C言語処理系 (Version 2) について

木村　宏*・久米健司**

*岡山理科大学情報処理センタ

**岡山理科大学大学院応用数学専攻

（昭和58年9月20日　受理）

1. はじめに

C言語は1972年 D. Ritchie により Bell 研究所で開発された．汎用言語ではあるが，当初 UNIX* オペレーティング・システムの作成に用いられたため，むしろシステム記述言語として高く評価されている．C言語の規模は Pascal 言語と同程度であるが，簡潔な表現形式，構造化プログラミングを意識した制御構造，豊富な演算子群，関数の再帰呼出し，分割コンパイル機能などを備えており，1970年代後半以降の代表的言語の一つである．最近では UNIX と共に16ビット・パーソナル・コンピュータ（MC 68000 系）の主要高水準言語として普及しつつある．

筆者らはシステム記述言語としての効果を期待して，ターゲット・マシンを MELCOM COSMO 700Ⅲ/800Ⅲ とするC言語処理系（Version 1）を作成し，さらにこれの機能を充実させ，コード生成部に改良を加えて，ほぼ実用に耐える Version 2 を作成した．

本処理系の作成に当たって，次の点を考慮した．

1) フルセットのCコンパイラとする．

標準言語仕様として B. Kernighan[1] らの仕様を設定し，少くともこれを満たすものとする．これを越える拡張機能は次版以降で具体化することとし，まず互換性の高いフルセットCコンパイラを実現させる．

2) 実用に耐える処理系とする．

筆者らの現在のプログラミング環境において，他の言語処理系と同等以上の実用性を提供できるよう，オブジェクト効率（特に時間効率），操作性，保守の容易性に重点を置く．

3) ターゲット・マシンのハードウェア特性を生かしたオブジェクト生成を行う．

一般に，コンパイラの移植性とコンパイル速度とは同時に満たしがたい要求である．2) とのバランス上，本システムではマクロ・アセンブラでネイティブ・コードに変換できる程度の，ターゲット・マシンに近いレベルの中間言語を設定し，中間コード生成フェーズで最適化を行い，ターゲット・マシン上での処理効率が上がるようにする．

4) 処理系の記述言語は高水準言語を用いる．

---

*UNIX は Bell 研究所の登録商標である．

保守の容易性，及び本システムを次版のブート・システムとするため，コンパイラのロジックが明解に表現できる高水準言語を使用する．

本論文では，2章で処理系の構成，3章で効率改善を考慮したオブジェクト生成法，4章で処理環境について述べる．5章では本処理系の性能について検討する．

2. 処理系の構成

本処理系は図1に示すように，C言語で書かれたソース・プログラムをネイティブ・コードに変換するコンパイラ本体と，実行環境としての実行時ライブラリ・ルーチン群から成る．

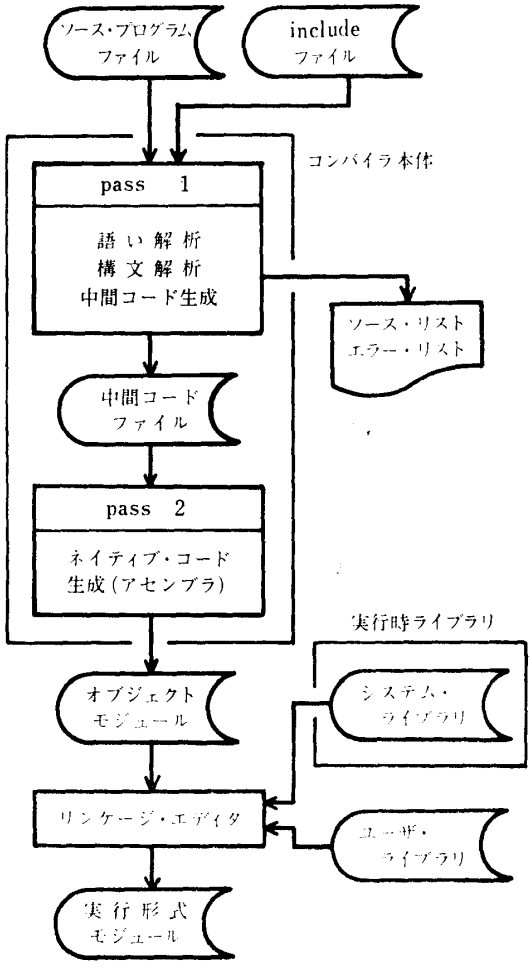

図1. C言語処理系（Version 2）の構成

2.1 コンパイラの構成

一般にコンパイラの構成は，設計目標に大きく影響される．本システムでは前述の目標を実現するため，次のような構成とする．

1) コンパイラ本体を2パス編成とし，パス1で語い解析，構文解析，及び中間コードへの変換を行う．パス2で中間コードをホスト・マシン共通のリロケータブル・オブジェクト・モジュールに変換する．機能分散と作成のしやすさから，両パスは独立したプログラムとするが，利用者の入力コマンドを減らすため，ホスト・マシンのコマンド処理機能

（コマンド・プロセッサ）を用いて，単一コマンドで起動できるようにする．

2） C言語仕様上のマクロ命令（代入型マクロ定義（define 文）とテキスト・ファイルの挿入機能（include 文）など）は，独立パス（プリプロセッサ）とせず，語い解析部の機能としてパス1に組み込む．これにより

- ソース・リスティングの合理化
- Pascal 風のエラー位置の明示[6) 7)]
- トークン走査の高速化（重複の抑制）

が可能となる．今後ホスト・マシン上で UNIX 流のパイプライン機能が実現できれば，この部分は別タスクとして独立させてもよい．

3） パス1は保守の容易性を重視し，主に Pascal 言語で言述し，パス2はC言語自身で記述する（表1参照）．Pascal 処理系の分割コンパイル機能の乏しさは問題であるが，データ構造の選択とモジュール化に留意することでこの欠点を補うことにする．Version 1 ではパス1のファイルの入出力部分を Fortran 77 で記述していたが，Fortran ライブラリの空間効率の悪さからアセンブリ言語を用いて書き換えた．また，パス2は逆に Version 1 ではマクロ・アセンブラを直接用いていたが，時間効率の悪さから，C自身でブーティングを行い改善をはかった．

4） 構文解析法として，作成と変更の容易な再帰的下向き構文解析法[6) 7) 8) 9)] を用いる．標準言語仕様の細かい解釈にはいくつかの疑問点があったが，基本的に Algol 系の言語であるので，構文解析法としてはこれで十分である．本システムでは1パスで中間コード生成を行うため，構文解析アルゴリズム中に局所的最適化機能をいかにして組み込むかが，重要な検討課題となった．これについては次章で述べる．

表1． 処理系のモジュール概要

モジュール名		記述言語	行数	機能
コンパイラ	パス 1	Pascal	6500	ソース・プログラムから中間コードへの変換
		Assembler	400	ファイルの入出力
	パス 2	C	1000	オブジェクト・モジュールへの変換
ライブラリ	低水準	Assembler	600	基本機能（OS とのインタフェース）
	高水準	C	1200	ファイルの入出力など
	基本関数	C	120	初等関数の近似計算
標準ヘッダ		C	45	標準マクロ定義集
コマンド処理定義		コマンド記述言語	40	処理系起動コマンド・パラメータ

2．2 中間言語

パス2への入力となる中間言語コードとしては，Pascal のPコード[10)]のように機械独

立な体系[4]を取るか，あるいは完全にターゲット・マシンに依存した体系を取るかの両極端が考えられる．今回のシステムでは，16個の機能分担された汎用レジスタをもつワード・アドレス・マシンという，レジスタ・マシンとしては古典的なコンピュータをターゲットとしなければならないため，次のことを考慮した．

1) ネイティブ・コードへの変換が容易である．

2) 機能の重複が起こらないコード体系とする．

3) 単純な書式のテキスト・ファイルを出力する．

1)と2)を考慮し，翻訳パタンを検討した結果，標準的な43種類の機械命令と10種類の疑似命令だけで，十分効率の高いネイティブ・コードを生成できることがわかった．機械命令は，実行制御と4タイプの基本データの移動及び演算・操作から成る．疑似命令はデータの生成と，分割コンパイルに伴う外部名の定義及び参照から成る．一般形式は

[label] op [, r] arg [, arg, ...]

とする．ここで，label は記号名，op は命令ニモニック，r はレジスタ番号，arg は値またはアドレス式である．

3)の選択により，Version 1 から 2 への改版，コンパイラ作成時のデバッグ作業，及び最適化の効果の検討が簡単になる．中間ファイルは行単位の記録から成る順ファイルとし，通常はコンパイル終了時に自動消滅させる．

3. オブジェクト生成

3.1 ターゲット・マシンの特性

ターゲット・マシンは基本的にワード・アドレス・マシンである．アドレス空間として別途，バイト，ハーフ・ワード(2バイト)，ダブル・ワード(8バイト)の3空間をもつ．サイズはいずれも512K バイトである．ハード的にはベース・レジスタを用いて16Mバイトまで拡張できるが，今回はサポートしていない．

汎用レジスタはR 0からR15までの16個があり，全て1ワード長(32ビット)である．この中でR 1からR 7までがインデックス・レジスタとなる．

ハードウェアが処理する基本データは，アドレス空間と同じ4種類ある．他に255バイトまでのバイト列も扱えるがC言語ではサポートしない．

3.2 データ・オブジェクト

C言語のデータ型は，整数型 (char, short, int, long, unsigned)，実数型(float, double)，ポインタ型，配列型，構造型 (struct, union)，関数型である．これらのうち基本型とポインタ型のサイズを表2のように定めた．配列型のサイズは要素型のサイズに要素数を乗じた値となる．構造型は各要素のサイズの合計を含む最小の64ビット・ブロックとする．

なお，文字集合は EBCDIC 体系の全文字を扱う．

表2．基本データ型のサイズ(ビット数)

型　名	サイズ	型　名	サイズ
char	8	unsigned	32
short	16	float	32
int	32	double	64
long	32	pointer	32

3．3　アドレス割り付け

ターゲット・マシンの性格上，ワード長以外のデータをアクセスする場合，そのロケーションによってはインデックス修飾が不可欠となる．本コンパイラでは，オブジェクトの単純化(高速化)をはかるため，インデックス修飾をできるだけ減らすことを考え，ワード長以下のデータはワード境界，それよりも長いデータはダブル・ワード境界から始まるよう，アドレス割り付けを行った．

ただし，配列だけは連続領域の割り付けが望ましいので，要素型に従った境界調整を行わない．

アドレス割り付けの具体例を図2に示す．

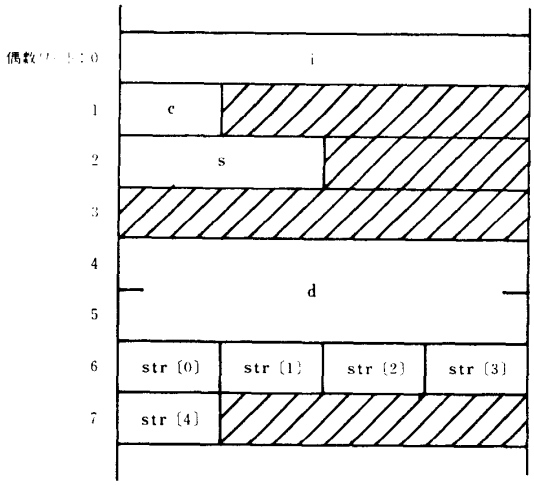

図2．アドレス割り付け例（宣言は下記）

```
int i; char c; short s;
double d; char str[5];
```

3．4　スタック

C言語は関数の再帰呼出しを許すので，関数引数，自動変数，作業領域などにスタックを用いてインプリメントする．ターゲット・マシンには，ハードウェア・スタックが用意されているが，ベース・ポインタがないこと，ワード長以外のデータの取り扱いが困難なことから，本コンパイラではソフトウェア・スタックを用いる．具体的には，異なるデータ長に対応した3種類のベース・ポインタを設けて，それを汎用レジスタに割り当てる方

式をとる．これにより，ローカル・データのアクセスが単純化され，アクセス効率がよくなる．

3．5　レジスタ割り付け

汎用レジスタに関する残りの制約条件は，ダブル・ワード・データを偶数・奇数のペア・レジスタで扱う必要があることと，整数データに対する演算命令の一部に偶数と奇数レジスタのどちらを指定するかで，機能が異なるものがあることである．したがって，本システムでは表3のようにレジスタの用途を定め，ハードウェアの特殊性を回避するようにした．

前節のスタック用ベース・ポインタは表3のとおり3本用意し，利用度の低いハーフ・ワード用ベース・ポインタは必要に応じてポインタ用レジスタでシミュレートする．この部分は機種依存性が強いが，本システムの int 型がワード長であることから全体の性能に与える影響は少ない．これらのベース・ポインタは常にスタック上の同一ロケーションを，それぞれのアドレス値でポイントする機能をもっている．

表3．レジスタ割り付け

レジスタ番号	用　途
R0	ワーク・レジスタ
R1，R2，R3	スタック・ベース・ポインタ (順にワード用，ダブル・ワード用，バイト用)
R4，R5， R6，R7	ポインタ用レジスタ(インデックス・レジスタ) R7はポインタ関数値用
R9，R11， R13，R15	整数データ用レジスタ R9は整数関数値用
R8−9，R10−11， R12−13，R14−15	実数(ダブル・ワード)データ用レジスタ R8−9は実数関数値用

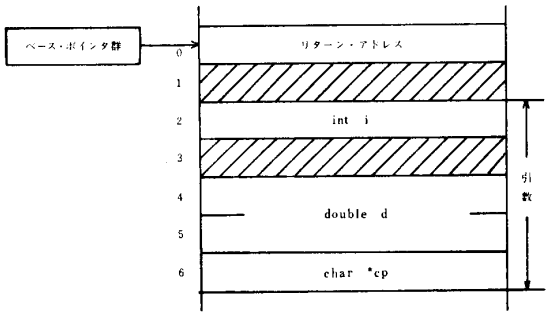

図3．関数の先頭でのスタック割り付け例

関数は func (i, d, cp)
　　int i; double d; char *cp;
　　{ /* function body */ }
斜線の箱は境界調整に伴なう空きエリア

3．6　関数と関数呼出し

C言語では，一まとまりの機能単位を関数のみで表現する．主プログラムも，実行時ライブラリ中の真の主プログラムから呼ばれる関数として実現する．さらに，引数の受渡しが基本的に値取りしかないので，関数の先頭でのスタック割り付けは簡単である．本システムにおける仮引数と自動変数の割り付け例を図3に示す．

関数呼出しは次の手順でスタック操作を行う．

①使用中のレジスタをスタック上のレジスタ退避領域へ格納する．（図4のⓐ）

②実引数を呼ばれた関数の仮引数領域に積み込む．（図4のⓑ）

③各ベース・ポインタを変更する．（図4のⓒ）

④関数へ制御を移す．

⑤関数側のスタック要求量を満たすかどうか検査する．

⑥復帰アドレスをスタックへ格納する．（図4のⓓ）

⑦関数の処理終了後，復帰アドレスを参照して呼出し側へ復帰する．

⑧ベース・ポインタを旧状態へ復元する．（図4のⓔ）

関数値は，整数，実数，ポインタのそれぞれに対し一定のレジスタに格納して返す．

モジュール間のインタフェースが簡単なため，他言語で記述されたモジュールとの連結も容易である．

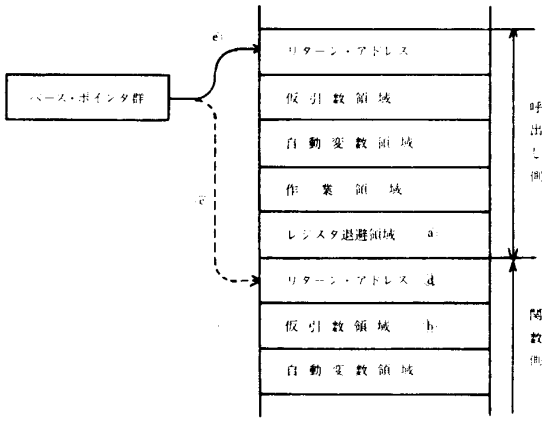

図4．　関数呼出し時のスタック配置

3．7　コード生成

文及び式の構文解析には，再帰的下向き解析法を用いる．文の構文は Algol 系の定義がなされているので，この方法が単純で実現しやすい．一方，式に対しこれを適用すれば，再帰レベルの上下が増え処理速度が落ちる可能性があり，演算子順位法[4) 8)] の採用も検討したが，次の理由から上記の方法をとることにした．

①式は豊富な演算子を用いるため簡単な構造となる場合が多く，処理速度への影響は少ない．

②構文解析フェーズから直接中間コードを生成するので，このアルゴリズムが有利であ

る．

③この方法を用いても，コード生成のタイミングを工夫すれば，局所的最適化が可能である．

最適化のねらいは無駄なコード生成を押えることであり，Version 1 のオブジェクトを検討した結果，以下の部分の改善で効率が改善されることが判明した．局所的最適化法の基本的アルゴリズムは次のとおりである．式の中からコード生成可能な構文単位を認識した段階で，最適化可能なパタンを検出すれば，すぐコード生成を行わず未生成のままスタックに登録しておき，解析が進行した段階でスタックより取り出しコード化する．

ここで，本コンパイラに組み込んだ主な最適化機能について示す．

1)　レジスタ利用の最適化

ポインタ用と演算用のレジスタを複数個割り当てて，レジスタの退避命令を減らす．また翻訳パタンを工夫してレジスタ間のデータ移動を減らす．

2)　定数式のコンパイル時評価（次節参照）

3)　アドレス計算の最適化

配列の添字式や構造体のオフセット値が単一の定数に帰着する場合はコンパイル時に評価し，インデックス修飾を有効に利用する．

4)　オペランドの評価順序の変更

可換な二項演算では，オペランドの評価順序を入れ換えるとコード数が減る場合があるので，これを利用する．

5)　状態コードの活用

生成された命令による状態コード変更の有無をコンパイラに認識させて，構造を持つ文などのコード生成を最適化する．

[例]　i＞j　(i, j, k は int 型とする)

この式が k＝i＞j; のように使用されると，比較結果に応じて1(真)または0(偽)の値を生成しなければならないが，if (i＞j)・・・ のように使用される場合は，値の生成部分は不要である．この場合には6命令削減できる．

3．8　定数式の処理

ここでは定数式の認識と処理について，加法式を例にとって述べる．

加法式の構文図を図5に示す．この構文に対する処理アルゴリズムを Pascal 風に記述

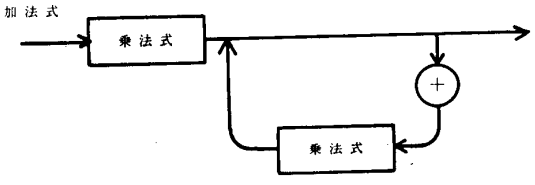

図5．　加法式の構文（演算子は＋のみに限定）

すると図6のようになる．図6の e1 と e2 は部分式に関する情報を指すポインタとし，codegen と constoperate は演算コードを生成する関数と定数計算を行う関数とし，expr は部分式を評価する手続きとする．

```
expr (e1) ;
while opr = PLUS do begin
    expr (e2) ;
    if (e1^. kind = CONSTANT) and
      (e2^. kind = CONSTANT) then
       e1 : = constoperate (PLUS, e1, e2)
    else
       e1 : = codegen (PLUS, e1, e2)
end
```

図6． 加法式の処理

```
e3 : = nil ;
expr (e1) ;
while opr = PLUS do begin
    expr (e2) ;
    if e2^. kind = CONSTANT then
       if e3 = nil then
          e3 : = e2
       else
          e3 : = constoperate (PLUS, e3, e2)
    else if e3 = nil then
          e1 : = codegen (PLUS, e1, e2)
    else begin
          e1 : = codegen (PLUS, e1, e3) ;
          e1 : = codegen (PLUS, e1, e2) ;
          e3 : = nil
    end
end ;
if e3 < > nil then
    if e1^. kind = CONSTANT then
          e1 : = constoperate (PLUS, e1, e3)
    else
          e1 : = codegen (PLUS, e1, e3)
```

図7． 加法式の処理（改良版）

```
C – compiler version 2.0
1.  /* Convert lower to upper */
2.  #include <stdio. h>
3.  main ( ) {     int c ;
4.        while ((c = getchar ( ))  ! = EOF)
5.            if (c  > = 'a' && c  < = 'z')
6.                putchar (c–'a' +'A') ;
7.            else
8.                putchar (c) ;
9.  }
```

図8． C言語のプログラム例（ソース・リスト）

```
                SYSTEM      ASMBLC
                CSECT       1
main            CI, 1       #STACKEND-#F1
                BG          #STKOVF
                STW, 15     0, 1
#L2             EQU         ¥
                FCL, 4      0+getchar
                STW, 9      2, 1
                CI, 9       -1
                BE          #L3
                LW, 9       2, 1
                CI, 9       129
                BL          #L6
                LW, 9       2, 1
                CI, 9       169
                BG          #L4
#L6             EQU         #L4
                LW, 9       2, 1
                AI, 9       64
                STW, 9      6, 1
                FCL, 4      0+putchar
                B           #L5
#L4             EQU         ¥
                LW, 9       2, 1
                STW, 9      6, 1
                FCL, 4      0+putchar
#L5             EQU         ¥
                B           #L2
#L3             EQU         ¥
                LW, 5       0, 1
                B           0, 5
#F1             EQU         7
#STATIC         CSECT       0
                REF         #STACKEND
                REF         #STKOVF
                REF         putchar
                REF         getchar
                DEF         main
                REF         _strtab
                REF         _iob
                REF         alloc
                REF         fopen
                END
```

図9. コンパイル結果（中間コード）

しかし，この方法では i+5+3 という式の定数部分を評価できない．そこで，本システムでは図7のようにアルゴリズムを改良している．

最後に，本コンパイラのパス1の翻訳例を示す．図8はソース・プログラム，図9はパス1の出力結果である．

4. 実行環境

本処理系に係わる実行環境は，実行時ライブラリとコマンド処理系とリンケージ・エディタが主なものである．

実行時ライブラリは UNIX 上のCライブラリに準拠し，ライブラリ・ルーチン名とその機能を一致させることで，ソース・プログラムの移植性を高める．ルーチン群は表1に示すように，次の3種類に大別できる．

①低水準ライブラリ

②高水準ライブラリ

③基本関数ライブラリ

①はオペレーティング・システムの各種サービスとのインタフェースを行うルーチン群を含み，主なものは入出力とメモリ管理である．②は①の上位モジュールで，Cユーザのプログラミングを支援するルーチン群を含む．これはC自身でコーディングしてある．③は数値計算用に準備したライブラリで，三角関数などを含む．

コマンド処理系は，コンパイルからロード・モジュールの作成までを単一コマンドで連続処理させるためのシステムである．ホスト・マシンのコマンド・プロセッサを用い，それに与えるパラメータ群を登録してある．C言語処理系の操作性はこれで決まるので，他の言語系の起動方式と一致するよう配慮してある．

通常，自作システムでは独自のリンケージ・エディタを作成し，閉じた処理環境を設定する場合が多い．しかし，本処理系ではパス2の出力をホスト・マシンのリンケージ・エディタの入力形式に合せたので，別言語のモジュールとも容易にリンクできる．

実用性を指向する場合には，これらの他にプログラム開発ツール群が要求される．UNIX のCが普及している理由の一つには，ツール群の充実が考えられるので，今後清書プログラムやランタイム・デバッガなどの開発を進める予定である．

5. 処理系の性能

本処理系に適用した局所的最適化の効果を確認するために，オブジェクト・プログラムの時間効率の調査を行った．一般に言語処理系は，それぞれ個有の目的に合致した設計がなされているため，与える課題により性能差が見られる．また，同一言語であってもコーディングの仕方により処理時間に差が現われるものである．したがって今回行った測定結果だけで評価しきれるとは考えていないが，大まかな傾向はつかめるであろう．

まず測定結果を表4に示す．この実験は同一のアルゴリズムを以下の言語処理系を用いてプログラミングし，オブジェクト・プログラムの実行時間を測定する方法をとった．各言語の特性を生かした標準的プログラムを作成するよう注意し，実行に際して純粋の処理時間が得られるよう，実行時チェック機能などは省いた．

- C（本処理系）

表4. 実行時間の比較 (単位は秒)

番号	課題	C	COSMO-C	Fortran 77	Pascal
1	シェル・ソート (整数) データ数は 20000 個, 逆順	8.5	12.1	9.3	13.8
2	シェル・ソート (実数) データ数は2000個, 逆順	9.4	19.0	9.9	14.1
3	整数→実数型変換の反復 500×500回のループ	3.8	27.0	3.0	5.3
4	Ackermann 関数 : A (3, 4) 100 回反復	16.8	31.5		36.6
5	行列の積 (整数) (8,9)×(9,13) を1000回反復	16.8	25.0	17.7	37.7
6	行列の積 (実数) (8,9)×(9,13) を1000回反復	20.5	35.0	19.2	39.9

- COSMO-C[5] *(Whitesmith 系 C 言語)
- Fortran 77 (ホスト・マシンの標準的処理系)
- Pascal (Pascal 8000系)

課題①と②のシェル・ソートは配列要素のランダム・アクセス, ⑤と⑥の行列の積は定まった順序での配列要素のアクセスを行う. ③は単純なループと整数から実数への型変換である. ④の Ackermann 関数は, 再帰呼出しをテストするので, Fortran 77 は除外した.

この結果から, 本処理系は Fortran 77 とほぼ同程度の性能を有していると言える. ホスト・マシンの Fortran 77 処理系は, オプティマイズ指定でコンパイルさせたので, 主に DO ループと配列に対して最適化がなされている. C言語特有の簡潔な表現を用い, 本処理系程度の局所的最適化処理をほどこしただけで, 実用度の高いプログラム作成が可能であることがわかった.

生成されたオブジェクトを検討した結果, Pascal とCとの差は処理すべきデータをレジスタに取り込むまでのステップに起因することがわかった. Pascal は本来教育目的の言語であり, 移植性に重点を置いて処理系を作成してきた経緯がある. したがって, 本処理系のコンパイル速度をさらに改善するには, C自身を用いてコンパイラを書き替えればよい.

COSMO-C はまだ開発途上の処理系のようであり, 逐次改良されると思われるが, 本システムとの差は, ポインタの処理, 関数呼出し系列, レジスタ割り付けなどに起因して

* 本処理系とほぼ同時期に公開された, 三菱電機製のCコンパイラ (Verslon B00) である. ライブラリ以外の仕様は本処理系とほぼ同一である.

いる．

表4以外に，Cとハンド・アセンブルしたプログラムとで効率比較を行ったところ，約2.5倍差があった．筆者らは，C言語の場合にはこの差を2倍以下に改善できると考えているが，これは今後の課題である．

6. おわりに

言語自体が実行効率の向上を指向したC言語の場合には，効率面の追求は不可欠であり，本研究ではターゲット・マシンの時間効率改善を目的としたC言語処理系の作成を行った．

再帰的下向き構文解析法に局所的最適化機能を組み込むことで，十分実用に耐える処理系の作成が可能であることを確認した．出現頻度の高い中間コード・パタンの最適化により，Version 2 は旧版に較べて約20%の効率改善が得られた．

作成に要した期間は Version 1 の作成も含めて，約12人月であった．

今後は，本処理系を用いてC言語自身でCコンパイラを作成する予定である．そこでは，ターゲット・マシン上での大部分のシステム・プログラミングが可能となるよう，ライブラリの充実も含めた機能拡張を計画している．

参考文献

1) Kernighan, B. W. and Ritchie, D. M.: The C Programming Language, Prentice-Hall (1978)
2) Zahn, C. T.: C Notes: A Guide to C Programming Language, YOUDON Press (1979)
3) 石田晴久： UNIX システム入門10〜15, bit, Vol. 14, No. 8 〜 Vol. 14, No. 13 (1982)
4) 黒田，辻野，萩原，荒木，都倉： システム記述用言語Cのポータブルコンパイラの作成，情報処理学会論文誌，Vol. 21, No. 6, pp. 461-468 (1980)
5) 五月女健治，西田親生： MELCOM-COSMO マシンへのCコンパイラの移植，情報処理学会第25回全国大会，3C-11 (1982)
6) Wirth, N.: Pascal-S: A Subset and its Implementation, Pascal-The Language and its Implementation, pp. 199-260, Wiley (1981)
7) 宮本衛市： Pascal プログラミングと翻訳技法，森北出版 (1982)
8) 中田育男： コンパイラ，産業図書 (1981)
9) 中西正和，大野義夫： やさしいコンパイラの作り方，共立出版 (1980)
10) Nori, K. V., Ammanna, U., Jensen, K., Nageli. H. H. and Jacobi, Ch.: Pascal-P Implementation Notes, Pascal-The Language and its Implementation, pp. 125-170, Wiley (1981)

On the Implementation of C Language Processor (Version 2)

Hiroshi KIMURA*, Kenji KUME**

* *Computing Center, Okayama University of Science*
** *Graduate School of Applied Mathematics, Okayama University of Science*
Ridaicho 1-1, Okayama 700, Japan

Abstract

C is a general-purpose programming language which is suited to structured programming. It has the ability of system programming, as it is a relative low level language that lets users specify every detail in program logic to achieve maximum computer efficiency.

The C language processor (Version 2) containning the compiler and the run time library routines is implemented on MELCOM COSMO model 700 III / 800 III computers. As the recursive top down syntax analysis method and some local optimizing features are applied to construct the compiler, compact and efficient obiect programs can be generated. Its performance reaches the level of the optimized Fortran 77 processor on host machine.

The large part of this compiler is coded in Pascal language to make maintenance easy.