記　号　処　理 78－3
（1995. 3. 24）

PHLの新コンパイラ

寺島 元章 *　山本 洋司 *
古川 敦司 *　渡辺 美苗 **

* 電気通信大学大学院　情報システム学研究科
** 電気通信大学　情報工学科

概要

本稿ではLispプログラムをCのソースプログラムに変換する高度な可搬性を有するPHLコンパイラの設計方針とその機能について述べる．生成されたCプログラムは対象計算機のCコンパイラ(gcc)がオブジェクトコードに直してPHLと結合するという点でPHLは完全に計算機独立な処理系である．また，PHLは豊富なデータ型を有し，それらデータ表現も対象計算機のアーキテクチャに依存しないような方式を採用している．こうしたデータオブジェクトは256MBを超えるような大容量記憶空間に配置可能である．

Implementation of a new PHL Compiler

Motoaki Terashima *　Hiroshi Yamamoto *
Atsushi Furukawa *　Minae Watanabe **

*Graduate School of Information Systems
** Department of Computer Science

University of Electro–Communications
1–5–1 Chofugaoka, Chofu-Shi, Tokyo 182 Japan

Abstract

The design and implementation of a portable PHL compiler which translates Lisp programs into equivalent C source programs are described. The translated C programs are compiled by the C (gcc) compiler of a target machine, and then their object codes are linked to PHL. Therefore, PHL is a machine-independent Lisp system. PHL is also characterized by the existence of many data types, and their data representation is designed to be free from specific machine architectures. A great amount of storage more than 256 MB can be allocated to their data objects.

1 序

当研究室では数年前から PHL(Portable Hashed Lisp) と名付けたリスト処理系を作成している. PHL は Lisp の一方言で, 当初は Common Lisp [1] (以下, CL と呼ぶ) に準拠した仕様で設計され, その処理系 (インタプリタ) も稼働した. その後, Standard LISP[2] (以下, SLISP と呼ぶ) との諸機能の融合を目指して PHL の仕様を見直し, 新仕様の PHL 処理系 (インタプリタ [3]) も種々の計算機上で稼働している. 今回は処理速度の向上のために対象計算機非依存のコンパイラを作成したので, その設計方針と実装, 実行速度等の評価について述べる.

PHL は対象計算機が自由に選択できるという可搬性の実現のため Lisp と C 言語で記述されている. そこで, PHL コンパイラは Lisp プログラムから対応する C のソースプログラムを生成する変換系の役目だけにし, それを対象計算機の C コンパイラ (gcc) がオブジェクトコードに直して PHL のオブジェクトと結合するという, 完全に計算機独立な処理系にする構想で作られた. このため Lisp プログラムの翻訳実行は一旦 PHL 処理系を出て「新たな処理系」で実行するという方式になる. これは対話的機能の放棄になるが, 翻訳作業は数式処理プログラムに見られるように完成度の高い洗練されたプログラム群に対していわば恒久的な目的で行なわれることや PHL インタプリタの高速化も実現されていることなどから, こうしたバッチ処理的な作業でも問題ないと考えている. Lisp プログラム処理の高速化が対象計算機の C コンパイラと make コマンドだけで実現できるという完全に計算機独立な処理系が構築できるという点を重視したのである.

PHL の特徴はこうしたコンパイラ指向とともに豊富なデータ型の存在にある. それらデータ表現も対象計算機のアーキテクチャに依存しないような方式を採用している. また, こうしたデータオブジェクトは 256MB を超えるような大容量記憶空間に配置可能である. それらはガーベッジコレクション (以下, GC と呼ぶ) により効率的に管理される. これらの諸機能は多様で大容量データを扱う最近の Lisp 応用プログラムの処理を強力に支援する.

2 PHL

2.1 仕様

PHL の仕様は一言で言うと CL と SLISP の折衷案になっている[1]. 具体的には,

1. SLISP で動作するプログラムは PHL 処理系で実行可能で, かつ同じ動作をする
2. SLISP 非依存で記述された PHL プログラムはそのまま CL 処理系でも実行可能で, かつ同じ動作をする

である.

2.2 データ型とその表現

PHL のデータ型は, スカラ型と非スカラ型 (構造をもつ型) とに大別される. スカラ型データには, シンボル, 数値, 文字がある. 数値には整数と浮動小数点数がある. シンボルと整数の長さには制限はなく, 表記がすべて意味をもつ. 整数は短整数 (絶対値が 1 億未満の埋め込み即値表現整数) と長整数 (bignum), 浮動小数点数は短実数 (埋め込み即値表現実数) と任意精度浮動小数点数 (bigfloatnum) とからなるが, これは埋め込み即値表現数値を (データ格納領域が存在しないアドレスとして) 効率良く具現するための公知の技法である. 任意精度浮動小数点数はまだ具現されていない. 非スカラ型データには, コンス, ベクトル, 配列がある. ベクトルと配列の成分や成分の個数に特別な制約はない. 文字列は (その成分型が文字であるという) ベクトルの一種である. 配列やベクトルはブロックと呼ばれる処理系が提供する構造体として具現される.

PHL のデータは 1 語 (32 ビット) を用いて表現される. 1 語はタグ部とアドレス (データ) 部とに分けられる. PHL で採用したタグ方式はいわゆるポインタタグと呼ばれるもので, そのデータを指す側 (ポインタ) がその型を識別する情報をもつ. 各語の最上位ビット (MSB) はデータグループの識別に使

[1]CL は SLISP のスーパーセットのように見られるが, 実際に SLISP で記述されたプログラムがすべて CL 準拠の処理系で正しく動作するわけではない. このため, REDUCE3[4] の核言語である SLISP を CL 準拠に書き直すプロジェクトがかなり以前から米英で進行中である.

用される．その1つのグループはAグループと呼ばれ，CONSデータ，シンボル，ブロック，長整数がこれに属す．Aグループのタグは MSB [2] を含めて3ビットである．残りの29ビットでデータ格納領域上に置かれたそれら実体の先頭(語)を指す．これをアドレス部と呼ぶ．現在主流となったバイトアドレシングの採用を前提にすると，アドレス部の下位2ビットは常に00となる．この2ビットはGCで使用される．アドレス部の29ビットがすべて活用できるかどうかは処理系による[3]．

他のグループはDグループと呼ばれ，短実数，短整数，文字の各即値表現データがこれに属す．タグ中の1ビットをGCのマーキングビットに使用するため，DグループのタグはMSBを含めて4ビットとなる(図1参照)．タグを除いた28ビットがデータ部であり，即値表現と言われるようにそれらの値が直に置かれる．

当然のこととして，CONSデータの参照はマスキング演算なしに(高速に)行なうことができるようにタグが設定される．ベクトルや配列の属性(型や成分の個数など)はブロックの先頭の語(block-header)が保持する．これはいわばデータタグ的な扱いとなるが，これらの型は他の型ほど頻繁に参照されることはないので，全体としての実行速度にさほどの影響を与えないとの考慮からである．文字列の終端はC言語仕様と同じに零の値となる．シンボルは，値，関数定義，属性リストと文字列のための領域から構成される．また必要に応じてシステム用の1語が確保される．図1はSONY RISC-Newsワークステーション上のPHL処理系のデータ表現を示したものである．なお，図中の0と1はそのアーキテクチャから要請された値である．

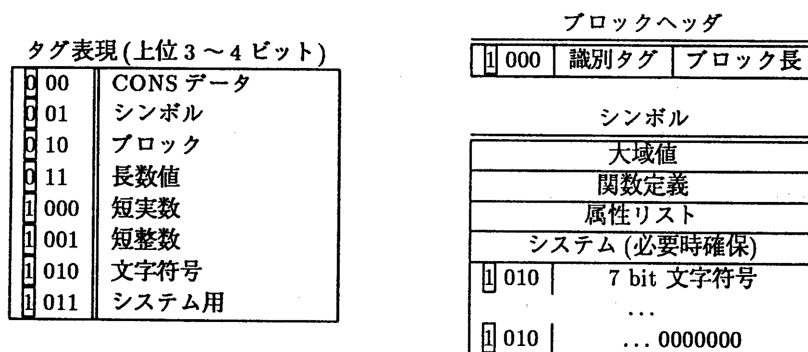


図1: データ表現

このような表現を採用することで，以下に述べるような利点が得られる．

1. 整数はその絶対値が1億未満については即値表現(短整数)となり，記憶を消費しないし，同値関係も高速に判定できる．

2. 実数は精度が20ビット以内の単精度浮動小数点数の範囲にあれば，即値表現(短実数)となり，CLの埋め込み即値表現実数よりも精度は高い．短実数は数式処理に広く含まれる実数計算を強力に支援する．

3. 文字はASCII 7 bits符号であれば，4文字まで(1語に)詰められるし，漢字も使用できる．

PHLのデータオブジェクトはデータ格納領域とスタック領域に置かれる．データ格納領域に置かれたオブジェクトは一部を除いてGCの対象になる．このGCは圧縮方式[5]を採用しており，ブロックを含む可変容量オブジェクトの効率的な管理を実現している．

2.3 インタプリタ

PHLインタプリタの設計目標は処理の高速化にあった．これを端的に言うと，多少の親切さを犠牲にしても実行効率を上げることである．当然のこととして，traceの機能や引数の型検査はあるが，引数の個

[2]AグループのMSBは大部分の計算機アーキテクチャにおいて零であるが，これが1となる事例もあるらしい．

[3]例えば，Sony RISC-Newsワークステーションではgccのmalloc関数の返す値が0x10000000のゲタをはいているため，アドレス部の先頭ビットは1になる．このためデータ格納領域として使用できる容量は28ビット分の256KBになる．

数の検査やエラー発生後のバックトレースの情報は通常表示しない[4]などの徹底した効率化が図られている．インタプリタが陽に使用するスタック領域はユーザスタック (U スタック) と制御スタック (C スタック) がその両端から伸びる．U スタックは関数の引数の引渡しや変数束縛, (多) 値の返却, 関数実行時の作業用に使用される．GC 時の根 (root) とすべきデータオブジェクトはすべて U スタックに積む必要がある．C スタックはエラーや throw, SLISP の errorset などによる大域的な出口への移行時のフレームの調整に使用される．LIFO 的な制御の移行は C 言語によるシステムスタックが自然に行なってくれるからである．そこに積まれるものは引数である FP や EP などのフレームポインタ, 関数からの戻り番地やレジスタ値である．

U スタックが形成する領域は一般にフレームと呼ばれるが, PHL のフレームは次のような特徴をもつ．

1. インタプリタが解釈実行する関数[5]では引数値とそれに対応するシンボルが隣接する (図 2 参照)．機能的には a-list 構造がスタック上に展開されたと考えればよい．
2. 動的変数のシンボルとその束縛値の対も同じフレーム内に置かれる．この種の環境は制御スタックと呼ばれる別のスタックにとるのが一般的な技法であるが, 単純局所変数と動的変数の値を同じフレーム内に格納することによって, この両者の値の参照はインタプリタで同じ手続きにより行なうことができる．また, 制御スタックが簡素化され, インタプリタでは関数の戻りの操作が不要となる．
3. 各フレーム (またはその一部) は必要に応じて連鎖状になる．このため, 関数によっては動的変数と透過的な単純局所変数が混在すると複雑な構造となる．
4. インタプリタによるフレーム内走査はパラメタ表記とは逆の順序で (後から) 行なわれる[6]．

3 PHL コンパイラ

3.1 コンパイラ仕様

Lisp プログラムから対象計算機のオブジェクトコードを生成する方式は大別して次の 2 通りがある．

1. 対象計算機のオブジェクトコードを直接生成する方式
   対象計算機アーキテクチャの利点を引出すような最適コードが生成できる半面で, 計算機依存が高く可搬性は劣る．Lisp コンパイラは伝統的にこの方式が多かった [6]．
2. プログラム変換による方式
   ソース異種言語のプログラムに変換し, その言語コンパイラが変換されたプログラムをオブジェクトコードに生成する．結合部を除けば計算機非依存になるし, 既製のコンパイラが良質のオブジェクトコードを生成する言語を選定すれば効果も大きい．C 言語を採用した KCL[7] はこの典型である．

なお, これらの折衷案として仮想計算機の中間言語を経由する方式もあるが, 普遍的な言語の設計が難しく, しかもコード生成部は計算機依存となる．

PHL コンパイラは可搬性をさらに高めるためにソースプログラム変換という方式を採用した．Lisp プログラムは PHL で記述されたコンパイラで C 言語のソースプログラムに変換され, ファイルに格納される．これらのプログラムは C(gcc) コンパイラによりオブジェクトコードに変換され, PHL 処理系の他のルーチンとともに結合され, 新たな処理系として実行に供される．なお, コンパイラはブートストラップ可能で, 変換された C プログラムはすべて編集や修整も可能である．端的に言えば, C(gcc) コンパイラと make コマンドだけで翻訳実行を行なうことができることになる．

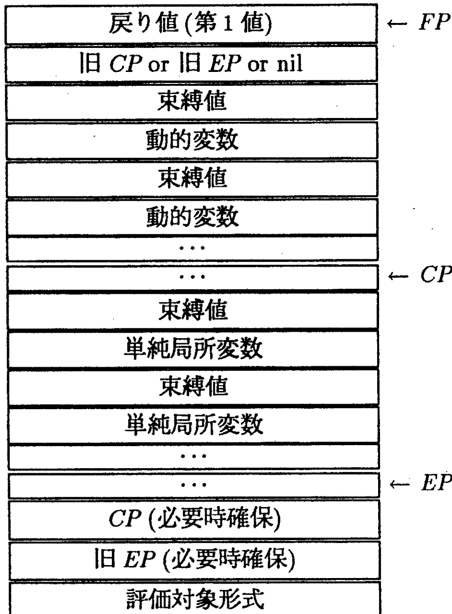
この半面で, 翻訳実行の作業は一旦 PHL 処理系を出て新たな処理系を実行することになる．これは Lisp の伝統的機能である対話環境の放棄になるが, 翻訳作業は完成度の高い洗練された数多くのプログラムに対して少ない頻度で求められることや PHL インタプリタの高速化も実現されていることなどから, こうした処理でも大規模アプリケーションプログラムの作成は効果的に行なえる．

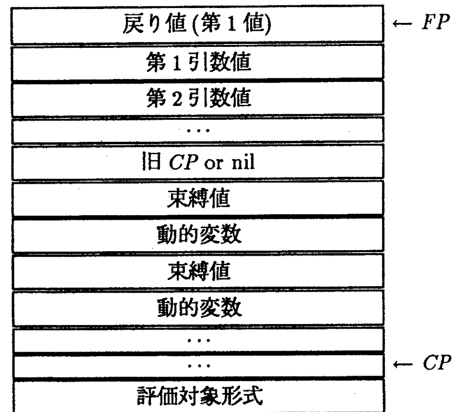
---

[4] こうしたデバッグ機能の追加は C(gcc) のマクロプリプロセッサに対する指示という形になっている．

[5] SLISP の de や df, dm で定義された関数のこと

[6] この仕様は Lisp 1.5 [6] と異なる．





記号
FP: フレームポインタ
CP: フレーム内動的変数領域
EP: フレーム内単純局所変数領域

図 2: フレームの構成

3.2 機能

PHL コンパイラは現インタプリタの仕様を満足する. これはインタプリタとコンパイラの整合を厳格にした CL 仕様[7] の反映であるが, 一部の機能を除いてインタプリタで実行できるプログラム (関数) は翻訳実行が可能である. より良質のコードを生成するための最適化を除けば, Lisp からそれと等価な C プログラムを生成すること自体に技術的な問題はなく, コンパイラの作成は比較的容易である. ただ, PHL コンパイラでは Lisp オブジェクトを再入力可能書式で C 言語のプログラムに変換することとそれらの結合法, 及び, 記憶管理法が課題となった. そこで, これらの対処法について述べる.

3.2.1 再入力可能書式

Lisp から C へのプログラム変換で問題となるのはプログラム中に存在する, 関数名や大域的変数などのシンボル, 文字列や数値, `quote` の引数などのデータの扱いである. 翻訳実行を処理系内で一貫して行なう場合はこうしたデータは記号表で効果的に管理され, コンパイラはそうしたデータの存在だけを意識すれば済む. しかし, プログラムソースを生成する場合は, こうしたデータは内部表現から一旦, 再入力可能書式として出力 (記述) する必要がある. C プログラムではそれらは文字列として記述される.

[7]例えば, 名付きオブジェクト (変数など) に対する両者の整合問題の解決などがある.

PHL ではデータを再入力可能書式で出力する関数 (prin3) を用意した. なお, D グループについてはこうした処理は不要で, そのまま C の定数に直される (図 3 参照).

図 3: Lisp オブジェクトの書式

データ型	例	一般入力書式	通常出力書式	C 用の書式
CONS データ		(α . β)	(α . β)	(α . β)
シンボル	A	A または a	A	A
	a"b	| a"b |	a"b	| a\"b |
文字列	a b	"a b"	"a b"	\"a b\"
長数値	-10^8	-100000000	-1_00000000	-100000000
短整数	1	1	1	Czero+1
短実数	0	0.0	0.0	Crzero
文字	a	#\a	#\a	Tag_char | 'a'<<21

3.2.2 結合

PHL コンパイラは A グループに属するオブジェクトがプログラム中にデータとして存在した場合, それに一義的な名前[8] を付ける. なお, 関数名は, 'lc' の接頭辞を付ける. 例えば,

(defun f (x) (cons x 1)) は

```
void lcf(Object *pf) {
  *(pf+2)=Czero+1; advance_utop(pf+2); lccons(pf); return; };
```

と変換されるが,

(defun g (x) (cons x 100000000)) は

```
void lcg(Object *pf) {
  *(pf+2)=Lg0; advance_utop(pf+2); lccons(pf); return; };
```

と変換される. この 'Lg0' が 10^8の Lisp データを表す C の名前である. そこで, Lg0 が 10^8のデータを値に持てばよいのであるが, この設定は「新たな処理系」が起動する時に行なわれる. 具体的には, compiler_init という関数内で
symbol_link(&Lg0, "100000000", NULL);
で設定される. その動作は, read が文字列中の 100000000 を読取り, それを格納領域に構築し, 返したデータ値が Lg0 に代入されるというものである. また, シンボル F の関数定義に lcf のアドレスを設定することも, この関数を利用して
symbol_link(&dummy, "f", lcf);
で行なう. この場合, 第一引数は使用しない.

3.3 記憶管理

PHL 処理系では Lisp データオブジェクトはすべて 1 つのデータ格納領域に置かれる. それらはアドレスの小さい方から順に配置される. また, 圧縮方式 GC は使用中オブジェクトをアドレスの小さい向きに移動する. このため, 幾多の GC を経て存在し続けるデータオブジェクトはアドレスの小さい格納領域の一端に溜まり, 錨 (anchor) と呼ばれるオブジェクトの集団を形成する. 錨は移動もないし, 条件しだいではポインタ補正も不要となる好都合な集団である. これらを排他的に管理する GC 技法 [5] が具現されているが, C プログラム中に名前として存在する Lisp データの管理にこの技法が活用できる.

新処理系の起動時に compiler_init で作られるこうしたデータは当然, アドレスの小さい場所に配置される. これらのデータオブジェクトはポインタ補正の対象となる可能性はあるが, 再配置の対象には

[8]同値のデータでも別の名前が付けられることもあるが, C の変数名が多くなるなるだけで, 特に問題はない.

ならない. これは錨と同じ性質であり, PHL の GC はこうしたオブジェクトを含む錨を効果的かつ効率的に処理する.

4 実行結果

PHL は当研究室にある種々の計算機上で稼働中である. そのインタプリタとコンパイラによるテストプログラム [8] の実行時間を図 4 に示す. 単位は秒である. なお, 対象計算機の C コンパイラは最適化版を用いた.

図 4: プログラム実行時間 (秒)

プログラム	機種	SPARC(classic)	NEWS(3410)	DELL(466MXV)	NEWS(5000)
(tarai 12 6 0)	解釈系	845	976	350	224
	翻訳系	39	74	29	17
(tak 18 12 6)	解釈系	5.4	5.1	1.8	1.2
	翻訳系	0.20	0.43	0.18	0.10
(nrev list30)	解釈系	0.049	0.040	0.015	0.010
	翻訳系	0.006	0.006	0.003	0.001
TPU	解釈系	22.49			5.64
	翻訳系	1.45			0.50

図 5 と図 6 は, その最初のプログラム (TARAI) の両者の実行時間とその比を図示したものである. 時間比に大きな差異がないのが興味深いところである. また, C コンパイラの最適化を省くと, プログラムによっては 2 倍もの開きがでることも別の実験からわかった.

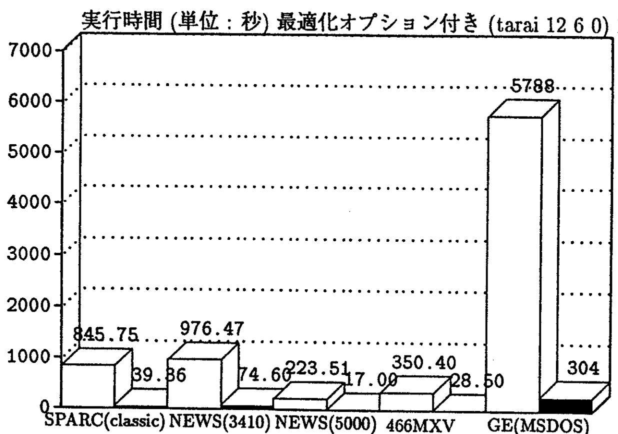


図 5: 実行時間

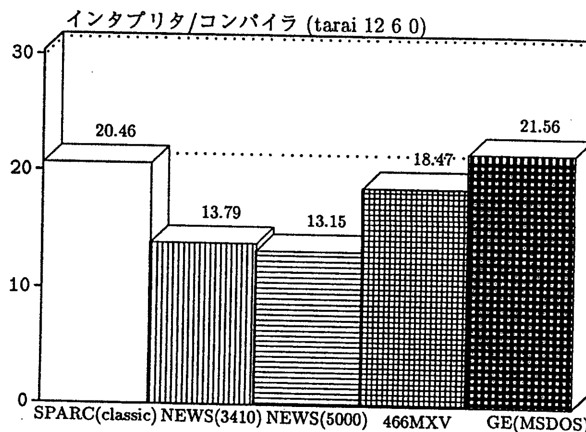


図 6: 実行時間比

5 まとめ

本稿ではプログラム変換に基づく PHL コンパイラの仕様と機能について述べた. コンパイラを含めた PHL 処理系はこれという移植作業もなく研究室内の異種計算機で稼働しており, 可搬性の実証になっている. 現 PHL コンパイラでも多少の最適化を行なっているが, それでもプログラムの実行速度は対象計算機で使用する C コンパイラの性能に大きく左右される. PHL コンパイラ方式によるオブジェクトコードの最適化には限界があり, 対象計算機を意識して生成された洗練済のコードには及ばないことは明らかで, 実際の処理時間はそれを反映したものになっている. 今後ともハードウエアの進歩によりアーキテクチャの異なる種々の計算機が登場することが予想される. 開発者の手から離れた処理系を永く持続させるためには, 移植のノウハウが不要で完全に計算機独立であることが強く求められよう.

6 参考文献

1. Steele, G. L. Jr. : *Common LISP, (2nd ed)*, Digital Press, MA. (1990).
2. Marti, J. et. al. : STANDARD LISP REPORT, SIGPLAN Notices, 14 (10) pp.48–68 (1979).
3. 寺島元章 : PHL の新インタプリタ, 記号処理研究会資料, SYM 73–5, pp.33–40 (1994).
4. Hearn, A. C. : REDUCE User's Manual (Ver. 3.3), The Rand Corporation, CA. (1990).
5. 寺島元章 : 古典的ガーベッジコレクションからの話題, 記号処理研究会資料, SYM 68–5, pp.31–38 (1993).
6. McCarthy, J. et al. : *LISP 1.5 Programmer's Manual*, MIT Press, MA. (1962).
7. Yuasa, T : Design and Implementation of Kyoto Common Lisp, Journal of Information Processing, 13 (3) pp. 284–295 (1990).
8. 竹内郁雄 : LISP 処理系コンテストの結果, 記号処理研究会資料, SYM 5–3 (1978).