Vol. 21 No. 6　　情報処理学会論文誌　　Nov. 1980

システム記述用言語 C のポータブルコンパイラの作成†

黒田 壽祐†† 辻野 嘉宏††
萩原 兼一†† 荒木 俊郎†† 都倉 信樹††

システム記述用言語Cのポータブルコンパイラを NOVA 3 上でインプリメントした．すでに，部分仕様のCコンパイラをインプリメントし，実用に供しているが，これを用いてブートストラップにより，移植性の高い完全仕様のCコンパイラを作成した．

Cは，汎用プログラミング言語として知られているが，ビット演算やシフト演算などの低いレベルの演算子，レジスタ割付，分離コンパイル，インライン機能（アセンブリ言語の埋込機能）などシステム記述に適した特徴をもち，UNIX システムにも利用されている．Cのポータブルコンパイラを作成するにあたっては，いかにしてこれらの特徴を損うことなく移植性を高めるかという点が問題となる．

このCコンパイラは移植性を考え，ソース・プログラムを中間言語（Cコード）に変換するフェーズⅠと，このCコードを機械に適した効率のよい目的プログラムに変換するフェーズⅡに分かれている．フェーズⅠは，機械の特徴をいくつかのパラメータとして与えることによって，機械に独立に処理を行う．フェーズⅡは，機械に依存する部分の処理を行う．このコンパイラ自身Cで書かれており（約 3,500 ステップ），ブートストラップによって容易にほかの機械へ移植できる．

本論文では，主にCコンパイラの移植性について考察し，その構成とフェーズⅠで行う処理について説明する．

1. まえがき

プログラミング言語 C[1]は，構造化プログラミング向きの制御文，基本データ型（**表1**）をもとに構成される柔軟性のあるデータ構造，豊富で強力な 45 の演算子（ビット演算やシフト演算も可能—**表2**），分割コンパイルなどを特徴とするシステム記述用言語で，UNIX システム[2]にも利用されていることで知られている．筆者らは，すでに NOVA 3 上で部分仕様のCコンパイラをインプリメントし，実用に供している[3]．これを用いてブートストラップにより，高い移植性をもつCコンパイラを作成した．

このCコンパイラは，完全仕様を満たしているだけでなく，次の3点で拡張されている．

① switch 文の case ラベルの構文の拡張
② プログラムの実行を停止する stop 文の導入
③ コンマ演算子（,）と対になるダブルコンマ演算子（,,）の導入

コンパイラの作成にあたっては，次の点を考慮した．

1) 高い移植性をもつこと
2) 効率のよい目的プログラムを出力すること
3) コンパイル速度の向上

本論文では，**2**章で移植性を高めるためにコンパイラをどのような構成にしたか，特に，フェーズⅠとフェーズⅡに分けたことについて述べ，**3**章でフェーズ

表 1 基本データ型
Table 1 Basic data types.

予約語	基本データ型
char	文字型
short	短整数型
int	整数型
unsigned	符号なし整数型
long	倍長整数型
float	浮動小数点数型
double	倍精度浮動小数点数型

表 2 Cに特有な演算子
Table 2 Typical operators of C.

++	increment
−−	decrement
sizeof	data size
(type-name)	type conversion
>>	right shift
<<	left shift
&	bitwise-and
^	bitwise-exclusive-or
|	bitwise-inclusive-or
+=	add & assignment
−=	subtract & assignment
,	sequential evaluation & result right value
,,	sequential evaluation & result left value

† Portability of a C-Compiler and its Implementation by Yoshisuke Kuroda, Yoshihiro Tsujino, Kenichi Hagihara, Toshiro Araki and Nobuki Tokura (Department of Information and Computer Sciences, Faculty of Engineering Science, Osaka University).

†† 大阪大学基礎工学部情報工学科

Iの一般的な処理について述べる．4章では移植性を確保するために，フェーズIで特に考慮しなければならない問題について考察し，5章ではコンパイラを複数の変換プログラムから成るシステムとして考えた場合，ユーザが使い易いためにはその構成をどうすればよいかについて述べる．

2. Cコンパイラの構成

2.1 Cコンパイラの移植性

コンパイラの移植性に関しては，次の2つの点に着目しなければならない．

① コンパイラがどのような機械にも対応する目的プログラムを生成できるかどうか

② コンパイラ自体をほかの機械へ容易にインプリメントできるかどうか

一般にコンパイラの生成する目的プログラムは，機械に依存している．この機械による目的プログラムの違いをコンパイラで吸収するために，次のような方法が考えられる．

I] 各機械による差異をパラメータ化してコンパイラに与えることによって，それぞれの目的プログラムを得る．

II] Pコードを用いた PASCAL コンパイラ[4]のように，一度仮想的な機械のための中間言語に変換し，これを機械のインタプリタで処理，あるいはトランスレータで変換して目的プログラムを得る．

Cコンパイラについてこれらの方法を検討すると，I] の方法は，各機械の特徴を正確に反映できるという利点があるが，Cにはビット操作などのかなり低いレベルの演算子もあり，レジスタの数や命令セットなどのパラメータが多くなり，コンパイラの処理が複雑になってしまう．II] の方法は移植のための労力は小さいが，Cには変数の型や演算子の数が多く，レジスタ割付などもあるために，機械に独立でかつインタプリタを作成しやすい中間言語をどのように設定するかが問題になる．

筆者らはそれぞれの方法の欠点を補うために，I] とII] の方法を併用することにした．つまり，コンパイラは，ソースプログラムを機械に独立な中間言語(以下Cコードと呼ぶ) に変換するフェーズIとCコードを各機械に適した効率のよい目的プログラムに変換するフェーズIIに分かれており，フェーズIの処理で機械に依存する部分は，各機械のパラメータ(4章参照)をいくつか指定することによって，その機械の上でもプログラムの内部を変更せずに動くようにした．

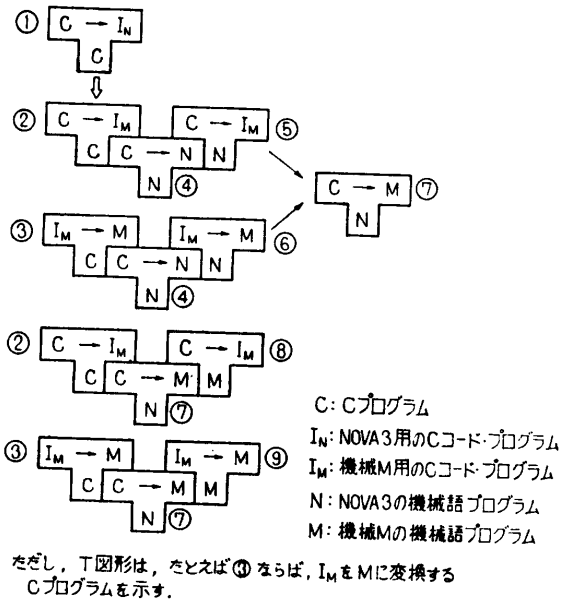

図1 移植の手順
Fig. 1 Process of transplantation.

このCコンパイラの NOVA3 (NOVA3とは限らない) からほかの機械 M への移植は，ブートストラップを用いて図1のような手順で行われる．つまり，①,④ のプログラムはすでにあり，人手を必要とするのは，次の2点のみであり，移植のための負担は軽い．

イ) ① のプログラムのパラメータ (11 種) を設定し直して(②)，コンパイルし機械 M 用のフェーズI⑤を得ること．

ロ) NOVA3用のフェーズ II の機械に依存する部分を手直しして (③) 機械 M 用のフェーズ II⑥ を作成すること．

機械 M 上でユーザ・プログラムを実行するためには，⑧ と ⑨ のプログラムを使えばよい．

2.2 C コ ー ド

Cコード (付録) は，次のような点を考慮して設定した．

1) 機械に独立な仮想スタック・コンピュータのための命令であること．

2) フェーズ II の処理を簡便にするため，単純な書式であること．

3) 機械語への変換の際に自由度があり，各機械に適した効率の良い目的プログラムを出力できること．

1)の点について説明すると，Cプログラムは再帰呼

び出しの許された関数の集まりであるが，その関数はスタック上で次のように実現される．1つの関数が呼ばれるとスタック上に**図2**のような領域が取られる．リターン・ブロックは返すべき関数値，前の MP(mark pointer- 呼んだ関数のリターン・ブロックの先頭番地)， PC (program counter- この関数から戻るときの戻り番地) を入れる領域である．変数ブロックは，引数や動的変数を入れるための領域である．すべての演算は，スタック上の作業領域で行われ，呼んだ関数に戻るとき，この関数のために取られた領域は解放される．

2)の点を考慮して，フェーズIの出力は行（改行記号で区切られる）を単位とし，次の3種類がある．ただし，( )の中の文字はその行の先頭の文字であり，行の1文字目を見ればその行の種類がわかるようになっている．

i) Cコード行(空白)．

ii) 注釈行(セミコロン)．

iii) アセンブリ行 ('A')．

注釈行は，ソース・プログラムの情報（たとえば行番号など）をフェーズ II へ伝えるためのものである．アセンブリ行は，ソース・プログラムでのインライン・コーディングを考慮したものであり，そのままの形がフェーズ II の出力となる．

Cコードをその働きによって分類すると，

イ) プログラムの実行を制御するための命令．

ロ) スタックのデータを演算，操作するための命令．

ハ) フェーズ II に対する疑似命令．

の3種類に分けられる．

Cコードは，最初の3文字がニーモニックを，次の2文字がオペランドの型を，それ以降がオペランドを表わす．その構文を拡張 BNF 記法で示すと**図3**のようになる．

また，オペランドとなる値〈value〉は，すべて 16 進表現された1語長のデータとした（この1語長はパラメータによって決定する)． したがって，1語長より大きい型のデータを扱うためには，1語単位に分割してCコードを出力する．この場合，出力されるCコードの数はふえるが，フェーズ II における〈value〉の扱いが簡単になり，汎用性も増す．

Cコードを決定する場合，Cの特徴であるデータ型や演算子の種類が多いことが問題となる．型については，ニーモニックの数をふやさず，Cコードのパラメータの1つとして実現している．また，一部の演算子(+=, -=などの代入演算—**表2**)については，そのまま対応するCコードを設けず，ほかの2つのCコードに置換えることによって解決している．このようにして設定したCコードの種類は 62 で，PASCAL の Pコード[4]と同程度である．

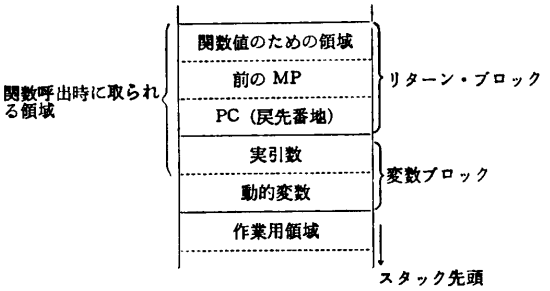

図 2 スタックによる関数呼出の実現
Fig. 2 Stack area for function call.

```
〈Ccode〉=〈mnemo〉[〈type〉[〈operand〉]ᵏ₀]ⁿ₀
〈mnemo〉=〈c〉〈c〉〈c〉
〈type〉=〈c〉␣|␣␣|〈hd〉〈hd〉
〈operand〉=〈label〉|〈value〉|〈c〉␣
〈label〉=*〈hd〉〈hd〉〈hd〉〈hd〉|[〈c〉|_][〈c〉|_|〈d〉]
〈value〉=0[〈hd〉]ᵏ₀
〈hd〉=〈d〉|A|B|C|D|E|F
〈c〉 は英字
〈d〉 は数字
[α]ᵏᵢ は αⁱ|αⁱ⁺¹|…|αʲ を意味する．ただし，α⁰は空系列．
k は1語データを 16 進表示するのに必要な桁．
```

図 3 Cコードの構文
Fig. 3 Syntax of C-code.

3. フェーズIにおける処理

3.1 宣言の処理

Cでは各変数に対して，ストレジ・クラス (storage class) と型の2つの属性を与えることができる．ストレジ・クラスには**表3**のように4種類ある．

① の extern 変数に対しては， ほかのコンパイル単位からも参照できるように，プログラム中に現れたそのままの名前を使用する．局所的変数は，有効範囲(scope)が入れ子になるのを許されており，内部ブロックで同一名が使われる可能性があるので，それらの名

表 3 ストレジ・クラス
Table 3 Storage classes.

予約語	特徴
① extern	静的，大域的 (global)
② static	静的，局所的 (local)
③ auto	動的，局所的
④ register	動的，局所的，高速演算

前を一意の内部ラベルに変換する．静的変数はスタック上ではなく，命令コードの一部として領域を割付ける．動的変数には，その変数を定義した関数のリターン・ブロックの先頭からのオフセットが与えられる．つまり，動的変数は関数内で宣言された順にスタックに積まれる．また，④ のレジスタ変数もフェーズ I では ③ の auto 変数と同様に扱う．レジスタ変数については **4.2** 節で詳説する．

型には7種類の基本データ型(表1)とこれらの型をもとに構成される構造体型，ポインタ型，配列型，関数型とがある．これらの型は，ほぼ自由に組合せることができる（関数型の場合には一部制限があるが，これもポインタ型を媒介として組合せることができる)．コンパイラは，各変数の宣言時に型情報を有向グラフ形式に表現し，文法的に正しい組合せであるかどうかを確め，その変数の占める領域の大きさを計算する．たとえば，**図4**（a）の宣言に対し，同図（b）のようなデータ構造を作る．これは型チェックにも用いる．

3.2 文の処理

文のならびは，再帰的下向法 (recursive descent 法)[6]を用いて処理している．ただし，Cでは式のあとにセミコロンが続くと文として扱うが，その場合を含め，式は再帰的下向法でなく，主に順位関数 (precedence function)[6]の方法で構文解析する．その他，goto 文の1パス方式による処理と，拡張した switch 文を取上げて説明する．

3.2.1 式の処理

式の処理は2パスで行う．第1パスでは式に対応する構文木を作り，各演算子が適当な型かどうかを調べ，定数式があればその値を計算する．第2パスではこの構文木を後行順 (postorder)[7]でたどりながら，Cコードを出力する．

Cに特有な演算子の1つに postfix 演算子（＋＋，－－）がある．これは変数を評価したあと，その値が更新される．たとえば，次の return 文（1）は関数値としては1を返すが，変数 a の値は2に更新される．

```
a=1;
return(a++);                    (1)
```

この“余波作用”ともいうべき考え方を拡張して，コンマ演算子と対になるダブルコンマ演算子 (,,) を導入した．つまり，評価は同様に左から行うが，結果の値は左の式の値とする．たとえば，変数 x と変数 y の値を交換する式は，一時的変数を使わずに（2）のように書ける．

```
(1) char A[2][3];
(2) long *F( );
(3) struct T {
      struct T *P;
      char *M[3];
      int N;
    } S;
```

図4（a） 宣言の例

Fig. 4 (a) An example of declarations.

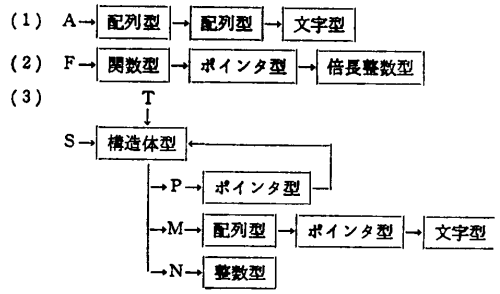

図4（b） 同図（a）に対応するデータ構造

Fig. 4 (b) Data structure diagram corresponding to Fig. 4 (a).

```
x=(y,,y=x);                     (2)
```

3.2.2 goto 文の処理

Cでは各ブロックの先頭で変数を宣言することを許していて，これらのブロックへ飛込む，あるいは，ブロックから飛出す goto 文が使える．したがって，コンパイラはブロックから飛出す goto 文に対して，このブロックまでに取られている変数領域と行先のブロックで取られている変数領域の差だけ，領域を解放しなければならない．また，ブロックへの飛込に際しては，この逆のことを行わなければならないが，この goto 文の処理は次のように1パスで行っている．

図5 のような goto 文とラベルがあるとき，まず

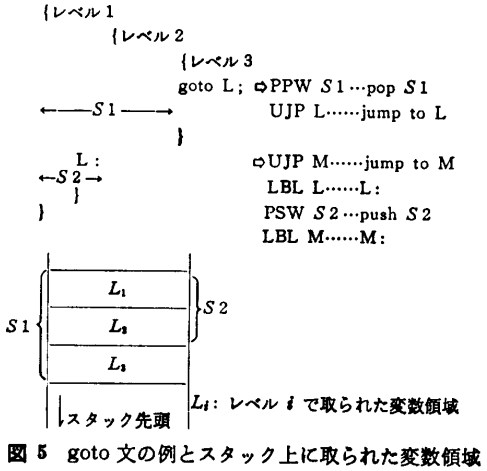

図5 goto 文の例とスタック上に取られた変数領域

Fig. 5 Goto statement and local Stack area.

goto 文が現れたところで，レベル1を基準として，この内部ブロック（レベル3）までに取られた変数領域（大きさ$S1$）を解放し，ラベルLへのジャンプ命令を出す．処理がラベルLのところへくると，ラベルLを立て，レベル1を基準として，ラベルLを含む内部ブロック（レベル2）までに取られた変数領域（大きさ$S2$）を確保する．

3.2.3 switch 文の処理

標準 C の switch 文 (PASCAL の case 文にあたる) の case ラベルとしては，定数式のみが許されているが，この case ラベルの構文を関係演算子（<，<=，>，>=）や記号（..）を使って，範囲を書けるように拡張した．

たとえば，図6のように 'A' から 'Z' までというように指定することにより，式 c の値がある定数と一致するかどうかだけでなく，その値が指定された範囲に入っているかどうかという条件によって文を選択できる．ただし，これらの範囲は重複して指定できない．

文を選択するCコードには次の2種類がある．

i）定数式の値により，選択すべき文の番地を直接求めるCコード (SDJ).

ii）文の番地表を2分探索法によって調べるCコード (SBJ).

コンパイラは，指定された定数の範囲が case ラベルの数に対して比較的狭い場合 i）を，その他の場合には ii）のCコードを自動的に使い分けて出力する．

3.3 実行時のスタック管理のための処理

3.3.1 実引数のチェック

Cでは分割コンパイルを許しているため，実引数と仮引数が一致しているかどうかのチェックは，コンパイル時にはしにくい．また，一致しない使い方も誤りではない．筆者らは，実引数の占める大きさが仮引数の大きさと一致するかどうかを，実行時にチェックするための C コード (CKP) を設定した．先に 2.2 節で述べたように，関数を呼ぶと図2のようなリターン・ブロックをスタック上に作り，その上に実引数を積む．このとき SP (stack pointer) は実引数の最後の番地を指しているから，この値を調べれば実引数の大きさ (SP—MP—(リターン・ブロックの大きさ)) を知ることができる．このため，コンパイラは図7のように，関数 $f2$ の最初の部分で仮引数全体の占める大きさ (2)を C コードとして出力し，実行時に実引数の大きさ(3)と比較できるようにした．

Cでは実引数と仮引数が一致しない使い方をすることがあるので，実行時に実引数の領域が仮引数の領域よりも大きい場合は，スタックをその差だけポップし，小さい場合には，スタックにその差だけ値0をプッシュする．

```
switch (c) {                UJP L1 ……jump to L1
  case 〈'#'  : 文1;
  case  '%'   : 文2;        [文1，2，…，n に対応するCコード]
  case 'A'…'Z' : 文3;       UJP L2 ……jump to L2
                            LBL LS……LS:
        ……                  [番地表]
        ……
                            LBL L1……L1 :
  default    : 文 n;        PSW ……push S
                            SDJ LS
}                             or ……switch jump
                            SBJ LS
                            LBL L2……L2 :
```

図 6 拡張した switch 文の例と対応するCコード
Fig. 6 Extended switch statement and its C-code.

```
f1( )                       …CKP 0 …check parameter
{                             CKS L1 …check SP
  f2(a, b, c); …  [関数 f2 を呼ぶためのCコード]
}                           …LBL L1 …L1 :
                              CSV S1 …S1
f2 (A, B)
  int A, B;                 …CKP 2 …check parameter
{                             CKS L2 …check SP
}                           …LBL L2 …L2 :
                              CSV S2 …S2
```

図 7 スタック管理のためのCコード
Fig. 7 Stack management C-code.

3.3.2 スタックポインタのチェック

関数呼出によって，そのための領域がスタックにプッシュされると，スタック・オーバフローが生じる可能性があるが，これを実行時にチェックするCコード (CKS) が用意されている．これは図7のように，コンパイル時に1つの関数定義の処理がすべて終った時点で，それまでに計算したスタックの最大使用量Sを出力する．実行時には，このSの値をそのときの SP に加えることができるかどうかによって，スタック・オーバフローが生じるかどうかをチェックする．

4. 移植性の確保

ここでは，フェーズIにおける処理の中で問題となる，機械に依存する部分の処理について考察する．ただし，対象となる機械は1語が8ビット以上であることを仮定している．

4.1 データ表現形式の差異

4.1.1 データ型の扱い方

表1で示したように，Cの基本的なデータ型には7種類あり，現実に移植されている機械におけるそれぞ

れのデータ型の大きさ（そのデータ型を表現するために必要な領域の大きさ）を示すと表4のようになる．つまり，各機械において，データ型の大きさはそれぞれ異なっている．したがって，移植を容易にするためには，フェーズIの処理において次の2点を考慮する必要がある．

1) コンパイラ内部で，これらのデータ型の大きさを表現するのに基準となる尺度は何か

2) データ型の表現に関する情報の中で，コンパイラがCコードを出すために最低限必要な情報は何か

これらの情報は，たとえば，次のような処理をするとき必要である．スタック上に割付ける動的変数は名前ではなく，番地（リターン・ブロックからのオフセット）で参照するが，番地を計算するためには，それらの変数がどれだけの領域を占めるかを知る必要がある．また，大きさが1より小さい変数（文字型やフィールド型—1語をビット単位のデータに分割したもの）については，特に番地付が困難でない限り，1語内に複数個のデータを割付ける方が空間効率のよい目的プログラムを出力できる．逆に，1語より大きな変数はワード・アラインメント(word alignment—データ領域を語境界から割付ける）を行う必要がある．

これらの処理内容を検討した結果，コンパイラ内部でデータの大きさを表現するには，バイト（8ビットデータをさす．ただし，表4のHoneywell 6000の例では，1文字を9ビットで表わしている．このような場合，例外的に「バイト」は9ビットであると扱うこととする）を単位として使うのが適当であり，このとき1語あたりのバイト数もワード・アラインメントを行うために必要であることがわかった．また，フィールド型を処理するためには，1語あたりのビット数が必要である．つまり，フェーズIに対する必要最小限のパラメータとして，これらの値と各データ型の大きさ（バイト単位）を表5のような定数名（constant identifier）で表わし，コンパイラをこれらの定数名を用いて書いた．

ただし，これらの定数名を使う際には，次のことを仮定している．

i）整数型の大きさは，その機械の1語長と一致する．

ii）各データ型の間には，(3)式の大小関係が成り立っていること．

$$\text{SHORTSIZE} \leq \text{INTSIZE} = \text{UNSIGNEDSIZE} \leq \text{LONGSIZE},$$

表4 各機械において表現されるデータ型の大きさ
Table 4 Various data sizes on machines.

データ型＼機械	DEC PDP-11	Honeywell 6000	IBM 370	Interdata 8/32	NOVA 3 (本論文)
char	8 (ASCII)	9 (ASCII)	8 (EBCDIC)	8 (ASCII)	8 (ASCII)
short	16	36	16	16	16
int	16	36	32	32	16
unsigned	16	36	32	32	16
long	32	36	32	32	32
float	32	36	32	32	32
double	64	72	64	64	32

表5 パラメータとして与えられる定数名とその意味
Table 5 Machine dependent parameters.

定数名	意味	NOVA 3の場合
BYTEPW	1語あたりのバイト数	2
BITPW	1語あたりのビット数	16
CHARSIZE	文字型の大きさ（単位はバイト）	1
SHORTSIZE	短整数型の大きさ 〃	2
INTSIZE	整数型の大きさ 〃	2
UNSIGNEDSIZE	符号なし整数型の大きさ 〃	2
LONGSIZE	倍長整数型の大きさ 〃	4
FLOATSIZE	浮動小数点数型の大きさ 〃	4
DOUBLESIZE	倍精度浮動小数点数型の大きさ 〃	4
WORDPSIZE	ワードポインタ型の大きさ 〃	2
BYTEPSIZE	バイトポインタ型の大きさ 〃	2

$$\text{CHARSIZE} \leq \text{INTSIZE},$$
$$\text{FLOATSIZE} \leq \text{DOUBLESIZE},$$
$$\text{INTSIZE} \leq \text{WORDPSIZE},$$
$$\text{INTSIZE} \leq \text{BYTEPSIZE}. \quad (3)$$

ポインタ型の大きさを表わす定数名としてWORDPSIZE, BYTEPSIZEを使い分けているが，これは主にフェーズIIで効率のよい目的プログラムを生成するために設けられたものである．

4.1.2 コンパイラにおける定数名の使用例

番地計算のために，ワード・アラインメントを施す式は定数名を使って(4)式のように書いた．ただし，変数offsetはリターン・ブロックの先頭からの相対番地を表わし，単位はバイトである．

$$\text{offset} = ((\text{offset}-1)/\text{BYTEPW}+1)*\text{BYTEPW} \quad (4)$$

フィールド型変数を示す番地はリターン・ブロックの先頭からのバイト・オフセットと語境界からのビット・オフセットを1語に納めて表現するが，これは(5)式のように書いた．ここで ≪ は左シフト演算子を示す．また，変数logbitpwはBITPWの対数をとったものである．

$$\text{word} = (\text{byteoffset} \ll \text{logbitpw}) + \text{bitoffset} \quad (5)$$

4.2 レジスタ変数の扱い

コンパイラはレジスタ変数を実際にレジスタに割付けるのが望ましいが，レジスタという概念は機械に依存するものであり，フェーズIで特定のレジスタを想定してCコードを出力することは困難である．これは次の理由による．

① レジスタ演算のためのCコードを出力するためには，実際に割付けることのできるレジスタについての情報が必要なこと．

② レジスタ演算を許すようなCコードを設定すると，Cコードの機械独立性が損われること．

そのため，フェーズIではレジスタ変数を動的変数と同様に扱い，スタック上に割付ける．ただし，このとき変数の番地とレジスタ番号を対にした情報をCコードとして出力しておく．フェーズ II ではこのCコードからレジスタ定義表を作り，そのレジスタが割付可能ならばレジスタ間の演算命令を出し，割付可能でなければ動的変数と同様に扱う．レジスタに割付けた場合，スタック上の領域は演算のためには使わないが，関数呼出の際にはレジスタからの退避場所として用いる．

5. コンパイラ・システム

ユーザの書いたCのソース・プログラムを実行形式に変換する場合，コンパイラをいくつかの変換プログラムからなるシステムとして扱う方が，次のような点で都合がよい．

① Cでは言語仕様の一部に，プリプロセッサで処理すべきこと（たとえば，define 文（マクロ定義機能）や include 文（テキスト・ファイルの挿入機能）などの処理）を含めている[1]．

② 移植性の点から，筆者らは，機械に独立なフェーズIと，そうでないフェーズ II に分けた．

③ Cの特徴の1つは，分割コンパイルできることであり，リンケージ・エディタで結合する．

④ 機能ごとに分割した方が保守しやすい．

⑤ 各変換プログラムは小さく，作成しやすい．

このようなことから，コンパイラ・システムを図8のような構成にした．ただし，ユーザにとって使いやすいシステムにするため，次のような点を考慮した．

1) ユーザの指定すべきコマンド数が増加しないように，コマンド処理プログラムを作り，これら一連の変換プログラムを1つのコマンドで起動できるようにした．

2) プリプロセッサの出力にソース・プログラムの行番号とファイル名を埋め込むことにより，フェーズIではソース・プログラムの行番号によってエラー・メッセージを出力できる．

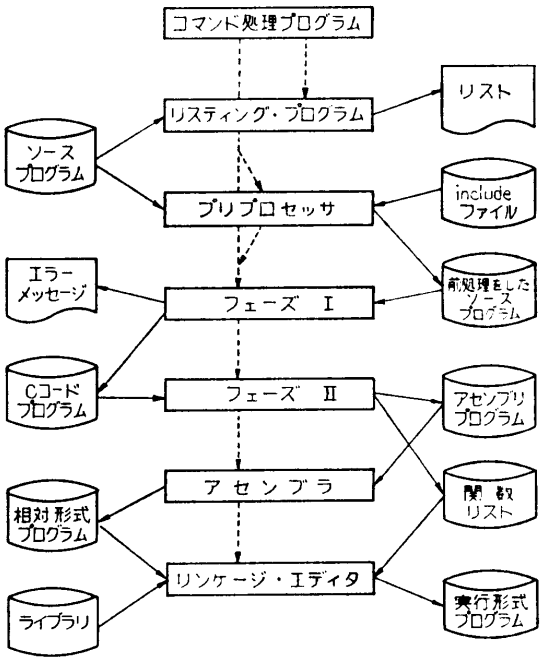

図 8 NOVA 3 におけるコンパイラ・システム
Fig. 8 The compiler system on the NOVA 3.

3) フェーズIIにおいて，ユーザ・プログラムで使われた関数名を登録した関数リストファイルを作り，リンケージ・エディタで，必要なライブラリを自動的に選択，結合するようにした．

6. むすび

このCコンパイラは，移植性を考慮してソース・プログラムを一度中間言語に変換する方式をとっているが，このためにCのシステム記述用言語としての特徴が損われることはない．これは，Cコードが移植性だけでなく，目的プログラムの効率についても十分考慮されているため，フェーズ II においてCコードを効率（空間効率並びに速度効率）のよい目的プログラムに変換できるからである．また，インライン機能（アセンブリ言語の埋込み機能）も備えている．

このコンパイラは以前に作った部分仕様のCコンパイラ（PASCAL で書いた）に比べて，目的プログラムの効率，コンパイル速度，完全仕様(stop 文, switch 文の case ラベル，ダブルコンマ演算子などで拡張されている)の点ですぐれている．作成に要した期間は，

付　録

Cコード表（1）

ニーモニック	タイプ	オペランド	機能
▶スタックのデータを演算，操作するためのCコード			
LCM	*t*		logical complement
BCM	*t*		bitwise complement
NEG	*t*		negative
ADD	*t*		add
SUB	*t*	*v*	subtract
MLT	*t*		multiply
DIV	*t*		divide
MOD	*t*		modulo
LSH	*t*		left shift
RSH	*t*		right shift
BAN	*t*		bitwise and
BOR	*t*		bitwise or
BEO	*t*		bitwise exculusive-or
POI	*t*	*v*	postfix increment
POD	*t*	*v*	postfix decrement
PRI	*t*	*v*	prefix increment
PRD	*t*	*v*	prefix decrement
REQ	*t*		equal
RNE	*t*		not equal
RLT	*t*		less than
RLE	*t*		less than or equal
RGT	*t*		greater than
RGE	*t*		greater than or equal
MST			mark stack
MAD	*t*	*v*	get member address
FAD	*b*	*v*	get field address
PAD	*t*	*v*	get pointer address
LPA	*t*	*v*	load constant & get pointer address
PPW	*b*	*v*	pop *v* words
PSW	*b*	*v*	push *v* words
TCV	t_1	t_2	type conversion (t_2 to t_1)
LDS	*t/d*		load with stack top address, save address
LDU	*t/d*		load with stack top address, unsave address
LOD	*t*	*l/v*	load with *l*(*v*) unsave address
LDR	*t*	*l/v*	LOD for register
LAD	*t*	*l/v*	load address
LAR	*t*	*l/v*	LAD for register
STS	*t/d*		store, save value
STU	*t/d*		store, unsave value
LDC	*b*	*v*	load constant
CKP	*b*	*v*	check parameter size
CKS	*b*	*l*	check max stack pointer
MSR	*b*	*v*	move stack data to register *v*

Cコード表（2）

ニーモニック	タイプ	オペランド	機能
▶プログラムの実行を制御するためのCコード			
RET	*t/b*		return to caller
STP			stop program
END			end program
LAN	*t*	*l*	logical and & jump to *l* if false
LOR	*t*	*l*	logical or & jump to *l* if true
FJP	*t*	*l*	jump to *l* if false
TJP	*t*	*l*	jump to *l* if true
UJP	*b*	*l*	unconditional jump
SDJ	*b*	*l*	switch direct jump
SBJ	*b*	*l*	switch binary jump
CFS	*t*		call function, save value
CFU			call function, unsave value
▶フェーズIIに対する疑似命令			
LBL	*b*	*l*	label:
CSL	*b*	*l*	constant (label)
CSV	*b*	*v*	constant (value, one word)
EXP	*b*	*l*	export label
IMP	*b*	*l*	import label
DFR	*d*	*v*	define register *d* to offset *v*
RRN	*b*	*v*	release register up to *v*

ただし，記号は次の意味を示す．
t : ⟨C⟩ ␣
b : ␣ ␣
d : ⟨*d*⟩⟨*d*⟩
v : ⟨value⟩
l : ⟨label⟩

言語仕様についての検討期間を含めて，フェーズIに4人月，フェーズ II に1人月であった．

現在，このポータブル・コンパイラの移植性を確認するため，いくつかの機械への移植を計画中である．

参考文献

1) Kernighan, B. W. and Ritchie, D. M.: The C Programming Language, Prentice-Hall (1978).
2) 石田晴久：ベル研の軽装 OS-UNIX，情報処理，Vol. 18, No. 9, pp. 942-949 (1977-9).
3) 黒田，辻野，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，電子通信学会情報・システム部門全国大会，420 (1979-10).
4) 疋田輝雄：コンパイラのキットを用いた PASCAL の移植，日経エレクトロニクス，Vol. 12, No. 13, pp. 100-130 (1976-12).
5) 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ルの作成について，電子通信学会技術研究報告（電子計算機），EC 79-81 (1980-2).
6) Gries, D.: Compiler Construction for Digital Computers, Wiley (1971).
牛島和夫訳：コンパイラ作成の技法，日本コンピュータ協会 (1978).
7) Aho, A. V., Hopcroft, J. E. and Ullman, J. D.: The Design and Analysis of Computer Algorithm, Addison-Wesley, Reading, MA. (1974).
野崎昭弘，野下浩平他訳：アルゴリズムの設計と解析 I，サイエンス社 (1978).

(昭和55年3月12日受付)
(昭和55年6月19日採録)