寄　　書

LISP におけるインタプリタとコンパイラとの関係*

辻　三郎**　田村浩一郎***　守屋　朋子****
黒崎冨美子****　小川　明宏****　穂鷹　良介****

Abstract

Analogy of two systems——LISP interpreter and LISP compiler——is investigated through the use of recursion.

In the section 2, two systems which we developed are briefly sketched.

In the section 3, remarkable properties of the LISP interpreter, in the section 4, the meta-expression of LISP compiler together with its logic, in the section 5, some examples of LISP compiler are respectively investigated.

Finally in the section 6, some statistics concerning about the execution time between the interpreter and the compiler are compared.

1. はじめに

最近，われわれは LISP インタプリタと LISP コンパイラとを開発した注1)．LISP インタプリタについては文献 1) にかなり詳しくその内容が紹介されているが，コンパイラについては暗示的ないくつかの説明がなされているだけで，体系的な説明はないように思われる．

本稿は文献 1) に基づいて実際に LISP インタプリタとコンパイラとを作成する方法について述べるものであるが，その際，両者の内部構造の類似性と統一性とに特に注意を向けている．すなわち，コンパイラもインタプリタと同様にエレガントな記述をM式の形で与えることができること，またそのM式がインタプリタのそれに非常に類似していることを示す．この両者の記述では再帰 (recursion) が重要な役割を果たしている．**2.** ではシステム全体の構造を概観する．**3.** ではインタプリタの内部構成について，その一部の機能がいかにシステムの他の部分，またはユーザの普通のプログラムで使われるか，したがって，またユーザが使うことのできる LISP の機能の一部分は，特にユーザ用として作らなくてもシステム内に自然にできあがっているということを示す．

4. では LISP 1.5 流のコンパイラをリカージョンを用いて implement することができることを示し，インタプリタと同様にコンパイラシステムの動きを表現する meta-expression を提示する．**5.** では具体的な LISP プログラムのコンパイルが **4.** で示した方法でどのようになされるかを例示する．最後に **6.** では LISP プログラムの実例について，インタプリタとコンパイラとで計算を実行したときの差などについて述べる．

2. システム全体の構成

今回開発した LISP システムは，大別するとインタプリタとコンパイラとの二つから構成される．

インタプリタは，S式を評価してその値を求めるという機能のほかに，コンパイラやコンパイルされたオブジェクトプログラムが使用するリスト処理に便利な

---

* Relation between the interpreter and the Compiler in LISP, by Saburo Tsuji (Osaka Univ. Dept. of Fundermental Engineering), Koichiro Tamura (Electrotechnical Laboratory, Div. of Information Sciences), Tomoko Moriya, Fumiko Kurosaki, Akihiro Ogawa and Ryosuke Hotaka (Nippon Software Co. Ltd.)

** 大阪大学基礎工学部
*** 電子技術総合研究所パターン情報部
**** 日本ソフトウエア株式会社

注1) 計算機は NEAC 3100，主記憶は 32K 語で1語18ビットの構成である．システムはアセンブリ言語 SPASE によって書かれた．

諸関数を内蔵している．

コンパイラは一つの関数としてインタプリタのなかに埋没していて，コンパイル時，実行時ともに共通のインタフェイスを使用してインタプリタのもつ諸機能をできるだけ利用してシステム作成の効率化を図っている．

応用上の必要性から LISP 側から FORTRAN プログラムにリンクする機能がつけ加えられ，LISP で行なうよりも効率のよい数値計算などを行なっている．FORTRAN プログラムは，メモリに余裕のあるときには LISP システムと同時に主記憶に置かれて実行されるが，ユーザの指定によって LISP システムを一度二次記憶にスワップアウトしてからコントロールを渡す方式も用意されている．このような，非常駐の FORTRAN プログラムの実行後再び LISP プログラムがスワップインされて実行を継続する．これらのコントロールは LISP モニタがつかさどる．

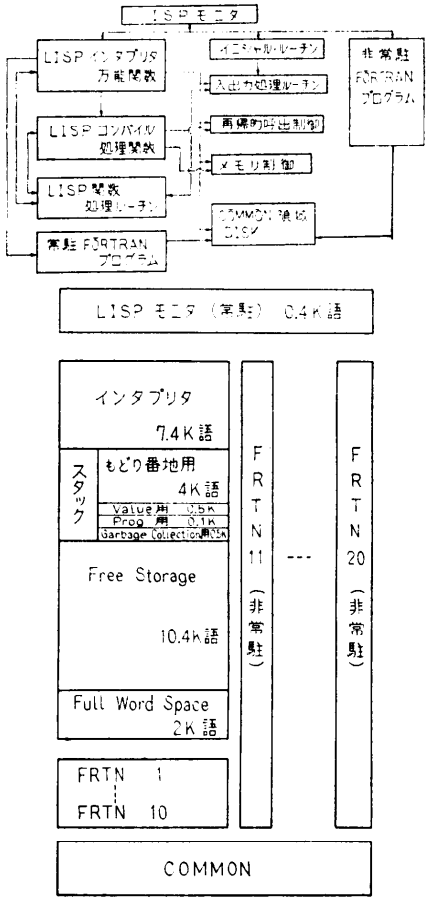

図 2.1 Strucure of the LISP system and the memory map

LISP から呼ばれるこれらの外部関数は，実は FORTRAN プログラムだけに限るものではなく，FORTRAN プログラムと同様のリンクの規約を守っているものであるならばなんでもよい．LISP とこれらの関数との情報交換は COMMON 領域とディスク上にとられたファイルを通じて行なう．

これらの機能を稼動させるために，LISP 1.5 にはない関数を本システムに備えている．しかし，これらの詳細については，本論文では，これ以上触れない．

最後に，LISP システムの全体構成図を 図 2.1 に示す．

3. LISP インタプリタ

インタプリタは，コンパイラのベースとなるものである．ここではコンパイラ作成のときに基本となるインタプリタのいくつかの性質を述べる．もちろん，これらの性質は，インタプリタを単独で稼動させる場合にも，重要な性質である．

(1) 関数処理ルーチン

われわれが開発した LISP インタプリタは，70個ほどの処理サブルーチンから構成される．このうち約半分が関数処理ルーチンで，残りは関数の補助ルーチンと parser などの外郭ルーチンである．

関数処理ルーチンは，car, cdr のような基本関数処理ルーチンを核として構成され，複雑な関数処理はこの基本関数処理ルーチンの組み合わせでできる．この処理ルーチンを作成する場合，関数としては存在しないが，ある機能を果たす補助ルーチンを派生的に作成する場合がある．このような補助ルーチンは，システムが内蔵しているもので，ユーザがこのルーチンのもつ機能を直接利用することができないこともあるが，新しい関数の関数処理ルーチンとして定義しようと思えば定義できるものである．

このように，ある関数の処理ルーチンを作成するために，システムの内蔵するルーチンが最大限に利用され，また，ユーザもこれらのルーチンを関数として利用することができる．たとえば nconc というルーチンは，関数 car, cdr, rplacd などの処理ルーチンを使用している．このことは，LISP コンパイラについても同じで，コンパイラはリスト構造の引数に処理を加えるときに，インタプリタの関数処理ルーチンを使用している．

図 3.1 は，関数とプログラムの関係と，そのプロ

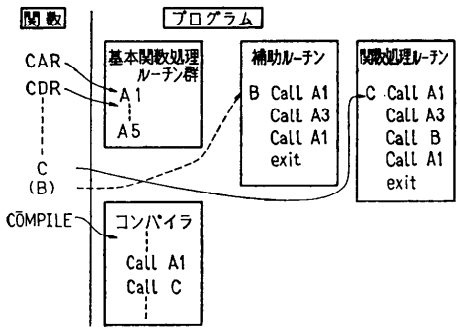

図3.1 Relations between the LISP function

グラムの構成の概要を例示するものである.

(2) 再帰的呼び出し処理ルーチン

インタプリタの処理ルーチン群は，再帰的呼び出しがされるものとそうでないものとに分かれる.

図3.1 の B, C のプログラムは，非再帰的なルーチンの例であるが，これが再帰的な構成をもつとこのように単純ではなくなる. たとえば，図3.2 のプログラムXは，このままでは A1, A2の値が失われるし，帰り番地も設定できないので，正常に動作できない. このことから，引数と帰り番地を push down, pop up する操作が必要となる.

これを考慮して 図3.2 を完全な再帰的ルーチンにすると 図3.3 のようになる. われわれの作成したインタプリタでは push down するものを，第1引数，第2引数，a-list, 帰り番地の四つとした.

(3) リストの保護

各処理ルーチンでは，一時的にリストを保存しなけ

```
X(A 1, A 2)
      X    Call A (A 1)
           Call B (A の結果)
           if B の結果=0, exit
           Call X (A 2, B の結果)
           Call A (B の結果)
           exit
```

図3.2 An example of a non-recursive routine

```
X(A 1, A 2)
X     push down (第1引数, 第2引数, 帰り番地)
      Call A (A 1)
      Call B (A の結果)
      if B の結果=0, go EXIT
      Call X (A 2, B の結果)
      Call A (B の結果)
EXIT  帰り番地の pop up
      帰り番地セット
      exit
```

図3.3 An example of a recursive rontine

```
X(A 1, A 2)
X     push down (第1引数, 第2引数, 帰り番地)
      Call B (A 1)
      if B の結果=0, go EXIT
      B の結果のストア⟶push down (B の結果)
      Call X (A 2, B の結果)
      B の結果のロード⟶B の結果を pop up
      Call A (B の結果)
EXIT  帰り番地を pop up
      帰り番地セット
      exit
```

図3.4 Example of list head protection

ればならないことがよくある. このリストは，garbage collector あるいは再帰的呼び出しによってこわされる可能性があるので，保護する必要がある.

garbage collector は，未使用リストがなくなったときに稼動する. しかし，この稼動要求は，各処理ルーチンで発生し，しかもそのルーチンのいくつかの箇所で発生が予想されるので，その各場所で必要リストを点検し，これを garbage collector から保護するのは容易なことではない. だが，これはスタックを使用することで簡単に解決がつく. 一時的な記憶としてワークエリアを使用するのではなく，スタックを使用すればよいのである. このようにすることによって garbage collector は，保護するリストの対象としてこのスタックエリアだけを監視していればよい. この方法は，garbage collector からのリスト保護だけでなく，処理ルーチンが再帰的である場合のリスト保護には特に有効な手段である. たとえば，図3.4の再帰的なルーチンで，Bの結果をある特定のワークエリアにストアするのではなくて，スタックエリアに push down しておくとよい. これによってBの結果のリストは保護される.

(4) スタック

関数の呼び出しが必ずしも再帰的ではなくてもスタックのテクニックは，広く応用がきくものである. 特に，リスト構造を追跡するときに，分岐点を記憶しておくのに使用すると有効である. LISP のリストには後向きのポインタがないので，リストを逆向きにさか

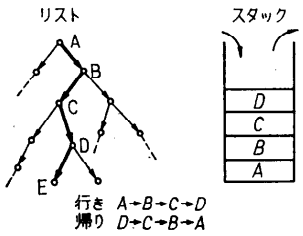

図3.5 Traversing of a list

のぼれないからである．われわれが開発した LISP インタプリタでは，read, print, garbage collector の各処理ルーチンで，このようなスタックを使用している．

4. LISP コンパイラ

LISP におけるコンパイラには文献 1) のインタプリタをベースとするものと，文献 2) のようにコンパイラをベースとするものなどがある．われわれは文献 1) の方式をとったが，文献 1) では LAP を経由して LISP 言語で開発されたコンパイラを implement している．この方法は，インタプリタと LAP さえあれば，どのマシンでもコンパイラを implement できるという点ですぐれている．しかし，コンパイラを LISP で書くということは，かなりの熟練を要するし，われわれの場合には storage の大きさに制限があったので，LISP で書いたコンパイラはコアにはいりきらないと思われた．そこでわれわれは，文献 1) の方式をとったが，LAP を作成しないで，アセンブラで LISP コンパイラを作成した．

(1) コンパイラの構成

はじめに，インタプリタの評価の方法について，

$f=\lambda(\ )(-(\div(+\ 3\ 5)\ 2))$

という簡単な例をあげて説明する．まず (−……) が引数となって評価ルーチンに渡され，そこでまた引数

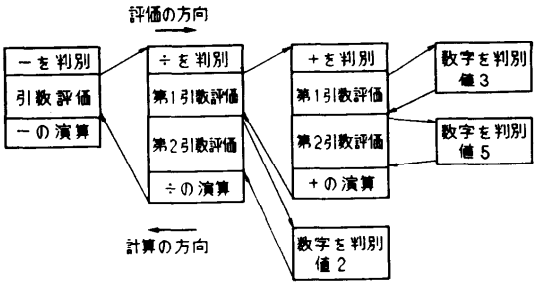

図 4.1 Computation of (− (÷ (+ 3 5) 2))

```
値3をもつロケーションをロード
値5をもつロケーションをロード
+ を実行するプログラム
値2をもつロケーションをロード
÷ を実行するプログラム
− を実行するプログラム
```

図 4.2 The object program of the expression (− (÷ (+ 3 5) 2))

の評価をするために，(÷……) を引数として評価ルーチンを呼ぶ．こうして計算可能な単位が得られるまで評価ルーチンが，何回もリカーシブに呼ばれる．図 4.1 は，評価のようすを評価順に図示したものである．各ボックスは，一つの評価ルーチンをリカーシブに呼んで，その評価ルーチンの各部分で÷を判別したり，数字を判別したりしているにすぎない．したがって，

```
compile-1[form]=[
  atom[form]→atomp[form];
  eq[car[form]; QUOTE]→cquote[cdr[form]];
  eq[car[form]; FUNCTION]→cfunct[cdr[form]];
  eq[car[form]; COND]→ccond[cdr[form]];
  eq[car[form]; PROG]→cprog[cdr[form]];
  atom[car[form]]→[
    get[car[form]; EXPR]→cexpr[[form]; expr; car[form]];
    get[car[form]; SUBR]→cfsubr[[form]; SUBR; car[form]];
    eq[car[form]; GO]→cgo[cdr[form]];
    eq[car[form]; CSETQ]→csetqc[cdr[form]];
    eq[car[form]; SETQ]→ccsetq[cdr[form]];
    eq[car[form]; LAMBDA]→clamb[cdr[form]];
    eq[car[form]; LABEL]→clabel[cdr[form]];
    get[car[form]; FSUBR]→cfsubr[cdr[form];FSUBR;car[form]];
    get[car[form]; ARRAY]→carray[cdr[form]; array];
    get[car[form]; COMMON]→ccomon[cdr[form]];
    T→error[C 5]];
  T→compile-2[form]]

compile-2[X]=prog[[ ];
      compile-1[car[X]];
      makob[ ];
      arglist[cdr[X]];
      makob [ ]]

  atomp[X]=null[X]→makob[NIL];
           mumberp[X]→makob[X];
           sslist[X]→makob[value of X];
           get[form; APVAL]→makob[apval];
           solist[form]→makob[value of X];
           T→error [C 3]]

ccond[X]=[null[EFLAG]→makob[ ];
         null[PFLAG]→makob[ ];
         null[X]→NIL;
         T→prog[[u]
              u: X;
           A  compile-1[caar[u]];
              makob[ ];
              compile-1[cdar[u]];
              [null[cdr[u]]→NIL];
              u: =cdr[u];
              go[A]]]

clamb[X]=prog[[ ];
         olist:=nconc[cons[function name; NIL]; olist];
         molist[ ];
         compile-1[cdr[X]];
         unbind[ ]]

arglist[X]=prog[[u];
           u: X;
           makob[ ];
        A  [null[u]→go[B]];
           compile-1[car[u]];
           makob[ ];
           u: =cdr[u];
           go[A];
        B  makob[ ]]
```

図 4.3 Meta-expression of LISP compiler

プログラムは一つの機能を果たすものは，評価ルーチンの一部ですんでいる．その結果，プログラムはこの図より整然とした構成をもつ．図4.1のように計算をすると (3+5) の計算⟶(前の計算結果 ÷2) の計算⟶(−前の計算結果) の計算，となって値を −4 と出力することになる．

これまで述べたことについて，評価するという立場ではなく，評価をコンパイルと置き換えてみると，図4.2のように f の計算を実行するオブジェクトプログラムが出力されることがわかる (実例については図5.1に詳細な説明をつけておいたので参照のこと)．このことから帰納すると，コンパイラのシンタックスアナライザは，インタプリタの評価ルーチンとほぼ同じ構成にすればよいことが推測でき，このようにしてわれわれが使ったコンパイラの meta-expression を得る．

(2) コンパイラの meta-expression

コンパイラの meta-expression を図4.3に示す．

図のM式のなかで使用されている処理ルーチン名について説明する．

compile-1:=コンパイル処理
atomp:=アトム 処理
cexpr:=EXPR 処理
cfsubr:=SUBR, FSUBR 処理
cquote:=QUOTE 処理
cfunct:=FUNCTION 処理
ccond:=COND 処理
cprog:=PROG 処理
cgo:=GO 処理
csetqc:=CSETQ 処理
ccsetq:=SETQ処理
clamb:=LAMBDA 処理
clabel:=LABEL 処理
carray:=ARRAY 処理
ccomon:=COMMON 処理
compile-2:=car[form] がアトムでない処理
makob:=オブジェクトプログラム作成処理
arglist:=引数処理
sslist:=特殊変数処理
solist:=通常変数処理
molist:=変数バインド処理
unbind=変数のバインドを解く処理

次に上のM式について，インタプリタの評価ルーチン (eval) と対応させて特に異なる点について説明する．

(i) 値について

インタプリタの評価ルーチンでは，各処理ルーチンの値が，その評価ルーチンの値となった．しかしコンパイラの compile-1 は，値を求めるのではなく，オブジェクトプログラムを作成することが目的なので，これ自身が値をもたない．したがって各処理ルーチンも基本的には値をもたない．

(ii) SUBR の処理について

インディケータ SUBR をもつ関数については，インタプリタでは，引数の評価を評価ルーチン (eval) が受けもっているので，その処理ルーチンは，引数を与えられた固定的なものとして，その処理を実行するだけである．したがって，コンパイラはこの処理ルーチンを利用することができ，かつインディケータ SUBR をもつ関数のオブジェクトプログラムを作成する必要がなく，インタプリタの処理ルーチンを呼ぶオブジェクトを作成すればよいことになる．

(iii) 関数部分が変数の場合について

関数部分が変数の場合，インタプリタでは，a-list によって具体的な関数名を与えられる．このような場合，コンパイラではコンパイル前にあらかじめ COMMON の宣言がなされなければならない．このことから，インタプリタのM式でこの処理を行なっている部分は，コンパイラではコモン処理になる．したがって，形式上インタプリタのM式に対応しているものは，コンパイラでは C5 のエラーを出力していて，インタプリタをM式で記述している部分と若干形式的な差がでてくる．

(3) コンパイラの再帰的構造

LISP 関数は，引数に対して値を出力するという機能をもっている．このことをプログラムサイドでみると，インタプリタの関数処理ルーチンは，3個以内の引数を受け取り，なんらかの操作を加えたあとで値を返してやるということになる．このように引数に対してだけ操作を加えるという性質は，関数処理ルーチンが再帰的な構造をもつとき，その制御を容易にさせる一因となる．基本的なプログラム構成としては，前にも述べたように，引数と帰り番地をスタックすればよいからである．

このことをコンパイラについてみると，ほとんどの処理サブルーチンが値を計算するということにかわって，マシンコードを作成するということがその機能になる．しかし各処理サブルーチンを呼ぶ場合には，インタプリタと同様，リスト構造の引数を基本的には1個渡せばよい．したがって再帰的呼び出しの制御という面からみるとインタプリタの場合とほぼ同じになり，インタプリタの再帰的呼び出し制御ルーチンを全面的に使用できることになる．

(4) 変数の取り扱い

インタプリタとコンパイラの構成のテクニックは，

基本的には同じであることを述べたが，細部の処理になるとかなりようすが異なっている．たとえば変数の扱いなどは，大きく異なる．

インタプリタでの変数の概念は，インディケータ APVAL で値が与えられる自由変数と，a-list と呼ばれる push down 上で値が決定される束縛変数（通常変数）に限定されていた．しかしコンパイラでは，コンパイル時と実行時で変数の状態が異なるので，このような方法をとるわけにはいかない．コンパイラでは，変数は，固定のロケーションに変換される．このため，変数の値は，個々の変数に割り当てられたロケーションをさすことによって得られる．しかし通常変数では，オブジェクトプログラムが再帰的である場合，変数の値も push down しなければならないので，a-list にかわるものとして個々の変数に対してスタックがとられる．なかでも変数の扱いが特にインタプリタと大きく異なる点は，次のような場合にコンパイル前に宣言が必要なことである．自由変数は，コンパイル時にインディケータ APVAL をもつことが期待できないので，実行時にそれが与えられるということをコンパイラに知らせなければならない．また通常変数でも関数部分に現われ，関数とバインドされるものは，実行がインタプリタにまかされるので特殊処理をしなければならず，このような場合も，あらかじめコンパイラは知っておかなければならない．このように変数の扱いは，コンパイラではインタプリタよりかなりめんどうになる．

5. コンパイルの例題

n! を求めるプログラムを用いて，関数をコンパイルし，オブジェクトプログラムを作成するようすを考察する．

n! を求める関数を，再帰的に定義する．

```
DEFINE ((
        (FACTORIAL(LAMBDA(N)(COND
        ((ZEROP N)1)(T(TIMES N
        (FACTORIAL(SUB1 N)))))))))
```

この定義のなかで使われている関数 ZEROP, TIMES, SUB 1 は，それぞれ SUBR, FSUBR, SUBR で登録されているシステムサブルーチンである．

この関数をコンパイルするときのコンパイル処理ルーチンと，そのルーチンに渡される引数，出力されるオブジェクトプログラムの関係を図 5.1 に示す．

図示されている処理ルーチンの概要を説明する．

（i） COMPILE

コンパイル処理の前処理ルーチンで，インタプリタから呼ばれる．以下の処理ルーチンは，コンパイル解析ルーチンでこのルーチンから呼ばれている．

（ii） COMPILE-0

オブジェクトプログラムを作成するための前処理，後処理を行なうルーチンである

（iii） COMPILE-1

関数を定義しているＳ式の解析を行ない，それぞれの処理ルーチンにコントロールを渡す処理を行なうルーチンである．この処理ルーチンは，再帰的に定義されている．

（iv） LAMBDA 処理

LAMBDA 変数の処理を行なうルーチンである．

（v） COND 処理

各ステートメントごとに処理していく．このルーチンでは，ステートメントのコントロールだけで，各要素の解析は，COMPILE-1 ルーチンで行なう．

（vi） SUBR 処理

インディケータ SUBR で登録してある関数を CALL するオブジェクトプログラムを作成するルーチンである．

（vii） FSUBR 処理

インディケータ FSUBR で登録している関数を CALL するオブジェクトプログラムを作成するルーチンである．ここで処理される FSUBR の関数は，特殊処理の必要がなく，処理が SUBR の関数と同じ性質の関数である．

（viii） 引数処理

関数に与えられる引数の処理を行なうルーチンである．

図示してある処理ルーチンの関係は，右方向にネストしていくようすを示し，左方向にそのネストを解いていくようすを示してある．処理ルーチンの流れをみればわかるように，COMPILE-1 ルーチンで，該当処理ルーチンにコントロールが渡されても，その引数が解析しているＳ式の最小コンポーネントでなければ，その処理ルーチンから，そのＳ式を解析するために再び COMPILE-1 ルーチンが呼ばれる．最後まで解析した時点で必要な処理をしながら，次々に呼ばれたサブルーチンにコントロールをもどす．これは，引数であるＳ式をみても自明である．

再帰的使用がなされているため，各処理ルーチンは，push down と pop up をしている．図では，右方向

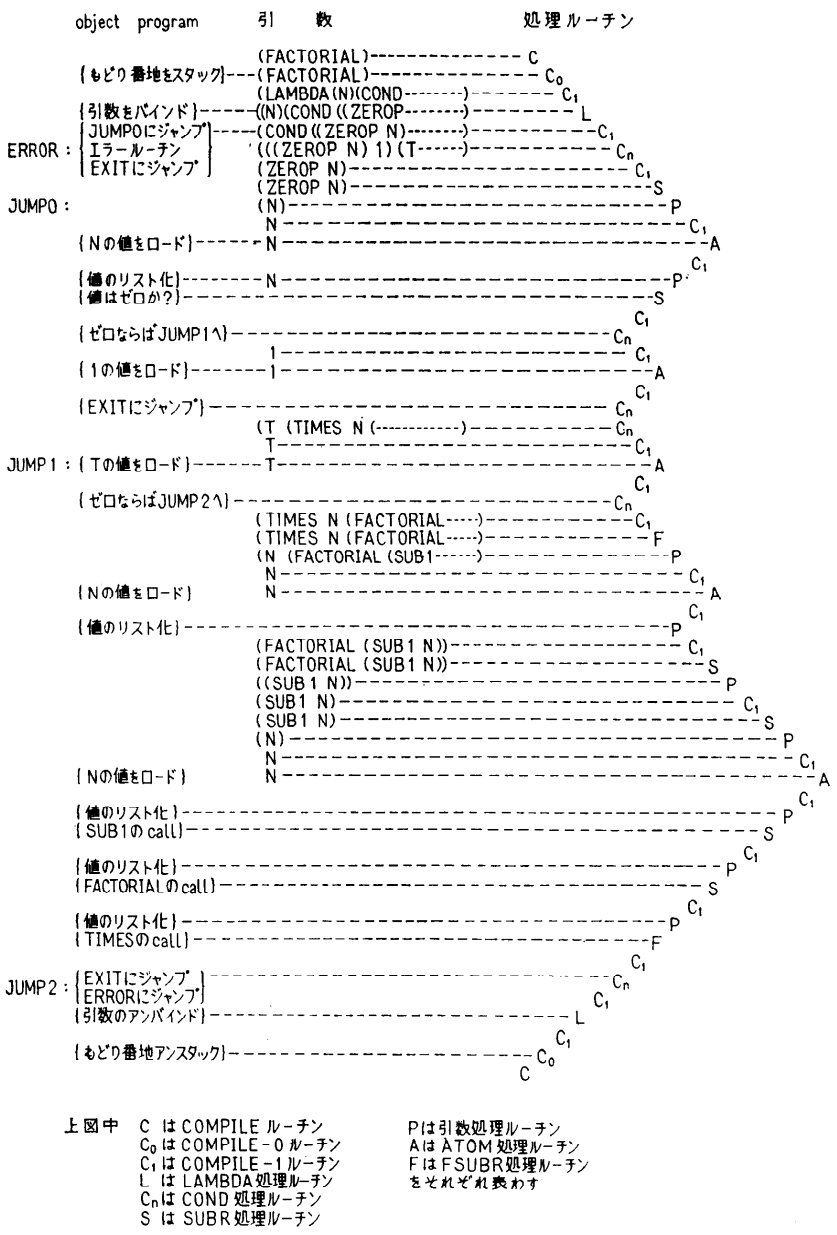

図 5.1 Compiler's behavior

に処理するときに，必要な情報を push down し，左方向に処理するときに pop up をする．この図を縦に見ると，必ず処理ルーチン名が対になっており，push down と pop up のようすがうかがえる．でき上がったオブジェクトプログラムは，図 5.1 の左側に示したとおりである．

6. インタプリタとコンパイラの比較

一般にインタプリタは，コンパイラに比べて実行速度がおそいといわれている．この理由は二つ考えられる．インタプリタでは，変数が a-list 上でバインドされており，変数値決定のため随時 a-list を捜す手間が

表6.1 ソートのプログラムにおけるコンパイラとインタプリタの比較

ソートの個数＼項目	コンパイラ				インタプリタ			
	EXCISE 実行前	ガーベージ回数	EXCISE 実行後	ガーベージ回数	EXCISE 実行前	ガーベージ回数	EXCISE 実行後	ガーベージ回数
20個	17秒	6	14秒	2	50秒	8	48秒	4
40個	1分2秒	23	54秒	12	3分14秒	32	2分58秒	15
60個	2分22秒	55	2分1秒	26	push down エリアオーバで実行不可		push down エリアオーバで実行不可	
100個	7分21秒	182	5分44秒	78	push down エリアオーバで実行不可		push down エリアオーバで実行不可	

```
equal[x;y]=[atom[x]→eq[x;y]; atom[y]→NIL; equal[car[x];
        car[y]]→egual[cdr[x]; cdr[y]]; T→NIL]
append[x;y]=[null[x]→y; T→cons[car[x]; append[cdr[x]; y]]]
smaller[x;y]=[greaterp[y;x]→x; T→y]
minm[l]=[null[y]→NIL; null[cdr[l]]→car[l]; T→smaller[car[l];
        minm[cdr[l]]]]
difflist[a;x]=[null[x]→NIL; equal[a;car[x]]→difflist[a;cdr[x]];
        T→cons[car[x]; difflist[a; cdr[x]]]]
sequence[l]=prog[[u; v; w];
        u:=l;
        v:=minm[l];
        w:=NIL;
      A[null[u]→return[w]];
        v: =minm[u];
        u: =difflist[v;u];
        w: =append[w;list[v]];
        go[A]]
```

図6.1 Definition of the sort program : sequence

かかること，また関数定義体が p-list 上にあるので必要に応じて p-list を捜し，そのS式を解析するためである．

コンパイルをするとこれらの手間が軽減され，さらにインタプリタに比べると，その関数が専有する Free Storage List が少なくなるので garbage collection の発生回数が減るという間接的効果もある．

本システムでは，インタプリタとコンパイラのプログラムの実行速度の差をテストプログラムを使って調べてみた．その結果が 表6.1 である．

このテストプログラムは 図6.1 に示すもので，コア・ソートする関数のプログラムであり，5個の関数によって定義してある．ここでインタプリタとコンパイラとの差になる部分は，p-list 上にある関数を捜し，それを評価解析する時間である．この関数は，引数が1個なので，a-list を捜す時間は問題にならない．本システムでは，再帰的呼び出しを制御するための push down エリアの大きさは固定なので，あまり複雑な再帰を行なうプログラムはスタックオーバーフローとなる．インタプリタでは，関数はS式のままの形でたくわえられているためその評価のために push down エリアの消費が多いが，コンパイラの場合は，S式を評価しないので，ある関数に対して push down エリアの消費は1個ですむ．したがって，ソートの個数が60個以上になると，インタプリタでは push down エリアがあふれて実行可能になってもコンパイルされたプログラムでは，それが生じないという差がでる．

Free Storage List は，EXCISE 実行前は 5.3kW，EXCISE 実行後は 8.3kW である．EXCISE を実行することで garbage collection の発生回数が約半分になる．

次に，実行時間を比べてみると，コンパイルされたプログラムの実行速度は，インタプリタのそれの約3倍ぐらいになっている．

7. あとがき

本システムの作成にあたっては，日本電気株式会社中央研究所コンピュータ研究部，研究マネージャー木地和夫氏のご協力をいただいた．ここに感謝の意を表したい．

参考文献

1) LISP 1.5 Programmer's Manual, The Computation Center and Research Laboratory of Electronics, MIT Press (1968).
2) Clark Weissman : LISP Primer, A self tutor for Q-32 LISP 1.5, SDC.

(昭和46年6月26日受付，7月26日再受付)