開放性を持つ複言語の構成手法†

杉本重雄†† 阪口哲男†† 田畑孝一††

プログラミング対象の多様化とソフトウェア技術の進化に伴って多様なパラダイムに基づくプログラミングの重要性が認識されるようになってきた．そのため，1980年代には多様パラダイム指向言語が発展した．多様なパラダイムでのプログラミングを柔軟に行うためには，特定のパラダイムを対象とする環境を用意するだけでは十分ではなく，対象に応じて必要なパラダイムの言語を利用できる開放性を持つ環境を用意することが必要である．本論文では，逐次処理プログラムの範囲において，多様なパラダイムのプログラミング言語を要素とするプログラミング言語複合体 (Programming Language Complex) の構成手法を提案する．プログラミング言語複合体は複数の要素言語からなる統合された環境を利用者に提供するものであるので，本論文ではこれを複言語と呼ぶ．複言語を構成するために要素言語を言語固有の要素と言語間に共通な要素に明確に分離し，それらを階層的にとらえ言語固有要素を上位，言語間共通要素を下位に定義する．こうすることで要素言語処理系を言語間共通要素に基づいて構築することができるので，要素言語間のコミュニケーションに制約を設けることや，言語間コミュニケーションのための ad hoc な記述形式を設けることの必要がなくなる．本論文では，階層の定義とそれに基づく複言語の構成方式を示す．また，LISP, Prolog, Smalltalk-80 に基づく要素言語を持つ複言語システムの例を示す．

1. はじめに

1980年代にはオブジェクト指向型や論理型等新しいプログラミングパラダイムが定着し，さらにそれらに基づく多様パラダイム指向言語が開発された．また，多様なプログラミング言語を用いて記述することは，多様パラダイム指向言語が登場する以前より言語処理系とリンカを組み合わせることで行われてきた．一般に，プログラムの対象となる問題はいくつもの性質の異なる部分問題を含むことが多く，これを単一のパラダイムの言語で記述するよりは，部分問題ごとに適したパラダイムの記述ができる多様パラダイム指向のプログラミング環境が望ましい[3),7)]．こうした観点から多様なパラダイムでのプログラミングを可能にする様々な言語が生み出されてきた．LOOPS はオブジェクト指向パラダイムを基盤としてデータ指向パラダイムやルール指向パラダイムを持つ知識ベース向きシステムである[1)]．TAO はオブジェクト指向記述と論理型記述を取り込んだ LISP である[10)]．CLOS は Common LISP におけるオブジェクト指向記述を定めている[5)]．文献 11) は通信ソフトウェアでの多様パラダイム記述の有効性を示している．われわれの開発したマルチメディア知識ベースシステムでは第1階述語論理とオブジェクト指向記述による知識表現を用いた．そこにおいて，オブジェクト指向記述によってカプセル化した視聴覚情報記述を第1階述語論理に基づいて表現する概念的・記号的知識の中に埋め込むことで，異なるパラダイムの知識のための整合の取れた表現形式を得た[4),8),9)]．

われわれは，多様なパラダイムによるプログラミングのための柔軟な環境を構築するには，手続きや関数等ひとまとまりの機能を記述するプログラムの構成単位ごと（プログラム単位と呼ぶ）に記述言語を自由に選択できる環境を用意することが重要であると考える．そのため，本論文では，逐次処理プログラムの範囲において，多様パラダイム指向のプログラミング言語複合体 (Programming Language Complex) の構成方式を提案する．プログラミング言語複合体は複数のプログラミング言語を要素言語として持ち，要素言語を自由に組み合わせてプログラムを記述できる環境を提供するものである．利用者の観点からは複数の要素言語を一つの言語のように利用できるので，プログラミング言語複合体を複言語と呼ぶ．複言語を定義するために，本論文では，要素言語の機能を階層的にとらえ，要素言語間に共通な機能と要素言語ごとに定まる機能とを明確に分離して定義する方式を提案する．

複言語は，多様なパラダイムによるプログラミングのために次のような特徴を備える．

(1) プログラム単位ごとに記述言語を選ぶことができる．
(2) 異なるパラダイムで記述されたプログラム単位を結合する際に，パラダイムの違いを陽に記述

† A Configuration Method of an Open-Ended Programming Language Complex by SHIGEO SUGIMOTO, TETSUO SAKAGUCHI and KOICHI TABATA (University of Library and Information Science).

†† 図書館情報大学

する必要がない．そのため，あるプログラム単位を異なるパラダイムの言語で記述し直した場合にも，変更が他の部分に影響しない．

（3） プログラマに多様なパラダイムの記述を不必要に強いない．すなわち，複数の言語を知らなくても利用できる．

（4） 新しい要素言語を容易に付加することができる．この性質を新しい言語に対する開放性と呼ぶ．

上の特徴は多様なパラダイムによるプログラミングのために有用であり，かつ，従来のリンカ利用型の多言語系，多様パラダイム指向言語では十分に得られないものである．

2. 開放性を持つ複言語

2.1 プログラミング言語の構成要素

プログラミング言語の階層モデルを定義するために，逐次処理の範囲におけるプログラミング言語の構成要素を以下の観点から分類する．

（1） 構文的要素
文や式，変数等，構文的に定義される要素

（2） 実体的要素
データ実体や変数実体等，実行時に存在し利用される要素

（3） 機構的要素
記憶域の割付や入出力制御といった物理環境とのインタフェースのための要素

（1） 構文的要素

構文的要素には大別して 1. 前処理要素，2. 宣言・定義，および 3. 文・式がある．前処理要素はいわゆるマクロであり，コンパイル時あるいは解釈時に他の要素に展開されて利用される．宣言・定義には手続き，関数，メソッド，節といったひとまとまりの計算単位を表す要素（プログラム単位），変数や定数などの要素，クラスやタイプなどのデータ型を表す要素がある．文・式は順序制御や演算，入出力など制御構造を表す要素である．

（2） 実体的要素

実体的要素には，1. データ要素，2. プログラムコード要素，3. 作業領域要素がある．データ要素には単純変数や配列，クラスインスタンスなど内容を書き換えることができる書換可能要素と，定数要素のように内容を書き換えられない書換不能要素がある．プログラムコード要素にはソースプログラム，中間言語プログラム，オブジェクトプログラムがある．作業領域要素は実行時スタック，ヒープあるいは入出力のためのバッファなどである．

（3） 機構的要素

プログラミング言語を与えられた物理的環境の上に実現するための基本的要素である．すなわち，上記の実体的要素を生成し，プログラムを実行するための基本的要素である．処理系がインタプリタの場合，インタプリタ自身の機能としてこれら機構的要素を含み，コンパイラの場合はこれらを実現するコードを生成する．機構的要素は次のように分類できる．

（a） 入出力のための表現変換制御：外部装置上での表現形式とプログラム実行時の実体的要素との間の相互変換機構．

（b） 記憶域制御：実体的要素を記憶領域上に置くための管理機構．（動的割付機構，静的割付機構，ガーベジコレクションなど）

（c） 参照制御：プログラム実行時に参照する実体的要素を選定し，参照を実現する機構．

異なる言語で記述されたプログラム単位間でコミュニケーションが可能であるためには，言語間で共通理解できる要素がなければならない．プログラム単位間で受け渡しされるデータ実体を言語間で共通に識別できることと言語間で共通理解できるデータ型があることはそのための必要条件である．例えば，異なる言語間でパラメタ・返値を交換できねばならない．また，プログラムの実行は制御点（実行中の文・式，あるいは解釈中の節など）の移動を意味するので，異なる言語のプログラム単位の参照の際に，言語間での制御の移動が可能であることも必要条件である．

プログラム単位の参照には，参照先を決定することと，パラメタ・返値を受け渡しすることが必要である．一般には，プログラム単位の名前のみによって参照先を決定する言語が多い．一方，PL/I の総称的手続きや LISP の総称的関数，オブジェクト指向パラダイムのメソッドのように同一の名前のプログラム単位をパラメタのデータ型等で識別するものがある．こうした総称的（generic）な参照の場合，参照制御を行うためにはプログラム単位の名前のほかパラメタや返値のデータ型等を利用する必要がある．

以上の点をふまえて複言語を考えるために，プログラミング言語の要素を次のように階層的にとらえる．

（1） 言語間共通要素：実体的要素（以下，実体と呼ぶ）の識別機能とデータ型は言語間に共通であ

ると考える．このことは，すべてのデータ型をすべての言語で解釈できねばならないということを意味するものではなく，各々のデータ型がそれに関係する言語間のみで解釈できれば十分である．

（2） 参照機能要素：プログラム単位間の参照とプログラム単位間に共通な変数（大域変数）への参照機能である．プログラム単位間参照のために受け渡しされるデータの実体とデータ型は言語間共通要素として言語間に共通に定義されている必要がある．

（3） 言語固有要素：構文規則は言語ごとに定められるものであり，前節に示した構文的要素は言語ごとに固有である．言語処理系（インタプリタやコンパイラ）は，参照機能要素と言語間共通要素に基づき構文的要素の解釈規則を実現するものである．

2.2 複言語のための階層モデル

本論文で提案する複言語は，前節に示した言語要素の階層に基づく階層構造を持つ．言語固有要素を実現する層は言語ごとに定まる解釈規則を定義するので言語層と呼ぶ．参照機能要素を実現する層はプログラム単位や変数の外部参照など言語間にまたがる参照関係を取り扱うので参照層と呼ぶ．言語間共通要素を実現する層はデータ型の定義や実体の生成規則を定める層であり，計算機の物理環境上にプログラミング言語のための論理的な環境を作り出すのでデータ抽象化層と呼ぶ．すなわち，要素言語の解釈規則を実現する要素言語処理系は言語層に位置づけられ，下位層の機能に基づいて定義される．要素言語処理系はコンパイラであってもインタプリタであってもかまわない．インタプリタとコンパイラの違いは，前者が構文的要素によって構成されるソースプログラムの解釈規則を実体的要素上に実現するのに対し，後者は実体的要素上の操作を実現するオブジェクトコードをソースプログラムから生成する点である．

この階層構造を**図1**に示す．図に示すように，下位の2層の定義があらかじめ与えられれば，その定義に基づいて言語の解釈規則を定義することで任意の言語をシステムに組み入れることができる．また，新しい要素言語を導入するために十分な定義を下位の層が持っていない場合でも，各層に必要な定義を加えることも容易である．この方式では言語にまたがる参照のための共通規則が与えられるので，プログラム単位間の相互参照のための特別な機構を要素言語の組ごとに用意する必要がない．そのため，複言語には新しい要素言語を追加することが容易であり，「開放性」を持つ．以下に，複言語をリンカ利用型多言語系および多様パラダイム指向言語と比較する．

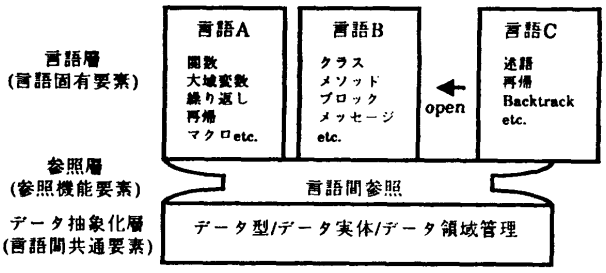


図1 複言語の階層モデル
Fig. 1 Layered model for programming language complex.

従来の言語処理系（コンパイラ，インタプリタ）は構文解析，変数割付，入出力など，上に示した要素を閉じた系として実現している．言語処理系は物理環境が提供するデータ概念（例えば，固定小数点数，浮動小数点数，文字，バイト，ビット）と制御概念（命令セット）を利用して実現される．そのため，いわゆる外部参照を実現するリンカの場合には，物理環境が提供するデータ概念と制御概念に基づいてプログラム同士を結合するしかない（**図2**参照）．

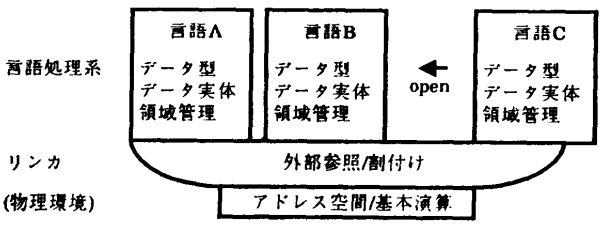


図2 リンカ利用型多言語系のモデル
Fig. 2 Layered model for linker-based languages.

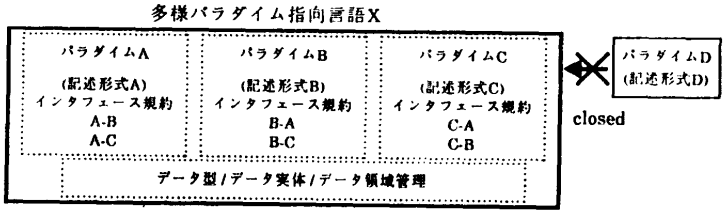


図3 多様パラダイム指向言語のモデル
Fig. 3 Model of multi-paradigm oriented programming language.

これまでに開発された多様パラダイム指向言語の場合はパラダイム間のインタフェース規約がパラダイム相互間で決められるため，新たなパラダイムを導入すると既存のパラダイムの記述形式を変更せざるを得ない．そのため，多様パラダイム指向言語システムの場合は新たな記述形式の導入が困難である．これは多様パラダイム指向言語が開放性を持たない閉じた言語であるということにほかならない（**図3**参照）．

3. 複言語の階層的定義方式

本章では，ネットワークアーキテクチャの階層的な定義にならって，各層が持つサービスとエンティティを示すことで階層を定義する．**図4**は全階層のサービスとエンティティを示している．（図4ではデータ抽象化層（DL），参照層（RL），言語層（LL）に加えて，最下層に物理層（PL），最上層に応用層（AL）があるものとしている．）

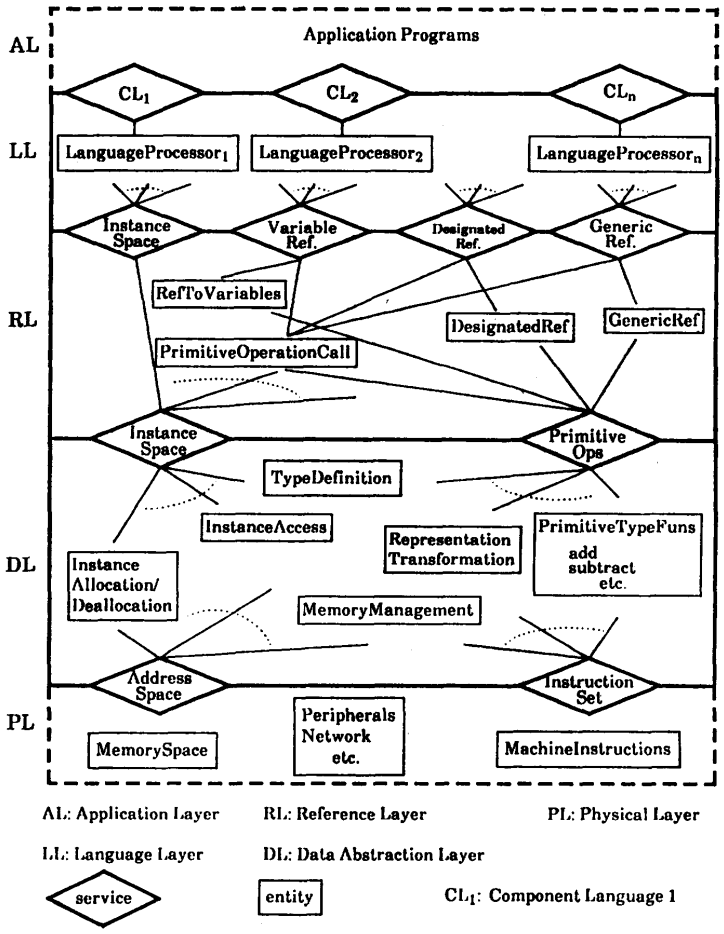


図4 階層構造（サービスとエンティティ）
Fig. 4 Layered configuration (services and entities).

3.1 データ抽象化層

データ抽象化層は物理層の上にデータ型に基づいて実体を定義する層であり，実体とその基本操作を上位層に提供する．**図5**はデータ抽象化層のサービスとエンティティを表している．InstanceSpaceは実体によって構成される空間，すなわち実体の集合である．実体には値（Values），変数（Variables）およびデータ型（DataTypes）がある．また，エンティティには1. 実体生成・削除（InstanceAllocation/Deallocation），2. データ型定義生成（TypeDefinition），3. 実体アクセス（InstanceAccess），4. データ領域管理（MemoryManagement），5.（入出力のための）表現形式変換（RepresentationTransformation），6. 基本演算（PrimitiveTypeFunctions）がある．これらは実体的要素と機構的要素を実現するためのエンティティである．

```
service(DL) = { InstanceSpace, PrimitiveOperations }
    InstanceSpace = 実体の集合
    PrimitiveOperations = 基本操作の集合

    InstanceSpace = ∪( DataTypes, Values, Variables )
        DataTypes = ∪( PrimitiveTypes, UserDefinedTypes )
        PrimitiveTypes = ∪( SimplePrimitiveTypes, AggregatedPrimitiveTypes )
        Variables = ∪( SimpleVariables, AggregatedVariables )

entities(DL) = { InstanceAllocation/Deallocation,
                 TypeDefinition, MemoryManagement,
                 RepresentationTransformation (for Input/Output),
                 InstanceAccess, PrimitiveTypeFunctions }

    ∪(A,B,...)は集合和(union)を表す。
```

図5 データ抽象化層のサービスとエンティティ
Fig. 5 Services and entities in data abstraction layer.

データ型は実体の構造と実体上での操作の集合を定義する．データ型には言語定義に含まれる基本データ型とユーザ定義によるデータ型がある．基本データ型は各々操作の集合を持つ．この集合に含まれる操作を基本操作と呼ぶ．基本操作には算術演算や論理演算な

どデータ型固有の操作とデータ型定義，実体生成などデータ型に共通の操作がある．一般に，プログラミング言語の定義は基本データ型の集合を与える．したがって，複言語では，要素言語の基本データ型の集合の和集合を複言語全体の基本データ型の集合とする．基本データ型以外のデータ型をすべてユーザデータ型と呼ぶ．ユーザデータ型は基本データ型を基にして定義される．ユーザデータ型を定義する操作，ユーザデータ型の実体を生成する操作などは基本操作であるが，ユーザデータ型ごとの操作（例えば，ユーザ定義クラスのメソッド）は基本操作には含まれない．

要素言語がクラスを導入している場合，言語層の要素言語処理系はその継承概念，例えば単一継承や多重継承に基づき定義対象クラスが持つべき論理構造を決定し，データ抽象化層の機能を用いてデータ型を定義する．したがって，データ抽象化層は継承関係に基づくデータ型定義を持つ．さらに，データ抽象化層は入出力に関して，実体の生成，外部表現形式と実体との対応付けを行う．また，データ抽象化層は実体間の等価性・同形性に関する定義を与える．

3.2 参照層

参照層はプログラム単位間の参照と大域変数の参照のための機能を定義する層である．**図6**に参照層のサービスとエンティティを示す．参照層はデータ抽象化層で定義される実体に基づいて定義される．実体は言語層でも利用されるので，参照層は実体の空間（InstanceSpace）を言語層に提供する．プログラム単位への参照機能については参照先を直接指定する場合と，名前とパラメタを与えて最適な参照先の選定を参照層にまかす場合がある．前者を直接参照（designated reference），後者を総称参照（generic reference）と呼ぶ．また，言語層からデータ抽象化層で定義された基本機能を参照するために，参照層は基本演算呼出し機能（PrimitiveOperationCall）を提供する．変数参照（VariableReference）はプログラム単位間で共有される変数（大域変数）への参照機能である．

3.3 言語層

言語層は参照層（およびデータ抽象化層）が提供する機能に基づいて言語の解釈規則を定義する層である．

service(RL) = {InstanceSpace, DesignatedReference, GenericReference, VariableReference}

entities(RL) = {DesignatedReference, GenericReference, RefToVariables, PrimitiveOperationCall}

図6 参照層のサービスとエンティティ
Fig. 6 Services and entities in reference layer.

service(LL) = $\{CL_1, CL_2, \ldots, CL_n\}$
CLiは要素言語 (Component Language)

entities(LL) = $\{LP_1, LP_2, \ldots, LP_n\}$
LPiは要素言語の処理系(Language Processor)

図7 言語層のサービスとエンティティ
Fig. 7 Services and entities in language layer.

言語層では複数の要素言語（component language）の解釈規則を定義する．（**図7参照**．）

4. 複言語システムの構成

本章では，前章で述べた階層的定義方式に基づく複言語システムの構成方法と実現例を示す．

4.1 データ抽象化層

3.1節に述べたようにデータ抽象化層は実体からなる空間（InstanceSpace）とその上に定義された基本操作を上位の層に提供する．この層での実体や操作等の表現形式は実現方法に依存する．例えば，後述のMPSはLISP上にシステムを実現した．そのため，LISPが物理層に対応し，データ抽象化層の定義はLISPのデータ型および基本演算を基礎とし，ユーザデータ型定義機能等を加えて構成した．

4.2 参照層

参照層は，抽象化された参照規約の定義形式を与え，参照操作を実現する．参照層の機能には変数参照，直接参照，総称参照がある．参照は，あるプログラム単位の中で他のプログラム単位あるいは変数への計算開始の要求（関数呼び出し，メッセージ送信，論理式の証明，変数への値の代入など）によって始まり，要求に対する返答が返された時点で終わる．すなわち，言語層は参照を実行するために参照層に計算開始要求を発し，結果を受け取る．変数参照と直接参照は機能的に明らかであるので，ここでは総称参照機能の実現方法について論じる．

実行中のプログラム単位において総称参照を意味する式が現れると，言語層は参照層を介して参照先のプログラム単位にパラメタを渡し，返値を受け取る．この過程には1）プログラム単位の選定，2）パラメタ送受，3）返値送受の操作がある．プログラム単位の選定条件は，プログラム単位の名前が一致することとパラメタの対応関係が満たされていることである．異なる言語で記述されたプログラム単位間の参照のために，参照層では，参照の要求を表す形式とプログラム単位の受理可能性を表す形式を要素言語とは独立に統一的に定義する．この形式を計算開始要求形式（要求形式

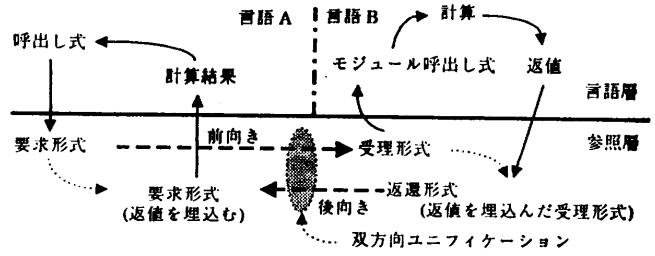


図 8 双方向ユニフィケーション
Fig. 8 Bi-directional unification.

と記す）と計算開始要求受理形式（受理形式と記す）と呼ぶ．要素言語処理系はプログラム単位ごとに受理形式を生成し，参照層に登録する．プログラム単位の参照を意味する式（あるいは文）を実行する際には，その式から要求形式を生成し，参照層に参照操作の実行を依頼する．**図 8** に示すように，ユニフィケーションを参照されるプログラム単位への往復で行い，パラメタと返値の送受を行う．

前向き：要求形式と照合することのできる受理形式を同定し，制御を渡すプログラム単位の選定とパラメタの送受を行う．

後向き：実行結果を受理形式に埋め込んで生成する返還形式を要求形式とユニファイすることで値を返す．

パラメタ送受規則として，call-by-name (**cbname**), call-by-value (**cbval**), call-by-reference (**cbref**), call-by-return (**cbrt**), call-by-value-return (**cbvr**), unification (**unify**) を対象として両形式の定義方法を示す．これらを次の 3 通りに分ける．

(1) **cbval, cbrt, cbvr: cbrt** と **cbval** の機能は **cbvr** の機能に含まれる．

(2) **cbref, cbname: cbref** の場合，実引数のアドレスを渡す．**cbname** の場合，実引数に対応する評価式を渡す．アドレスおよび評価式それぞれを表すデータ型を定義できるので，仮引数が受け取るべきデータ型を定義することができる．したがって，**cbname** と **cbref** の場合はパラメタのデータ型を適切に定義することで，値の送受と同様に扱うことができる．

(3) **unify: unify** の特徴はパラメタ間での値の送受に方向性を持たないことである．これは，要求側パラメタ（実引数）に値を割り当てられていない変数 (uninstantiated variable) が存在すること，受理側パラメタ（仮引数）に定数を書くことができるということを考慮した規則を与えることで **cbvr** と同様に扱うことができる．

(1)(2)(3)より，本方式では，プログラム単位間のパラメタ送受に関して，**cbvr** を基本的な送受方法ととらえる．本方式では先に示したユニフィケーション操作によって **cbvr** を実現する．次に両形式の定義を示す．

受理形式の定義：**図 9** に受理形式の定義を示す．1 個のプログラム単位は少なくとも 1 個の受理形式を持つ．受理形式はプログラム単位自身の名前とパラメタの型を表す記述の並びである．<パラメタ型>は送受方法とデータ型を指定する．<パラメタブロック>は組になっているパラメタを表現する．照合条件は要求形式の中の対応するパラメタ（実引数）との照合のための条件を定義する．たとえば，*V: number は実引数が number 型の値であれば照合可能である．下に LISP と Smalltalk-80 に関する受理形式の例を示す．

LISP の例

関数定義頭部

(defun foo (x y &optional z &rest r)...)

受理形式

(foo *V *V *OV *VL *R)

Smalltalk-80 の例

インスタンスメソッド定義頭部（クラス aClass とする）

aMethod : x paramA : a

受理形式

(aMethod *V : aClass *V V : paramA *V *R)

要求形式の定義：**図 10** に要求形式の定義を示す．要求形式はプログラム単位名とパラメタおよび返値の

<受理形式> ::= (<プログラム単位名><パラメタ型>...)
<プログラム単位名>は関数名、述語名、メソッド名等
<パラメタ型> ::= <UV>|<V>|<R>|<O>|<val>|<oval>|
<OV>|<VL>|<パラメタブロック>
<パラメタブロック> ::= (<パラメタ型>...)|O:(<パラメタ型>...)

照合条件

<UV>	::= *UV:type	値もしくは未定義値を受理する。
<V>	::= *V:type	値を受理する。
<OV>	::= *OV:type	省略可能パラメタ
<VL>	::= *VL:type	可変個数の値を受理する
<val>	::= V:value	特定の値のみ受理する。
<oval>	::= O:value	省略可能な特定値
<R>	::= *R:type	値を返す。
<O>	::= *O:type	要求形式に対応する返値があれば値を返す。

ただし、
:type は値の型を指定するものであり、これが省略された場合は任意の型の値が許されることとする。
未定義値: 値を持たない変数が実引数に与えられた場合、その変数は未定義値 (undefined value型の値 void) を持つと呼ぶ。

図 9 受理形式
Fig. 9 Definition of acceptance form.

<要求形式> ::= (<プログラム名><パラメタ> ... <返値型>)
<パラメタ> ::= <置換可能パラメタ>|<置換不能パラメタ>
<置換可能パラメタ> ::= !value
<置換不能パラメタ> ::= value
<返値型> ::= *o:type|*r:type

ただし、
valueは実引数値を意味する。
<置換可能パラメタ>は返還形式の対応するパラメタと置換える。
*oは呼び出したモジュールが必ずしも値を返さなくても良いことを意味し、一方、*rは必ず値を返すことを意味する。
:typeは返値の型を指定する。

図 10 要求形式の定義
Fig. 10 Definition of requesting form.

記述の並びである．置換可能パラメタは後向きユニフィケーションによって返還形式の中の対応する値で置き換えられるパラメタである．**cbvr** では，実引数値から要求形式中の置換可能パラメタを作り，受理形式中の対応するパラメタを介して目的のプログラム単位の仮引数に値を渡す．返還形式から要求形式の置換可能パラメタに返された値を実引数に代入することで，実引数に値を返す．**cbval** は **cbvr** と同様に置換可能パラメタで実現する．ただし，**cbval** では要求形式に返された値を言語層では利用しない．**unify** では，要求側のパラメタが instantiate されていれば実引数を置換不能パラメタとし，instantiate されていない変数の場合は，置換可能パラメタとする．一方，返値を必須とする手続き，関数やメソッドは *r，必須でない述語やサブルーチンは *o を指定する．

前向きユニフィケーションではパラメタのデータ型は継承関係も含めて照合可能性を定める．前向きユニフィケーション可能な受理形式が複数存在することがある．その場合，受理形式のパラメタのデータ型定義が要求形式に最も適合するものを選択する．パラメタ型の適合性まで等しい場合には言語間に優先順位を定義して選択する．同一言語で記述された複数のプログラム単位が同一の受理形式を持つ場合，プログラム単位選択は言語層にまかせる．一方，要求形式のパラメタが置換不能であり，かつ返還形式の対応する要素と照合できない場合，後向きユニフィケーションは不成立となる．先に述べたパラメタ送受規約の内，後向きユニフィケーション不成立の可能性があるのは **unify** のみである．

4.3 言 語 層

言語層は言語ごとに定義される解釈規則に基づき，下位の層から提供される機能を利用して言語処理系を定義する．原則的に要素言語はプログラム単位間のコミュニケーションのために記述上の特別な規約を持つ必要はない．しかしながら，異なった言語で記述されたプログラム単位同士が関わり合う部分には言語間のギャップを埋めるための記述を陽に行う必要がある場合がある．以下に言語間の相違に関する問題およびその解決方法を示す．

非決定性：Prolog のようにバックトラック機構を持つ言語の場合，外部のプログラム単位が再試行可能であるかどうかが問題となる．外部のプログラム単位に再試行を要求するかどうかはバックトラック概念を持つ言語の定義によるものとする．

省略可能な返値：一般に関数は値を返すが，手続きは値を返さない．また，式によっては関数を呼んでも値を利用しないこともある．省略可能な返値はこうした場合に利用する．例えば，Prolog から他の言語で書かれたプログラム単位を呼ぶ場合，Prolog はゴールの証明が成立したか否かによって真偽を決定するので，必ずしも真理値を返値とする必要はない．また，Prolog の側で関数等からの返値を出力パラメタとして利用することもある．**図 11** に MPS で用いた LISP と Prolog に関する規約，およびそれに基づく参照の例を示す．

受理形式定義

例1 LISP関数foo
(defun foo (x y) < body >)
受理形式 (foo *V *V *R)

例2 Prolog述語p
p(*x, a) :- < condition part >
受理形式 (p *UV V:a *O)

要求形式定義

例3 LISP式 (p 1)
要求形式 (p !1 *r)
参照可能。*rにはaが与えられる。

例4 LISP式 (p 1 'a)
要求形式 (p !1 !a *r)
参照可能。*rには真/偽が与えられる。

例5 LISP式 (p 1 2)
要求形式 (p !1 !2 *r)
前向きユニフィケーション不能。

例6 Prolog式 foo(a b)
要求形式 (foo a b *o)
参照可能。*oにはfooの返値

例7 Prolog式 foo(a b c)
要求形式 (foo a b c *o)
fooの返り値がcであれば参照可能。

例8 Prolog式 foo(*x a)
要求形式 (foo !void a *o) または
(foo some-value a *o)
変数 *x がinstantiate されている場合(即ち some-value)参照可能、
そうでなければ(即ち !void)前向きユニフィケーション不能。

図 11 受理形式と要求形式の例
Fig. 11 Examples of acceptance and requesting forms for ls and pls.

4.4 複言語システム：MPS

4.4.1 概　　要

MPS[6] は LISP, Prolog, Smalltalk-80 を基礎とする三つの要素言語 ls, pls, sts からなる複言語システムである．各要素言語はそれぞれの母言語の小さなサブセットであるが，複言語の実験のために母言語にはない機能を加えた．簡単化のため構文は S-式表現である．前章までに述べた階層的定義手法に基づいて処理系を Common LISP 上に作成した．

(1) ls : ls は Common LISP に基づき，実験のために call-by-value-return 型のパラメタ送受を行う機能を加えた言語である．ls および pls では，自身のデータ型に含まれない sts のクラスインスタンスをアトムと同様に扱う．

(2) pls : pls はカットを含め Prolog の基本的機能を実現している．pls から他言語を呼ぶ場合，論理式の値はそれをゴールとする推論が成功したか否かによって決まるので，前向きユニフィケーション，プログラム単位の実行，後向きユニフィケーションがすべて成功した場合に論理値は真になる．外部参照を表す論理式は決定的な述語として定義されていると解釈する．pls は %-式と呼ぶ関数項を持つ．%-式は図 12 に示す例のように展開して実行する．したがって，ls 関数や sts メソッドへの参照を関数項として表すことができる．この展開操作はルールの読み込み時になされる．

(3) sts : sts は Smalltalk-80 に基づく．sts の大域変数は ls の大域変数と共通であり，MPS システム全体を有効範囲とする．また，sts はインスタンスに名前を付けることができ，名前付きのインスタンスはクラスと同様名前で直接指示することができる．

4.4.2 例　　題

sts で定義したクラス **person** と **group** を図 13 に示す．クラス **person** はインスタンス変数 **firstname** と **lastname**，それらの値を取出すインスタンスメソッド **firstname?** と **lastname?**，および初期化用のインスタンスメソッド **init** とクラスメソッド **new** を持つ．クラス **group** のインスタンスは名前で識別され，**person** クラスのインスタンスのリストを持つインスタンス変数 **mem** とメンバ数を示す変数 **n** を持つ．インスタンスメソッド **add-person** は新しいメンバの追加，**mem?** と **how-many?** はそれぞれメンバのリスト，メンバ数を求める．図 14 の ls 関数 **list-members**

```
(deflogic (p (%fx a) *x) :- (q (%gx *x) *y)(r (%hx (%jx (*x . *y)))))
の%-式を次のように展開して登録する
(deflogic (p *var000 *x) :-
  (fx a *var000)(gx *x *var001)(q *var001 *y)
  (jx (*x . *y) *var002)(hx *var002 *var003)(r *var003))
```

図 12 %-式
Fig. 12 %-expression.

```
(Class person                          ;クラス = person, スーパクラス = "object"
 (ClassMethods                         ;クラスメソッド定義
  (new (ln fn)() (↑ ((super new) init ln fn)))    ;クラスメソッド = new
                                       ;パラメタ = "ln, fn", 一時変数なし
 (InstanceVariables lastname firstname)  ;インスタンス変数の定義
 (InstanceMethods                      ;インスタンスメソッド定義
  (firstname? () () (↑ firstname))     ;インスタンスメソッド firstname?
  (lastname? () () (↑ lastname))       ;インスタンスメソッド lastname?
  (init (ln fn) ()                     ;インスタンスメソッド init
        (lastname ← ln)
        (firstname ← fn))))

(Class group                           ;クラス =group, スーパクラス = "object"
 (ClassMehtods                         ;クラスメソッド定義
  (new (name) () (↑ ((super new name) init))))  ;名前付きインスタンスの生成
 (InstanceVariables mem n)             ;インスタンス変数の定義
 (InstanceMethods                      ;インスタンスメソッド定義
  (add-person (p) ()                   ;インスタンスメソッド add-person
    (mem ← (p cons mem))
    (n ← (n + 1)))
  (member? () () (↑ mem))              ;インスタンスメソッド member?
  (how-many? () () (↑ n))              ;インスタンスメソッド how-many?
  (init () ()                          ;インスタンスメソッド init
    (mem ← ('?list new))
    (n ← 0))))
```

図 13 sts プログラム例
Fig. 13 sts example program.

```
(defunls list-members (group firstname)      ;関数 list-members
    (let ((l (member? group)))               ; first name = firstnameである personを返す
        (check-first-name firstname l)))     ;member? = sts method

(defunls check-first-name (name l)           ;関数 check-first-name
  (if (null l) ()
      (if (eq name (firstname? (car l)))     ;firstname? = sts method
          (cons (car l) (check-first-name name (cdr l)))
          (check-first-name name (cdr l)))))
```

図 14 ls プログラム例
Fig. 14 ls example program.

```
(deflogic (get-members *g *y) :-             ;述語 get-members
          (member? *g (*x . *d)) ;  first name = firstnameである personを返す
          (get-last-names (%firstname? *x)(%member? *g) *y))
                                   ;firstname?, member?, lastname? = sts method
(deflogic (get-last-names *name () ()))
(deflogic (get-last-names *name (*p1 . *p) (*q . *x)) :-
          (lastname? *p1 *q) (firstname? *p1 *name)(get-last-names *name *p *x))
(deflogic (get-last-names *name (*p1 . *p) *x) :- (get-last-names *name *p *x))
```

図 15 pls プログラム例
Fig. 15 pls example program.

```
sts% ('group new 'ulis)                          ;sts式の入力,名前がulisのインスタンスを
ulis                                             ;生成する。返値 ulis
sts% ('ulis add-person ('person new 'smith 'john))  ;sts式の入力,john smithをメンバ
ulis                                             ;に加える。返値 ulis
sts% ('ulis add-person ('person new 'lewis 'john))  ;sts式の入力,john lewisをメンバ
ulis                                             ;に加える。返値 ulis
sts% ('ulis add-person ('person new 'jackson 'dick)) ;sts式の入力,dick jacksonをメンバ
ulis                                             ;に加える。返値 ulis
ls% (mapcar 'lastname? (list-members 'ulis 'john))  ;ls式の入力,firstnameがjohnのメンバ
(lewis smith)                                    ;リストを作る。返値 (lewis smith)
pls% (get-members ulis *y)                       ;pls式の入力,
  *y = (jackson)                                 ;返値 真,変数*yの値は(jackson)
```

図 16 実行例
Fig. 16 Execution examples.

は **group** のメンバの中で与えられた **firstname** を持つもののリストを作る．**図 15** の pls 述語 **get-members** は **group** のメンバの中で先頭（最後に加わったメンバ）と同じ **firstname** を持つメンバのリストを作る．実行例を**図 16** に示す．

5. 検　討

本論文では多様なパラメタ送受規則に対応するため双方向ユニフィケーションによって総称参照機構を実現した．要素言語間のパラメタ送受規則が単純な場合にはより高速の総称参照機構を実現することも可能である．複言語システムの上で作成されたシステム（プロトタイプ）を，実行環境に合わせてコンパイルすることで物理環境に適合した最適化を行うことができると考えている．

本論文で述べた方式は，次の観点から有用なものであると考えている．

(1) プログラマは記述言語とは無関係に既存のプログラム単位を利用することができるので，ソフトウェアのライブラリが構成しやすくなる．

(2) 本方式では，参照のための条件を満たしておれば，プログラム単位を異なる要素言語に書き換えてもそれを参照する側には影響を与えない．研究者や開発者にとっては，ソフトウェアを作ることによってそのソフトウェアの構造を明らかにしてゆくことが目的の場合がある[2]．すなわち，プログラミング言語は単にプログラムを記述する道具であるばかりでなく，思考と試作の道具でもある．そうした場合，プログラム全体を書き換えることなしに，ソフトウェア構成を進化させてゆくに従って部分ごとの適切なプログラミングパラダイムを選択できる．

(3) 計算のモデルとデータのモデルという二つの観点から見た場合，数値や文字といった基本的なデータ型，配列や構造体といった構造を持つデータ型がほとんどの言語に共通であることから，プログラミングパラダイムの違いは主に計算モデルの違いによると考えられる．本論文で述べた階層的定義方式では，計算モデルの定義を言語層で与え，参照層は異なる計算モデルの間での通信手順を与え，データ抽象化層は通信されるデータの構造を定めるととらえることができる．すなわち，データ抽象化層は，物理環境に含まれる様々な要素を抽象化し論理的なデータのモデルを与えているととらえることができる．

6. おわりに

従来，プログラミング言語の定義は言語ごとになされ，そのため言語間のコミュニケーションはリンカの機能に制約されてきた．また，従来の多様パラダイム指向プログラミング言語も特定のパラダイム（あるいは特定の言語）を対象として，ad hoc な形での多様パラダイム記述を可能にしているにすぎない．本論文では，データ実体，データ型，プログラム単位参照など逐次処理プログラミング言語に共通な概念を階層化することで，異なる言語を要素として持つ複言語 (Language Complex) を実現することが可能であることを示した．本論文では逐次処理の範囲でのプログラミングパラダイムだけを対象としたが，並行処理，並列処理，分散処理といったプログラミングパラダイムを考えることは今後の課題である．

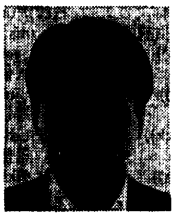
謝辞 カリフォルニア大学サンタバーバラ校・Richard A. Kemmerer 教授と名古屋大学工学部・阿草清滋教授にはこの研究を進めるための有益な環境とコメントをいただいた．末筆ながら感謝の意を表したい．

参考文献

1) Bobrow, D. G.: The LOOPS Manual, Xerox (1983).
2) Gabriel, R. P. (ed.): Draft Report on Requirements for a Common Prototyping System, *SIGPLAN Notices*, Vol. 24, No. 3, pp. 93-165 (1989).
3) Hailpern, B. (ed.): Special Issue on Multiparadigm Languages and Environments, *IEEE Softw.*, pp. 10-77 (Jan. 1986).
4) Sakaguchi, T., Fujita, T., Sugimoto, S. and Tabata, K.: A Multi-Media Knowledge-based

System, *Proc. of COMPSAC '91*, pp. 118-123 (Sept. 1991).
5) Steele, G. L.: *Common LISP: the Language* (2nd ed.), Digital Press (1990).
6) 杉本重雄，阪口哲男，田畑孝一：階層的定義に基づく多様パラダイム指向型プログラミング言語，第41回情報処理学会全国大会論文集，Vol. 5, pp. 50-51 (1990).
7) 田畑孝一，杉本重雄：協同型処理におけるプログラミングパラダイム，情報処理，Vol. 28, No. 3, pp. 286-294 (1987).
8) Tabata, K. and Sugimoto, S.: A Knowledge-based System with Audio-Visual Aids, *Interacting with Computers*, Butterworth & CO., Vol. 1, No. 3, pp. 245-258 (1989).
9) 田畑孝一，杉本重雄：視聴覚情報援用型知識ベースシステム，情報処理学会論文誌，Vol. 30, No. 3, pp. 286-293 (1989).
10) Takeuchi, I.: TAO—A Lisp Dialect with Multiple Paradigms, *J. Inf. Process.*, Vol. 13, No. 3, pp. 318-326 (1990).
11) Zave, P.: A Compositional Approach to Multiparadigm Programming, *IEEE Softw.*, pp. 15-25 (Sept. 1989).

(平成3年5月8日受付)
(平成4年3月12日採録)



杉本 重雄（正会員）

1953年生．1977年京都大学工学部情報工学科卒業．1982年同大学院博士課程修了（情報工学専攻）．京都大学工学博士．1982年京都大学工学部助手．現在，図書館情報大学図書館情報学部助教授．マルチパラダイム言語，協同処理記述言語，マルチメディアシステム，知識情報処理等に興味を持つ．ソフトウェア科学会，人工知能学会，ACM, IEEE-CS 各会員．

阪口 哲男（正会員）

1965年生．1988年図書館情報大学図書館情報学部卒業．1990年同大学院修士課程修了．学術修士．1990年より図書館情報大学図書館情報学部助手．並行処理プログラミング言語，プログラミングパラダイム，マルチメディアシステム，計算機ネットワーク，および分散処理環境などに関心を持つ．

田畑 孝一（正会員）

1941年生．1963年京都大学工学部電気工学科卒業．京都大学工学博士．1973年京都大学助教授．1982年図書館情報大学教授．これまでの主たる研究テーマは音声の多変量解析，コンピュータネットワーク，Concurrent LISP，マルチメディア知識ベースシステム．1986年より日本規格協会 OSI-JIS 調査研究委員会委員長．共著「コンピュータネットワーク技術」（情報処理学会）．編著「OSI—明日へのコンピュータネットワーク」，単著「OSI のおはなし」（いずれも日本規格協会）．