Common Lisp インタプリタ制御の高速化手法と実験評価†

湯浦克彦†† 安村通晃††

Common Lisp に準拠した高速の Lisp システムである HiLISP のインタプリタ制御方式について述べる．Common Lisp は，インタプリタとコンパイルされた関数との動作の一致のために静的スコープやクロージャ機能をもつ標準化指向の Lisp 言語であり，かつ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備えている．Lisp では，インタプリタの対話環境でプログラミングを効率的にすすめることができるが，静的スコープやクロージャ機能を含めて実現することによって，インタプリタの負荷が大幅に増大してしまうと考えられていた．HiLISP においては，通常の変数結合をスタック上の深い結合で行い，静的スコープを高速に実現した．一方，クロージャ機能に対しては，適宜ヒープに値を保存する処理と“見えないポインタ”を用いて値を共通に参照する方式を開発し，これを実現した．関数制御では，多値返答の拡張が問題となったが，単値と多値とで関数復帰先を分けて別経路で処理する方式により，負荷を避けることができた．HiLISP 処理系は HITAC M シリーズ汎用大型機上に作成した．さらに変数管理法の方式評価のために，従来仕様による浅い結合方式の HiLISP および Common Lisp 仕様の従来実現方式である a リスト方式の HiLISP を作成した．ベンチマークによる性能の比較評価により，本方式での改善効果を明らかにした．

1. まえがき

Lisp は，古くより様々の方言と処理系が作られ，知識処理システム，設計自動化，自然言語解析等の研究用の言語として使用されてきた．最近 Common Lisp[1] による言語仕様の標準化の動きがあり，応用ソフトウェアの互換性が保証される，機能が豊富であるなどの期待からその利用が広まってきている[2]．

Common Lisp では，従来の Lisp 言語の問題であったインタプリタとコンパイルされた関数での変数規則の不一致を解決するため，(a)静的スコープ，(b)クロージャ機能を定めている．また記述性の向上を図るために(c)多値返答や(d)局所関数などの制御構造を拡張している．応用の範囲を拡大するためデータ型や関数も増補し，さらに，(e)キーワード・パラメータや(f)汎用関数の考え方を取り入れて個々の関数の機能を充実させている．

Lisp 言語では，プログラムの実行方式としてインタプリタとコンパイラを用いる．インタプリタでは，ソース・プログラムのままの関数を直接解釈実行する．そこではエディタ，デバガ，ステッパなどの対話型のプログラミング支援ツールを，実行しながら利用してデバグを効率的にすすめることができる．十分完成されたプログラムはコンパイルして，コンパイルされた関数として実行することができる．Lisp 言語においてはプロトタイプとして複雑な制御やデータ構造を扱うプログラムを作成する場合が多いので，インタプリタで実行する割合が高く，インタプリタでの高速実行への期待は大きい[2]．

Common Lisp のインタプリタの実現では，先に述べた(a)～(f)の拡張機能の実現が課題となる．しかし，これらはいずれも Lisp の基本制御に関わり，既に作成された Common Lisp インタプリタでは，従来の小規模な言語仕様による処理系[3] に比べて，大幅な性能劣化が免れられなかった[4]．

以上の背景をもとに，報告者らは，Common Lisp に準拠した HiLISP (High Performance List Processor) のインタプリタ／コンパイラを設計し，HITAC M シリーズ汎用計算機上に処理系を作成した．設計の目標としては，コンパイルされた関数での性能の向上[5] とともに，インタプリタでの高速性を確保することを取り上げた．

本稿では，HiLISP インタプリタ高速化のために開発した実現方式のうち，(a)静的スコープ，(b)クロージャ機能を実現する変数管理方式および(c)多値返答を実現する関数制御方式について述べる．さらに作成した処理系での変数管理方式の評価について述べる．

2. Common Lisp の機能と HiLISP 実現の方針

2.1 Common Lisp の変数管理機能

Common Lisp においては，変数をはじめとする環境の参照規則をスコープとエクステントという概念を

† Some High Speed Methods and Their Experimental Evaluations for Common Lisp Interpreters by KATSUHIKO YUURA and MICHIAKI YASUMURA (Central Research Laboratory, Hitachi, Ltd.).

†† (株)日立製作所中央研究所

```
(setq x 5)
(defun f (x) (g))
(defun g () x )
(f 7) --> { インタプリタでは          7
          { コンパイルされた関数では  5
```

図 1 従来 Lisp 言語の問題
Fig. 1 A problem of conventional Lisp dialects

用いて規定している．スコープはソース・プログラムのテキスト上文脈的に確立される環境の有効な範囲を示すものであり，エクステントはプログラムの動作上時間的に確立される環境の寿命を示すものである．変数の結合については，静的スコープかつ無限エクステントをもつことが規定されている．

従来の Lisp 言語では，変数の参照規則を，コンパイルされた関数では静的スコープを基本として実現しているのに対し，インタプリタでは動的エクステントに従っている例がほとんどであった．このため，コンパイルした場合に異なる値を返答する場合があり，使用上問題となっていた．従来の Lisp 言語におけるインタプリタとコンパイルされた関数での仕様の違いの例を図1に示す．式 (f 7)をインタプリタで評価する場合，関数 f の呼出し時に起こる変数 x と値7の結合が，インタプリタ内で関数 f の本体である式 (g)の評価が終了するまでの寿命をもつことになる．ところが，関数 f および関数 g をコンパイルすると，関数 f の変数 x は f のスコープで有効な変数として解釈されるのに対し，g では f で結合された値を参照せず，グローバルな値を参照することになる．

以上の問題を解決するため，Common Lisp ではインタプリタにおいても変数参照に静的スコープを採用している．なお，動的エクステントでの参照を行う動的変数も用いることができるが，これは変数に special 宣言を付加し，区別して使用する．

クロージャ (closure) とは，関数をその定義場所の静的環境とともに保存する機能である．Common Lisp では，特殊形式 function などの実行地点の静的環境，つまり，静的スコープに含まれるすべての変数，局所関数などを保存し，無限エクステントで参照可能としており，これにより Lisp 言語の古くからの課題である funarg 問題[6] を全面的に解決している．

クロージャ機能を用いて，動的エクステントの外の変数を参照する例を図2 (a) に示す．この例で，関数 f の本体に現れる変数 x は，funcall による f の起動時点で元々の場所での定義は解除されているが，関数 f の環境として参照することができる．また，クロージャに取り込まれた変数は，クロージャを定義した場所からも，起動されたクロージャ内からも共通に参照，更新されなければならない．こうしたクロージャの例を図2 (b) に示す．関数 f の静的環境としての変数 x は，関数 f の定義後に12より7に値が更新されている．関数 f の起動時に変数 x の値は，12ではなく7として参照しなければならない．

```
(let ((x 12)) (setq f #'(lambda () (1+ x))))
(funcall f) --> 13
```

(a) 動的エクステント外での参照

```
(let ((x 12)) (setq f #'(lambda () (1+ x)))
              (setq x 7))
(funcall f) --> 8
```

(b) クロージャ内外よりの共通のアクセス

図 2 クロージャ機能
Fig. 2 Closure.

2.2 Common Lisp の多値返答機能

従来の Lisp 言語では，関数は唯一の値 (以下，単値と称す) を返答したが，Common Lisp では，多値 (0個ないし複数個の値) を返答するように拡張された．

Common Lisp の組込関数には，従来通り単値を返答する関数のほかに多値を返答する関数があり，これらを組み合せて定義するユーザ関数では，単値でも多値でも返答できるようにしなければならない．一方，呼出し側での返答値の利用は3つのタイプがある．第1のタイプは末尾呼出しであり，返答値をそのまま呼出し側関数の値とする．第2のタイプは通常の引数評価呼出しであり，返答値の数にかかわらず第1番目の値 (値の数0のときは既定値 nil) を用い，第3のタイプの多値呼出しではすべての値を用いる．

2.3 HiLISP 実現の方針

2.1節，2.2節で述べた機能を含む HiLISP インタプリタの設計を以下の方針で進めた[7]．

(1) メモリ構成

ヒープとスタックをもつ．スタックは制御スタックと結合スタックの2本である．制御スタックには，インタプリタや関数の制御情報のほか，静的変数の結合に関する情報を置く．結合スタックには，動的変数の結合に関する情報を置く．

(2) 関数インタフェース

制御スタックのフレームを1つの関数処理に対応して使用し，入力引数はスタック渡し，出力結果はレジ

スタで返答する.

3. 変数管理方式

3.1 従来方式での問題点

従来仕様の処理系で採用されていた浅い結合 (shallow binding)[3] では, Lisp 環境全体として変数名に対応した特定の記憶場所を唯一設け, そこに値を直接設定するので, 各関数のスコープごとに変数を扱うことができない. 深い結合 (deep binding) は, 変数名と値との結合表を生成する方式であり, 結合表をスコープごとに設けて使い分けることにより静的スコープを実現することができる.

深い結合の実現方式としては, ヒープに変数名と値の対をリストで保持するaリスト方式[8] が最も一般的である. aリスト方式の概要を図3(a)に示す. aリスト方式は, 後で述べるクロージャ機能の実現には適しているが, 変数結合が生ずるたびに cons セルを消費することや変数参照のたびにリストの探索が必要であり, 効率上好ましくない.

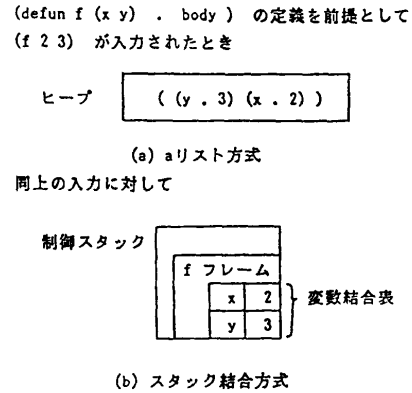


図3 静的変数の深い結合
Fig. 3 Deep binding methods for lexical scope.

3.2 スタック結合による静的スコープの実現

HiLISP では, 変数名と値との結合表をスタック上の配列として実現するスタック結合方式を採用した. スタック結合方式の概要を図3(b)に示す. スタック結合方式では, 変数結合時にヒープより cons セルをアロケートする負荷が避けられ, ガーベジ・コレクション (GC) の頻度を下げることもできる. 変数参照時には, リストを手操ることなく配列の順次検索で済ますことができる.

3.3 見えないポインタを用いたクロージャの実現

HiLISP においては, スタック結合方式のもとに前述のクロージャ機能を実現する方式を開発した. 図4にその概要を示す.

変数は無限エクステントであるが, クロージャ機能を用いない場合には動的エクステントの内でもあるので, スタック上に結合情報を置いて参照することができる. 特殊形式 function などによって動的エクステントの外で変数を参照する可能性がある場合には, この場合にのみ, 値をヒープ上に移動し, クロージャとして保存することにする. 特殊形式 function などで保存する変数と値は, 同時に保存する関数の実行のためのものである. そこで, 関数本体より参照される変数を選んで保存すれば効率的であるが, 関数本体にマクロ形式が現れると関数起動時にはあらゆる変数の参照がありうることになるので, クロージャには, 定義されているすべての変数を保存しておく必要がある. 特殊形式 function のほか, 局所関数を定義, 起動する特殊形式 flet などでも変数の保存が必要であり, クロージャを用いて実現した[9].

スタックよりヒープに値を移動することに伴って, 図2(b)に示した関数の定義場所での値と関数本体での値の一致を考慮しなければならない. そこで, スタックの変数結合表の元の値の位置には移動先への "見えないポインタ (invisible pointer)" を設定した. またヒープのクロージャにも変数

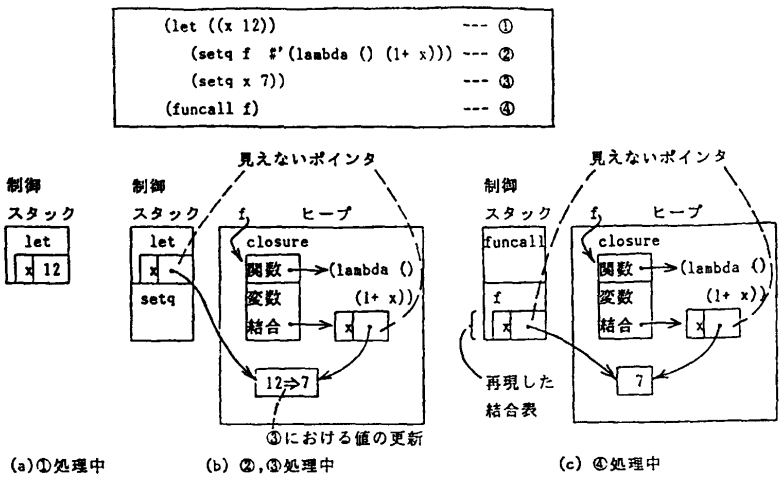


図4 見えないポインタの利用によるクロージャの実現
Fig. 4 An implementation of closure using invisible pointers.

結合表を生成し，値の位置に同様の見えないポインタを設定した．見えないポインタとは参照すると自動的にそのポイント先を参照するようにしたものである．これにより，関数定義場所で静的変数をスタック上の変数結合表から参照し，値を更新した場合には，同時にクロージャに保存した値も更新されることになり（図4（b）），矛盾を生じない．クロージャを呼び出す時には，ヒープに保存した見えないポインタ付きの変数結合表をスタック上に再現し（図4（c）），スタックを介して高速に参照することができる．

4. 関数制御方式

4.1 従来方式での問題点

多値返答を含む関数復帰では，多値を格納する場所とともに値の数を呼出し側に知らせる処理が必要と考えられた．そこで，従来の Common Lisp 処理系[5]では図5（a）に示すように値をスタックに返答し，2つのスタック・ポインタによって値の数を連絡していた．この方式によれば単値も多値も同じ形式で保持され，通常の引数評価呼出しにおいても，値の先頭を示すスタック・ポインタより値を1つ得る手順で一様に処理することができる．しかし，HiLISP 設計の基本方針として挙げたレジスタによる返答方式と比べると，関数復帰ごとにスタック操作が必須となり，特に汎用計算機上の実現では負荷増が問題となる．

一方，基本方針に沿って単値の場合にはそのままレジスタで返答するとすれば，図5（b）に示すように，多値の場合には，多値を専用の配列等に保持しそれへのポインタを単値の場合と同一のレジスタに設定する方式が考えられる．この方式では，引数評価呼出しの呼出し側の処理で，返答レジスタに単値そのものが設定されているか多値専用配列が設定されているかの判定が必要となってしまう．

4.2 HiLISP における多値経路振分け方式

HiLISP においては，図5（b）の保持形式を基本と

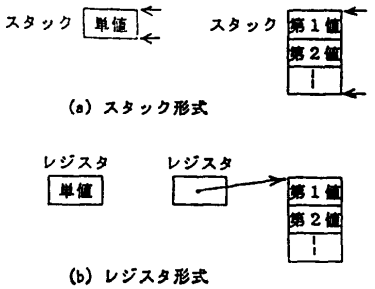


図5 単値／多値の保持形式
Fig. 5 Representations of single/multiple values.

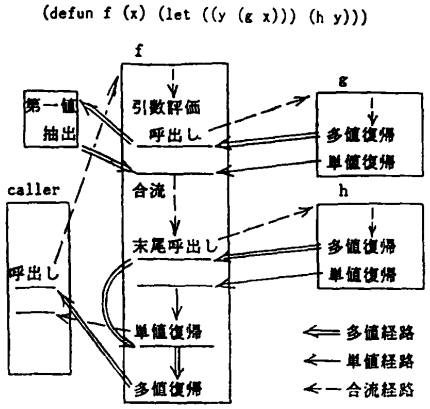


図6 多値経路振分け方式
Fig. 6 A return method for single/mutiple values having different paths.

して，単値と多値で関数復帰時の戻りアドレスを分けて，単値／多値の判定を省略した．今回開発した多値経路振分け方式の動作を図6に示す．

引数評価呼出しでは，多値で戻ると第1値を抽出し返答レジスタにセットして単値の処理と合流する．末尾呼出しでは，単値で戻るとさらにそこから単値戻りアドレスへ，また多値で戻るとさらにそこから多値戻りアドレスへと経路を分けて，単値を返答しているか多値を返答しているかの情報を伝播させる．一般に戻りアドレスを分ける方式を採ると，経路が次々に生成され，プログラム・サイズが膨大となる危険性がある．しかし，多値は生成されても呼出し側の処理でいったん参照されると必ず単値化されるので，現在単値を処理しているか多値を処理しているかの高々2つの経路をもてばよいことになる．

多値経路振分け方式による関数復帰処理を，従来の単値のみを返答する仕様の関数復帰処理と比較する．多値が返答された場合には，1ないし2回の分岐増および第1値の抽出処理などがあり負荷は増加するが，単値が返答された場合には，レジスタに値を保持したまま呼出し側に戻るだけであり，これは従来仕様での関数復帰処理と同じシーケンスである．Common Lisp においても大多数の関数復帰は単値によると考えられるので，総合的にはほとんど負荷増は生じないことになる．

5. 変数管理の方式評価

5.1 方式比較実験プログラム

スタック結合方式による本来の HiLISP のほか，変数管理をaリスト方式に改造した HiLISP（aリス

ト版）および浅い結合方式に改造した HiLISP（浅い結合版）をMシリーズ計算機上に作成し，性能を比較した．

（1） aリスト版

静的変数結合を行う部分の内容をスタックへの設定よりaリストの生成へ変更し，静的変数参照を行う部分の内容をスタック上の配列探索よりaリスト探索へ変更した．特殊形式 function などでのクロージャの生成では，aリストをそのまま変数結合表として保存する．以上のほか，aリストの GC に関する処理などの改造を行った．aリストの生成のコーディングでは，cons セル割当てをインラインで行い，機械語レベルでの最適化の程度を保つように留意した．

（2） 浅い結合版

HiLISP では，special 宣言が付加された変数（動的変数）の結合を浅い結合で実現している．HiLISP における浅い結合は，旧値を結合スタックへ退避しつつ値を変数シンボルの value 部へ設定し，結合解除するときは退避した旧値を復帰させている．変数参照は変数シンボルの value 部をロードすることにより行う．

浅い結合版では，HiLISP の静的変数結合実現部を動的結合実現部と同一の処理に置き換え，静的変数参照実現部を動的変数参照実現部と同一の処理に置き換えて，すべて動的結合，動的参照となるようにした．特殊形式 function でのクロージャ生成では，静的変数なしのシステムということで，変数結合表は無条件に nil とした．以上のほか，変数結合処理時に変数の special 宣言の有無の判定が不要になるのでこれを省略するなどの改造を行った．

5.2 基本操作の性能評価

（1） 変数参照

HiLISP でのスタック結合方式（以下，スタック方式と略す），aリスト方式および浅い結合方式について，変数参照に用する時間の測定結果を図7に示す．

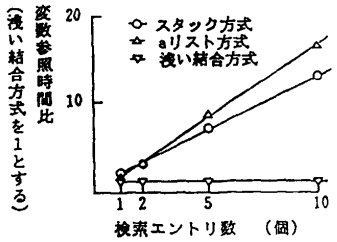


図7 検索エントリ数と変数参照時間
Fig. 7 Numbers of searching entries of variables and time ratio of references of variables.

スタック方式およびaリスト方式では当該の変数をスタック上およびaリスト上の変数表で検索するので，参照時間は，当該の変数が見つかるまでの変数表の長さ（検索エントリ数）に比例する．浅い結合方式は一定で高速である．スタック方式と浅い結合方式は，検索エントリ数が少ない範囲では差は小さいが，検索エントリ数が多くなるとスタック方式が高速である．これは，変数表検索がaリストを手操りながらの検索から配列の順次検索に改良されている効果と考えられる．

（2） 変数結合

スタック方式，aリスト方式および浅い結合方式における変数結合時間の測定結果を表1に示す．浅い結合方式では結合解除に負荷を生ずるので結合解除の時間を含むように測定した．これによると，ラムダ式と let と両方で，スタック方式と浅い結合方式とはほぼ同等であるのに対し，aリスト方式はこれらに比べて約 1.4 倍遅いということになる．スタック方式と浅い結合方式とでは，スタック方式の変数表作成や special 宣言の有無判定の処理の負荷と浅い結合方式の結合解除の負荷が相殺されたと見ることができる．スタック方式とaリスト方式では，aリスト方式での cons セルの割当ての負荷により差が生じたと考えられる．

5.3 ベンチマーク・プログラムでの比較と分析

Lisp コンテスト[10]の代表 10 題のベンチマーク・プログラムによる，スタック方式，aリスト方式および浅い結合方式のインタプリタ性能の測定結果を表2に示す．浅い結合方式は静的スコープを実現しないが，これらのベンチマークでは誤動作しない．スタック方式での実行時間を1とすると，平均でaリスト方式が 1.07，浅い結合方式が 0.90 であった．#7のプログラムでは特殊形式 function の実行回数が多く，クロージャ生成の負荷の差が表面化したものと考えられる．浅い結合方式のクロージャでは変数を保存しないので，部分的には動作が異なることになる．なお，この測定では GC 時間を含んでいない．aリスト方式では，変数結合に cons セルを消費するので他の2方式

表1 変数結合時間の比
Table 1 Time ratio of bindings of variables.

形式	aリスト方式	スタック方式	浅い結合方式
lambda 式	1.4	1	1.1
特殊形式 let	2.0	1.4	1.3

表 2 ベンチマークによる実行時間比較
Table 2 Time ratio for benchmark programs.

#	ベンチマーク	aリスト方式	スタック結合方式	浅い結合方式
1	tarai-5	1.16	1	0.88
2	list-tarai-4	1.12	1	0.90
3	string-tarai-4	1.08	1	0.94
4	flonum-tarai-4	1.13	1	0.89
5	bubble-5	1.01	1	0.92
6	sequence-100	1.03	1	0.99
7	bita-6	0.88	1	0.76
8	tpu-6	1.11	1	0.85
9	diff-5	1.11	1	0.93
10	Boyer	1.10	1	0.93
	平均	1.07	1	0.90
	(#7を除く平均)	(1.09)	(1)	(0.91)

on HITAC M 680 H

に比べて GC の生起頻度が高く，GC 時間を含む実質性能ではさらに差が開くと考えられる．

6. むすび

インタプリタとコンパイルされた関数での動作を同一にするために，Common Lisp で定められた静的スコープ，クロージャ機能をサポートしようとすると，インタプリタにかかる負荷は大きく，従来のaリスト方式では GC の頻度も増加して性能上問題は大きい．性能の劣化は変数参照速度の劣化と変数結合速度の劣化がある．このうち，変数結合速度については本稿で提案する“見えないポインタ”を用いたスタック結合方式の採用によりほぼ回復が可能である．インタプリタ全体の性能劣化は半ば（ベンチマークで17%のうち7%）に抑えられ，さらにaリスト方式特有の GC 頻度の増加も抑制されるので，インタプリタの実用性をおおいに高めることができる．Common Lisp での多値返答の拡張も，本稿で提案の多値経路振分け方式によりほとんど負荷の増加なく実現することができる．残された課題としては，上記の GC 抑制の効果の精密な評価と変数参照速度の改善がある．

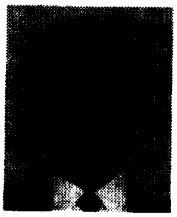
参考文献

1) Steele, Guy L., Jr.: *Common Lisp the Language*, Digital Press (1984).
2) 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］, pp. 85-125 (1986).
3) 近山：Utilisp システムの開発，情報処理学会論文誌，Vol. 24, No. 5, pp. 599-604 (1983).
4) Okuno, H.: The Report of the Third Lisp Contest and the First Prolog Contest, 情報処理学会記号処理研究会資料，33-4 (1985).
5) 安村ほか：Common Lisp 用最適化コンパイラの設計と試作，情報処理学会論文誌，Vol. 28, No. 11, pp. 1169-1176 (1987).
6) 寺島：LISP—その発展の方向，情報処理，Vol. 26, No. 7, pp. 711-720 (1985).
7) 武市ほか：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).
8) 湯浅ほか：Kyoto Common Lisp の実現，情報処理学会記号処理研究会資料，34-1 (1985).
9) 湯浦ほか：HiLISP インタプリタの高速制御方式，情報処理学会記号処理研究会資料，40-5 (1987).
10) 奥野：第3回 LISP コンテストと第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，28-4 (1984).

(昭和63年1月25日受付)
(平成元年4月11日採録)

湯浦 克彦（正会員）

1955年生．1978年東京大学工学部精密機械工学科卒業．1980年同大学工学系大学院修士課程修了．同年より(株)日立製作所中央研究所に勤務．Lisp を中心とするプログラミング言語の研究のほか，自然言語処理，知識処理プログラミング技法等の研究に従事．日本ソフトウェア科学会会員．



安村 通晃（正会員）

1947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入所．現在，同所主任研究員．理学博士．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，並列ソフトウェア等．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），等．日本ソフトウェア科学会，ACM, IEEE Computer Society 各会員．