TR-I-0255

汎用階層型プラン認識システム LAYLA
(LAYered pLAn recognition system)

山岡孝行　　西村 仁志*　　飯田 仁
Takayuki YAMAOKA　Hitoshi NISHIMURA　Hitoshi IIDA

1992.3

概要

本報告では、階層型プラン認識モデルの実装を行なったシステムLAYLAについて述べる。LAYLAは従来のシステムに比べ、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、より柔軟な発話の解釈を可能とする、といった特徴を持つ。本システムの実現により、対話構造解析がより現実的速度で行なうことが可能となり、対話システムへの文脈情報の提供が可能となる。

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

目次

第 1 章

はじめに

本報告では、階層型プラン認識モデルを計算機上に実装したシステム LAYLA について説明する。

LAYLA は一つの推論機構として独立したシステムであるが、ATR の対象となっている協調的目標指向型対話の対話構造を解析するためのシステムとして実現した。この目的のため、本報告では、対話構造解析を動作例を中心にして説明する。また、扱う知識（プランスキーマ）もこの目的のためのものである。しかしながら、プラン認識が適用可能であると考えられる問題には、知識ベースの置換によって利用できるように設計を行なった。ここで対話構造とは、対話中に現れる各発話の目的（発話の意図）の関係を表現した構造であり、行なわれている対話に対する一つの理解状態を表すことになる。

本報告では、対話翻訳のための文脈理解の一方法としてプラン認識を採用する時に活用できる道具としてのシステムについて解説することを目指している。従って、本報告は以下の 3 つの部分で構成する:

1. 背景知識とモデルの解説 (第 2 章)
2. システム機能設計 (第 3 章)
3. システムの実装とマニュアル (第 4 章・付録)

第 2 章では、まず LAYLA のとりあえずの目標である文脈理解の問題を解説し、本報告で利用する用語の説明を行なう。そして、従来の階層型プラン認識モデル [1] を概説する。第 3 章では、プラン認識の処理能力の観点から、再度問題について分析し、そこで必要となる新たな機能について述べる。そして、それらの計算機上への実装を念頭においた機能設計を行なう。第 4 章は、LAYLA 利用の手引である。関数リファレンスおよび利用マニュアルからなる。また、付録として、システムインタフェースやさまざまな利用例、及び対話構造解析実験結果を掲載する。

第 2 章

階層型プラン認識モデル

階層型プラン認識モデル (layered plan recognition model) は、対話理解のためのモデルである。本章では、このモデルの対象とする問題や技術の背景となる考え方について概説する。

2.1 文脈理解

対話翻訳において文脈の利用は必須である。

対話翻訳の問題 文翻訳の基本的操作は言語表現の変換であるが、自動翻訳電話のようなコミュニケーションの手段として用いる時、すなわち対話翻訳においては、単に陽に現れる言語表現のみを変換することでは不十分である。つまり、入力原言語表現を構成している要素（言葉）を、対応する出力目的言語の要素に置き換えるだけでは滑らかなコミュニケーションに支障を来すと考えられる。その原因となる事情は多様である。例えば、言語運用における文化的差異がある。例えば、日本人の「はい」を英語の “yes” には訳せないことがある。直訳でなく、意訳が必要なのである。的確な意訳を行なうには、その発話がなされた文脈や状況を考慮する必要がある。

言語表現の問題 また、一般に対話におけるある発話の言語表現は、その伝えるべき情報をすべて明示的に表現しているとは限らない。このような現象の典型例として、省略がある。例えば、「わかりました」という日本語は、一種の省略表現であり、多様な解釈が可能である。この表現が利用される典型的状況は、発話者（話し手）が何かを（話し手のモデルの中で）理解した時であるが、この言語表現には、その何かは明示されていない。また、この表現は何か（命題内容: propositional contents）を理解した時のみならず、行動の要求に対する行為者の約束の言明（行為受諾）にも利用される。さらに、韻律の変化により表出ではなく、要求の発話としても機能すること

がある。このような場合、発話の聞き手は、言語表現により伝達されない情報を参照しながら発話の解釈を行なうだろう。

文脈の機能 広義に文脈とは、上で述べたような伝達されない情報参照の場として考えることができる。具体的に情報提供の場としての文脈は以下のようなタイプに分類できることが知られている [2]: 発話状況・領域知識・注意状態・発話意図、である。これらの要素は、それぞれ、指示・省略・照応・間接性といった実際の言語表現が有する特定の問題への有効性が指摘されている (このあたりのより詳しい解説については、上記参考文献 [2] 参照)。従って、文脈において発話を理解するとは、上に掲げられたような情報参照の場で発話の内容を吟味することであり、そのためには、これら文脈と呼ばれる場を提供する機能が必要になる。

ここでは、上で述べたような文脈の要素となる場を提供できることを、文脈を理解したと考えることにする。この文脈を理解した状態を表現するものが対話構造 (dialogue structure) であり、その状態を作り出すための処理がプラン認識である。

対話構造は、文脈の一側面を表現しているに過ぎない。あらかじめ注意しておくが、ここで説明するシステムあるいは理論的枠組によって、上のすべての場が提供できるわけではない。文脈とは広範かつ繁雑なものであり、本システムはその一面をサポートしているに過ぎない。例えば、発話状況と呼ばれているものは、本稿での合目的的な対話構造から導出することは困難であると考えられる。むしろ、言語表現そのものに有意な情報が含まれていることがある [3]。また、他の要素についても、それぞれに応用の際に必要となる処理が少なからず存在するだろう。そうであっても、対話の履歴・脈絡というものを、ある側面から吟味し、保持していることは、文脈処理全体に対する貢献は大きいはずである。

前段の意味において、ここでの文脈理解は、完全なものではない。対話構造構築という、文脈の積極的定義内において、健全なモデルとシステムの構築を眼前の問題とするものである。

2.2 対話構造

対話構造は、システムが理解した対話の状態を表す一表現である。

ある構造を考える際には、次のような点を考慮する必要があろう: 表現対象の種類、構造の構成要素、構造の解釈。以下では、本報告の目標である対話の内容を対象にして、これらの説明を試みる。

対象対話（協調的目標指向型対話） 対話は多様である。例えば、山梨 [4] は対話のタイプを、その社会行動的意義に着目して、目的指向型（目標指向型）[1]と自由展開型（およびこれらの混合型）に区分した。一言でいってしまえば、グローバルな目的を持つか否かの分類である。グローバルな目的とは、その対話で成就される最大の目的ぐらいに考えると良い。ここで目的の種類には言及しない。しかし、対話を楽しむためといった目的はそれではない。ATR の対象としている（国際会議に関する）問い合わせ電話対話は、目標指向型対話である。質問者は、タスクに関連した目的を持っている。例えば、会議に参加することがかなうように、電話する。また、事務局もその目的の達成のために努力する。彼ら対話参加者は、協調的に対話を行なうと考える。協調的に行なうとは、ここでは、目的から外れた無駄なことはしないぐらいの意味に考えれば良い。特に、問い合わせの電話では、見ず知らずの人間が対話する場合が多いので、井戸端会議的対話の展開（自由展開）は考えない。従って、我々の処理の対象は、協調的目標指向型対話 (cooperative task-oriented dialogue) である。[2]

入力（発話表現） 協調的目標指向型対話に限らず、対話の構成において中心となる要素は、発話 (utterance) である。発話とは、文字通り、言葉を発することであり、その行為者は、対話参加者の一人である。発話には、さまざまな情報が含まれている。しかし、発話が（暗黙的に）含意する情報、意味・意図と呼ばれるようなもの、および、それを解釈する上で利用される知識は、対話構造構築という問題においては（それらを利用し、処理していくことが重要なのだが）二次的なものに過ぎない。利用する知識はデータとして取捨選択できるが、入力発話については、それができない。対話構造構築は、そこで行なわれていることが観察できる発話そのものとその時点で保持されている対話構造そのものを関連づけることが、中心課題である。この意味において、対話の構造を考える上で、唯一明示的入力となり得るのは、発話の記号的側面に過ぎない。

目標（発話意図） 目標指向型対話には目標 (goal) がある。対話構造はその目標に関する構造である。しかし、この目標を決定することが、対話構造構築の最終目的ではない。目標も、対話構造というデータ構造においては、その構成要素に過ぎない。対話内の（最終的）目標は、その下位目標（副目標: subgoal) に分解することができる [10]。（以下では、明らかな区別の必要がない限り、最終目標・下位目標を単に目標と総称する。）すなわち、構造内における目標間の関係は、上位下位関係に束縛する

[1]山梨の用語では、『目的指向型』であるが、“goal” という語の訳の convention に従い、本稿では以降『目標指向型』を用いる。この 2 つの用語に、意味的差異は皆無である。

[2]協調的目標指向型対話の厳密な定義を行なうことは、困難を要するので、ここではこれ以上言及しない。しかし、処理対象が明確でないことには、モデルの評価およびそのモデルに基づいて出てきた出力の評価が困難になる。すべては、この段のような（人々の）常識的解釈を信じるのみである。会話の協調性や合目的的要素をより深く考察するのであれば、Grice の協働原則 [5] や [6],[7] を参考にすると良い。

ことができる。これらの関係をある構造として表現したものが対話構造である。この目標は、しばしば、対話の構成要素である発話の目的という意味で、発話の意図と呼ばれることがある。

対話構造は、木構造で表現できる。構成要素である発話と目標をノードとし、それらの上位下位関係をリンクで表現する。ここで関係を表すリンクの種類は、後述する推論規則に基づく。推論規則は、ノード間の連鎖の可能性を規定した規則である。

対話構造の解釈 一つの対話構造は、ある一つの理解の状態を表現しているに過ぎない。対話構造が木構造である限り、任意のノードが複数の上位ノードを持つことができない。一つの対話構造内では、ある発話や副目標は唯一の目標ために存在すると考える。逆に、対話構造内では、目標の下位分解の際の選言 (disjunction) を表現しない。従って、一つの対話構造に構造的曖昧さはない。

しかし、現実には、ある発話の関与する目標は、特定の文脈においてさえ複数考えられることがある。先の例「わかりました」は良い例である [1]。例えば、以下の対話に続いたとしよう；

質問者 「登録用紙を送って下さい。」
事務局 「ご住所を、お願いします。」
質問者 「住所は、京都府相楽郡です。」
事務局 「わかりました。」

ここでは、登録用紙の送付という目標に対して行為受諾をしているのかあるいは住所について確認をしているのか、文脈におけるその意図においてさえ曖昧である。このような時は、それぞれの解釈に構造を割り当てる。つまり、システムは2つの対話構造を、その時点（「わかりました」の発話後）の理解状態として、持つことになる。これら複数の対話構造は、対話の進展により淘汰されていく。ある構造が、その中で後続する発話を関連付けられなかった時（かつ他の構造で関連付けが可能であった時）、その構造は、理解状態から外される。例えば、上の例の後に、事務局が「それでは用紙をお送り致します。」と続けたとすれば、「わかりました。」の解釈は、住所の確認となろう。なぜなら、「わかりました。」を行為受諾としてしまうと、次の発話がその構造と関連を持つには、住所確認のそれより間に入る目標が多くなり、複雑になる[3]。繰り返すが、これら各々の対話構造には構造的曖昧さはない。ただ、曖昧な解釈（意味的曖昧さ）は、システムが複数の対話構造を保持することで、表される。

[3]残念ながら、現状の LAYLA では、これを処理するに足る知識を持っていないので、この2つの例は候補として残ってしまう。

以上簡単に整理すると、対話構造は、システムが保持する対話の一理解状態であり、発話と目標とそれらの関係からなる木構造である。さらに、システムは、解釈の曖昧さに対応して複数の理解状態を保持することができる。ここで述べたことは、一つの考え方である。計算機実装においては、対話構造管理のための内部データとして、全く同じ部分を複数持つ必要はない。この点については後述する。また、選言の記述については、直接対話構造管理の問題ではなく知識表現上の問題と考えられるが、今後の課題として残る。

2.3 プラン認識

プラン認識手法は、行為者の持つ知識を利用した一推論手法である。

問題 プラン認識の問題は次のように一般化される [8]:

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

対話構造解析では、各々、発話の系列、発話の意図、対話構造、に対応する。つまり対話理解におけるプラン認識は、入力発話の系列から発話の意図を推論し、その推論結果を対話構造として構築するプロセスになる。

漸進的処理 対話処理は、行なわれる発話ごとに、漸進的に進めなければならない。まず、対話は動的な現象である。文脈は入力された発話に影響され動的に変化する。一方で、文脈を利用する立場から考えれば、各処理時点での文脈情報を参照したい。完了した対話の発話をまとめて処理するのでは、意味がないし、そのような処理は、対話処理においては不適切である。従って、対話理解におけるプラン認識では、このような動的な現象をとらえるために、時系列上順次なされる発話に応じて順次処理を進めていかなければならない。

プランの記述 プラン (plan) とは、ある目標実現に関する知識である。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。各項目は、プランスキーマのスロットの値として記述する。このような表現方法は、しばしば、STRIPS 表現と呼ばれる。我々の対話行動から考えれば、前節で述べた対話構造内の目標について（ある意味では発話についても）、このようなプランが設定できる。これらプランの具体的種類・内容については、次節で述べる。

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。前節で少し触れたが、対話構造解析における推論規則は、発話および目標間の関係を規定するものでなければならない。

プラン認識のための推論規則を Allen[9] に従って説明する。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。Allen の例は、文章理解であるが、これは対話行動に置いても一般化できる。前段のプランの表現に基づけば、以下のような推論規則がある:

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。[4]

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。

LAYLA では、以上 3 つの推論規則を採用する [1]。

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:

[4]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。

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。

単一化　プランの間の連鎖の成功不成功、スキーマ記述中の項の等価性のチェックは、単一化 (unnification) によりチェックする。従って、上の 3 の変数（変項）や状態記述の整合は、単一化による変数束縛や項書換えによって伝播される。実際の対話を扱う際には、この単一化の条件、等価性の定義、が問題になることがある。例えば、「ご住所をお願いします」という発話と「送り先をお願いします」という発話は、細かいレベルの意図の違いはあるにしても、「登録用紙を送るために送り先を知る」という目標から見れば、ほぼ同じ用件を満たす内容とみて良い。（例えば pure prolog で採用されているような）厳格な単一化では、このような問題を扱うことができない。**LAYLA** では、概念と表層表現の関係を記述した知識ベースを利用し、集合としての同一性を計算するように拡張した単一化のメカニズムを用いる。詳細は次章で述べる。

探索問題　プラン認識は、基本的に探索問題である。特に、アルゴリズムの 2 における知識ベースからのスキーマの取り出しでは、システムの持つ知識が膨大になるに従い組合せの爆発の問題に晒されることになる。**LAYLA** の基本モデルである階層型プラン認識モデルは、この組合せ爆発問題に対する効率的探索として、知識ベースの階層化を提示している。

2.4 階層型プラン認識モデル

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。前節で説明したプラン認識が、GPS のよ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。

4階層プラン 対話理解のための階層型プラン認識モデルでは、4つのタイプのプランを階層知識ベースとして設定している [1]。以下にそれらのプランについて説明する:

インタラクションプラン 対話による情報交換において、協調的な応答実行のための知識である。対話の局所的な目標である情報交換は、対話参加者（話し手と聞き手）の間の順序付けられた発話により実現できる。例えば、住所を知りたい時は、情報の受け手が適切な質問（値の要求）の発話を行なった後に、送り手が協調的な応答を行なえば良い。（「ご住所をお願いします。」「大阪市...です。」）

コミュニケーションプラン ある話題についての協調的な対話を実現するための知識である。例えば、会議申込のためには登録用紙が必要であり、申込希望者が用紙を入手したい時、対話によりこの目標を実現しようと思えば、希望者は送付依頼を意図する発話（例えば「登録用紙を送って下さい」）を行なうようにすれば良い。この目標の実現は、聞き手の応答の種類に左右されるが、対話としてこの話題を完了させるにはどのような情報交換で行なうかを考えるだけで十分である。つまり、コミュニケーションプランは、その時点の話題の種類と情報交換実行のためのインタラクションプランを結び付ける知識と考えることができる。

ドメインプラン ある行為達成のための一般的行動を記述したプランである。（前節のプラン記述の例を参照）
ドメインプランは、文字通り、ある（話題となっている）領域に依存した知識である。

ダイアログプラン 対話の対極的な展開の知識である。例えば、電話対話であれば、始めに相手の確認を行ない、話題を述べ、用件終了の相図や挨拶を行なうことにより、一つの対話を完了することができる。

これらのプランは、その記述対象になる目標の性質にしたがって、下位の方から、インタラクションプラン・コミュニケーションプラン・ドメインプラン・ダイアログプラン、という階層順序を設定することができる。従って、このような階層化した知識を利用したプラン認識は、処理している対象に応じて、その探索対象（多くの場合は、その対象が属する階層以上のタイプのプラン）を限定すれば良い。

階層型プラン認識アルゴリズム 上で述べた知識ベースの階層性を扱うために、前節のプラン認識アルゴリズムを拡張する。基本的には、ステップ1と2を拡張すれば良い：

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。

アルゴリズムの詳細は、次章で説明する。

第 3 章

システム設計

本章では、LAYLA システムの基本的構成および追加した機能について説明する。LAYLA は基本的に、第2章で説明した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。ここでは、それらの問題に対処するために、以下のような機能を検討する:

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、
3. 現実的処理のための、単一化の拡張。

以下本章では、システムの基本的構成とデータ構造・処理方法について説明を行なった後、各拡張機能についての検討と実装方法について述べる。

3.1 汎化階層型プラン認識システム

3.1.1 システム構成

LAYLA の基本的システム構成を図 3.1に示す。これは、第2章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、出力は目標構造（理解状態）である。対話理解の問題においては、入力が発話（発話文の独立した解釈）、目標構造が対話構造になる。システムの主要な部分は、推論部 (inference engine)、知識（プラン）ベース (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、推論部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

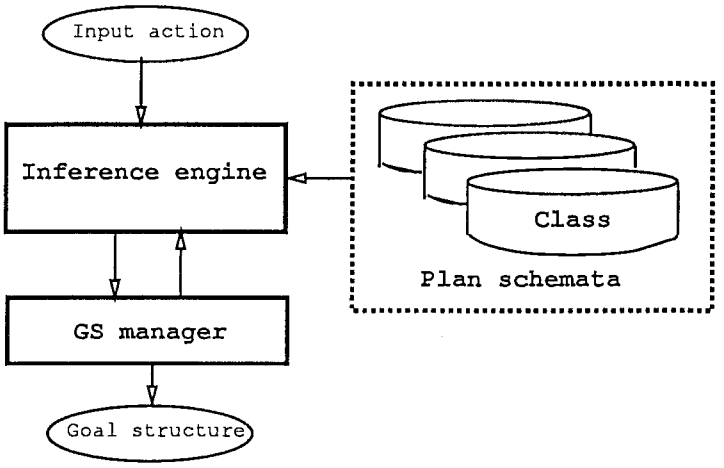


図 3.1: LAYLA のシステム基本構成

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。

3.1.2 データ構造

LAYLA が扱うデータは、基本的に以下の 4 つである（括弧内はそのデータのデータ構造の呼び名を表す）：

- 行為（アクション），
- プラン（プランスキーマ），
- 目標構造（ゴールスタック），
- 制御オブジェクト（コントロール）．

行為は文字通り、ある行為を表現したデータ構造である。ここで、プランもまたある行為についての記述であるので、行為とプランは同一の構造で表現可能である。つまり、2.3で説明した STRIPS 表現により、（冗長ではあるが）双方は表現可能である[1]。また、目標構造は、第 2 章で説明した対話理解モデルにおける対話構造に相当する構造である。従って、その内容は、入力行為とプランを要素として、それらの関係（シ

[1]以下では、そのデータが意味するものを明確に区別したい時は、「行為」と「プラン」の使いわけを行なう。目標に対する（意味ある）行為の系列をプランと呼ぶ見方もあるが、ここでは特にそれに拘らない。単一の行為でも、それが展開できるような記述（つまり STRIPS 表現におけるスロットの内容を持つ記述）であれば、それをプランと呼べることにする。逆に、見出しのみの展開できない行為は、プランと呼ぶことを極力避ける。（ここでの）対話理解においては、発話はプランではなく行為である。

表 3.1: 構造体アクションのスロット

スロット名	データタイプ	内容
header	list	行為の見出し
decomposition	list of lists	副行為の系列
precondition	list of lists	行為実行の前提条件
effect	list of lists	行為実行による効果
constraint	–	行為実行の制約条件
type	symbol	行為のタイプ（クラス）

ステムの理解状態）を表現している。制御オブジェクトについては、3.2でその内容について詳しく述べる。

アクション

構造体**アクション** (action) は、2.3で説明したSTRIPS表現を拡張して実装したデータ構造である。その内容を表3.1に示す。

アクションの見出し (header) は、ある行為の表現であり、それは述語表現形式のリストになる。decomposition は、その行為を実現するための副行為の系列であり、その副行為の見出しが要素となる。precondition, effect は、それぞれ実行前に充足されていなければならない前提条件と実行後に与えられる効果を記述するスロットであり、状態の記述 (行為の見出しの記述と同型式) が入る。constraint は、プラン適用における制約条件を記述する[2]。type は、行為（プラン）のタイプであり、階層型知識ベースにおけるプランスキーマの所属するクラス（あるいはそれ以外で定義されたもの）を与える。

プランスキーマ

プランスキーマは、知識ベース中へのプランの記述であり、構造体アクションを用いて表現する。表3.2にプランスキーマの記述例を示す。これは、値の入手を行なう発話対 (インタラクションプラン:GET-VALUE-UNIT) の例である。例中の、先頭に“?”がついたシンボルは、変項を表す。また、“?”の後にリストが続く項はタイプつき変数である (3.3参照)。このプランは、3つの副行為 (ASK-VALUE,INFORM-VALUE, CONFIRMATION) により成就される。各々の副行為は、ここでは、それを（行為として）表している発話により実現される。例えば、2.2節の「住所」に関する対話例は、このプランによ

[2]現在のインプリメンテーションでは、この制約条件には、唯一 :ORDERED というシンボルのみを与えることができる。これは、プランの decomposition の実行に順序 (リストの先頭から終りの方向) がある場合に指定する。

```
#S(ACTION
    TYPE     :INTERACTION-PLAN
    HEADER   (GET-VALUE-UNIT ?SP ?HR ?(OBJ value) (IS ?OBJ ?VAL))
    PRECONDITION  ((KNOW ?HR (IS ?OBJ ?VAL)))
    DELETE-LIST   NIL
    EFFECT        ((KNOW ?SP (IS ?OBJ ?VAL)))
    DECOMPOSITION ((ASK-VALUE    ?SP ?HR ?OBJ (IS ?OBJ ?VAL))
                   (INFORM-VALUE ?HR ?SP ?OBJ (IS ?OBJ ?VAL))
                   (CONFIRMATION ?SP ?HR ?OBJ (IS ?OBJ ?VAL)))
    CONSTRAINT    NIL
    )
```

図 3.2: プランスキーマの記述例

り実現されていると考えることができる。このプランの完了によりその効果 (KNOW) が理解事項に加えられる。このプランにより行なわれる情報交換の話題は、何らかの値を持つオブジェクト (?(OBJ value)) であり、その命題内容は日本語の「?OBJ は?VAL だ」に相当する (IS ?OBJ ?VAL) で表現されている。

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。このスキーマ登録用マクロは、type の値により、登録先知識ベースを判断する。(**defschema**(*Macro*): 第4章参照)

発話 (入力行為)

対話理解における唯一明示的入力は、発話である。発話行為論に基づけば、任意の発話を行為として解釈することができる。ここでの解釈は、発話の (文脈に依存しない) 独立した発話文としての解釈に相当するものである。LAYLA では、文の構成 (構文的知識) までプラン認識 (プラン) の対象として扱わない。このような対話理解の範囲においては、入力を表すアクションは、一般的に precondision, effect, decomposition スロットの値を持たないアクションである。なぜなら、文脈に依存しない発話の行為としての記述は、文脈に依存しないがゆえに、その構文的 (あるいは一文内での意味的) 情報しかそのスロットを埋めてはならない。そして、そのような情報はここでの対話理解では扱わないし、また、扱ってもあまり有意ではなさそうな気がする。

LAYLA では、入力行為の所属する行為のクラスとして、:input を与えている。これらの見出しは、入力された行為の記述 (発話解釈でいう発話の表現 [12]) の内容 (そのもの) である。対話理解システムとしての LAYLA では、発話の表現を (type = :input である) 構造体アクションに変換するインタフェースを備えている (**set-input-**

表 3.2: 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)

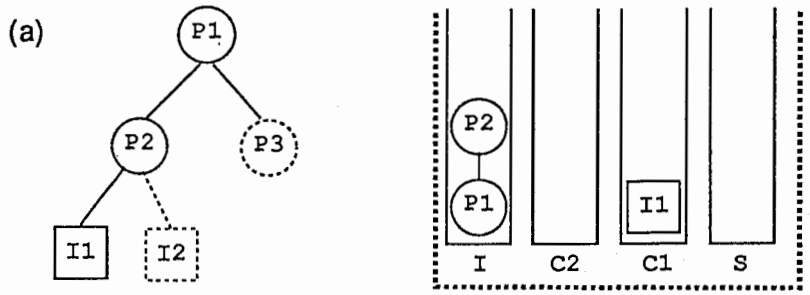


図 3.3: ゴールスタックのイメージ

action(*Function*): 第 4 章参照)

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 3.2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。

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。

ゴールスタックの内容と解釈イメージの簡単な例を、3.3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[3]、P_2 は **C2** に移される。

ゴールスタック管理の詳細については、次節で述べる。

3.1.3 階層プラン認識アルゴリズム

上で述べたデータ構造を利用してプラン認識メカニズム実装のためのアルゴリズムを示す。以下の説明では、基本的に次のような記法を用いる:

1. データ構造はイタリック体 (e.g. A_x など) で表す。
2. 関数表現として、 Function_name(*Arg1, Arg2, ...*) を用いる[4]。

プラン推論規則

2.3節で述べたように、プラン認識のための推論規則として、

1. decomposition chain,
2. effect chain,
3. precondition chain,

---

[3]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。

[4]以下の説明で関数を導入する場合、その意味が明示的と思われるものについては、その関数の説明を省略する。例えば、Lisp における構造体のアクセス関数（構造体名＋'-'＋スロット名, e.g. action-header, etc.）などの説明は省略する。これらの意味がわからない時は、Lisp の入門書を参考にしていただきたい。

の3つを設定する。これらの連鎖のための規則は、基本的に、構造体アクションの間の連結可能性を定義したものになる。そして、これらの内容の違いは、対象としているアクションのどのスロットを参照するかの違いのみである[5]。以下、各規則のアルゴリズムを述べる。

---

decomposition-chain(A_i, A_g)

入力: A_i: 入力となる行為、任意の行為

A_g: 目標となる行為、任意の行為

出力: 連鎖後の行為のリスト (list A_i, A_g)

1. x を **action-header**(A_i), y_k を **action-decomposition**(A_g) とする ($k = (1, n)$ n は副行為の数)。
2. for k from 1 to n,
   (a) y_k がすでに連鎖している状態であれば、$k+1$ として 2 へ、
   (b) x と y_k の単一化 (unify(x, y_k)) を行ない、
      i. 成功すれば、3 へ、
      ii. それ以外は、$k+1$ として 2 へ
   (c) $n < k$ になれば失敗終了。
3. 単一化の結果を x, y_k に対して適用し (変数の束縛:binding)、それらを要素とするリストを戻して終了。

---

effect-chain: effect chain は、基本的に、decomposition chain の目標となる行為の検査対象を副行為から効果に変更することで実現できる。従って、上で述べた decomosition chain の step.1 を以下の記述に変更すれば良い。

1. x を **action-header**(A_i), y_k を **action-effect**(A_g) とする ($k = (1, n)$ n は効果 (の状態記述) の数)。

[5] Allen[9] は、入力のタイプ (action,state,goal) によりこれらの利用を区別している。このような考え方の導入は、制御機構 (3.2節) で行なう。

effect chain の検査対象について、対話理解のためのモデルでは、発話が遂行する行為の記述（情報伝達行為に基づく発話の表現）とそれにより運ばれる情報内容（命題内容）を明確に区別しているので、入力行為の検査対象にも注意する必要がある。これは、以下の2点である：

1. effect chain の対象は、命題内容である。従って、effect chain の入力対象は、アクションの見出しの命題内容にする。

2. 対話における発話では、しばしば、命題内容が省略されることがある。このようなものを対象したとき、effect chain は無条件に成功してしまう。（なぜなら、省略された命題内容は変項として表され、変項に対する単一化はいつも成功する。）命題内容が省略された発話をこのように扱ってしまうと、対話の解釈の数が爆発的に増大する。よって、効率的処理の観点から、このような場合は effect chain の対象から排除する必要がある。

この2点を鑑みると、対話理解を対象にしたプラン認識における effect chain は、以下のようにした方が良い:

---

effect-chain(A_i, A_g)
入出力については **decomposition-chain**(A_i, A_g) と同じ。

1. x を **propositional-contents-of**(**action-header**(A_i)) （A_i の命題内容），y_k を **action-effect**(A_g) とする（$k = (1, n)$ n は効果（の状態記述）の数）。

   (a) この時、x が変項であれば、失敗終了。

   2,3 は **decomposition-chain**(A_i, A_g) と同じ。

LAYLA では、この手続きを実装している。これは最初に説明した一般的なものに用意に変更可能である。

precondition chain: precondition chain についても、effect chain 同様、その検査対象がことなる以外は、decomposition chain と同様のアルゴリズムで実現できる。しかし、precondition chain の入力側の対象が複数になるので、ループが一つ増える。

---

precondition-chain(A_i, A_g)
入出力については **decomposition-chain**(A_i, A_g) と同じ。

1. x_j を action-effect(A_i), ($j = (1,m)$ n は効果（の状態記述）の数), y_k を action-precondition(A_g) とする ($k = (1,n)$ n は前提条件（の状態記述）の数)。

2. for j from 1 to m,

   (a) x_j がすでに連鎖している状態であれば、$j+1$ として 2 へ、

   (b) x を x_j とする以外 decompositin chain の step.2 に同じ

   (c) $m < j$ になれば失敗終了。

3. は decomposition-chain(A_i, A_g) と同じ。

ここでの 3 つの推論規則のアルゴリズムでは、連鎖が一つ見つかった時点で、それを回答とし処理を終了する。この方法は、可能な解釈をできる限り保持するという、我々の対話理解のモデルに反するような気がするであろう。しかし、行為間の連鎖の種類や（行為の記述中の）連鎖の対象に言及せず、「ある行為とある行為が連鎖する」という事実だけがわかるだけで十分のような気がする。もちろん、これらの違いを明示することにより、解釈の可能性が広がるであろうことは容易に想像できるし、それを残すためにも、（特定の行為間についても）可能な連鎖を全て求めた方が良いかもしれない。しかし、このようになってしまうのは、知識の記述に問題があるような気がする。つまり、ある特定の行為間の連鎖が複数できるような知識の記述は、曖昧性のない明確な知識表現の観点からするとおかしいと考える。結局、どちらにすれば良いかはさらに検討が必要であろう。

階層プラン推論 (行為の連鎖)

プラン推論は、プラン認識処理の中心であり、究極的には入力行為から目標行為（多くの場合は期待行為）の間の行為の連鎖を求めることである。行為の連鎖は、基本的に、知識ベースと目標構造とを対象にした探索問題と見ることができる。以下で、その探索アルゴリズムについて説明する。

ここでの探索アルゴリズムは、A* アルゴリズムを基本にして、それを拡張する[6]。A* を簡単に復習する。（A* の詳細や議論については [13] を参照）データ構造ノードは、この探索が扱うデータであり、基本的に状態スロット (state) とリンクスロット (link) を持つ。I を入力状態のノード、G を目標状態のノード、また局所変数とし

[6]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。

て、$OPEN$ を開ノードを格納するためのリスト（初期状態では I のみが入る）、$CLOSE$ を閉ノードを格納するためのリストとすれば、基本的手続きとして以下のものを準備すれば良い (各手続きの説明の最後の括弧内はその機能を実現した関数名を表す):

1. $OPEN$ から評価値最良のノード（P とする）の取り出し。(**a*getinput**($OPEN$))
2. P と G の等価性の評価。(**a*success**(P, G))
3. P の可能な継続点 (Q_i とする) への展開。(**a*operators**(P))
4. Q_i のリンク管理と $OPEN, CLOSE$ への格納。(副作用を持つので独立して定義していない。)

プラン認識の出力は、行為の連鎖である。これを上の構成から導き出すには、目標ノード (G) から入力ノード (I) までのリンクをたどり、その途中に現れるノードの状態 (P の系列; ここでは、構造体アクションのインスタンスで表現されている) を順にしまっておけば良い。このためには、展開されたノードを記憶しておかなければならない。（理論的には、$CLOSE$ を記憶しておくだけで良い。）

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 の拡張については、上の手続き 3のノード展開で、展開に利用される知識を知識ベースの階層性により制限すれば良い。このために、以下の設定を行なう:

1. ノードのデータ構造に、このようなどこまで探索したかを記憶しておくスロット（queue と呼ぶことにする）を設ける。このスロットには、まだ探索されていない階層クラスのタイプをリスト（順番も意味を持つ）として保持する。
2. queue に値を持つノードを一時的に記憶しておくリストを、新たな局所変数として利用する。(これを $QUEUE$ と呼ぶことにする。)

1 は、例えば、2.4節のモデルで入力のクラスがコミュニケーションプランであれば、コミュニケーションプラン・ドメインプラン・ダイアログプランの 3 階層を探索できる。もちろん、この時、各クラスの知識ベースの探索順位には、階層順序があるので、

1回で探索されるのは1クラスの知識ベースのみである。また、2により、手続き3では、$QUEUE$ の内容から展開するようにすれば良い。[7]

2点目の拡張に関しては、以下のことを行なえば良い:

1. **A*** 処理の初期状態として、$OPEN$ に複数ノードを受け入れる。
2. 手続き2の内容を、G に関するループにする (つまり検索する)。

1については、従来の **A*** の枠組で可能である。これは、対話における発話のように、入力行為に対して曖昧な解釈が可能な時の処置であるが、もしこの曖昧さに何らかの尺度が付けられるとすれば、それを手続き1の評価関数の項に組み込んでやれば良い。しかし、現在の実装では、**A*** の上のレベルで、複数入力の評価をコントロールしている。従って、もし **A*** に複数入力が入る場合は、解釈に甲乙が付けられない場合である。2については、LAYLA では G は構造体ゴールスタックのリストになる。そして、手続き2の目標は、それらのゴールスタックの incomplete の内容 (つまり期待行為のリスト) から P にマッチするものを検索することになる。この検索は、コントロール可能である (次節参照)。

以上の点を鑑みて、複数入力から複数目標に対して階層的知識ベースの検索を行ない行為の連鎖を求める関数を定義する。

A*chain(I, G)

入力: I: 入力ノードのリスト (その状態はアクション (発話の解釈))
G: 現在保持しているゴールスタックのリスト

局所変数: $OPEN$: 開ノードを格納するためのリスト
$CLOSE$: 閉ノードを格納するためのリスト
$QUEUE$: 完全に展開されていないノードを格納するためのリスト

1. I を $OPEN$ とする、
2. $OPEN$ が空であれば終了。

---

[7] リストを $QUEUE$ と呼ぶ理由は後述するが、直観的には、完全に展開されていないノードは $OPEN$ に残しておき、ノードの評価関数に、queue に関する評価の項を組み込めば良いように思える。しかし、現実的にはこのような関数の定義は困難であろう。従って、ここでは、手続き1、つまりノード状態の評価は従来のままにしておく。この方法では、一度手続き1で提示されたノードは、最後まで展開されることになる。この点で、階層型探索の長所が失われているように思える。今後、うまい評価関数の設定方法が与えられることを期待する。

3. $P =$ a*getinput($OPEN$) を $QUEUE$ に入れる。
4. a*success(P, G) が成功すれば, P からリンクをたどった結果求められたアクションのリストを答とし、終了。
5. $QUEUE$ から展開すべきノード Q をもとめ、$Q_i =$ a*operators(Q) とする。
   この時[8]、
   (a) a*node-queue(Q) が値を持たなくなれば、Q を $CLOSE$ へ。
   (b) それ以外は、そのまま。
6. $Q_i(i = 1, n)$ (n は継続点の数) について、従来の A* 同様の、リンクの張りかえと、$OPEN, CLOSE$ への格納をおこない、2 へもどる。

上の中で、a*getinput($OPEN$) は評価値の比較で良い。評価値を決定する評価関数の定義の説明は省略する[9]。説明で利用しているそれ以外の関数を定義する。

---

a*success(N, $GS_{now,j}$)

入力: N: ノード

$GS_{now,j}$: 現時点で保持されているゴールスタック (j=(1,n) n は理解状態の数)

出力: 連鎖のリスト (プラン推論規則参照)/nil

1. A を N の状態 (アクションインスタンス) とする。
2. $E_{j,k} =$ gs-incomplete($GS_{now,j}$) (k=(1,m) m はその要素数) とすると、
   for j from 1 to n, for k from 1 to m について、direct-chain(A, $E_{j,k}$) が成功すればそれを答として、終了。

ここで、direct-chain(A, E) は可能なプラン推論規則の適用により 2 つの引数間の直接連鎖を求める手続きである。(プラン推論規則の適用制御については、3.2節で述べる。)

---

a*operator(N)

---

[8] a*operators(Q) は Q に対する副作用を持つ

[9] 現在の LAYLA の実装では、単純に各リンクのコストを 1 として探索の深さで評価している。

入力: N: ノード

出力: ノードのリスト

1. A を N の状態 (アクションインスタンス) とする。
2. $type$ = pop(a*node-queue(N)) とする。$type$ が nil であれば失敗終了.
3. $type$ の知識ベースから、direct-chain(A, X) が成功する X を全て求め、
   (a) X が見つからなければ、2へ。
   (b) それ以外であれば、X のリストを答として終了。

以上説明してきた関数 a*chain(I, G) により、複数の (曖昧な) 入力行為の目的構造に対する行為の連鎖を求めることが可能である。この方法は、最適探索戦略といわれている A* アルゴリズムの性質を継承しつつも、階層型知識ベースの探索を可能にしている。

ここで、上の関数 a*chain(I, G) の説明では、その出力は連鎖のデータ構造 (リスト) ではなく、連鎖の成否 (T/nil) になる。LAYLA の実装では、手続きの最後として、成功した目標 (G) に対して入力 (I) への連鎖を求めて、それを結果として返している。この出力関数は、途中生成されたノードを管理するノード管理機構ないの検査により実現している。

目標構造管理

次に目標構造の管理について説明する。目標構造管理は、現在の入力から求められた行為の連鎖をその時点の理解状態である目標構造へ結合することである。これにより、新たな入力による理解状態の変化に追従して、新たな理解状態を生成することになる。

目標構造管理は、先に定義したデータ構造ゴールスタックとプラン推論により求められた行為の連鎖 (このデータ構造は、構造体アクションのインスタンスのリストになる) を扱う。この管理は基本的に以下の 2 点に集約される:

1. まず、各データ構造の持つオブジェクトの整合をとる。
2. 連鎖内のアクションをゴールスタックに格納する。

1の代表的なものに、変項の扱いがある。ゴールスタックとの結合前の連鎖の中のアクションの内容は、その目標となったアクションの内容しか反映していない。逆に、ゴールスタックは新たにはいってきた情報を反映した形になっていない。例えば、2.2節の「登録用紙送付」の対話例の3番目の質問者の発話が入力された時点では、プラン推論が求める行為の連鎖の内容には、直前の質問により具現化された「住所の入手」というプランと当該の発話しかなく、そこで始めて具体的な値を持つ“住所＝京都府相楽郡”についての情報は「登録用紙送付」のプランに反映されていない。このプランは、ゴールスタックの中にある。そこで、連鎖の内容からこのプランの（部分的な）内容を具体化する必要がある。逆に、すでにゴールスタック内では既知の情報となっているが、求められた連鎖の内容には反映されていない情報も存在し得る。さらには、これらの相互作用により新たに決定されてくる情報も出てくる可能性がある。このような問題は、Prologにおける変数管理に対応する。

LAYLAでは、基本的操作を:

- 各アクションは、自身のオブジェクト情報を管理するスロットを持つ。このスロットは、以下のオブジェクト整合処理毎に更新される。
- 変項と変項の単一化が起こった際には、入力側のラベルを変項のラベルとする。

とし、連鎖の目標となったアクションGから、ゴールスタックの内容に対して、Gの子供と親について、上の操作を再帰的に行なっている。この時、あるアクションのオブジェクト情報のスロットが、それ以上更新する必要がなければ、そこで再帰処理を停止しても良い。ここでは細かい説明は省く。

各データ構造の内容の整合がとれたところで、処理の2によりの結合を行なう。これは、前節のゴールスタックの定義から、以下の3つにわかれる[10]:

1. ゴールスタックのincompleteスタック内の、連鎖の目標となったアクションの子孫アクションを、complete2スタックへ移動。
2. 連鎖内容の各アクションを、適切なスタックへの移動。
3. ゴールスタックのincompeteスタック内の、連鎖の目標となったアクションの祖先アクションの中で、充足されたものをcomplete1スタックへ移動。(再帰処理)

上の考えに基づいて、ゴールスタックと連鎖を結合する関数を定義する:

[10]ただし、complete2スタック内のアクションを目標として扱うことを可能とする場合は、この限りではない。しかし、その場合についてもこの処理の変形により可能になると考えられる。

---

gs-append-chain(C, GS)

入力: C 行為の連鎖(アクションのリスト): 要素の順序は、リストの先頭が G(目標) とし、以下順に P_1,,,P_j,,,P_{m-1}=I(入力) までとする.

GS ゴールスタック

出力: 新たなゴールスタック

1. GS の内容についてオブジェクトの整合をとる。
2. IS = gs-incomplete(GS), その内容を A_i(i はスタックの先頭から 1...n, n は要素の数), $CS1$ = gs-complete1(GS), $CS2$ = gs-complete2(GS) とする。
3. $A_i = G$ の時、A_1 から A_{i-1} を $CS2$ にプッシュする。
4. for j from 1 to m-1,
   (a) P_j が充足されていれば、$CS1$ にプッシュする。
   (b) それ以外であれば、IS にプッシュする。
5. $G = A_i$ とし、
   (a) A_i が充足されていなければ終了。
   (b) A_i が充足されていれば、それを $CS1$ にプッシュする。
   (c) parent-action(A_i)($i < n$) を新たな A_i として、5(a) に戻る.

3.2 プラン認識アルゴリズムの制御

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。3.1.3節で述べた 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、対話現象のような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。ここでの探索制御は、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、対話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。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。

本節では、以上のような考察に基づき、3.1.3節で示した汎用階層型プラン認識アルゴリズムの制御手法について述べる。まず、基本的メカニズムに対して推論過程を制御するための形態(制御モード)を示す。そして、その制御モードの取り扱いを実装するために、推論メカニズムを拡張する。ここで、探索制御規則は、対象となっている対話状況に固有な制御モードの状態(制御パラメータ)の組合せとして与えることができる。

3.2.1 制御の形態

図3.4に、3.1.3節で示した階層型プラン認識メカニズムの概要を示す。図の内容を再度説明する。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 (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

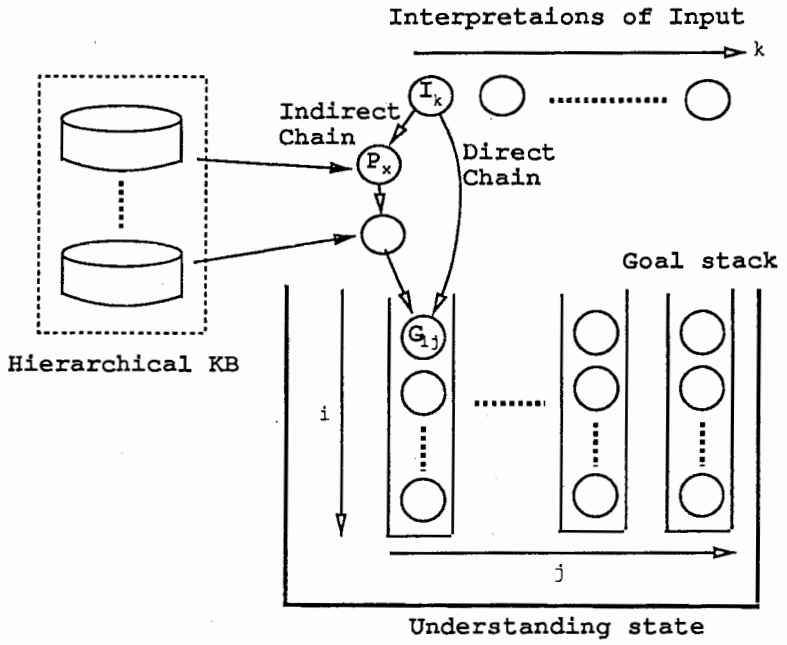


図 3.4: 階層型プラン認識の概要

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。

対象ゴールモード (i,j の方向およびその順序)

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[11]。

間接連鎖回数 (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。

[11]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。

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。

直接間接モード (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。

最短優先モード (yes,no)

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。

検索対象モード (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 (single,layered)

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 (single)、階層順序に従って行なうか (layered) を指定する。

入力順序モード (sequential,parallel)

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 (sequential)、すべての入力を同時に進めるか (parallel) 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。

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。3.1.3節で示した基本的な階層型プラン認識アルゴリズムでの制御パラメータは、上から順に

[default, j, indirect, yes, default, layered, parallel]

となる。

3.2.2 制御の実装

設定した各制御モードを有効にするために、前章での階層型プラン認識アルゴリズムを拡張する。拡張は、制御に必要な情報を格納するデータ構造を定義し、その内容に従って処理の内容を変化させる関数の再定義することにより、実現する。

表 3.3: 構造体コントロールのスロット

スロット名	データタイプ	内容
inference-rules	list of symbols	適用推論規則モード
goal-direction	:BR(j 優先)/:DP(i 優先)	対象ゴールモード
indirect-depth	integer	間接連鎖回数
chain-mode	symbol(:direct/:indirect)	直接間接モード
first-hit	t(yes)/nil(no)	最短優先モード
schema-type	list of classes	検索対象モード
layer-mode	t(layer)/nil(single)	階層モード
input-order	:BR(parallel)/:DP(sequential)	入力順序モード
search-stop-level	numeric	探索停止の深さ
answer-type	:matrix/:tree	出力データ構造の型指定

データ構造: コントロール

まず、設定した制御モードを記憶しておくためのデータ構造 (制御オブジェクト: コントロール) を準備する。データ構造コントロールは、基本的に上で述べた制御モードを、スロットとして持つ。現在LAYLAで定義しているデータ構造コントロールの内容を表 3.3に示す。

表中新たに定義したスロットについて説明する。

search-stop-level は、例えば、対象ゴールモードが :BR で最短優先モードが on の時などに、どこまで推論を進めるかをしておく。このモードでは、最短優先とはいえ、最初に見つかった解のみが、その要請を満たしているわけではない。同じレベル (i) の中には、他の経路 (長さあるいは評価値が先に見つかったものと同じ) で連鎖可能なものが存在する場合もあるので、そのような解を見つける可能性を残しておかなければならない。これは、LAYLA が最適解を見つけるためのシステムではなく、ある指定された範囲で可能な解を探すことを目的としているためである。そのような場合、最初に見つかった目標に至ったノードの評価値を、このスロットに記録しておき、その後この範囲で探索を行なえば良い。

answer-type は、単に出力の形式を指定するのみで、プラン推論内部に影響を与えない。ちなみに、:matrix ではノードを要素とするリスト (解答となる行為の系列に対応する) のリストを返し、:tree では :matrix の中で重複している部分をまとめて木構造敵に表現したリストにして返す。これは、ゴールスタックの構造、あるいはその操作をどのようにするかに依存する。現在は、:matrix 型を採用している。

コントロールは、プラン推論処理を通して、副作用を持つデータ構造である。なぜなら、推論を進めるに従って、状況は変化する可能性があり、その変化に追従した

制御を行なうことが望ましいと考えるからである。従って、対話処理における制御オブジェクトは、対話状況に関するパラメータの表現と見ることもできる。しかし、この内容を実際の状況に合わせて、適切かつ十分な記述を行なうには、今後の努力が必要であろう。

被制御関数

次に、3.1.3で示した関数中で、上の制御オブジェクトを参照し、各モード毎に推論の制御を行なう方法について述べる。各関数の引数には、自動的に制御オブジェクト (Cnt) を加えることになる。

適用推論規則モード 直接連鎖を求める手続き direct-chain(A, E, Cnt) (24頁) 中で参照される。 direct-chain(A, E, Cnt) は、基本的にプラン推論規則に関するループであるが、このループの回数を制限することになる。

対象ゴールモード 認識成功不成功の判別をおこなう関数 a*success($Node, GS, Cnt$) (24頁) のステップ2のループを制御する。つまり、モード値 :BR の時は、 k を先に回すことにより、理解状態中の各ゴールスタック (GS_j) の同レベルの $E_{j,k}$ について判別を行なうことができる。

間接連鎖回数 中間ノード (P_x) の数を制限するには、以下の2つのことを行なえばよい;

1. A* のデータ構造ノードに入力からの距離を記憶しておくスロットを準備する。
   このスロットの書き込みのタイミングは、 A*chain(I, G) のステップ5において行なう。また、同ステップ6において書き換えられる可能性もある。
2. a*getinput($OPEN$) において、開ノードを比較する際に、その対象を距離スロットの値により制限すれば良い。具体的には、$OPEN$ に入っているノードの内、

   ノードの距離の値 $\leq$ 間接連鎖回数の値

   を満たすものだけを、展開の対象とすれば良い。

直接間接モード この制御に関しては、 A*chain(I, G) の外でおこなう。
先に定義した A*chain() は間接連鎖を前提とした手続きである。従って、このモードが :indirect の時は、何も気にせずこの手続きにはいれば良いが、: direct のときは、この手続きの前に、各入力・目標に対して、 direct-chain(I, G) を回す必要がある[12]。

[12]現在の LAYLA では、このモードは使用していない。

最短優先モード a*success(N, GS) により連鎖成功が判別された後に、return の制御を行なうために参照する。この時注意しなければならないのは、search-stop-level スロットのところで説明した通り、あるモードによってはさらに探索を行なう必要があるという点である。

検索対象モード A*chain() のすてっぷ 3 において展開の対象となったノードを QUEUE に入れる際に、ノードの queue スロットにこの値を書き込む。
これは、同ステップ 5 の展開で、その要素が示し知識ベースの検索が終れば queue スロットから削除される。

階層モード 検索対象モードを一括して参照すれば良い。(つまり、検索対象モードのリストの要素全ての知識ベースを一つの知識ベースとみなせば良い。従って、タイミングも同じ。)

入力順序モード このモードを利用には、2 つの方法が考えられる。一つは、各入力ノードに対して、A* の評価関数のパラメータとして扱われる要素に予め重みをつけることである。そうすれば、探索処理中のノード評価の際、この値が良いものが優先して展開対象として提示されることになるので、疑似的に、入力に順序がつくことになる。今一つは、単純に A*chain() の前に入力の比較を行なって、その順序に基づいて、A*chain() を回すことである[13]。

3.3 集合としての同一性による単一化

本節では、プラン認識をより柔軟な推論機構にするための、単一化の拡張について述べる。

特に対話処理などでは、同じものや概念を指示しているにもか変わらず、発話状況や話者の嗜好により違う表現を取ることがある。代名詞化や直示表現なども、その一例であろう。また、そのような顕著な例でなくとも、例えば、“名前”という概念は、“お名前”と表現されたり“氏名”といわれたりする。2.3節でも触れたように、発話中でこのような表現の差はあっても、対話全体における意図として、その発話の意味には差がほとんどないことが多い。

一方、プラン認識の等価性判別は単一化によっておこなう。単一化は基本的に、同値判別 (LISP 関数, eq, eql, equal などに対応する) の拡張であるから、もし、記号の字面のみでその評価を行なってしまえば、上の“お名前”と“氏名”は単一化できない。しかし、対話理解という対極的な視点からは、これらを同一視する方が都合が良いことが多い。例えば、これらの単一化の失敗により、理解状態が倍化する。あ

[13]直観的には、前者の方がすっきりして良い方法であると思われるが、現在の LAYLA では、後者を採用する。これは、何度も述べているように、評価関数の定義ができていないためである。

るいは、これらの理解を容認するためにプランスキーマを加えて必要が出る。結局、理解状態や検索処理の指数的な増加により、現実的な範囲で理解が行なえないことになる。

このような問題に対処するために、LAYLAでは、表現の集合としての同一性の概念[15]に基づき、その集合を利用して意味の同値性を評価する単一化手法を実装する。この実現のために以下のことを行なう:

1. 概念および表現間の関係を記述した知識ベース(シソーラス)[16]を準備する。
2. 知識ベースから、入力に近い表現(あるいは概念)の集合を求めるための検索機構を準備する。
3. 2で求められた集合を対象として、等価性判別を行なうことにより、従来の単一化を拡張する。

さらに、この単一化をこれまで述べてきたプラン認識の枠組に統合するために、プランスキーマの記述を拡張する。この拡張は、プランスキーマ内部の命題内容記述中に**タイプ付変数**を許すことで行なう。タイプには、任意の概念・表現を指定できる。このタイプ付変数を導入することにより、上の単一化への対処という目的の他に、任意変数(タイプの付かない変数)による解釈可能性の爆発を押えることができる。

以上により、対話参加者によって違った表現がとられる場合等、発話の意味の大局的同値性による判断を可能にし、柔軟なプラン認識が可能になる。さらにタイプ付変数の導入により、柔軟なプランスキーマの記述と効率的な処理が可能になる。

3.3.1 知識ベース

概念と表現に関する知識は、2つのネットワーク形式の知識ベースに展開される。1つは概念間の関係表す概念ネットワークであり、もう1つは概念とそれらを表現するための言語表現感の関係を表す表現ネットワークである[14]。

図3.5に知識ベース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[15]。

[14]現在のLAYLAの実装では、概念ネットワークしか利用していない。これは、LAYLAへの入力が、日本語解析機構により解析された意味表現であり、この内容には概念化された辞書記述のラベルが用いられているからである。

[15]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。

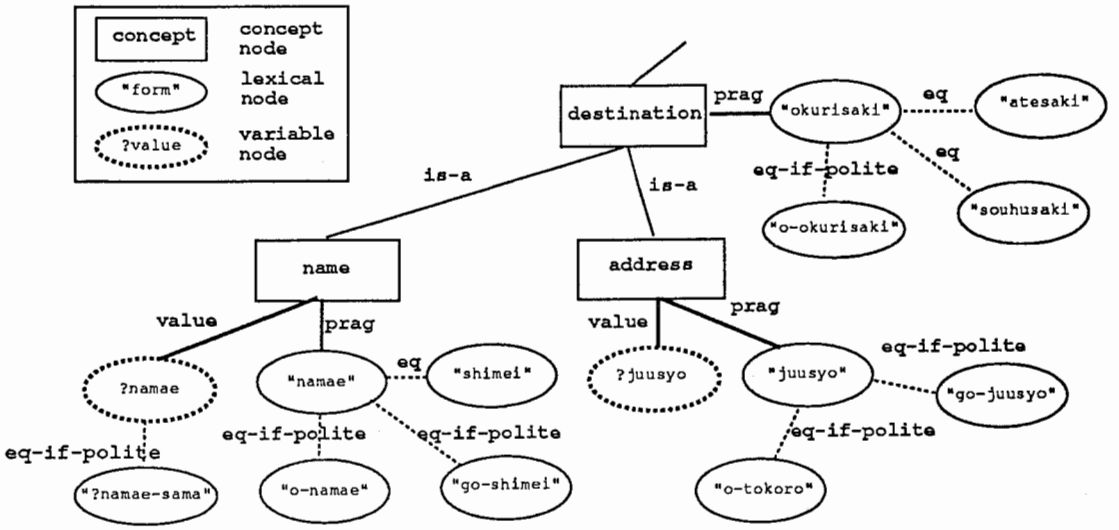


図 3.5: 概念・表現知識ベース

知識ベースのエントリの編集は、LISP マクロによって行なう。以下に、上図のネットワークを構成するためのエントリの記述例を示す。

```
(defconcept name
            (is-a destination))

(defword ''namae''
         (prag name))
(defword ''shimei''
         (eq  ''namae''))
```

この例は、概念“名前 (name)” は概念“送り先 (destination)” と、“名前” の言語表現である“名前 (namae)”,“氏名 (shimei)” のエントリを記述している。マクロ名 **defconcept, defword** は、それぞれ、概念エントリ、表現エントリを表し、前者で定義されたエントリは概念ネットワークへ、後者は表現ネットワークへ格納される。第1引数が、ノードを表すラベル名であり、第2引数以下にリンクの種類とその行き先を記述している。

3.3.2 概念集合検索

次に、上で構成した知識ベースの検索について説明する。ここでの検索の目的は、集合としての単一化のために、入力となる概念あるいは表現から関連する要素を集めてくることである。このため、検索には以下の機能を設定する:

知識ベースの種類による検索関数の設定 概念ネットワークと表現ネットワークを各々の内容のみ検索する機能と、双方まとめて検索する機能。

階層方向による検索制限 リンクの上位ノードを検索するか、下位ノードを検索するかを指定する。

リンクの長さによる検索制限 基本的には、直上・直下が検索できれば、ループの回数を指定するだけで良い。

リンクの種類による検索制限 複数の種類を与えることができるようにする。

2番目以降の機能は、検索の主関数となる1番目の関数のキーワード変数として指定できるようにしておく。

以上のような検索機能により、与えられた概念や表現のラベルから、関連する概念や表現の集合を、状況に応じて求めることができる。

3.3.3 単一化アルゴリズムの拡張

以上述べてきたものから、単一化を集合による単一化に拡張する方法を説明する。拡張は、まず、抽象的な概念による変数のタイプ制限を可能にするために、タイプ付変数の概念を導入する。次に、そのように拡張したデータ構造と入力を集合として扱うように、単一化アルゴリズムを変更する。

タイプ付変数

タイプ付変数は、指定されたタイプに関連する具体値と単一化が可能な変数である。タイプは、概念ネットワーク中の任意の概念を指定できる。また、関連する要素とは、指定されたタイプから、単一化規則の許す範囲で、検索されたノードを要素である。このような要素は、上の検索機構により、それら要素の集合として与えられる。

タイプ付変数は、以下の記法で表現する。

?(変数名 タイプ1 タイプ2 ..... タイプn)

タイプは、第2引数以下何個でも指定できる。また、リストが長さ1(変数名しか持たない)タイプ付変数は、従来の任意変数 (?変数名) と同義となる。タイプの並びに順序はないが、以下で述べる手続きの関係から、左の方のタイプが優先して検索対象になる。

単一化アルゴリズム

純粋な単一化のアルゴリズムとして、Charniak[14] のアルゴリズムを基本にする。以下に、任意の2つのデータ構造を単一化するアルゴリズム (unify(x,y)) の概要を紹介する。なお、このアルゴリズムは、データ構造としてリストを扱うが、その長さや順序に意味がある、いわゆる term unify である。

まず、対象 (データ構造) についての実質的トップレベルでの場合分けを行なう:

1. どちらか一方が変数の時、 var-unify(x,y) へ (次の段落で説明)
2. 一方がアトミックであれば, equal(x,y) が成り立てば成功, それ以外は失敗.
3. それ以外の時, 双方の各要素について以上の再帰処理を行なう。

次に、上のステップ1の場合について説明する (var-unify(x,y), x を変数とする):

1. equal(x,y) が成り立つか、または同じ変数 (ラベルが同じ) であれば成功.
2. これまでの単一化処理の履歴を参照して, 変数 x がすでに他のものと単一化していれば,
   x をその値にしてトップレベルを繰り返す.
3. これまでの単一化処理の履歴を参照して, y の中に x が埋め込まれていないかをチェックする.
   そのようなものが見つかった時は, 失敗. (変数の循環の禁止)
4. それ以外の時は, 成功.

上の単一化アルゴリズムでは、その対象となるデータ構造をアトミックデータまたは変数 (あるいはそれらを要素とするリスト) に制限している。我々は、これに加えタイプ付変数を扱いたい。以下でそのための、非常に簡単な一手法を紹介する。

まず、任意変数とタイプ付変数を区別する述語の定義を行なう。ここでは、タイプ付変数は、任意変数の下位分類と考える。従って、任意変数の判別述語関数にタイ

プ付変数を与えた時は、成功するが、逆にタイプ付変数の判別関数に任意変数を与えた時は失敗する。

この定義から、上のトップレベルの場合分けはそのままで、変数の時についての処理にタイプ付変数の扱いを加えれば良いことになる。ステップ1から3は、一方がタイプ付変数でも問題はない。問題は、一方がタイプ付変数であり、かつ他方がそれと同じものでない場合である。この場合わけをステップ4とする:

4. x がタイプつき変数の時は, typed-unify(x, y) へ (次段で説明)

5. それ以外の時は, 成功.

一方がタイプつき変数の場合は、他方について以下の3通りを考えれば良い[16]:

1. タイプ付変数である場合,
   双方のタイプ付変数のタイプ要素が等しい時、単一化成功.

2. 任意変数である場合,
   無条件に成功.

3. それ以外の場合,
   x のタイプ要素から導出される具体値の集合の中に、y が存在すれば成功.

最後のタイプ要素からの集合の導出は、先の概念集合検索を行なうことである。

ステップ1、2は、変数同士、しかも少なくとも一方はタイプを持つ変数、の単一化であるので、そのタイプを伝播に注意する必要がある。ステップ2に場合は、単純に y に x のタイプを付加してやれば良い。従って、この時から、y はタイプ付変数になる。以下で、ステップ1について説明する。

まず、タイプ要素の等価性を定義する。先にも述べたように、あるタイプ付変数には、複数かつ任意のレベルのタイプを与えることができる。等価性は、ある2つのタイプリストが、基本的に次のどちらかを満たしていれば良いとする:

[16] この場合わけに入らないものは、それまでのステップで処理されているはずである。例えば、x がタイプつき変数であっても、y が同じもの時は、var-unify() のステップ1で処理される。この時タイプのマージが問題となるが、LAYLA ではデータ読み込みの際に、すでに同じラベルを持つ変数がシステム中に存在していれば、その記述に関係なく、先に記述されたものと同じ性質にしてしまう。従って、同じラベルを持つタイプ付変数は、システム内では、同じ (タイプを持つ) ものを指示するという大原則がある。

また一方で、タイプ付変数はリストとは単一化しない。これは LAYLA でのリストデータの扱いに起因する。リストは、主に、発話全体の意味や、動詞・名詞といった基本オブジェクトから構成される命題内容を表現している。つまり、LAYLA の中でのリストは、ネットワーク知識ベース中に記述できるようなオブジェクトを表現し得ない。

- intersection が存在する。
- 一方の任意のタイプ要素が、概念シソーラスにおいて、他方の任意のタイプ要素の子どもである。

従って、図3.5の例では、?(x address) と, ?(y destination) や ?(z address name) は単一化可能であるが、?(w name) とは不可能である。また、ステップ1におけるタイプ要素のマージは、この等価性を満たす要素のみを新たなタイプ要素として、双方の変数に与える。従って、上の例で成功する時は、(address) のみになる。

第 4 章

LAYLA マニュアル

本章では、第3章で述べた機能を実装したシステムLAYLAの利用方法および関数について説明する。

4.1 ファイル構成

LAYLAのファイル構成とその内容を表4.1に示す[1]。

システムの起動

起動 LAYLAの起動は、以下の通り:

- LISP を立ち上げる。
- LAYLAの `load.lisp` をロードする。
  (概念検索・予測機構も同時にロードされる)

4.2 データ記法

4.2.1 シンボル・変数・リスト

- 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。

[1]現在、このプログラムソースおよびオブジェクトコードは、以下のところにある:

``as21:/home/yamaoka/System/III/LAYLA/*''

表 4.1: LAYLA のファイル構成

ファイル名	内容
主プログラム群	
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	
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	
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 (``as21:/home/yamaoka/System/III/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

- 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。

4.2.2 概念・命題の定義

- 概念 (*concept*) は、ネットワーク知識ベース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。
  概念定義の例:

```
(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。

- 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。

4.2.3 入力の定義

- 入力 (*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

4.2.4 プランスキーマの定義

- 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、3.1.2節参照):

  schema ::= ('defschema' class body)
  class ::= 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。
プランスキーマの記述例:

```
(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。

4.3 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 (4.3.5節) はパッケージ “NP” である。注意されたい。

4.3.1 ロード・初期化 (load.lisp)

大域変数

---

data-path [*Variable*]

タイプ: pathname

初期値: “as21:/home/yamaoka/System/III/Data” [2]

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

[2] Sparc station の時

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

概念辞書のエントリを記述したファイル名
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。

---

proposition-grammar-filename [*Variable*]

タイプ: pathname

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

命題辞書を記述したファイル名
命題辞書とは、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*)

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。

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・命題辞書・概念辞書をロードする。(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)

4.3.2 プラン推論

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

命題記述ファイル *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

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.

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。

4.3.3 入出力・知識ベース

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).

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). (内容詳細は、3.1.2節参照)

---

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、4.2.4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。

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 3.2参照)

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)

4.3.4 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.

4.3.5 概念検索

以下の関数・変数のパッケージは、"NP" である。

ロード・初期化

概念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"

概念の知識ベースファイル名
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。

---

name-word-file [*Variable*]

タイプ: string

初期値: “word-nodes.lisp”

表現の知識ベースファイル名
初期値 “word-nodes.lisp” には、ATR サンプル会話に出現する名詞句表現の一部および情報伝達行為タイプに対応する文末表現が記述されている。

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

---

word-network [*Variable*]

タイプ: 構造体ネットワーク

初期値: nil network

表現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・表現のノード (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。

---

defword *value info &rest links* [*Macro*]

引数:

1. *value*: ラベル (symbol)
2. *info*: 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表現ノードを生成する。
*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

---

init-word [*Function*]

引数: なし
リターン値:

表現ネットワークの初期化

---

initialize-np [*Function*]

引数: なし
リターン値: 登録ノード数 (第 1 値: 概念, 第 2 値: 表現)

現在のネットワークをクリアして、新たに概念・表現双方の知識ベースファイル (default: *name-concept-file*, *name-word-file*) をロードする。

---

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

概念知識ベースファイルをロードして、概念ネットワークを作る

---

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

引数:

1. *file (&key)*: ファイル名 (default: (merge-pathnames *name-word-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

表現知識ベースファイルをロードして、表現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。

---

remove-word *value* [*Macro*]

引数:

1. *value*: ノードラベル

リターン値: 削除後の表現ネットワーク

*value*で指定されたノードを, 表現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

---

count-word-nodes [*Macro*]

引数: なし
リターン値: 表現ノード数

---

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。

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。

第 5 章

おわりに

本報告では、階層型プラン認識システムLAYLAについて解説した。

LAYLAは、対話構造解析など複雑な問題を扱うための、現実的処理を目指して、

1. 複数入力・複数目標の一括した取り扱い、
2. プラン推論制御機構、
3. 概念の集合としての単一化、

の機能を備える。

現在LAYLAは、ATRサンプル対話を対象に対話構造解析を行なう実験に利用している。ここでは、詳細な実験データを掲載しなかったが、上の機能を備えない従来のシステムよりも、効率的処理・効果的対話構造の構築が可能となった。

今後は、プログラムレベルにおいてより洗練された実装を行なう。また、より多くのデータを処理することにより、制御方法を改良していくのみならず、応用問題として対話を扱うことにより、対話におけるより高度の知識に関する知見を得る基礎としたい。

謝辞

本報告、特にシステムマニュアルの作成が遅れたことを関係各位に深謝致します。

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] 小暮 潔: "解析過程の制御を考慮した句構造文法解析機構の検討", ATR テクニカルレポート, TR-I-0064, (1989)

[12] 山岡 孝行: "情報伝達行為解析システム CAET", ATR テクニカルレポート, TR-I-0254, (1992)

[13] 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))

[14] Charniak, Riesbeck, and MacDermott: "Artificial Intelligence Programming", Second edition, (1985)

[15] 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技報,NL-72-1 (1989-5)

[16] 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技法, NL-81-13 (1991-1)

索引

関数索引

大域変数:

関数・マクロ:

付録 A

データ記述例

A.1 プランスキーマ

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。

```
;
; /home/yamaoka/System/III/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)
  )"
)
(gplanner::defschema :INTERACTION-PLAN
  "(
  :HEAD (CONFIRM-STATEMENT-UNIT ?SP ?HR ?STATE ?PRP)
  :PREC ()
  :DELE ()
  :EFFE ()
  :DECO ((CONFIRM-STATEMENT ?SP ?HR ?STATE ?PRP)
         (INFORM-STATEMENT  ?HR ?SP ?STATE ?PRP)
         (CONFIRMATION      ?SP ?HR ?STATE ?PRP))
  :CONS ()
  )"
)

(gplanner::defschema :INTERACTION-PLAN
  "(
  :HEAD (ASK-STATEMENT-UNIT ?SP ?HR ?STATE ?PRP)
  :PREC ()
  :DELE ()
  :EFFE ((know ?sp ?prp))
  :DECO ((ASK-STATEMENT    ?SP ?HR ?STATE ?PRP)
         (INFORM-STATEMENT ?HR ?SP ?STATE ?PRP)
         (CONFIRMATION     ?SP ?HR ?STATE ?PRP))
  :CONS ()
  )"
)

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

```
  "(
  :HEAD (CONFIRM-ACTION-UNIT ?SP ?HR ?ACT ?PRP)
  :PREC ()
  :DELE ()
  :EFFE ((know ?sp ?prp))
  :DECO ((CONFIRM-ACTION ?SP ?HR ?ACT ?PRP)
         (INFORM-ACTION ?HR ?SP ?ACT ?PRP)
         (CONFIRMATION ?SP ?HR ?ACT ?PRP))
  :CONS ()
  )"
)

(gplanner::defschema :INTERACTION-PLAN
  "(
  :HEAD (CONFIRM-ACTION-UNIT ?SP ?HR ?ACT ?PRP)
  :PREC ()
  :DELE ()
  :EFFE ((know ?sp ?prp))
  :DECO ((CONFIRM-ACTION ?SP ?HR ?ACT ?PRP)
         (NEGATIVE-U     ?HR ?SP ?ACT-1 ?PRP)
         (CONFIRMATION   ?SP ?HR ?ACT ?PRP))
  :CONS (:ordered)
  )"
)

(gplanner::defschema :INTERACTION-PLAN
  "(
  :HEAD (CONFIRM-ACTION-UNIT ?SP ?HR ?ACT ?PRP)
  :PREC ()
  :DELE ()
  :EFFE ((know ?sp ?prp))
  :DECO ((CONFIRM-ACTION ?SP ?HR ?ACT ?PRP)
         (AFFIRMATIVE-U ?HR ?SP ?ACT ?PRP)
         (CONFIRMATION ?SP ?HR ?ACT ?PRP))
  :CONS (:ordered)
  )"
)

(gplanner::defschema :INTERACTION-PLAN
  "(
  :HEAD (ASK-ACTION-UNIT ?SP ?HR ?ACT ?PRP)
  :PREC ()
  :DELE ()
  :EFFE ((know ?sp ?prp))
```

```
  :DECO ((ASK-ACTION ?SP ?HR ?ACT ?PRP)
         (INFORM-ACTION ?HR ?SP ?ACT ?PRP)
         (CONFIRMATION ?SP ?HR ?ACT ?PRP))
  :CONS ()
  )"
)

(gplanner::defschema :INTERACTION-PLAN
  "(
  :HEAD (OFFER-ACTION-UNIT ?SP ?HR ?ACT ?PRP)
  :PREC ()
  :DELE ()
  :EFFE ()
  :DECO ((OFFER-ACTION ?SP ?HR ?ACT ?PRP)
         (REJECT-OFFER ?HR ?SP ?ACT ?PRP)
         (CONFIRMATION ?SP ?HR ?ACT ?PRP))
  :CONS ()
  )"
)
#|
(gplanner::defschema :INTERACTION-PLAN
  "(
  :HEAD (OFFER-ACTION-UNIT ?SP ?HR ?ACT ?PRP)
  :PREC ()
  :DELE ()
  :EFFE ()
  :DECO ((OFFER-ACTION ?SP ?HR ?ACT ?PRP)
         (ACCEPT-OFFER ?HR ?SP ?ACT ?PRP)
         (OFFER-ACTION ?SP ?HR ?ACT ?PRP))
  :CONS ()
  )"
)
|#

(gplanner::defschema :INTERACTION-PLAN
  "(
  :HEAD (OFFER-ACTION-UNIT ?SP ?HR ?ACT ?PRP)
  :PREC ()
  :DELE ()
  :EFFE ()
  :DECO ((OFFER-ACTION ?SP ?HR ?ACT ?PRP)
         (ACCEPT-OFFER ?HR ?SP ?ACT ?PRP)
         (CONFIRMATION ?SP ?HR ?ACT ?PRP))
  :CONS ()
```

```
  )"
)

(gplanner::defschema :INTERACTION-PLAN
  "(
  :HEAD (REQUEST-ACTION-UNIT ?SP ?HR ?ACT ?PRP)
  :PREC ()
  :DELE ()
  :EFFE ()
  :DECO ((REQUEST-ACTION ?SP ?HR ?ACT ?PRP)
         (REJECT-ACTION ?HR ?SP ?ACT ?PRP)
         (CONFIRMATION ?SP ?HR ?ACT ?PRP))
  :CONS (:ordered)
  )"
)

(gplanner::defschema :INTERACTION-PLAN
  "(
  :HEAD (REQUEST-ACTION-UNIT ?SP ?HR ?ACT ?PRP)
  :PREC ()
  :DELE ()
  :EFFE ()
  :DECO ((REQUEST-ACTION ?SP ?HR ?ACT ?PRP)
         (ACCEPT-ACTION ?HR ?SP ?ACT ?PRP)
         ;;(CONFIRMATION ?SP ?HR ?ACT ?PRP)
 )
  :CONS (:ordered)
  )"
)

(gplanner::defschema :INTERACTION-PLAN
  "(
  :HEAD (NEGATIVE-U ?SP ?HR ?OBJ ?PRP)
  :PREC ()
  :DELE ()
  :EFFE ()
  :DECO ((NEGATIVE ?SP ?HR ?OBJ ?PRP)
         (NEGATIVE ?SP ?HR ?OBJ ?PRP))
  :CONS (:ordered)
  )"
)

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

```
  :HEAD (NEGATIVE-U ?SP ?HR ?OBJ ?PRP)
  :PREC ()
  :DELE ()
  :EFFE ()
  :DECO ((NEGATIVE ?SP ?HR ?OBJ ?PRP))
  :CONS ()
  )"
)

(gplanner::defschema :INTERACTION-PLAN
  "(
  :HEAD (AFFIRMATIVE-U ?SP ?HR ?TPC ?PRP)
  :PREC ()
  :DELE ()
  :EFFE ()
  :DECO ((AFFIRMATIVE ?SP ?HR ?TPC ?PRP))
  :CONS ()
  )"
)

(gplanner::defschema :INTERACTION-PLAN
  "(
  :HEAD (AFFIRMATIVE-U ?SP ?HR ?TPC ?PRP)
  :PREC ()
  :DELE ()
  :EFFE ()
  :DECO ((AFFIRMATIVE ?SP ?HR ?TPC ?PRP)
         (AFFIRMATIVE ?SP ?HR ?TPC ?PRP))
  :CONS (:ordered)
  )"
)

(gplanner::defschema :INTERACTION-PLAN
  "(
  :HEAD (CONFIRM-VALUE-UNIT ?SP ?HR ?(VALUE value))
  :PREC ()
  :DELE ()
  :EFFE ()
  :DECO ((CONFIRM-VALUE ?SP ?HR ?OBJ (IS ?VALUE ?VAL))
         (INFORM-VALUE  ?HR ?SP ?OBJ (IS ?VALUE ?VAL))
         (CONFIRMATION  ?SP ?HR ?OBJ (IS ?VALUE ?VAL)))
  :CONS ()
  )"
)
```

```
(gplanner::defschema :INTERACTION-PLAN
  "(
  :HEAD (CONFIRM-VALUE-UNIT ?SP ?HR ?(VALUE value))
  :PREC ()
  :DELE ()
  :EFFE ()
  :DECO ((CONFIRM-VALUE ?SP ?HR ?OBJ (IS ?VALUE ?VAL))
         (NEGATIVE-U    ?HR ?SP ?OBJ (IS ?VALUE ?VAL))
         (CONFIRMATION ?SP ?HR ?OBJ (IS ?VALUE ?VAL)))
  :CONS (:ordered)
  )"
)

(gplanner::defschema :INTERACTION-PLAN
  "(
  :HEAD (CONFIRM-VALUE-UNIT ?SP ?HR ?(VALUE value))
  :PREC ()
  :DELE ()
  :EFFE ()
  :DECO ((CONFIRM-VALUE ?SP ?HR ?OBJ (IS ?VALUE ?VAL))
         (AFFIRMATIVE-U ?HR ?SP ?OBJ (IS ?VALUE ?VAL))
         (CONFIRMATION ?SP ?HR ?OBJ (IS ?VALUE ?VAL)))
  :CONS (:ordered)
  )"
)

(gplanner::defschema :INTERACTION-PLAN
  "(
  :HEAD (GET-VALUE-UNIT ?SP ?HR ?(OBJ value))
;;  :PREC ((KNOW ?HR (is ?OBJ ?VAL)))
  :PREC ()
  :DELE ()
  :EFFE ((KNOW ?SP (is ?OBJ ?VAL)))
  :DECO ((ASK-VALUE    ?SP ?HR ?OBJ (IS ?OBJ ?VAL))
         (INFORM-VALUE ?HR ?SP ?OBJ (IS ?OBJ ?VAL))
         (CONFIRMATION ?SP ?HR ?OBJ (IS ?OBJ ?VAL)))
  :CONS ()
  )"
)

(gplanner::defschema :INTERACTION-PLAN
  "(
  :HEAD (INFORM-WANT-UNIT ?SP ?HR ?(STATE proposition) ?PRP)
```

```
  :PREC ()
  :DELE ()
  :EFFE ((WANT ?SP ?PRP))
  :DECO ((ASK-STATEMENT ?HR ?SP ?WANT (IS ?WANT ?STATE))
         (INFORM-WANT   ?SP ?HR ?STATE ?PRP)
         (CONFIRMATION  ?HR ?SP ?STATE ?PRP))
  :CONS ()
  )"
)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Communication plans
;;;
(gplanner::defschema :COMMUNICATION-PLAN
  "(
  :HEAD (INTRODUCE-ACTION ?AGN ?RCP ?ACT ?PRP)
  :PREC ((want ?agn ?prp)
 (know ?agn (do ?agn ?act1)))
  :DELE ()
  :EFFE ()
  :DECO ((EXECUTE-ACTION ?AGN ?RCP ?ACT ?PRP))
  :CONS ()
  )"
)

(gplanner::defschema :COMMUNICATION-PLAN
  "(
  :HEAD (EXECUTE-ACTION ?AGN ?RCP ?ACT ?PRP)
  :PREC ()
  :DELE ()
  :EFFE ()
  :DECO ((OFFER-ACTION-UNIT ?AGN ?RCP ?ACT ?PRP))
  :CONS ()
  )"
)
(gplanner::defschema :COMMUNICATION-PLAN
  "(
  :HEAD (EXECUTE-ACTION ?AGN ?RCP ?ACT ?PRP)
  :PREC ()
  :DELE ()
  :EFFE ()
  :DECO ((REQUEST-ACTION-UNIT ?RCP ?AGN ?ACT ?PRP))
```

```
  :CONS ()
  )"
)
(gplanner::defschema :COMMUNICATION-PLAN
  "(
  :HEAD (EXPLAIN-STATEMENT ?AGN ?RCP ?fact1 ?PRP)
  :PREC ()
  :DELE ()
  :EFFE ((know ?rcp ?prp))
  :DECO ((CONFIRM-STATEMENT-UNIT ?rcp ?agn ?fact1 ?prp)
         (INFORM-STATEMENT       ?agn ?rcp ?fact2 ?prp)
         )
  :CONS (:ordered)
  )"
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; 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 ;;;;
```

A.2 入力行為

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。これは、情報伝達行為解析システム CATE[12] から出力されたものである。

```
(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")
  )
)
```

付録 B

システムユーザインタフェース

本章は、LAYLA のデモンストレーション用システムインタフェースの解説およびマニュアルである。

LAYLA デモンストレーションシステムは、現在のところ symbolics lisp machine (MacIvory を含む) で動作する。

1　システムの操作手順

・階層型プラン認識、次発話の予測、音声入力候補絞り込みシステムを使用するときの、流れを次に示す。

（1）データの用意[1]

入力文データ・概念辞書・表現辞書・プランスキーマ・プロポジション文法・語尾・掛かり受け結果・情報伝達行為ファイルを用意する。

（2）システムの起動

1）システムのロードをする

ーー＞　リスプリスナーで、

```
load lm06:>nishimura>g-planner-new>load.lisp
```

を入力する。

2）初期画面の表示をする

ーー＞　ｓｅｌｅｃｔ　ー　；　を押すと初期画面が表示される。

3）対話番号の指定をする（デフォルトは「会話ａ」）

ーー＞　Ｄｉａｌｏｇコマンドをクリックし、対話を指定する。[2]

（同時にシステムの初期化も成される）

4）プランニングモードを指定する

（デフォルトは「Ａｕｔｏ　Ｍｏｄｅ」が「Ｏｎ」になっている）

ーー＞　Ｐｌａｎｎｉｎｇ－Ｍｏｄｅコマンドをクリックし、プランニングモードを指定する。[3]

5）システムの初期化をする

ーー＞　Ｉｎｉｔコマンドをクリックする

6）入力文の１番目からプラン認識・次発話の予測・表層表現の選択の処理を行う。

ーー＞　Ｓｔａｒｔ（またはＮｅｘｔ）コマンドをクリックする。

7）対話構造表示メニューなどで、その詳細を表示する。[4]

8）カレントの文の次の処理を行う。

ーー＞　ＮＥＸＴコマンドをクリックする。

9）3）～8）を繰り返す。

---

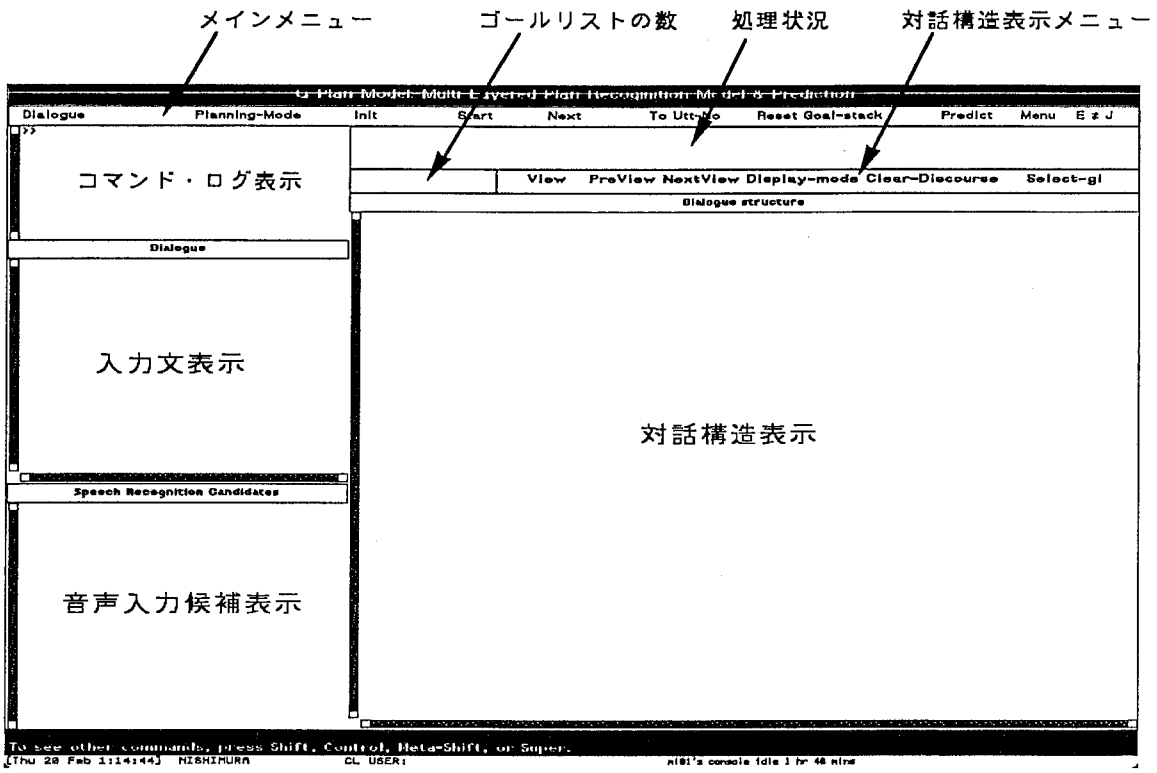
[1]「6．データのフォーマット」参照

[2]「3．メインメニューの説明」の「Ｄｉａｌｏｇｕｅ」を参照。

[3]「3．メインメニューの説明」の「Ｐｌａｎｎｉｎｇ－Ｍｏｄｅ」を参照。

[4]「4．対話構造表示メニューの操作」・「5．ウィンドウ上の操作」参照。

2　メニュー・ウィンドウの説明



(1) メインメニュー[5]

・プラン認識処理のモードを指定したり、システムの起動を行うメニューの集まり。
次の１０個のメニューから成る。

・Ｄｉａｌｏｇ、Ｐｌａｎｎｉｎｇ−Ｍｏｄｅ、Ｉｎｉｔ、Ｓｔａｒｔ、Ｎｅｘｔ、Ｔｏ　Ｕｔｔ−Ｎｏ、Ｒｅｓｅｔ−Ｇｏａｌ−Ｓｔａｃｋ、Ｐｒｅｄｉｃｔ、Ｍｅｎｕ、ＥＪがある。

(2) ゴールリストの数

・プラン認識処理の結果得られる対話構造（ゴールスタック）の数を表示する。

(3) 処理状況

・プラン認識・次発話の予測・表層表現処理の内、処理中の処理の名前を強調表示して示す。

(4) 対話構造表示メニュー[6]

・対話構造の表示スタイルを指定したり、指定した対話構造を削除するメニューの集まり。
次の６個のメニューから成る。

・Ｖｉｅｗ、ＰｒｅＶｉｅｗ、ＮｅｘｔＶｉｅｗ、Ｄｉｓｐｌａｙ−ｍｏｄｅ、Ｃｌｅａｒ−Ｄｉｓｃｏｕｒｓｅ、Ｓｅｌｅｃｔ−ｇｌの６個のメニューから成る。

[5]「3．メインメニューの説明」参照

[6]「4．対話構造表示メニューの説明」参照

（5）コマンド・ログ表示

ユーザが使用したコマンドのログを表示する。

（6）入力文表示ウィンドウ

・実験対象の文を表示するウィンドウ
・現在、解析対象の文を強調表示する。

（7）音声入力候補表示ウィンドウ

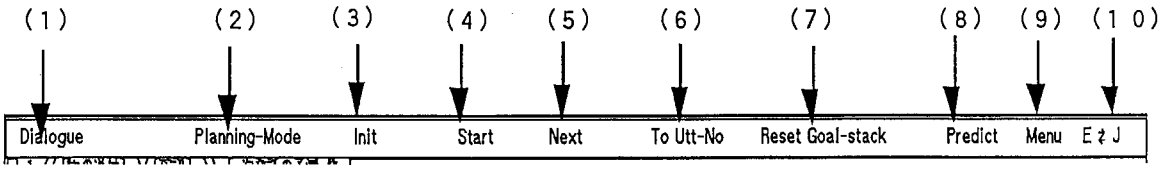
・音声入力候補を表示するウィンドウ
・次発話の音声入力候補を強調表示して示す。

（8）対話構造表示ウィンドウ

・プラン認識処理により得られた対話構造を表次するウィンドウ
・次発話の候補を強調表示して示す。

3　メインメニューの説明

プラン認識処理のモードを指定したり、システムの起動を行うメニューの集まり。



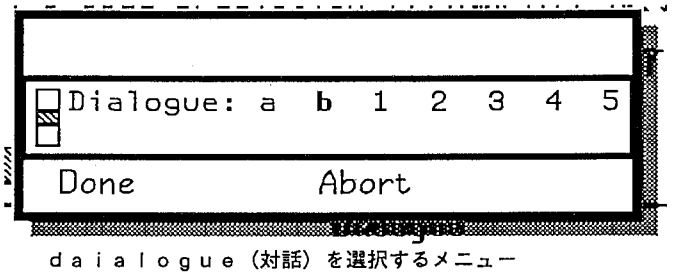
(1) Dialogue

・機能

対話を選択する

・操作

(a) Dialogueをクリックすると、次のメニューが表示される



daialogue（対話）を選択するメニュー

ここで、指定されるDialogueのファイルは次のファイル名の「?」を指定されるDialogueにしたものである。

lm06:>nishimura>g-planner-new>Data>kaiwa-?.lisp

(b) 指定したいDialogue.をクリックし、「Done」をクリックする。

指定を中断したいときは、「Abort」をクリックする。

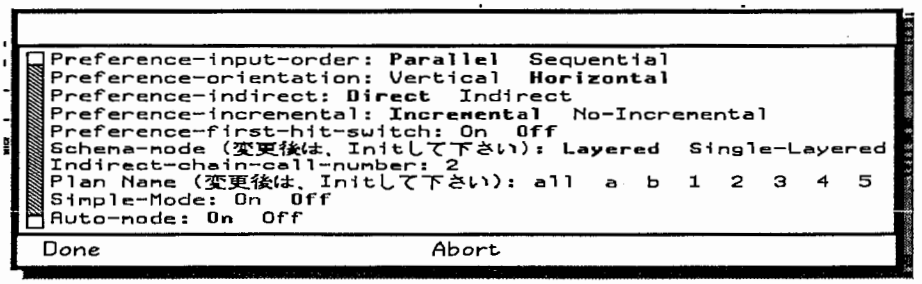
(2) Planning−Mode

・機能

プランニングモードを指定する[7]

・操作

(a) Planning−Modeをクリックすると、次のメニューが表示される



プランングモードを指定するメニュー

(b) 指定したいModeをクリックし、最後に、「Done」をクリックする。
指定を中断したいときは、「Abort」をクリックする。

・モードの説明

(あ) preference−input−oreder
入力順序モードを指定する
- parallel　複数入力に対して、入力の順序で推論を進める
- sequential　複数入力に対して、すべての入力を同時に進める

(い) preferenc−orientation
対象ゴールモード
- Vertical　理解状態1つずつに対して推論を行う
- Horizontal　すべての理解状態を同時に行う

(う) preference−indirect
直接間接モード
- Direct　ある理解状態に対して、まずすべてのゴールリストに直接連鎖を求めたのち間接連鎖を求める
- Indirect　ある理解状態に対して、直接・間接をゴール毎に行う

(え) preference−incremental
横型・縦型モード
- Incremental　間接連鎖を求める際、横型の探索を行う
- NO−incremental　間接連鎖を求める際、盾型の探索を行う

(お) preference−first−hit−switch
最短優先モード
- On　連鎖の際、解が見つかったところでそれ以降の処理を行わないようにする
- Off　間接連鎖回数の範囲をすべて検索する

[7] I章の「3.9　プランニング探索の制御」参照

(か) Ｓｃｈｅｍａ－ｍｏｄｅ

階層モード

・Ｌａｙｅｒｄ　　階層順序に従って知識ベースを検索する

・Ｓｉｎｇｌｅ－Ｌａｙｅｒｅｄ　知識ベースを１階層のものとして検索する

(き) Ｉｎｄｉｒｅｃｔ－ｃｈａｉｎ－ｃａｌｌ－ｎｕｍｂｅｒ

間接連鎖回数

間接連鎖の際に、知識ベースから引き出せるスキーマのインスタンスンスの最大数をしている

(く) Ｐｌａｎ　Ｎａｍｅ

知識ベース名

・プランニングで使用する、知識ベースの名前を指定する

・ここで、指定されるＤｉａｌｏｇｕｅのファイルは次のファイル名の「？」を指定されるＰｌａｎ　Ｎａｍｅにしたものである。

```
lm06:>nishimura>g-planner-new>Data>
                                ?.plan
```

(け) Ｓｉｍｐｌｅ－Ｍｏｄｅ

シンプルモード

・Ｏｎ　知識ベースをインタラクションプランのみとし、プランニングを行う。

使用される知識ベース

知識ベース名　　－＞　ｉ

このとき、他のモードも次のようにセットされる

入力順序モード　－＞　Ｐａｒａｌｌｅｌ

対象ゴールモード　－＞　Ｈｏｒｉｚｏｎｔａｌ

直接間接モード　－＞　ｄｉｒｅｃｔ

横型・縦型モード　　－＞　Ｉｎｃｒｅｍｅｎｔａｌ

最短優先モード　－＞　Ｏｆｆ

階層モード　　－＞　Ｌａｙｅｒｅｄ

間接連鎖回数　　－＞　１

・Ｏｆｆ　他の制御モードどうりの探索を行う

(こ) Ａｕｔｏ−ｍｏｄｅ

制御モード自動セット

・Ｏｎ　　予め指定した制御モードを参照し、プランニングを行う。

・制御モードの指定は、

ファイル

lm06:>nishimura>g-planner-new>Planner>
preference-control.lisp

の変数　*default-sentence-mode-list*

にバインドして行う

・バインドの順序

((" 対話番号"　" 知識ベース名"

(文番号　入力順序モード　対象ゴールモード　直接間接モード

横型・縦型モード　最短優先モード　階層モード　間接連鎖回数)

(文番号　　・・・　　)

・・・)

(" 対話番号"　・・・

) ・・・)

・Ｏｆｆ　　本メニューで指定した制御モードを参照して、プランニングを行う

(3) Ｉｎｉｔ

・機能

システムを初期化する。

・操作

「Ｉｎｉｔ」をクリックする

(4) Ｓｔａｒｔ

・機能

１番目の文をプランニング・次発話の予測・表層表現の選択処理をする。

・操作

「Ｓｔａｒｔ」をクリックする

(5) Ｎｅｘｔ

・機能

次の文を処理をする。

・操作

「Ｎｅｘｔ」をクリックする

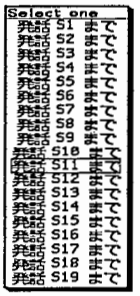
(6) To-Utt-No

・機能

どの発話までかを指定して、連続処理をする

・操作

(a) To-Utt-Noをクリックすると、次のメニューが表示される



どの発話まで連続して処理するかを指定するメニュー

(b) どの発話までかをクリックする。

指定を中断したいときは、マウスをメニューから離す。

(7) Reset-Goal-Stack

・機能

カレントのゴールスタックの状態を指定した文番号のものにする

・操作

(a) Reset-Goal-Stackをクリックすると、次のメニューが表示される

(b) 指定したいゴールスタックをクリックする

指定を中断したいときは、マウスをメニューから離す。

(8) Predict

・機能

次発話の予測・表層表現の選択処理をする。

・操作

「Predict」をクリックする

(9) Menu

・機能

対話またはプランの表示を行う

・操作

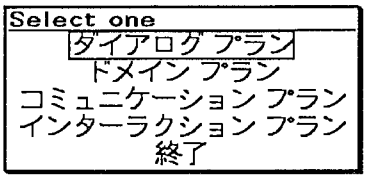
(a) Menuをクリックすると、次のメニューが表示される

各種表示メニュー

(b) 「対話の表示」をクリックすると、以下のように対話を表示する。

```
対話番号 : a
S1:([質問者]) もしもし
S2:([質問者]) そちらは会議事務局ですか
S3:([事務局]) はい
S4:([事務局]) そうです
S5:([質問者]) 会議に申込みたいのですが
S6:([事務局]) 登録用紙は既にお持ちでしょうか
S7:([質問者]) いいえ
S8:([質問者]) まだです
S9:([事務局]) 分かりました
S10:([事務局]) それでは登録用紙をお送り致します
S11:([事務局]) ご住所とお名前をお願いします
S12:([質問者]) 住所は大阪市北区茶屋町二十三です
S13:([質問者]) 名前は鈴木真弓です
S14:([事務局]) 分かりました
S15:([事務局]) 登録用紙を至急送らせていただきます
S16:([事務局]) 分からない点がございましたらいつでもお聞
き下さい
S17:([質問者]) 有難うございます
S18:([質問者]) それでは失礼します
S19:([事務局]) どうも失礼致します
```

(c) 「プランの表示」をクリックすると、どのプランを表示するかを選択するメニューが表示される。



・表示したいプランをクリックする (下図はダイアログプランをクリックしたときのものである)

```
====== ダイアログ プラン : 2 plans ======

-----[ 1 ]--------
-- Action --([会話内容])------
type  :          DIALOGUE-PLAN
Header:          ([会話内容] ?SP1 ?SP2 [料金支払])
Efeect:          NIL
Precondition:    NIL
Deconposition: (([料金支払] ?SP ?HR ?FEE))

-----[ 2 ]--------
-- Action --([会話内容])------
type  :          DIALOGUE-PLAN
Header:          ([会話内容] ?SP1 ?SP2 [会議参加])
Efeect:          NIL
Precondition:    NIL
Deconposition: (([会議参加] ?SP ?CONF))
```

(10) E→J

・機能

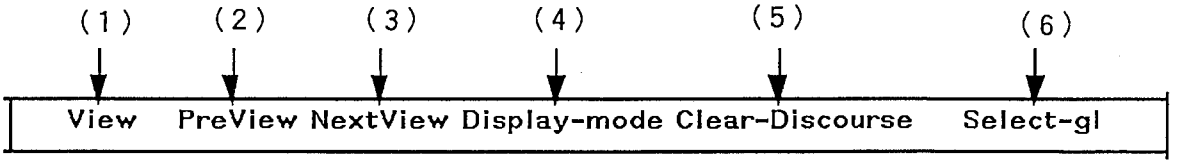
テキストの表示を日本語にするか英語にするかを切り変える。

・操作

「E→J」をクリックする

4　対話構造表示メニューの説明

対話構造の表示スタイルを指定したり、指定した対話構造を削除するメニューの集まり。



(1) Ｖｉｅｗ

・機能

カレントのゴールリストを表示する

・操作

「Ｖｉｅｗ」をクリックする

(2) Ｐｒｅｖｉｅｗ

・機能

カレントのゴールリストより１つ前のゴールリストを表示す

・操作

「Ｐｒｅｖｉｅｗ」をクリックする

(3) ＮｅｘｔＶｉｅｗ

・機能

カレントのゴールリストより１つ後ろのゴールリストを表示する。

・操作

「ＮｅｘｔＶｉｅｗ」をクリックする

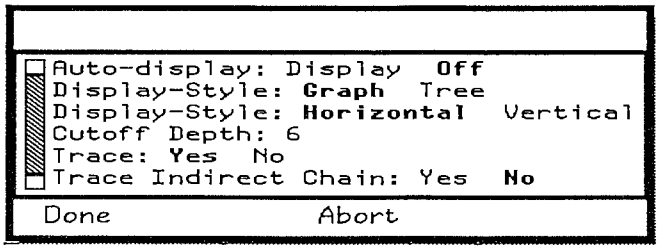
(4) Ｄｉｓｐｌａｙ－ｍｏｄｅ

・機能

ゴールリストを表示するときのモードを指定する

・操作

(a) Ｄｉｓｐｌａｙ－Ｍｏｄｅをクリックすると、次のメニューが表示される



(b) 指定したいＭｏｄｅをクリックし、最後に、「Ｄｏｎｅ」をクリックする。
指定を中断したいときは、「Ａｂｏｒｔ」をクリックする

・モードの説明

(あ)　Ａｕｔｏ－ｄｉｓｐｌａｙ（デフォルトはｏｆｆ）

発話理解が終了した時点で、すべてのゴールリストを表示するかどうかを指定する。

・Ｄｉｓｐｌａｙ

すべてのゴールリストを表示する。

・ｏｆｆ

ゴールリストの内、一つだけ表示する

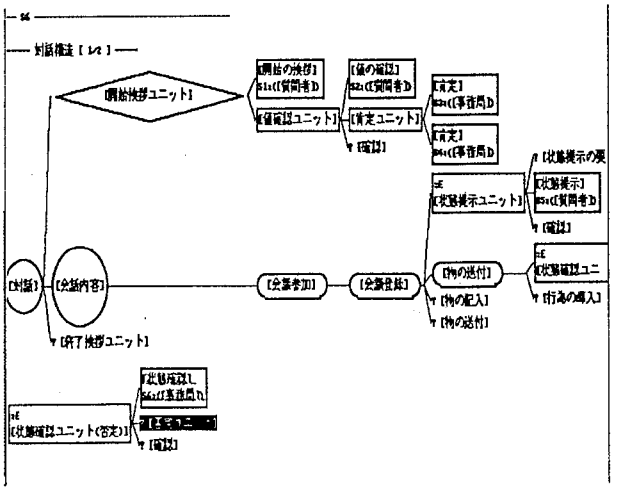
(い) Ｄｉｓｐｌａｙ－Ｓｔｙｌｅ（デフォルトは「Ｇｒａｐｈ」）

ゴールスタックを表示する際の表示スタイルを指定する

・Ｇｒａｐｈ　グラフ表現で表示する（下図参照）

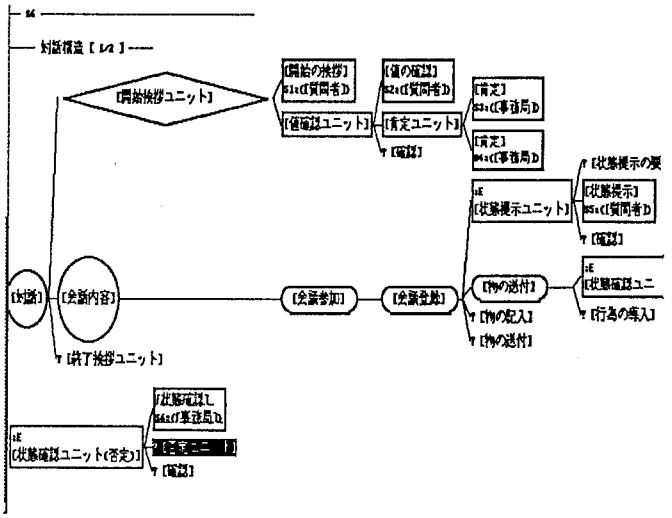
```
-- S6 ------------------------------------------

------ 対話構造 [ 1/2 ] -----
+--[対話]
     |--[開始前挨拶ユニット]
     |    |--[開始の挨拶] S1:([質問者])
     |    +--[値確認ユニット]
     |         |--[値の確認] S2:([質問者])
     |         |--[肯定ユニット]
     |         |    |--[肯定] S3:([事務局])
     |         |    +--[肯定] S4:([事務局])
     |         +--? [確認]
     |--[会話内容]
     |    +--[会議参加[]]
     |         +--[会議登録]
     |              |--:E [状態提示ユニット]
     |              |    |--? [状態提示の要求]
     |              |    |--[状態提示] S5:([質問者])
     |              |    +--? [確認]
     |              |--[物の送付]
     |              |    |--:E [状態確認ユニット(否定)]
     |              |    |    |--[状態確認] S6:([事務局])
     |              |    |    |--[否定ユニット]
     |              |    |    +--? [確認]
     |              |    +--? [行為の導入]
     |              |--? [物の記入]
     |              +--? [物の送付]
     +--? [終了挨拶ユニット]
```

・Ｔｒｅｅ　木構造表現で表示する（下図参照）

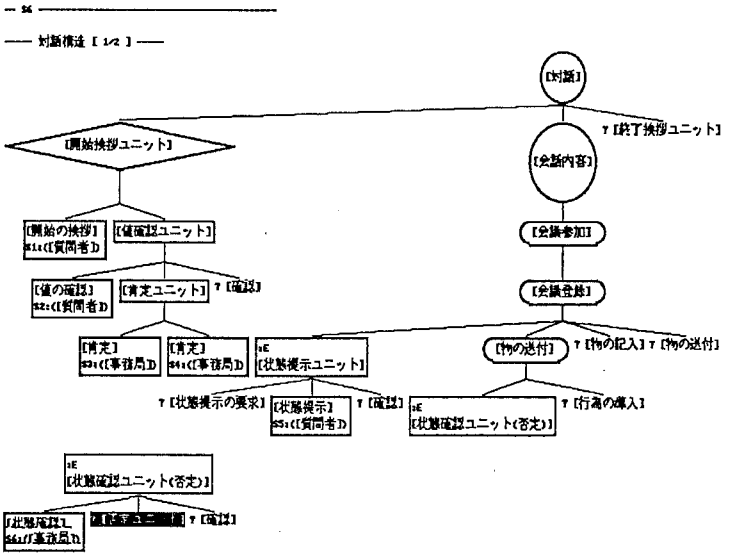


（う）Ｄｉｓｐｌａｙ－ｓｔｙｌｅ（デフォルトは「Ｈｏｒｉｚｏｎｔａｌ」）
（い）のツリー表現の際、ツリーの向きを指定する

・Ｈｏｒｉｚｏｎｔａｌ　横向き（下図参照）



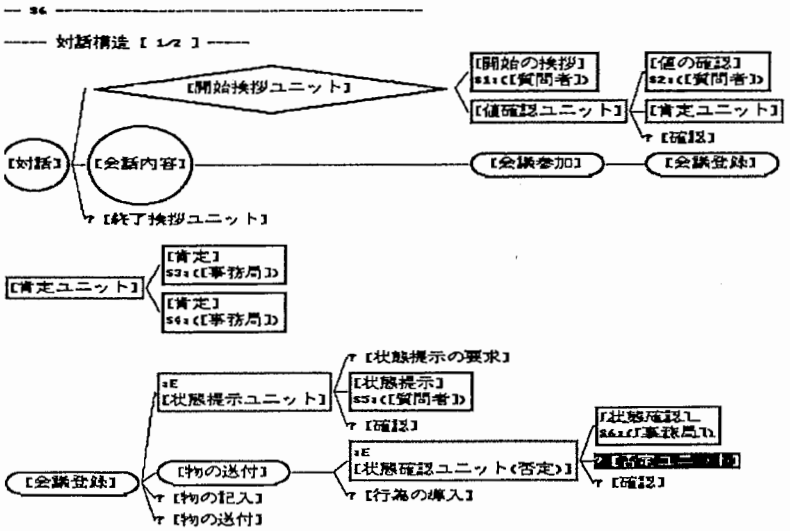
・Ｖｅｒｔｉｃａｌ　縦向き（下図参照）



（え）Ｃｕｔｏｆｆ－Ｄｅｐｔｈ（デフォルトは「6」）

ゴールスタックを表示する際の深さを数字で指定する

（下図はＣｕｔｏｆｆ－Ｄｅｐｔｈを「4」にしたときの表示例である）



（お）Ｔｒａｃｅ（デフォルトは「Ｙｅｓ」）

プランニング処理をしている際、システムが連鎖があるを調べているゴールを強調表示するかどうかを指定する

・Ｙｅｓ　強調表示する

・Ｎｏ　強調表示しない

（か）Ｔｒａｃｅ　Ｉｎｄｉｒｅｃｔ　Ｃｈａｉｎ（デフォルトは「ＮＯ」）

間接連鎖のトレースを表示するかどうか

・Ｙｅｓ　表示する

・Ｎｏ　表示しない

（5）Ｃｌｅａｒ－Ｄｉｓｃｏｕｒｓｅ

・機能

ゴールリスト表示ウィンドウをクリアーする。（ゴールスタック自身はクリアーされない）

・操作

「Ｃｌｅａｒ－Ｄｉｓｃｏｕｒｓｅ」をクリックする

（6）Ｓｅｌｅｃｔ

・機能

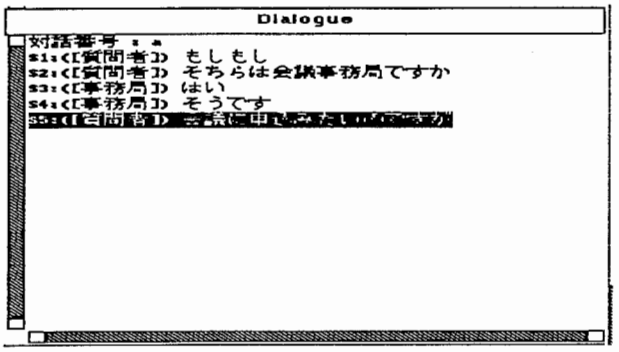
指定した以外のゴールリストを削除する。

・操作

「Ｓｅｌｅｃｔ」をクリックする

5　ウィンドウ上の操作

（1）入力文表示ウィンドウ



・機能

文の内容を表示する

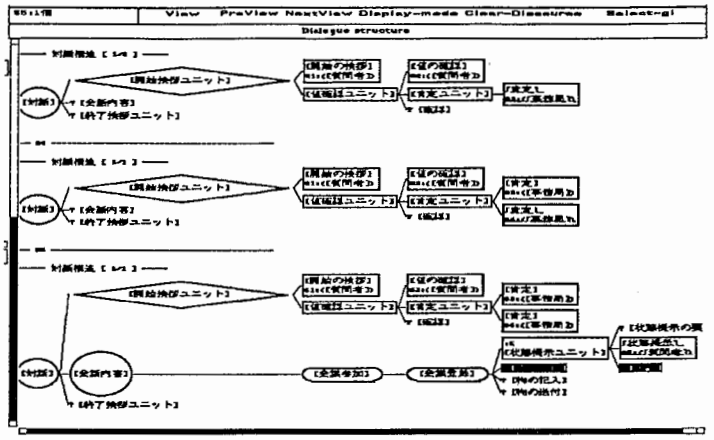
・操作

文をクリックする

（下図は「会議に申し込みたいのですが」をクリックした表示例である）

```
-- sentence --(S5)------------
Sentence: 会議に申込みたいのですが
Speaker: [質問者]     Hearer: [事務局]
・proposition 1
   cat   : [状態提示]
   topic: ?TPCS5
```

（2）対話構造表示



・機能

ノード（アクション・プロポジション）の内容を表示する

・操作

ノードをクリックする

（あ）アクションの「状態確認ユニット（否定）」をクリックした例

```
-- Action --([状態確認]ユニット(否定)])-------------
type   :          INTERACTION-PLAN
Header:           ([状態確認ユニット(否定)] [事務局] [質
問者]
                   (HAVE [質問者] 登録用紙-1))
Efeect:           NIL
Precondition:     NIL
Decomposition: (((47) [状態確認] [事務局] [質問者]
                    登録用紙-1 (HAVE [質問者] 登録用紙
-1))
                  ([否定ユニット] [質問者] [事務局] 登
録用紙-1
                    (HAVE [質問者] 登録用紙-1))
                  ([確認] [事務局] [質問者] 登録用紙-1
                    (HAVE [質問者] 登録用紙-1)))
```

（い）プロポジション「否定ユニット」をクリックした例

```
--- Proposition ---------------
(NEGATIVE-U SP1 SP2 登録用紙-1 (HAVE SP1 登録用紙-
1))
  ---([否定ユニット])--------
Cat        :   [否定ユニット]
Speaker    :   [質問者]
Hearer     :   [事務局]
Topic      :   登録用紙-1
Proposition:   (HAVE [質問者] 登録用紙-1)

  -- 予測 --([否定ユニット])---------------
次発話候補: ([否定ユニット] [質問者] [事務局] 登録用紙-
1
             (HAVE [質問者] 登録用紙-1))
(frozen): ([否定ユニット] NEGATIVE-U-PRAG いいえ ま
だです
             ません)
(tail)   : ([否定ユニット] NEGATIVE-U-PRAG いいえ ま
だです
             ません)
(fuzoku)   : (登録用紙-1 登録用紙)■
```

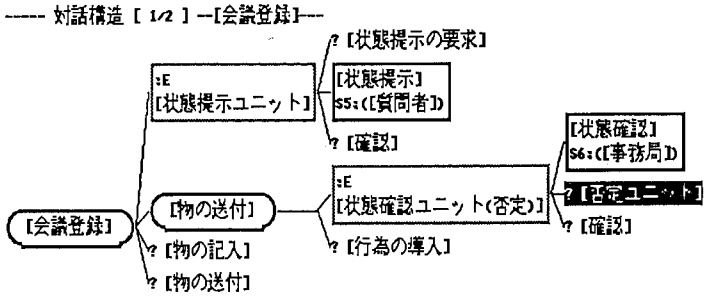
・機能

クリックしたノード以下を表示する

・操作

ノードを２回クリックする

（下図は「会議登録」を２回クリックした表示例である）



6．データのフォーマット

（1）入力文

（a）フォーマット

（文番号　（情報伝達行為　発話者　聞き手　トピック　プロポジション　ストリング）
　　（情報伝達行為　・・・）
　・・・）

プロポジションはプロポジション文法で指定された文法で定義する。

（b）例

```
(s22  (inform-statement  sp2  sp1  ?tpc
        ((pred . participate)
         (agn . sp1)
         (obj . 会議-1))
       "会議に申し込みたいのですが"))
```

（2）プロポジション文法

（a）フォーマット

（プロポジション名　引数の列）

（b）例

```
(send-something  agn  rcp  obj)
```

（3）プランスキーマ

（a）フォーマット

（ヘッダー　前提条件（precondition）のリスト
　　削除のリスト
　　効果（effect）のリスト
　　分解（decomposition）のリスト
　　制約（constrain）のリスト　　；；現在の推論エンジンでは未使用。
　　階層クラス）

階層クラスには次の４つがある

：interaction-plan
：communicatein-plan
：domain-plan
：dialogue-plan

（b）例

```
(((send-something  ?agn  ?rcp  ?something))
 ((have  ?agn  ?something)
  (know  ?agn  ?住所-1)
  (know  ?agn  ?名前-1))
 ((have  ?agn  ?something))
 ((have  ?rcp  ?something))
 ((introduce-action  ?agn  ?rcp  ?tpc
                     (send  ?agn  ?rcp  ?something)))
 nil
 :domain-plan)
```

(4) 情報伝達行為文法ファイル

(a) フォーマット

(情報伝達行為名　行為者　受手　トピック　プロポジション)

(b) 例

```
(ask-value ?sp ?hr ?obj ?prp)
```

(5) 概念辞書

(a) フォーマット

(defconcept　コンセプト名
　(リンク名　(リンクするコンセプト名または値) のリスト)
　(リンク名　・・・))

(b) 例

```
(defconcept 住所-1
            (is-a juusyo))
```

(6) 表現辞書

(a) フォーマット

(defword　単語名
　(リンク名　(リンクする単語)
　(リンク名　・・・))

(b) 例

```
(defword 住所 (if-eq-polite ご住所))
```

(7) 掛かり受け結果

・例

```
(def-bunsetsu-lattice 12
 v1 v999
 ((E1 V1 V999 "そちらは" (:prob 0.013672))
  (E2 V1 V2 "そちら")
  (E3 V2 V999 "  (:prob 0.875027)))
 (:nframes 222 :input "SOCHIRAWA"))
```