翻訳電話研究所

ATR Interpreting Telephony Research Laboratories



*日本電子計算株式会社

## 概要

本書は、階層型プラン認識モデルの実装を行なったシステムLAYLAの利用マニュアルである。LAYLAは従来のプラン認識システムに比べ、1) プラン探索のための制御機構を備え、2) 複数入力・複数目標を対象にしたプラン認識を行なうことができ、3) 概念シソーラスを利用した単一化を行なうことにより、より柔軟な認識を可能とする、といった特徴を持つ。本システムの実現により、プラン認識がより現実的な速度で行なうことが可能となる。

本書は利用マニュアルである。そのため、LAYLAの理論的説明について詳しくは述べていない。理論的説明については、飯田 仁, 有田 英一：“4階層プラン認識モデルを使った対話の理解”, 情処学論, **31**, 6, pp.810-821(1990-6) などに記述されているので、参照して欲しい。

LAYLAは汎用的なプラン認識プログラムであるが、本マニュアル中では対話構造の解析への応用を想定して記述している部分があるのでご留意願いたい。

本マニュアルの構成を以下に示す。

1章 システム概要 では、LAYLAの機能、動作環境、システム構成、ファイル構成、使用するデータについて説明している。

2章 利用方法 では、LAYLAで利用できる関数の使用方法が説明してある。

3章 操作例では、付録にあるサンプルデータを用いて、利用方法を具体的に説明してある。

4章 関数リファレンスでは、LAYLAの主な関数・マクロ・大域変数について説明してある。

付録 データ記述例では、LAYLAのサンプルデータが添付してある。

# 目次

# 第 1 章

# システム概要

本章ではLAYLAの機能、動作環境、システム構成、ファイル構成、使用するデータについて説明する。

## 1.1 LAYLA の機能

LAYLAは、入力となる行為に対し、システムが持つ知識を推論規則により連鎖していって、可能なプランの連結関係を求めるシステムである。

LAYLAは、階層型プラン認識モデル (1.3.2節参照) に基づいている。

LAYLAは一つの推論機構システムであるので、推論エンジンと知識ベースを持っている。そして、知識ベースがエンジンと独立しているので、知識ベースを問題に応じて用意することによりさまざまな問題に対処することができる。

## 1.2 LAYLA 稼働環境

LAYLAの稼働環境を表1.1に示す。

表 1.1: LAYLA 稼働環境

| | |
|---|---|
| マシン | リスプマシン、Sun |
| 使用言語 | Common Lisp |

## 1.3 LAYLA のモデル

### 1.3.1 プラン認識

プラン認識は、行為者の持つ知識を利用した一推論である。つまり、観察された入力に対して、システムが持つ知識を推論規則により連鎖していって可能な連結関係を求めることである。さらに、プラン認識の問題は次のように一般化される [8]:

1. 入力として行為の系列 (sequence of actions) を取る
2. 行為者の目標 (goal) を推論する
3. 行為の系列を構造 (plan structure) として構築する

### 1.3.2 階層型プラン認識モデル

LAYLA はプラン認識モデルを拡張した階層型プラン認識モデルに基づいている。階層型プラン認識モデルは、プラン認識に利用する知識を階層的に設定し、利用していくモデルである。1.3.1節で説明したプラン認識が、GPS(general problem solver) のような単純なプランニング (planning) の原理と考えれば、階層型プラン認識はその発展形であるいわゆる階層プランニング (hierarchical planning) に対応する。従って、従来のプラン認識に比べ効率的な処理が可能になる。

## 1.4 LAYLA の特徴

LAYLA は基本的に、階層型プラン認識モデルに基づいている。しかし、純粋な階層型プラン認識モデルに基づく実装では、現実的問題を扱う際に、効率性・処理対象範囲の点でさまざまな問題が出る。そこで、それらの問題に対処するために、以下のような機能を持っている:

1. 複数入力・複数目標の一括した取り扱い
2. 効率的プラン推論（探索）のための、推論制御機構 (1.9参照)
3. 現実的処理のための、単一化の拡張。(1.7.2参照)

## 1.5 システム構成

**LAYLA** の基本的システム構成を図 1.1に示す。**LAYLA** は、1.3節で説明したモデルに基づいている。システムへの入力は任意の観察された行為であり、出力はゴールスタック(目標構造＝理解状態) である。システムの主要な部分は、推論エンジン(inference engine)、プランスキーマ (plan schemata)、目標構造管理部 (GS manager) である。システムの基本的な動作は、推論エンジンが入力行為および目標構造管理部からその時点での目標構造を受けとり、それらの連鎖を求める。この時必要であれば知識ベース中のプランを参照する。以上により構築された新たな目標構造は、目標構造管理部へ渡され管理される。

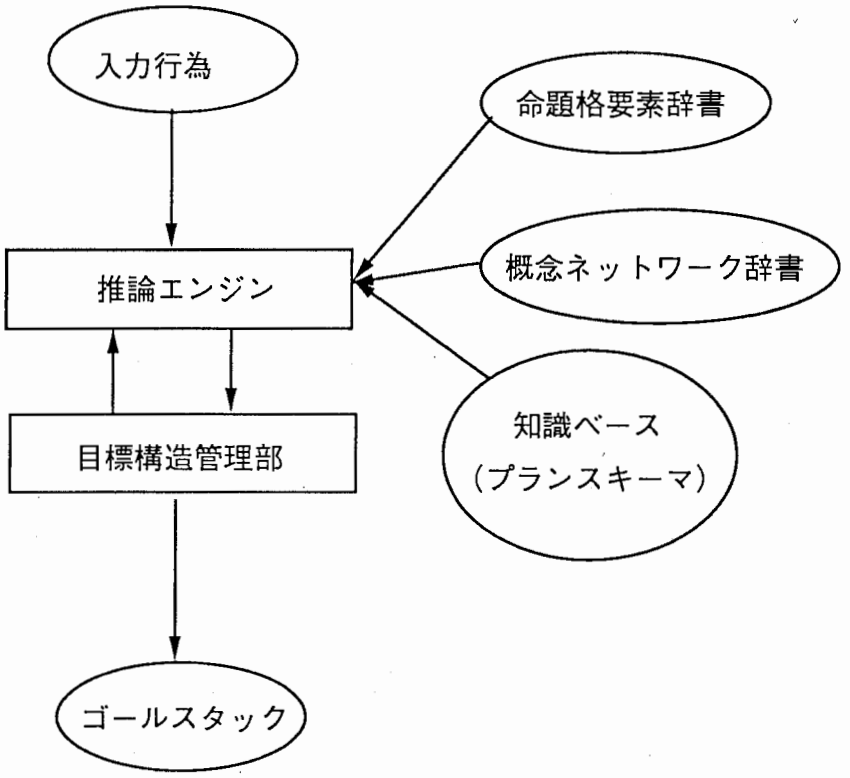


図 1.1: **LAYLA** のシステム基本構成

## 1.6 ファイル構成

LAYLA のファイル構成とその内容を表 1.2に示す。

表 1.2: LAYLA のファイル構成

| ファイル名 | 内容 |
|---|---|
| 主プログラム群 (``PlanRec/LAYLA/*'') | |
| load.lisp | LAYLA のロード、初期化関数 |
| unify.lisp | 単純な単一化 |
| unify2.lisp | 集合としての単一化、タイプ付変数 |
| chain.lisp | プラン推論、連鎖 |
| goal-stack.lisp | 目標構造管理 |
| decomposition.lisp | 推論規則 decomposition chain |
| precondition.lisp | 推論規則 precondition chain |
| effect.lisp | 推論規則 effect chain |
| 入出力・知識ベース (``PlanRec/LAYLA/*'') | |
| schema.lisp | プランスキーマの定義 |
| input.lisp | 入力インタフェイス |
| control.lisp | 制御オブジェクト操作とヒューリスティクス |
| ユーティリティ (``PlanRec/LAYLA/*'') | |
| tools.lisp | 便利な関数 |
| tree.lisp | 木構造表示、マトリクス・木構造変換 |
| display.lisp | プラン認識デバッグ用表示 |
| 概念検索 (``PlanRec/NP/*'') | |
| load.lisp | 検索機構のロードファイル |
| network.lisp | ネットワークデータ構成と検索機構 |
| display-network.lisp | ネットワークの表示 |
| ドキュメント (``PlanRec/Doc/*'') 本マニュアル (LATEX で記述) | |

表 1.3: サンプルデータ (付録)

| ファイル名 | 内容 |
|---|---|
| 主プログラム群 (``PlanRec/Data/*'') | |
| case-grammar.lisp | 命題格要素辞書 |
| concept-nodes.lisp | 概念ネットワーク辞書 |
| kaiwa-*.data | 入力行為データ |
| simple.plans | プランスキーマ 1(目標に関する行動の規定の記述が、副行為 (decomposition chain) のみのプランスキーマの例。推論制御モードがシンプルモードのとき使用する) |
| test.plans | プランスキーマ 2(simple.plans を含む。推論制御モードがシンプルモード以外のとき使用する) |

## 1.7 LAYLA で扱うデータ

### 1.7.1 入力データ

- 入力行為
  任意の観察された行為

### 1.7.2 知識ベース

- 命題格要素辞書

  命題 (2.5.1参照) のうち、グラフユニフィケーション (命題間の格要素の数が異なっていても単一化できる) ができる命題を指定する辞書。
  例:
  命題格要素辞書に
  (申し込む -1 AGEN RECP OBJE)
  と書いておけば、

```
(unify '(x y (申し込む -1 ?A ?B ?C)) '(x y (申し込む -1 ?s)))
  => '(((A . ?s) (B . nil) (C . nil)))

(unify '(x (y (申し込む -1 ?A ?B))) '(x (y (申し込む -1 ?s ?t ?u))))
  => '(((A . ?s) (B . ?t) (u . nil)))
```

  となり、単一化できる。(書いておかないと単一化できない) つまり、この場合、単一化しようするものに、(申し込む -1 .....) があり、申し込む -1 の格要素の数が 3 個以内であれば、成功するということである。

- プランスキーマ

  プランスキーマは、プラン (plan)の記述である。プランとは、ある目標実現に関する知識である。LAYLA は、入力行為に対してシステムが持つ推論規則によりこのプランを連結していって、可能な連結関係を求めている。

  プランは、行為 (action) と状態 (state)から記述する。つまり、ある目標実現に必要な (他の) 行為やあるべき状態を規定したのもがプランである。プランが実現する目標そのものも、しばしば、行為と呼ぶことがある。つまり、「何々をするために、、、」などという言語表現は、目標が行為であり、その後に続く内容を含めて文全体がプランを表現していることになる。従って、プランは、ある行為についての知識の記述であるともいえる。

  例えば、積木を他の積木の上に乗せたい時、その積木を摑んで運んで目標の積木の上に置けば良い。その時、それをするためには、双方の積木の上に何かが載っていたり、摑むための手が塞がっていたりする状態では、直接 (すぐに)

その行為を実行できない。それでも、目標達成を目指すのであれば、これらの状態を満たすようにする他のプランを参照して、そのプランから実行すべきである。

プランは、このようなある目標に関する行動や状態の規定を記述する。その典型的内容は:

**見出し (header)**: 目標の行為の記述 (e.g. ある積木を目標の積木の上に積む)

**副行為 (decomposition)**: 見出し達成するためにしなければならない行為の系列 (e.g. その積木を摑む、運ぶ、置く etc.)

**前提条件 (precondition)**: 見出しが実現できるための前提条件となる状態 (e.g. (双方の) 積木の上にものがない、手が空いている etc.)

**効果 (effects)**: 見出しの実現により変化する状態 (e.g. 目標の積木の上に積木がある、積木が元の場所にはない etc.)

システムでは以上のようなプランの内容を、スキーマ (**プランスキーマ**) として記述しておく。

プランスキーマの知識ベースへの登録は、マクロを用いる。スキーマは、type の値により、登録先知識ベースが判断され、type で階層化される。なお、階層の順序は検索対象モード (1.9.2参照) で指定することができる。 (**defschema**(*Macro*)2.5.2節参照)

- 概念ネットワーク辞書

  集合としての同一性による単一化を行なうとき参照する知識ベースである。たとえば、つぎのように用いる。同一概念を指す言葉でも異なる表現が用いられることがある。そのため、記号の字面のみでその評価を行なってしまえば、"お名前" と "氏名" を単一化できない。しかし、同じ概念を同一視する方が都合が良いことが多い。そのような時、それらを単一化したいのなら、概念ネットワークに "お名前" と "氏名" を is-a リンクで結んでおけばよい。

  このように、概念ネットワークは同一概念のことなる表現間を単一化するための知識ベース (シソーラス)[14] である。

図 1.2に概念ネットーワーク辞書を視覚化した一例を示す。この図では、"送り先 (destination)", "住所 (address)", "名前 (name)" といった概念とそれらを表現するための言語表現の関係が示されている。ネットワークは、ノード (ラベル) とリンクで構成する。リンクには、任意の関係が記述できる[1]。

[1] しかし、現在の実際の単一化処理では、is-a リンクしか使用していない。

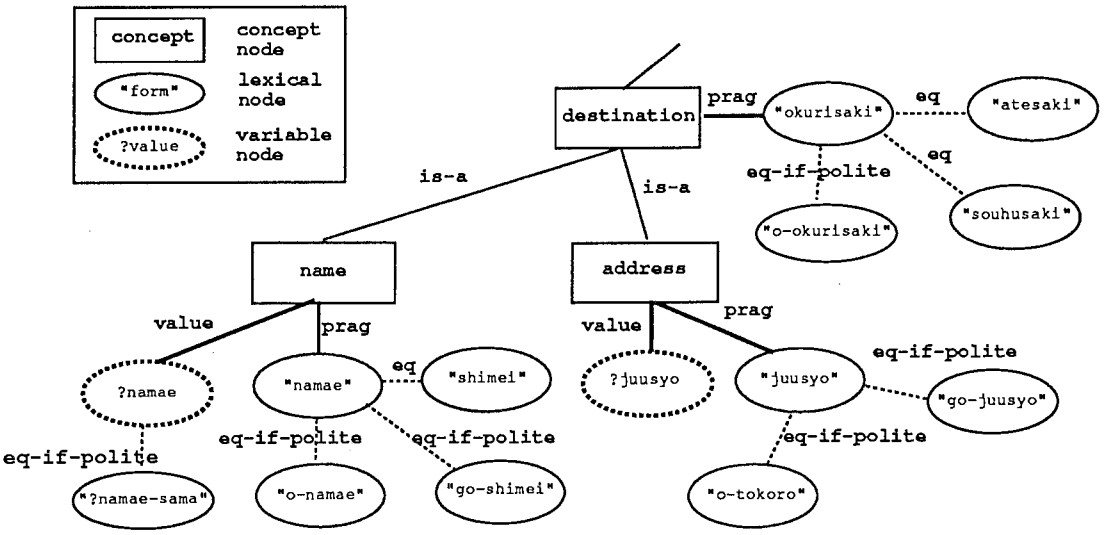


図 1.2: 概念ネットーワーク辞書

### 1.7.3 出力データ

- ゴールスタック (目標構造 = 理解状態)
  ゴールスタックは、目標構造を保持・管理する構造体である。その内容は、表 1.4に示す 4 つのリストよりなる。各々のスタックには、プランスキーマあるいは状態記述 (プランスキーマの effect スロットに記述される list のみ) が要素としてはいる。

表 1.4: 構造体ゴールスタックのスロット

| スロット名 | データタイプ | 内容 |
|---|---|---|
| incomplete | list of actions | 未充足プラン |
| complete1 | list of actions | 充足 (完了) プラン |
| complete2 | list of actions | 未充足かつ談話セグメント完了プラン |
| statements | list of lists | 共通理解事項 (状態記述) |

incomplete は、未充足のプランスキーマを格納するプッシュダウンスタックである。complete1 は、すべてのスロットが充足されたプランスキーマを格納するプッシュダウンスタックである。(発話を表すアクションは常にここにはいる。) また、complete2 は、プラン自体は未充足であるが、既に談話セグメント (discourse segment)[7] が他へ移ってしまったものを格納しておく。さらに、statements には、充足されたプランの効果を格納しておく。これらは、残りの対話において共通理解事項として扱うことができる。

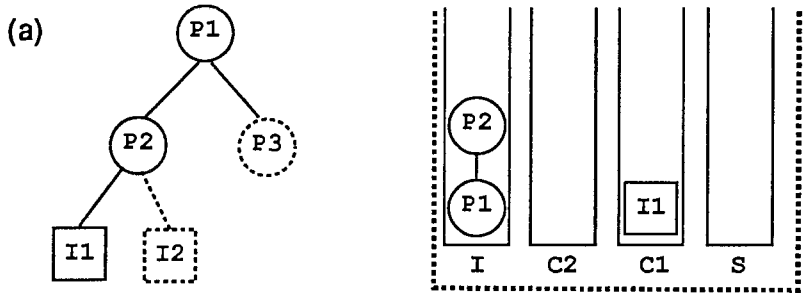


図 1.3: ゴールスタックのイメージ

ゴールスタックの内容と解釈イメージの簡単な例を、図 1.3に示す。図中、右の木構造は目標構造のイメージを、左の箱は構造体ゴールスタックのイメージを表している。各ノードは丸がプラン、四角は入力行為を表しており、実線は具現化 (instantiate) されているもの、点線はまだいずれへの連鎖も行なわれていないものを示している。また、リンクは連鎖を表す。また、構造体イメージのスタックの下の文字は、各スロットの頭文字をとっている。

この図は、入力 $I_1$ が観察され、プラン認識を行なった結果の目標構造（の一つ）である。この構造は、最終的目標 $P_1$ のもとで、その成就のためのプラン $P_2$ の一部を実現する行為 $I_1$ が観察されていると解釈できる。入力行為 $I_1$ は、その定義から完成（充足）されたアクションであるので、complete1 スタック **C1** にある。具現化されたプラン $P_1$ と $P_2$ は、そのスロットに未充足のアクション $I_2$ を持つので、incomplete スタック **I** にある。この後、図の $P_2$ に対応するような行為が観察されれば、$P_2$ は **C1** に移る。そうではなく、$P_3$ に連鎖するような行為が入力された時は、新たに $P_3$ が **I** にプッシュされる。そののち、$P_2$ が充足されず、$P_3$ の充足により **P1** が充足されたと判断されれば[2]、$P_2$ は **C2** に移される。

## 1.8 プラン認識を行なうためのアルゴリズム

### 1.8.1 推論規則

プラン認識は、推論メカニズムである。従って、推論規則を持つ。その推論規則を Allen[9] に従って説明する。

[2]このプラン充足基準は微妙である。しかし、現在のインプリメンテーションでは、この基準を採用している。すなわち、ここでのプラン充足基準は、あるプランの全ての下位行為の完全な充足ではなく、全ての下位行為が具現化されかつ最終的に焦点の当たっている下位行為が充足された時に、そのプラン自体も充足されたと考える。これは、協調的な問題解決においては、なるべく順序だって作業が勧められるであろうというヒューリスティックな判断からである。もしそうでなくても、**C2** を **I** と同様に扱う、もしくは clue (「先ほどの件ですが、」) などの手がかりによる処理により、この手の問題は解決できる。

Allen によれば、推論規則は行為および状態間の因果関係 (causal relationship) である。

以下に、**LAYLA** で採用している推論規則を 1.7.2節のプランの表現に基づいて示す。[1]。

- decomposition chain
  ある行為の上位プランへの連鎖である。プランの見出しが他のプランの副行為スロットの要素である時、これらは decomposition chain による連鎖が可能になる。

  例えば、電話の中で話し相手の住所を聞き出すためには、何らかの質問を発する必要がある。従って、住所を尋ねる質問は、その発話者が聞き手の住所を聞き出す（知る）ための副行為であり、例えば、「ご住所をお願いします。」という発話は、「話し手が聞き手の住所を知る」というプランへの decomposition chain が可能である。

- effect chain
  ある行為の実現が上位プランの効果となる時の連鎖である。行為の内容（行為の内容から導き出される状態）が他のプランの効果スロットの要素である時、これらは effect chain による連鎖が可能になる。

  例えば、「会議に参加したいのですが。」という表明は、発話者の会議参加という行為が発話の内容として示されている。従って、この発話は、会議に参加できる状態を効果を持つ、「会議参加」のプランと effect chain が可能になる。[3]

- precondition chain
  あるプランの効果と、上位のプランの前提条件スロットの状態への連鎖である。

  例えば、「聞き手の送り先を知っている」という状態は、「聞き手に登録用紙を送る」ことの前提条件であるから、「話し手が聞き手の住所を知る」というプランと「話し手が聞き手に登録用紙を送る」プランは precondition chain が可能である。

### 1.8.2 プラン認識アルゴリズム

プラン認識の基本的な考え方は、観察された入力に対して、システムが持つ知識を推論規則により連鎖していって、可能なプランの連結関係を求めることである。プラン認識メカニズムの最終的停止条件は、システムの持つ知識（プランスキーマベース）の中に、連鎖可能なプランが存在しなくなった時である。結果は、それまでに構

[3]対話処理におけるプラン認識の effect chain としては、実際のところ、例の発話のような、希望の表明に対して適用されるのがほとんどである。一方で、effect chain は、（実際のプログラム中では）注意深く利用しないと、組合せの爆発を招くことになりかねない。

築された入力とプランの連結関係であり、それ以上上位のプランに連鎖を求められないプランの見出しなる行為がその入力の最終的目標と考えられる。

この最終的目標となる行為を、逆に、展開していけば、その行為を可能とする行為の連鎖が求められることになる。Allen の用語によれば、これらの行為を期待行為 (expectation) と呼ぶ。プラン認識の多くの問題は、この期待行為に対して、入力からの連鎖を求めることになる。例えば、上の例では「ご住所をお願いします」という発話は「話し手が聞き手に登録用紙を送る」という目標の元に発せられたことがわかる。この時、「ご住所をお願いします」という発話から「話し手が聞き手の住所を知る」というプランを介して、「話し手が聞き手に登録用紙を送る」というプランへの連鎖の構造が求められる。これらのプランの記述中で、今だ実現がなされていない (他のどの観察された行為やそこから連鎖可能なプランに連鎖していない) 副行為の記述が、期待行為となり得る。従って、「聞き手が住所を答える」という発話や、「話し手が登録用紙を送ることを約束する」などという行動は、期待行為となり得る。

以下に上の作業を簡単にまとめたプラン認識のアルゴリズムを示す:

1. 入力から各期待行為に対して、推論規則を適用して連鎖を求める。(直接連鎖)
   成功すれば 3 へ。

2. 1 で全ての期待行為に対して (直接) 連鎖が求まらなければ、システムの持つ知識ベースから入力と推論規則により連鎖可能なプランスキーマを求める。さらに求めたプランスキーマを新たな入力として、1 を行なう (再帰処理)。(間接連鎖)
   知識ベース中に連鎖可能なスキーマがなければ、失敗終了。

3. 上の処理により連鎖に成功したプランの間の変数や状態記述の整合をとる。さらに、求められたプランの副行為を新たな期待行為として登録する。

### 1.8.3 階層型プラン認識アルゴリズム

LAYLA では、知識ベース (プランスキーマ) の階層性 (1.3.2参照) を扱うために、前節のプラン認識アルゴリズムを拡張している。基本的には、ステップ 1 と 2 を拡張している。

1. 直接連鎖の対象となる期待行為の提出順序を、プランの階層順序にしたがったものにする。

2. 間接連鎖の際の知識ベースの探索を、プランの階層順序にしたがったものにする。

実装としては、A* アルゴリズムを基本にして、それを拡張している[4]。(A* の詳細や議論については [11] を参照)

[4]現在の LAYLA では、ヒューリスティクスや厳密な推測値を用いていないので、A アルゴリズムの拡張ともいえるが、実際の実装では、推測値が利用できるようにしてある。

拡張は、次の 2 点である:

1. 階層化された知識ベースの探索を可能にする、

2. 複数入力行為、複数目標行為の取り扱いを可能にする。

## 1.9 プラン認識アルゴリズムの制御

### 1.9.1 探索制御の概要

プラン認識アルゴリズムは、基本的に探索アルゴリズムとして考えることができる。すなわち、入力行為を推論規則により展開してできる木構造の中で、目標行為に至る可能な行為の系列を探し出すことである。探索問題には、組合せ的爆発が伴う。従って、実用的なシステム構築においては、探索の効率化が課題となる。

この問題を解決する手段として、ヒューリスティクスすなわち問題領域における知識の利用がある。**A*** アルゴリズムにおける代表的なヒューリスティクスの利用方法は、開ノードリスト $OPEN$ の評価において利用される。しかしながら、多様性を含む問題を考えた時、利用される知識を、初めから完全に記述し尽くすことは困難である。ゆえに、問題領域における特徴的な知識を効果的に獲得することが重要となる。ただ単純に知識を拡張したり、複雑化したのでは、再び組合せ的爆発の危機にさらされたり、知識適用条件評価のための計算コストの増大を招く。

一方、推論過程そのものを制御することにより、無駄な探索を防ぐことが考えられる。LAYLA の探索制御は、この立場をとるものである。つまり、探索制御規則を取り扱うことのできる推論機構を準備し、状況に依存した推論を行なおうとする立場である。前段で述べたようなヒューリスティクスの分析・獲得は、ここでいう探索制御規則を問題に応じて定義することになる。なお、推論制御そのもののメカニズムは、利用される知識の性質に依存しない。

### 1.9.2 制御の形態

図 1.4に、1.8節で示した階層型プラン認識メカニズムの概要を示す。図の内容を説明する。図中の円は **A*** のノードを示す。ノードの内容は、構造体アクションである。入力行為に対して複数の解釈 (interpretations of input: $I_k$, k は解釈に付けられた番号) が可能である。システムの保持する理解状態 (understanding state) も複数である。一つの理解状態は、構造体ゴールスタック (Goal stack: $GS_j$, j は理解状態の中でゴールスタックに付けられた番号) で保持されている。(図中の一本の箱は、便宜上そのゴールスタックの incomplete スタックを表しているものと考える。従って、スタック中の円は、期待行為 = 目標 (goal: $G_{i,j}$, i は incomplete スタック内の期待行為に付けられた番号) と考えることができる。) プラン認識は、ある入力の円から、ある期待行為の円への連鎖を求めることであり、これには直接連鎖 (direct chain) と知識ベース中のプランスキーマ ($P_x$) を介する間接連鎖 (indirect chain) がある。階層型知

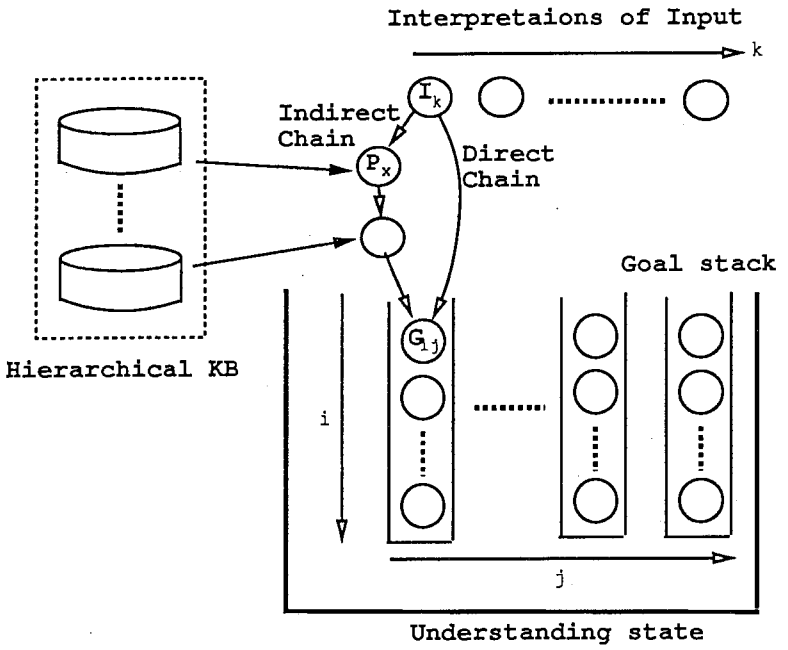


図 1.4: 階層型プラン認識メカニズムの概要

識ベース (hierarchical KB) には、階層クラス分けされたプランスキーマが格納されている。

以下に、図のような構成におけるプラン認識の探索制御モードを説明する。() 内は、モードの値である。

**適用推論規則モード [Inference Rules ]** (推論規則を表すシンボルのリスト)

あるプラン推論において適用する推論規則を指定する。また、リストの要素の並びにより、規則適用順序も指定できるようにする。(default として全ての推論規則をとる)

例えば、この値が (list effect-chain decomposition-chain) であれば、まず、effect-chain により連鎖の成否を求める。もしそれが失敗した時は、decomposition-chian により試みる。この値では、precondition-chain は適用しない。

**検索対象モード [Schema Type ]** (スキーマタイプのリスト)

検索を行なう知識ベースの属性を指定する。default として処理入力となっている行為の属する上位クラスのリスト (入力自身のクラスを含む) を取る。

**階層モード [Layer Mode ]** (t,nil)

検索対象モードの値で指定された知識ベースを、マージして検索を行なうか (nil)、階層順序に従って行なうか (t) を指定する。

**入力順序モード [Input Order ]** (t,nil)
複数入力に対して、ある順序にしたがって推論を進めるか (t)、すべての入力を同時に進めるか (nil) を指定する。

これは、例えば発話解釈において、文解析結果に何らかの尤度が与えられとき有効となる。

**対象ゴールモード [Goal Direction ]** (BR(j 優先),DP(i 優先)
目標構造管理機構で管理されている理解状態について、目標構造 1 つずつに対して推論を行なうか (i 方向優先)、すべての目標構造を同時に行なうか (j 方向優先) を指定する。

前者は、目標構造間に解釈に何らかの順位が設定できる時有効であり、その時理解状態内のゴールスタックの順序を指定することにより、優先処理を可能にする[5]。

**直接間接モード [Chain Mode ]** (:direct,:indeirect)
ある理解状態に対して、まずすべての目標に対して直接連鎖を求めたのち間接連鎖を求めるか (direct)、直接・間接連鎖双方を各目標毎に行なうか (indirect) を指定する。

例えば、解釈の自由度が高い「わかりました。」のような発話が入力である場合は、 direct により、直接連鎖できるものを処理する方が効率的である。

**間接連鎖回数 [Indirect Depth ]** (正数)
間接連鎖の際に仲介するプランスキーマのインスタンスの最大数 (図の $P_x$ の $x$) を指定する。

対話理解においては、この数字が大きいほど間接的な解釈、つまり婉曲的な言い回しを含む対話が処理できるようになる。

**最短優先モード [First Hit ]** (t,nil)
連鎖の際に、解が見つかったところでそれ以降の処理を行なわないようにするか (yes)、あるいは間接連鎖回数の範囲すべてを検索するか (no) を指定する。

**シンプルモード [Simple Mode ]** (t,nil)
プラン推論において適用する推論規則を、目標に関する行動の副行為 (decomposition chain) のみとし、間接連鎖回数は 1 以下であるようにする。

プラン認識の推論過程は、上記のモードの設定により多様に変化することになる。

1.8節で示した階層型プラン認識アルゴリズムでの制御モードのデフォルト値は、次のとおりである。

---

[5]現在の LAYLA の実装では、この目標構造に対する優先順位は設定していない。

```
> (display-mode)
--- Control Modes of 0 ---
Inference Rules      = (DECOMPOSITION-CHAIN EFFECT-CHAIN PRECONDITION-CHAIN)
Schema Typee         = (INTERACTION-PLAN COMMUNICATION-PLAN
                        DOMAIN-PLAN DIALOGUE-PLAN)
Layer Mode           = T
Input Order          = NIL
Goal Direction       = BR
Chain Mode           = DIRECT
Indirect Depth       = NIL
First Hit            = NIL
Simple Mode          = NIL
Trace                = NIL
NIL
>
```

## 1.10 関連システム

LAYLA は一つの推論機構として独立したシステムである。しかし、本来、LAYLA は協調的目標指向型対話の対話構造を解析し、その結果を用いて次発話を予測する対話理解システムのサブシステムとして作成された。そこでの LAYLA の位置を図 1.5に示す。

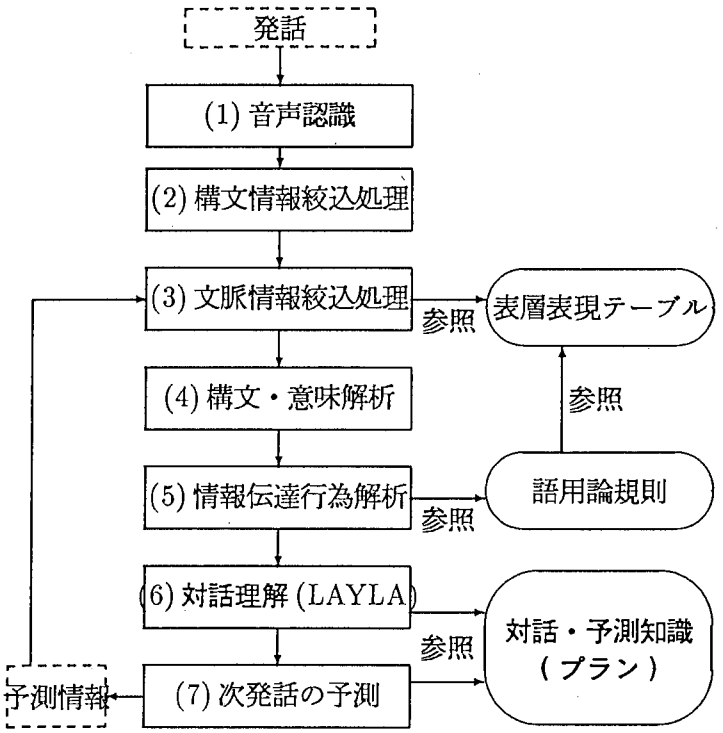


図 1.5: 音声言語処理システムの構成

# 第 2 章

# 利用方法

本章では、第1章で述べた機能を実装したシステム LAYLA の利用方法について説明する。

## 2.1 LAYLA のインストール

LAYLA のインストールは、以下の通りである。

1. システムにログインする
   login: ユーザ名
   password: パスワード

2. LAYLA をインストールするディレクトリへ移動する
   cd ディレクトリ名

3. MT から DISK へコピーする
   tar xvf デバイス名

4. パス名をセットする。
   ファイル **LAYLA**/load.lisp を編集し、*PlanRec-path* の値を 3でコピーした LAYLA のディレクトリ名にする
   (defvar *PlanRec-path* “ディレクトリ名 /PlanRec/”)

## 2.2 LAYLA の起動の準備

1. LISP を立ち上げる

2. **LAYLA** のディレクトリへ移動する
   (cd “ディレクトリ名 /PlanRec/”)

3. **LAYLA** をロードする
   (load “load.lisp”)

4. パッケージを:gplanner にする
   (in-package :gplanner)

5. **LAYLA** をコンパイルする。(この処理は最初の準備のときのみ)
   (compile-layla)

## 2.3 データの用意

1. 入力データ、知識ベース (命題格要素辞書、プランスキーマ、概念ネットワーク辞書) を用意する。(2.5参照)。

2. 定義したデータのファイル名を変数にセットする。
   load-data.lisp を編集する。(表 2.1参照)

3. データファイルをロードする。
   (load "load-data.lisp")

表 2.1: データのファイル名をセットする変数

| 変数名 | 説明 |
|---|---|
| *data-path* | プランスキーマなど、データ・知識を記述したファイルのあるディレクトリ |
| *schemata-filename* | プランスキーマを記述したファイル名 |
| *np-concept-filename* | 概念ネットワーク辞書のファイル名 |
| *proposition-grammar-filename* | 命題格要素辞書のファイル名 |
| *kaiwa-input-filename* | 入力行為データのファイル名 |

## 2.4 システムの操作

### 2.4.1 プラン認識を行なう手順

1. *filename*のファイルから入力命題をロードする。
   (load-input-actions *&optional (filename* *kaiwa-input-filename**))

2. layla を初期化する。
   目標構造・入力履歴の初期化を行なった後、プランスキーマ・概念ネットワーク辞書をロードする。(知識ベースは全てデフォールトのファイルをロードする)

   (init-all)

3. プランニングの制御モード (表 2.2参照及び 1.9.2節参照) を指定する。

4. プラン認識を 1 つ進める。
   (plan-inference-next)

5. 現在の理解状態の概観を *stream*へ表示する。
   (tprint-gsl *&optional (stream t)*)

表 2.2: 制御モードを指定する関数

| 指定するための関数 | 説明 |
|---|---|
| (set-inference-rules rules) | 適用推論規則モード rules をセットする |
| (set-schema-type schema-type) | 検索対象モード schema-type をセットする |
| (layer-mode-on) | 階層モードを layered にする。 |
| (layer-mode-off) | 階層モードを single にする。 |
| (input-order-on) | 入力順序モードを sequential にして、入力間の優先順位を設定した処理を行なう。 |
| (input-order-off) | 入力順序モードを parallel にする。 |
| (goal-direction-br) | 対象ゴールモードを br にする。 |
| (goal-direction-dp) | 対象ゴールモードを dp にする。 |
| (chain-mode-direct) | 直接間接モードを direct にする。 |
| (chain-mode-indirect) | 直接間接モードを indirect にする。 |
| (set-indirect-depth num) | 間接連鎖回数 num を設定する。 |
| (first-hit-on) | 最短優先モードで処理を行なう。 |
| (first-hit-off) | 最短優先で行なわない (すなわち求められる範囲のものは全て求める) |
| (simple-mode-on) | シンプルモードにする。 |
| (simple-mode-off) | シンプルモードでないようにする。 |
| (trace-on) | トレース表示を行なうようにする。 |
| (trace-off) | トレース表示を行なわない。 |

### 2.4.2 その他の主な操作

- 知識ベースのロード
  - プランスキーマ (*file*) をロードする。
    それまでのプランスキーマはクリアされる。
    (load-schemata *&optional (file *schemata-filename*)*)
  - 概念ネットワーク辞書をロードして、概念ネットワークを作る
    (load-concept-network *&key file package*)

- 知識ベースの表示
  - プランスキーマテーブルの内容全体を *stream*へ表示する.
    (print-schemata *&optional (stream t)*)

- 概念ネットワークの全てのノードを端末出力へ木構造で表示する。
  (np::pprint-all-nodes *&key (start :bottom) (type-list nil) (network :concept)*)

- プランニング処理

  - プランニングの制御モードを表示する。
    (display-mode)
  - *ID*(2.5節参照) で指定された入力を処理した時点の状態に戻す。
    (reset-gplanner*&optional ID*)

## 2.5 データ記法

### 2.5.1 記号の定義

- シンボル (*symbol*) は、"?" 以外で始まる LISP で利用できるシンボルに等しい。
- 任意変数 (*free-var*) は、"?" で始まる LISP シンボルで表す:

  *free-var ::= ?symbol*

- タイプ付変数 (*typed-var*) は、"?" で始まり、その直後に LISP リスト (その要素はシンボルのみ) をとる:

  *typed-var ::= ?(label type)*
  *label ::= symbol*
  *type ::= ≪concept-name≫*

  特に指定がなければ、任意変数・タイプ付変数を総じて変数 (*var*) と呼ぶ。

  *var ::= free-var | typed-var*

- リスト (*list*) は、シンボル・変数・リストを要素としてとる LISP リストである。
- 命題 (*prp*) は、特定の指定された述語 (*pred*) で始まるリストである。

  *prp ::= (pred case)*
  *pred ::= symbol*
  *case ::= ≪ concept | var | prp ≫*

  命題のみを要素とする LISP リストを、複合命題 (*complex-prp*) と呼ぶ。命題は、行為の見出しや状態を表現するために利用する。

  例:

```
(ASK-STATEMENT      SP2 SP1 ?TPCD1-5
                      (だ -IDENTICAL ?OBJED1-5 用件 -1)
                      "どのようなご用件でしょうか")
```

### 2.5.2 データの定義方法

1. 概念の定義 (A.2参照)

   - 概念 (*concept*) は、概念ネットワーク辞書で管理されるデータであり、ラベルとリンクから構成される。

     *concept* ::= *('defconcept' concept-name link)*
     *concept-name* ::= *symbol*
     *link* ::= ≪*(link-name concept-name)*≫
     *link-name* ::= *symbol*

     'defconcept' は概念ノード定義用のマクロである。

   - 記述例:

     ```
     (np::defconcept  住所 -1
                 (is-a 送り先 -1))
     ```

     この例は、概念 "住所 -1" は概念 "送り先 -1" の is-a 関係による下位概念であることを表す。

2. 命題格要素の定義 (A.3参照)

   - 命題格要素の定義は述語 (*pred*) と格要素 case (ここでの case は変数のみである) で指定されたリストである。

     *prp* ::= *(pred case)*
     *pred* ::= *symbol*
     *case* ::= ≪ *var* ≫

   - 記述例:

     ```
     (だ -IDENTICAL    OBJE IDEN)
     ```

3. 入力行為の定義 (A.1参照)

   - 入力行為 (*input*) は、ユニークな ID を第 1 要素とする命題 (および複合命題) の LISP リストである:

     *input* ::= *(ID* ≪ *prp* | *complex-prp* ≫*)*
     *ID* ::= *symbol*

   - 記述例:

```
(D1-5
  (
    (ASK-STATEMENT      SP2 SP1 ?TPCD1-5
                        (だ -IDENTICAL ?OBJED1-5 用件 -1)
                        "どのようなご用件でしょうか ")
  )
)
```

4. プランスキーマの定義 (A.4参照)

   - プランスキーマは、以下のように記述する (内容については、 1.7.2節参照):

     *schema* ::= *('defschema' type body)*
     *type* ::= *symbol*
     *body* ::= *"(':HEAD' prp ≪slot≫ )"*
     *slot* ::= *slot-name slot-body*
     *slot-name* ::= *':DECO', ':PREC', ':EFFE', ':DELE', ':CONS'*
     *slot-body* ::= *( ≪prp≫ )*

     'defschema' はスキーマ定義用のマクロである。

   - 記述例:

```
(gplanner::defschema :DOMAIN-PLAN
  "(
  :HEAD (SEND-SOMETHING ?AGN ?RCP ?(SOMETHING object))
  :PREC ((HAVE ?AGN ?SOMETHING)
         (KNOW ?AGN (is ?(DEST 送り先 -1) ?(VAL pronoun))))
  :DELE ((HAVE ?AGN ?SOMETHING))
  :EFFE ((HAVE ?RCP ?SOMETHING))
  :DECO ((INTRODUCE-ACTION ?AGN ?RCP ?TPC (SEND ?AGN ?RCP ?SOMETHING)))
  :CONS ()
  )"
)
```

     この例は、対話において「(AGN が RCP に) 何か (SOMETHING) を送る」ことを実現するためのプランである。'()' は nil として解釈する。

# 第 3 章

# 操作例

Appendix としてついているサンプルデータ (入力行為データ、概念ネットワーク辞書、命題格要素辞書、プランスキーマ) を使い操作例を示す。

なお、本例で使用したマシンは SunSPARCstation2 であり、 LISP は nemacs から Lucid Common Lisp を起動して使っている。

## 3.1 LAYLA をインストールする

1. システムにログインする
   login:LAYLA
   password:LAYLA のパスワード

2. LAYLA をインストールするディレクトリへ移動する
   cd /home/usr

3. MT から DISK へコピーする (カレントディレクトリの下にディレクトリ PlanRec が作られ、コピーされる)
   (デバイス名は /dev/rst0 とする)

   tar xvf /dev/rst0

4. パス名をセットする。
   ファイル **LAYLA/load.lisp** を編集し、 *PlanRec-path* の値を 3でコピーした LAYLA のディレクトリ名にする

   (defvar *PlanRec-path* "/home/user/PlanRec/")

## 3.2 LAYLA の起動の準備

1. LISP を nemacs から起動する

   `m-x lucid` を入力すると次のメッセージが表示される。

```
Starting /usr/local/bin/lisp ...
;;; Sun Common Lisp, Development Environment 4.0.0 , 6 July 1990
;;; Sun-4 Version for SunOS 4.0.x and sunOS 4.1
;;;
;;; Copyright (c) 1985, 1986, 1987, 1988, 1989, 1990
;;;            by Sun Microsystems, Inc.,  All Rights Reserved
;;; Copyright (c) 1985, 1986, 1987, 1988, 1989, 1990
;;;            by Lucid, Inc., All Rights Reserved
;;; This software product contains confidential and trade secret
;;; information belonging to Sun Microsystems, Inc.  It may not be copied
;;; for any reason other than for archival and backup purposes.
;;;
;;; Sun, Sun-4, and Sun Common Lisp are trademarks of Sun Microsystems Inc.

>
>
```

2. LAYLA のディレクトリへ移動する

```
> (cd "/home/user/PlanRec/")
#P"/home/user/PlanRec/"
>
```

3. LAYLA をロードする

```
> (load "load")
;;; Loading source file "load.lisp"
;;; Warning: File "load.lisp" does not begin with IN-PACKAGE.  Loading
into package "USER"
;;; Loading source file "LAYLA/test.lisp"
;;; Loading source file "NP/load.lisp"

                --- 中略 ---

;;; Loading source file "LAYLA/precondition.lisp"
;;; Loading source file "NP/network.lisp"
;;; Loading source file "NP/display-network.lisp"
#P"/home/user/PlanRec/load.lisp"
>
```

4. パッケージを:gplanner にする

```
> (in-package :gplanner)
#<Package "GPLANNER" 6344B6>
>
```

5. LAYLA をコンパイルする。(この処理は最初の準備のときのみ)

```
> (compile-layla)
;;; You are using the compiler in development mode (compilation-speed = 3)
;;; If you want faster code at the expense of longer compile time,
;;; you should use the production mode of the compiler, which can be obtained
;;; by evaluating (proclaim '(optimize (compilation-speed 0)))
;;; Generation of full safety checking code is enabled (safety = 3)
;;; Optimization of tail calls is disabled (speed = 2)
;;; Reading source file "Utility/utility.lisp"
;;; While compiling CHANGE-EJ

                --- 中略 ---

;;; Reading source file "LAYLA/precondition.lisp"
;;; Writing binary file "LAYLA/precondition.sbin"
;;; Loading binary file "LAYLA/precondition.sbin"
NIL
>
```

## 3.3 データの用意

1. 入力データ、知識ベース (命題格要素辞書、プンスキーマ、概念ネットワーク辞書) を用意する。

   ここでは、付録のサンプルデータを使用する。

2. 定義したデータのファイル名を変数にセットする。
   ここでは、付録のサンプルデータを使用するので、編集しない。
   以下に、load-data.lisp の内容を示す。

```
(in-package :gplanner)
;;; Data
(defvar *data-path*
    (concatenate 'string user::*PlanRec-path* "Data/"))
(defvar *schemata-filename*
  (merge-pathnames "test.plans" *data-path*))
(defvar *np-concept-filename*
  (merge-pathnames "concept-nodes.lisp" *data-path*))
(defvar *proposition-grammar-filename*
```

```
  (merge-pathnames "case-grammar.lisp" *data-path*))
(defvar *kaiwa-input-filename*
  (merge-pathnames "kaiwa-1.data" *data-path*))
;;;;;;;;;;;;
(init-all)
```

3. データファイルをロードする。

```
> (load "load-data")
;;; Loading source file "load-data.lisp"
;;; Loading source file "Data/test.plans"
;;; Warning: File "Data/test.plans" does not begin with IN-PACKAGE.
Loading into package "GPLANNER"
;;; Loading source file "Data/simple.plans"
;;; Warning: File "Data/simple.plans" does not begin with IN-PACKAGE.
Loading into package "GPLANNER"
;;; Loading source file "Data/concept-nodes.lisp"
;;; Warning: File "Data/concept-nodes.lisp" does not begin with
IN-PACKAGE.  Loading into package "GPLANNER"
#P"/home/user/PlanRec/load-data.lisp"
>
```

## 3.4 プラン認識を行なう

1. 入力行為データをロードする。

```
> (load-input-actions)
20
>
```

2. layla を初期化する。

```
> (init-all)
;;; Loading source file "Data/test.plans"
;;; Warning: File "Data/test.plans" does not begin with IN-PACKAGE.
Loading into package "GPLANNER"
;;; Loading source file "Data/simple.plans"
;;; Warning: File "Data/simple.plans" does not begin with IN-PACKAGE.
 Loading into package "GPLANNER"
;;; Loading source file "Data/concept-nodes.lisp"
;;; Warning: File "Data/concept-nodes.lisp" does not begin with
IN-PACKAGE.  Loading into package "GPLANNER"
472
>
```

3. 最短優先モードをオンにする。

```
> (first-hit-on)
--- Control Modes of 0 ---
Inference Rules      = (DECOMPOSITION-CHAIN EFFECT-CHAIN PRECONDITION-CHAIN)
Schema Typee         = (INTERACTION-PLAN COMMUNICATION-PLAN
                        DOMAIN-PLAN DIALOGUE-PLAN)
Layer Mode           = T
Input Order          = NIL
Goal Direction       = BR
Chain Mode           = DIRECT
Indirect Depth       = NIL
First Hit            = T
Simple Mode          = NIL
Trace                = NIL
NIL
>
```

4. プラン認識を1つ進める。

```
> (plan-inference-next)
---(D1-1)-------- Plan-inference with input 1
(GREETING-OPEN SP1 SP2 OPENING (OPEN-DIALOGUE))
  CHAIN              3    3.182
--------------- Result Number is 3
3
>
```

5. 現在の理解状態の概観を表示する。

```
> (tprint-gsl)
+--DIALOGUE
    |--[D]: GREETING-OPEN-UNIT
    |   |--[E]: (KNOW SP2 (IS SP1 ?NAME-5))
    |   |--[D]: GREETING-OPEN (SP1) もしもし
    |   +--[D]: (INFORM-VALUE SP1 SP2 ?TPC-5 (IS SP1 ?NAME-5))
    |--[D]: (CONTENTS ?S3-45 ?S4-45 ?TOPIC-45)
    +--[D]: (GREETING-CLOSE-UNIT ?S5-45 ?S6-45 CLOSING)

+--DIALOGUE
    |--[D]: GREETING-OPEN-UNIT
    |   |--[D]: GREETING-OPEN (SP1) もしもし
    |   +--[D]: (CONFIRM-VALUE-UNIT SP1 SP2 ?NAME-6)
    |--[D]: (CONTENTS ?S3-45 ?S4-45 ?TOPIC-45)
```

```
    +--[D]: (GREETING-CLOSE-UNIT ?S5-45 ?S6-45 CLOSING)

+--DIALOGUE
    |--[D]: GREETING-OPEN-UNIT
    |   |--[D]: GREETING-OPEN (SP1) もしもし
    |   +--[D]: (GREETING-OPEN SP2 SP1 OPENING (OPEN-DIALOGUE))
    |--[D]: (CONTENTS ?S3-45 ?S4-45 ?TOPIC-45)
    +--[D]: (GREETING-CLOSE-UNIT ?S5-45 ?S6-45 CLOSING)
(NIL NIL NIL)
>
```

## 3.5 その他の主な操作

### 3.5.1 知識ベースのロード

1. プランスキーマ (*file*) をロードする。それまでのプランスキーマはクリアされる。

```
> (load-schemata)
;;; Loading source file "Data/test.plans"
;;; Warning: File "Data/test.plans" does not begin with IN-PACKAGE.
Loading into package "GPLANNER"
;;; Loading source file "Data/simple.plans"
;;; Warning: File "Data/simple.plans" does not begin with IN-PACKAGE.
Loading into package "GPLANNER"
#P"/home/user/PlanRec/Data/test.plans"
>
```

2. 概念ネットワーク辞書をロードして、概念ネットワークを作る。

```
> (np::load-concept-network)
;;; Loading source file "Data/concept-nodes.lisp"
;;; Warning: File "Data/concept-nodes.lisp" does not begin with
IN-PACKAGE.  Loading into package "GPLANNER"
472
>
```

### 3.5.2 知識ベースの表示

1. プランスキーマテーブルの内容全体を *stream*へ表示する.

```
> (print-schemata)

:DOMAIN-PLAN
  PRESENT-PAPER          :(G158)
  MAKE-REGISTRATION      :(G157)
```

```
  PAY-FEE                 :(G156)
  JOIN-EVENT              :(G155)
  SEND-SOMETHING          :(G154)
  WRITE-SOMETHING         :(G153)

:COMMUNICATION-PLAN
  EXPLAIN-STATEMENT       :(G152)
  EXECUTE-ACTION          :(G151 G150)
  INTRODUCE-ACTION        :(G149)

:INTERACTION-PLAN
  INFORM-WANT-UNIT        :(G148)
  GET-VALUE-UNIT          :(G147)
  CONFIRM-VALUE-UNIT      :(G146 G145 G144)
  AFFIRMATIVE-U           :(G143 G142)
  NEGATIVE-U              :(G141 G140)
  REQUEST-ACTION-UNIT     :(G139 G138)
  OFFER-ACTION-UNIT       :(G137 G136)
  ASK-ACTION-UNIT         :(G135)
  CONFIRM-ACTION-UNIT     :(G134 G133 G132)
  ASK-STATEMENT-UNIT      :(G131)
  CONFIRM-STATEMENT-UNIT  :(G130 G129 G128)
  GREETING-CLOSE-UNIT     :(G127 G126)
  GREETING-OPEN-UNIT      :(G125 G124 G123)

:DIALOGUE-PLAN
  CONTENTS                :(G122 G121 G120)
NIL
>
```

2. 概念ネットワークの全てのノードを端末出力へ木構造で表示する。(サンプルデータの場合、大量に表示される)

```
> (np::pprint-all-nodes)


+--(お願い -1)
    |--PRAG(お願い)
    +--IS-A(BEHAVIOR)
        +--IS-A(NOUN)

+--(もの -1)
    |--PRAG(もの)
    +--IS-A(OBJECT)
        |--IS-A(NOUN)
        +--HAS-A-PROP(値段 -1)
```

```
            |--IS-A(QUANTITY)
            |    +--IS-A(VALUE)
            |         +--IS-A(NOUN)
            +--PRAG(値段)

 - 中略 -

+--(ASK-ACTION)
    |--IS-A(ASK)
    |    |--IS-A(DEMAND)
    |    |    +--IS-A(CAT)
    |    +--PRAG(ASK-PRAG)
    +--PRAG(ASK-ACTION-PRAG)

+--(REQUEST)
    |--IS-A(DEMAND)
    |    +--IS-A(CAT)
    +--PRAG(REQUEST-PRAG)
NIL
>
```

### 3.5.3 プランニング処理関係のコマンド

1. プランニングの制御モードを表示する。

```
> (display-mode)
--- Control Modes of D1-1 ---
Inference Rules      = (DECOMPOSITION-CHAIN EFFECT-CHAIN
                        PRECONDITION-CHAIN)
Schema Typee         = (INTERACTION-PLAN COMMUNICATION-PLAN
                        DOMAIN-PLAN DIALOGUE-PLAN)
Layer Mode           = T
Input Order          = NIL
Goal Direction       = BR
Chain Mode           = DIRECT
Indirect Depth       = NIL
First Hit            = NIL
Simple Mode          = NIL
Trace                = NIL
NIL
>
```

2. *ID*で指定された入力を処理した時点の状態に戻す。

```
> (reset-gplanner 'd1-2)
D1-2
>
```

# 第 4 章

# 関数リファレンス

LAYLA の主な関数・マクロ・大域変数について、説明する。

LAYLA のパッケージは “gplanner” である。しかし、概念検索機構 (4.6節) はパッケージ “NP” である。注意されたい。

## 4.1 システムロード・初期化 (load.lisp)

### 4.1.1 初期化

---

**initialize-gplanner** [ *Function* ]

---

引数: なし

LAYLA を初期化する。
目標構造・入力履歴が初期化される。さらに、プランスキーマの知識ベースをクリアする。従って、この初期化を行なった後は、知識ベースを改めてロードする必要がある。

---

**reset-gplanner** *&optional ID* [ *Function* ]

---

引数:

1. *ID (&optional)*: 入力の ID

リターン値: *ID*

*ID*で指定された入力を処理した時点の状態に戻す。*ID*が与えられなければ、プランスキーマをクリアしない以外は、 **initialize-gplanner()** に同じ。

---

init-all [ *Function* ]

---

引数: なし initialize-gplanner() を行なった後、プランスキーマ・命題格要素辞書・概念ネットワーク辞書をロードする。(知識ベースは全てデフォールトのファイルをロードする)

## 4.2 データロード (load-data.lisp)

### 4.2.1 大域変数

---

*data-path* [ *Variable* ]

---

**タイプ:** pathname

**初期値:** (concatenate 'string user::*PlanRec-path* "Data/")) [1]

プランスキーマなど、データ・知識を記述したファイルのあるディレクトリを指定

---

*schemata-filename* [ *Variable* ]

---

**タイプ:** pathname

**初期値:** (merge-pathnames "test.plans" *data-path*)

プランスキーマを記述したファイル名
初期値 "test.plans" には、ATR サンプル会話 (現在のところ, A,B,1–3) に対するプランスキーマが記述されている。

---

*np-concept-filename* [ *Variable* ]

---

**タイプ:** pathname

**初期値:** (merge-pathnames "concept-nodes.lisp" *data-path*)

概念ネットワーク辞書のファイル名
初期値 "concept-nodes.lisp" には、ATR サンプル会話に出現する名詞概念・動詞概念をカバーするものが記述されている。

---

[1] Sparc station の時

---

*proposition-grammar-filename* [ *Variable* ]

---

**タイプ:** pathname

**初期値:** (merge-pathnames "case-grammar.lisp" *data-path*)

命題格要素辞書のファイル名
命題要素格辞書とは、LAYLA 内部で扱うことのできる命題述語とその格要素を指定したものである。初期値 "case-grammar.lisp" には、ATR サンプル会話に出現する命題内容をカバーするものが記述されている。

---

*kaiwa-input-filename* [ *Variable* ]

---

**タイプ:** pathname

**初期値:** (merge-pathnames "kaiwa-1.data" *data-path*)

入力行為データを記述したファイル名
初期値 "kaiwa-1.data" には、ATR サンプル会話 1 の入力形式データが記述されている。

## 4.3 プラン推論

### 4.3.1 単一化 (unify.lisp, unify2.lisp)

---

pcvar [ *Structure* ]

---

| スロット名 | 初期値 | 内容 |
|---|---|---|
| id | nil | 変数のラベル |
| type | nil | タイプ要素 |
| info | nil | 備考 |

LAYLA の変数を表す内部データ構造体

---

unify *pat1 pat2* [ *Function* ]

---

引数:

1. *pat1*: 任意のデータ構造

2. *pat2*: 任意のデータ構造

リターン値: substitution(a-list/nil)

*pat1* と *pat2* の単一化を行なう。リターン値 substitution は、単一化の結果束縛された変数のラベルとその値の対を要素とする alist (を要素とするリスト) である。 nil の時は、単一化が失敗。
(注): (nil) の時は、変数束縛が起こらない単一化成功である。

---

**types-equal** *var1 var2* [ *Function* ]

---

引数:

1. *var1*: タイプ付変数
2. *var2*: タイプ付変数

リターン値: 共通タイプ要素 (list)/nil

タイプ付変数 *var1, var2* のタイプを比較し、等価なものがあればその全てを要素としたリストを返す。それ以外であれば nil を返す。

---

**nm-unify** *pat1 pat2* [ *Function* ]

---

引数:

1. *pat1*: 概念 (symbol)
2. *pat2*: 概念 (symbol)

集合による同一性の判断。
もし、 *pat1* から導出される集合の要素に、 *pat2* に等価 (基本的に **eqaul**) なものがあれば、 nil 以外を、それ以外は nil を返す。

---

**make-nm-entry-list** *pat* [ *Function* ]

---

引数:

1. *pat*: 概念 (symbol)

リターン値: 関連要素集合 (list)

入力 *pat*に関連する要素の集合を、概念ネットワークから検索して返す。
導出の範囲は、現在のところ、 *pat*の子ども (1 世代) まで.

---

pcvar-binding *pcvar subst* [ *Function* ]

---

引数:

1. *pcvar*: 変数 (pcvar)
2. *subst*: substitution (alist)

リターン値: 具体値 /nil

*subst* の中に、変数 *pcvar* を束縛するような要素があれば、その値を返す。

---

bind-list *list subst* [ *Function* ]

---

引数:

1. *list*: 任意のリスト (list)
2. *subst*: substitution (alist)

リターン値: 束縛後の *list*

*list* の要素の中で、 *subst* により具体値に束縛される変数を、その値に置き換えたリストを作り返す。副作用はない。

---

with-gplanner-readtable *&rest body* [ *Macro* ]

---

引数:

1. *body (&rest)*: リスプ式

リターン値: *body* の結果

変数を読み込むための read-table を利用するマクロ
"?" を変数定義のマクロキャラクタとして読む

デバッグ関連

---

***unify-counter*** [ *Variable* ]

---

**タイプ:** integer

**初期値:** 0

unify($pat1, pat2$) の内部手続き unify-1()(これが実際の単一化を行なっている, 再帰処理である) が呼び出された回数をカウントする。

---

***unify-monitor*** [ *Variable* ]

---

**初期値:** nil

この値が nil 以外であれば、 unify($pat1, pat2$) の処理過程を端末出力に表示する。

---

***proposition-grammar*** [ *Variable* ]

---

**タイプ:** list

**初期値:** nil

命題格要素辞書の内容を格納している大域変数 (*cf.* *proposition-grammar-filename*)

---

**load-proposition-grammar** [ *Function* ]

---

引数: なし
リターン値: 登録された命題の数

命題格要素ファイル *proposition-grammar-filename* の内容をロードして、 *proposition-grammar* に登録する。

---

**get-pg** *pred* [ *Function* ]

---

引数:

1. *pred*: 命題述語 (symbol)

リターン値: 命題 (list)

述語 *pred*の命題記述を返す。そのようなものがなければ nil

### 4.3.2 推論・連鎖 (chain.lisp)

---

**plan-inference-next** [ *Function* ]

---

引数: なし
リターン値: 理解状態 (構造体ゴールスタックのリスト)

プラン認識を 1 つ進める。
現在の LAYLA では、入力行為をファイル (*kaiwa-input-filename*) からバッチで読み込み、それを対象に処理を行なっている。本関数は、読み込まれた順に、入力をプラン認識していく。

---

**plan-inference-by-id** *id* [ *Function* ]

---

引数:

1. *id*: 入力 ID(symbol)

リターン値: 理解状態 (構造体ゴールスタックのリスト)

*id*に対応する入力行為を読み込んだデータから探して、それを入力行為としてプラン認識を進める。そのような行為がなければ、無視する。

---

**plan-inference** *input-id input-list &key stream goal-stack control* [ *Function* ]

---

引数:

1. *input-id*: 入力 ID(symbol)
2. *input-list*: 入力行為の解釈リスト (list)
3. *stream (&key)*: トレース出力ストリーム (defualt: **display-window***)
4. *goal-stack (&key)*: 理解状態 (default: **goal-stack-list***)
5. *control (&key)*: 制御オブジェクト (default: **default-control***)

リターン値: 理解状態 (構造体ゴールスタックのリスト)

プラン認識の主関数
*input-id* を ID として、*input-list*を理解状態 *goal-stack*の基でプラン認識を行なう。その際、トレース出力の要請があれば、*stream*に表示する。*control*は、処理内部で参照される制御オブジェクトである。

---

**chain** *actions gsl control* [ *Function* ]

---

引数:

1. *actions*: 入力行為 (構造体アクションのリスト: list)
2. *gsl*: 理解状態 (構造体ゴールスタックのリスト: list)
3. *control*: 制御オブジェクト

リターン値: 連鎖後の理解状態 /nil

連鎖の主関数. *actions*と *gsl*の連鎖を行ない、それを結合した新たな理解状態を返す。

---

**plan-inference-rules** *input goal control* [ *Function* ]

---

引数:

1. *input*: 入力行為 (action)
2. *goal*: 目標行為 (action)
3. *control*: 制御オブジェクト

リターン値: 推論規則のリスト

3つの入力から、*input*と *goal*の連鎖を行なう際に利用すべき推論規則のリストを返す。

---

**direct-chain** *action goal control* [ *Function* ]

---

引数:

1. *action*: 入力行為 (action)

2. *goal*: 目標行為 (action)
3. *control*: 制御オブジェクト

リターン値: 連鎖 (list of actions)/nil

*action* と *goal* の直接連鎖を求める。連鎖に失敗すれば、 nil を返す。

---

**chained-p** *prp* [ *Function* ]

---

引数:

1. *prp*: 命題

リターン値: t/nil

*prp* がすでに他のプランへ連鎖していれば nil 以外を、していなければ nil を返す。多くの場合、 *prp* はプランスキーマのスロット中に記述された命題である。

### A* 関連

---

**a*node** [ *Structure* ]

---

| スロット名 | 初期値 | 内容 |
|---|---|---|
| id | nil | ID |
| link | nil | リンク先の ID のリスト |
| value | nil | ノードの評価値 |
| state | nil | ノードの状態 |
| forwards | nil | 展開したノードのリスト (逆リンク) |
| queue | nil | 検索可能な知識ベースのクラスのリスト |
| info | nil | 備考 |

A* 探索で利用されるデータ構造.
探索内部処理では、全てこのデータ構造を対象に行なう。

---

**initialize-a*** [ *Function* ]

---

引数: なし
リターン値: 初期化された A* ノードのプール

A* 探索の記憶領域を初期化する。
LAYLA の 1 度の探索処理中は、生成されたノードを特定のプール（ハッシュテーブル）に記憶している。これは、解答作成の多様性やトレース・デバッグを容易に行なうために利用される。

---

**a*chain-main** *goal open- close- control-* [ *Function* ]

---

引数:

1. *goal*: 目標 (任意のデータ構造)
2. *open-*: 開ノードリストの初期値 (list)
3. *close-*: 閉ノードリストの初期値 (list)
4. *control*: 制御オブジェクトの初期値

リターン値: a*return で指定された解構造

A* 探索の主関数
*goal*は任意のデータであり、これは目標判別関数 success 内の定義に依存する。また、この探索による解答は関数 a*return により定義される。基本的には、この関数呼び出しの時点では、*open-*は入力ノードのリストであり、*close-*は nil である。

---

**a*success** *input goal control* [ *Function* ]

---

引数:

1. *input*: 入力ノード (a*node)
2. *goal*: 目標 (任意)
3. control: 制御オブジェクト

リターン値: t/nil

A* 探索の目標一致判別を行なう。もし、*input*の状態が目標を満たすようなものであれば、nil 以外を、そうでなければ nil を返す。
現在の LAYLA の実装では、目標 *goal*には、構造体アクションまたは構造体ゴールスタックのリストを指定することができる。他のデータ構造を扱いたい時は、カスタマイズすれば良い。

---

**a*getinput** *stack control* [ *Function* ]

---

引数:

1. *stack*: ノードのリスト (list)
2. *control*: 制御オブジェクト

リターン値: ノード /nil

*stack*から、評価値最良のノードを返す。そのようなものがなければ nil.
多くの場合, *stack*は $OPEN$ リスト.

---

**a*operators** *node gsl control* [ *Function* ]

---

引数:

1. *node*: 被展開ノード (a*node)
2. *gsl*: 理解状態 (goal-stack)
3. *control*: 制御オブジェクト

リターン値: 展開したノードのリスト

*node*を知識ベースを利用して展開して、展開した全てノードを要素とする
リスト返す。 *gsl*は無駄な展開を避けるために参照される。

---

**a*p-getnode** *id* [ *Function* ]

---

引数:

1. *id*: ノード ID(symbol)

リターン値: **A*** ノード /nil

(現在の探索における)*id*に対応する **A*** ノードを返す。そのようなものが
なければ、 nil.

---

**a*p-count** [ *Function* ]

---

引数: なし
リターン値: **A*** ノードの数

現在の探索において生成されたノードの数を返す.

### 4.3.3 目標構造管理 (goal-stack.lisp)

**理解状態**

---

***current-input-id*** [ *Variable* ]

---

**タイプ:** symbol

**初期値:** nil

処理中の入力 ID.

---

***goal-stack-list*** [ *Variable* ]

---

**タイプ:** list

**初期値:** nil

理解状態。その時点で残っている構造体ゴールスタックを要素とするリスト。

---

***gs-history*** [ *Variable* ]

---

**タイプ:** alist

**初期値:** nil

理解状態に履歴を保存しておく A リスト. 入力 ID と理解状態の対からなる。

---

**initialize-gsl** [ *Function* ]

---

引数: なし
リターン値: 初期理解状態

理解状態の初期化

---

**gsl-get-gs** *id* [ *Function* ]

---

引数:

1. *id*: 目標構造の ID(symbol)

リターン値: 目標構造ゴールスタック /nil

*id*に対応する目標構造を現在の理解状態から探し、そのようなものがあればその構造を、なければ nil を返す。

---

**initialize-gs-history** *&optional goal-stack-list* [ *Function* ]

---

引数:

1. *goal-stack-list (&optional)*: 理解状態 (list)

リターン値: 初期理解状態履歴リスト

理解状態の履歴リスト *gs-history* を初期化する。

---

**gshis-get-gsl** *input-id* [ *Function* ]

---

引数:

1. *input-id*: 入力行為の ID(symbol)

リターン値: 理解状態

*input-id*に対応する理解状態を理解状態の履歴リストから探し、そのようなものがあればその理解状態を、なければ nil を返す。

---

**gshis-reset-gsl** *input-id* [ *Function* ]

---

引数:

1. *input-id*: 入力行為の ID(symbol)

リターン値: *input-id*(第 1 値), 理解状態の数 (第 2 値)

現在の理解状態を *input-id*の時点ものに戻す。

目標構造

---

gs [ *Structure* ]

---

| スロット名 | 初期値 | 内容 |
|---|---|---|
| id | nil | ID |
| incomplete | nil | 未充足プランのリスト |
| complete1 | nil | 充足プラン のリスト |
| complete2 | nil | 未充足かつ談話セグメント完了プランのリスト |
| statements | nil | 共通理解事項のリスト |
| unrelate | nil | 当目標構造に無関係な入力からの連鎖 |
| prediction | nil | 当目標構造からの予測行為 |
| selection | nil | prediction に基づいて次の入力から選択された行為 |

ある一つの目標構造であるゴールスタックのデータ構造。

---

**gse-complete-event-p** *event* [ *Function* ]

---

引数:

1. *event*: スロットの要素 (action)

リターン値: t/nil

---

**gs-append-chain** *chain gs* [ *Function* ]

---

引数:

1. *chain*: 行為の連鎖 (list)
2. *gs*: 目標構造 (goal-stack)

リターン値: 連鎖結合後の目標構造

*chain*を *gs*に結合して、結合後の目標構造を返す。入力目標構造 *gs*に副作用は起こらない。すなわち出力目標構造は、新しい ID を持つ目標構造である。

---

**gs-check-complete-event** *event gs* [ *Function* ]

---

引数:

1. *event*: スロットの要素 (action)
2. *gs*: 目標構造 (goal-stack)

リターン値: t/nil

*event*が *gs*の中で完了している状態であれば nil 以外を、そうでなければ nil を返す。入力 *event*は、構造体ゴールスタックのスロットの各スタックに入り得る要素であり、現在の **LAYLA** では多くの場合、構造体アクションである。

## 4.4 入出力・知識ベース

### 4.4.1 入力 (input.lisp)

---

*inputs* [ *Variable* ]

---

**タイプ**: list

**初期値**: nil

処理を行なった入力行為 (構造体アクション) の系列を格納したリスト

---

*input-sequence* [ *Variable* ]

---

**タイプ**: list

**初期値**: nil

load-input-actions 二より読み込まれた入力行為の系列を格納したリスト。
この要素は構造体アクションでなく、命題形式 (リスト) である。

---

clear-inputs [ *Function* ]

---

引数: なし
リターン値: nil

入力データのリスト *inputs* を初期化 (nil) する。

---

**get-input-action** *action-id* [ *Function* ]

---

引数:

1. *action-id*: 入力 ID(symbol)

リターン値: 構造体アクション /nil

*action-id*に対応するような入力行為を *inputs* から探し、そのようなものがあればそのデータを、なければ nil を返す。

---

**set-input-action** *id pat* [ *Function* ]

---

引数:

1. *id*: 入力 ID(symbol)
2. *pat*: 命題パタン (list)

リターン値: 構造体アクション

入力命題 *pat*の ID を *id*として、構造体アクションを生成する。同時に *inputs* に登録される。

---

**load-input-actions** *&optional filename* [ *Function* ]

---

引数:

1. *filename (&optional)*: ロードするファイル名 (pathname), 初期値は *kaiwa-input-filename*.

リターン値: 読み込んだ入力の個数

*filename*のファイルから入力命題をロードする。ロードされたデータは *input-sequence* に格納する。(*inputs* は初期化される).

### 4.4.2 プランスキーマ (schema.lisp)

---

***schemata*** [ *Variable* ]

---

**タイプ:** hash table

**初期値:** empty hash

プランスキーマを格納したテーブル

---

**action** [ *Structure* ]

---

| スロット名 | 初期値 | 内容 |
|---|---|---|
| id | nil | スキーマの ID |
| type | nil | 所属する知識ベースのクラス |
| header | nil | 見出し (命題) |
| precondition | nil | 前提条件 (命題のリスト) |
| delete-list | nil | 削除効果 (命題のリスト) |
| effect | nil | 追加効果 (命題のリスト) |
| decomposition | nil | 副行為列 (命題のリスト) |
| constraint | nil | 制約条件 |
| link-id | nil | 見出しの連鎖先 |
| goal-id | nil | インスタンスの ID |

プランスキーマの内部データ構造 (構造体アクション)

---

**variablep** *var* [ *Macro* ]

---

引数:

1. *var*: 任意のデータ

リターン値: t/nil

*var*が変数であれば nil 以外を、それ以外は nil を返す。

---

**typed-var-p** *var* [ *Macro* ]

---

引数:

1. *var*: 任意のデータ

リターン値: t/nil

*var*がタイプ付変数であれば nil 以外を、それ以外は nil を返す。

---

plan-type-p *type* [ *Function* ]

---

引数:

1. *type*: 任意のデータ

リターン値: t/nil

*type*が知識ベースのクラスを表すデータであれば nil 以外を、それ以外は nil を返す。

**編集**

---

defschema *type body-string* [ *Macro* ]

---

引数:

1. *type*: 知識ベースのクラス
2. *body-string*: 内容

リターン値: プランスキーマ

プランスキーマ定義用マクロ. (書式の詳細については、2.5.2節参照)

---

load-schemata *&optional (file *schemata-filename*)* [ *Function* ]

---

引数:

1. *file (&optional)*: ファイル名

*file (&optional)*をロードする。それまでのプランスキーマはクリアされる。

---

clear-schemata [ *Function* ]

---

引数: なし
リターン値: *schemata*

プランスキーマのテーブル *schemata* を初期化する. 格納されていたすべてのプランスキーマは、削除される.

---

remove-schema *key &key type* [ *Function* ]

---

引数:

1. *key*: キー
2. *type (&key)*: クラス

リターン値: 削除したスキーマが属していた知識ベース

*key*で指定されたプランスキーマをテーブルから削除する. *key*にはスキーマ ID あるいはスキーマ見出しの述語を指定することができる. キーワード変数 :type には、そのスキーマが属するのクラスを指定することができる.

---

save-schema *action &optional (stream t)* [ *Function* ]

---

引数:

1. *action*: 構造体アクション
2. *stream (&optional)*: 出力ストリーム

*action*を読み出し可能な書式で、 *stream*へ出力する。

---

save-tsp-schema *type file* [ *Function* ]

---

引数:

1. *type*: クラス
2. *file*: ファイル名

*type*で指定されたクラスに属するプランスキーマ全てを読み出し可能な書式で、 *file*へ書き出す。

---

save-schemata *&optional (filename *schemata-filename*)* [ *Function* ]

---

引数:

1. *filename (&optional)*: ファイル名

リターン値:

現在のプランスキーマテーブルの内容全てを読み出し可能な書式で、*filename*へ書き出す。

### 検索

---

**fetch-schema** *key &optional type* [ *Function* ]

---

引数:

1. *key*: キー
2. *type (&optional)*: クラス

リターン値: プランスキーマ(のリスト)

*key*で指定されたプランスキーマをテーブルから検索する. *key*には, スキーマ ID あるいはスキーマ見出しの述語を与えることが出来る. リターン値は、入力が ID であればそれに対応するプランスキーマ（構造アクション）、述語であればそれに対応する全てのスキーマのリストである.

---

**match-schema** *pat slot &optional type* [ *Function* ]

---

引数:

1. *pat*: 命題パタン
2. *slot*: スロット名
3. *type (&optional)*: クラス

リターン値: プランインスタンスのリスト

*pat*で指定されたパタンに、*slot*の要素が単一化可能なプランスキーマを検索する. リターン値は、単一化に成功したプランスキーマインスタンスのリストであり、そのインスタンスには変数の束縛がなされている. *type*検索したいスキーマのクラスを与えることが出来る.

---

**schemata-schema-list** [ *Macro* ]

---

引数: なし
リターン値: プランスキーマのリスト
現在システムが保持している全てのプランスキーマを返す。

---

**tsp-schema-list** *type* [ *Function* ]

---

引数:

1. *type*: クラス (symbol)

リターン値:

クラス *type*に属する全てのプランスキーマを返す。

---

**tsp-count** *type* [ *Function* ]

---

引数:

1. *type*: クラス

リターン値: スキーマの数

*type*で指定されたクラスに属するスキーマの数を返す。

---

**count-schemata** [ *Function* ]

---

引数: なし
リターン値: スキーマの数
現在システムが持っているスキーマの数を返す。

---

**instantiate-action** *action subst* [ *Function* ]

---

引数:

1. *action*: 構造体アクション
2. *subst*: substitution

リターン値: アクションのインスタンス

*action*のインスタンスを生成する。変数の束縛には、*subst*が用いられる。

### 4.4.3 コントロール (control.lisp)

---

***default-control*** [ *Variable* ]

---

初期値: make-control

何も指定されない時用いられる制御オブジェクト

---

***default-schema-order-list*** [ *Variable* ]

---

初期値: (list :interaction-plan :communication-plan :domain-plan :dialogue-plan))

何も指定されない時用いられる知識ベースの検索順序

---

**control** [ *Structure* ]

---

制御オブジェクトのデータ構造 (スロットの詳細は 1.9参照)

**制御モード操作**

(注意:) 現在の LAYLA の実装では、以下のモード操作全て ***default-control** に対して行なっている。

---

**input-order-on** [ *Function* ]

---

引数: なし
リターン値: T

入力順序モードを sequential にして、入力間の優先順位を設定した処理を行なう。

---

**input-order-off** [ *Function* ]

---

引数: なし
リターン値: nil

入力順序モードを parallel にする。

---

first-hit-on [ *Function* ]

---

引数: なし
リターン値: T

最短優先モードで処理を行なう。

---

first-hit-off *&optional (control *default-control*)* [ *Function* ]

---

引数: なし
リターン値: nil

最短優先で行なわない (すなわち求められる範囲のものは全て求める)

---

layer-mode-on [ *Function* ]

---

引数: なし
リターン値: T

階層モードを layered にする。

---

layer-mode-off [ *Function* ]

---

引数: なし
リターン値: nil

階層モードを single にする。

---

trace-on [ *Function* ]

---

引数: なし
リターン値: T

トレース表示を行なうようにする。

---

**trace-off** [ *Function* ]

---

引数: なし
リターン値: nil

トレース表示を行なわない。

---

**set-indirect-depth** *num* [ *Function* ]

---

引数:

1. *num*: 回数 (integer)

リターン値: *num*

間接連鎖回数を *num*に設定する。

### 検索

---

**control-get-tsp-order** *input control* [ *Function* ]

---

引数:

1. *input*: 入力行為 (action)
2. *control*: 制御オブジェクト

リターン値: 知識ベースのクラスのリスト

*input*からの展開に利用すべき知識ベースのクラスのリストを返す。
(検索対象モードに対応)

---

**control-get-inference-rules** *input goal control* [ *Function* ]

---

引数:

1. *input*: 入力行為 (action)
2. *goal*: 任意のデータ
3. *control*: 制御オブジェクト

リターン値: 推論規則のリスト

*input*と *goal*の連鎖に利用すべき推論規則のリストを返す。
(適用規則モードに対応)

---

**control-gsl-order** *gsl input-action control* [ *Function* ]

---

引数:

1. *gsl*: 理解状態
2. *input-action*: 入力行為
3. *control*: 制御オブジェクト

リターン値: ソート後の目標構造のリスト

*input-action*のプラン認識を行なう際の、*gsl*の中の目標構造の順序を決め、
その順序にしたがった目標構造のリストをつくって返す。
(対象ゴールモードに対応)

---

**control-ordered-input-list** *input-list gsl control* [ *Function* ]

---

引数:

1. *input-list*: 入力行為のリスト
2. *gsl*: 理解状態
3. *control*: 制御オブジェクト

リターン値: ソート後の入力行為のリスト

*input-list*を *gsl*に対してプラン認識を行なう時の、入力優先順序を決め、
その順序にしたがった入力行為のリストを作って返す。
(入力順序モードに対応)

## 4.5 トレース・表示

---

***display*** [ *Variable* ]

---

初期値: t
表示のスイッチ. これが nil 以外であれば、プラン認識の処理過程を *monitor-stream* に表示する。

---

*monitor-stream* [ *Variable* ]

---

**タイプ:** output stream

**初期値:** t

プラン認識の処理過程を表示するためのストリーム。

---

*time* [ *Variable* ]

---

初期値: t

処理時間表示のためのスイッチ. この値が nil 以外であれば、処理時間を *monitor-stream* に表示する。

---

with-runtime *input-id form* [ *Macro* ]

---

引数:

1. *input-id*: 表示用ラベル
2. *form*: 計時対象関数

リターン値: *form* の結果

*form*で与えられたリスプ form を評価し、form の結果を返す. この時, *time* の値が nil 以外であれば, その処理時間を *monitor-stream* に表示する. 表示は, *input-id*, *form* の結果 (リストの時はその長さ), 実行時間.

---

print-internal-chain [ *Function* ]

---

引数: なし
リターン値:

現在の探索において生成された A* ノードの連鎖関係 (多くは木構造になる) を *monitor-stream* に表示する。

---

pprint-gs *gs &optional (stream t)* [ *Function* ]

---

引数:

1. *gs*: 目標構造 (goal-stack)
2. *stream (&optional)*: 表示ストリーム (output stream)

リターン値: nil

*gs*の詳細情報を *stream*へ表示する。

---

simprint-gs *gs &optional (stream t)* [ *Function* ]

---

引数:

1. *gs*: 目標構造 (goal-stack)
2. *stream (&optional)*: 表示ストリーム (output stream)

リターン値: nil

*gs*の概観を *stream*へ表示する。

---

pprint-gsl *&optional (stream t)* [ *Function* ]

---

引数:

1. *stream (&optional)*: 表示ストリーム (output stream)

リターン値: 表示した目標構造の数

現在の理解状態の概観を *stream*へ表示する。

---

simprint-gsl *&optional (stream t)* [ *Function* ]

---

引数:

1. *stream (&optional)*: 表示ストリーム (output stream)

リターン値: 表示した目標構造の数

現在の理解状態の詳細情報を *stream*へ表示する。

---

**pprint-action** *action &optional (stream t)* [ *Function* ]

---

引数:

1. *action*: 構造体アクション
2. *stream (&optional)*: 出力ストリーム

*action*の内容を *stream*へ表示する.

---

**print-schema** *id &optional (stream t)* [ *Function* ]

---

引数:

1. *id*: スキーマ ID
2. *stream (&optional)*: 出力ストリーム

*id*で指定されたプランスキーマの内容を *stream*へ表示する. *id*には、スキーマ ID あるいはスキーマ見出しの述語を指定することができる.

---

**print-schemata** *&optional (stream t)* [ *Function* ]

---

引数:

1. *stream (&optional)*: 出力ストリーム

プランスキーマテーブルの内容全体を *stream*へ表示する.

---

**print-tsp** *type &optional (stream t)* [ *Macro* ]

---

引数:

1. *type*: クラス
2. *stream (&optional)*: 出力ストリーム

*type*で指定されたクラスに属するプランスキーマの内容を *stream*へ表示する.

## 4.6 概念検索

以下の関数・変数のパッケージは、"NP" である。

### 4.6.1 ロード・初期化

概念ネットワーク辞書のセットアップ・検索機構のロードは、**LAYLA** 立ち上げと同時に自動的に行なわれる。

---

***name-concept-file*** [ *Variable* ]

---

**タイプ:** string

**初期値:** "concept-nodes.lisp"

概念ネットワーク辞書のファイル名
初期値 "concept-nodes.lisp" には、ATR サンプル会話に出現する名詞概念・動詞概念をカバーするものが記述されている。

### 4.6.2 ネットワーク・検索 (network.lisp)

---

***concept-network*** [ *Variable* ]

---

**タイプ:** 構造体ネットワーク

**初期値:** nil network

概念ネットワークの内部データ構造を格納

---

**network** [ *Structure* ]

---

| スロット名 | 初期値 | 内容 |
|---|---|---|
| nodes | nil | ノードのリスト |
| links | nil | リンクのリスト |

ネットワークの内部データ構造
nodes スロットには、概念のノード (構造体ノード) が入る。links スロットには、ノード感の関係を持つ構造体リンクが入る。

---

**node** [ *Structure* ]

---

| スロット名 | データタイプ | 内容 |
|---|---|---|
| id | symbol | ノードのID |
| value | symbol | ノードのラベル |
| type | symbol | ノードのタイプ (:concept/:word) |
| links | 構造体ノード | 関連するノードのリスト |
| network | 構造体ネットワーク | 所属するネットワーク |

ノードの内部データ構造

---

**link** [ *Structure* ]

---

| スロット名 | データタイプ | 内容 |
|---|---|---|
| type | symbol | リンクのタイプ |
| parent-node | 構造体ノード | 親ノード |
| relate-node | 構造体ノード | 子ノード |

リンクの内部データ構造

---

**defconcept** *value &optional info &rest links* [ *Macro* ]

---

引数:

1. *value*: ラベル (symbol)
2. *info (&optional)*: ドキュメント (string)
3. *links (&rest)*: リンク情報 (list)

リターン値: 構造体ノード

*value*をラベル, *links*をリンク情報として概念ノードを生成する。
*links*はリンクタイプとその値 (リンク先のノードのラベル) の1つ以上の並びである。また、*info*に文字列が指定されれば、それをノードのドキュメンテーションとして記憶する。

---

**init-concept** [ *Function* ]

---

引数: なし

概念ネットワークの初期化

---

load-concept-network *&key file package* [ *Function* ]

---

引数:

1. *file (&key)*: ファイル名 (default: (merge-pathnames *name-concept-file* *data-path*))
2. *package (&key)*: 読み込みパッケージ (default: (package-name *package*))

リターン値: 登録ノード数

概念ネットワーク辞書ファイルをロードして、概念ネットワークを作る

---

remove-concept *value* [ *Macro* ]

---

引数:

1. *value*: ノードラベル

リターン値: 削除後の概念ネットワーク

*value*で指定されたノードを, 概念ネットワークから削除する。そのようなノードがなければ何もせず nil を返す。

---

replace-a-link-of-node-from-value *&key value link-data network* [ *Function* ]

---

引数:

1. *value (&key)*: ノードのラベル (symbol)
2. *link-data (&key)*: リンクの記述 (list)
3. *network (&key)*: ネットワーク (default: *concept-network*)

リターン値: 修正後のノード

*network*の中の *value*で指定されたノードの、*link-data*の第1引数に対応するリンクタイプのリンクデータを *link-data*の記述で置き換える. 上記リンクタイプ以外のリンクデータはそのまま.

---

**count-concept-nodes** [ *Macro* ]

---

引数: なし
リターン値: 概念ノード数

---

**get-related-list** *value type-list &key max-level search-type direction network* [ *Function* ]

---

引数:

1. *value*: ノードのラベル (symbol)
2. *type-list*: リンクタイプのリスト (list)
3. *max-level (&key)*: 最大検索レベル (integer)
4. *search-type (&key)*: 検索方法 (:simple/:eq-level-all)
5. *direction (&key)*: 検索方向 (:both/:relate/:parent)
6. *network (&key)*: ネットワーク (:concept/:word)

リターン値: 関連ノードのラベルのリスト

*value*で指定されたノードから *type-list*で辿ることのできるノードを検索し、それらノードのラベルのリストを返す。*type-list*に nil が与えられれば、全てのタイプのリンクを検索する。
*:max-level*にはリンクを何回まで辿るかを指定する。default は nil (= 辿れる範囲は全て辿る) である。
*:search-type*には、2つある。:simple は *direction*で指定された一方向のみを検索する。一方、:eq-level-all では、自分の兄弟ノードから検索可能なノードも同時に検索する。default は :simple.
*:direction*は検索方向。:parent は親方向、:relate は子方向, :both は両方向. default は :both.
*:network*には、検索対象のネットワークを指定する。

### 4.6.3 表示 (display-network.lisp)

---

**pprint-node** *value &key type-list direction network* [ *Function* ]

---

引数:

1. *value*: ノードのラベル (symbol)
2. *type-list (&key)*: リンクタイプのリスト (list)
3. *direction (&key)*: 検索方向 (:relate/:parent)
4. *network (&key)*: ネットワーク (:concept/:word)

*value*で指定されたノードから関連ノードを端末出力へ木構造で表示する。

*:type-list*は表示する関連ノードを検索する際のリンクタイプを制限する。*type-list*に nil が与えられれば、全てのタイプのリンクを検索, 表示する。default は nil.
*:direction*は検索方向。:parent は親方向、:relate は子方向. default は :relate.
*:network*には、検索・表示対象のネットワークを指定する。

---

pprint-all-nodes *&key start type-list network* [ *Function* ]

---

引数:

1. *start (&key)*: 表示始点 (:bottom/:top)
2. *type-list (&key)*: リンクタイプのリスト (list)
3. *network (&key)*: ネットワーク (:concept/:word)

*network*で指定されたネットワークの全てのノードを端末出力へ木構造で表示する。
*:start*は表示の始点であり、:bottom は一番孫のノードから、逆に:top は一番祖先のノードから表示する。default は :bottom.
*:type-list*は表示するノードを検索する際のリンクタイプを制限する。*type-list*に nil が与えられれば、全てのタイプのリンクを検索, 表示する。default は nil.
*:network*には、検索・表示対象のネットワークを指定する。

# 付録 A

# データ記述例

## A.1 入力行為データ (PlanRec/Data/kaiwa-1.data)

以下に、ATR サンプル会話 1 の入力発話の、LAYLA 入力用記述を掲載する。

```
(D1-1
  (
    (GREETING-OPEN      SP1 SP2 OPENING
                        (OPEN-DIALOGUE)
                        "もしもし")
  )
)

(D1-2
  (
    (CONFIRM-VALUE      SP1 SP2 そちら-1
                        (だ -IDENTICAL そちら-1 会議事務局-1)
                        "そちらは会議事務局ですか")
  )
)

(D1-3
  (
    (GREETING-OPEN      SP2 SP1 OPENING
                        (OPEN-DIALOGUE)
                        "はい")
  )
  (
    (AFFIRMATIVE        SP2 SP1 ?TPCD1-3
                        ?PRPD1-3
                        "はい")
  )
)
```

```
(D1-4
  (
    (AFFIRMATIVE        SP2 SP1 ?TPCD1-4
                        ?PRPD1-4
                        "そうです")
  )
)

(D1-5
  (
    (ASK-STATEMENT      SP2 SP1 ?TPCD1-5
                        (だ -IDENTICAL ?OBJED1-5 用件 -1)
                        "どのようなご用件でしょうか")
  )
)

(D1-6
  (
    (INFORM-WANT        SP1 SP2 ?TPCD1-6
                        (申し込む -1 SP1 ?RECPD1-6 ?OBJED1-6)
                        "会議に申し込みたいのですが")
  )
)

(D1-7
  (
    (ASK-ACTION         SP1 SP2 ?TPCD1-7
                        (する -1 ?AGEND1-7 手続き -1)
                        "どのような手続きをすればよろしいのでしょうか")
  )
)

(D1-8
  (
    (REQUEST-ACTION     SP2 SP1 ?TPCD1-8
                        (する -1 SP1 手続き -1)
                        "登録用紙で手続きをして下さい")
  )
  (
    (INFORM-ACTION      SP2 SP1 ?TPCD1-8
                        (する -1 SP1 手続き -1)
                        "登録用紙で手続きをして下さい")
  )
```

```
)

(D1-9
  (
    (CONFIRM-STATEMENT  SP2 SP1 登録用紙-1
                        (持つ-1 ?AGEND1-9 登録用紙-1)
                        "登録用紙は既にお持ちでしょうか")
  )
)

(D1-10
  (
    (NEGATIVE           SP1 SP2 ?TPCD1-10
                        ?PRPD1-10
                        "いいえ")
  )
)

(D1-11
  (
    (NEGATIVE           SP1 SP2 ?TPCD1-11
                        ?PRPD1-11
                        "まだです")
  )
)

(D1-12
  (
    (CONFIRMATION       SP2 SP1 ?TPCD1-12
                        ?PRPD1-12
                        "分かりました")
  )
  (
    (ACCEPT-ACTION      SP2 SP1 ?TPCD1-12
                        ?PRPD1-12
                        "分かりました")
  )
)

(D1-13
  (
    (OFFER-ACTION       SP2 SP1 ?TPCD1-13
                        (送る-1 SP2 SP1 登録用紙-1)
                        "それでは登録用紙をお送り致します")
```

```
  )
)

(D1-14
  (
    (ASK-VALUE          SP2 SP1 名前-1
                       ?PRPD1-14
                       "ご住所とお名前をお願いします")
    (ASK-VALUE          SP2 SP1 住所-1
                       ?PRPD1-14
                       "ご住所とお名前をお願いします")
  )
)

(D1-15
  (
    (INFORM-VALUE       SP1 SP2 住所-1
                       (だ-IDENTICAL 住所-1 ADDRESS)
                       "住所は大阪市北区茶屋町二十三です")
  )
)

(D1-16
  (
    (INFORM-VALUE       SP1 SP2 名前-1
                       (だ-IDENTICAL 名前-1 NAME)
                       "名前は鈴木真弓です")
  )
)

(D1-17
  (
    (CONFIRMATION       SP2 SP1 ?TPCD1-17
                       ?PRPD1-17
                       "分かりました")
  )
  (
    (ACCEPT-ACTION      SP2 SP1 ?TPCD1-17
                       ?PRPD1-17
                       "分かりました")
  )
)

(D1-18
```

```
  (
    (OFFER-ACTION        SP2 SP1 ?TPCD1-18
                        (送る -1 SP2 ?RECPD1-18 登録用紙 -1)
                        "登録用紙を至急送らせて頂きます")
  )
)

(D1-19
  (
    (REQUEST-ACTION     SP1 SP2 ?TPCD1-19
                        ?PRPD1-19
                        "よろしくお願いします")
  )
  (
    (ACCEPT-OFFER       SP1 SP2 ?TPCD1-19
                        ?PRPD1-19
                        "よろしくお願いします")
  )
)

(D1-20
  (
    (GREETING-CLOSE     SP1 SP2 CLOSING
                        (CLOSE-DIALOGUE)
                        "それでは失礼します")
  )
)
```

## A.2 概念ネットワーク辞書 (PlanRec/Data/concept-nodes.lisp)

以下に、ATR サンプル対話に対する概念記述の一部を掲載する。

```
;;;
;;;      Concept nodes
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Communicative act types
;;;
(np::defconcept CAT
            )
(np::defconcept DEMAND
            (is-a CAT))
(np::defconcept RESPONSE
            (is-a CAT))
(np::defconcept ACKNOWLEDGE
            (is-a CAT))

(np::defconcept ASK
            (is-a demand)
    (prag ask-prag))
(np::defconcept REQUEST
            (is-a demand)
    (prag request-prag))
(np::defconcept CONFIRM
            (is-a demand)
    (prag confirm-prag))
(np::defconcept INFORM
            (is-a response)
    (prag inform-prag))

;;; ASK
(np::defconcept ASK-ACTION
            (is-a ask)
    (prag ask-action-prag))
(np::defconcept ASK-VALUE
            (is-a ask)
    (prag ask-value-prag))
(np::defconcept ASK-STATEMENT
            (is-a ask)
    (prag ask-statement-prag))
;;; CONFIRM
(np::defconcept CONFIRM-ACTION
```

```
            (is-a confirm)
    (prag CONFIRM-ACTION-prag))
(np::defconcept CONFIRM-VALUE
            (is-a confirm)
    (prag CONFIRM-VALUE-prag))
(np::defconcept CONFIRM-STATEMENT
            (is-a confirm)
    (prag CONFIRM-STATEMENT-prag))
;;; ACTION
(np::defconcept REQUEST-ACTION
            (is-a demand)
    (prag REQUEST-ACTION-prag))
(np::defconcept OFFER-ACTION
            (is-a demand)
    (prag OFFER-ACTION-prag))

;;; RESPONSE
;;;
;;; INFORM
(np::defconcept INFORM-ACTION
            (is-a inform)
    (prag INFORM-ACTION-prag))
(np::defconcept INFORM-VALUE
            (is-a inform)
    (prag INFORM-VALUE-prag))
(np::defconcept INFORM-STATEMENT
            (is-a inform)
    (prag INFORM-STATEMENT-prag))
(np::defconcept INFORM-WANT
            (is-a inform)
    (prag INFORM-STATEMENT-prag))

;;; SIMPLE RESPONSE
(np::defconcept AFFIRMATIVE
            (is-a inform)
    (prag AFFIRMATIVE-prag))
(np::defconcept NEGATIVE
            (is-a inform)
    (prag NEGATIVE-prag))

(np::defconcept AFFIRMATIVE-U
            (is-a affirmative)
    (prag AFFIRMATIVE-prag))
(np::defconcept NEGATIVE-U
```

```
            (is-a negative)
    (prag NEGATIVE-prag))

(np::defconcept ACCEPT-ACTION
            (is-a response)
    (prag ACCEPT-ACTION-prag))
(np::defconcept REJECT-ACTION
            (is-a response)
    (prag REJECT-ACTION-prag))

(np::defconcept ACCEPT-OFFER
            (is-a response)
    (prag ACCEPT-OFFER-prag))
(np::defconcept REJECT-OFFER
            (is-a response)
    (prag REJECT-OFFER-prag))

;;; CONFIRMATION
;;;
(np::defconcept CONFIRMATION
            (is-a acknowledge)
    (prag CONFIRMATION-prag))

;;; GREETING
;;;
(np::defconcept GREETING-OPEN
            (is-a demand)
            (is-a response)
    (prag GREETING-OPEN-prag))
(np::defconcept GREETING-CLOSE
            (is-a demand)
            (is-a response)
    (prag GREETING-CLOSE-prag))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;
;;;; VERBs in kaiwa 1-10
;;;;

(np::defconcept movement
    (is-a action))
(np::defconcept transference
    (is-a action))
(np::defconcept creation
```

```
    (is-a action))
(np::defconcept interaction
    (is-a action))
(np::defconcept perceptivity
    (is-a action))
(np::defconcept activity
    (is-a action))

(np::defconcept relation
    (is-a statement))
(np::defconcept possession
    (is-a statement))
(np::defconcept occurance
    (is-a statement))
(np::defconcept existance
    (is-a statement))
(np::defconcept change
    (is-a statement))


            ----- 中略 -----


;; for action
(np::defconcept どのように-1
    (prag どのように)
            (is-a wh-behavior))
(np::defconcept どう-1
    (prag どう)
            (is-a wh-behavior))
(np::defconcept どうやって-1
    (prag どうやって)
            (is-a wh-behavior))

;; neutral
(np::defconcept どのような-1
    (prag どのような)
            (is-a wh))
(np::defconcept どういう-1
    (prag どういう)
            (is-a wh))

;;
;; PRONOUN
```

```
;;
(np::defconcept PRONOUN
    (is-a value))
(np::defconcept これ-1
    (prag これ)
            (is-a pronoun))
(np::defconcept そちら-1
    (prag そちら)
            (is-a pronoun))
(np::defconcept それ-1
    (prag それ)
            (is-a pronoun))
(np::defconcept 私ども-1
    (prag 私ども)
            (is-a pronoun))
(np::defconcept 者-1
    (prag 者)
            (is-a pronoun))
(np::defconcept 他-1
    (prag 他)
            (is-a pronoun))
;;
;; ABSTract noun (formal noun)
;;
(np::defconcept こと-1
    (prag こと)
            (is-a value)
            (is-a event))
(np::defconcept もの-1
    (prag もの)
            (is-a object))
(np::defconcept お願い-1
    (prag お願い)
            (is-a behavior))

;;; end nouns

;;; END ;;;
```

## A.3 命題格要素辞書 (PlanRec/Data/case-grammar.lisp)

以下に、ATR サンプル対話に対する命題記述を掲載する。

```
;;;
;;; Proposition case frames
;;;  by comparing the results of CATE with the work dictionary of ATR.
;;;
;;;   produced by Yamaoka
;;;   originated on 02/06/92
;;;
;;  Format:
;;   PRP ::= (list Predicate Case1 Case2 ... )
;;
(is              OBJE IDEN)
(だ-IDENTICAL    OBJE IDEN) ;
(だ-STATEMENT    OBJE IDEN) ; TDEP

(go         AGEN DEST)
(行く-1     AGEN DEST) ; INST
(行く-3     AGEN DEST) ; INST

(participate    AGEN LOCT)
(参加する-1     AGEN LOCT) ; OBJE PURP INST ACCM

(register    AGEN OBJE)
(申し込む-1 AGEN RECP OBJE)              ; LOCT DEST
(申し込む-2 AGEN LOCT OBJE) ; ??

(send        AGEN RECP OBJE)
(送る-1      AGEN RECP OBJE) ; INST MANN TLOC
(同封する-1 AGEN OBJE) ;

(write       AGEN OBJE LOCT)
(書く-1      AGEN OBJE LOCT) ; INST
(記入する-1 AGEN OBJE LOCT) ; CAUS

(pay         AGEN RECP OBJE)
(支払う-1    AGEN RECP OBJE) ; MANN
(振り込む-1 AGEN DEST OBJE) ; TLOC

(have       AGEN OBJE)
(持つ-1     AGEN OBJE) ;
(ある-1     OBJE) ; TLOC LOCT DEGR
```

```
(do         AGEN OBJE)
(する -1    AGEN OBJE)                    ; INST
(行なう -1  AGEN OBJE) ; TLOC
(よい -1          OBJE) ; COND
(いい -1          OBJE) ; COND

(present       AGEN OBJE LOCT) ;
(発表する -1    AGEN OBJE LOCT) ;

(hold       AGEN OBJE)
(含む -1          OBJE LOCT) ; LOCT <- ALOC ?
(用意する -1 AGEN RECP OBJE) ;
;(ある -1    OBJE) ; TLOC LOCT DEGR

(need          OBJE) ;
(要る -1       OBJE) ;
(必要だ -1     OBJE) ; ROLE

(する -2    AGEN OBJE RESL) ; INST
(する -4    AGEN RECP OBJE) ; INST
(行く -2    OBJE DEST) ; INST
;
; V-kernel
;
; optional case names and comments
(かかる -1  OBJE DEGR)                    ; DEGR in kakaru-2 is not obligated in the dict
; PURP
(なる -2    OBJE RESL) ; LOCT ; 成る in dict
(分かる -1  EXPR OBJE) ; MANN
;(教える -1  AGEN RECP OBJE)               ; RANG maybe substitutes the OBJE's value
(教える -1  AGEN RECP OBJE RANG)          ; test
(教える -2  AGEN OBJE RANG) ;
(教える -3  AGEN RANG) ;
(近い -1    OBJE) ; LOCT GOAL
(見る -1    AGEN OBJE) ;
(取る -1    AGEN LOCT OBJE) ; DEGR TDES TDEP
;(尋ねる -1  AGEN RECP OBJE)               ; RANG == OBJE
(尋ねる -1  AGEN RECP RANG) ;
(待つ -1    AGEN OBJE) ;
(泊まる -1  AGEN LOCT) ; TDEP (LOCT <- SLOC)
(聞く -1    AGEN OBJE) ; ORIG
(聞く -3    AGEN RECP OBJE) ; DEGR

;
```

```
; N-sahen
;
(開催する -1     AGEN OBJE) ; LOCT TDES TDEP
(紹介する -1     AGEN OBJE RECP) ; LOCT
(利用する -1     AGEN PURP OBJE) ;

;
; Not in the dictinary
;
(いる -1         OBJE LOCT)  ; TLOC
(載る -1         OBJE LOCT) ;
(知らせる -1     AGEN RECP OBJE) ;
(調べる -1       AGEN OBJE) ;
(答える -1       AGEN OBJE) ;
(払い戻す -1     AGEN OBJE) ;
(取り消す -1     AGEN OBJE) ;

(見学する -1     AGEN OBJE TLOC) ;
;(質問する -1     AGEN RECP OBJE) ; RANG == OBJE
(質問する -1     AGEN RECP RANG) ; RANG == OBJE
(問題ある -1     LOCT) ; MANN
(予約する -1     AGEN OBJE) ; INST

;
; Others
;
;(ない -ADJECTIVE OBJE) ; in the dict nai needs SLOC
; may be not V
;
; The followings are secondary candidates
;

;
; special
;
(open-dialogue)
(close-dialogue)

;;; end of case-grammar ;;;;
```

## A.4 プランスキーマ (PlanRec/Data/test.plans)

以下に、ATR サンプル対話に対する対話構造解析用プランスキーマ記述の一部を掲載する。

```
;
; PlanRec/Data/test.plans and simple.plans
;
;;;
;;; Plan schemata file
;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Dialogue plans
;;;
(gplanner::defschema :DIALOGUE-PLAN
  "(
  :HEAD (CONTENTS ?SP1 ?SP2 PAY-FEE)
  :PREC ((know SP2 (is ?fee 用件-1)))
  :DELE ()
  :EFFE ()
  :DECO ((PAY-FEE SP1 SP2 ?fee))
  :CONS ()
  )"
)
(gplanner::defschema :DIALOGUE-PLAN
  "(
  :HEAD (CONTENTS ?SP1 ?SP2 JOIN-EVENT)
  :PREC ((know SP2 (is ?event 用件-1)))
  :DELE ()
  :EFFE ()
  :DECO ((JOIN-EVENT SP1 SP2 ?event))
  :CONS ()
  )"
)
(gplanner::defschema :DIALOGUE-PLAN
  "(
  :HEAD (CONTENTS ?SP1 ?SP2 PRESENT-PAPER)
  :PREC ((know SP2 (is ?event 用件-1)))
  :DELE ()
  :EFFE ()
  :DECO ((PRESENT-PAPER SP1 SP2 ?(paper 論文-1) ?event))
  :CONS ()
  )"
```

```
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Interaction plans
;;;
(gplanner::defschema :INTERACTION-PLAN
  "(
  :HEAD (GREETING-OPEN-UNIT ?SP ?HR ?GREET)
  :PREC ()
  :DELE ()
  :EFFE ((KNOW ?HR (IS ?SP ?NAME)))
  :DECO ((GREETING-OPEN ?SP ?HR ?GREET ?PRP)
         (INFORM-VALUE ?SP ?HR ?TPC (IS ?SP ?NAME)))
  :CONS ()
  )"
)

(gplanner::defschema :INTERACTION-PLAN
  "(
  :HEAD (GREETING-OPEN-UNIT ?SP ?HR ?GREET)
  :PREC ()
  :DELE ()
  :EFFE ()
  :DECO ((GREETING-OPEN ?SP ?HR ?GREET ?PRP)
         (CONFIRM-VALUE-UNIT ?SP ?HR ?NAME))
  :CONS ()
  )"
)

(gplanner::defschema :INTERACTION-PLAN
  "(
  :HEAD (GREETING-OPEN-UNIT ?SP ?HR ?GREET)
  :PREC ()
  :DELE ()
  :EFFE ()
  :DECO ((GREETING-OPEN ?SP ?HR ?GREET ?PRP)
         (GREETING-OPEN ?HR ?SP ?GREET ?PRP))
  :CONS ()
  )"
)

(gplanner::defschema :INTERACTION-PLAN
  "(
```

```
  :HEAD (GREETING-CLOSE-UNIT ?SP ?HR ?GREET)
  :PREC ()
  :DELE ()
  :EFFE ()
  :DECO ((GREETING-CLOSE ?SP ?HR ?GREET ?PRP)
         (GREETING-CLOSE ?SP ?HR ?GREET ?PRP))
  :CONS ()
  )"
)

(gplanner::defschema :INTERACTION-PLAN
  "(
  :HEAD (GREETING-CLOSE-UNIT ?SP ?HR ?GREET)
  :PREC ()
  :DELE ()
  :EFFE ()
  :DECO ((GREETING-CLOSE ?SP ?HR ?GREET ?PRP)
         (GREETING-CLOSE ?HR ?SP ?GREET ?PRP))
  :CONS ()
  )"
)
(gplanner::defschema :INTERACTION-PLAN
  "(
  :HEAD (CONFIRM-STATEMENT-UNIT ?SP ?HR ?STATE ?PRP)
  :PREC ()
  :DELE ()
  :EFFE ()
  :DECO ((CONFIRM-STATEMENT ?SP ?HR ?STATE ?PRP)
         (NEGATIVE-U        ?HR ?SP ?STATE ?PRP)
         (CONFIRMATION      ?SP ?HR ?STATE ?PRP))
  :CONS (:ordered)
  )"
)
(gplanner::defschema :INTERACTION-PLAN
  "(
  :HEAD (CONFIRM-STATEMENT-UNIT ?SP ?HR ?STATE ?PRP)
  :PREC ()
  :DELE ()
  :EFFE ()
  :DECO ((CONFIRM-STATEMENT ?SP ?HR ?STATE ?PRP)
         (AFFIRMATIVE-U     ?HR ?SP ?STATE ?PRP)
         (CONFIRMATION      ?SP ?HR ?STATE ?PRP))
  :CONS (:ordered)
  )"
```

```
)

          - 中略 -

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; DOMAIN plans
;;;
(gplanner::defschema :DOMAIN-PLAN
  "(
  :HEAD (WRITE-SOMETHING ?AGN ?SOMETHING ?(PAPER object))
  :PREC ((HAVE ?AGN ?PAPER))
  :DELE ()
  :EFFE ()
  :DECO ((INTRODUCE-ACTION ?AGN ?RCP ?TPC (WRITE ?AGN ?SOMETHING ?PAPER)))
  :CONS ()
  )"
)

(gplanner::defschema :DOMAIN-PLAN
  "(
  :HEAD (SEND-SOMETHING ?AGN ?RCP ?(SOMETHING object))
  :PREC ((HAVE ?AGN ?SOMETHING)
         (KNOW ?AGN (is 住所-1 address))
         (KNOW ?AGN (is 名前-1 name)))
  :DELE ((HAVE ?AGN ?SOMETHING))
  :EFFE ((HAVE ?RCP ?SOMETHING))
  :DECO ((INTRODUCE-ACTION ?AGN ?RCP ?TPC (SEND ?AGN ?RCP ?SOMETHING)))
  :CONS ()
  )"
)

(gplanner::defschema :DOMAIN-PLAN
  "(
  :HEAD (JOIN-EVENT ?agn ?rcp ?CONF)
  :PREC ((know ?agn (need ?(fee 費用-1))))
  :DELE ()
  :EFFE ((PARTICIPATE ?agn ?CONF))
  :DECO ((INTRODUCE-ACTION  ?agn ?rcp ?act (PARTICIPATE ?agn ?CONF))
 (MAKE-REGISTRATION ?agn ?rcp ?CONF)
 (PAY-FEE           ?agn ?rcp ?fee)
 )
```

```
  :CONS ()
  )"
)

(gplanner::defschema :DOMAIN-PLAN
  "(
  :HEAD (PAY-FEE ?agn ?rcp ?fee)
  :PREC ((know ?agn (is ?fee ?(val quantity)))
         (know ?agn (do ?rcp 割引-1))
         (know ?agn (is 期限-1 ?(day time)))
         )
  :DELE ()
  :EFFE ((pay ?agn ?rcp ?fee))
  :DECO ((INTRODUCE-ACTION  ?agn ?rcp ?act (PAY ?agn ?rcp ?fee))
    ;; (transfer-something ?agn ?rcp ?fee)
 )
  :CONS ()
  )"
)

(gplanner::defschema :DOMAIN-PLAN
  "(
  :HEAD (MAKE-REGISTRATION ?agn ?rcp ?obj)
  :PREC ()
  :DELE ()
  :EFFE ((REGISTER ?agn ?obj))
  :DECO ((INTRODUCE-ACTION ?AGN ?RCP ?act (REGISTER ?agn ?obj))
 (SEND-SOMETHING ?rcp ?agn ?form)
         (WRITE-SOMETHING ?agn ?cont ?form)
         (SEND-SOMETHING ?agn ?rcp ?form)
 )
  :CONS ()
  )"
)
(gplanner::defschema :DOMAIN-PLAN
  "(
  :HEAD (PRESENT-PAPER ?agn ?rcp ?obj ?event)
  :PREC ((know ?agn (is 言語-1 ?(lang 言語-1)))
 (know ?agn (is 内容-1 ?(conf event))) ;; ?event の内容
         (know ?agn (hold ?rcp 同時通訳-1))
         )
  :DELE ()
  :EFFE ((PRESENT ?agn ?obj ?event))
  :DECO ((INTRODUCE-ACTION ?AGN ?RCP ?act (PRESENT ?agn ?obj ?event))
```

```
        (WRITE-SOMETHING ?agn ?obj ?form)
 (SEND-SOMETHING  ?agn ?rcp ?obj)
 )
  :CONS ()
  )"
)

;;;; end of plans ;;;;
```

# 参考文献


[1] 飯田 仁, 有田 英一：“4 階層プラン認識モデルを使った対話の理解”, 情処学論, **31**, 6, pp.810-821(1990-6)

[2] 片桐 恭弘: “文脈理解 – 文脈理解のモデル”, 情報処理, **30**, 10, pp.1199–1206(1989-10)

[3] Kohji Dohsaka: “Identifying the referents of zero-pronouns in Japanese based on pragmatic constraint interpretation”, *In the Proceedings of ECAI'90*, (1990-8)

[4] 山梨 正明: ‘対話理解の基本的側面 – 言語学からの方法論と問題点 –”, 『対話行動の認知科学的研究』研究会資料 (1984-2)

[5] Grice,H.P.: “Logic and Conversation”, Syntax and Semantics vol.3, Academic Press (1975)

[6] Allen,J.F. and Perrault,C.R.: “Analyzing Intention in Uterances”, Artificial Intelligence, **15**, pp.143-178 (1980)

[7] Grosz,B.J. and Sinder,C.L.: “Attention, Intention and the Structure of Discourse”, Computational Linguistics, **12**, 3, pp.175–204 (1986)

[8] Schmidt, C.F., Sridharan, N. S., and Goodson, J.L.: “The Plan Recognition Problem: An Intersection of Psychology and Artificial Intelligence”, Artificial Intelligence, **11**, pp.45–83 (1978)

[9] Allen, J. F.: “Natural Language Understanding”, *Chapter* III, *Context and World Knowledge*, The Benjamin/Cummings Publishing (1987)

[10] 山岡 孝行, 飯田 仁：“話し手の考えを理解しながら対話を理解する計算機モデル”, ATR ジャーナル, **10**, pp.15–20 (1991 秋)

[11] Nilsson, N. J.: “Principles of Artificial Intelligence”, Tioga Publishing, (1980), (邦訳 白井他訳: “人工知能の原理”, 日本コンピュータ協会, (1983))

[12] Charniak, Riesbeck, and MacDermott: “Artificial Intelligence Programming”, Second edition, (1985)

[13] 野垣内 出, 飯田 仁：“キーボード会話における名詞句の同一性の理解”, 情処学自然言語処理研報,NL-72-1 (1989-5)

[14] 有田 英一, 山岡 孝行, 飯田 仁：“電話対話における次発話内の名詞句表現の予測”, 情処学自然言語処理研報, NL-81-13 (1991-1)

# 索引