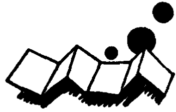
Vol. 30 No. 4　　情　報　処　理　　Apr. 1989

解　説

4. 新しい技術の応用



4.3 AI 技術を応用したプログラミング環境†

毛　利　友　治††

1. はじめに

1970年代の構造化プログラミングを中心としたソフトウェア工学の研究は，1980年代に入って，従来のウォータフォール型ライフサイクル・モデルの限界説が囁かれ始めるとともに，ラピッド・プロトタイピングやオブジェクト指向，論理型プログラミングといった新しいパラダイムの模索へと発展しつつある．同時に，知識工学の発達にともない，AI 技術を取り入れたソフトウェア開発支援に関する関心が高まってきている[1]．

知識工学の発達とは別に，ワークステーションの急速な発達はプログラミング環境に関する関心を喚起し，Cedar[2] などにみられるような高度に統合化されたプログラミング環境の研究開発が開始された．この流れはここしばらくの間続くものと思われる．

本稿では，このような二つの流れから必然的に起こってくる，プログラミング環境に AI 技術を応用する試みの，最近の研究動向のいくつかを紹介する．なお，プログラミング環境とは，狭い意味では，プログラミング言語によるコーディングからデバッグまでの工程を支援する環境と捉えられるが，本稿ではややその範囲を広げ，設計工程における支援環境も含めて解説する．

2. プログラミング環境と AI 研究

プログラミング環境という言葉は古いようでいて，比較的新しい．コンパイラ，リンケージ・エディタといった基本的なツールはソフトウェアの誕生当初から存在したが，プログラミング作業を支援するためのツールの統合化や，総合的な環境が強く意識され始めたのは1970年代後半からであろう．ソフトウェア工学国際会議での発表を中心にみていくと，第2回(1976年) に Programmer's Workbench が発表され，その後，Computer Aided Environment (第4回：1979年)，Software Environment (第5回：1981年)，Programming Environment (第6回：1982年) といったセッションが設けられている．Ada プログラミング環境の備えるべき要件を定めた Stoneman が発表されたのも1980年であり，これらの動きと時期を一にしている．

プログラミング環境は具体的にはプログラミングを支援する種々のツールの集合体となる．プログラミング環境のもつべき機能を前述の Stoneman でみてみると，次の3層になっている[3]．

(1) KAPSE (Kernel Ada Programming Support Environment)

核機能．プログラムの実行時支援，入出力支援，端末との通信管理などを行う．OS に相当する．

(2) MAPSE (Minimal APSE)

プログラミングに最小限必要な機能．エディタ，コンパイラ，デバッガ，リンカ/ローダ，コマンド (JCL) インタプリタ，文書清書機能などからなる．

(3) APSE (Ada Programming Support Environment)

Ada 向きの高度な支援機能．構成管理やプロジェクト管理，ヘルプ機能など．

一方，プログラミング活動とは，渡辺らのソフトウェア開発3元モデル[4]によると，「処理系（プログラマ）が資産系（ノウハウ，設計情報，ソフトウェア部品など）を参照し，種々の判断／推論を交えつつ，対象系（仕様，設計書，プログラムなど）を，仕様から順次詳細化してプログラムへと変換していく過程」ということができる．このような変換活動を支えていくのがプログラミング環境ということができるであろう．

このような観点からみると，MAPSE には変換結果

† Programming environment using AI technology by Tomoharu MOHRI (FUJITSU LABORATORIES Ltd. Software Laboratory).

†† (株)富士通研究所ソフトウェア研究部

を清書する機能と，変換操作を支援する機能の一部は備わっていても，判断／推論を支援したり，資産再利用を支援する機能が欠けている．プログラミング環境への AI 応用の研究は，その欠けている部分を中心に研究開発が進められているようである．

以下 **3.**で，資産再利用に AI 技術を応用する試み，**4.**で，プログラム変換に AI 技術を応用し，変換操作を一部自動化する試みを紹介する．また，直接的ではないが，コンサルテーション／アドバイスという形で判断作業を側面から支援する試みを，**5.**で紹介する．

3. 再利用支援によるプログラミング環境

3.1 IDeA

AI 技術に基づいた再利用支援を目指した研究の例として，まずイリノイ大の Lubars らにより開発されている IDeA[5),6)] というシステムを取り上げる．

IDeA は設計スキーマという考え方に基づいたソフトウェア開発パラダイムを支援する．設計スキーマはいわば抽象化された設計であり，類似の設計のクラス（設計ファミリ）を代表している．設計スキーマは再利用可能な設計情報の集まりとして知識ベース化されており，スキーマが適用できる問題のクラスが大きいほど，再利用性も高い．設計スキーマに含まれる情報としては，スキーマが実現を意図する機能（領域依存のオペレーション），設計上の決定事項間の関係，特定化（specialization）と詳細化（refinement）の規則，動作可能なインスタンスを作り出すためのプロトタイピング情報などがある．

設計スキーマは特定化と詳細化の二つの手段により，下位の設計スキーマと関係づけられる．特定化とは is-a 関係に相当する．すなわち，ある抽象的な設計スキーマの制約の一部を指定することにより，そのスキーマを特定の分野に適用した場合の設計スキーマが得られる．設計ファミリとは実はこの特定化された設計スキーマの集合のことである．**図-1** に特定化によるスキーマの関係付けの例を示す．IDeA では設計スキーマの表現手法として，構造化分析[7)] に用いられるデータフロー図を採用している．この例では，Compute Dependent Revenue という設計スキーマにおいて，入力である Revenue Dependence が fixed rate タイプ（この場合はパーセンテージ）の場合は Compute Fixed Rate Dependent Revenue という特定化された設計スキーマとなり，Revenue Dependence が schedule タイプの場合は Compute Scheduled Dependent Revenue という特定化された設計スキーマとなることを表している．

特定化が is-a 関係であるのに対して，詳細化とは part-of 関係に相当する．**図-2** に詳細化による関係づけの例を示す．

設計スキーマと単純な設計テンプレートの決定的な違いは制約（constraint）の伝播の概念である．制約とはいわば抽象的な設計スキーマを特定化する際に指定される設計上の決定事項といって良い．図-1 の特定化の例において，Revenue Dependence のタイプを Fixed-rate にするか Schedule とするかがこれにあたる．ある設計上の決定が行われ，制約が具体的に定まると，特定の設計スキーマが選択されることになる．特定化された設計スキーマに，それに対応した詳細化

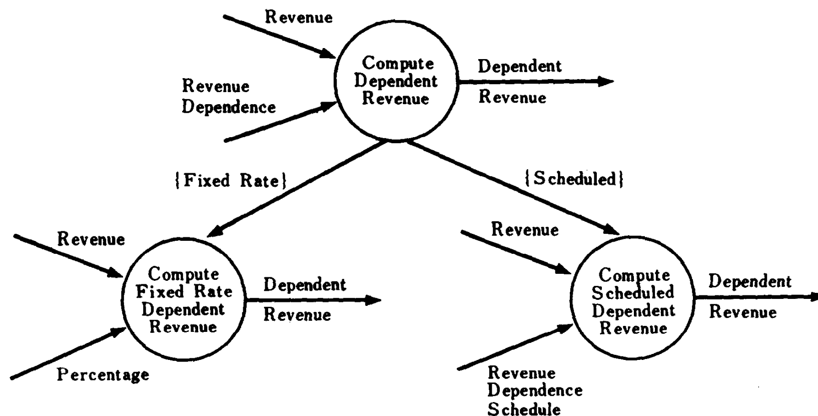


図-1 設計ファミリの特定化

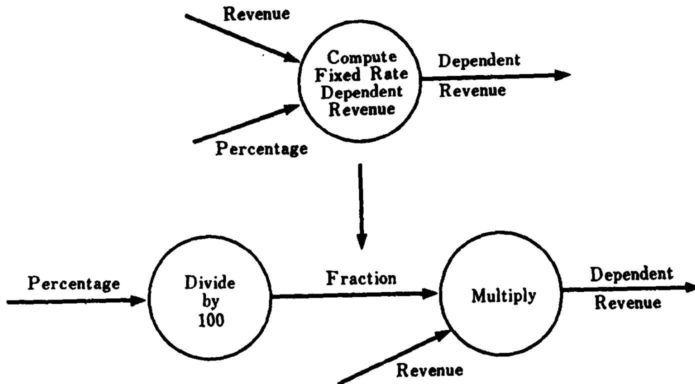


図-2 設計サブファミリの詳細化

が定義されていれば，それにより詳細化が一段進むことになる．定められた制約は詳細化された設計スキーマにも適用される．逆に，詳細化が新たな制約を付け加えることもあり，その場合，その制約は設計の他の部分にも波及し，さらに詳細化を押し進める．このようにして，一つの制約，すなわち設計上の決定事項が設計の詳細化の連鎖を引き起こす．このことは現実の設計上の決定事項の依存関係を良くモデル化している．

IDeA は上記のような性質をもつ設計スキーマにより構成された知識ベースを用いて，ユーザの設計スキーマの選択，特定化を支援し，詳細化と制約の伝播を自動化している．設計の完全性はデータフロー図においてすべての入出力が指定されているか否かによって判断し，無矛盾性は束縛の伝播を自動化することにより保証している．IDeA は現実の設計活動を比較的良くモデル化するとともに，設計の再利用に関する多くの示唆を与えている．設計スキーマはかなり限定された領域で構築されることになろうが，その上で，最小限のコストで特定のユーザ向けアプリケーションの設計が得られるよう工夫されている．

3.2 REMAP

REMAP (REpresentation and MAintenance of Process knowledge)[8] は New York 大学で研究されているシステムであり，設計過程に関する知識を知識ベース化し，要求仕様の変更や類似システムの開発時に活用することを目指している．

REMAP の設計過程に関する知識の基本的な考え方は以下のようである．①設計過程は相互に依存した設計上の決定事項 (design decisions) の連鎖からなる．②決定事項間の関係はアプリケーション固有の規則に基づいている．③システムがプロトタイピングにより漸次に開発される場合は，開発者は，あるサブシステムから得られた経験を他のサブシステムの類似のコンポネントの開発時に，類推により利用している．

REMAP は上記の考察に基づき，①設計上の決定事項間の依存関係の表現と，決定事項の変更時の影響のトレース，②依存関係の一般的基礎となるアプリケーション固有の規則の抽出，③サブシステム中の類似性の検出，といった機能を有する．

設計の表現には IDeA と同様に，構造化分析に用いられるデータフロー図を用いている．データフロー図に関する知識はフレームを用いてシステム内にモデル化されており，設計者はそのインスタンスを定義することにより，設計を定義する．**図-3** にインスタンスの定義例を示す．また，対応するデータフロー図を**図-4** に示す．

インスタンス上の各フレームは一つの設計上の単位 (“設計オブジェクト” と呼んでいる) に対応する．REMAP は一つの設計オブジェクトの出現により，一つの設計上の決定がなされたと解釈する．したがって，設計上の決定事項間の依存関係は，設計オブジェクト間の依存関係と解釈され，この依存関係を表現するために，設計オブジェクトには特別のスロット (“because-of” スロット) が設けられている．図-3 の London-direct-sales-invoices のフレームの because-of スロットは設計オブジェクト London を値としてもっているが，これは「London という設計オブジェクトが存在しなくなれば，London-direct-sales-invoices という設計オブジェクトはその存在根拠を失

```
{identifier : London
 type       : external-entity
 because-of : ()
 inputs     : ()
 outputs    : (London-direct-sales-invoices,
               London-assigned-sales-invoices,
               London-statistical-sales-invoices)}

{identifier : London-direct-sales-invoices,
 type       : dataflow
 because-of : (London)
 part-of    : ()
 medium     : magnetic-tape
 from       : London
 to         : auto-load-and-edit }
```

図-3 データフロー図のインスタンス定義例

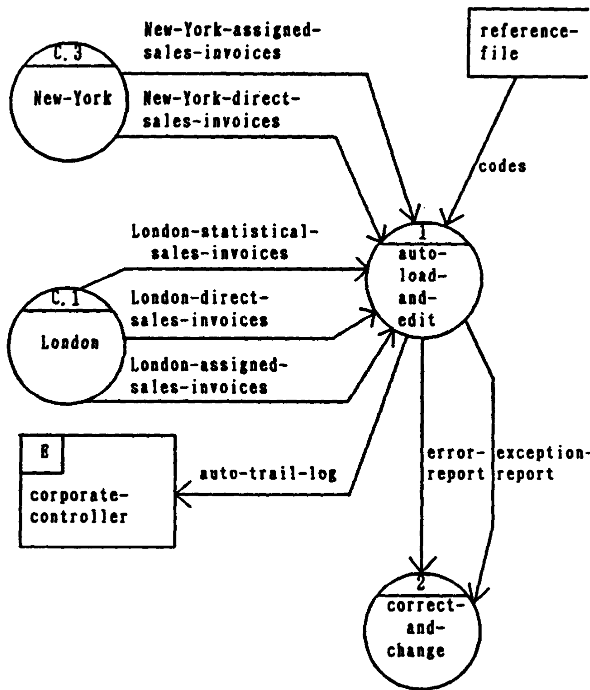


図-4 データフロー図の例

う」と解釈される．したがって，設計変更により London がなくなれば，London-direct-sales-invoices も同時に消去されることになる．REMAP はこのスロットによる設計オブジェクト間のネットワークを把握しておくことにより，設計変更時の影響範囲を把握することができる．

REMAP は次に，このような依存関係の例から，アプリケーション固有の一般化された規則を導くことを目指している．たとえば，図-4 の例における auto-load-and-edit は New-York からの二つのデータフロー(New-York-assigned-sales-invoices, New-York-direct-sales-invoices) によりその存在根拠を保証されているが，実はそれらのデータフローのすべての性質が必要なのではなく，それらのデータフローが"計算機化されている"場合にのみその存在根拠がある．このことを一般化されたアプリケーション固有の規則として書くと以下のようになる．

{dataflow medium : computerized}
⇨auto-load-and-edit.
{dataflow medium : paper}
⇨manual-add-and-edit.

このような一般化された規則を知識としてもつことは設計変更時や類似システムの開発時の支援に大きな力を発揮することが予想される．

前述の一般化された規則は，dataflow をさらに分類し，computerized-invoice, paper-invoice のような下位オブジェクトを設定することにより，

{computerized-invoices}
⇨auto-load-and-edit.
{paper-invoices}
⇨manual-add-and-edit.

と表現することもできる．このように，REMAP は設計オブジェクトに関する知識を，設計オブジェクトの一般化の階層構造と，その階層構造中のノードに対応づけたルールとして記憶している．類推による設計知識の適用は，まず，新しい設計オブジェクトが一般化の階層構造のどの抽象レベルに位置するかを決定することから始め，次にそのレベルで適用可能なルールを新しい設計オブジェクトに適用する．その後，一般化の階層構造をさらに上位にたどりつつ，適用可能なルールがあれば順次適用していく．

このようにして，REMAP は設計過程の知識を動的に獲得し，システムの保守や開発に再利用することを目指している．現在のところ REMAP は①依存関係の一般化規則を導く理論，手法が示されていない，②新しい設計オブジェクトの抽象レベルを決定する具体的方法が示されていない，などの問題はあるが，新しい設計知識を比較的容易に追加し，再利用できる枠組みを提供している．

4. プログラム変換によるプログラミング環境

4.1 Programmer's Apprentice

Programmer's Apprentice[9]は，MIT の Waters, Rich, Shrobe などによって，70 年代半ばから進められているプロジェクトである．これは，熟練プログラマの助手としてソフトウェア作成全般を補助するシステムを実現しようという試みであり，以下のようなアイデアに基づくものである．

(1) アシスタント・アプローチ

完全な自動合成システムを目指すのではなく，熟練

プログラマのアシスタントを実現する．プログラマとアシスタントであるシステムとの会話を実効のあるものにするため，システムに内蔵された共有知識クリイシェ (Cliche) が用いられる．

（2） クリイシェ

クリイシェとは「決まり文句」という意味であるが，ここでは頻繁に用いる標準的なプログラムパターンのことをいう．システムはクリイシェのライブラリをもっており，プログラマの指示に従ってそれを組み合わせることによってプログラムを作成する．

（3） プラン[11]

プログラムやクリイシェを表現するのに，プランという表現形式を用いている．この表現は以下のような利点をもっている．

- 特定のプログラミング言語に依存せず，アルゴリズムやデータ構造を記述できる．
- 複数のプランを組み合わせて，新しいプランを作ることが容易である．
- 意味を厳密に定義する手段を有しており，検証が可能である．
- プラン間にオーバレイという関係を定義することによって，複数の視点からデータ構造やアルゴリズムを記述することができる．

（4） 汎用の演繹システム

プログラミングに関する推論は試行錯誤を含む複雑な推論であり，Programmer's Apprentice はそのような推論を行うシステムを必要とする．これは，McAllester の TMS (Truth Maintenance System) をベースとして研究されている[12]．

現在は以上の(1)〜(4)のアイデアに基づいて，プログラミングを支援する知的エディタ KBEmacs が動作している[10]．KBEmacs のアーキテクチャは図-5のようになっている．

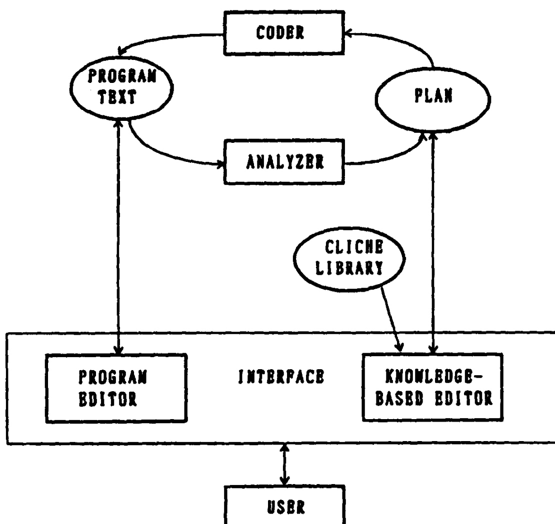


図-5 KBEmacs のアーキテクチャ

このシステムでは，ユーザは，クリイシェに相当する高レベルの語彙を用いた自然言語に近いコマンド，たとえば「表Xはハッシュ・テーブルで実現せよ」という形でエディタに指示を与える．これに対してシステムは，対応するクリイシェをライブラリから取り出し，関連する箇所に埋め込む．このとき，クリイシェ内に記述された制約条件に従ってプログラムは修正される．なお，編集されるプログラムは，内部的にはプランによって表現されているが，ユーザに対するインタフェースのため，また，最終的にプログラムを出力するため，外見上は特定の言語のプログラムの形で表現される．このため，システムはプラン表現を特定のプログラミング言語の記述に変換する機能（コーダ）をもっている．現在のシステムでは，Lisp と Ada のサブセットに対するコーダが実現されている．

また，通常の Emacs の編集コマンドを用いた直接編集も可能にするため，プログラムコードをプラン表現に変換する機能（アナライザ）を有している．これにより，ユーザは高レベルのコマンドと通常のコマンドを使い分け，柔軟に編集を行うことができる．

このシステムは，抽象的なアルゴリズムやデータ構造自体をユーザとシステムの共有知識とみなす点が特徴である．使用するクリイシェの選択など，核となるプログラミング作業はユーザに委ね，システムは組込みにともなう補助的な処理を推論機構によって行う．

4.2 CYPRESS

PSI[13]，CHI[14] といった自動プログラミングの研究で名高い C. Green が設立した Kestel Institute ではその後ライフサイクル全体に渡って知識の活用を目指した KBSA (Knowledge-Based Software Assistant) プロジェクト[15]を開始している．CYPRESS[16] はその主要メンバの一人である D. Smith により開発されたアルゴリズム設計支援システムである．CYPRESS では異なるアルゴリズム・クラスに関する知識を汎用プログラム・スキーマの形で知識ベース中に定義してお

き，それに対して部分スキーマを適用して，スキーマの各構成要素を具体化していくことにより，プログラムを作成する．

たとえば，Divide-and-conquer アルゴリズムのスキーマは以下のように表現される．

F :x＝if
  Primitive :x→DirectlySolve :x □
  ¬ Primitive :x→
    Compose○(G×F)○Decompose :x □
  fi.

これは，「もし入力xがプリミティブならば直接解いて答えとする．そうでなければxを二つに分け，一つ目はGを適用して解き，二つ目は本アルゴリズム自身を適用して解き，その二つの結果を合成して答えを出す．」という意味である．

本スキーマを merge-sort のプログラム作成に適用すると，Primitive :x は「xが空またはただ一つの要素からなるシーケンスか否か」のテストとなり，DirectlySolve : x はx自身を返す関数となる．Decompose はシーケンスを2分割し，最初の要素をシーケンスの前半分，2番目の要素をシーケンスの後半分とする関数となる．G は F 自身として具体化され，Decompose により分割された二つのシーケンスはそれぞれFにより再帰的にソートされることになる．Compose は，その結果ソートされた二つのシーケンスのマージ操作を実行する関数となる．

この同一の divide-and-conquer のスキーマを用いて quick-sort や insertion-sort, selection-sort のプログラムも生成できる[16),17)]．

KBSA のアプローチでは CYPRESS にもみられるように，述語論理を基礎としてプログラミングの諸相に関する知識を形式化し，それに基づいた広汎かつ統合化されたソフトウェア開発環境を構築することを目指している．

5. コンサルテーション/アドバイスによるプログラミング環境

5.1 UC (Unix Consultant)

UC[18),19)] は California 大学 Berkley 校の Wilensky を中心としたプロジェクトである．UNIX の新しいユーザが UNIX のコマンドの使い方を容易に学べることを目的としており，自然言語（英語）インタフェースを提供している．UC とユーザとの対話例を図-6 に掲げる．UC は UNIX システムから直接情報

```
User: How do I print a file on the line
      printer?
UC:   To print a file on the line printer,
      type 'lpr filename'.
  .
  .   (intervening commands and questions)
  .
User: Has the file foo been printed yet?
UC:   The file foo is in the line printer
      queue.
User: How can I cancel it?
UC:   To remove files from the line printer
      queue, type 'lprm username'.
```

図-6 UC の対話例

を受け取り，現在ユーザがどのような状況下におかれているかを認識した上で，最適なコマンドとその使用方法をユーザの抱えている問題に則して回答してくる．

UC はプログラミング環境としてよりも，自然言語処理の一環として研究されている色彩が濃いが，今後のプログラミング環境における知的ヘルプのフレーム・ワークとして興味ある研究である．

5.2 LISP-PAL

LISP-PAL[20),21)] は富士通研究所の上原らにより開発されているシステムである．LISP プログラミングの初心者に，LISP プログラミングのノウハウを伝えることを目的としたコンサルテーション・システムであり，前述の UC に多くの示唆を受けている．元来，教育用に開発されたが，システムは LISP プログラミング環境でエディタから呼び出すことができるようになっており，プログラミング環境における知的アドバイザとして利用することもできる．システムの構造を図-7 に示す．

LISP に関する知識ベースには，LISP の用語や組込み関数に関する知識，基本的なプログラミング・ノウハウがフレーム形式で蓄えられており，それを用いて行う質問応答の戦略もルール形式で知識ベース化されている．

ユーザとシステムとの会話は日本語により行うことができるようになっており，LISP プログラミングに関する，日本語による種々の質問に答える．図-8 に質問応答例を示す．

具体的な知識表現には，SEM 表現という知識表現形式を用いている[21)]．ユーザの問合せはすべて SEM 表現に変換される．質問応答モジュールは受け取った SEM に対して質問応答ルールを適用し，LISP 知識ベースから該当する知識を探し出して，応答を SEM 形式で作り上げる．その後，トランスレー

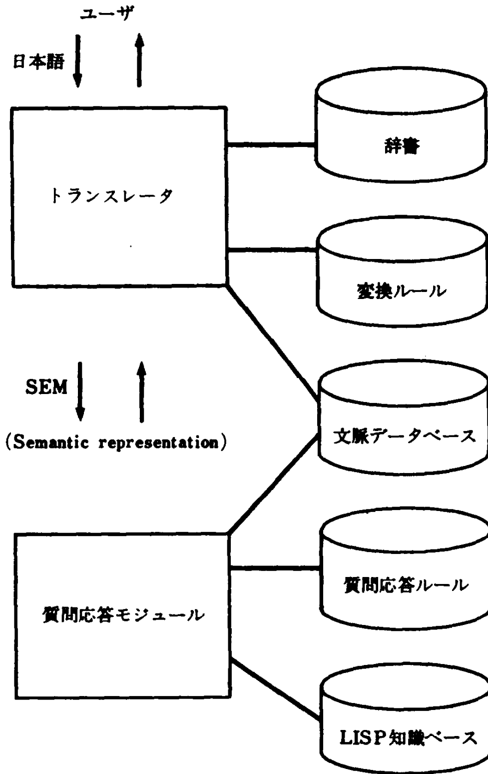


図-7 LISP—PAL のシステム構造

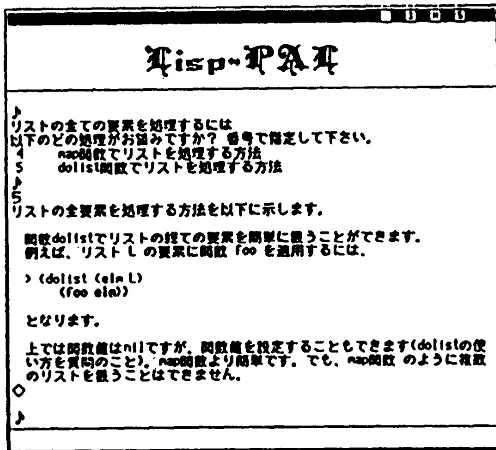


図-8 LISP—PAL の質問応答例

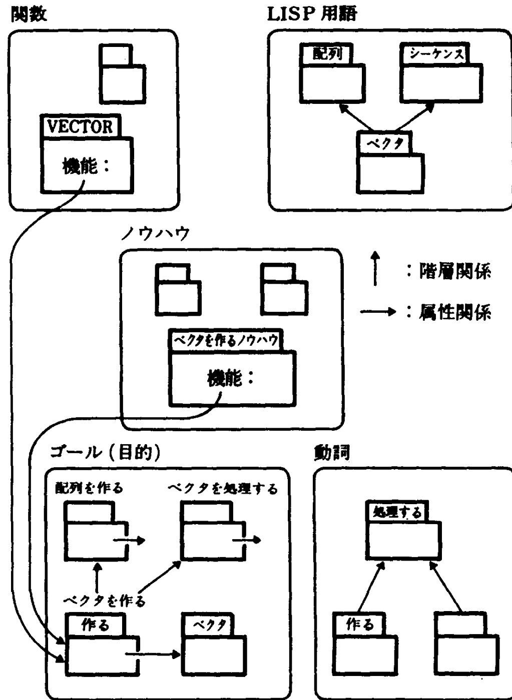


図-9 LISP 知識ベースにおける“機能”のフレーム表現

タが SEM 表現から日本語の応答文を作り，ユーザに提示する．

この SEM 表現は，フレームがネストし，ネットワーク上になっている知識表現を容易に扱えるように工夫したものである．質問応答ルールの記述において，ルール中にフレームのネスト表現をそのまま記述でき，それをパターン・マッチングにより操作できるので，記述をコンパクト化することができる．SEM 表現は UC の知識表現[19]に影響を受けているが，UC では知識ベース構築時にハッシュ・インデックスの指定を行うのに対して，SEM 表現ではその必要がないので，知識ベース構築や知識の追加がより容易になっている．

プログラミングにおけるコンサルテーション・システムという観点からは，「～するには？」といった，プログラミングしたい機能を述べた質問に答えることが重要になってくる．LISP-PAL ではこの種の質問に答えるために，知識ベース中に蓄えられている各関数やノウハウが実現している機能を述べた文を知識検索のためのキー文とし，この文における動詞によって指示される機能に注目する．知識ベース中には質問文に現れると想定される種々の機能が，その機能を表す動詞を中心としたフレームのネットワークとして定義されている．このフレーム・ネットワークは，各関数やノウハウのフレームの機能スロットからリンクされている．一例を図-9 に示す．このように，機能のフ

レームを知識ベース中に独立に定義して知識を整理することによって，前記のような質問に容易に応答することができる．

LISP-PAL は，プログラミングの最中に，エディタで保持されているプログラム・コードの一部をプログラミング・ノウハウとして，日本語を用いて容易に登録できる機能を用意している．これにより，LISP-PAL におけるプログラミング知識ベースは実際のプログラミング環境において容易に成長していくことができる．

現在 LISP-PAL の知識ベースには，Common Lisp に関する用語，全組込み関数（約 600），基本的なプログラミングノウハウ（約 100）が蓄えられている．

6. おわりに

AI 研究はその当初からプログラミングを一つの研究対象領域としてきた．それはプログラミングが人間の高次の頭脳活動の一つであるとの認識に立ったからにほかならない．プログラミングがきわめて属人性の高い作業であり，プログラマの質により，生産性に 10 倍以上の開きがあることは良く知られている．それだけに，熟練プログラマのノウハウを知識ベース化し，自動化，教育，コンサルテーションと，あらゆる局面で活用する試みが精力的に行われている．本稿では特にプログラミング環境に限ってその活動の一部を紹介したが，YES/MVS[22] のように運用・保守段階で着実に成果をあげつつあるものもあり，ソフト工学的立場からより広い観点に立って AI 技術のソフトウェア開発への応用を模索していく必要がある．

参考文献

1) 玉井：ソフトウェア開発への知識工学の応用，情報処理学会誌，Vol. 28, No. 7, pp. 898-905 (1987).
2) Teitelman, W.: A Tour Through Cedar, IEEE Software, Vol. 1, No. 2, pp. 44-73 (1984).
3) 斉藤：ソフトウェア開発環境，情報処理学会誌，Vol. 28, No. 7, pp. 887-897 (1987).
4) 渡辺，小野，堂山：ソフトウェア開発モデルと知識処理適用法，昭和 61 年度電子通信学会全国大会，pp. 8-211-8-212 (1986).
5) Lubars, M. D. and Harandi, M. T.: Knowledge-Based Software Design Using Design Schemas, Proc. International Conference on Software Engineering, pp. 253-262 (1987).
6) Lubars, M. D. and Harandi, M. T.: Intelligent Support for Software Specification and Design, IEEE EXPERT, Vol. 1, No. 4, pp. 33-41 (1986).
7) DeMarco, T.: Structured Analysis and System Specification, p. 366, YOURDON Inc., New York (1978).
8) Dhar, V. and Jarke, M.: Using Teleological Design Knowledge for Large Systems Development and Maintenance, Proc. International Workshop on Expert Systems & Their Applications, pp. 359-388 (1986).
9) Rich, C. and Shrobe, H.: Initial Report on a Lisp Programmer's Apprentice, IEEE Trans. Softw. Eng., Vol. SE-4, pp. 456-467 (1978).
10) Waters, R.: The Programmer's Apprentice: a Session with KBEmacs, IEEE Trans. Softw. Eng., Vol. SE-11, pp. 1296-1320 (1985).
11) Rich, C.: A Formal Representation for Plans in the Programmer's Apprentice, Proc. 7th IJCAI, pp. 1044-1052 (1981).
12) Rich, C.: The Layerd Architecture of a System for Reasoning about Programs, Proc. 9th IJCAI, pp. 540-546 (1985).
13) Green C.: The Design of the PSI Program Synthsis System, Proc. 2nd IJCAI, pp. 4-8 (1976).
14) Green, C. et al.: Results in Knowledge Based Program Synthesis, Proc. 6th IJCAI, pp. 342-344 (1979).
15) Green, C., Luckham, D., Balzer, R., Cheatham, T. and Rixh, C: Report on a Knowledge-Based Software Assistant, pp. 377-428, in Rich, C., Water, R. C. (Eds), Readings in Artifitial Intelligence and Software Engineering, Morgan Kaufmann Publishers Inc., p. 602 (1986).
16) Smith, D. R.: Top-down Synthesis of Divide-and-Conquer Algorithms, Artificial Intelligence, Vol. 27, No. 1, pp. 43-96 (1985).
17) Goldberg, A. T.: Knowledge-based Programming: A Survey of Program Design and Construction Techniques, IEEE Trans. Softw. Eng., Vol. SE-12, No. 7 (1986).
18) Wilensky, R., Arens, Y. and Chin, D.: Talking to UNIX in English: An Overview of UC, Comm. ACM, Vol. 27, No. 6, pp. 574-593 (1984).
19) Chin, D.: A Case Study of Knowledge Representation in UC, Proc. 8th IJCAI, pp. 388-390 (1983).
20) Uehara, S., Nishioka, R. and Ogawa, T.: LISP-PAL: An Approach to Natural Language Consultation in a Programming Environment, Proc. 8th Phoenix Conference on Computers and Communications (1989).
21) Uehara, S., Nishioka, R., Ogawa, T. and Mohri, T.: Implementing Programming Consultation System LISP-PAL: Framework for Handling Frame-Based Knowledge in Object-Oriented Paradigm, Proc. 2nd International Conference on Artificial Intelligence, pp. 409-423 (1986).
22) Griesmer, J. H. et al.: YES/MVS: A Continuous Real Time Expert System, AAAI-84, pp. 130-136 (1984).

（平成元年 2 月 15 日受付）