U.D.C. 681.327.26.067

Common Lispオブジェクト指向機能CLOSの実現と利用環境

Implementation of Common Lisp Object System and Its Programming Environments

湯浦克彦* *Katsuhiko Yuura*
長田充弘** *Mitsuhiro Nagata*
古田昌幸*** *Masayuki Furuta*
青島利久* *Toshihisa Aoshima*
高橋　久**** *Hisashi Takahashi*

Lisp処理系LISP E2※)にオブジェクト指向言語CLOS(Common Lisp Object System)を搭載した。オブジェクト指向は,「もの」を中心に対象をありのままに記述することを特長とするプログラミングの方法論である。CLOSは,これに加えて,既存部分を再利用しながらプログラムを容易に作成していくための多重継承(Multiple Inheritance)やメソッド結合(Method Combination)などの機能を提供している。CLOS実用化のためには処理系の高速化が大きな課題である。LISP E2では,総称関数でのメソッドのキャッシングの改良方式を開発し,従来形言語と同レベルの性能を実現した。また,LISP E2では,オブジェクト指向プログラムの解析・検証を視覚的に支援する対話形インタフェースCLOSビュアも開発した。

1 緒　　言

オブジェクト指向は,「もの」を中心とした新しいプログラミングの方法論であり,ユーザーインタフェース,知識処理,データベース,システム記述など幅広い分野でその考え方が取り入れられようとしている[1)]。一方,Lispは,人工知能用言語として優れた機能を持ち,標準仕様を目指したCommon Lisp[2)]の出現によってその応用も広まりつつある[3)]。

CLOS(Common Lisp Object System)は,Common Lisp上のオブジェクト指向システムとして米国を中心に研究者間で検討され,1988年6月,ANSI(米国国家規格協会)に提案されたものである[4)]。CLOSでは,従来の数々のLispオブジェクト指向システムで開発された多くの機能が統合されている[5)]。また,Common Lispとの整合性が高く多種多様の応用プログラムの記述に適している。以上のことから,CLOSは,ユーザーインタフェースツール,設計支援,文書処理,推論実験などの先端的かつ大規模なシステムの開発での利用が期待されている[6)]。

このような背景をもとに,今回CLOSを搭載したLISP E2※)(**図1**)を,大形汎(はん)用計算機HITAC Mシリーズおよび日立ワークステーション2050シリーズ上に開発した。この論文では,まずCLOSを用いたプログラミングの方法を述べる。次に,その方法を実用的なものとするためにLISP E2で開発した高速化技術,および専用開発支援ツールについて述べる。

2 CLOS言語の特徴

2.1 CLOSの概要

CLOS言語の機能と利用形態の概要を**表1**に示す。一般にオブジェクト指向言語では,処理の対象となる「もの」を中心としたわかりやすく信頼性の高いプログラミングが可能である。CLOSでは,さらに以下の3点が強化されている。

(1) 記述効率,拡張性

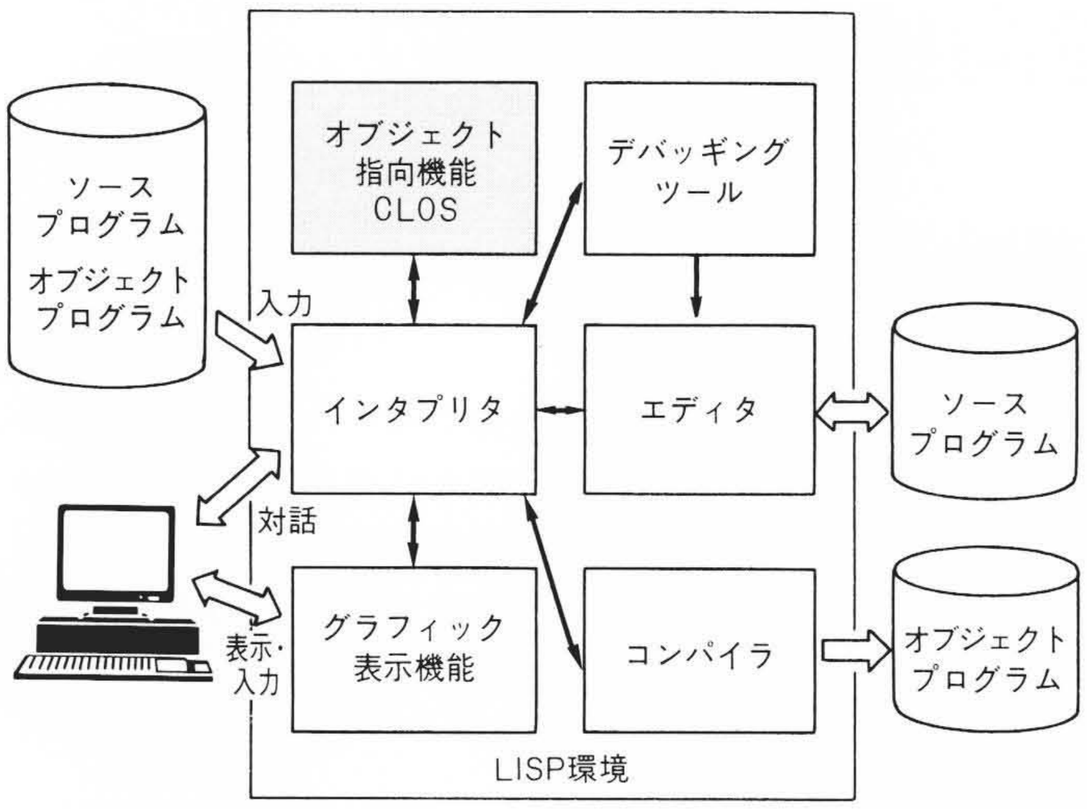
多重継承(Multiple Inheritance)およびメソッド結合(Method Combination)の機能により,既存部のデータ定義や手続き定義を十分に再利用してプログラムを作成することができる。

(2) 幅広い応用のためのプログラミング機能

Common Lispの豊富な関数(約600種),データ形(約40種)

※) HITAC Mシリーズ上のLispの名称は“LISP E2”,日立ワークステーション2050シリーズ上のLispの名称は“LISP”であるが,ここでは統一して“LISP E2”の名称を用いる。

* 日立製作所 中央研究所　** 日立製作所 ソフトウェア工場　*** 日立ソフトウェアエンジニアリング株式会社
**** 日立超LSIエンジニアリング株式会社



注：略語説明 CLOS (Common Lisp Object System)
LISP (List Processor)

図1 LISP E2の構成 LISP E2は，インタプリタを中心に関数，コンパイラ，専用エディタ，デバッガ，CLOSなどで構成する。

表1 CLOSの概要 CLOSは多重継承やメソッド結合などの拡張機能を持ち，特にプログラムの拡張性の面で先進的な言語である。

No.	項　　目	CLOS
1	同時に利用できる従来形の言語	Common Lisp
2	利用形態	対　　話
3	プログラムの変更	逐次可能
4	メソッドの実行制御	総称関数(複数の引き数のクラスに合うメソッドを自動選択)
5	継承*	多重(複数クラスから可能)
6	メソッド結合**	10種以上の結合形式あり
7	メモリ管理	不要メモリを自動回収

注：* 他のクラスからデータや手続きの定義を引き継ぐ機能
** 複数のメソッド(手続き)を継承して組み合わせ実行する機能

をオブジェクト指向のプログラミングのなかで自由に使うことができる。

(3) 対話性，柔軟性

編集したソースプログラムを，対話環境で直接実行させることができる。また，データ形や手続きの定義の追加や変更が容易なため，テスト，デバッグなどの作業を効率的に行うことができる。

2.2 CLOSの機能と利用例

以下，図形処理プログラムを例題として，CLOSの機能について説明する。

2.2.1 オブジェクトと総称関数によるわかりやすいプログラミング

処理対象となる図形の種類(直線，折れ線，箱など)ごとに，その図形が持つべき属性，要素，状態などの変数と，その図形に適用される手続きを定義する(**図2**)。ここで変数をスロット，手続きをメソッドと呼び，両者を一括したものをクラスという。このクラスに基づいて生成される個々のデータがオブジェクト(一般にはインスタンスと呼ばれる。)である。オブジェクト指向では，このように「もの」を記述したクラスを単位として対象世界をありのままに表現していくことができる。

オブジェクト指向プログラムは，オブジェクトへメッセージを送ることで実行される。CLOSでは，メッセージの概念と関数の概念を併せ持つ総称関数(Generic Function)の呼出による。総称関数呼出では，関数名とオブジェクトを与えると，オブジェクトの属するクラスのメソッドが自動的に呼び出される(**図3**)。つまり，対象ごとの手続きの違いやその振り分けを考慮することなく，簡潔に手続きの実行を記述することができる。

2.2.2 多重継承とメソッド結合を用いたプログラムの統合化と記述性の向上

大規模なオブジェクト指向プログラムでは，クラスの数が多数となり，クラスの管理が問題となる。また，同じ内容のスロットやメソッドの定義を，多くのクラスで繰り返して記述量や保守性の面で問題を生ずることがある。

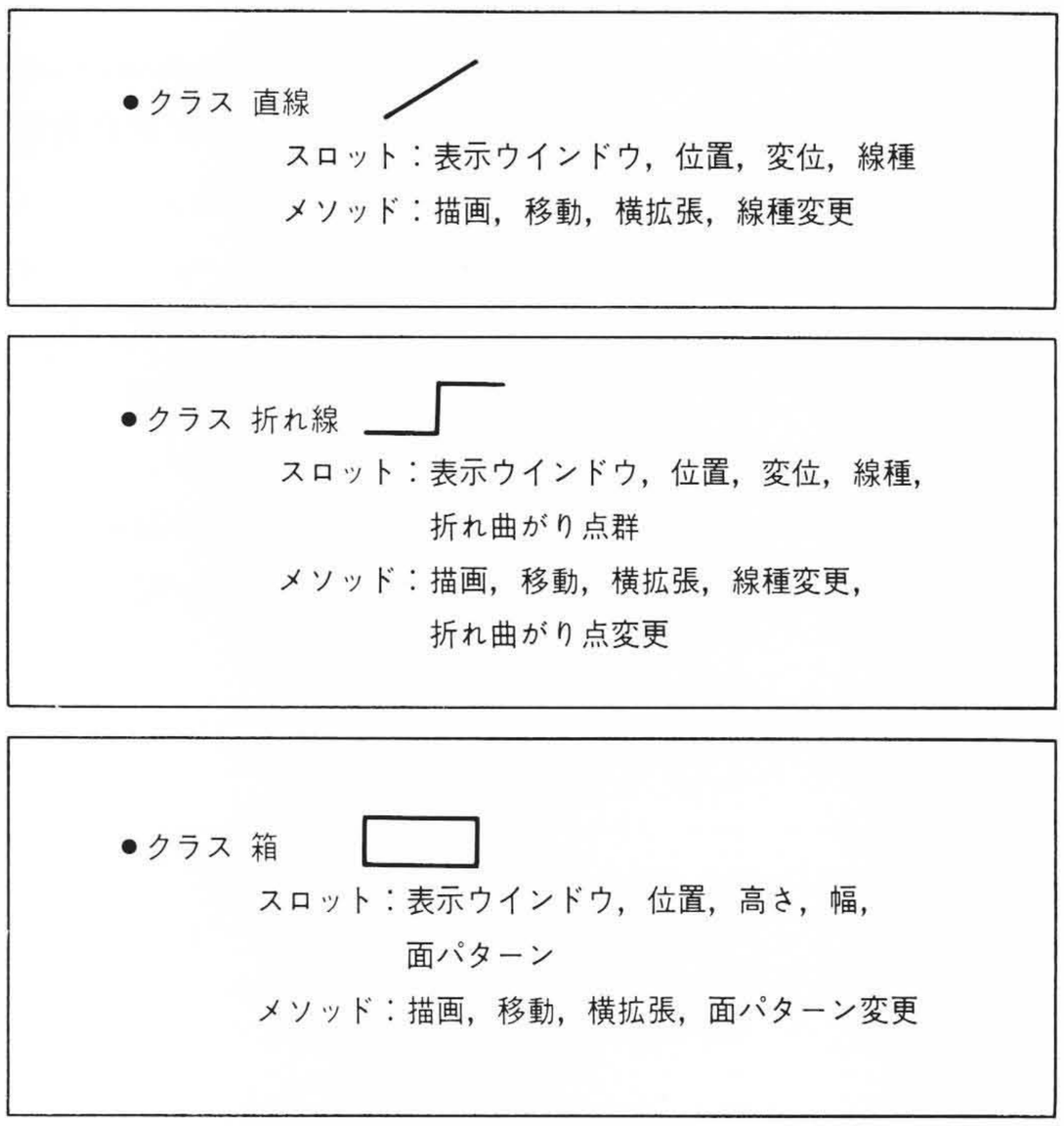


図2 クラス定義の例 クラスとはスロット(変数)の定義とメソッド(手続き)の定義を一括化したものであり，クラスに基づいて個々のオブジェクト(もの)を定義する。

継承(Inheritance)は，上位クラスの定義を引き継いで下位のクラスを定義する機能である。CLOSでは，特に複数のクラスから継承する多重継承が可能であり，これによりクラスを系統化し，定義の共通化を進めることができる(**図4**)。このように，オブジェクト指向機能を利用してのアプリケーションプログラム構築(拡張)では，既存のクラスを継承することによって，従来の手続き(メソッド)を再利用し，異なる処理を新たに追加する，いわゆる差分プログラミングが行われる。

メソッド結合は，メソッドの継承をよりきめ細かな組み合わせで実現しようとするものである。単純な継承では上位クラスの同名メソッドのうち一つをそのまま引き継ぐことしかできない。これに対し，メソッド結合では，上位クラスの同名メソッド群のいくつかを副手続きとして継承し，組み合わせで実行することができる。そこで，継承クラス間で部分的に同一のメソッド群についても継承を適用して統合化を図ることができる(**表2**)。

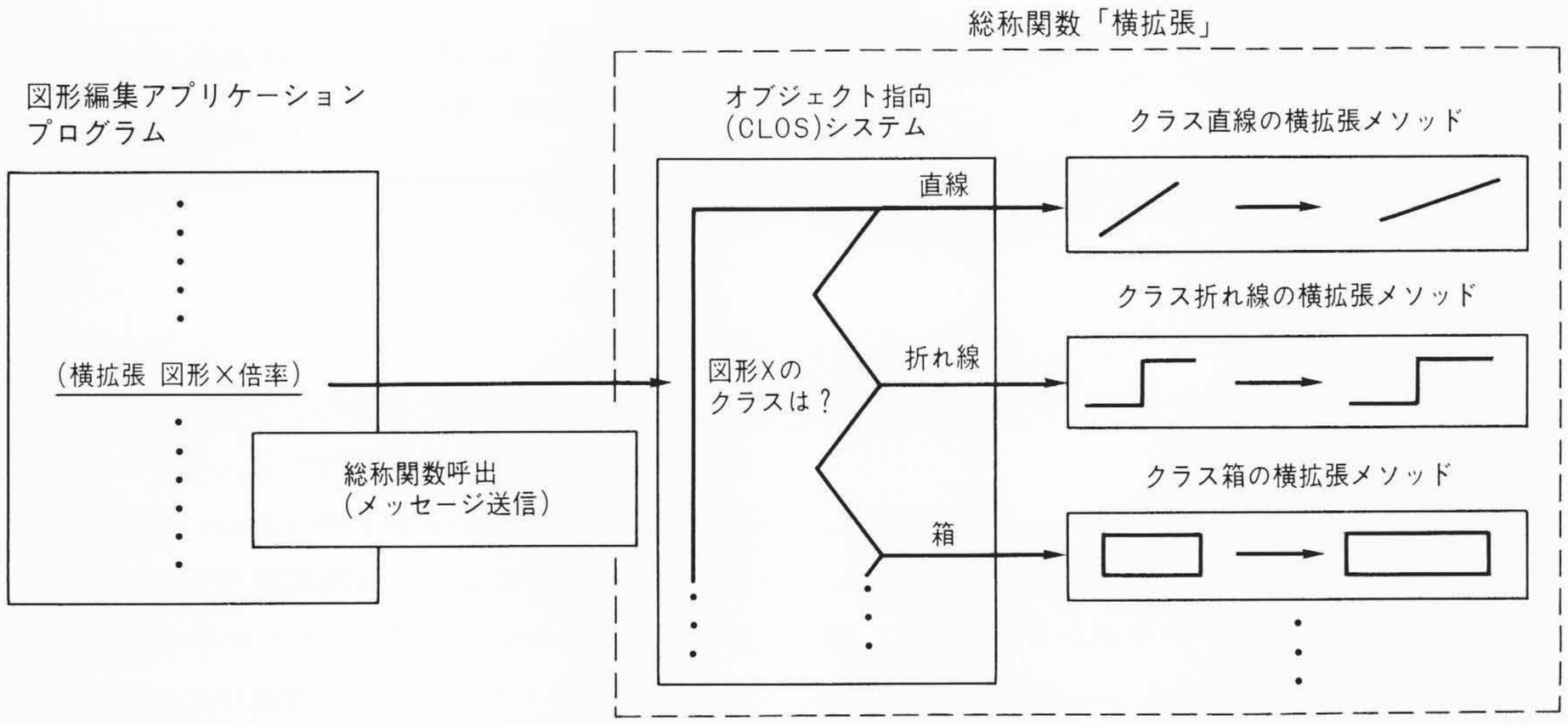


図3　総称関数の利用例　　総称関数では，操作名称(関数名)とその実現(メソッド定義群)を分けて扱うことができる。つまり，ユーザーは操作名称さえ与えればよく，処理対象ごとの手続きの違いやその振り分けを考慮する必要がない。

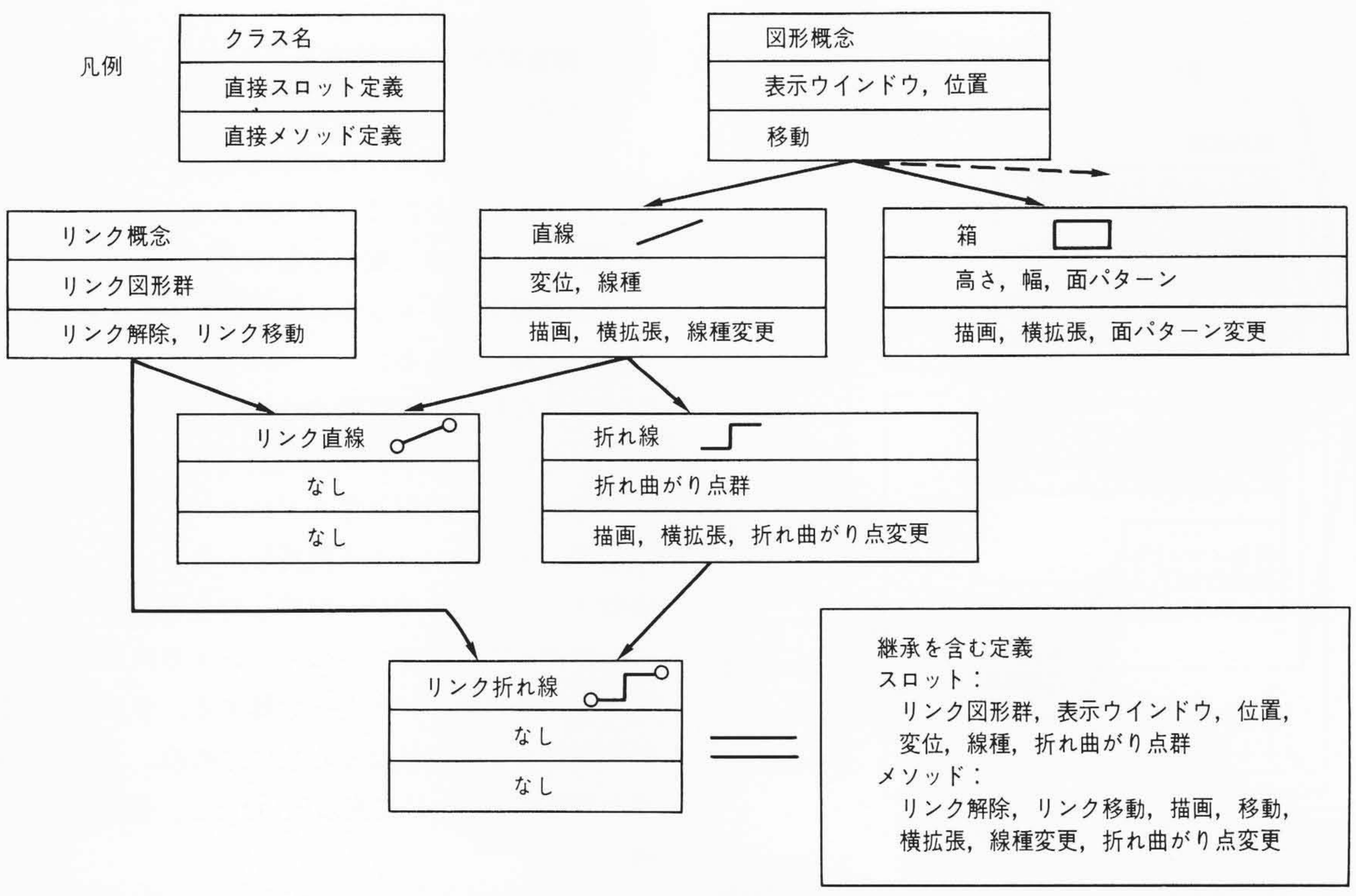


図4　多重継承を用いたクラスの系統化の例　　CLOSでは，継承機能によって上位クラスの定義を引き継いで下位のクラスを定義することができる。特に，複数のクラスから継承することを多重継承と呼ぶ。

表2 メソッド結合の利用例 メソッド結合機能では，自己のクラスと継承クラスの同名メソッド群を結合(Combination)して実行させることができる。これにより，よりコンパクトで拡張性の良いオブジェクト指向プログラミングが可能である。

クラス	「パラメータ設定」メソッドの定義			メソッド結合の結果	
	ID	種別	定義内容	継承クラス順位	メソッド実行順
図形概念	p1	主メソッド	座標指定によって位置設定	——	p1を実行
直線	p2	主メソッド	上位メソッドを呼び出し，次に座標指定によって変位設定	図形概念	p2(p1の呼出を含む。)を実行
折れ線	a1	後置メソッド	上位メソッドを呼び出し，次に座標指定によって変位設定	直線，図形概念	p2(p1の呼出しを含む。)，a1の順に実行
箱	p3	主メソッド	上位メソッドを呼び出し，次に座標指定によって高さと幅を設定	図形概念	p3(p1の呼出を含む。)を実行
リンク概念	p4	主メソッド	リンク図形指定によって位置と変位を設定	——	p4を実行
リンク直線	—	——	な し	リンク概念，直線，図形概念	p4を実行*
リンク折れ線	—	——	な し	リンク概念，折れ線，直線，図形概念	p4，a1の順に実行*

注：* リンク概念クラスが直線クラスに優先するので，主メソッドはp4を実行しp2は実行しない。

3 CLOSの実現法

3.1 CLOSの動作と性能面での課題

CLOSの実行に関する基本制御としては，(1) 総称関数呼出，(2) スロットアクセス，(3) インスタンス生成があげられる。通常，スロットアクセス関数も総称関数として定義されるため，特にその呼出性能が処理系の性能を左右する。

総称関数呼出では，定義された同一名称を持つメソッド群から，与えられた実引き数のクラスに合致するメソッド群の結合形式を生成しそれを実行する。実行されるメソッドの結合形式は，実効メソッドと呼ばれる。総称関数呼出での標準的な動作を**図5**に示す。総称関数呼出は，実効メソッドの計算のために，外見上は同じである通常の関数呼出の100倍以上ものオーバヘッドを要する。関数呼出部は，関数本体の処理に比べるとたいへん小さいが，総称関数では，呼出部の負荷を無視できなくなるわけである。LISP E2では，総称関数呼出を高速化するためにいくつかの方法を採用した。以下にその方法について説明する。

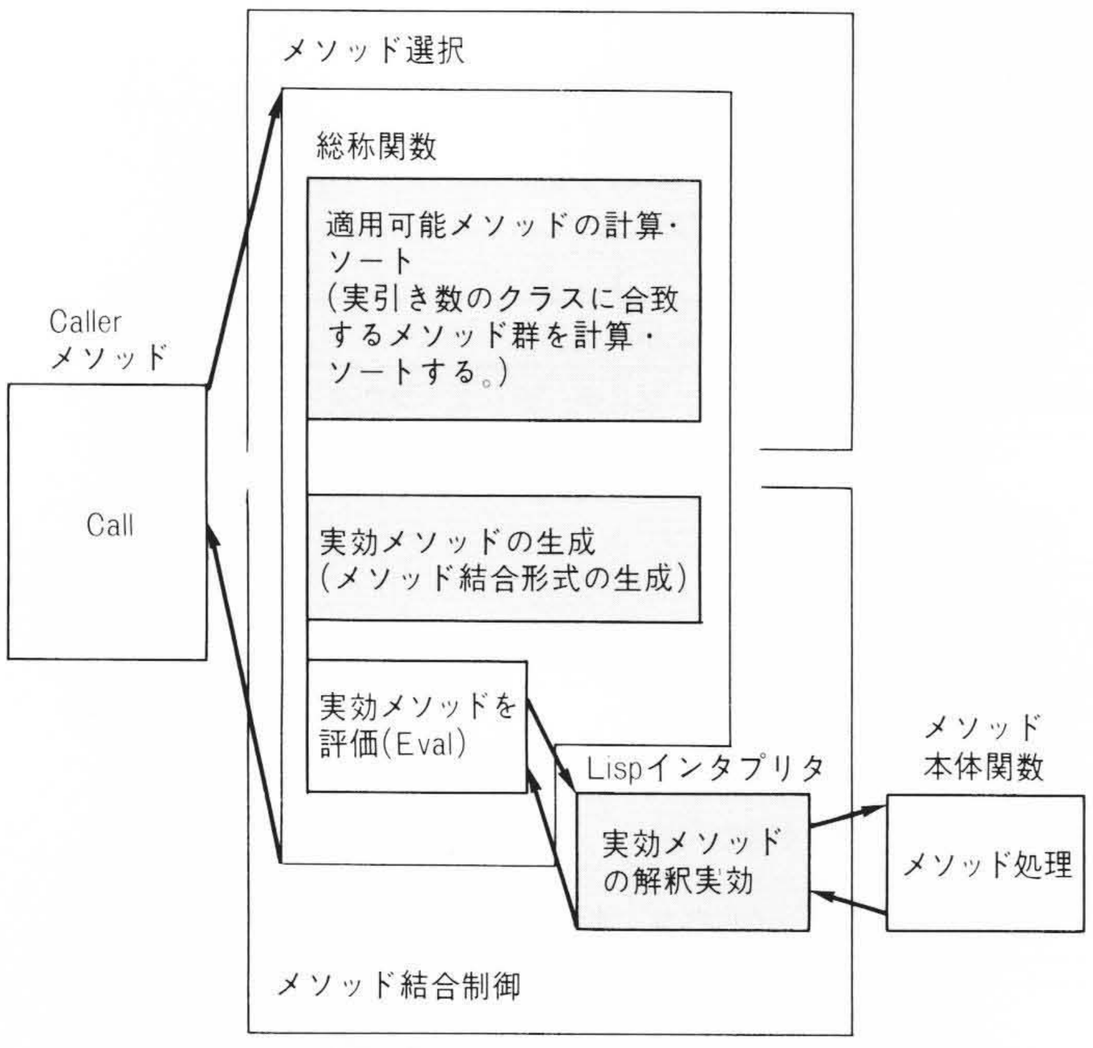


図5 総称関数制御の標準動作 総称関数呼出には，これだけの処理が含まれる。細かい関数呼出は省略しているが，総称関数呼出は，通常の関数呼出の100倍以上ものオーバヘッドを必要とする。

3.2 総称関数呼出の高速化

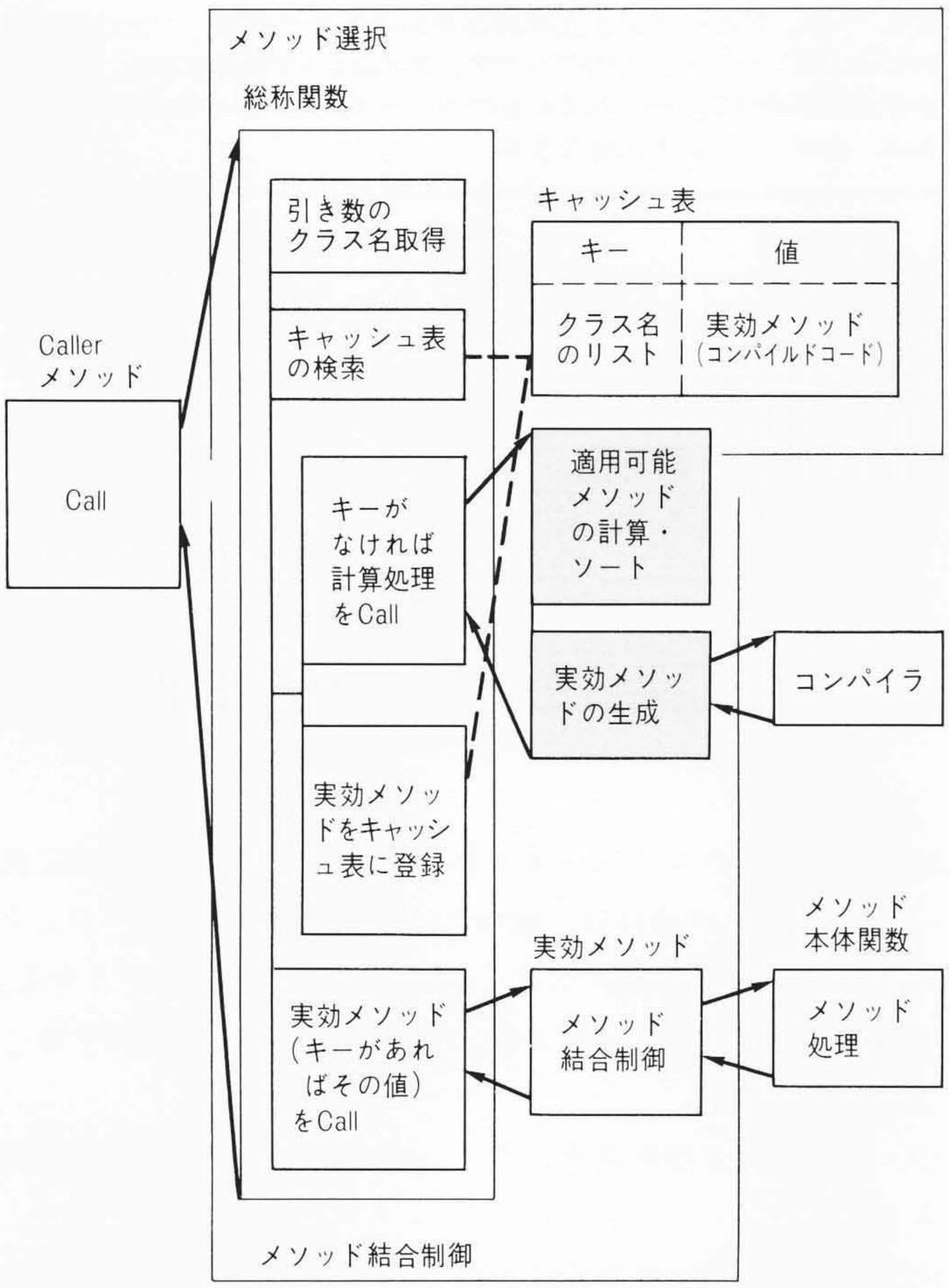
総称関数呼出を高速にするには，実効メソッドの計算に要する時間を短縮しなければならない。この矛盾を解決するために，従来使用されているのがメソッドキャッシュ方式である(**図6**)。これは，実引き数のクラスをキーとして実効メソッドを登録するキャッシュ表を利用するもので，概要は次に述べるとおりである。

(1) 与えられた実引き数のクラスがキャッシュ表に存在すれば，ともに登録されている実効メソッドを用いる。

(2) 与えられた実引き数のクラスがキャッシュ表に存在しなければ，実効メソッドを計算し，キャッシュ表に登録する。実効メソッドとしては，計算したものを用いる。

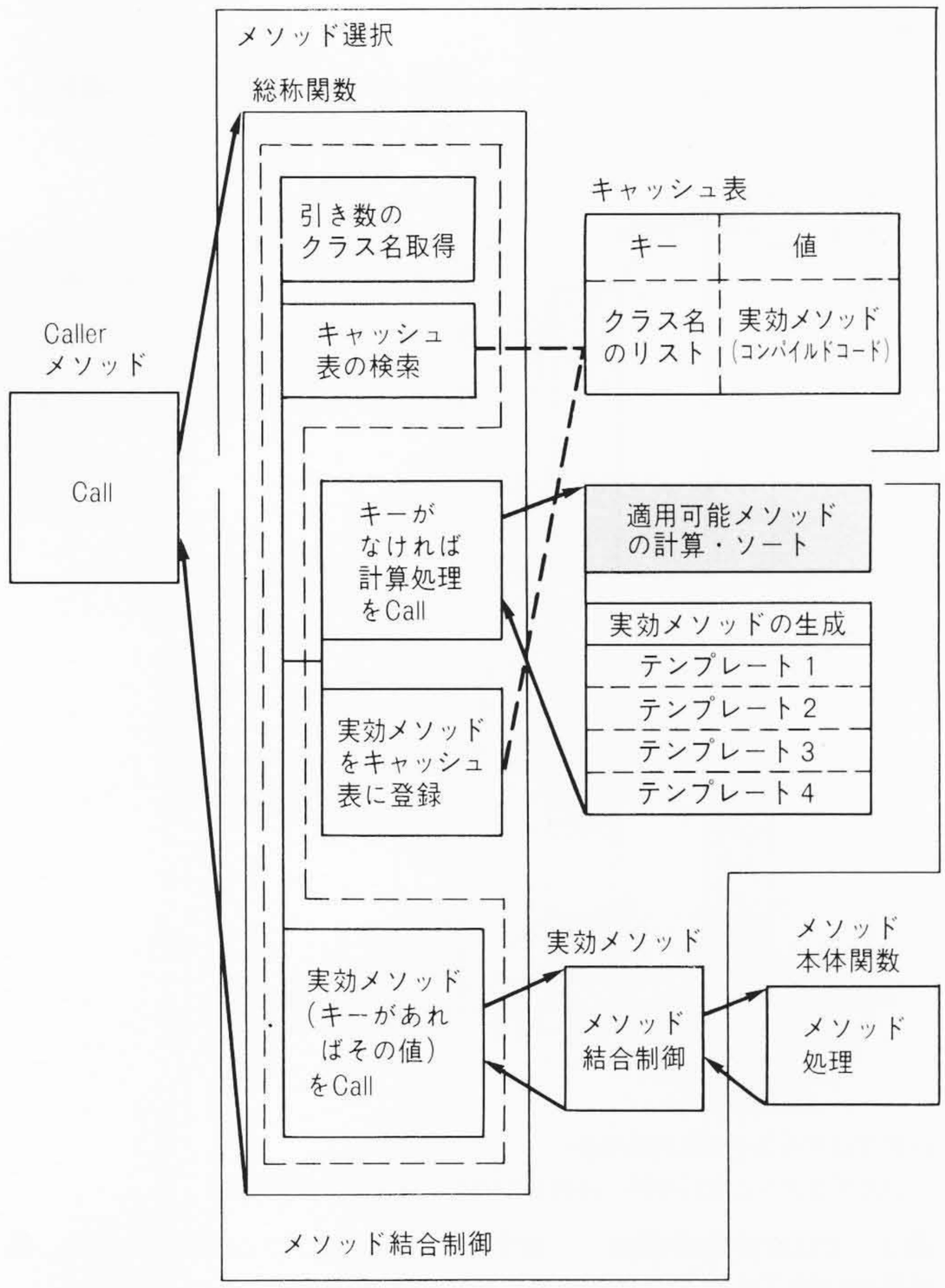
したがって，メソッドキャッシュ方式を用いれば，最初の呼出時には実効メソッドを計算する，すなわち，**図5**相当の標準動作をしなければならないものの，2回目以降は計算過程を省略できるわけである[7]。ただし，依然として次の問題は残る。

(1) 実効メソッドはS式の形をしており，コンパイルしなければ性能はでない。



注：→（関数呼出と復帰），□（処理の負荷が大きい。）

図6　メソッドキャッシュ方式　　1回実行した実効メソッドを，実引き数のクラス名をキーにキャッシュ表に登録することによって，次の呼出時にはその実効メソッド計算の過程を省略できる。ただし，初回の実効メソッド生成時にはコンパイル処理が必要となる。



注：→（関数呼出と復帰），□（処理の負荷が大きい。）

図7　改良形メソッドキャッシュ方式　　実効メソッドの計算にコンパイルドコード化されたテンプレートを用いることにより，初回の実効メソッド生成時のコンパイル処理を不要とした。これにより，全体的に安定した性能が得られる。

(2)　コンパイルすれば(キャッシュ登録時)，コンパイルに要するオーバヘッドを無視できない場合が生じる。

そこで，LISP E2では，このメソッドキャッシュを改良した改良形メソッドキャッシュを開発した(**図7**)。改良形メソッドキャッシュでは，実行時に実効メソッドをコンパイルする代わりに，その共通部分を抽出しコンパイルドコード化したテンプレートを用いるようにした。テンプレートは，さらにメソッド数などのタイプに応じて4種類に分類されており，それぞれ最適なテンプレートが選択される。これにより，キャッシュを適用できない場合の標準的な呼出動作が高速になり，したがって最初の総称関数呼出の負荷も軽減できる。また，CLOSでは，引き数の値自身でメソッド選択を行う特殊メソッドを許しているため，引き数のクラスをキーとするキャッシュが適用できない場合も存在するが，テンプレートの使用はこの場合にも有効である。

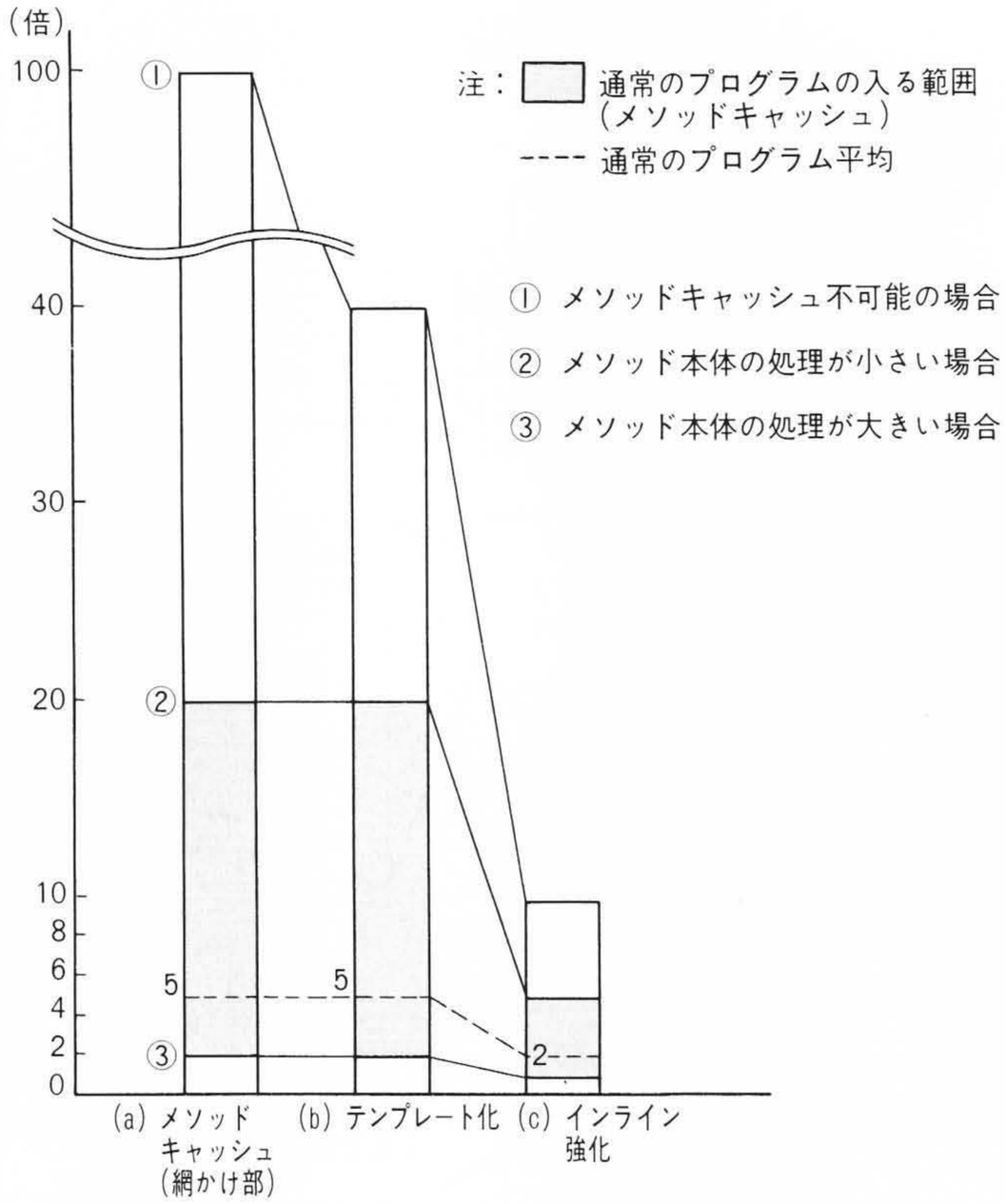
一方，CLOSはLispで記述されているため，CLOS向けに最適化したLispコンパイラを用いて**図7**の鎖線内をすべて機械語イメージでインラインに展開することにより，いっそうの高速化も図った。特に，キャッシュヒット時には最短のルートを通るよう調整を行った。

これらすべての改良の結果として，キャッシュヒット時の実効メソッド実行までのオーバヘッドを，通常の関数呼出の5～10倍程度に抑えることができた。なお，これはメソッドキャッシュを用いなかった場合の$\frac{1}{15}$程度となっている。

3.3　CLOS処理性能評価

Lispのベンチマークとそれを CLOSで書き替えたものの実行時間を比較した。結果を**図8**に示す。メソッドキャッシュによる性能向上は，10～20倍であり，テンプレートの使用は，メソッドキャッシュが不可能な場合に特に有効となっている。最終的には，同図(c)の結果になった。全体として一様な結果が得られなかったのは，メソッド本体の処理が小さい場合，総称関数呼出部の負荷が強調されることによる。

本来Lispで書かれたベンチマークの構造体定義(Defstruct)および関数定義(Defun)を，一様にCLOSのクラス定義(Defclass)およびメソッド定義(Defmethod)に置き換えたため，CLOSには不利な条件であったが，平均でLispの$\frac{1}{2}$～1倍の性



LISPプログラムとの実行時間比
(CLOSプログラム実行時間÷同内容のLISPプログラム実行時間)

図8 CLOS性能評価値 数字は，CLOSプログラムの実行時間を，同内容のLISPプログラム実行時間で割った値である。

能となった。一方，CLOSは，型(クラス)判定の機構を内蔵しているため，プログラムによってはLispよりも速くなるケースがある。実際，オブジェクト指向風のプログラムをLispで書き替えたもので比較したところ，最高で5倍の性能が得られた。

4 CLOS専用支援環境CLOSビュア

クラス継承を利用した差分プログラミングを効率よく行うには，既存のプログラム中で，どのようなクラスやメソッドが定義されているか，またそれらがどのように関連づいているかなどを容易に把握できることが不可欠である。

LISP E2[※1)]では，オブジェクト指向の解析・検証に役立つツールを統合化したCLOSビュアを開発した。

4.1 CLOSビュアの機能と特長

CLOSビュアは，メモリ中のクラス，総称関数およびメソッドを処理対象にして，それぞれの関連を把握することができるツール群で構成している。より具体的には，**表3**に示す機能を持っている。

これら機能に対し，CLOSビュアでは，自由度の高い操作性を目標に次のような特長を実現した。

(1) 各ツールの実行結果をマルチウインドウで表示し，どのウインドウからでも操作(コマンド実行)できる。

表3 ポップアップメニューのコマンドとその機能 CLOSビュアのすべてのツールで，このポップアップメニューが表示される。つまり，操作方法がすべて統一されているので，一つのツールで操作をマスタすれば，他のツールにも応用できる。

コマンド名	機能
編集	クラス・総称関数・メソッドの編集
クラス一覧	クラスの一覧表示
上位クラス階層	上位クラスの階層の表示
下位クラス階層	下位クラスの階層の表示
総称関数一覧	総称関数の一覧表示
メソッド一覧	メソッドの一覧表示
パッケージ一覧	パッケージの一覧表示と処理対象パッケージの設定
キー入力	キー入力ウインドウの表示
全終了	CLOSビュアの終了
終了	対話権を持つCLOSビュアウインドウの終了

(2) 各ツールのコマンド操作を「何をどうする」の形式に統一し，「何を」の操作は，画面上に表示されているオブジェクト(クラス，総称関数，メソッド)をマウスで選び，「どうする」の操作は，ポップアップメニュー中のコマンドを選択することによって行う。

(3) 処理対象を複数選択して，一度に処理(コマンド実行)できる。

(4) 各ツール間に階層関係がなく，どのツールからでも他のツールを呼び出すことができる。

これらの特長は，オブジェクト指向の概念に合致するもので，したがってCLOSビュアは，CLOS自身で記述されている。

4.2 CLOSビュアの適用例

図4の例題のCLOSを用いたプログラム例を**図9**に示す。これは，既存の図形プログラム(clos.lファイル)に図形と図形を線で結ぶリンク機能を追加(exclos.lファイル)しようというものである。

CLOSでは，クラス定義をdefclassマクロで行い，メソッド定義をdefmethodマクロで行う。それぞれの構文の概略は次のとおりである({ }*は，0回以上の繰り返しを表す)。

```
(defclass クラス名 ({継承クラス}*)
  ({クラススロット}*))
(defmethod メソッド名 ({(仮引き数名 適用クラス)}*)
  処理)
```

このプログラムで，例えば「追加しようとしているクラス“リンク折れ線”のインスタンスを引き数にしてどんなメソッドが動き得るか？」を，CLOSビュアを用いて検証してみよう。その例を**図10**に示す。ここでは，「プログラム中の最下位のクラスに属する(適用できる)メソッドを求める」操作を想定し，次の手順を踏んでいる。(1) どのようなクラスが定義されているかをクラス一覧で調べる。(2) 最上位クラスから下位クラス階層を表示して，最下位クラスを求める。(3) 最下位クラス(リンク折れ線)を処理対象(反転表示)にして，総称関数一覧とメ

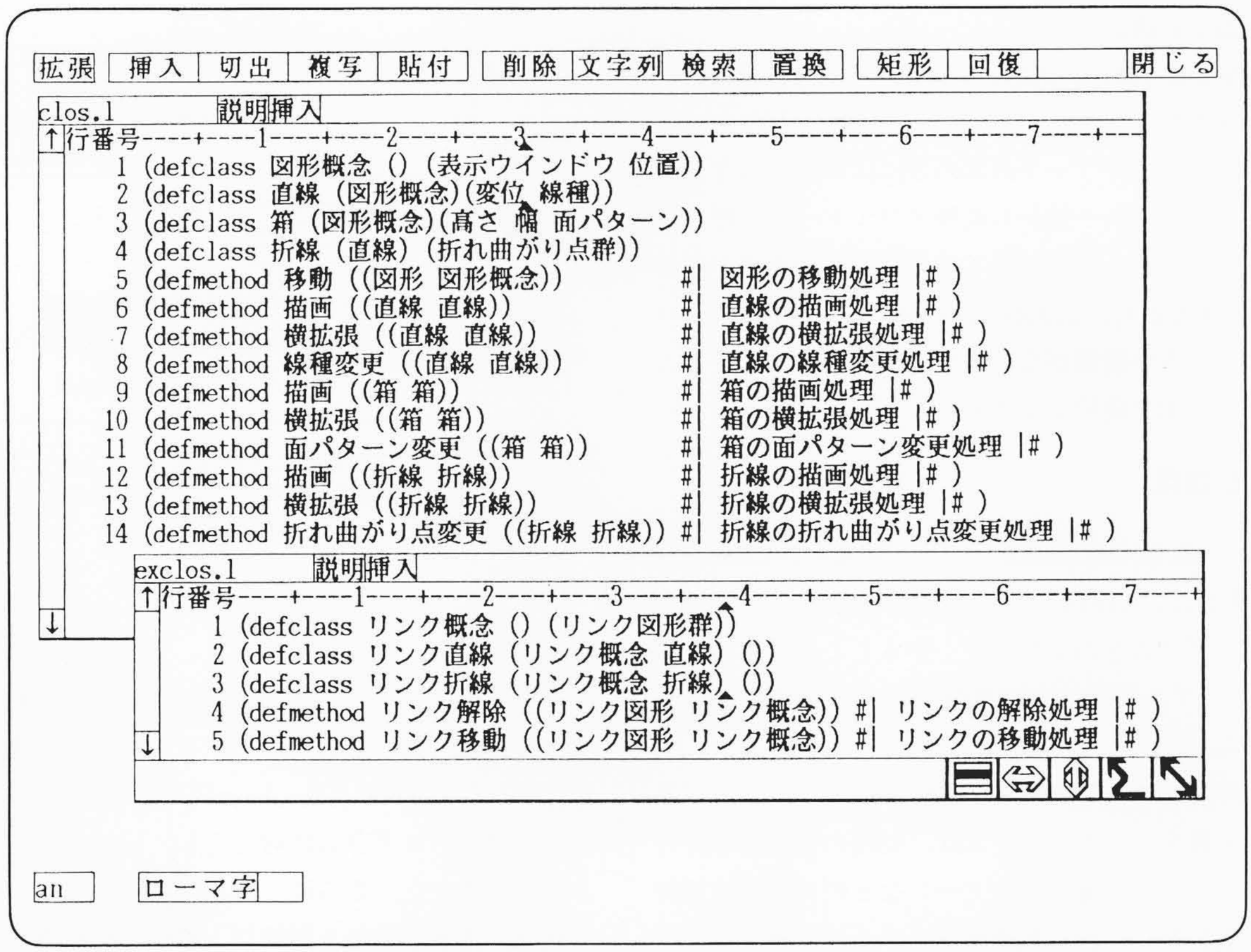


図9　多重継承を用いたクラスとそのメソッドのソースプログラム例　　CLOSでは，クラスとメソッドをそれぞれdefclassマクロとdefmethodマクロで定義する。両者の関連づけは，メソッドの引き数でクラスを指定して行う。

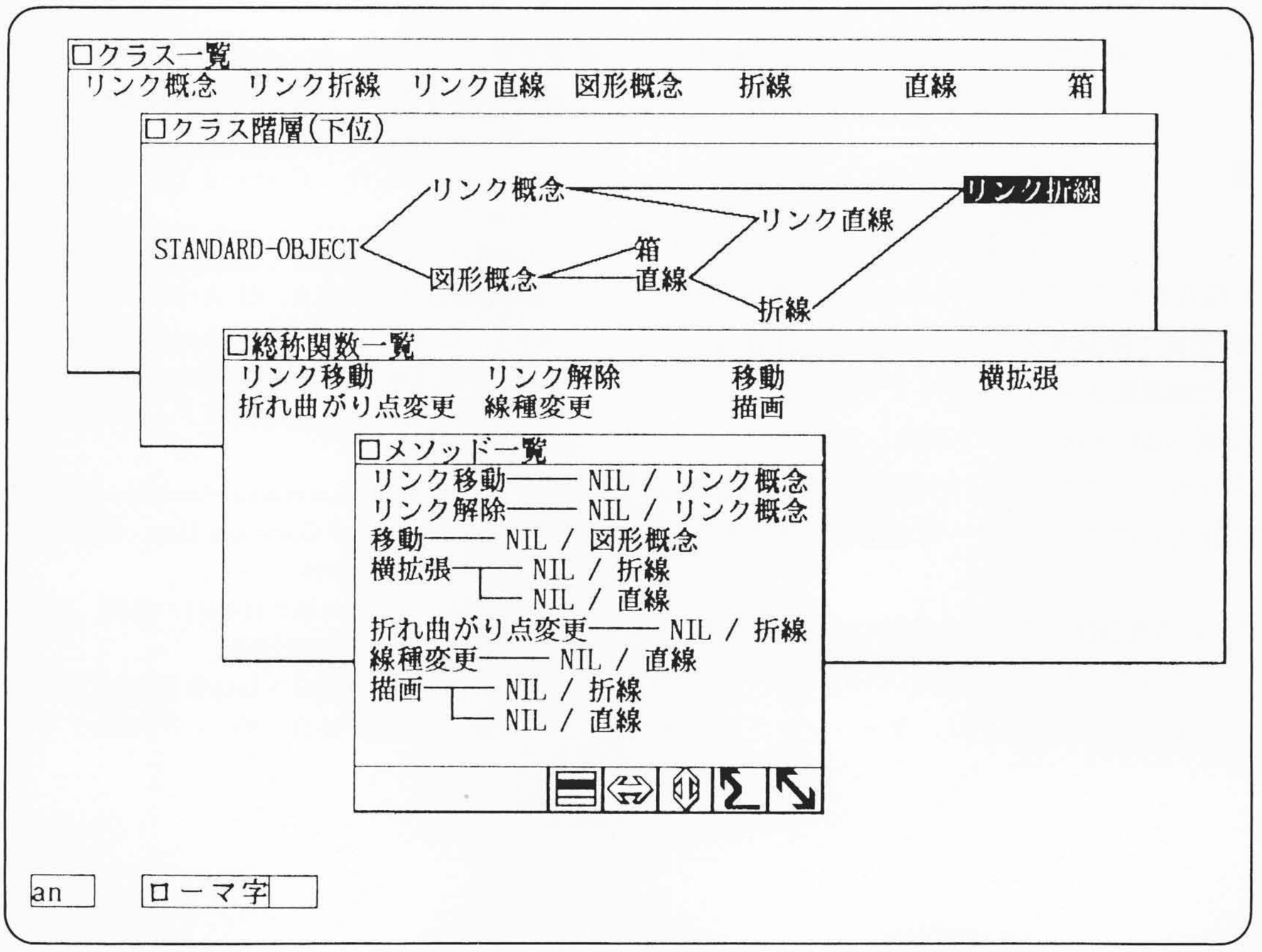


図10　クラス一覧からメソッド一覧を表示する過程　　クラス一覧からクラス階層を表示後，「リンク折れ線」を処理対象にして総称関数一覧とメソッド一覧を表示させた。

ソッド一覧を表示し確認する。

この例でわかるように，CLOSビュアを用いれば，実際にプログラムを動作させることなく，視覚的な検証が可能になる。なお，クラスに属するメソッドの求め方にはいろいろな方法があり，例えば，クラス一覧から直接メソッド一覧を呼び出すこともできる。また，初期起動でメソッド一覧を表示することもできる。すなわち，CLOSビュアのツール群には，ツール間の呼出にいっさい制限がないので，利用者のプログラム理解度に応じて自由な使用ができる。

5 適用例と効果

まず第一に，前述のCLOSビュアの開発に，LISP E2 (CLOS)を利用した。CLOSビュアの各画面要素をクラスとすることで，プログラムをわかりやすく作ることができた。加えて，CLOSビュアを利用してのCLOSビュア自身の機能拡張も円滑に進められた。

さらに，図形処理プログラムを例題としてCLOSによる記述の定量的評価実験を行った。クラスは，13種の基本図形に対して定義したほか，色，線，面パターンなどの共通属性26種に対しても定義した。これら共通属性クラスは，基本図形クラスに多重継承され平均約3倍に利用することができた。また，同じ例題を従来のCommon Lispの機能だけで記述した場合と比較すると，記述量は約$\frac{1}{2}$となった(**図11**)。削減の内訳を解析すると，多重継承およびメソッド結合による効果がほとんどを占めていた。なお，Common Lispの記述性については，例題20種をC言語と書き比べた場合に$\frac{1}{1.0}$～$\frac{1}{16.1}$，平均約$\frac{1}{3}$の記述量で済んだという報告がなされている[8)]。

6 結　　言

CLOSでは，オブジェクト指向の基本的な概念に加えて，多重継承やメソッド結合などプログラムの再利用機能が強化されており，記述量を大きく削減することができる。LISP E2では，さらにクラス継承図をベースとしてクラスやメソッドの定義を対話的に調べるCLOSビュアを開発した。これにより，再利用の対象となるクラスやメソッドの検索や再利用結果の確認を効率的に行えるので，いっそう再利用を促進することができる。

CLOSプログラムの実行は，その中心機構である総称関数の制御にキャッシングのアイディアを適用して最適化できる余地が大きい。この論文で述べた方式は，キャッシングの対象

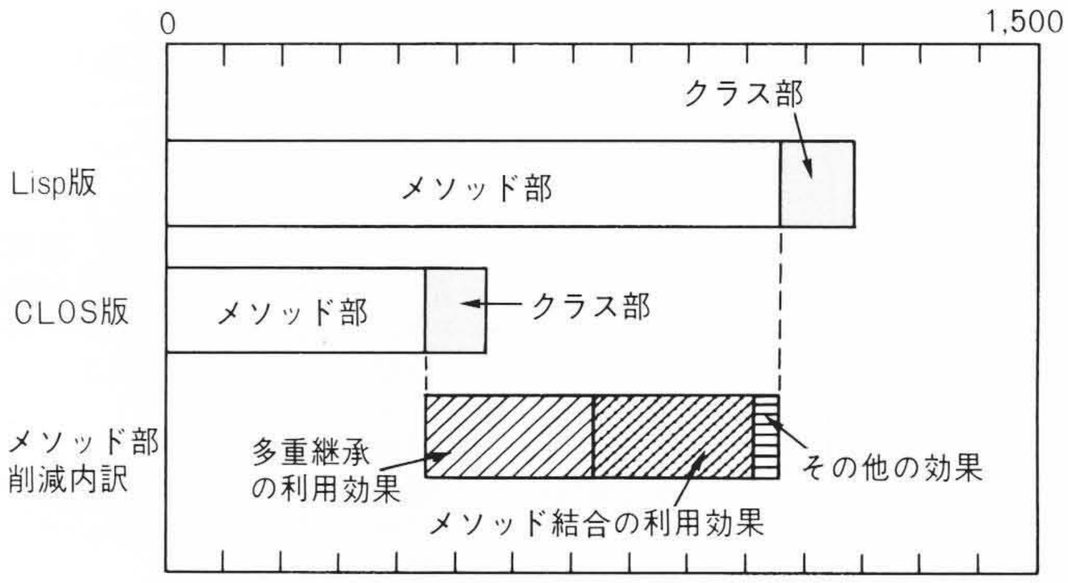


図11　例題によるLispとの記述量比較(単位ステップ)　　CLOSでは，多重継承とメソッド結合の利用によって，メソッド(手続き)の記述量が半減している。

を単一のメソッドからメソッド結合結果に拡張し，かつキャッシュ初回登録に必要なメソッド結合計算の速度を改善するものである。これにより，Lisp記述を単にCLOS記述に書き替えただけのプログラムに対しても，Lispの$\frac{1}{2}$程度の性能を得ることができた。さらに，これをオブジェクト指向風に記述すれば，従来形言語と同等以上の性能を期待できる。

CLOSのプログラミングでは，CLOSによるクラスの財産が増えるほど再利用が生かされ生産性が向上する。より早く，より広い範囲のユーザーに実用性の高いCLOSシステムを提供し，社会に多くのCLOSの財産が形成されるよう努力していきたい。

参考文献

1) 情報処理学会：大特集オブジェクト指向プログラミング，情報処理，Vol.29，No.4，289～410(1988)
2) G. L. Steele Jr.：Common Lisp the Language, Digital Press(1984)
3) 日本電子工業振興協会：Lisp技術動向に関する調査報告書，日本電子工業振興協会，63-A-279(1988)
4) D. G. Bobrow, et al.：Common Lisp Object System Specification, Draft X3J13 Document 88-002R(1988)
5) 井田，外：Common Lispオブジェクトシステム，bit別冊，共立出版(1989)
6) 井田，外：A Requirement Analysis for Portable Window System on Top of Common Lisp，情報処理学会第40回全国大会，919～920(1990)
7) 湯浦，外：CLOS準拠のHiOBJの概要，情報処理学会第39回全国大会，1348～1349(1989)
8) 日本電子工業振興協会：Lisp技術動向に関する調査報告書，日本電子工業振興協会，89-パ-7(1989)