記 号 処 理 47－4
（1988. 9. 22）

マルチパラダイム処理系MC上のCommon Lispの実現

松野 年宏*　　井田 哲雄**

* ファコム・ハイタック㈱　　** 筑波大学・電子情報工学系

汎用大型計算機上で動作するLisp-Prolog 融合処理系ＭＣ (Meta Computing environment) のLisp部分について実現法を報告する．本Lisp処理系はCommon Lisp に準拠している．Common Lisp は従来のLispに比べ，多値の返答，関数閉包による静的環境の保存，型チェックの付加の増大等による処理効率低下の要因を含んでいる．本稿では，とくにコンパイラが効率のよいコードを生成するためのこれらの実現法を示すとともに，その評価を行う．

Implementation of Common Lisp on MC

Toshihiro MATSUNO (FACOM-HITAC LIMITED 6-2 Sanban-cho, Chiyoda-ku, Tokyo 102, Japan) and Tetsuo IDA (University of Tsukuba)

Implementation of Lisp which is one of main part of multi-paradigm language MC (Meta Computing environment) is reported in this paper. Language-specification of this Lisp is based on Common Lisp. The emphasis of this paper is mainly on the technique to get efficient compiled code and on showing the effect of optimization with the result of some benchmark tests.

1．はじめに

我々が直面する問題は複雑・多岐であり，今後１つのプログラミングパラダイムが問題解決の万能薬でありえる可能性は極めて低い．むしろ個々の問題の特性に合わせた問題の定式化と解法アルゴリズムの探究が求められる．現時点における計算の体系に関する我々の理解と言語処理系実現のためのソフトウェアテクノロジーを考え合わせた時，必然的に我々はマルチパラダイムを指向せざるを得ない．ここでいうマルチパラダイムとは，「複数個の計算モデルが統一的枠組みの下で共存する」という意味である．このような考え方をとる研究は我々のみならず，国内におけるＴＡＯ〔６〕，海外におけるBobrow〔７〕，Robinson〔８〕らのものがある．我々はＭＣ(Meta Computing environment)と呼ぶ処理系を構築中であるが，今回報告するするものは，ＭＣのうちでも中心的な役割を果たすLispとPrologに関するものである．より具体的には，Prologの実現を意識した上でのLispの処理系に関するものとする．

2．LispとPrologとＭＣ

記号処理向きのプログラミング言語としては，現在のところLispとPrologが代表的であるが，両者の適用範囲および記述能力は包含関係にあるのではなく，相補的な関係にある．Prologの「関係」を主体とした宣言的プログラムは非決定的な問題の記述には適している．しかし，解法アルゴリズムが決定的な問題に対してはPrologの制御構造の記述能力は弱く，むしろLispの「関数」を基本とし，マクロや特殊形式を併用した記述が適している．以上のような理由から「関数」と「関係」を共に用いることのできるLispとPrologの融合処理系の考え方は比較的古くからあった．しかし，実現された処理系をみると処理系実現の仕方は，LispとPrologの融合というよりはLispの処理系の上にPrologの処理系をインプリメントしたものであり，Prologの処理のオーバーヘッドが大きくなっている．

ＭＣでは中核となるメモリ管理機構を統一的に実現し，その上にそれぞれの言語処理系を実現するという方針で設計されている．シンタックスに関してはPROLOG/KRやLOGLISPと同様に，Prologの述語をＳ表現で表し，表記上LispとPrologのプログラムの互換性をとった．LispとPrologのプログラムを統一的にＳ表現とすることは，みかけ上の統一性をとるといったことのみならず，

（ｉ）Lispの関数からPrologの述語の呼び出し（あるいはその逆）をシンタック上一様に記述できる．

（ii）プログラムの変換，メタサーキュラーインタープリタの作成にみられるようなメタプログラミングが容易に行なえる．

といった利点がある．また，Prologのシンタックスを現在広く用いられているDEC-10流に変更するのは，特に困難な問題はないと考えられる．

ＭＣにおけるLisp（以下LISP/MC と表記する）はCommon Lispに準拠している．LISP/MC をCommon Lisp に準拠させたのはCommon Lisp が標準的Lispの地位を得つつあることや比較的良く整理された仕様があることの他に，識別するスコープに静的スコープを採用し，これまで問題とされたコンパイラとインタプリタの不整合を解決している点にある．

Common Lisp の仕様はコンパイラを重視して作られているので，LISP/MC においても，インタープリタでの高速化の試みは特に行わず，コンパイラの生成コードの最適化に重点を置いた設計を行った．

3．ＭＣの記憶管理

3．1　記憶領域の分割

LispやPrologのような言語処理系では，記憶空間の設計の良し悪しが，処理効率を決定する大きな要因となる．ＭＣでは，通常の計算機に見られる基本的な命令（例えば，ロード，ストア，比較命令）を用いても効率良くリスト処理やデータ型のチェックが行なえるように記憶空間を設計した．なお，ＭＣは現在FACOM M-780/20で稼働しており，IBM/370 アーキテクチャに基づいた設計になっているが，コンパイラはIBM/370 の限られた数の基本命令しか生成していない．ＭＣの記憶空間を図１に示す．

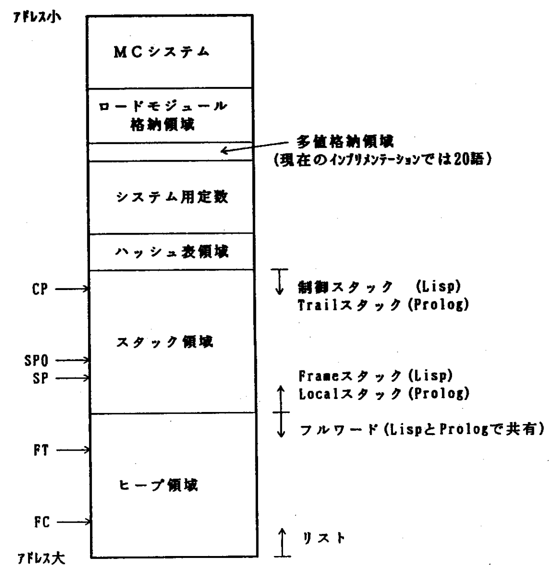

図１　メモリレイアウト

ＭＣを起動する時に，ユーザは，ユーザ記憶空間におけるロードモジュール格納領域，ハッシュ表領域，スタック領域，ヒープ領域の占有比率を指定することができる．

Lisp処理系が使用する部分空間のうち大きなものは，図１で示した部分空間のうち，スタック領域とヒープ領域である．スタックはLISP/MC では制御スタックとフレームスタックに，ヒープはリストとそれ以外の通常フルワードと呼ばれるタグ付，可変長のブロック用に用いられる．両者は使用上は分割されていたほうが都合がよい．しかし，あらかじめ，これらを各々２分割してしまうと領域の使用効率が良くない．LISP/MC ではスタック領域，リスト領域ともに，上／下から別々に，下方向／上方向に伸びるポインタをハードウェアレジスタに持たせて，スタック領域，リスト領域ともに最大限に用いることができるようにしている．

３．２　スタック領域の参照法

LISP/MC では上から下へ（アドレスの小さいほうから大きいほうへ）伸びるスタックを制御スタック，下から上へ伸びるスタックをフレームスタックとして用いている．制御スタックは先頭を指すハードウェアレジスタCPを通じてのみアクセスされる．制御スタックは生成，消去，参照がLIFOでしか行われない．制御スタックはcatch，throwの非正規的制御，特殊変数の束縛，非局所的なblock，return-fromおよびtagbody，go で用いられる．

フレームスタックは，一般的なスタックであり，２つのハードウェアレジスタSPO，SP で現在フレームと１つ前のフレームを指し，スタックフレーム上の変数の参照やフレームの生成，削除を効率良く行うようにしている．値スタックはProlog処理系においても述語を表現するフレームとしても用いられる，また制御スタックは，PrologのTrail スタックを兼用している．実際，ＭＣにおいてはPrologにおけるバックトラックに伴って起こる変数の未束縛状態への復元処理は，Lispにおけるthrow あるいはgoに伴って生じる特殊変数の旧束縛値回復処理と同じとなっている．

SPあるいはCPの変更に伴ってSPとCPが交差したとき，スタックのオーバーフローが起こり，システムによる回復処理が行われる．

３．３　ヒープの管理

ヒープは，最初の空領域を示す２つのハードウェアレジスタにより管理される．FTはフルワードの空領域を示すポインタでありFCは最後に使用されたリストセルを示す．

FCあるいはFTの変更に伴ってFCとFTが交差したときにガーベジコレクタが起動される．上記のような空領域の管理を行っているので，ガーベジコレクションには，圧縮再配置ガーベジコレクションが必要とされる．フルワードの圧縮再配置ガーベジコレクションはリストのガーベジコレクションに比べて計算量が多いので，ガーベジコレクションの起動間にポインタFTがFCの動きより大幅に大きい時にのみ，リストとフルワードを圧縮再配置するガーベジコレクタを起動させ，さもなければ，リストのみをガーベジコレクトする方式をとっている．

４．コンパイラ

Lispコンパイラは他の，より複雑なシンタックスをもつ言語のコンパイラと異なり，一般に次の特徴を持つ．

（ａ）構文解析は不要である．その代わりに，マクロ展開にみられるような，プログラムのソースレベルでのプログラムの変換が必要になる．

（ｂ）意味解析はマクロ展開されたプログラムにおける変数に対する記憶の割り当て，各特殊形式ごとに制御フローを定めることである．

Common Lisp の言語仕様がマクロを用いた言語の拡張を前提としていることから，（ａ）におけるマクロ展開は，Common Lisp のコンパイラでは重要な処理となる．（ｂ）の意味解析は，計算のモデルを中間に介在させることによって，変数の処理を各特殊形式ごとの意味解析以前に終了させるようにしている．ここで用いる計算のモデルはラムダ計算あるいはもっと一般的に関数型言語のモデルとして用いられるＣＣＣ(Cartesian Closed Category) である．

以上の処理を次に示す４つのフェイズ（各フェイズごとに１つのパスをもつ）によって行っている．

４．１　コンパイラのフェイズ

図２にLISP/MC コンパイラの４つのフェイズを示す．各フェイズの役割は次のようである．

（ｉ）前処理

マクロ展開，および高速化あるいはプログラムの意味の明確化を目的としたソースレベルでのプログラム変換を行う．

（ⅱ）ＣＣＣへの変換

コンパイラのインプリメンテーションの観点からは，ＣＣＣはプログラムが関数の組み合わせからなる表現となっていること，および変数は環境における変数の相対位置を示す射影になっていることを特徴とする．ＣＣＣにより以降の意味処理を統一的に行うことができる．

（ⅲ）ＬＡＰ(Lisp Assembly Program) の生成

Ｓ表現のマクロアセンブラに変換する．

（ⅳ）機械語コードの生成

　　　ＬＡＰが機械語のビットパターンに変換される．

本コンパイラはＭＣのLisp-Prolog インターフェイスの機能を利用して，主にPROLOG/ MCによって記述されている．このため，ソースコードおよび中間コードの意味解析でPrologのユニフィケーション機能を利用することによって，Lispで記述する場合に比して規模が小さくなっている．また，Prologのバックトラック機能を利用することによって，意味解釈のやり直しが自動的に行える．このため，本コンパイラは中間ファイルを生成することなく，自然な形で上記の４つのパスが記述されている．

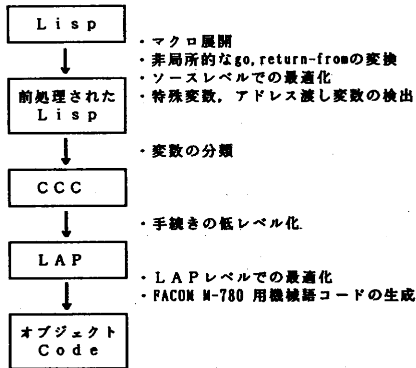

図２　LISP/MC コンパイラの処理の流れ

４．２　変数の処理

ラムダ計算のモデルの構成法からもわかるように，計算の実行において複雑度の低い計算のモデルを作り出すためには，変数を除去することが望まれる．言語処理系のインプリメントの観点からすれば，計算のモデルとして用いられたコンビネータ論理やＣＣＣは変数除去の方法に特徴のあるものであった．我々は，高階関数を扱うことの少ないLispにおいては，上記２つのモデルのうち，ＣＣＣのほうが適切であると考え，ＣＣＣを採用した．

Lispの場合，変数は（ラムダ計算で扱うような）静的スコープを持つ変数に加えて，動的スコープを持つ特殊変数がある（図３）．静的スコープを持つ変数はインプ

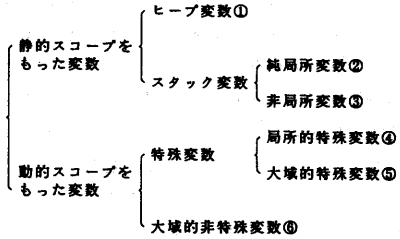

図３　コンパイラによる変数の分類

リメンテーションの観点から，変数の寿命が関数の呼び出し，リターンに同期するスタック原理に従うスタック変数とそれ以外の変数（これをヒープ変数と呼ぶ）に分類できる．スタック変数は，さらに純局所変数と非局所変数に分類される．ここで純局所変数とはその変数に対応する変数の宣言が純局所変数出現を含む最内側のラムダ表現に現れるものをいい，それ以外のスタック変数を非局所変数という．ここで，上の分類は，宣言された同一名の変数に対するものではなく，各変数の出現ごとに対する分類であることに注意．

例えば

```
(lambda (x) (cons x
                  ①
  ((lambda (y) (setf x (+ x  y))) 1)))     (*)
                     ②   ③  ④
```

において，変数出現①は純局所変数，②，③は非局所変数，④は純局所変数である．

スタック変数はその名前が示すようにスタック上に束縛値を格納するセルがとられるものであるが，純局所／非局所によって参照の仕方が異なる．純局所変数は，スタックフレームの先頭を示すレジスタSP0 からの相対番地で参照されるが，非局所変数は，SP0, SPしかフレームへのポインタを持たない我々のシステムでは効率良くアクセスする手段がない．ALGOL60 のようなブロック構造をもった言語の処理系では，display の手法が用いられるが，Lispでは非局所変数は純局所変数に比べて使用される頻度は一般に少なく，このような手法はかえって非効率を招く．我々は，変数のリンクを付けるフェイズでプログラムを変換することにおよって，この問題を解決

```
(defun f ()
  (let ((x 1)(y 2))
    (let ((z #'(lambda ()
                  (setq x (+ x y)))))
      (funcall z)
      (list x y))))
```

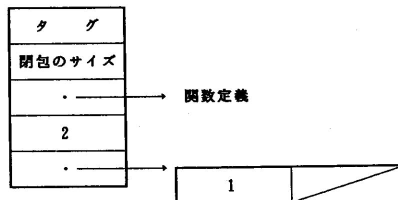

図４　コンパイルされた閉包の構造

している．すなわち，(*) 式は

```
(lambda (x)
  (cons x
    ((lambda (x)
       ((lambda (y) (setf !x y)) 1)) @x)))
```

と変形することによって，非局所変数へのポインタを

```
#'(lambda (y) (setf !x y))
```

に渡している．ここで，@xはx のアドレス（ポインタ）を示し，!xはx の内容を１回とった(dereferenceした)値を示す．コンパイラは非局所変数の使われ方をみて（上の例ではx の値が参照されるのみならず，更新される）ポインタを渡すか値を渡すかを決定している．(*)の代わりに，

```
(lambda (x)
  (cons x
    ((lambda (y) (+ x (+ x y))) 1)))          (**)
```

とすると，コンパイラは(**)式を

```
(lambda (x)
  (cons x
    ((lambda (x)
       ((lambda (y) (+ x (+ x y))) 1)) x)))
```

のように変形する．この場合には，x の束縛値のみが内側のラムダ表現で用いられるからである．

ヒープ変数は，その寿命がスタック原理に従わないものであり，閉包の生成に関連して作られる．閉包は関数の定義と，それが生成された場所での静的な環境（静的スコープの変数，局所関数，block 名，goの飛び先）との対である．通常はLispユーザからは見えないCommon Lispのオブジェクトである．閉包生成時の静的な環境はヒープ上の閉包内に格納される必要があるが，その形式として２つの場合が考えられる．すなわち，図４の変数x のように閉包内で代入される変数は以後その副作用が反映されるようにヒープ上に値をコピーしそれへのポインタを受け渡す必要があるのに対し，変数y のよう参照されるだけの変数の値はスタックで受け渡すことができ，閉包にはその値を格納すればよい．LISP/MC コンパイラでは前処理の段階で閉包内での静的変数への代入を検出し，ヒープへの値のコピーを減らすようにしている．なお，ヒープへの値のコピーはその変数のスコープにおける最初の出発点において
(setq x (cons x nil))のコードを付加することによって実現している．

特殊変数は，そのスコープの始まりで旧値の保存と新値の設定，スコープの終わりで旧値の復元を行うようなシステム関数を付加することによって実現されている．proclaim, defvarで宣言された大域的な特殊変数は，局所的なスペシャル宣言が可能なすべての場所で宣言されたものとして処理される．以上の変数に関する処理はフェイズ（ｉ）で行われる．

次にフェイズ（ⅱ）において，スタック変数のＣＣＣ表現への変換が行われる．既に述べたように，中間コードとしてＣＣＣ表現を生成することの１つの意義は，ＣＣＣへの変換によって静的な変数の名前が除去され，対応する射影へと変換されることにある．射影はコード生成を行う際，スタックでの現在フレーム内のオフセットを計算するのに用いられる．

ヒープ変数はポインタによって間接的にアクセスされなければならないが，前節で用いた変数並び中の相対位置からスタック上のアドレスを計算する演算子@ と，アドレスから変数値を間接的に参照する演算子! の２つを用いて実現している．

特殊変数の処理は既に前処理の過程でなされているので，ＣＣＣ表現では，動的スコープをもった変数は，それが図３の④～⑥のいずれの種類かに関係なくその変数の名前が以降の意味解析のフェイズに渡されることになる．

５．最適化の手法

Common Lisp は従来のLispに比べてデータ型の増大，多値返答関数の導入，型の一般化と総称関数の導入等処理速度の低下をもたらす要因が多く存在する．これらの問題の一部はコンパイラによる簡単な文脈解析によって解決されるが，システム全体の設計の検討を必要とする問題も含まれている．このような問題として多値の扱いと関数呼び出しの手続きについて次に述べる．また，LISP/MC ではLispコンパイラの一般的な高速化技法として，基本的関数のインライン展開，末尾再帰の反復処理への変換を行っているが，ここではそれ以外の最適化の手法について述べる．

(1)多値の処理

Common Lisp の仕様では，関数は多値を返すことができるが，これが実際に用いられることは稀である．このため，関数が単値を返す場合の負荷をいかに軽減するかが問題となる．LISP/MC では３つのレジスタを利用して多値を処理する．それらはそれぞれ，第１値を返すレジスタ，第１値を除く多値の数を返すレジスタ，第１値を

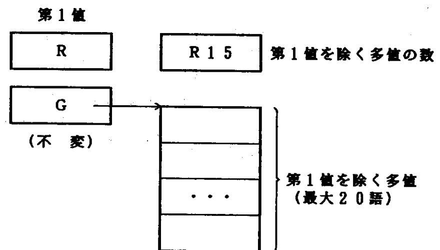

図５　多値の処理

除く多値が格納されている先頭アドレスが格納される（図５）．このうち，多値の先頭アドレスは不変であるので，多値を扱うためのオーバーヘッドは多値の数をレジスタへセットすることだけである．さらに，多値の数を格納するレジスタと関数呼び出し時の飛び先のアドレスを格納するレジスタを共有し，多値の数の最大数を超える数を０と同一視することにした．これにより，関数呼び出し時に，多値の数を格納するレジスタは０（を意味する数）にセットされるため，システム関数が単値を返す場合には多値を返すことによるオーバーヘッドは存在しないことになる．

⑵関数呼び出しの簡素化

フレームスタック上には各関数に対応してフレームが割り当てられ，引数の受け渡しやレジスタの退避に利用される．よく用いられる方式は，各関数で必要となるスタックの大きさの最大値を関数が呼び出されるごとにオーバーラップさせないように割り当て，引数領域の範囲をレジスタで示すものである（図６（ａ））．引数の数をｎとするとｋ番目の引数は引数領域の先頭を示すレジスタＡを基準とし，オフセット値４（ｎ－ｋ）でアクセスされる（１語を４バイトとする）．しかし，この方式には次の問題がある．

（ｉ）空きが生じる分だけスタックを浪費しているのみならず，GCのために未使用領域を初期化するオーバーヘッドがある．

（ii）ある関数が別の関数を呼んだとき，引数領域の上限を示すレジスタＡが破壊されため，その関数の復帰後に引数をアクセスするためにはＡの値をレジスタ退避領域から改めてロードし直す必要がある．

LISP/MC ではこれらの問題を（ｂ）のようにフレームを重ねて割り当てることによって解決している．ここでｋ番目の引数はレジスタSP0 を基準としてオフセット値

フレーム２のサイズ
＋レジスタ退避領域のサイズ
＋４（ｎ－ｋ）バイト

でアクセスされる．レジスタSP0 の値は関数復帰時の処理で復元されので，（ａ）に比べ引数アクセスのオーバーヘッドが軽減されている．また，関数呼び出し時には子の関数のフレームの下限を示すレジスタSPが引数領域の先頭を知らせる役目を兼ねているため，（ａ）で用いたレジスタＡは不要になる．

⑶コンパイラによるレジスタ状態の追跡

LISP/MC で関数の呼び出しを行う手順は次のようである．

（ｉ）引数を評価してフレームスタックに格納する操作は関数呼び出しフォームの実引数を左から順に評価しながら行う．

（ii）飛び先のアドレスをレジスタにセットして分岐する．

（ｉ）で最後の実引数を評価した結果は，呼び出される関数の頭ではレジスタに残っているため，これを直接用いれば，スタックフレームに格納されている値を参照するよりも高速である．システム関数はこのことを利用している．また，コンパイラも値が格納されるレジスタに各時点でどの引数が格納されているかの情報をトレースすることによって引数の不要な再ロードを避けている．⑴で述べた多値の数を格納するレジスタについても同様に，状態の追跡を行っている．すなわち，関数が呼び出された時点では，飛び先のアドレス（多値の数の最大個数を超えるので０を意味する）が格納されており，以後関数呼び出しが行われる前に単値を返す箇所では，多値の数を格納するレジスタに値をセットすることを抑制している．例えば，

```
(defun factorial (n)
  (if (eq n 0) ......... ①
      1 ................ ②
      (factorial (1- n))))
```

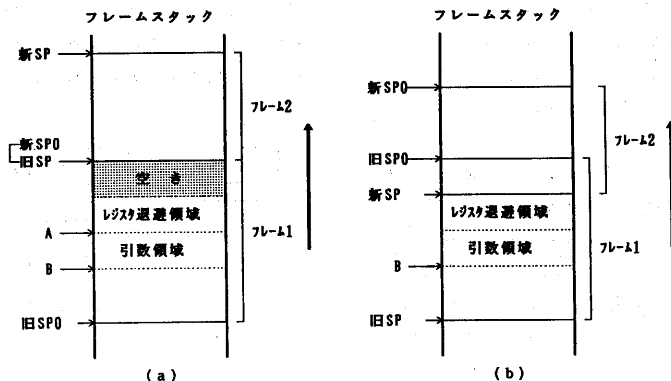

図６　スタックフレームの構成

では，①のｎの値は最後の引数であるのでレジスタに存在しており，これを直接アクセスする．②で１を返す時点では（eqがインライン展開されるとすれば）関数の先頭から関数呼び出しがないので多値の数を格納せずに値だけをレジスタに格納して呼び出し元へ復帰する．

⑷閉包生成の抑制

閉包の生成は実行時に行う必要があり，その負荷も高い．実際には，閉包は，その時点の環境を別の環境に持ち出す場合のみに生成する必要がある．map 関数やfuncall の次に閉包が現れる場合には別の環境に持ち出されることはないので，コンパイル時に前処理によって図７のように閉包を生成しない形に変形される．

⑸car，cdrは簡単な処理であるのでコンパイル時にインライン展開されるのが普通であるが，Common Lispの仕様では nilのcar，cdrを nilとする例外的な処理があり，引数が nilかどうかのチェックが必要なため従来のLispにくらべてオーバーヘッドが大きい．実際には，引数を nil としてこれらの関数が呼ばれることは稀であるので，引数がコンスの場合に負荷を軽くすることが課題となる．本コンパイラでは，car，cdrのインライン展開で図８のように例外処理を関数からの復帰処理の後に置くことによって引数がコンスの場合の命令数を減らしている．

６．評価

Gabriel のベンチマークプログラム集に基づいた測定結果を表１に示す．非局所的な制御に移動を伴うctakは制御スタックと値スタックの分離によって他の処理系に比べて良い結果となっている．また，リスト処理を含むtaklについてはcar，cdr の展開形を変更することによって約12%の速度向上が図られた．

７．おわりに

Common Lisp 準拠の処理系について，その実現法を中心に述べた．高速化については，他の計算機環境でも共通に利用できるものについて幾つか紹介した．コンパイラの開発では利用頻度の高い処理に関しては，小さな最適化のが以外に大きな効果をもたらすことが確認された．現在，さらなる高速化のために型推論による最適化コードの生成部分を開発中である．これについては別の機会に報告する予定である．

```
(funcall #'(lambda (x y z) (+ x y z))
              (print (+ x y z)))
          e1 e2 e3)

                ⇩

(let ((x e1)(y e2)(z e3))
  (print (+ x y z)))
```

```
(mapc #'(lambda (x y z)
              (print (+ x y z)))
      ls1 ls2 ls3)

                ⇩

(do ((#:g0001 ls1 (cdr #:g0001))
     (#:g0002 ls2 (cdr #:g0002))
     (#:g0003 ls3 (cdr #:g0003)))
    ((or (atom #:g0001)
         (atom #:g0002)
         (atom #:g0003))
     ls1)
  (let ((x (car #:g0001))
        (y (car #:g0002))
        (z (car #:g0003)))
    (print (+ x y z))))
```

図７　クロージャの変換

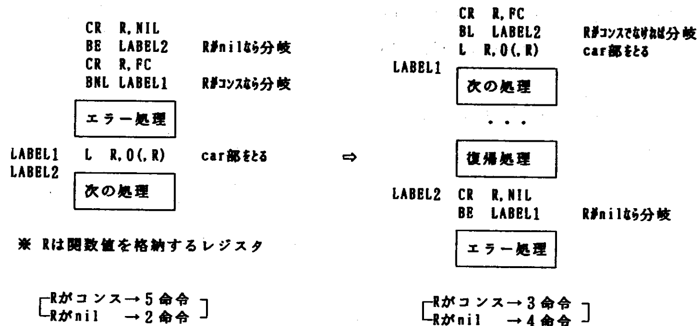

図８　car のインライン展開

Benchmarks	LISP/MC	UTILISP*	PSL**	評価項目
(nfib 20)	11 msec	12 msec	—— msec	関数呼び出し
(tak 18 12 6)	28	30	44	関数呼び出し
(stak 18 12 6)	117	104	1130	特殊変数の処理
(ctak 18 12 6)	63	97	570	catch & throw
(takl 18 12 6)	192	287	300	リスト処理
derive	119	114	1300	リスト処理, map関数
dderive	152	184	1440	リスト処理, map関数
destructive	135	114	440	リスト処理
div2 (test-1)	89	82	570	リスト処理
div2 (test-2)	91	101	600	リスト処理

表1　ベンチマークテスト（コンパイラ）結果

* on FACOM M-780/20
** on CRAY-1
(from "Performance and Evaluation of Lisp Systems", R.P.Gabriel, MIT Press, 1985)

参考文献

〔1〕Guy L. Steele Jr.; Common Lisp the Language, Digital Press, 1984

〔2〕R.P.Gabriel; Performance and Evaluation of Lisp Systems, MIT Press, 1985

〔3〕松野，井田；マルチパラダイム環境MC上のLisp コンパイラの実現，第36回全国大会論文集，1988

〔4〕松野，井田；LISP/MCコンパイラの型推論，情報処理学会第37回大会論文集，1988

〔5〕井田；ラムダ計算とそのモデル「カルテシアン閉カテゴリ」によるCOMMON LISP の解釈と新しい処理系，コンピュータソフトウェア，Vol.4，1987

〔6〕I.Takeuchi, H.Okuno and N.Osato; A list processing language TAO with multiple programming paradigms, New Generation Computing, 401-444, April, 1986

〔7〕P.G.Bobrow; If Prolog is the answer, what is the question?, Proc. International conference on fifth generation computer systems, 138-145, Nov. 1984

〔8〕J.A.Robinson and E.E.Silbert ; LOGLISP: Design and Implementation