記号処理　40－4
（1987. 1. 13）

ＥＬＩＳにおけるＬｉｓｐコンパイラ

神尾視教　酒井高志　竹内郁雄

ＮＴＴ電気通信研究所

本報告は、ＡＩワークステーションＥＬＩＳ上のＬｉｓｐコンパイラの構成、特徴、およびその評価結果について述べる。

ＥＬＩＳ上では、かなりの関数がファームウェア化された高性能のＴＡＯインタプリタが実現されており、コンパイラではそのファームウェアを有効に利用している。本コンパイラは、インタプリタとの完全な互換性をもっており、コンパイルコード中でＳ式をそのまま実行したり、変数のバインディング情報を容易に得られるという特徴をもっている。コンパイラによって実行速度はインタプリタの２－５倍、オブジェクトプログラムのメモリ占有量は１／１０－３／１０になる。

IMPLEMENTATION OF LISP COMPILER ON ELIS

Minoru Kamio　Takashi Sakai　Ikuo Takeuchi

NTT Electorical Communication Laboratories

This paper describes a Lisp compiler on the lisp machine ELIS. ELIS has already a high performance interpreter, and the large part of the interpreter is written in microcode.

The compiler is completely compatible with the interpreter, which enables uncompiled part of S-expression code to be interpreted within the compiled function. The compatibility also enables variable binding informations to be retained just as the interpreter thus the user can get full backtrace information as in the interpreter even in the compiled run.

Using the compiler, the programs are executed 2 - 5 times faster than in case of the interpreter, and the memory space for programs are reduced to 1/10 - 3/10.

1．まえがき

LispマシンELISにおいては、既に高性能のTAOインタプリタが稼働している。〔1〕〔2〕 我々は、いままでのインタプリタのみのシステムにコンパイラを加え、実行速度、メモリ消費量の改善をはかった。このコンパイラは、ELISの核言語であるTAOで記述されたトランスレータと、マイクロコードで記述されたLAPインタプリタとから構成されている。本報告では、ELISのLispコンパイラシステムの構成と特徴について述べる。また、実行速度、メモリ消費量についてインタプリタとの比較を行う。

2．コンパイラの設計方針

（１）コンパイラ用中間言語はバイトコードにし、基本的関数呼び出しやローカル変数アクセスは１バイトで済むようにして、メモリの節約とバイトコードアクセスの高速化をはかる。

（２）フレーム構造、変数、その他の文脈情報（GOのタグ、BLOCKの名前など）の管理をインタプリタと同一にすることによりコンパイラとインタプリタの完全互換をはかる。特に１つの関数本体の中でインタプリタとコンパイラの混合コードを完全に動作させることができるようにする。これにより、コンパイラの段階的な開発が可能になる。これは、Lisp，オブジェクト指向，論理型の融合言語であるTAOのコンパイラを段階的に開発するために必要な条件である。

3．コンパイラの構成

本コンパイラにおいて、関数をコンパイルして実行可能にするまでを、図１に示す。

（１）トランスレータはソースファイルをLispの中間言語LAP(Lisp Assembly Program)に変換する。このトランスレータはCOMPILE-FILE又は、COMPILEによってLisp関数として起動される。コンパイラによって生成されるLAPの構成を図２に示す。

（２）変換されたLAPをLAPローダによってロードし、LAPバイトコードに変換することによりインタプリタの実行環境内に取り込む。コンパイルされた関数のメモリ構成を図３に示す。

（３）LAPインタプリタはコンパイルされてロードされた関数が呼び出される時にインタプリタから制御が移される。ELISはメモリからの読み込みと書き込みのできる汎用レジスタMGRを持っており，このMGRはauto-increment可能なインデックスレジスタ(SDC)によりバイト単位でアクセス可能である。〔3〕 LAPインタプリタはこのMGRとSDCを用いてバイト領域上に展開されたLAPコードを８バイト単位にMGRに読み込み，SDCを用いて１バイト毎に解釈実行をおこなう。 LAPインタプリタの実行が終了すると，制御はインタプリタに戻される。

4．コンパイラの特徴

4．1 関数呼び出し

（１）マイクロ関数(SUBR)の呼び出し

TAOでは基本的な関数をほとんどファームウェア化しており、それを使ったほうがインライン展開するよりも高速な場合にはコンパイルコードからもそれらを使う。そのため、各マイクロ関数にLAPコードを割り当ててあり、その数は約４００個である。マイクロ関数の呼び出しにおいては、フレームは作らずマイクロ関数本体に直

接ブランチする。トランスレータは、スタック上に引数を積むLAPコードに続けて各関数に割り当てられたLAPコードを展開する。

```
(CAR x)=〉@x CAR
(APPEND x1 x2 ...xN) =〉 @x1 @x2 APPEND2 @x3 APPEND2 ...APPEND2 (N〉=2)
(APPEND x1) =〉 @x1
(APPEND)    =〉 @NIL
```

ここで@xは、xのコンパイル結果を示す。

（２）特殊形式のコンパイル

一方、PROG、LETのように新しい環境を作るものについては、スタックフレーム作成用のLAPコードを用意する。PROGの変換規則を以下に例示する。

```
(PROG exit-id (var1 var2...) tag1 form1 ... formN)
  =〉 (BLK (DCONST (MAKE-GO-VECTOR tag1...))
           (CONST exit-id)
           (DCONST (MAKE-VAR-VECTOR PROG var1 var2 ... )
            PROG 1NIL ... !tag1 @^form1 ... @formN POP-EXIT-LINK EXITFRAME)
```

ここで、@^formはformの値をポップするコンパイルをすることを示す。またDCONSTはLAPロード時にベクタの作成を指示するものであり、作成されたベクタは定数ベクタに格納され、実行時にスタック上にプッシュされる。ここでは、GOタグベクタと変数ベクタを作成している。GOタグベクタはタグ名と対応するアドレスのペアのテーブルでありインタプリタからも参照される。

PROGで作成するスタックフレームを図４に示す。図４で、変数（引数）ベクタは変数名（引数名）とその性質（必須、オプション、補助等）を示すテーブルであり、一般にインタプリタではS式の実行時に変数ベクタを使って変数の位置を知る必要があるが、コンパイルコードではスタック内相対位置が計算されているので、本来は不用である。しかし、本コンパイラでは、コンパイルコード中でも未コンパイルのままのS式の実行を可能とするためや、コンパイルコード実行中でも変数のバインディング情報を容易に得られるようにするために、変数ベクタをインタプリタと同様にスタックに積むことにしている。

（３）S式関数(EXPR)の呼び出し

マイクロコードで書かれていない（つまりS式で書かれた）関数を呼び出すときには、事前に関数情報をスタックフレームに積む。トランスレータは関数呼び出しに対してはロード時の環境が不明なため，最適の呼び出し方法を決定することができない。このため関数の呼び出し方法の決定はロード時の環境によってLAPローダが決定する。

①未コンパイル関数の呼び出し

関数名による呼び出しを行うLAPコードに展開する。この場合LAPコードMKFRAMEはその関数名が持つ関数本体のアドレスをスタックに積む。

```
(FOO arg1 arg2 ... argN)
=〉 (MKFRAME FOO) @arg1 @arg2 ... @argN (CALL FOO)
```

②コンパイル関数の呼び出し

同一のLAPファイルにある関数の呼び出しや、既にロードされているコンパイル関数の呼び出しについては，定数ベクタに呼び出される関数のapplobj（関数オブジェクト）と定数ベクタを格納する。LAPインタプリタは、applobjと定数ベクタをスタックに積んだあと関数呼び出しをおこなう。これにより関数名を経由するメモリアクセスの手間を削減することができる。ただし、この場合コンパイル済みの関数中の一部だけを再定義した場合，古い関数が呼ばれることになる。なお、宣言があればコンパイルされた関数に対しても通常の関数名経由による呼び出しを行うことができる。

③単純再帰

再帰呼び出しを行う場合，スタック上で必要なフレームをコピーすれば、単純なジャンプ命令で再帰呼び出しが行える。（メモリアクセスに比べてスタックのアクセスは４倍以上速い。）ただし，この場合関数内で呼んでいる関数は静的なスコープをもつものでなければならない。FACTORIALの例を以下に示す。

```
(DEFUN FACT (N)
   (COND ((= N 0) 1)
         (T (* N (FACT (1- N))))))
=> (LAP DEFUN FACT (N)
    (SR-BEGIN
     !1 (CONST 0) (IF/= 2)(CONST 1) (SR-END 1)  !2 SP3 SP4 1- (SR 1) MUL
     (SR-END 1)))                                              exit
```

４．２　復帰処理の最適化

EXIT,RETURN-FROM,GOを使って関数から脱出する場合、多段のフレームをたたまなければならない時がある。このため、1段と多段のフレームたたみのLAPコードを用意し、最適なコンパイルコードの展開を可能としている。

・EXITFRAME　MULTIPLE-EXITFRAME　BUMP-EF　BUMP-TF　BUMP-EF=TF

４．３　フレーム作成、解放回数の削減

本コンパイラは、インタプリタとの完全な互換を目指して設計されており、フレームの構成もインタプリタと同一のフレームを作成するようになっている。しかし、文脈によりフレームの作成、解放の回数を減らすことができる。例えばLOOP関数が補助変数の定義をもたなければ、フレームを作らない。

４．４　定数のコンパイル

文字列や一般の数値、あるいはquoteされたリストデータのように、バイトコードに納まらない定数は、トランスレート時に(CONST x)として変換され、これらはLAPロード時に定数ベクタに格納される。実行時にはこれらの定数をベクタ内のオフセ

ットをキーにしてアクセスする。定数ベクタはLAP実行時にレジスタによって保持される（一種のベースレジスタ）。LAPローダは定数ベクタ作成にあたって、絶対値の小さい整数、文字（日本語文字を含む）といった短い固定長を持つものについてはLAPコード中に展開し、メモリアクセスの削減をはかる。

４．５　変数のコンパイル

（１）ローカル変数

ELISはハードウェアスタックを持っており、TAOはマルチプログラミングの容易性を考慮して深い束縛を採用している。トランスレータでは、スタック上の位置によりローカル変数をアクセスするLAPコードを出力する。このLAPコードには以下のものがある。

①スタックトップからの相対位置によるアクセス
②フレームからの相対位置によるアクセス
③N個の連続したスタックのプッシュ／ポップ

（２）スペシャル変数

スタック上のスペシャル変数のアクセスは、特別に(splvar)というタグのついた変数をサーチする(*)LAPコードになる。しかし、DECLAREがあれば、トランスレータはラムダ引数にその変数があるかどうかのチェックを行い、なければスペシャル変数をサーチした直後にそれをローカルフレームに束縛するLAPコードを出力する。つまり、サーチ結果は関数の処理が終わるまで無名のままローカルフレームに保持され、その変数のアクセスはローカル変数へのアクセスと同様の速度で行うことができる。

```
(DEFUN FOO () (DECLARE (SPECIAL X)) (CAR X))
 =〉 (LAP DEFUN FOO (&AUX X) ((CONST X) (REFER-SPECIAL 2) VARO CAR))
```

（３）宣言のない変数

宣言のない変数についてはインタプリティブに実行するべきコードと警告メッセージを出力する。つまり,トランスレータは以下のLAPコードを生成する。

```
(CONST  VAR-NAME)  EVAL
```

また、未コンパイルの式は上と同じ形に展開される。

（＊）ほとんどの場合スペシャル変数のキャッシュがヒットするので実際にサーチは起こらない。

５．評価

このコンパイラの性能評価として、インタプリタの実行速度ならびにコンパイルされたコードの実行速度、およびメモリ使用量の改善度を表１、表２に示す。

６．考察

（１）実行速度について

トップのフレーム中の変数アクセスをインタプリタとコンパイラで比較した場合、約6.5倍である。また、１段下位のフレームにおける変数に対するアクセスでは4.3

倍になる（ただし、インタプリタでもローカル変数のアクセスは”見えないポインタ”による前処理が行われているのでサーチは行われず、十分高速である）。

マイクロ関数の呼び出しのみからなる関数のコンパイルコードの実行では、変数の束縛に要する時間と式自身に対するメモリアクセス時間が短縮されるだけである。

単純再帰の効果は、TARAI-4関数の場合、単純再帰のオプティマイズを行わない場合の1.4倍となった。

（２）メモリ使用量について

表２に示されるようにコンパイルコードのメモリ使用量は、インタプリタの１割から３割の範囲に納まっている。これは、バイトコードに展開していることによる。ただし、極端に定数ベクタの大きいもの、つまり関数本体の中に文字列、ビグナムやquoteされた定数が多い場合には削減率は小さくなる。

（３）インタプリタとの共存の効果

変数名を保持しているため、BACKTRACE、BREAK関数実行時に変数名でアクセスできインタプリタ並のデバグ情報を得ることが出来る。また、LAP中に未コンパイルのS式をそのまま記述できることは、コンパイラの開発において大いに有効であった。

７．まとめ

LispマシンELISのLispコンパイラについて報告した。ELISではTAOインタプリタが動作しており、その上でCommon Lispインタプリタがエミュレートされている。今回報告したLispコンパイラはTAOのLisp部分のコンパイラである。現在、オブジェクト指向部分および、Common Lisp部分のコンパイラについても、ほぼ動作しており、これについては別の機会に報告したい。

本コンパイラの開発にあたり御協力いただいたELISグループの皆様に感謝いたします。

［参考文献］

〔1〕日比野，渡辺，大里 "LISPﾏｼﾝELISの基本設計"情処，記処研資 12-15，1980

〔2〕竹内，奥乃，大里 "LISPﾏｼﾝELIS上の新しいLISP TAO"情処，記処研資 20-5，1982

〔3〕日比野，渡辺，大里 "LISPﾏｼﾝELISのｱｰｷﾃｸﾁｭｱ：ﾒﾓﾘﾚｼﾞｽﾀの汎用化とその効果"情処，記処研資 24-3，1983

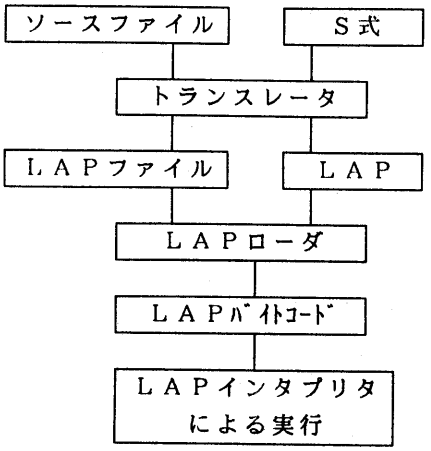

図1　コンパイル処理の流れ

```
(LAP                ;LAPコードの識別.
  DE又はDEFUN       ;関数タイプの識別.
  function-name
  lambda-list       ;引数リスト
  compiled-body
      ;補助変数用スタック域の確保ためのコード
      ;定数ベクタ確保のためのコード
      ;変数ベクタ確保のためのコード

  ...  )
```

図2　コンパイルされた関数の構造

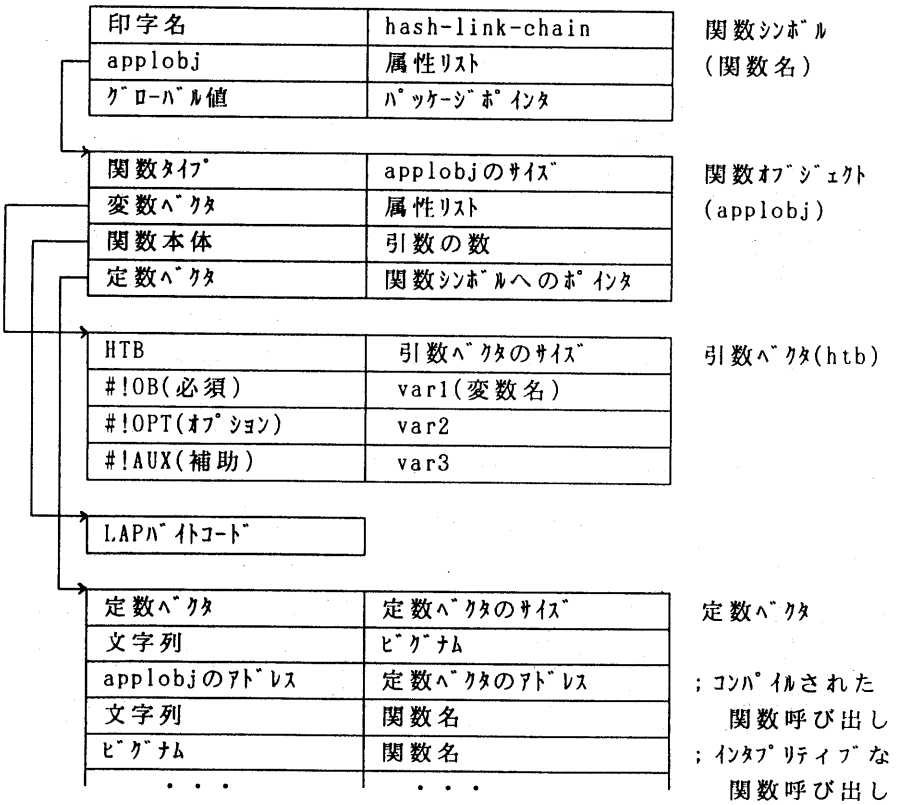

図3　コンパイルされた関数のメモリ構成

補助変数	; 補助変数用のｽｯﾀｯｸ域
正規の引数	; &OB,&OPTIONAL等の引数のｽﾀｯｸ域
GOﾀｸﾞﾍﾞｸﾀ	
exit-id	
exit-link	; 大域脱出用のﾘﾝｸ
self-env-frame	; 自分自身を指すﾌﾚｰﾑﾎﾟｲﾝﾀ
tf	; 1つ上のﾌﾚｰﾑへのﾎﾟｲﾝﾀ
ef	; 1つ上の環境ﾌﾚｰﾑへのﾎﾟｲﾝﾀ
変数ﾍﾞｸﾀ	; 変数の管理ﾃｰﾌﾞﾙ
my-cbase	; このﾌﾚｰﾑの定数ﾍﾞｸﾀ(非常時回復用)
lap-return-address	; LAPｲﾝﾀﾌﾟﾘﾀへのｱﾄﾞﾚｽ
SDC3	; PC+SDC3で呼び出し元ｱﾄﾞﾚｽを示す
PC	
caller's-cbase(保存)	; 呼び出し元の定数ﾍﾞｸﾀ

図4　PROGのｽﾀｯｸﾌﾚｰﾑ

表1.　ｲﾝﾀﾌﾟﾘﾀとｺﾝﾊﾟｲﾗにおける実行時間の比較

	ｲﾝﾀﾌﾟﾘﾀ(I)	ｺﾝﾊﾟｲﾗ(C)	倍率(I/C)
TARAI-4	790　MSEC	152　MSEC	5.20
LIST-TARAI-4	1416	416	3.40
FIB(20)	1716	366	4.69
TPU-2	2716	1216	2.16

本測定は、14MHzのTTL版ELIS(Magnesium)で行ったものである。
(ﾌﾟﾛﾄﾀｲﾌﾟおよびLSI版は16.67MHzでこれより約20%高速.)

表2.　ｲﾝﾀﾌﾟﾘﾀとｺﾝﾊﾟｲﾗにおけるﾒﾓﾘ消費量の比較

	ｲﾝﾀﾌﾟﾘﾀ(I)	ｺﾝﾊﾟｲﾗ(C)	倍率(C/I)
TARAI	440　BYTE	53　BYTE	0.12
OPEN	8192	1404	0.17
CLOSE	792	215	0.27
COPY-FILE	1392	326	0.23
RENAME-FILE	2448	481	0.20