Code Generator Generator

三橋　二彩子、佐治　信之　日本電気㈱
石川　浩之　日本電気技術情報システム開発㈱

1．はじめに

近年マイクロコンピュータのめざましい進歩により、これらの領域でのソフトウェア開発は大規模化している。多種多様なマイクロコンピュータのソフトウェア開発を効率良く行なうためには、高品質で高い実行効率を持ち、機種間で共通の仕様を持った高級言語のコンパイラを迅速に提供する必要がある。このためには、ターゲットのマシンアーキテクチャ毎のコンパイラを効率よく開発しなければならない。

我々は、このような Retargetable なコンパイラ開発を容易にするツール COO（compiler object code generator generator）を開発し、実際のマイコン向けコンパイラ開発に適用したので紹介する。

2．Retargetable コンパイラ

古くからコンパイラの移植性や、Retargetability に関して、いろいろな試みがなされてきた。Ganapathi [1] によれば Retargetable なコード生成の試みとしては、

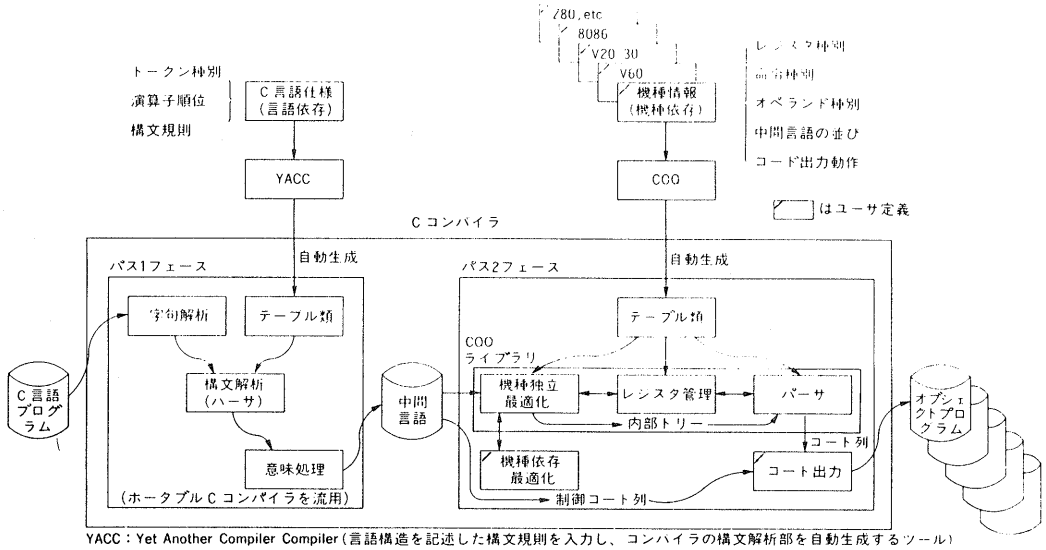
1）インタプリタ方式
2）テンプレートマッチング方式
3）テーブル駆動方式

があげられる。

1）のインタプリタ方式では、コンパイラがまず仮想マシンコードを出力し、これをインタプリタによって実マシンコードに変換する。この方式では、汎用性のある仮想マシンコードをいかに設定するかという点や、機種依存の最適化への対応が難しいなどの問題点がある。

また、2）のテンプレートマッチング方式では、生成される命令列のテンプレートを用意しておき、式の難易度に応じて最適なテンプレートを選んでコード出力する。この方式では、テンプレートの『場合を尽くす』作業において漏れによるコード品質の低下やバグを引き起こしやすいなどの問題点がある。

3）のテーブル駆動の方法は、式トリーのパターンとコード出力の記述からパーサを生成し、これを用いてコード生成する。この方式では 1)、2）の方式に比べて改善の余地がまだ多く残されており、我々は今回、Graham らの方法 [2] をもとに、強力な最適化機能と、実用に耐えるコンパイル性能を実現するための工夫を行なって、コード生成処理の一部を自動生成するツール COO を開発した。



YACC：Yet Another Compiler Compiler（言語構造を記述した構文規則を入力し、コンパイラの構文解析部を自動生成するツール）

図1　COO を用いたＣコンパイラの構成例

3．COO の構成

COO は、Graham らの方法を基礎にしたコンパイラのコード生成部のジェネレータである。対象とする言語はCとし、UNIX 上のポータブルCコンパイラ（PCC）を参考にした。PCC のパス１フェーズをそのまま流用し、パス２フェーズのみ新規に作成した。したがって中間言語も PCC の仕様と同じである。COO の位置付けを図１に示す。

COO は機種依存情報を入力してコード生成部を出力する。この入力記述を CGD（code generator description）と呼ぶ。CGD は以下のセクションで構成されている。

1）オペレータセクション
- オペレータクラス、ユーザ定義オペレータ

2）コードセクション
- オペレータニモニック
- 四則演算用命令、転送命令出力パターン

3）レジスタセクション
- レジスタニモニック
- 作業用レジスタ、ペアレジスタ分類
- スタックフレームベースレジスタ指定

4）オペランドセクション
- メモリ直接、イミディエイト等のトリーパターン
- オペランド文字列パターン

5）ルールセクション
- 式の中間表現のトリーパターン
- コード出力動作

6）ソースセクション
- 機種依存最適化…コード生成前に各機種向きに式トリーの変形を行なうCプログラム
- 手続き呼び出しの前・後処理のコード出力関数
- 転送命令のコード出力関数
- 命令パターン展開用関数

これらの記述を入力すると COO は CGD に変換を施して以下のようなCのソースプログラムとして出力する。

2）　…→　命令出力用関数

3）　…→　レジスタ管理テーブル

4),1) …→　オペランドパーサ

5),1) …→　トリーパーサ + コード出力部

6）　…→　（そのまま）

このプログラムと COO の提供するライブラリ
- 式分解とコード生成ルーチン
- 機種独立最適化ルーチン
- レジスタ管理ルーチン
- 共通ルーチン

をリンクしてコード生成部を生成する。これらのライブラリはすべて機種独立に作られている。

4．コード生成規則

コード生成規則はコード生成処理を形式的に定義するために、コード生成部への入力となる中間表現を記述する。中間表現はトリー構造を成し、トリーを構成する各ノードには LEAF、UNARY、BINARY の種別がなされている。これらは中間言語の仕様として定められており、トリー構造の表現はポーランド記法で表現する。

コード生成は文脈によりつぎの４種類に分類される。

a）計算後の値は不要

b）計算結果をレジスタにおく

c）計算結果の CC（コンディション・コード）が必要

d）計算結果をメモリ（スタック）に退避する

これらの文脈は、たとえば次のような例で明らかである。

```
int a, b, c;
if(a==b)          ・・・ a==bはccのみ必要
   a=b+c;         ・・・ b+cはレジスタにあげ
else                     a=b+c後の値は不要
   func(a);       ・・・ aはメモリにプッシュする
```

COO ではこれらの文脈を、

a）— $eff　　side-effect only

b）— $reg　　general use

c）— $cc　　condition code only

d）— $temp　　memory save

という記号で表わす。

中間木とそれに対応する出力動作は次の形式で記述する。

書き換え後のノード ： 書き換え前のトリー表現
{書き換え時の命令出力動作}

書き換え後のノードとしては、$eff, $reg, $cc, $temp のうちのいずれかがが指定される。

上記の例に対応する記述は次のようになる。

```
$eff  : '='  NAME $reg  {代入のコード出力}
      | $reg            { no-operation }
$cc   : '==' $reg $reg  {等価比較のコード出力}
$temp : NAME            {スタックへのプッシュ}
      | $reg            {   〃          }
$reg  : '+'  $reg $reg  {加算のコード出力}
      | NAME            {レジスタへロード}
```

ここで $reg ルールは計算結果をレジスタにのせるコードを出力するため最も汎用性があるといえる。そこで $eff、$cc、$temp の各ルールは $reg ルールをデフォールトルールとして持つように規則を記述すると、ルールの効率的な共有が可能となる。

それでは実際のコード生成規則の例をあげる。

```
A = B + C;
```

のような式を解析するための規則は次のようになる。

```
$eff  :  '='  mem  $reg
         { MOV($3,$2); }
$reg  :  '+'  $reg  $reg
         { equate_reg($2,$0); ADD($3,$0); }
$reg  :  mem
         { MOV($1,$0); }
```

前述のように各ルールの左辺シンボルのうち、$eff はコード生成の結果を捨ててよい場合を表わし、$reg は結果をレジスタに置かねばならないことを表わす。つまり右辺に現われる $reg はレジスタノードであることが保証される。{と} で囲ってある部分で、実際のコード出力動作を記述する。この部分はC言語で書く。$0 は左辺シンボルが、$1, $2,...は右辺の左から順に各シンボルが対応する。

また COO では各シンボルにデータ型及びレジスタの修飾子を記述することができる。たとえば、

```
$reg  :  '+'.W   $reg(R1)  $reg
```

のように書くと、これはW(ワード)型の加算演算に関して、左辺を強制的に R1 レジスタにあげることを表わす。

同様に、オペランド文字列パターンも定義する。

```
mem  :  NAME                  "$1n"
mem  :  'U*'  $reg            "($2r)"
mem  :  'U*'  '+'  $reg  CON  "$4d($3r)"
```

ルールの後ろにある "パターン" は、オペランド解析時に展開されてオペランド文字列が作られる。また命令全体のフォーマットは、

```
MOV  "mov  $@,$@";
ADD  "add  $@,$@";
```

のように定義しておくことで、たとえば加算なら、

```
ADD ($3, $0) ;
```

という関数呼び出しをすれば、オペレータ、オペランドの各文字列が適当に展開されて命令を出力することができる ("..." 中の $ 文字は文字列の展開を指示する)。つまり、出力のフォーマッティングに関する雑務から、CGD 記述者は解放されるわけである。

5. 実現法

本節では、COO の実現の方法について紹介する。

5.1 パーサ

Graham らの方法によれば式の中間表現 (トリー表現) のパーサは既存のフラットな列を扱うＬＲ系のパーサを流用している。 COO においても最初の版では同様にＬＲ系パーサ (LALR(1)パーサ) を用いたが、この版では次のような問題が起った。

1) 遷移テーブルの生成時に shift/reduce や reduce /reduce conflict が多数生じる。shift/reduce の conflict は shift と解釈し、 reduce/reduce の conflict は最長の右辺ルールを持つものに reduce するという方法 ([2]においても同様) をとっているが、同じ長さの複数のルールがあった場合にはルールの選択に何らかの文脈情報を用いる必要がある。また conflict の不自然な解決でパージングに失敗する場合がでてくるため、バックトラックのメカニズムが必要となる。

2) 同型のトリーを共有させることで共通部分式最適化処理を行なおうとすると、一方のトリーの還元に伴なう書き換えによって他方も書き換ってしまう。その部分トリーがパージング中の lookahead トークンとなる場合をパーサは考慮する必要がある。

3) このパーサでは対象とするトリーをフラットになおす必要がある。つまり、トリー構造を生かした解析ができない (パージングの前段階での式の変形等の

ために、トリー構造がパージングの対象になる）。これらの問題点に対処して第１版を作成したが、テーブルのドライバは元の倍の大きさになり、速度も低下を余儀なくされた。

さらに実際に記述されたコード生成規則を見ると、ほとんどすべてのルールは自明なパージングの可能なものであり、我々は次の方針でパーサの置き換えを行なった。

a）再帰的下降方式によるパージング

b）最長一致規則とバックトラックの導入

c）ルール記述者に理解の容易な照合ルールの採用

（先読みトークン無し）

この方式では、COO はテーブルとドライバを出力するのでなく、プログラムを生成する。我々は実験的にこの方式に基づいて、しかもテーブルとドライバを出力する版も作成した。つまり３つの版を作成した。

ｱ）LALR(1) パーサ

ｲ）再帰的下降方式パーサ

ｳ）ｲ）のテーブル駆動版パーサ

これらの方式で生成されたコード生成部の性能比較は、

プログラムサイズ　ｱ）≒ ｲ）　ｱ),ｲ）＞ ｳ）

パージング時間　ｱ）≒ ｳ）　ｲ）＜ ｱ),ｳ）

となったが、最終的に ｲ）による方式を採用した。

また COO におけるトリーのルールは、第２節で述べたようにオペランドセクションとルールセクションに記述されるが、これら２つのセクションではパージングの方法が若干異なっている。

オペランドセクションでは、オペランドパターン間のバックトラックがおこる場合があるので、パージング途中で『コード生成＋トリー書き換え』が起こると、パーサの動きがルール記述者にとって分かりにくくなり、好ましくない。たとえば、

```
MEM : 'U*' '+' $reg ICON
```

のようなルールにおいては、$reg 部は一般的な式が許されるが、オペランドセクションのパーサは $reg が必ず shift されると仮定して shift してしまい、その後このルール全体が shift されたのち、$reg 部のコード生成を行なう、というメカニズムを採用している。

ルールセクションではオペランドセクションのような問題は起こらないので通常の解析手法で良い。たとえば、

```
$reg : '+' $reg MEM
```

のようなルールにおいては、+、$reg、MEM の shift に対応して各部分トリーのコード生成が順次行なわれる。

5.2 レジスタ管理

レジスタ管理は、CGD のレジスタ用途別クラス定義の情報をテーブル化したレジスタテーブルを参照することによりパージング時に、必要なレジスタの確保・解放等を行う。図２に、この定義の例を示す。

```
R0    "R0";
R1    "R1";
R2    "R2";
R3    "R3";
WREG  :  R0 | R1 | R2 | R3;
PREG  :  R0  R1 | R2  R3;
```

図２　CGD のレジスタ用途別クラス定義の例

この定義で、WREG は、R0、R1、R2 あるいは、R3 のいずれかのレジスタであるというレジスタクラス名である。また、PREG は、R0-R1 あるいは、R2-R3 のいずれかのペアレジスタであるというレジスタクラス名である。

パーサは、式トリーをトップダウンに下降し、末端の部分木を葉に還元しながら上昇する。この方式では下降時に部分木に渡す「相続属性」と、上昇時に葉となった部分木から返される「統合属性」を、レジスタ割付けに利用する。この１回のトリー走査でたとえ最適ではなくとも、ある程度良好なレジスタ割付けができればコンパイル速度、オブジェクト効率とも適切なものが得られることになる（最適なレジスタ割付けにこだわらずコンパイルの速度を重視した）。実際、前述のレジスタ修飾子を使うだけで、十分な品質のコードを得る事が可能である。

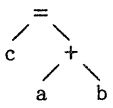
それではレジスタの属性の伝播について例を用いて説明する。

```
c＝a＋b；
```

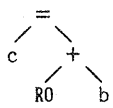
この文をコンパイルするためにはコード生成規則として

```
$eff  :  '='  NAME  $reg
$reg  :  '+'  $reg  NAME
$reg  :  NAME
```

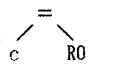
は最低限必要である。レジスタ管理に着目したコード生成動作を以下に示す。なおここでの "_" は、パーサの状態遷移位置を表現している。




```
shift    $eff  :  '='_ NAME  $reg
shift    $eff  :  '='  NAME_ $reg
shift    $reg  :  '+'_ $reg  NAME
shift    $reg  :  NAME_
reduce   任意のレジスタ（たとえばR0）の確保
```



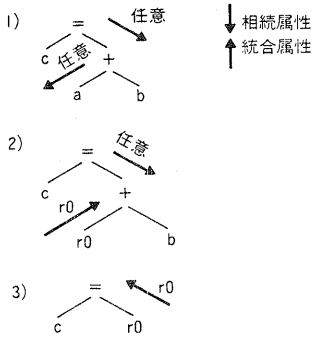

```
....     $reg  :  '+'  $reg_ NAME
shift    $reg  :  '+'  $reg  NAME_
reduce   レジスタ（R0）の統合
```




```
....     $eff  :  '='  NAME  $reg_
reduce   レジスタ（R0）の解放
```

またこの例のレジスタの属性の伝播について次に示す。コード生成規則にレジスタ修飾がなされていないので相続属性は任意のレジスタとなる。コード生成の過程を示す。

このとき、レジスタ本数の少ないマシン等ではコード生成の過程でレジスタがなくなってしまう場合も考えられる。このような場合にレジスタが要求されると、そのレジスタが実際に割り付けられている他のトリーを強制的にメモリのトリーに書き換えて（つまりそのレジスタ



をメモリに退避して）、レジスタを解放するという操作を自動的に行なっている。

次に、文脈に応じて使用するレジスタを限定する方法を与える。たとえばＶＡＸのＣコンパイラでは関数のリターン値は R0 レジスタにあげることにしてあるが、

```
return(式);
```

における『式』が一般に複雑になると、R0 にはあがらずに別のレジスタにあがって、それを R0 に転送する必要が生じる場合がある。

```
例) return(a-b);     mov  a,R0        mov  a,R1
                     sub  b,R0        sub  b,R1
                                      mov  R1,R0
                     最適なコード     まずいコード
```

このようなコード出力を避けるために、レジスタ属性の伝播をさらに改良する。一般的にルールの右辺のパターンのうち、$reg については、『レジスタの属性』を記述することを許し、レジスタ R0 の割り付けを強制的に行なわせることができる。

```
$eff  :  RET  $reg(R0)     { no-operation }
$reg  :  '-'  NAME  NAME   { 減算のコード }
---↓------↓----↓----↓---
   $0      $1    $2    $3
   $0にはトップダウンにR0が割り付けられている
   $2を$0に転送し、
   $3を$0から減算する
```

この例のように、一段の伝播の場合は簡単であるが、実際は様々なルールを通じてレジスタ属性が何段階にも渡ってゆく。このような状況でのレジスタ属性の伝播の方

法について明らかにする。たとえば、

```
$reg  :  '-'  $reg  $reg
```

というルールで上部から伝わってくるレジスタ属性は、どのように下位に伝えられるかを考える。明らかなように、$0と$2は同じレジスタと見なすと都合が良い。

```
$reg  :  '-'  $reg?  $reg
```

$2へ伝えるレジスタ属性は$0から降りてきたものをそのまま使う。コードとしては、次のものを出力すればよい

↓

「$3を$2(or $0)へ加える」

ここで?は、上位から伝わってきたレジスタ属性を下位に伝える表記である。これを一般化すると、次のようになる。

```
$reg : … $reg(Rm)? … $reg(Rn) …
  ↓          ↓          ↓
 $0    …    $m    …    $n
```

というルールに対して、$0から伝えられたレジスタ属性をR0とすると、

a) $mの下位に伝えられるレジスタ属性は、

$$R0 \cap^* Rm$$

b) $nの下位に伝えられるレジスタ属性は、

$$\bar{R0} \cap^* Rn$$

となる。ここで $\cap^*$演算は

$$a \cap^* b \equiv \begin{cases} a \cap b = \phi \text{ ならば } b \\ \text{o.w.} \quad a \cap b \end{cases}$$

とする（$\cap$および$\bar{}$は、通常の集合演算における和集合、補集合を意味する）。もしこの場合のような $0 と $m の間に強い関係が見出せないルールについては、全ての$mに対して

$$R0 \cap^* Rm$$

を伝播させればよい。

$\cap^*$ の演算を実行時に行なうオーバーヘッドは非常に大きい。そのため COO では、$a \cap^* b$ の伝播マトリックスをあらかじめ作っておいて、コード生成時に計算を行なわない方法を採用した。ここで集合 B は、コード生成ルールのレジスタ指定で与えられたレジスタ集合の全体集合を表わす。集合 A は、B によって定まる。

$A = \{ a1, a2, \ldots, am \}$.

$B = \{ b1, b2, \ldots, bn \}$.

$A \supseteq B$, $a \in A$, $b \in B$: $a \cap^* b \in A$, $\bar{a} \cap^* b \in A$.

［マトリックス］

	b1	b2		
a1	c11	c12	……	ai ε A,
a2	c21	c22		bi ε B,
	⋮			cij ε A.

ただし、n本のレジスタに対し最悪の場合 $|A| = 2^n - 1$ となり、マトリクスは非現実的な大きさになってしまう。特に、上記-b) を忠実に行なうと確実に A は発散する。また、ペアレジスタについても考慮するとさらに事態は悪化する。そこで、$\bar{R0}$ は行なわないことにしている。したがって上述したルールの "?" の記法も現在は使用できない（$\bar{R0}$を入れないと A は現実的な大きさでおさえられる）。このレジスタ伝播の効率的な実現手法については、今後に残された課題の１つになっている。

しかしながら、このような簡単な方法で十分効率の良いレジスタ割り付けが可能となる。特に専用レジスタ群で構成されるマシン（v20/30 等）では、このレジスタ伝播の方法は効果を発揮している。

5.3 最適化

最適化は、機種依存と機種独立な部分に分けられる。

機種依存最適化は、利用者がＣプログラムで任意に記述する。例えば VAX のＣコンパイラの場合では、コード生成部の入口で VAX のサポートしていない符号なしデータの除算等のために、部分トリーをランタイムルーチン呼び出しのトリーに書き換えたり、インデックスレジスタとして表現できる部分トリーを『インデックスモード』トリーに変形したりする。これにより利用者が独自に、対象とするアーキテクチャのためのトリーの変形を行うことができる。

機種独立最適化は、COO ライブラリとして COO が提供している機能であり、中間表現である式トリーの最適化を行う。本項では以降この機種独立最適化について述べる。これには主に次のものがある。

a）基本ブロックを越えた共通部分式の重ね合わせ

b）共通部分式削除後の定数の畳み込み

c）無用代入の削除

これ以外にＣ言語での

• comma 演算子と、後置＋＋、－－トリーの分解

• 関数コール、論理演算子、条件演算子トリーの分解

等も行う。

a）の共通部分式は１回のトリー探索でハッシングにより共通な部分式を認識して、１つのトリーで表現されるように変形する。例えば、

```
d＝a＋b＋c；      ……(e1)
e＝a＋b＋d；      ……(e2)
```

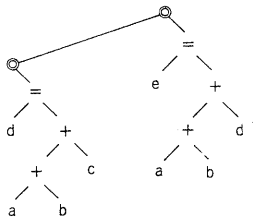
上の式において、次の関係が成立する。

同値関係： e1 式の "a＋b＋c" と "d"

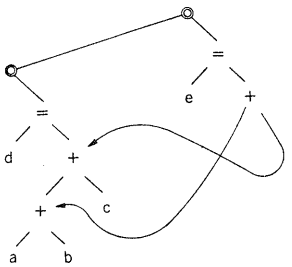
同形関係： e2、, e1 式の "a＋b"

この中間木は上の関係より、共通部分式を重ね合わせて次のように変形される。

最適化前：



最適化後：



この最適化された中間木のコード生成の過程は、次の通りである。

1) d=a+b+c　　aをレジスタｒにあげる

2) d=R0+b+c　　レジスタ R0 にｂを加算

3) d=R0+c　　レジスタ R0 をレジスタ R1 に転送

4) d=R1+c　　レジスタ R1 とｃを加算

5) d=R1　　レジスタ R1 をｄに転送

6) d=R0+R1　　レジスタ R0 とレジスタ R1 を加算

7) e=R1　　レジスタ R1 をｅに転送

また実際のオブジェクトは次のように生成されていく。

```
mov  a,R0    … 1)      mov: 転送命令
add  b,R0    … 2)      add: 加算命令
mov  R0,R1   … 3)      R0,R1: レジスタ名
add  c,R1    … 4)      a,b,c,d,e: 変数名
mov  R1,d    … 5)
add  R0,R1   … 6)
mov  R1,e    … 7)
```

上記の例は、基本ブロック内に対してであるが、基本ブロックを越える大域な範囲に対しても行う。従って、

```
x ＝ p－＞a－＞b；
if (p－＞a－＞c)
        y ＝ p－＞a－＞c；
else
        y ＝ －(p－＞a－＞c)；
z ＝ p－＞a－＞d；
```

のような条件文の文脈などにおいても、共通性の破壊の危険がない限り、連続して共通部分式の重ね合わせを行う。ここでの共通性の破壊の危険とは、

• 関数コール

• ポインタによるメモリへの値の代入

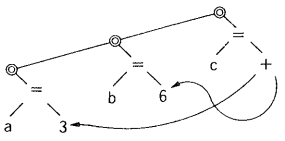
• ａｓｍ文

の部分トリーが対象となる。

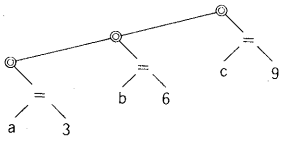
b）は、同値共通部分式の認識により定数項の演算となる場合に、コンパイル時にこの定数演算を畳み込む。

```
a＝1＋2；
b＝2＊3；
c＝a＋b；
```

共通部分式の重ね合わせ後：



定数の畳み込み後：



c）は、auto 変数への代入において代入の伝播を行う事により、return 文あるいは、ブロックエンドが後に続く場合はその代入を削除する。例えば、

```
{
        int  a, c;
        a=b;
        c=a+d;
        return  c;
}
```

というプログラムは、この最適化処理の結果

```
{
        int  a, c;
        return (b+d);
}
```

と記述された事と同じになる。

以上述べた最適化は、基本的なものばかりであるが、現在、より強力な最適化の導入を検討中である。

6．性能評価

現在のところ COO を利用して VAX、V60、68000、v20/30、Z80、PDP 等のＣコンパイラを作成し（あるいは作成中であり）、これらのコンパイラの作成経験から

1）コンパイラ規模

2）コンパイル時間

3）オブジェクトコードの効率

等を比較することで評価を行なっている。

ここでは、VAX 用に作成したコンパイラと VAX/UNIX上で広く使用されているポータブルＣコンパイラ（PCC）との比較を行なう。

まず開発規模（改造規模）を表１に示す。PCC はテンプレート・マッチング方式によるコード生成を行なっているが、この方式に比べて、COO 版では作成に要する改造規模が高々１／４で済むことがわかる。次に生成された VAX 用コンパイラのサイズを表２に示す。サイズに関しては、PCC にわずかに及ばない。次にコンパイラの性能を、コンパイル時間とオブジェクト・コードの性能の点から見る（表３）。この点に関しては、COO 版は PCC を凌駕している。

表１ CGG を利用した VAX 用Ｃコンパイラの規模

モジール	規模（行数）	PCC パス２規模
コード生成記述	557	
機種依存最適化	427	
コード出力	352	
合計	1336	5367

表２ 生成されたＣコンパイラのサイズ
（テキストサイズ： バイト）

COO 版	PCC 版
39936	33792

表３ Ｃコンパイラの性能評価

		CGG／PCC(%)
コンパイル時間		57
コード	サイズ	81 ～ 94
	実行速度	88 ～ 96

総合的に見て、COO 版は PCC に優っていると結論できる。まず記述量が減少した理由としては、

◇ トリーのパターン照合

◇ 複雑なアドレッシングモードの認識

◇ レジスタ割り付け・解放・退避

◇ 条件式の文脈認識

◇ 後置＋＋、－－の分離

等の処理を全て COO が提供するため、COO 使用者はコード生成の本質的な処理のみを記述すれば良いためと思われる。非手続き的な記述が主であるため、記述の複雑度が激減し、修正（改良、保守）も非常にし易くなっている。一方、オブジェクト効率の点で PCC を上まわった理由としては、

◇ コード生成の本質的な処理のみに着目すればよいので、適切なコード出力の選択に専念できること、

◇ 機種依存の最適化のフェーズを設けて、COO 使用者が任意にプログラムできるようになっていること、

◇ 機種独立の最適化については、（まだまだ不十分ではあるが）ある程度の処理（共通部分式の認識等）を提供していること、

などがあげられる。

7．おわりに

新しいコンパイラの開発において、既存のコンパイラの改造をする場合、複雑なコード生成のための処理を理解しなければならない。これに対して、COO を利用した場合には、その中核が非手続き的なルールの列で記述されているために、理解し易く、最適なオブジェクト・コード生成のための本質的な処理のみに着目するだけで済む。さらに生成されるコンパイラの品質も、COO 利用者によるばらつきが少なく、性能向上のための改善も行ない易い。このように、COO を用いることは、コンパイラ開発の生産性向上に大きく寄与する。

一方、現在の COO の課題として、

1）中間言語の表現力向上（ループ表現等）

2）レジスタ割り付けの強化

3）機種独立の最適化の強化

4）自動化率の向上

がある。これらを改善することにより、コードの品質等の改善が期待できる。今後評価を続け、さらに性能を向上させてゆく予定である。

謝辞

本システムの開発の機会を与えていただいたソフトウェア生産技術研究所の藤野喜一所長、寺本雅則部長、そして藤林信也課長に感謝致します。また本開発の技術面での助力、助言をいただいた保坂勇主任、城倉梨香嬢とマイクロコンピュータ技術本部の木村裕氏に感謝致します。

参考文献

[1] Ganapathi,M., Fischer,C.N. and Hennessy,J.L.: Retargetable Compiler Code Generation, ACM Computing Surveys Vol.14, No.4 (1982).

[2] Graham,S.L. and Glanville,R.S.: A New Method for Compiler Code Generation, 5th POPL (1978).

[3] 保坂、佐治、城倉： コンパイラコード生成系の実現、情処学会第29回大会 4D-2 (1984).

[4] 佐治、三橋、木村、保坂：コンパイラコード生成ジェネレータの実現法、情処学会第30回大会 4Q-9 (1985).

[5] 木村、三橋、佐治、保坂：コンパイラコード生成ジェネレータの評価、情処学会第30回大会 4Q-10 (1985).