Common Lisp 用最適化コンパイラの設計と試作†

安村通晃†† 高田綾子†† 青島利久††

汎用大型機上で動く，Common Lisp の最適化コンパイラを設計・試作した．Common Lisp は，人工知能等の分野における実用的な応用に必要な機能を十分に備え，かつ関数性を従来 Lisp 以上に取り込むことを目的とした近代的な Lisp 言語である．一方，従来の Lisp 処理系のユーザは，実行性能の点などで必ずしも満足していなかった．このため，我々は，Common Lisp に準拠した高速の処理系 HiLISP とそのコンパイラを設計・試作した．ここでは，HiLISP コンパイラの設計と最適化の方式を中心に述べる．HiLISP コンパイラは，高速性と移植性を考慮して，仮想 Lisp マシン語である Lcode を中間語として生成する．最適化の方式として，関数呼出しの最適化，型判定の最適化，局所最適化の各々の課題に対して，それぞれ，自己再帰展開，コンパイル時の型判定・型推定，パイプラインを意識した命令列の並べ替えなどの方式を設計し，試作した．試作した HiLISP コンパイラに対して，Lisp コンテスト代表 12 題ベンチマークにより，各最適化項目の性能を評価した．この結果，組込み関数展開の効果が最も大きく，次いで型判定・型推定の効果が大きいことがわかった．再帰関数展開，命令列の並べ替えなどの効果も確認できた．ここで提案する最適化方式は，Lisp コンパイラ，特に，Common Lisp コンパイラに有効な方式である．

1. はじめに

Lisp が誕生して 25 年以上経過するが，この間様々の Lisp の方言と処理系が作られてきた．最近，これらの Lisp 言語系のうち，主要な方言の共通仕様を定める形で，Common Lisp の言語仕様が決められた[1]．Common Lisp は，単に従来 Lisp の機能を包含するだけでなく，静的スコープ，完全関数閉包 (closure)，多値などの採用にみられるように，関数を一等級のオブジェクト (first class object) とした言語である．同時に，機能面においても，最近の人工知能分野等の応用の拡大に対応して，構造体などのデータ型の拡充，数値演算・列 (sequence)* 演算などにおける汎用関数化の徹底，さらに format を含む入出力の強化等，豊富な機能をもっている．このような機能の拡大に伴い，その処理系の実行性能の低下が懸念される．

一方では，これまでの Lisp のユーザは，Lisp の実行速度の遅さや，言語の機能不足・互換性の欠如，メモリサイズの制限，あるいは日本語機能が不十分であること，などの点に不満を持ってきた[9]．なかでも，実行性能に対する要望が最も強い．

我々は，研究所内での人工知能研究の基本的なツールとしての Lisp 言語を考え，Common Lisp に準拠し，上で述べた問題点の解決を目指した言語処理システム HiLISP (High performance Lisp) とそのコンパイラを設計・試作した．HiLISP システムのユーティリティのひとつである HiLISP コンパイラは，高速オブジェクトを生成するという点で，HiLISP システムの最も重要な部分である．

ここでは，HiLISP コンパイラの前提となる HiLISP システムの設計方針を最初に述べ，次に HiLISP コンパイラについて，その設計方式と，最適化を中心とする実現方式とについて述べる．最後に，最適化項目の評価を述べる．

2. 設計方針

汎用大型機上で動く Lisp システムとして，HiLISP システムの要求事項を次のように定めた．

(1) 高速であること—汎用機上でどこまで高速性能が引き出せるか，また，他言語 (Pascal, C 等) に比べて，どのくらいの性能になるかに注目した．さらに，次に述べる Common Lisp の仕様を満足した上で，従来 Lisp に比べどの程度の実行性能が出せるか，という点にも関心があった．

(2) Common Lisp 準拠の仕様—機能の点と標準指向の Lisp という点で，言語仕様は Common Lisp 準拠とした．

(3) 日本語機能の完全サポート—注釈やデータだ

† Design and Implementation of an Optimizing Compiler for Common Lisp by MICHIAKI YASUMURA, AYAKO TAKADA and TOSHIHISA AOSHIMA (Central Research Laboratory, Hitachi Ltd.).

[†† (株)日立製作所中央研究所

* リストと一次元ベクトルとの共通構造

けでなく，シンボル名などにも日本語が使えることが必要である．

（4） 使いやすい支援環境—構造画面エディタや使いやすいデバッガを備えていることも必要である．

（5） 広大なメモリ空間—従来の大型機上の Lisp は，16 MB 程度のメモリ空間しか使えないものが多かったが，2 GB 程度のメモリ空間が使えるようにする．

（6） その他—他言語呼出しや，OS とのインタフェースなども必須項目である．

以上のような要求事項に加えて，システム自身の移植性，保守性，開発の容易性も考慮することにした．

特に，HiLISP のコンパイラにとっては，（1）の高速性と，（2）の Common Lisp 準拠の2点が最も直接的な要求項目である．すなわち，Common Lisp の豊富な機能，たとえば，汎用関数化の徹底などに対して，性能を低下させることなく，いかに高速性能を実現するかが課題である．日本語機能は，組込み関数として，また，支援環境はエディタ・デバッガなどのユーティリティとして実現するため，コンパイラとの直接的な関係は少ない．

高速性は，その他の課題（たとえばメモリ効率や信頼性など）とトレードオフになることが少なくない．我々は，主要な処理はできるだけ性能を落とさないように，また，コンフリクトする目標に対しては，オプションで使い分けられるように考えた．たとえば，関数閉包や多値を含むプログラムで，関数閉包や多値が直接現れないプログラム部分では，オブジェクトの実行性能を落とさないようにする．また，最高速オプションを指定した場合は，コンパイル時間やメモリ効率より実行性能を優先させる，などである．

HiLISP システムは，インタプリタ，入出力，組込み関数群，コンパイラ，エディタ・デバッガ等のユーティリティなどから成る．インタプリタは，Common Lisp で書かれたソースプログラム（S 式）を，組込み関数等を使いながら，そのまま実行する．HiLISP のコンパイラは，S式を入力とし，機械命令列からなるオブジェクト（code）を出力する．HiLISP のコンパイラは，中間語として，マシン独立で仮想 Lisp マシン語である Lcode (Lisp intermediate Code) とマシン依存でアセンブラレベルの中間語である LAP (Lisp Assembly Program) とを途中で生成する．

今回新たに設計した Lcode は，仮想 Lisp マシンを想定した中間語であり，**表1**に，その主要な中間語の一覧を示す．仮想 Lisp マシンとしては，スタックと，ヒープおよび複数の汎用レジスタから成るマシンを想定した．演算は，レジスタ間，または，レジスタとメモリ（スタックまたはヒープ）との間で行われる．引数は，スタックにのせて渡し，結果は，特定のレジスタに入れて返す．

Lcode は，表1に示すとおり，関数インタフェース，変数アクセス，スタック操作，リスト操作，メモリ割当て，データアクセス，分岐，データ定義，ユーティリティ等に分類される．このほか，演算系の Lcode があるが，表では，省略した．Lcode の大部分が Lisp

表 1 主要 Lcode の一覧
Table 1 Summary of typical Lcode.

#	分類	Lcode	備考
1	関数インタフェース	¥ENTRY	関数入口
		¥PREPARE	フレーム作成
		¥CALL	関数呼出し
		¥RETURN	関数復帰
		¥EVAL	eval 呼出し
2	変数アクセス	¥VALUE	値参照
		¥SETVALUE	値設定
		¥INTERN	インターン
		¥BIND	束縛
		¥UNBIND	束縛戻し
3	スタック操作	¥PUSH	push
		¥POP	pop
		¥ARGREF	引数参照
		¥ARGSET	引数設定
4	リスト操作	¥CAR	car 参照
		¥CDR	cdr 参照
		¥CARSET	car 設定
		¥CDRSET	cdr 設定
5	メモリ割当て	¥CONS	cons
		¥LIST	list
6	データアクセス	¥GETSYM	symbol をレジスタに設定
		¥GETSTR	string をレジスタに設定
		¥GETFIX	fixnum をレジスタに設定
7	分岐	¥IFxxx	xxx はデータ型，t, nil 等
		¥GOTO	無条件分岐
8	データ定義	¥END	関数末尾
		¥CONST	定数値定義
		¥SYMBOL	シンボル定義
9	ユーティリティ	¥KEY	キーワードパラメタ照合
		¥GET	属性リスト検索
		¥TYPERROR	型エラー処理

のプリミティブに対応している.

Lcode を用いることによって，タグのビット位置やシステムの特定のアドレス，あるいは，レジスタ番号等，実現の詳細を意識する必要はなくなり，また，移植性も良くなる.

3. コンパイラの最適化方式

コンパイラの最適化の課題を明らかにするために，Lisp プログラムをいくつか解析した．Lisp プログラムでは，Fortran のプログラムなどと比べて，関数（サブプログラム）呼出し部分の実行時間比率が大きいのではないかと考えられるが，実プログラムでの解析の結果，関数呼出し部分の実行比率がきわめて大きく，全実行時間の約半分を占める場合も少なくない，ということが確認された．したがって，まず関数呼出しの高速化が課題である．次に，Lisp では他の言語(たとえば Fortran や Pascal など）と異なり，組込み関数の実行比率が高く，しかも組込み関数の多くが汎用関数 (generic function) になっている．汎用関数化は，特に Common Lisp において著しい．汎用関数においては，実行時の型判定をいかに減らすかが課題である．さらに，汎用大型機で十分速い性能を出すには，パイプラインを活かすことが重要である．そのためには，局所最適化が課題である.

以上の検討結果に基づき，HiLISP コンパイラの最適化の主要課題を次の3項目とした：

（ⅰ） 関数呼出しの最適化
（ⅱ） 型判定の最適化
（ⅲ） 局所最適化

以下では各課題に対する最適化の方式を述べる.

（ⅰ） 関数呼出しの最適化

関数呼出しの最適化のためには，(a)呼出し時間の削減と，(b)呼出し回数の減少との2通りの方法が考えられる.

関数呼出し時間を短縮するためには，関数呼出しの際のスタックフレームの準備と，復帰の際のスタックフレームの戻しの処理をできるだけ短くすればよい．通常の関数呼出しではインタプリタとのインタフェースを壊さない範囲で最短の命令列（でかつ最短のマシンサイクル数）となるように設計した．また同時に，呼出し先が，自分自身か，code に限るか，固定個の引数に限るか，などの条件を調べて，最も短い命令列を出力するようにした．このため，¥call の Lcode は，条件ごとに多様化させた.

次に，関数呼出し回数そのものを減らす方式として，従来から多くの Lisp コンパイラで行われてきた末尾再帰のループ化の実施や，組込み関数のインライン化の強化のほか，新たに，自己再帰関数の自動展開の方式を開発しこれを実現した．同時に，ユーザ関数展開も実現した.

ここで，自己再帰とは，自分自身を直接に呼び出す再帰であり，末尾再帰*もその一種である.

関数の展開は，Burstall, Darlington[8] が関数型言語に対して定式化したプログラム変換のうち unfolding と原理的には共通しており，また，関数名をユーザが指定すれば展開を行うプリプロセッサなどはあった．関数の自動展開は次に述べるような問題等があり，実用的なプログラムに対して自動化はあまり行われてこなかった．我々は，Lisp には自己再帰が多いこと，自分自身の関数本体の性質が解析しやすいこと，関数の定義変更の問題が無いこと，等を考慮して，自己再帰に着目した展開を自動的に行う方式を考えた.

展開を行ったとき，元のプログラムの意味の不変性と，効率低下の可能性とが問題となる．前者は，(a)展開後の名前の衝突，(b)nconc 等の副作用を引き起こす関数の使用，(c)グローバル変数への値の書き込み，などの問題であり，後者に関しては，(d)メモリの増大，(e)呼出し回数の増加，(f)末尾再帰ループ化との兼ね合い，などの問題である.

(e)の問題は，実引数に，ユーザ関数呼出し，あるいは，“複雑な”** 組込み関数呼出しになっている場合に，対応する仮引数がプログラム中に複数回現れるとき，かえって呼出し回数が増え効率が低下する，という問題である．この問題に対しては，一時変数を用いて，解決する．(f)の問題は，従来から末尾再帰のループ化という技法が知られているが，これと展開とでどちらが有利か，という問題である．メモリの点を考慮して，我々は，末尾再帰が有利と判断した.

これらの点を考慮して，自己再帰関数の展開を次の方針に従って行うこととした．(変数のセマンティクスについては後述.)

（1） 末尾再帰は展開しない.
（2） 自己再帰関数がネストしたときは，外側に関してのみ展開する.
（3） 展開は，各呼出しに関し一段のみ行う.

---

* 末尾再帰として，他の関数を呼び出す場合もあり，したがって，自分自身を直接に呼び出す末尾再帰は，厳密には末尾自己再帰とよばれる.

** “複雑な”組込み関数呼出しとは，組込み関数の規模が大きく，インライン展開できないような関数呼出しを言う.

(4) 関数定義中の再帰呼出しについてのみ展開する.

(5) グローバル変数への代入等副作用があるときは原則として展開しない.

(6) 展開部にユーザ関数呼出し，または，"複雑な"組込み関数呼出しがあるときなどの場合は，一時変数を用意して，あらかじめ関数値を計算しておく.

この方針に基づく自己再帰関数の展開の具体的手順は図1に示す．最初に関数定義部全体について，展開可能性（末尾再帰でない自己再帰の存在）と変換の正当性（副作用の有無），および効率（プログラムサイズ）の検査をして，展開を実施すべきかどうかきめる．次に各自己再帰呼出し部について，展開の可否を調べ可能なら展開を実施する．このとき，ローカル変数名の変更や一時変数の利用など必要に応じ行う.

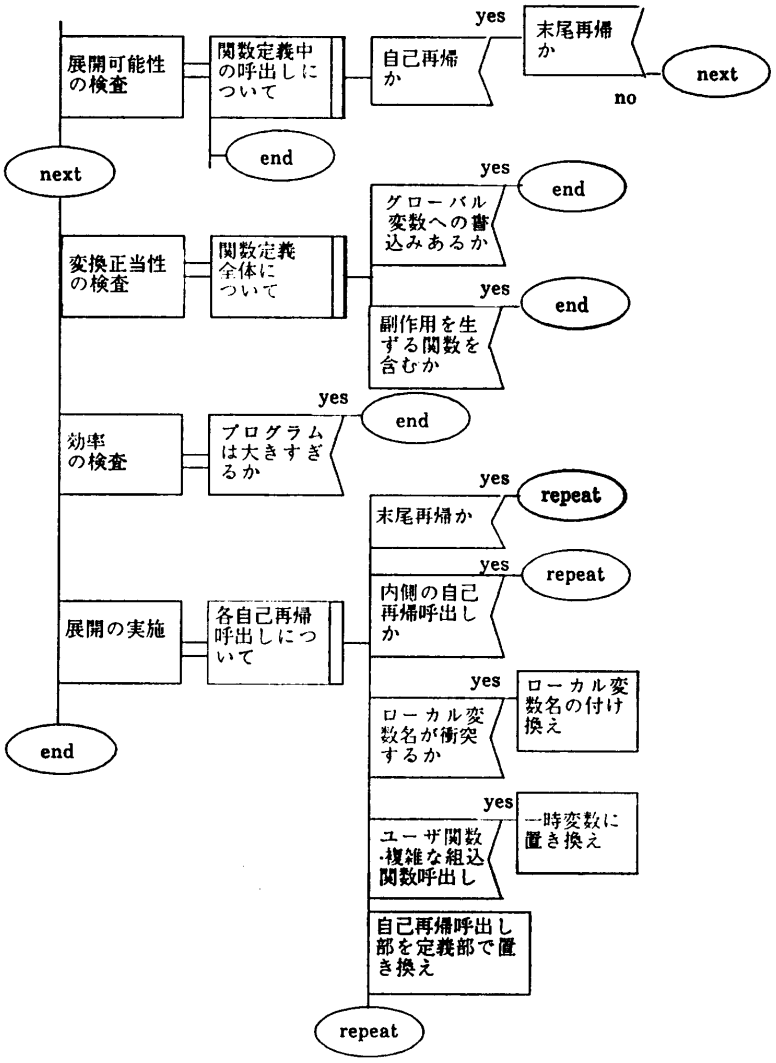


図1 自己再帰展開のアルゴリズム
Fig. 1 An algorithm of self recursive expansion.

自己再帰関数の展開の例を図2に示すが，ここでは，l-tarai の第1引数が組込み関数 copy-tree の呼出しになっているため，一時変数wを用いて，複数呼出しをwにおきかえて展開している．一般にはこのように，一時変数の利用が必要な場合が少なくない．nconc 等副作用を生じる関数呼出しを含む場合も，一時変数を用いることによって，呼出し箇所をまとめ，展開を行うという方法もある（図3）.

Common Lisp の関数は，値渡し引数 (call by value) で適用順評価 (applicative order evaluation) で計算され，純粋の関数型言語は，一般に名前渡し (call by name) で正規順評価 (normal order evaluation) で計算される．また，正規順評価の計算のクラスの方が適用順評価のそれよりも広いことが知られている[11]．自己再帰関数の展開は，呼出し部を展開することによって，その部分のみ，値渡し・適用順評価を名前渡し・正規順評価に変えたことに相当する．副作用を発生させる関数を含む場合に，引数の渡し方・評価順の違いにより，副作用の起こり方が異なってくる．純粋の関数型言語では副作用がないため，本方式のような副作用の検査は不要である．また，関数型言語では，効率向上のため，正規順評価を適用順評価に変換する場合があり，このとき，計算のクラスが変換前と同じか検査する必要があるが，本方式では逆に，適用順評価を正規順評価に変換しているため，この問題はない．しかし，正しくないプログラムの動作が変わる（たとえば，停止しないプログラムが展開後は停止するようになる（図4)），というようなことは起こりうる．この場合も，一時変数を用いることによって，問題を回避できる．すなわち，一時変数によって，実引数の値をすべてあらかじめ計算しておけば，値渡し引数・適用順評価と同じ結果になる．ただし，一時変数の使用がオーバヘッドとなることもあり，その使用は限定すべ

きである.

(ii) 型判定の最適化

Lisp における動的な型判定を最適化するためには,(a)実行時の型判定時間の削減と,(b)型判定そのもののコンパイル時の実施とがある.

実行時の型判定時間の削減には,より使用頻度の高いタグ判定を優先させることと,判定後の分岐の時間を最少にするための,命令列の並べ替えなどがある.後者は,局所最適化の一部として実現した.

一方,コンパイル時の型判定による動的型判定の削減のためには,ユーザの型宣言情報などをもとに対象となるデータや演算の型を型判定・型推定して同定することにより行う.

型判定・型推定は次の手順で行う:

(a) 型宣言情報から変数の型を判定

(b) 結果の型が単一の関数の型を判定

(c) 引数の型が単一の場合,その実引数の型を推定

(d) 引数の型によって,演算および結果の型が決まる汎用関数の型を推定

(e) 以上の組み合わせによる型推定

(a),(b)のように最初から型が確定する場合を型判定といい,それ以外を型推定と言う.一般に(e)組み合わせによる型推定では,(a),(b)と(d)を基本に内から外へ向かって推定していくが,これで型が決定不能のときは,(c)などにより外から内へと推定する.(b)の一種として,the による型指定も含まれる.

図5に簡単な型判定・型推定の例を示す.reverse は,配列,文字列,リストのいずれかの順番を逆転させる汎用関数であるが,演算および結果の型は,引数の型で決まる.この例では,sort も第1引数の型によって結果の型が決まる汎用関数であるから,結局,sort の引数の型判定から始めて,sort の結果,すなわち reverse の引数の型を推定し,最後に reverse の演算および結果の型がリスト型であると推定する.

このような型判定・型推定の結果,演算すべき型の範囲が限定され,それまで汎用関数の呼出しが必要であった組込み関数がインライン展開できる,といった

```
(defun l-tarai (x y z)
 (cond ((< (car x) (car y))
        (l-tarai (l-tarai (copy-tree (cdr x)) y z)
                 (l-tarai (copy-tree (cdr y)) z x)
                 (l-tarai (copy-tree (cdr z)) x y)))
       (t y)))
```

↓

```
(defun l-tarai (x y z)
 (cond ((< (car x) (car y))
    (l-tarai (progn (setq w (copy-tree (cdr x))
              (cond ((< (car w) (car y))
                     (l-tarai (l-tarai (copy-tree (cdr w)) y z)
                              (l-tarai (copy-tree (cdr y)) z w)
                              (l-tarai (copy-tree (cdr z)) w y)))
                    (t y)))
            (progn (setq w (copy-tree (cdr y))
              (cond ((< (car w) (car z))
                     (l-tarai (l-tarai (copy-tree (cdr w)) z x)
                              (l-tarai (copy-tree (cdr y)) x w)
                              (l-tarai (copy-tree (cdr z)) w z)))
                    (t z)))
           (progn (setq w (copy-tree (cdr z))
              (cond ((< (car w) (car x))
                     (l-tarai (l-tarai (copy-tree (cdr w)) x y)
                              (l-tarai (copy-tree (cdr z)) y w)
                              (l-tarai (copy-tree (cdr y)) w x)))
                    (t x)))
 (t y)))
```

図2 自己再帰展開の例
Fig. 2 An example of self-recursion expansion.

```
(defun foo (x y)
 (if () (length y) 5) y
       (list (foo x (nconc x y)))))                    ……(a)
(defun foo (x y)
 (if () (length y) 5) y
       (list (if () (length (nconc x y)) 5)(nconc x y)
             (list (foo x (nconc x y (nconc x y)))))))
                                                       ……(b)
(defun foo (x y)
 (if () (length y) 5) y
       (list (let ((w (nconc x y)))
             (if () (length w) 5) w
                 (list (foo x w)))))))                 ……(c)
(foo '(1 2 3) '(4 5 6))
  =)((1 2 3 4 5 6))                          ……(a) & (c)
  =)((1 2 3 4 5 6 4 5 6……))                            ……(b)
```

図3 副作用のある場合の自己再帰展開の例
Fig. 3 An example of self recursive expansion of function with side effect.

```
(defun baa (x y)
  (if (= x 0) 1 (1+ (baa (1- x) (baa (- x y)y)))))
```

図 4 停止性を充たさない関数の例
Fig. 4 An example of a function which does not halt.

副次的効果も期待できる.

Common Lisp 型の概念は，オブジェクトに付けられた動的な型を基本とし，それに，ユーザが与える型宣言情報を補助的に使うものであるから，Pascal や Ada などの強い型付けをもつ言語と異なり，コンパイル時に完全に決まると，というわけにはいかない．それでも，なおかつ，動的な型判定を減らすことは，汎用関数の出現頻度の高い Common Lisp の最適化コンパイラにとって，高速化の重要な手段である.

(ⅲ) 局所最適化

局所最適化は，Lcode レベルの最適化 (Local-opt1) と LAP レベルの最適化 (Local-opt 2) とに分類される.

Local-opt1 では，マシンに依存しない局所最適化を行う．主な最適化項目は，

(a) 不用命令の削除
(b) 分岐命令の最適化
(c) スタック限界チェック削減

などである．スタック限界チェック削減は，複数のスタック限界のチェックをまとめることにより行う．分岐命令の最適化は，条件の then 部が無条件分岐のとき判定条件を逆にしたり，分岐の連続を一つにまとめたりすることである.

一方，Local-opt2 は，マシンに依存した，命令語レベルの局所最適化を行う．主な最適化項目は，

(a) 不用命令の削除
(b) 命令列の並べ替え

等である．汎用大型機ではパイプライン制御が発達しており，多くの演算命令は実効的に 1 サイクルで実行できる．しかし，いったんパイプラインが乱れると実行時間は 1 サイクルより遅くなる．主なパイプラインを乱す要因としては，

① 条件判定コードの設定とその使用 (条件分岐)
② インデックスレジスタやベースレジスタの設定とその使用
③ 同一メモリ番地への値の設定とその使用

などである．HiLISP コンパイラの局所最適化では，これらの要因が生じるような命令列を，遅れの原因となる命令と結果の命令とを離すような並べ替えを行って改善する．条件分岐の場合，条件判定コードの設定

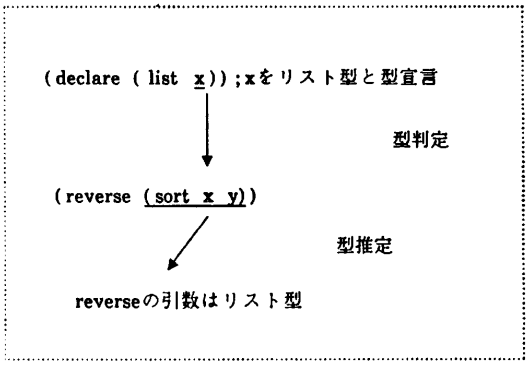


図 5 型判定・型推定の例
Fig. 5 An example of type check & type inference.

(たとえば，スタック限界チェック) とその使用 (たとえば，エラー処理への分岐) の間にその前後の別の命令を入れるような並べ替えを行う.

4. 性能評価

HiLISP コンパイラの最適化方式の性能を評価するために，Lisp コンテストプログラム[5]の中から，代表となる 12 題のプログラムを選び，その性能を測定した (表 2)．性能は，すべての最適化を効かせた場合と，次の各最適化項目を実施しなかった場合との比で求める.

(a) 自己再帰展開・ユーザ関数展開
(b) 末尾再帰のループ化
(c) 組込み関数展開
(d) 型判定・型推定
(e) 局所最適化

特に，(b)の末尾再帰のループ化と(c)の組込み関数展開とは，(a)の自己再帰展開・ユーザ関数展開の前提となるため，(b)，(c)の効果は，(a)の最適化を実施しない条件で測定した．型判定・型推定に関しては，型宣言の入れ方によって型判定・型推定の効果が変わってくる．そこで，従来 Lisp (MacLisp 系) で書かれたプログラムを参考にし，従来 Lisp プログラムで専用関数となっている部分に対応する変数の宣言をプログラムに入れて測定した.

この結果，12 題の相乗平均で，自己再帰展開・ユーザ関数展開の効果は 8%，末尾再帰のループ化の効果は 9%，組込み関数展開の効果は 157%，型判定・型推定の効果は 32%，局所最適化の効果は 6% である.

組込み関数展開の効果は当初の予想どおり，最も効果が大きい．また，型判定・型推定の効果が 32% と

表 2 Lisp コンテスト代表12題ジョブによる HiLISP コンパイラの性能評価
Table 2 Performance evaluation of HiLISP compiler for the 12 benchmarks in the Lisp contests.

Benchmark	①全最適化	②自己再帰展開・ユーザ関数展開無	③末尾再帰ループ化無＋②	④組込関数展開無＋②	⑤型推定・型判定無	⑥局所最適化無	比率				
							②/①	③/②	④/②	⑤/①	⑥/①
tarai-5	129ms	213ms	312ms	932ms	360ms	151ms	1.65	1.46	4.38	2.79	1.17
list-tarai	23.5	25.9	30.1	64.9	28.7	24.6	1.10	1.16	2.51	1.22	1.05
string-tarai	137	140	143	199	176	138	1.02	1.02	1.42	1.28	1.01
flonum-tarai	11.7	13.8	17.9	67.4	22.0	11.7	1.18	1.30	4.88	1.88	1.00
Bubble-sort	3.32	3.24	3.28	13.73	4.77	3.71	0.98	1.01	4.24	1.44	1.12
Seq-100	3.53	3.50	3.59	4.90	3.67	3.58	0.99	1.11	1.40	1.04	1.01
BITA-6	0.86	0.85	0.85	3.79	0.92	0.92	0.99	1.00	4.46	1.07	1.07
Sort-100	0.12	0.12	0.12	0.12	0.12	0.12	1.00	1.00	1.00	1.00	1.00
TPU-6	113.4	114.8	119.3	218.8	124.7	118.2	1.01	1.04	1.91	1.10	1.04
Prolog-sort	18.7	22.9	23.0	58.7	26.6	19.6	1.22	1.00	2.56	1.42	1.05
Diff-5	19.7	19.5	20.1	48.8	22.7	21.5	0.99	1.03	2.50	1.15	1.09
Boyer	751	770	847	2624	873	860	1.03	1.10	3.41	1.16	1.15
相乗平均	—	—	—	—	—	—	1.08	1.09	2.57	1.32	1.06

M680H

比較的大きいのは，型判定・型推定の結果，単に型判定コードが除去されるだけでなく，関数がインラインに展開されたり，またその結果としてさらに，スタックへの余分な退避（一般的には GC のために必要）が無くなることなどによる，複合的な効果である．

自己再帰展開・ユーザ関数展開は，末尾再帰のループ化と同程度くらいの効果があるが，これは，Lisp では関数呼出しが多いことを示している．

局所最適化の効果は，当初の予想よりは大きくない．これは，Lcode を展開した命令列のレベルで，あらかじめ，できるかぎりパイプラインを意識した命令列にしておいたためと考えられる．

最適化に要するオーバヘッドとして，オブジェクトサイズの増加，コンパイラサイズの増加，コンパイル時間の増大などがあるが，試作した HiLISP コンパイラでは，オブジェクトサイズ，コンパイラサイズの増加はいずれも約2割と比較的小さいが，コンパイル時間が約3倍増と大きい．（ただし，コンパイル時間は今後，改善の見通しがある．）

なお，HiLISP コンパイラオブジェクトの絶対性能に関しては，Lisp コンテストで最も高速な Lisp 処理系[6]と同一マシン上で比較して，HiLISP の方がより速いという結果が出ている[10]．

5. おわりに

Common Lisp では，汎用関数化の徹底など実行性能低下の要因がいくつかあるが，汎用機でも十分高速の Lisp 処理系が作れること，および，Lisp は Pascal や C などの手続き型言語とくらべても，処理系の作り方さえ工夫すれば，同程度の実行性能が出せる（付録参照）こと，とが言える．

ここで提案した最適化方式は，Lisp コンパイラ，特に Common Lisp コンパイラに有効な方式であり，今後多くの Lisp コンパイラで用いられるようになるであろう．

今回，設計し試作した HiLISP コンパイラは，整数における bignum と，多値などを含まないサブセットである．これらの追加により性能は，若干の低下が見られるものの大幅な低下にはならない見通しもある．今回試作した HiLISP をもとに，ほぼフルセットの Common Lisp を製品として開発しており，今後，フルセットの性能も評価していく予定である．

参考文献

1) Steele, G. L. Jr.: *Common Lisp: the Language*, p. 645, Digital Press (1984).
2) Allen, J. R.: *Anatomy of Lisp*, McGraw-Hill (1978).
3) 高田綾子，安村通晃，青島利久：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回大会論文集，pp. 97-100 (1986).
4) 湯浦克彦，安村通晃：HiLISP インタプリタの高速処理方式，情報処理学会記号処理研究会資料，No. 40-5 (1987).
5) 奥乃 博：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

処理研究会資料, 28 (1984).
6) Okuno, H.: The Report of the Third Lisp Contest and the First Prolog Contest, 情報処理学会研究報告, Vol. 85, No. 30 (1985).
7) 安村通晃, 高田綾子, 湯浦克彦: 再帰プログラムにおけるプログラム変換技法, 理研シンポジウム「関数プログラミング」, FP-87-03, pp. 20-27 (1987).
8) Burstall, R. M. and Darlington, J.: A Transformation System for Developing Recursive Programs, *J. ACM*, Vol. 24, No. 1, pp. 44-67 (1977).
9) 日本電子工業振興協会編: マイクロコンピュータに関する調査報告書 [II]—Common Lisp—, 61-A-235 [II] (1986).
10) 武市宣之, 安村通晃, 湯浦克彦, 森田和夫: 知識処理用言語 HiLISP の高速化方式, 日立評論, Vol. 69, No. 3, pp. 13-16 (1987).
11) Manna, Z.: *Mathematical Theory of Computation*, p. 448, McGraw-Hill (1974).
12) Brooks, R. A., Gabriel, R. P. and Steele, G. L. Jr.: An Optimizing Compiler for Lexically Scoped LISP, *Conf. Record of the 1982 ACM Symp. on Lisp and Functional Programming*, pp. 261-274 (1982).
13) 近山 隆: UtiLisp システムの開発, 情報処理学会論文誌, Vol. 24, No. 3, pp. 599-604 (1983).
14) Hagiya, M. and Yuasa, T.: Implementation of Kyoto Common Lisp, 日本ソフトウェア科学会第1回大会論文集, pp. 65-68 (1984).
15) Wholey, S. and Fahlman, S. E.: The Design of an Instruction Set for Common Lisp, *Conf. Record of the 1984 ACM Symp. on Lisp and Functional Programming*, pp. 150-158 (1984).
16) Yuasa, T. and Hagiya, M.: Implementation of Kyoto Common Lisp, 情報処理学会記号処理研究会資料, No. 34-1 (1985).

付表 Lisp と Pascal の性能比較
Appendix Performance comparison of Lisp with Pascal.

#	Benchmark	HiLISP	Pascal 8000	比率 (P/L)
1	Tak	41. ms	69. ms	1.68
2	Flo-tarai-4	22.	20.	0.91
3	Qsort-100 (list version)	37.	66.	1.8
4	Bubble-50 (array version)	5.3	2.6	0.49
5	Nqueen-8 (list version)	95.	304.	3.2
6	Nqueen-8 (array version)	53.	25.	0.47
	相乗平均	—	—	1.15

M680H

付録 Lisp と Pascal の性能比較

付表に6題のベンチマークに対する HiLISP と Pascal 8000 の性能比較を示すが, 相乗平均では 1.15 倍でほぼ同程度の性能である. 自己再帰関数呼出しが多く, 末尾再帰のループ化や自己再帰展開の効く tak では Lisp が速いが, 一般的には Qsort-100, Nqueen-8 などにみられるように, list 版で Lisp が速く, array 版で Pascal が速い.

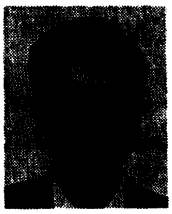
(昭和62年6月26日受付)
(昭和62年9月 9 日採録)

安村 通晃 (正会員)

1947年 (昭和22年) 生. 1971年東京大学理学部物理学科卒業. 1973年同大学理学系大学院修士課程修了. 1975年より2年間カリフォルニア大学ロスアンゼルス校 (UCLA) に留学. 1978年東京大学理学系大学院博士課程満期退学. 同年, (株)日立製作所中央研究所に入所. 現在, 主任研究員. 主たる研究分野は, プログラミング言語設計論, コンパイラ作成論, プログラミング方法論, および, 並列アルゴリズム等. 著書, 「プログラミング・セミナー」(共著), 「Pascal の標準化」(共訳著), 「プログラム変換」(共著). ACM, IEEE, 日本ソフトウェア科学会各会員.

高田 綾子 (正会員)

1960年生. 1982年東京農工大学工学部数理情報工学科卒業. 同年, (株)日立製作所中央研究所入社. 第8部勤務. 日本語出力システム, Lisp, 知識ベース利用等の研究・開発に従事.



青島 利久 (正会員)

昭和25年生. 昭和44年県立静岡工業高校電気科卒業. 同年(株)日立製作所中央研究所入社. 昭和49年日立京浜工業専門学校電子工学科研究課程修了. 以来日立中研において, グラフィックスを使用した LSI-CAD, 高速2次元・高精細3次元グラフィック端末の開発研究に従事. 現在, LISP 言語のコンパイラ, プログラム支援環境の開発に従事.