# コード生成部生成ツール COO†

三橋 二彩子†† 佐治 信之†† 石川 浩之†††
保坂 勇†††† 藤林 信也††

本論文では，コンパイラコード生成部の自動生成ツール COO (COmpiler Object code generator generator) について述べる．古くからコンパイラの移植性を高めるため，また早期実現をはかるために様々な研究がなされている．構文意味解析部に関しては，もはや実用化レベルの自動生成ツールが確立されており，コード生成部に関しても幾つかの自動生成の方式が議論されている．本論文で述べる COO は，コード生成部の自動生成における方式の1つであるテーブル駆動方式を基本として，これを改良し実用化している．COO は，与えられたターゲットマシンの記述からコード生成部の一部となる再帰的手続きを用いてトップダウンに照合を行うパターンマッチャを自動生成する．また COO はコード生成部中の機種独立に共用できる部分をライブラリ化して提供しており，COO を利用することにより短期間での実用的なコード生成部の作成が可能となっている．本論文では，COO への入力となるターゲットマシンの記述方法，COO の実現上の問題点と COO における解決方法，そして実際の適用例と評価について述べている．COO は，V60, i 8086, Z 80, VAX, MC 68020 などの機種用のコンパイラの開発に適用されている．

## 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うためには，高品質で実行効率の良い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，様々なマシンアーキテクチャごとのコンパイラを効率よく開発しなければならない．

コンパイラは，通常，構文意味解析を行うフロントエンドとコード生成を行うバックエンドから構成される．このような構成で，共通の中間言語を定め，機種依存のバックエンドのみを作り直すことにより開発効率を高めることができる．コード生成部を自動生成することは，この開発効率をさらに高め，また保守性を向上する上で重要である．現在，構文意味解析部に関しては UNIX 上の YACC[5] 等実用化レベルの自動生成ツールが確立されている．コード生成部に関しても幾つかの自動生成の方式が議論されそれに基づいたツールが開発され実用化レベルのもの (PQCC[7] 等) も存在しているが，まだまだより良いツールが求められている．ここで述べるコード生成部の自動生成ツール COO (COmpiler Object code generator generator) はコード生成部の自動生成における方式の1つであるテーブル駆動方式を採用し，これを改良して実用化している．COO は，与えられたターゲットマシンの記述からコード生成部の一部となる再帰的手続きを用いてトップダウンに照合を行う（以降，この方式を再帰的下降方式と呼ぶ）パターンマッチャを自動生成する．また COO は機種独立最適化等コード生成部の機種独立に共用できる部分をライブラリ化して提供しており，COO を利用することにより短時間での実用的なコード生成部の作成が可能となっている．

本論文では，まず第2章でこれまでになされたコード生成部の自動生成方式に関する研究について概観し，第3，4章で COO の構成とその入力となる機種依存情報の記述について述べ，第5章で COO の実現方法と実現にあたっての問題点を述べ，最後に COO の適用例と評価について報告する．

## 2. リターゲット容易なコンパイラ

古くからコンパイラの移植性や，リターゲット化に関して，いろいろな試みがなされてきた．Ganapathi[2] によれば，リターゲット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．

† Code Generator Generator: COO by FUSAKO MITSUHASHI, NOBUYUKI SAJI (NEC Corp.), HIROYUKI ISHIKAWA (NEC Scientific Information System Development Corp.), ISAMU HOSAKA (NEC Microcomputer Technology Co., Ltd.) and SHINYA FUJIBAYASHI (NEC Corp.).
†† 日本電気(株)
††† 日本電気技術情報システム開発(株)
†††† 日本電気マイコンテクノロジー(株)

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．この方式の代表例には，Pascal の P-code がある[1),8)]．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．この方式を採用している代表例としてUNIX上のポータブルCコンパイラ（PCC）がある[4),6)]．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式は，コード生成部への入力の中間言語のパターンとそれに対応するコード出力の記述から中間言語に対するパーサを生成し，これを用いてコード生成する．この方式は 1)，2) の方式に比べて新しい方式で基本方針が示された段階である．

我々は今回，1)，2) の方式に比べてレジスタ割り付けや最適化処理が組み込みやすいとの判断から 3) の方式を採用した．特に Graham らの方法[3)] をもとに，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．Graham らの方法での問題点とそれをどのように改良したかを含めて，中間言語のパターンマッチャの実現については，第4章で述べる．

## 3. COO の概要

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．COO を利用して作成するコンパイラの対象とする言語は C とし，構成は PCC を基本にした[9),10),12)]．つまり，COO を利用してコンパイラを作成するときには，フロントエンドは PCC のフロントエンドをそのまま流用し，バックエンドのみを COO を利用して新規に作成することにした．したがってコード生成部への入力である中間言語は PCC の仕様と同じものとする．コンパイラ作成における COO の位置付けを図1に示す．

### 3.1 対象とするマシンアーキテクチャ

COO を利用して生成するコード生成部は，以下のようなアーキテクチャを持つマシンを対象とする．

① 8〜32 ビットのレジスタを持つマシン
専用レジスタ群を持つマシンも対象としている．

② メモリ待避領域としてスタック構造で示すことができる領域を持つマシン
スタックの成長方向は，正負いずれでもよい．

### 3.2 COO への入力—機種依存情報

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おり，1)，2)，3)，4)，5) のそれぞれのセクションの記述をコード生成規則と呼ぶ．

1）オペレータセクション
中間言語オペレータの分類（オペレータクラス）と中間言語の仕様拡張に利用するユーザ定義オペ

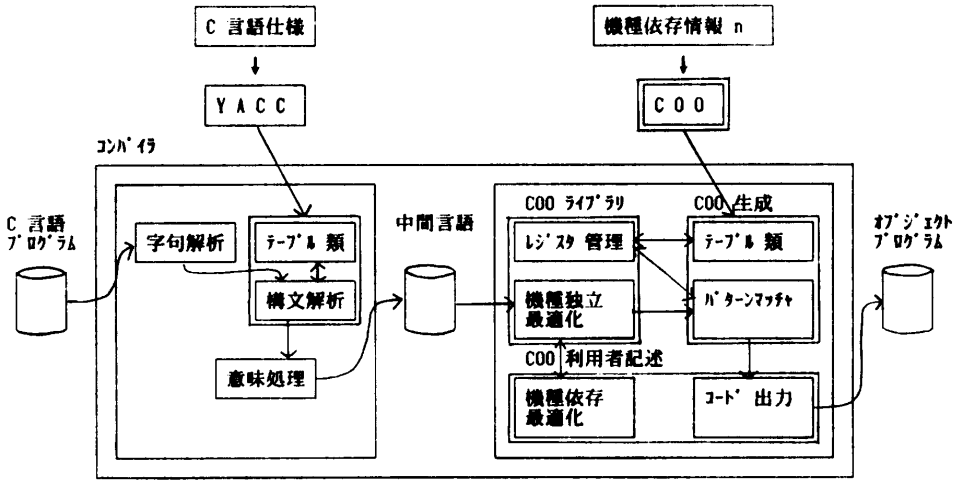


図1 コンパイラ生成における COO の位置付け
Fig. 1 COO's position in compiler construction.

レータを記述.

2）コードセクション

中間言語オペレータの命令コードニモニックと四則演算用命令，転送命令などの命令出力パターンを記述.

3）レジスタセクション

ターゲットマシンのレジスタとコード出力時のレジスタ名と作業用レジスタ，ペアレジスタなどCOO利用者による利用用途別レジスタの分類，スタックフレームベースレジスタなど特殊用途レジスタの指定を記述.

4）オペランドセクション

メモリ間接，イミディエイト等アドレシングモードのトリーパターンと対応するオペランド文字列パターンを記述（詳細は4章).

5）ルールセクション

式に対する中間言語のトリーパターン（式トリーパターン）とそれに対応したコード出力動作を記述(詳細は4章).

6）ソースセクション

機種依存最適化，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などをCで記述.

図2にCGDの記述例を示す.

### 3.3 COOを利用したコード生成部

CGDを入力するとCOOはCGDを変換して以下のようなCのソースプログラムを出力する.

```
%operator
    ADDOP : PLUS|MINUS ;
    INDEX  "UNARY" ;
%code
    PLUS   "add" ;
    ADD    "$*   $@, $@" ;
%register
    R0     "r0" ;
    R1     "r1" ;
        :
    R5     "r5" ;
    WREG : R0 | R1 | R2 ;
    $freg : R5 ;
%operand
    mem : NAME     "$1n"
%rule
    $eff : '=' NAME $reg {...}
         | $reg
         ;
    $reg : ....
    $temp : ...
    $cc   : ...
```

図2 CGDの記述例
Fig. 2 An example of CGD.

2) ····→ 命令出力用関数

3) ····→ レジスタ管理テーブル

4), 1)····→ オペランドパターンマッチャ

5), 1)····→ 式トリーパターンマッチャ
＋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

- コード生成部制御ルーチン
- 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汎用的に作られている.

## 4. コード生成規則

本章では，CGDに記述されるコード生成規則中，コード生成処理を行う上で重要なオペランド，ルールセクションの規則を中心に説明する．これらのセクションにおける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であるトリー構造の中間表現をポーランド記法で記述し，また，対応したコード出力動作を記述する.

### 4.1 ルールセクションにおけるコード生成規則

ルールセクションにおける中間表現とそれに対応するコード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というCOOで定められた記号のうちのいずれかが指定される．これらの記号は次に示すようなコード生成における4種類の文脈を示す.

a）$eff …計算後の値は不要な場合，例えば図3における“a＋b＋c”のような代入文.

b）$reg…計算結果をレジスタにおく必要がある場合，例えば図3における“a＝b＋c”の“b＋c”の計算結果.

```
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
```

図3 コード生成における文脈
Fig. 3 The context of code generation.

```
$eff   : '='   NAME $reg
               {MOV($3, $2); /*代入のコード出力*/}
       | $reg          /* no-operation */
$cc:   : '=='  $reg $reg
               {rel_ope($3, $2, $1); /*等価比較のコード出力*/}
$temp: NAME
               {PUSH($1); /*スタックへのプッシュ*/}
       | $reg
               {PUSH($1); /*スタックへのプッシュ*/}
$reg   : '+'   $reg $reg
               {cg_rforce($2, $0); ADD($3, $0); /*加算のコード出力*/}
       | NAME
               {MOV($1, $0); /*レジスタへロード*/}
```

図 4 ルールセクションにおけるコード生成規則の例
Fig. 4 An example of production rules in rule section.

c）$cc …計算結果の CC（コンディション・コード）が必要な場合，例えば図3における“a==b”.

d）$temp…計算結果をメモリ（スタック）に退避する必要がある場合，例えば図3における“func(a)”の“a”.

図3の例に対応するルールセクション記述の一例を図4に示す．図において，{と} で囲まれたコード出力動作の記述では，“$n” で左辺と右辺のシンボルを参照できる．$0 は左辺シンボルを，$1, $2, … は右辺の左から順に各シンボル（各要素）を参照する．

また，右辺の各シンボルにデータ型およびレジスタの修飾子を記述することができる．このことにより，式のデータ型や演算の種類に合わせたルール記述が簡単に行える．例えば，

$eff : FORCE. W $reg(R1)

のように書くと，これはW（ワード）型のリターン命令に関して，リターン値を強制的に R 1 レジスタにあげることを表す．

### 4.2 オペランドセクションにおけるコード生成規則

オペランドセクションにおける記述はコード出力動作の記述がオペランド文字列パターンの記述に変更されることを除いてルールセクションにおける記述と同様である．オペランドセクションの記述例を図5に示す．図5において規則の後ろにある“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（“…” 中の $ 文字は文字列の展開を指示する）．

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

図 5 オペランドセクションにおけるコード生成規則の例
Fig. 5 An example of production rules in operand section.

## 5. 実 現 法

本章では，COO の生成するコード生成部の実現方法について述べる．

### 5.1 パ ー サ

Graham らの方法[3]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 においても第0版では同様に LR 系パーサ（LALR (1) パーサ）を生成したが，この版では次のような問題が起こった．

1）遷移テーブルの生成時に競合 (conflict) が多数生じる．競合の解消は文献 3) と同様の方法をとっているが，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，競合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おう（後述）とすると，一方のトリーの還元に伴う書き換えによって他方も書き換わってしまう．その部分トリーがパージング中の先読みトークン (lookahead token) 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直す必要がある．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）．

これらの問題点に次のように対処して第1版を作成した．

1）失敗時のバックトラック機構の追加

2）先読みトークンの更新

3）トリートラバースルーチンの付加

しかしながら，このような対処により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．

さらに第1版への入力として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った．つまり，中間言語式トリーのパターンは LR 系パーサを必要とするほど複雑ではなかった．そこで，我々は処理速度を向

上すべく，次の方式でパーサの置き換えを行った．

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）

我々はこの方式に基づいて，プログラムを生成する版とテーブルとドライバを生成する版を試作した．つまり，次の3つの版を試作した．

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

COO により生成されたこれらの方式の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，処理速度を重視してイ）による方式を採用した．

### 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．

パーサは，式トリーをトップダウンに下降し，末端の部分トリーを葉に還元しながら上昇する．そこで，下降時に部分トリーに渡す「相続属性」と，上昇時に葉となった部分トリーから返される「統合属性」を，レジスタ割付けに利用する．1回のトリー走査で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により決定されるレジスタ属性を利用して，実用的な性能コードを得ることが可能である．

以下にレジスタの管理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**図6**（a）に示す規則は最低限必要である．レジスタ属性の伝播とレジスタ管理に着目したコード生成動作を図6（b），（c），（d）に示す．図6（a）に示されているコード生成規則にはレジスタ修飾がなされていないのでレジスタの相続属性は任意のレジスタとなる．なお図における“-”は，パーサの状態遷移位置を表現している．このようなレジスタ管理において，レジスタがなくなってしまう場合も考えられる．その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

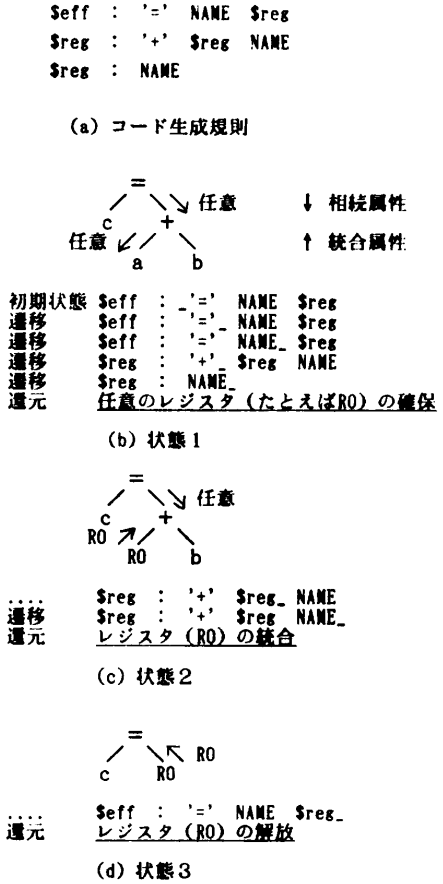


**図6** コード生成におけるレジスタ管理
Fig. 6 Register management at code generation.

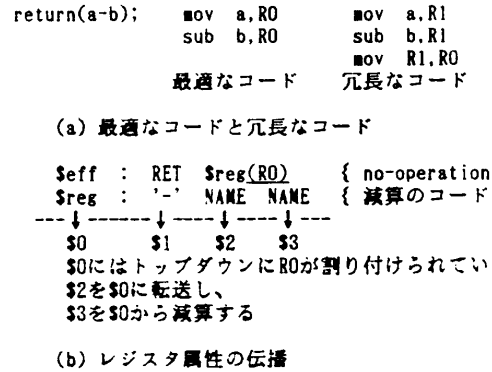


**図7** レジスタ属性によるコードの改善
Fig. 7 Code optimization by using register attribute.

行っている．

次に，レジスタ属性の伝播により文脈に応じて使用するレジスタを限定する方法を示す．例えば VAX UNIX 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（図7(a)）．このようなコード生成を避けるために，レジスタ属性の伝播を利用する．一般的にルールの右辺のシンボルのうち，$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わせることができる（図7(b)）．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．

以降，レジスタ属性の伝播の実現方針と実現方法について述べる．例えば，

$reg : '-' $reg $reg

というルールで上位（上段）から伝わってくるレジスタ属性は，どのように下位（下段）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．$3 を $2 (or $0) へ加えるコードを出力すれば効率の良いコード生成となる．そこで $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．上位から伝わってきたレジスタ属性をそのまま下位に伝える表記として“？”を導入する．

$reg : '-' $reg? $reg

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∩＊Rm

b）$n の下位に伝えられるレジスタ属性は，

$\overline{R0}$∩＊Rn

となる．ここで ∩＊ 演算は

$a \cap * b \equiv a \cap b = \phi$ ならば $b$
そうでなければ $a \cap b$

とする（∩および ¯ 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いだせないルールについては，すべての $m に対して

R0∩＊ Rm

を伝播させればよい．

∩＊ の演算を実行時に行うオーバ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

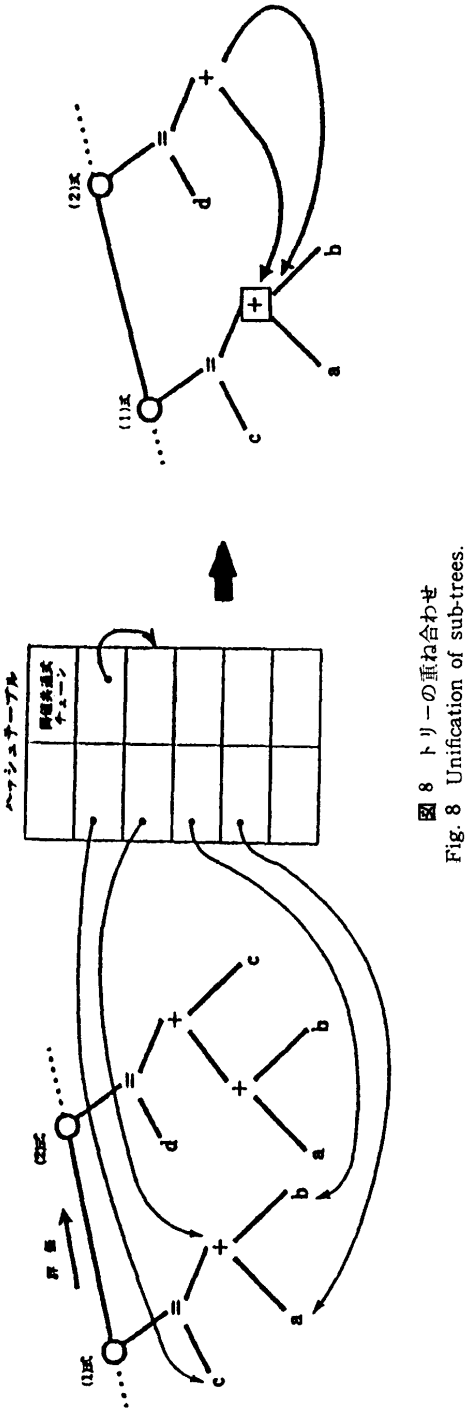


図8 トリーの重ね合わせ
Fig. 8 Unification of sub-trees.

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わない方法を採用した．

上位から伝播してきたレジスタ $ai$ に対し要求されるレジスタが $bj$ であるとき，最終的に伝播されるレジスタを上記の方法で計算し，これを $cij$ とすると伝播マトリックス $P$ の各要素は $P[ai,bj]=cij$ となる．

ただし，このような方法では $n$ 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）を忠実に行うと確実に $\bar{A}$ は発散する．R0 を入れないと $A$ は現実的な大きさでおさえられるので $\overline{R0}$ は行わないことにしている．$\overline{R0}$ を行わなくとも実用的なレジスタ割り付けが可能であり，特に専用レジスタ群で構成されるマシン（V 20/30 等）では，このレジスタ伝播の方法は効果を発揮している．

より効果的なレジスタ割り付けのための $\overline{R0}$ の効率的な実現方法については，今後に残された課題の1つである．

### 5.3 最適化

COO における最適化は，機種依存と機種独立な部分に分けられる．これらの最適化は中間言語レベルでの最適化であり，コード生成部に入力された中間言語に機種依存，機種独立の順序で最適化がなされる．

機種依存最適化は，COO 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，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する．これにより COO 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，中間言語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）共通部分式重ね合わせ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．

ここでは，前述のパーサやレジスタ管理との親和性に優れた最適化を実現している a）について説明する．

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つのトリーを共有する表現に変形する．

例えば，

$c=a+b;$ ……(1)

$d=a+b+c;$ ……(2)

において，次の関係が成立する．

同値関係：(1)式の "$a+b$" と "$c$"

同形関係：(1)，(2)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変形される（図8）．変形されたトリーはパーサに渡され，コード生成が行われる（図9）．パーサはコード生成においてコード出力が終わった部分トリーを演算結果が格納されたレジスタを示すノードに置き換えたり，結果が不要な場合は削除したりする．ただし，共通部分式となっているトリーは削除されず，以降のコード出力においてレジスタとして参照される．

現在のところ基本ブロックを越えた共通式の認識としては，if-then-else 文のみをサポートしている．つ

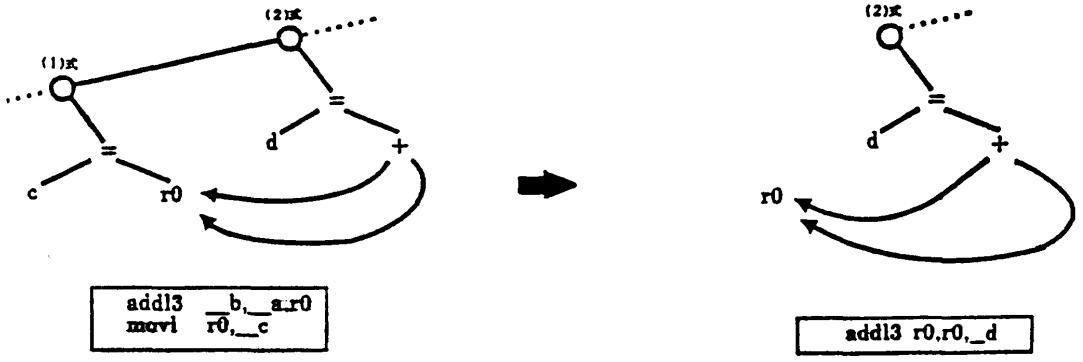


図 9 重ね合わされたトリーのコード生成
Fig. 9 Code generation of a unified tree.

まり，条件文の文脈においては，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．

以上述べた最適化は基本的なものであるが，COO が生成するパーサとの親和性に優れ，COO を適用したコンパイラの生成するコード効率を高めている[13]．

## 6. 性能評価

COO の性能評価は，COO を利用したコンパイラの

1）バックエンド作成のためのステップ数
2）コンパイラ規模
3）コンパイル時間
4）オブジェクトコードの効率

等を比較することで行った．

### 6.1 COO を利用したコンパイラ

現在のところ COO を利用して VAX，V 60，MC 68020，i 8086，Z 80 等のCコンパイラを作成し，あるいは作成中である．**表1**にこれらのコンパイラの開発規模，および性能を示す．

COO を利用したコンパイラのバックエンド作成のためのステップ数はターゲットマシンのアーキテクチャの複雑さに依存している．ここで言うアーキテクチャの複雑さとは以下の要因に依存する．

1）特殊用途のレジスタの存在
2）命令セットの非直交性
3）命令セットがCのオペレータと非対応

**表1** COO を利用したコンパイラ
Table 1 Development of C compiler backend using COO.

| ターゲット CPU | Z80 | i 8086 | V60 | 68020 | VAX |
|---|---|---|---|---|---|
| COO 規則数 | 127 | 93 | 104 | 168 | 91 |
| バックエンドステップ数 KL | 3.6 | 2.5 | 2.3 | 2.5 | 1.3 |
| バックエンドサイズ KB | 83 | 62 | 57 | 62 | 52 |
| コンパイルスピード L/S | 40 | 40 | 40 | 45 | 50 |
| オブジェクトサイズ KB/KL | — | 6 | 8 | — | 6 |

1) COO 規則数とは CGD 中の規則数のことである．
2) バックエンドステップ数とは CGD と COO 利用者記述のCプログラムを合わせたステップ数のことである．
3) KL：キロ行
4) KB：キロバイト
5) L/S：行/秒
6) MC 68020 と Z80 は，浮動小数点データをサポートしていない．
7) 表中の — は，未測定．

**表2** COO を利用した VAX 用Cコンパイラの規模
Table 2 A comparison of backend size of vax C compilers.

| モジュール | COO 版 | PCC |
|---|---|---|
| バックエンドステップ数 L | 557+427+352 1136 | 5367 |
| コンパイラサイズ KB | 234 KB | 191 KB |

1) バックエンドステップ数の COO 版の上段は次のようなモジュールのステップ数を示している．
コード生成＋機種依存最適化＋コード出力
2) KB：キロバイト

### 6.2 ポータブル C コンパイラ (PCC) との比較

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，COO を利用して作成するコンパイラの構成の基本としている PCC との比較を行う．

まず開発規模（改造規模）とコンパイラのサイズ比較を**表2**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たかだか 1/4 で済むことがわかる．COO 版のコンパイラのサイズは，PCC の約 1.2 倍となっている．次にコンパイル時間とオブジェクト・コードの性能比較を**表3**に示す．コンパイル時間に関しては，COO 版は PCC にわずかに及ばないが，オブジェクト・コードのサイズと実行スピードに関しては，ほぼ同等以上の性能をあげている．

総合的に見て，COO 版は PCC とほぼ同等の性能をあげていると結論できる．以下それぞれの比較項目に関して理由を考察する．

まず記述量は 1/4 となっているが，コード生成部作成にあたって COO 利用者が実際に軽減される記述は，

- 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

等の処理である．これらはすべて COO が提供するため，COO 利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．そしてこれらは非手続き的な記述が主であるため，記述の複雑度が減り，修正（改良，保

表 3 VAX Cコンパイラの性能比較
Table 3 An estimation of VAX C compilers.

(a) コンパイル性能

| プログラム (steps) | COO 版 | PCC |
|---|---|---|
| cb.c (374) | 10.1<br>3576+256<br>1014 | 10.3<br>3920+256<br>1136 |
| trees.c (1357) | 31.3<br>9136+364<br>3275 | 27.0<br>9172+364<br>3379 |
| ed.c (1652) | 32.4<br>10924+224<br>3639 | 28.8<br>11156+224<br>3795 |
| cggvax.c (3158) | 79.4<br>21228+8820<br>8522 | 71.7<br>21120+8820<br>9279 |

上段：コンパイル時間（秒）
中段：オブジェクトサイズ（テキスト+データ）（バイト）
下段：アセンブラ行数

(b) オブジェクト性能
(Dhrystone ベンチマーク[18] 結果)

| COO 版 | PCC 版 |
|---|---|
| 1019 | 984 |

(Dhrystones/second)

1) マシン：マイクロ-VAX
2) OS：UNIX 4.3 BSD
3) Dhrystone 値は，値が大きいほど，性能が良い．

守）も非常に行いやすくなっている．次に，コンパイラのサイズ，コンパイル時間においてわずかに PCC に及ばない理由としては，

- コード生成部中に機種独立最適化処理を含んでいること，
- 多種多様なプロセッサに対応するためにレジスタ管理などの処理を汎用化していること，

などがあげられる．そしてオブジェクト効率の点で PCC がわずかながらに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，
- 機種独立最適化については，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．

COO 版のコンパイラの PCC に及ばない部分を改良するためには，COO が提供している処理の汎用化の度合いを検討し直すことがあげられる．例えば，COO 利用者が必要とする処理を指定し，指定された処理のみを取り込んだコード生成部を作成することを可能とする，等である．機種独立な最適化処理に関しては，COO 版のコンパイラの出力するオブジェクト性能を向上する要因となっており，処理速度を上げる工夫とともにより一層の強化が必要である．

## 7. おわりに

高品質，高性能なコンパイラのリターゲット化を迅速に行うための支援ツール COO を実現した．

新しいコンパイラの開発において，既存のコンパイラを改造する場合，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，理解しやすく，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いやすい．COO により，コンパイラ開発の生産性が大きく向上する．

実際に COO を使用してコンパイラを作成し，従来方式と同等以上の品質のものを作成できることを示した．

## 8. 今後の課題

COO の今後の課題として，次のことがあげられる．

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

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．

## 参考文献

1) Amman, U.: On Code Generation in a Pascal Compiler, *Softw. Pract. Exper.*, Vol. 7, No. 3, pp. 391-423 (June/July 1977).
2) Ganapathi, M., Fischer, C. N. and Hennessy, J. L.: Retargetable Compiler Code Generation, *ACM Comput. Surv.*, Vol. 14, No. 4, pp. 573-592 (1982).
3) Graham, S. L. and Glanville, R. S.: A New Method for Compiler Code Generation, *5 th POPL*, pp. 231-240 (1978).

4) Johnson, S. C.: A Tour through the Portable C Compiler, AT & T Bell Labs. (1979).
5) Johnson, S. C.: YACC-Yet Another Compiler Compiler, UNIX Programmer's manual (1979).
6) Kristol, D. K.: Four Generations of Portable C Compiler, *USENIX Summer Conference*, pp. 335-343 (1986).
7) Catell, R. G. G.: Formalization and Automatic Derivation of Code Generators, *Computer Science System Programming*, UMI Research Press, Michigan (1982).
8) Penverton, S. and Daniels, M.: *Pascal Implementation*, Ellis Horwood Limited (1982); 武市正人, 木村友則共訳: Pascal の言語処理系, 近代科学社, 東京 (1984).
9) 保坂, 佐治, 城倉: コンパイラコード生成系の実現, 第 29 回情報処理学会全国大会論文集, 4D-2 (1984).
10) 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, 第 30 回情報処理学会全国大会論文集, 4Q-9 (1985).
11) 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, 第 30 回情報処理学会全国大会論文集, 4Q-10 (1985).
12) 三橋, 佐治, 石川: Code Generator Generator, 情報処理学会第 3 回プログラミング言語研究会, 85-PL-03 (1985).
13) 石川, 三橋: 汎用クロスコンパイラでの最適化実現法, 第 33 回情報処理学会全国大会論文集, 5E-7 (1986).
14) 佐治: COO: A Code Generator Generator, NEC 技報, Vol. 39, No. 5, p. 22 (1986).
15) 松井ほか: V60 開発用ソフトウェア, NEC 技報 Vol. 39, No. 10, pp. 30-32 (1986).
16) 保坂ほか: マイコン用マルチターゲットコンパイラ, NEC 技報, Vol. 40, No. 1, pp. 46-49 (1987).
17) 佐治ほか: COO: Compiler Object Generator Generator, *NEC R & D*, No. 85, pp. 55-59 (1987).
18) ベンチマーク: 32 ビット CPU ボード, 日経バイト, No. 35, pp. 137-145 (July/1987).

(昭和 62 年 2 月 4 日受付)
(昭和 62 年 11 月 11 日採録)

**三橋二彩子**

昭和 36 年生. 昭和 58 年, 津田塾大学学芸学部数学科卒業. 同年, 日本電気(株)入社. 以来, マイクロコンピュータ用ソフトウェア, 主に言語処理系の開発に従事. 現在, C & C 共通ソフトウェア開発本部環境システム開発部勤務.

**佐治 信之** (正会員)

昭和 32 年生. 昭和 57 年, 東京工業大学大学院理工学専攻科修了. 同年, 日本電気(株)入社. 以来, 言語処理系の開発に従事. 現在, C & C 共通ソフトウェア開発本部環境システム開発部勤務.

**石川 浩之** (正会員)

昭和 35 年生. 昭和 58 年東洋大学工学部情報工学科卒業. 同年日本電気技術情報システム開発(株)入社. 以来, C コンパイラの機種独立なレジスタ管理や最適化などの開発に従事. 現在, 汎用クロス C コンパイラの開発に携わっている.

**保坂 勇** (正会員)

昭和 26 年生. 昭和 50 年千葉大学工学部電子工学科卒業. 同年日本電気(株)入社. 以来, 言語処理プログラム, ソフトウェアツールの開発に従事. 現在, 日本電気マイコンテクノロジー(株)に勤務.

**藤林 信也** (正会員)

昭和 17 年生. 昭和 43 年, 京都大学大学院工学研究科修士課程修了. 同年, 日本電気(株)入社. 以来, プログラミングシステム, ソフトウェア工学に関する研究開発に従事. 現在, C & C 共通ソフトウェア開発本部環境システム開発部長. ACM 会員.