U.D.C. 〔800.92:519.685〕:〔681.32.06:159.95〕

知識処理用言語HiLISPの高速化方式

High Performance List Processor System

武市宣之* *Nobuyuki Takeichi*
安村通晃* *Michiaki Yasumura*
湯浦克彦* *Katsuhiko Yuura*
森田和夫** *Kazuo Morita*

知識処理用言語Lispは方言が多く，仕様の統一が遅れていたが，近年，国際標準化を指向したCommon Lispが発表され，統一が進みつつある。それに伴い，Lispの利用の拡大が予想され，実用性の高い高性能の処理系が求められている。本論文では，HITAC Mシリーズ上に開発したCommon Lisp準拠のHiLISPシステムの高速化方式について述べる。高速化方式として，プログラム変換法，データ形判定・推定法などを開発して，オブジェクトコードの性能に関して，情報処理学会で提案されたLispコンテストプログラムで，従来最高速の処理系の約1.6倍の性能が得られた。現在，本システムは，日立製作所研究所でCAD，パターン認識，自然言語処理などの研究に広く使用されている。

1 緒　　言

Lisp（List Processor）は，1960年，当時米国MIT（Massachusetts Institute of Technology）のJ. McCarthyによって考案されたリスト処理のためのプログラミング言語[1]である。以後，Lisp 1.5[2]，Lispの二大潮流となったMac Lisp，Inter Lispなど多数の処理系が開発されている。そして，数式処理，自然言語処理，パターン認識など，いわゆる人工知能と呼ばれる分野で広く使用されてきた。

しかし，Lispは関数呼出しとリスト処理を基本とすることから，実行効率が低いこと，メモリ使用量が多いこと，言語仕様の不統一による処理系間での互換性の不備，更に日本語処理がないことなどが問題になっていた[3]。一方，人工知能分野への関心の高まりとともに，研究者が性能やメモリ量などのハードウェアの制約に煩わされることなく使用できる実用性の高いLispシステムが求められていた。このような状況に対処すべく研究所での使用を目的として，(1) 高速性，(2) エディタ・デバッガなどの支援環境の充実，(3) 日本語サポートを重視したHiLISP（High performance List Processor）システムを開発した。開発したシステムの構成を**図1**に示す。言語仕様は，国際標準化を指向したCommon Lisp[4]に準拠し，高速でかつ多数が使用できるために，はん（汎）用計算機HITAC Mシリーズ上で実現した。本論文ではHiLISPインタプリタとコンパイラで採用した高速化方式について述べる[5),6)]。

2 HiLISPシステムの高速化における留意点

ソースプログラムを解釈して実行するインタプリタと，オブジェクトコードを生成するコンパイラの実現で，次に示す事項に留意して高速化を図った。

(1) Common Lisp仕様の実現
　(a) 600を超える関数，40を超えるデータ形など，豊富な言語機能の効率の良い処理方式
　(b) 1個の関数で，複数のデータ形を処理するはん用関数（Generic Function）の高速化
　(c) インタプリタでの計算結果と，コンパイルされたオブジェクトコードの結果を完全に一致させる静的スコープの変数管理とその高速化
(2) 十分なメモリ空間のサポート
　知識処理での大規模な実験を容易にするために，2Gバイトのアドレス空間のサポート

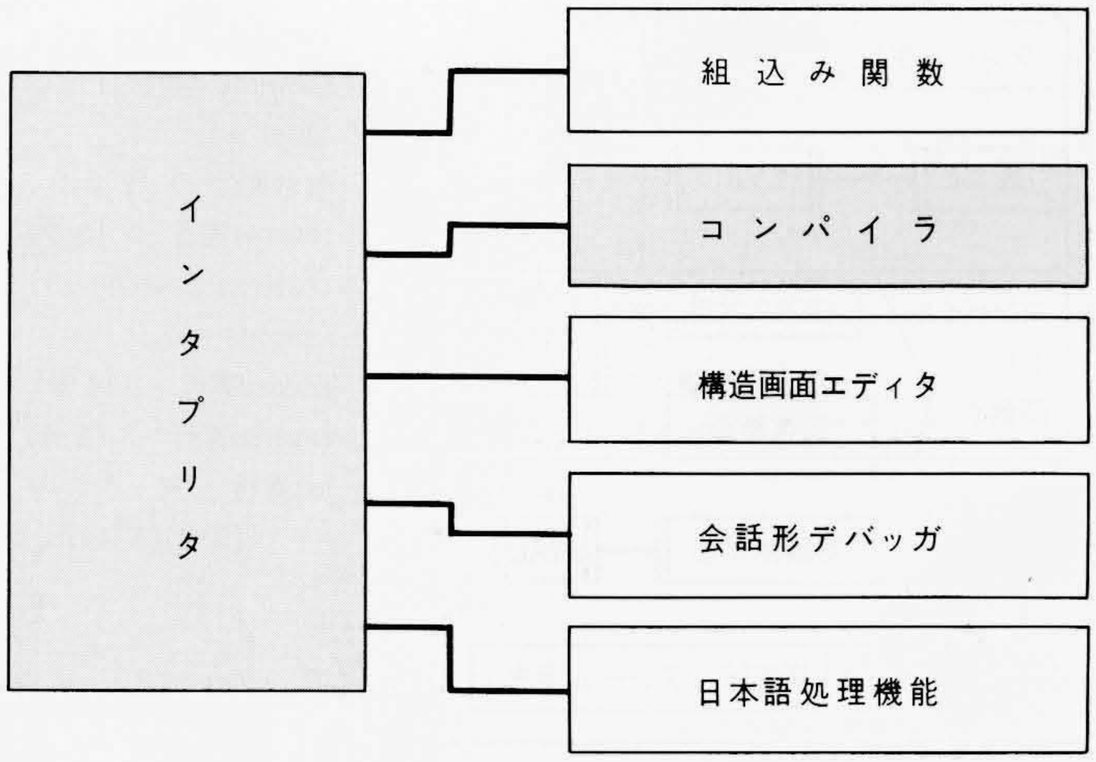


図1　Hi LISPシステムの構成　　Hi LISPシステムは，インタプリタ及びそれが制御する組込み関数，コンパイラ，構造画面エディタ，会話形デバッガ及び日本語処理機能から構成される。

* 日立製作所中央研究所　** 日立製作所ソフトウェア工場

(3) L-codeによるプログラムの移植性の確保

マシン非依存の高速の専用システム記述言語L-codeの開発とそれによるシステム記述

3 HiLispインタプリタの高速化

3.1 インタプリタの動作と高速化への課題

ソースプログラムを入力した場合のインタプリタの動作を**図2**に示す。式(twist '((2 3)(4 5)))が入力されると関数twistの定義が参照され,変数xと引数値((2 3)(4 5))が結合されて,式(list (reverse (first x))(reverse(second x)))の評価※1)を開始する。まず,その第1引数に当たる式(reverse (first x))を評価する。式(reverse (first x))の評価のなかでは,そのまた第1引数である式(first x)の評価,更にそのまた第1引数である変数xの参照が行われる。変数xの値((2 3)(4 5))が求められると,これを用いて関数firstを実行し,その結果(2 3)を得る。次に,関数reverseを実行し,その結果(3 2)を得る。第2引数である式(reverse (second x))も同様にして結果(5 4)を得ると,関数listを実行し,その結果である((3 2)(5 4))を入力された式(twist '((2 3)(4 5)))の値として出力する。

以上述べたLispインタプリタの動作のうちでは,変数の結合と参照の速度が高速化のポイントである。ことにCommon Lispでは言語仕様の厳密化を図るため,以下で述べる静的スコープやクロージャという特殊な機能が拡張されており,一般の変数結合,参照の速度を落とすことなく,これらの機能を実現することが課題となった。

```
(defun twist (x)
   (list (reverse (first x)) (reverse (second x)))
(twist '((2 3) (4 5)))  ——>  ((3 2) (5 4))
```

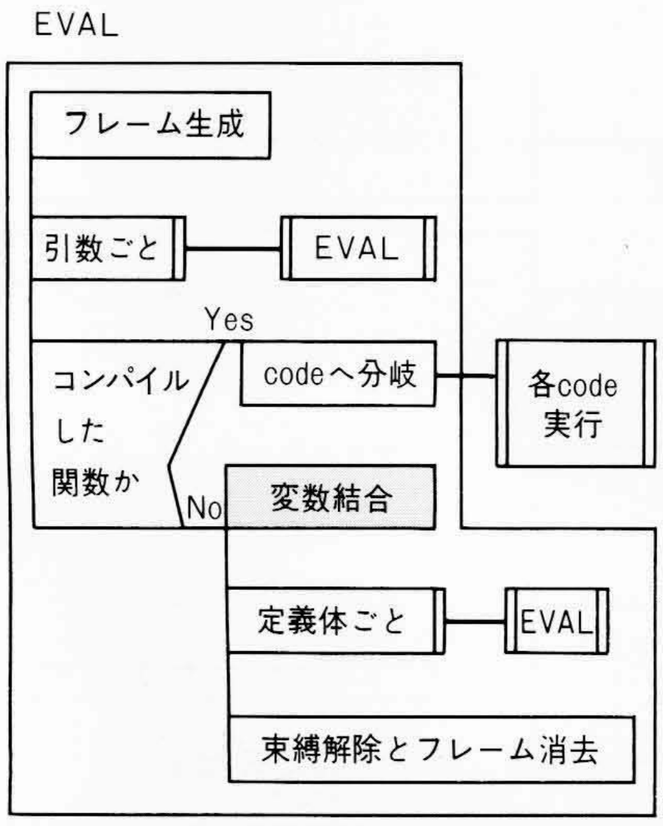


図2 インタプリタ動作 インタプリタでは,式の評価プログラムEVALの再帰呼出しによって,Lispソース プログラムを解釈しながら実行する。

※1) Lispでは,式の値を求めることを評価と言う。

```
(setq x 5)
(defun f (x) (g) )
(defun g () (print x) )
(f 7) --> ?
```

	インタプリンタ	compiled-func.
従来のLISP	7 (動的スコープ)	5 (静的スコープ)
Common Lisp	5 (静的スコープ)	5 (静的スコープ)

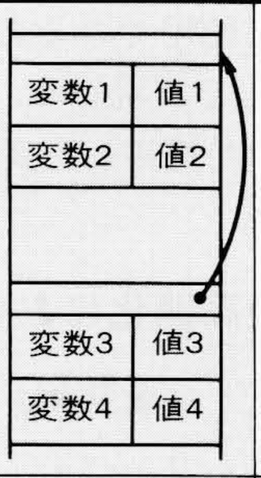
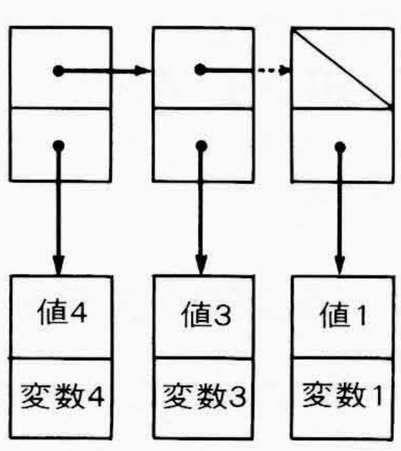
図3 静的スコープ 従来のLispのインタプリタでは動的スコープを採用していたが,Common Lispでは静的スコープを採用している。これによって,インタプリタの動作とコンパイルされた関数の動作が一致する。

3.2 スタック結合方式による静的スコープの実現

従来のLisp言語のインタプリタでは,**図3**に示すようにプログラム動作上最も近い時期に結合された値(同図の例では7)を参照していた(動的スコープ)。このため,プログラムの構造に従った値(同図の例では5)を参照する(静的スコープ)コンパイル後の関数と異なる値を参照する場合があった。Common Lispでは,インタプリタでの動作とコンパイルした関数での動作の仕様統一のため,インタプリタでの変数参照規則として静的スコープを採用することを定めている。

インタプリタでの変数管理方式の比較を**表1**に示す。従来の多くのLisp処理系では,変数名に対応する唯一のアドレスにその時点での値を設定するShallow Binding方式を採ることにより,高速の変数参照がなされていた。しかし,この方式は動的スコープの仕様に合致するもので,静的スコープを実現するためには,変数名と値との結合表をスコープごとに準備するDeep Binding方式を採用する必要があった。Hi LISP

表1 変数管理法 スタック上でのDeep Bindingは,ヒープ上のa-listによるものより高速であるが,クロージャの実現に問題があった。

	Shallow Binding	スタック上での Deep Binding	ヒープ上のa-listによる Deep Binding
方式	変数 値		
性能	◎	○	×
静的スコープ	×	○	○
クロージャ	×	問題あり:スタックは寿命あり。	問題なし

では，Deep Binding方式を高速に実行するため，変数名と値との結合表をスタック上のアレーとして実現するスタック結合方式を採用した。

3.3　クロージャと間接ポインタの併用

クロージャとは，関数を変数などの環境とともに定義する機能であり，この機能を実現するためには，変数などの環境情報を関数定義後も保存する必要がある。スタック結合方式では，スタック上に変数の結合表を持つため，プログラムの動作に伴ってスタックが上下すると参照不可能になる問題がある。Hi LISPでの間接ポインタを用いたクロージャの処理方式を**図4**に示す。クロージャが作られたときだけヒープ※2)上にも結合表を生成し，間接ポインタを用いて値を一括管理する方式を新たに開発した。これによりスタック経由の高速の参照とクロージャの保存を矛盾なく進めることが可能となり，従来の言語仕様のShallow Binding方式による変数管理と比べて，大きな性能の劣化もなく機能を拡張することができた。

4　HiLISPコンパイラの高速化方式

Lispのコンパイラは，リスト形式で与えられたソースプログラムを機械語形式のオブジェクトコードに翻訳し，実行速度を大幅に向上させるものである。HiLISPのコンパイラはこの中間段階で，マシン非依存のL-codeと呼ぶ中間語と，マシン命令対応のLAPと呼ぶアセンブラ語を順に出力する。L-codeは仮想Lispマシンの命令語であり，高速性実現のためアセンブラ語に展開される。

HiLISPのコンパイラの高速化方式を検討するために，Lispの実行特性の解析とCommon Lispの言語上の特徴の検討とを行った。その結果，Lispでは，関数呼出しの時間が実行時間の多くの時間を占めること，また繰返しの処理を再帰呼出しとして表すことが多いこと，などが分かった。また，Fortran，Pascalなどの言語では，変数に対してデータ形が定まるのに対し，Lispでは不定であり実行時の形判定の負荷が大きい。特にCommon Lispでは，データ形を意識しないはん用関数の考えが従来Lisp以上に徹底している。

以上の解析と検討を踏まえ，まず関数呼出し時間を軽減するために，関数呼出しパターンを複数に分類した。また，関数呼出し回数そのものを減らすため，自己再帰展開，ユーザ一関数自動展開などのプログラム変換を実現した。**図5**に自己再帰展開の例を示す。自己再帰展開は，末尾再帰以外の自己再帰呼出しを，自分自身の定義でインライン展開する技法であり，展開後はプログラムの大きさは大きくなるが，呼出し回数を約$\frac{1}{2}$に減少させることができる(同図)。自己再帰展開の際，副作用のため結果が異なる場合に対するチェック，引数が複雑な場合に対する一時変数の用意などにより，自動展開が可能となった。

更に，Common Lispでは，例えばreverse関数のように様々のデータ形を同一関数名で処理できる(はん用関数)。これに対してHiLISPコンパイラは，不完全な形情報からデータ形の形判定・形推定を行って，最適な内部専用関数に置き換える。形判定・形推定の例を**図6**に示すが，この例ではreverseがリスト，配列，文字列のいずれのデータ形に対しても，逆転を行うはん用関数である場合に，引数の形情報からreverseがリストに対する逆転であることを推定し，リスト専用のreverse関数(list-rev)に置き換え，データ形の動的判定のオーバヘッドを減少させている。

HiLISPコンパイラは，以上述べた関数呼出し多様化，プログラム変換，形判定・形推定の高速化技術のほかに，L-codeやLAPに対する最適化も実施している。

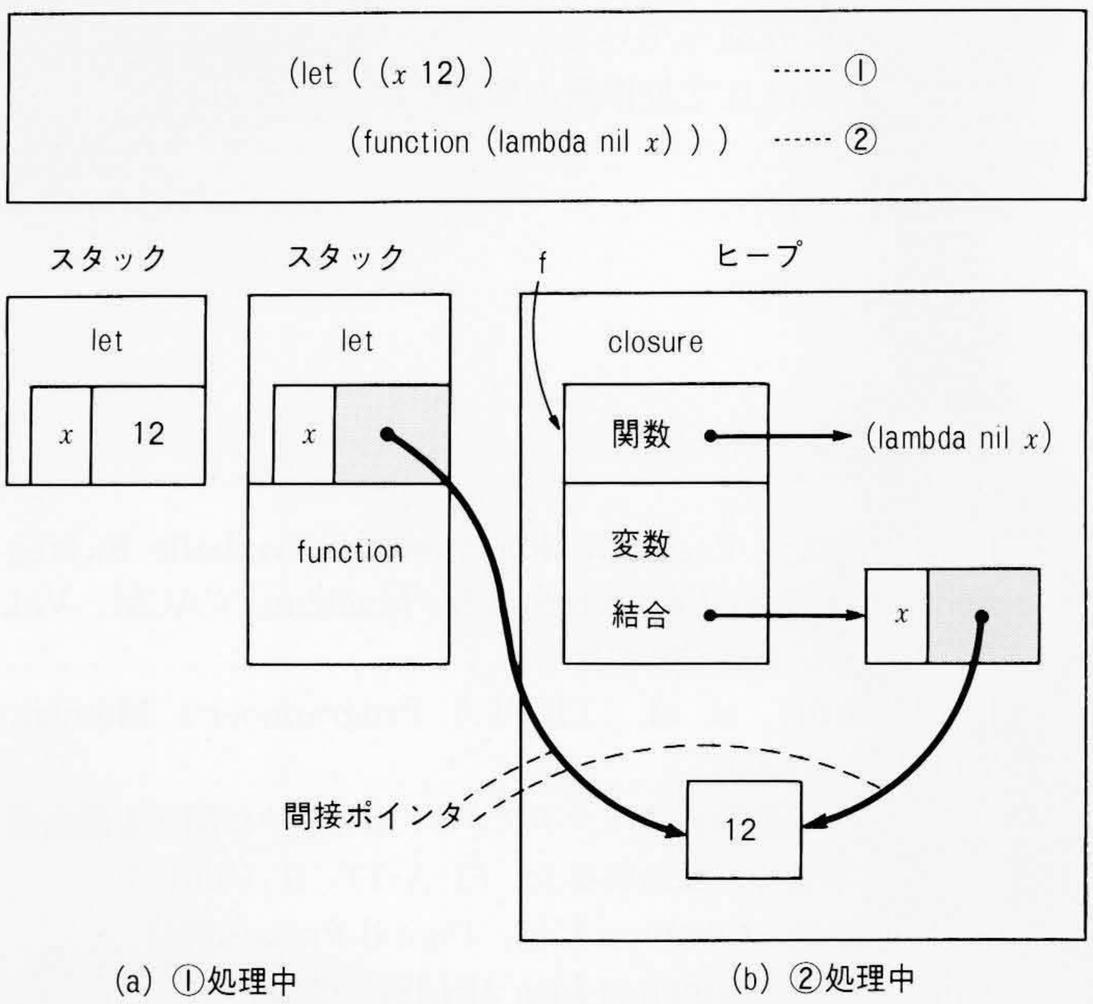


図4　スタック結合と間接ポインタの利用　　変数値を間接ポインタを用いて一括管理することによって，高速のスタック結合方式とクロージャでの変数情報保存を矛盾なく実現した。

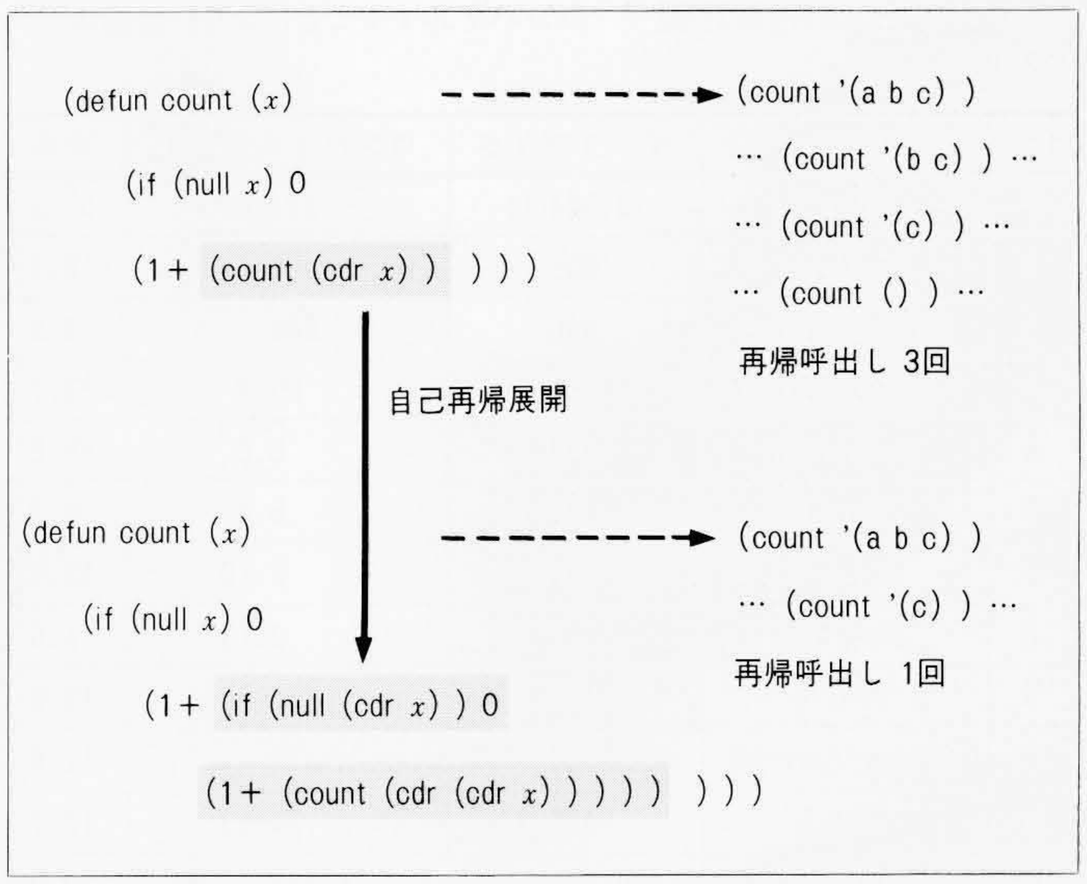


図5　自己再帰展開による再帰呼出し回数の削減　　自己再帰呼出しを自分自身の定義で1回インライン展開することにより，再帰呼出し回数を約$\frac{1}{2}$に削減した。

※2)　Lispのメモリはスタックとヒープで構成され，ヒープはランダム アクセスが可能な領域である。

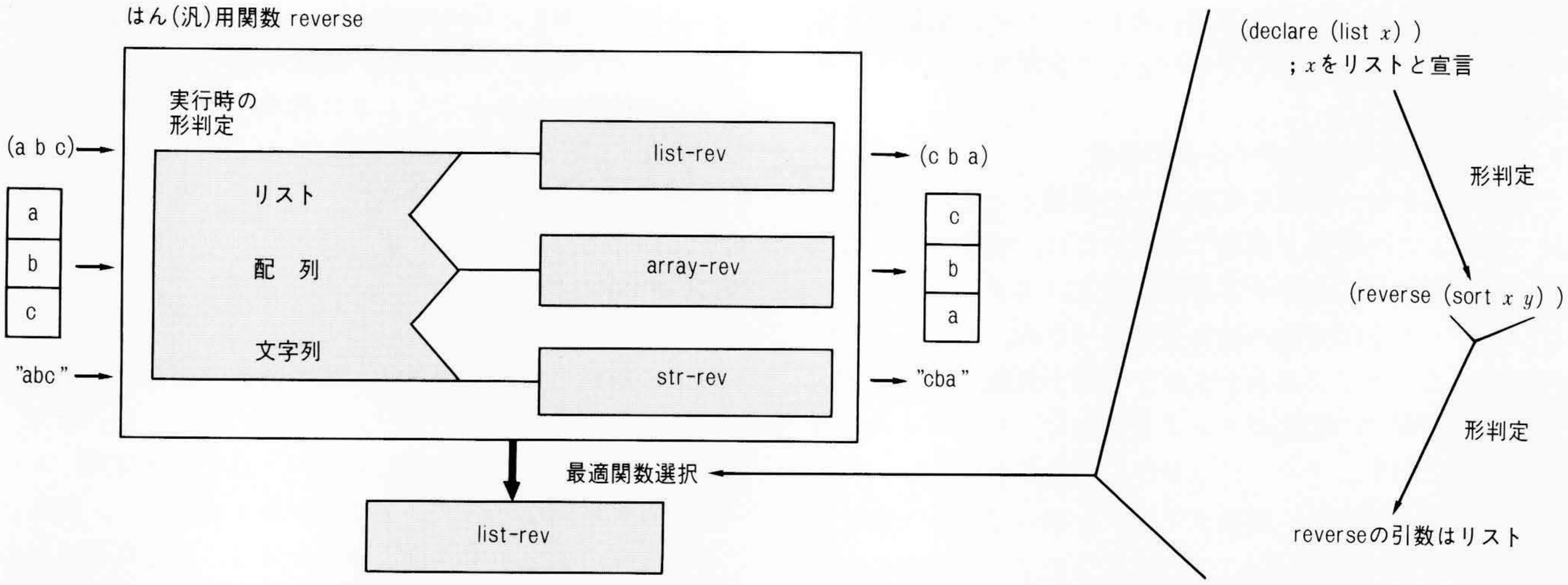


図6 形判定・形推定による最適関数選択 コンパイル時に，不完全な形情報から引数のデータ形を形判定・形推定する。これによってはん用関数を専用関数に置き換え，実行時の形判定の負荷を軽減した。

5 HiLISPシステムの性能評価

情報処理学会記号処理研究会が提案・実施したLispコンテストプログラム[7]のベンチマークから代表12題を選び，HITAC M-280H上で実測した結果をまとめると**表2**のようになる。

#8のベンチマークは，組込み関数だけの性能であり，コンパイル済みコードとインタプリタの性能差はない。残りのプログラムでは，コンパイルすることにより約4倍から60倍ぐらい速くなっている。平均で約20倍である。

この測定結果を，Lispコンテストで最高速のLisp処理系[8]と比較すると，インタプリタでは約半分の性能(静的スコープ採用のためなど)とはなるが，コンパイル済みコードでは前記の高速化技術の開発により，平均約1.6倍程度の高速になっている。

表2 Lispコンテスト代表12題ベンチマークプログラムによるHiLISPシステムの性能評価 コンパイルすることにより，性能は平均約20倍向上する。

#	ベンチマーク	インタプリタ	コンパイルコード	比率
1	Tarai-5	13,964ms	271ms	51.5
2	List-Tarai-4	635	50	12.7
3	String-Tarai-4	946	244	3.9
4	Flonum-Tarai-4	585.3	25.3	23.1
5	Bubble-50	334.7	6.0	55.8
6	Seq-100	35.0	5.2	6.7
7	BITA-6	26.66	2.10	12.7
8	Sort-100	2.19	2.13	1.0
9	TPU-6	3,118.4	229.5	13.6
10	Prolog-Sort-20	653	37	17.6
11	Diff-5	475.31	34.73	13.7
12	Boyer	30,009	1,448	20.7
—	平均	—	—	19.4

注：HITAC M-280Hによる。

6 結 言

インタプリタでの静的スコープの変数管理法，コンパイラでのプログラム変換法，データ形判定・推定による最適関数選択法などの高速化方式を開発して，はん用計算機上でCommon Lispの高速処理系を実現した。オブジェクトコードは，情報処理学会で提案されたLispコンテストプログラムに関して，既に発表されている最高速の処理系の約1.6倍の高性能を達成した。現在，本HiLISPシステムは，エディタ デバッガなどの支援環境も整備されて，日立製作所研究所でCAD，パターン認識，自然言語処理などの研究に広く使用され，研究の効率向上に役立っている。

なお，本方式は日立製作所の製品“VOS3 LISP”に適用の予定である。

参考文献

1) J. McCarthy：Recursive Functions of Symbolic Expressions and Their Computation by Machine, CACM, Vol. 3, No. 3, 184～195(1960)
2) J. McCarthy, et al.：LISP1.5 Programmer's Manual, MIT Press(1962)
3) 日本電子工業振興協会：マイクロコンピュータに関する調査報告書，日本電子工業振興協会，61-A-235(II)(昭61-3)
4) G. Steele Jr：Common Lisp, Digital Press(1984)
5) 湯浦，外：高速Common Lisp-HiLISPの実現，情報処理学会第33回全国大会，2E-1(1986)
6) 高田，外：HiLISPコンパイラにおける高速化方式，日本ソフトウェア科学会第3回大会，C-2-3(1986)
7) 奥野：第3回Lispコンテストおよび第1回Prologコンテストの課題案，情報処理学会記号処理，28-4(1984)
8) 奥野：第3回Lispコンテストおよび第1回Prologコンテスト報告，情報処理学会記号処理，33-1(1986)