多機種 μP 用セルフ・コンパイラ・コード生成部のコンパクト化手法†

杉 山 和 弘††　大 森 圭 祐††　葛 山 善 基††

多機種化が著しく，ライフサイクルの短いマイクロプロセッサ搭載ワークステーションのソフトウェア開発を支援するためには，ワークステーション上で走行する高級言語コンパイラの迅速な提供が必須となる．ジェネレータ方式を用いて高級言語セルフ・コンパイラの開発効率を向上するには，マイクロプロセッサのアーキテクチャ依存性が高く，ジェネレータ化による規模増大が著しいコード生成部を，いかにしてコンパクト化（コード生成時に必要なワークステーションの主記憶容量を小さくすること）するかが重要な課題となる．本論文では，強く型付けされた (strongly-typed) 言語仕様に対して適用可能なコード生成部のフェーズ分割アルゴリズムを中心とした，コード生成部のコンパクト化手法を提案する．コード生成部のコンパクト化手法を適用したマイクロプロセッサ用コード生成部を，ジェネレータ方式で作成し，特定マイクロプロセッサ用に個別作成された市販の高性能セルフ・コンパイラのコード生成部と比較したところ，次の評価結果を得た．①コード生成時に必要な最大主記憶容量は，64～120 k バイトの範囲に抑えることができる．②コード生成処理時間については，1 k ステップのソース・プログラムに対して，1 コード生成サブフェーズ当たり約 30 秒のオーバヘッドを生ずる．③オブジェクト・プログラムのメモリ占有量・実行速度については，まったくオーバヘッドを生じない．

1. まえがき

LSI 技術の急速な進歩に伴って，ライフサイクルの短いマイクロプロセッサ (μP) 搭載ワークステーション（以下では WS と略記する）の多機種化の傾向が著しい．

多機種 WS のソフトウェア開発を効率的に支援するためには，μP 用セルフ・コンパイラ*の迅速な提供が必須となる．これを実現するには，μP 用セルフ・コンパイラのジェネレータ方式による作成が，一つの有効な手段となる．ジェネレータ方式はコンパイラ作成の生産性向上を目的としているが，コンパイラの規模は μP 機種共通化により増大する．一方，μP 用セルフ・コンパイラは，WS の 64～128 k バイト程度の小さな主記憶上でも走行可能であることを要求されることが多い．このため，セルフ・コンパイラをジェネレータ方式で作成するには，生成されるセルフ・コンパイラをいかにしてコンパクト化するかが重要な課題となる．

コンパイラの語い・構文・意味解析部および最適化処理部は，μP のアーキテクチャ依存性がきわめて低いため，機種共通化によるコンパイラ本体の規模の増加はほとんどない．したがって，コンパイラ・ジェネレータ技術[1)~4)]を活用しても，これらの各コンパイル処理部の最大使用主記憶容量は，せいぜい 40 k バイト程度に抑えることができる．

一方，コンパイラのコード生成部は，レジスタ構成・機械語命令の機能・アドレッシング方式等の用途ごとに特殊化されたアーキテクチャを考慮して，高性能なオブジェクト・プログラムを生成する必要がある．このため，コード生成部の最大使用主記憶容量は，μP ごとに個別作成しても，最低 60 k バイトは必須となる．

従来のクロス・コンパイラ作成用コード生成部ジェネレータ技術[5),6)]や，コンパイラ・ソース・プログラム移植方式[7)]を用いて，コード生成部を機種共通化すれば，コード生成部の最大使用主記憶容量はさらに 4～5 倍増加する（**表 1**）．したがって，これらの技術をそのまま活用することはできない．

本論文では，最初に，次に示すコード生成部のコンパクト化手法を提案する．

(1) PL/I, ADA, FORTRAN 等の各種の高級言語仕様を適用対象として，規模の大きいスケルトン・テーブル* を複数のコード生成サブフェーズへ分割配置可能とする，分割コード生成アルゴリズム

---

† Compact Code-Generator for a Multi-target μP Compiler by KAZUHIRO SUGIYAMA, KEISUKE OHMORI and YOSHIKI KATSUYAMA (NTT Yokosuka Electrical Communication Laboratories).

†† 日本電信電話(株)横須賀電気通信研究所

* コンパイル処理と，生成したオブジェクト・プログラムの走行が，同一計算機上で実行されるコンパイラ．これらの処理が，異なる計算機上で実行される場合，クロス・コンパイラと呼ぶ．

* オブジェクト・プログラム生成規則を，意味解析部が生成する中間語命令ごと・中間語命令のオペランドの格納状態ごとに規定したテーブル．

表 1 セルフ・コンパイラ・コード生成部の規模
Table 1 Code generator size of self-compiler.

コンパイラ作成方式＼比較	コード生成部の規模		作成例（ジェネレータ名）	走行機種
	コード生成共通機能	スケルトン・テーブル		
手作業個別作成方式	30～40kバイト	20～30kバイト	FORTRAN	各種 WS
コンパイラ・ソースプログラム移植方式	70kバイト以上	100kバイト以上	C	ミニコン，大型機
〔本論文の対象〕セルフ・コンパイラ・ジェネレータ方式	〔目標値〕最小50kバイト／サブフェーズ	〔目標値〕最小10kバイト／サブフェーズ	MAPS-C (MAPS-CGEN)	各種 WS
(参考) クロス・コンパイラ・ジェネレータ方式	100kバイト程度	130kバイト程度	PMP-CE (PMP-CGEN)	大型機

（2） ビット・パタン形式に展開したデシジョン・テーブル形式のスケルトン・テーブルを用いた，コード生成処理手法

（3） データ接近機能・レジスタ管理機能のコード生成サブフェーズ化手法

次に，これらのコンパクト化手法を適用して，ジェネレータ方式で作成したセルフ・コンパイラ用コード生成部の性能評価結果について述べる．

2. 分割コード生成アルゴリズム

コード生成部は，コード生成共通機能と，スケルトン・テーブルとから構成される．スケルトン・テーブルは，もともと大規模であると同時に，μP のアーキテクチャによって，規模が大きく変動する．

本章では，コード生成部をコンパクト化するための1手法として，2項演算形式に展開された中間語命令を仮定し，その演算順序に着目して，スケルトン・テーブルを複数のコード生成サブフェーズへ分割配置可能とする，分割コード生成アルゴリズムを提案する．

2.1 前提条件

セルフ・コンパイラが実用的であるためには，①開発したソフトウェアの保守性が優れていること，②高性能なオブジェクト・プログラムが生成できること，が重要な要求条件となる．このため，セルフ・コンパイラの走行環境，ソース・プログラムの言語仕様，およびコンパイル処理方式に関して，次の前提条件を設定した．

〔前提1〕 セルフ・コンパイラの走行環境

セルフ・コンパイラは，8/16/32 ビット系の μP を搭載し，最低 64k バイトのセルフ・コンパイラ走行用主記憶と2次記憶装置をもつ，パーソナル・コンピュータ，インテリジェント・ターミナル等の任意の WS の任意のオペレーティング・システム配下で走行可能とする．

〔前提2〕 ソース・プログラムの言語仕様

ソース・プログラムの言語仕様としては，PL/I, FORTRAN, PASCAL, ADA, ALGOL, COBOL 等のように強く型付けされた (strongly-typed) 言語仕様*を対象とし，弱い型付けしかされていない PL/M 等や，まったく型付けされていない C 等の言語仕様は適用対象外とする（**表 2**）．

〔前提3〕 セルフ・コンパイラのフェーズ構成

セルフ・コンパイラは，次の理由により，語い解析部，構文解析部，意味解析部，最適化処理部，コード生成部，およびアセンブリ処理部からなる本格的なフェーズ構成とする（**図 1**）．

理由1：ソース・プログラムの構文上・意味上の厳密な正当性検査の実施

理由2：高性能なオブジェクト・プログラムの生成

〔前提4〕 コンパイル単位

コンパイル単位は，手続き単位とする．手続きの中に，複数の内部手続きを含んでいても構わない．

〔前提5〕 中間語命令の機能

ソース・プログラムは，右辺に多種多様な演算式を含む代入文，手続きコール文や IF 文等の制御文，および CASE 文や PROCEDURE 文等のプログラム構造文から構成される．

意味解析部は，ソース・プログラム中の文を，次のように，中間語命令列に展開する（**表 3**）．

① タイプ I～IV（代入文の右辺の演算式）

代入文の右辺の演算式は，2項演算形式の複数個の中間語命令列（算術演算／ポインタ演算／論理演算／

* 演算機能・種類ごとにオペランドとして指定可能なデータの型属性が厳密に規定された言語仕様．

表 2　演算機能とオペランドの型属性の関係
Table 2　Relationship between function and operand data-type.

<table>
<tr><th>演算種別＼型属性</th><th>強く型付けされた言語仕様のオペランド</th><th>弱く型付けされた言語仕様のオペランド</th><th>型付けのまったくない言語仕様のオペランド</th></tr>
<tr><td>ポインタ演算</td><td>ポインタ型／算術型</td><td>ポインタ型／算術型</td><td rowspan="6">算術型</td></tr>
<tr><td>算術演算</td><td>算術型</td><td rowspan="2">算術型</td></tr>
<tr><td>論理演算</td><td>論理型</td></tr>
<tr><td>文字列演算</td><td>文字列型</td><td>文字列型</td></tr>
<tr><td>比較演算</td><td rowspan="2">算術型／ポインタ型／論理型／文字列型</td><td rowspan="2">算術型／文字列型／ポインタ型</td></tr>
<tr><td>代入演算</td></tr>
<tr><td>代表的な高級言語仕様</td><td>PL/I, FORTRAN, COBOL ADA, PASCAL, ALGOL</td><td>PL/M</td><td>C</td></tr>
</table>

文字列演算／比較演算のいずれか）に展開する．このとき，中間語命令ごとに仮想レジスタ*が生成され，2 項演算の結果が格納される．仮想レジスタに格納された演算結果は，一つの代入文内に閉じて参照されることもあるし，手続き内であれば複数の代入文間にまたがって参照されることもある．

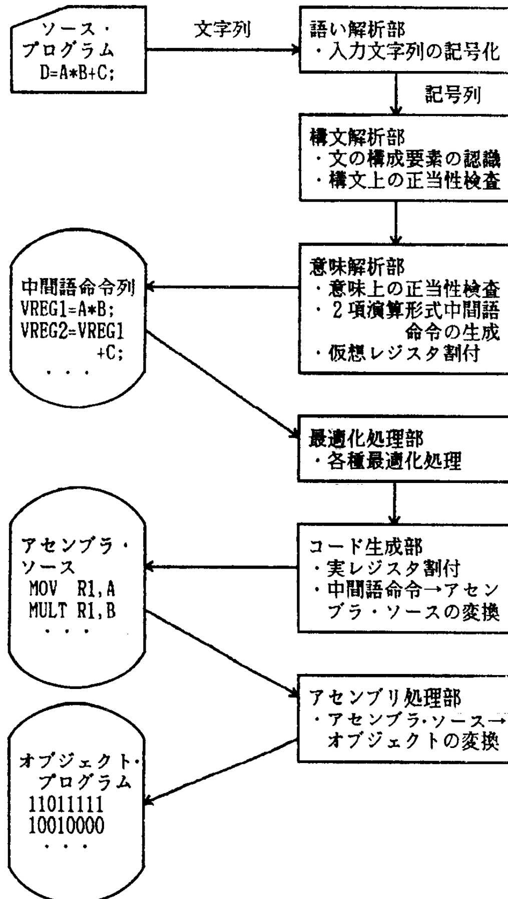


図 1　セルフ・コンパイラのフェーズ構成
Fig. 1　Phase organization of self-compiler.

* あらゆる種類の演算に使用可能で無限個存在すると仮定した汎用レジスタ．

② タイプ V-1（代入文の左辺）

代入文の左辺は，右辺の演算結果の仮想レジスタを代入先の変数に格納する，代入演算の中間語命令列に展開する．

③ タイプ V-2（条件分岐を伴う制御文・プログラム構造文）

IF 文や CASE 文等の条件分岐を伴う制御文・プログラム構造文は，比較演算を行う 2 項演算形式の中間語命令列と，比較演算結果の仮想レジスタの値に従って分岐する条件付分岐の中間語命令に展開する．

④ タイプVI（その他の制御文，プログラム構造文）

GOTO 文や手続きコール文等の制御文・プログラム構造文は，文内に演算式を含まないため，仮想レジスタの生成を伴わない中間語命令に展開する．このため，これらの文は，たとえ複数個の中間語命令に展開されても，中間語命令間でのデータの相互参照は発生しない．

〔例 1〕 中間語命令の展開例

（ソース・プログラム）

$ARI_1=(ARI_2+ARI_3*ARI_4)/ARI_5;$

⇩

（中間語命令列）

$\underline{VREG_1}=ARI_3*ARI_4;$

$\underline{VREG_2}=ARI_2+\underline{VREG_1};$

$\underline{VREG_3}=\underline{VREG_2}/ARI_5;$

$ARI_1=\underline{VREG_3};$

（注）　ARI_i：算術変数

$VREG_i$；仮想レジスタ

2.2 分割コード生成アルゴリズム

〔分割コード生成アルゴリズム〕

表 3 中間語命令の分類
Table 3 Classification of intermediate codes.

タイプ		大分類	小分類	機能	本論文中での中間語命令の表記法
I		代入文の右辺/制御文等の比較演算式	算術・ポインタ演算	加算 減算 乗算 除算 剰余	$VREG_i=ARI_j+ARI_k$; $VREG_i=ARI_j-ARI_k$; $VREG_i=ARI_j*ARI_k$; $VREG_i=ARI_j/ARI_k$; $VREG_i=ARI_j$ MOD ARI_k;
II		代入文の右辺/制御文等の比較演算式	論理演算	論理否定 論理和 論理積	$VREG_i$=NOT (BIT_j); $VREG_i=BIT_j$ OR BIT_k; $VREG_i=BIT_j$ AND BIT_k;
III		代入文の右辺/制御文等の比較演算式	文字列演算	結合 切り出し 挿入	$VREG_i=STR_j$ ^STR_k; $VREG_i$=SUBSTR (STR_j, k, m); SUBSTR (STR_i, j, k)=STR_m;
IV		代入文の右辺/制御文等の比較演算式	比較演算	算術比較 ポインタ比較 論理比較 文字列比較	$VREG_i=VAR_j$ (=| ^=|<|> |<=|>=) VAR_k;
V	1	代入文の左辺	代入演算	算術代入 ポインタ代入 論理代入 文字列代入 単純代入 ブロック代入	$ARI_i=VREG_j$; $ARI_i=VREG_j$; $BIT_i=VREG_j$; $STR_i=VREG_j$; $VAR_i=VAR_j$; (スカラ変数) $VAR_i=VAR_j$; (構造体, 配列等)
V	2			条件付き分岐	GOTO $LABEL_i$, $VREG_j$;
VI		制御文/プログラム構造文		無条件分岐 手続きコール 手続きのプロローグ 手続きのエピローグ ……	GOTO $LABEL_i$; CALL SUB_i; PROC ARG_i, ARG_j, … END ARG_i, ARG_j, …

注) ARI_i, ARI_j : 算術・ポインタ属性データ
BIT_i, BIT_j : 論理属性データ
STR_i, STR_j : 文字列属性データ
VAR_i, VAR_j : 任意の型属性データ
$VREG_i, VREG_j$: 仮想レジスタ
PAR_i, PAR_j : アーギュメント

強く型付けされた言語仕様で記述されたソース・プログラムの中間語命令列（論理属性・文字列属性から算術属性への型変換機能は除く）に対しては，次の手順に従ってばらばらにコード生成し，最後にそれをもとの中間語命令の順序にマージすれば，中間語命令列に対するコード生成を手続きの先頭から順に実行した場合と等価なオブジェクト・プログラムが生成できる．

〔コード生成手順 1〕

中間語命令列を手続きの先頭からサーチして，タイプ I の中間語命令だけ（算術演算・ポインタ演算）を，検出した順にコード生成する．

〔コード生成手順 2〕

中間語命令列を再び手続きの先頭からサーチして，タイプ II～IV の中間語命令だけ（論理演算・文字列演算・比較演算）を，検出した順にコード生成する．

〔コード生成手順 3〕

中間語命令列を再び手続きの先頭からサーチして，

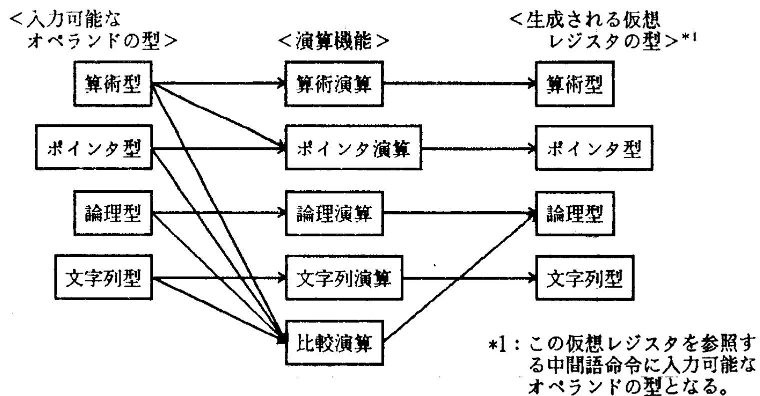


図 2 演算機能と仮想レジスタの型の関係
Fig. 2 Relation between functions and data types of virtual register.

タイプⅤの中間語命令だけ（代入演算と条件付き分岐）を，検出した順にコード生成する．

〔コード生成手順4〕

中間語命令列を再び手続きの先頭からサーチして，タイプⅥの中間語命令だけ（制御文・プログラム構造文）を，検出した順にコード生成する．ただし〔コード生成手順4〕は，〔コード生成手順1〕～〔コード生成手順3〕の任意の手順内で，コード生成してもよい．

〔証明〕

演算機能と，入力可能なオペランド（仮想レジスタも含む）の型，および生成される仮想レジスタの型の関係を，**図2**に示す．

仮想レジスタの型は，構文・意味解析時に，演算対象オペランドの型と演算機能の組合せによって，一意に決まる．ところが，仮想レジスタの，型以外の属性

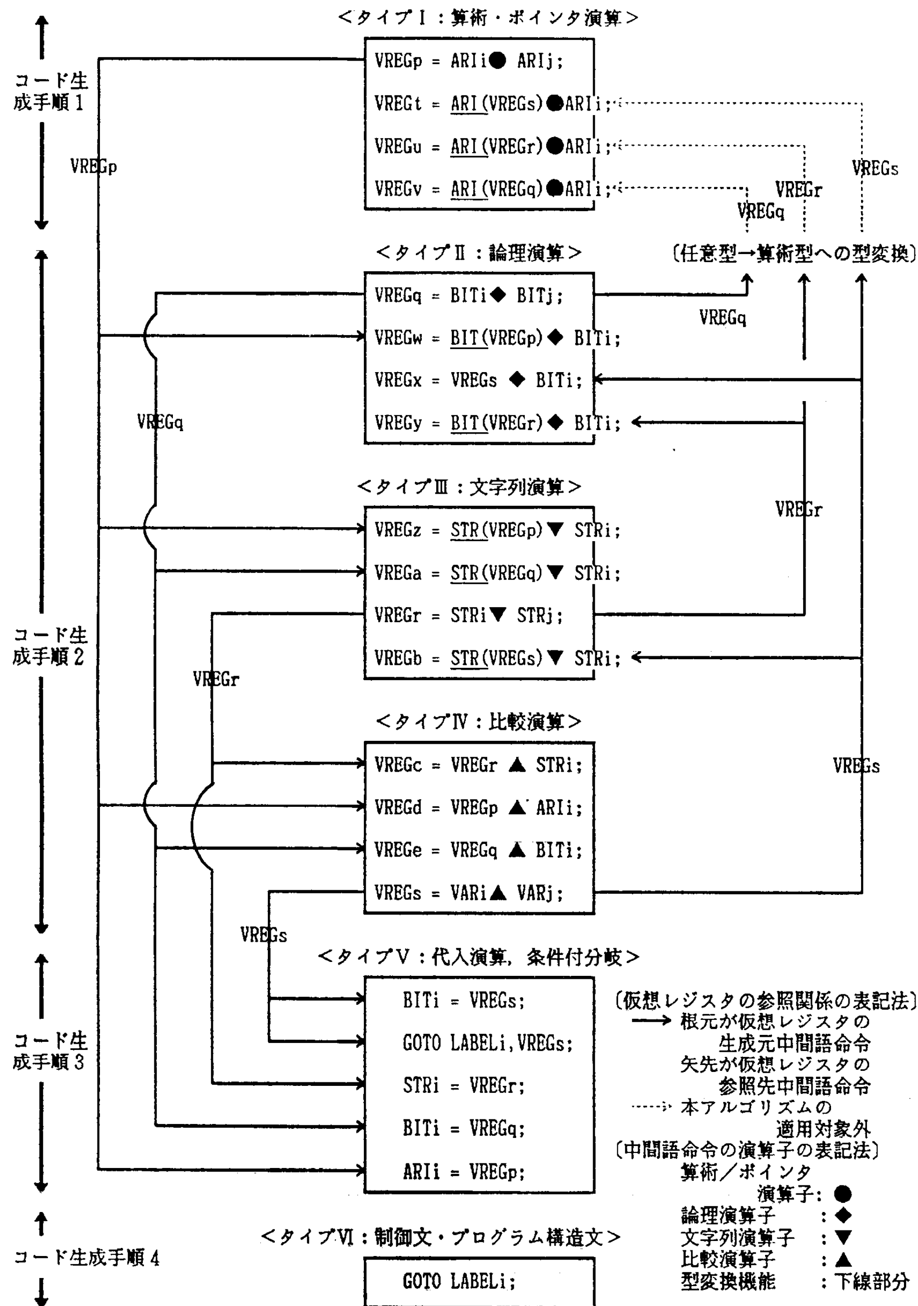


図3　中間語命令の仮想レジスタの参照関係
Fig. 3 Virtual register references among intermediate codes.

(演算結果のデータの精度，演算結果の仮想レジスタ上での格納開始ビット位置等）は，それを生成する中間語命令に対するコード生成を行って，はじめて決まる．この仮想レジスタを参照する中間語命令のコード生成を行うためには，この仮想レジスタの属性がわかっていなければならない．このため，仮想レジスタを生成する中間語命令より先に，仮想レジスタを参照する中間語命令のコード生成を行うことはできない．これらの観点より，中間語命令間での仮想レジスタの生成と参照について，あらゆる可能性を整理（**図3**）すると，次のことがわかる．

① タイプＩの中間語命令は，論理演算等の他のタイプの演算で参照される仮想レジスタを生成するだけで，他のタイプの演算結果の仮想レジスタを参照することはない．したがって，他のタイプの演算に先立ってコード生成する必要がある．

② タイプⅡ～Ⅳの中間語命令は，タイプＩの中間語命令で生成した仮想レジスタを参照すると同時に，これらの中間語命令間で，生成した仮想レジスタを相互に参照し合うため，同一時期にコード生成する必要がある．

③ タイプⅤの中間語命令は，タイプＩ～Ⅳの中間語命令で生成した仮想レジスタを参照するだけで，新たに仮想レジスタを生成することはない．したがって，仮想レジスタを生成するすべての中間語命令に対するコード生成が終了した後に，コード生成する必要がある．

④ タイプⅥの中間語命令は，仮想レジスタを生成することも参照することもないため，任意の時点でコード生成することができる．

以上より，中間語命令列を手続きの先頭から順にサーチして，最初にタイプＩの中間語命令だけを，取り出した順にコード生成する．続いて，タイプⅡ～Ⅳの中間語命令だけを，取り出した順にコード生成する．続いて，タイプⅤの中間語命令だけを，取り出した順にコード生成する．上記のいずれかのコード生成時，あるいは最後に，タイプⅥの中間語命令だけを，取り出した順にコード生成する．すべてのコード生成終了後に，意味解析部が生成した中間語命令列の順に，オブジェクト・プログラムをマージする．

このようにコード生成しても，その時点では，中間語命令間で引き継いだ仮想レジスタの生成元中間語命令は，コード生成済みであるため，その仮想レジスタの属性がすでに決まっており，問題はない．したがって，分割コード生成アルゴリズムを用いても，中間語命令列に対するコード生成を手続きの先頭から順に実行した場合と等価なオブジェクト・プログラムが生成できる（証明終り）．

2.3 型変換機能の処理方法

論理型・文字列型等の任意の型を算術型データに変換する（算術型変換）ために型変換機能が使用されると，タイプＩの中間語命令と，他のタイプの仮想レジスタを生成する中間語命令との間で，仮想レジスタの相互参照が発生するため，分割コード生成アルゴリズムの適用が不可能となる．

しかし，次の手法を意味解析部で採用することにより，分割コード生成アルゴリズムの適用が可能となる．

〔算術型変換の処理手法〕

算術型変換の発生可否は，ソース・プログラム上で陽に指定されなくても，意味解析部で識別可能である．算術型変換が発生した場合には，意味解析部において，算術型変換対象オペランドの属性から，変換処理結果の仮想レジスタの属性を予測する．これを，仮想レジスタにあらかじめ付与しておく．仮想レジスタの属性が複数パタン予測される場合は，安全な側の属性を付与する．

本手法は，参照先中間語命令で必要とする仮想レジスタの属性と異なる予測をした場合に，オブジェクト・プログラムの性能低下を生ずる場合もあるが，コンパイル・エラーや，オブジェクト・プログラムのバグを誘発することはない．

〔例2〕 算術型変換の処理方法

(ソース・プログラム)

$ARI_1 = ARI_2 + \underline{ARI(BIT_1\ AND\ BIT_2)} + ARI_3$;

⇩

(中間語命令列)

$VREG_1 = BIT_1\ AND\ BIT_2$; ……①

$\underline{VREG_2} = \underline{ARI(VREG_1)}$; …………②

$VREG_3 = ARI_2 + \underline{VREG_2}$; ………③

$VREG_4 = VREG_3 + ARI_3$; ………④

$ARI_1 = VREG_4$; ……………………⑤

(注) ARI（下線部）は，論理演算結果を一時的に算術属性に変換する算術型変換を示す．

この例では，意味解析部において，BIT_1 と BIT_2 の精度から仮想レジスタ $VREG_2$ の属性を予測（例えば，演算結果は 32 ビットで，仮想レジスタ上に右詰めで格納する）しておく．この情報を用いれば，

表 4 主なコード生成共通機能
Table 4 Common functions in code-generator.

機能分類		コード生成共通機能
コード生成条件判定機能	中間語命令のオペランドの状態判定機能	オペランドの格納状態（仮想レジスタ上／メモリ上／直接数値）
		型属性（算術／ポインタ／論理／文字列／…）
		仮想レジスタ種別（算術演算用／論理演算用／アドレッシング用）
		データの精度（有効データ長，格納開始ビット位置等）
		仮想レジスタの使用状態（ペア／シングル，破壊可／不可，等）
		アドレッシング状態（直接修飾／インデックス修飾／…）
		その他（内部／外部データ種別，リロケーション可否等）
コード生成機能	コード生成	スケルトン・テーブル起動制御
		オブジェクト生成コマンド
		その他（定数オペランドの算術演算・論理演算等）
	レジスタ管理	レジスタ割付（要求対象レジスタの割当，退避／回復）
		レジスタ使用状態管理（冗長オブジェクト生成の回避）
	データ接近	メモリ割付・使用状態管理
		ラベル生成・対応アドレス管理

①・②の中間語命令より先に，③・④の中間語命令に対するコード生成処理を行うことができる．

2.4 分割コード生成アルゴリズムの適用方法

コード生成部が，まったく同一のコード生成共通機能（表 4）をもつ複数のコード生成サブフェーズから構成される場合，スケルトン・テーブルは，図 4 の（ ）内に示す配置条件範囲内で，任意のコード生成サブフェーズに配置することができる．

各コード生成サブフェーズは，入力した中間語命令列の中から，割り当てられたスケルトン・テーブルに対応する中間語命令だけを選択・抽出し，コード生成する．

分割コード生成アルゴリズムを適用すると，中間語命令を二次記憶装置との間で入出力するため，コード生成の処理時間は増加するが，スケルトン・テーブル記述内容とコード生成処理順序の実質的な変更を伴わないため，オブジェクト・プログラムの性能は低下しない．

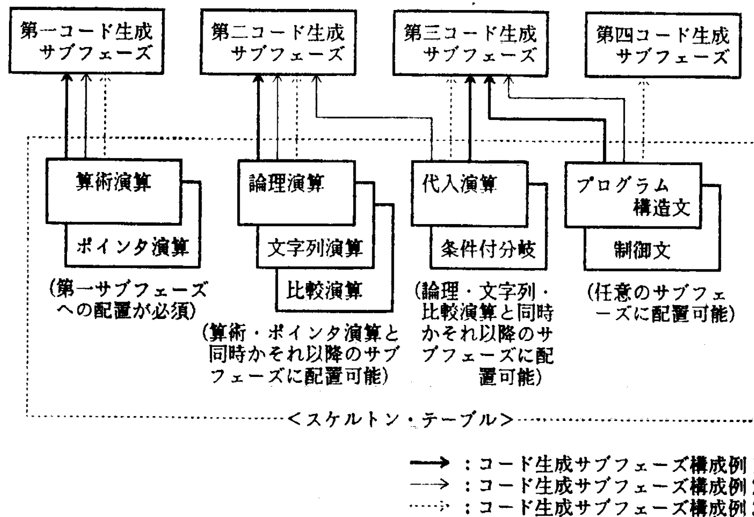


図 4 スケルトン・テーブルの配置規則
Fig. 4 Distribution rule of skeleton tables to sub code-generators.

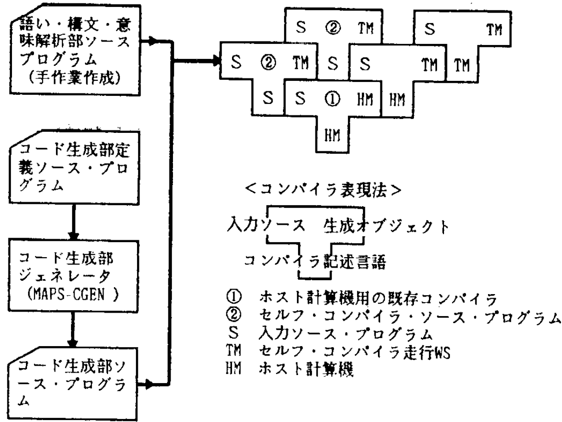


図 5　セルフ・コンパイラ生成方式
Fig. 5　Generation of self-compiler.

3. コード生成部の構成法

3.1　コード生成部ジェネレータの言語仕様

セルフ・コンパイラは，コード生成部ジェネレータとブート・ストラップ機構を用いて生成することを前提とした（図5）.

コード生成部は，コード生成部ジェネレータ（MAPS-CGEN：Multi-target And Portable Support software system-Compiler GENerator）に次の情報を与えることにより，ジェネレートする（図6）.

〔定義情報1〕

μP のアーキテクチャ情報（レジスタ構成，アドレッシング・モード，メモリ上データの反転配置有無,

```
$MINFO (アーキテクチャ定義)
  ADDR：2,2,2,・・・；（アドレス更新単位，命令境界，ワード境界，等）
  REVERSE ：N,R；（2／4バイト・データの主記憶上への格納方法）
      ・・・
$REGISTER  (レジスタ構成定義)
  R-GR1:@GR1,@GR2,・・・；  (レジスタ・クラスとレジスタの包含関係)
  @PR1 〔@GR1,@GR2 〕:1;    (レジスタのペア関係とレジスタ・コード)
      ・・・
$INSTRUCTION( 機械語命令仕様定義)
  ¥TYPE
    〔ADD,SUB 〕：
    /@GR1/ @GR1,@GR2/ 〔ADDR, SUBR〕 / OPC1 〔4〕,OFH 〔4〕, ・・・・
      ↑  演算対象データ     ↑            機械語命令コード
  演算結果格納先          ニーモニック
      ・・・
$SKELETON ( スケルトン・テーブル定義)
$SKELETON1 (第一コード生成サブフェーズ用スケルトン定義)
  ADD:ARIj,ARIk  (VREGi=ARIj+ARIk;のスケルトン・テーブル)
   <判定条件>                  <条件エントリ>
   C LENG  〔ARIj〕 = 8        'Y N Y . N . ' →(ARIj は8ビット精度か？)
   C LITF  〔ARIj〕            'N N Y Y . . ' →(ARIj はイミディエート・
   C LENG  〔ARIk〕 = 8        'Y N Y N . . '              データか？)
   C LITF  〔ARIk〕            'N N N N . . '
   C   ・                      '     .      '
   C   ・                      '     .      '
   C   ・                      '     .      '

  <オブジェクト生成コマンド>    <処理エントリ>
   A GETREG  〔R-GR1,WR1 〕    'X X X . . . ' →(R-GR1に属する任意のレジスタWR1 の確保)
   A OUT 〔LOAD WR1,ARIj 〕    'X X X . . . ' →(ARIj →WR1 のオブジェクト生成)
   A OUT 〔ADD WR1,WR1,ARIk〕  'X X X . . . ' →(ARIj+ARIk→WR1 のオブジェクト生成)
   A OUT 〔LOAD WR1,ARIk 〕    '. . . X X X '
   A OUT 〔ADD WR1,WR1,ARIj〕  '. . . X X X '
   A RESULT  〔WR1,16〕        '. X . . . . ' →(WR1の仮想レジスタVREGi への対応付け,
   A RESULT  〔WR1, 8〕        'X . X . . . '          演算結果の精度は16ビット)
   A   ・        コード生成──→ '     .      '
   A   ・          ルール1      '     .      '
   A   ・                      '     .      '

              〔スケルトン・テーブル記述規則〕
              条件エントリ  Y:条件判定結果が真
                            N:条件判定結果が偽
                            .:条件判定しない
              処理エントリ  X:オブジェクト生成コマンドを実行
                            .:何もしない

〔$SKELETON2 (第二コード生成サブフェーズ用スケルトン定義) 〕
      ・・・
〔$SKELETON3 (第三コード生成サブフェーズ用スケルトン定義) 〕
      ・・・
〔$SKELETON4 (第四コード生成サブフェーズ用スケルトン定義) 〕
      ・・・
$ACCESS  (データ接近方法規定用スケルトン定義)
      ・・・
$SAVEREG (レジスタ退避方法規定用スケルトン定義)
      ・・・
$LOADREG (レジスタ回復方法規定用スケルトン定義)
      ・・・
```

図 6　MAPS-CGEN 入力ソース・プログラムの構造
Fig. 6　Language specification of MAPS-CGEN.

データのバイト境界／ワード境界等への配置方法，等)

〔定義情報 2〕

機械語命令の仕様

〔定義情報 3〕

スケルトン・テーブル（コード生成サブフェーズ用，データ接近処理用，レジスタ退避・回復処理用）の仕様

スケルトン・テーブルは，コンパイル時に必要とする主記憶容量の削減，作成工数・デバグ工数の削減，コード生成条件の網羅性確認の容易化，および保守性向上のために，デシジョン・テーブル形式で記述する．

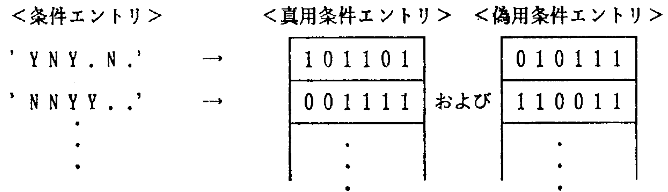
スケルトン・テーブルは，いくつかの特殊条件に対する異なったオブジェクト生成と，それ以外の条件に対する同一オブジェクト生成という形に分類されるのが一般的である．このため，コード生成ルール（条件エントリと処理エントリを含めて縦方向一列分）はこの順に定義し，コード生成時には，コード生成ルールごとに条件エントリを左側から順に検査して，最初に満足するものを選択する方式を採用し，スケルトン・テーブル定義量を削減した．

条件エントリ	真用ビット値	偽用ビット値
Y	1	0
N	0	1
·	1	1

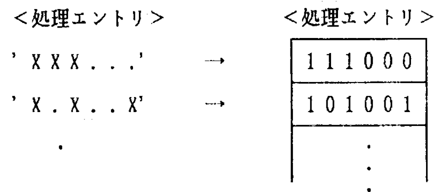
（a） 条件エントリ変換規則

処理エントリ	ビット値
X	1
·	0

（b） 処理エントリ変換規則



（c） 条件エントリのビット・パタン形式化処理例



（d） 処理エントリのビット・パタン形式化処理例

図 7　スケルトン・テーブルのビット・パタン形式への変換規則
Fig. 7　Transformation rule of skeleton table.

なお，コード生成部のオペレーティング・システム・インタフェースとオブジェクト・ファイル・インタフェースは，仮想化したが，走行環境間でほとんど共通性がないため，テーブル形式定義が困難であり，個別に変換ロジックを記述してコード生成部に結合する方式を採用した．

MAPS-CGEN の入力となるセルフ・コンパイラ定義ソース・プログラムの構造は，PMP-CGEN[8]のスケルトン定義部の構造を，コード生成部のフェーズ分割アルゴリズムが適用できるように，改良したものである．具体的には，スケルトン・テーブルを，配置先コード生成サブフェーズごとにグループ化して定義できるようにした．

3.2　コード生成処理手法

スケルトン・テーブルの規模は，Kirk が提案したテーブル展開方式（マスク法）[9]を修正したビット・パタン展開方式（**図 7**）を採用することにより，Pollack らが提案したロジック展開方式（トリー法）[10),11)]に比べて，大幅に削減することができた．さらに，ビット・パタン形式に展開されたスケルトン・テーブルは，左側のコード生成ルールから順に検査すればよいため，論理積を反復実行するだけの簡単なコード生成処理手法が採用できる（**図 8**）．これにより，コード生成処理時間は Pollack の方法より 2 割程度増加するが，コンパイル時にスケルトン・テーブルが占有する主記憶容量を 1/6 程度まで削減できるので，WSの主記憶容量制限が厳しい場合には非常に有効である（**表 5**）．

3.3　データ接近機能・レジスタ管理機能のコード生成サブフェーズ化手法

データ接近機能とレジスタ管理機能については，次の考え方を導入して，それぞれ，コード生成部から分離・サブフェーズ化することを可能とした．

① データ接近機能は，一つのオブジェクト・コード（機械語命令）に閉じた範囲で，オペランドのアドレッシング・モードを決定するものであり，複数のオブジェクト・コード間の相互関連はない．しかも，分岐命令の分岐先アドレス決定処理をアセンブル時に行えば，データ接近結果のアドレッシング用オブジェクト・コードを，コード生成結果のオブジェクト・コード中にあとから挿入しても矛盾は生じない．

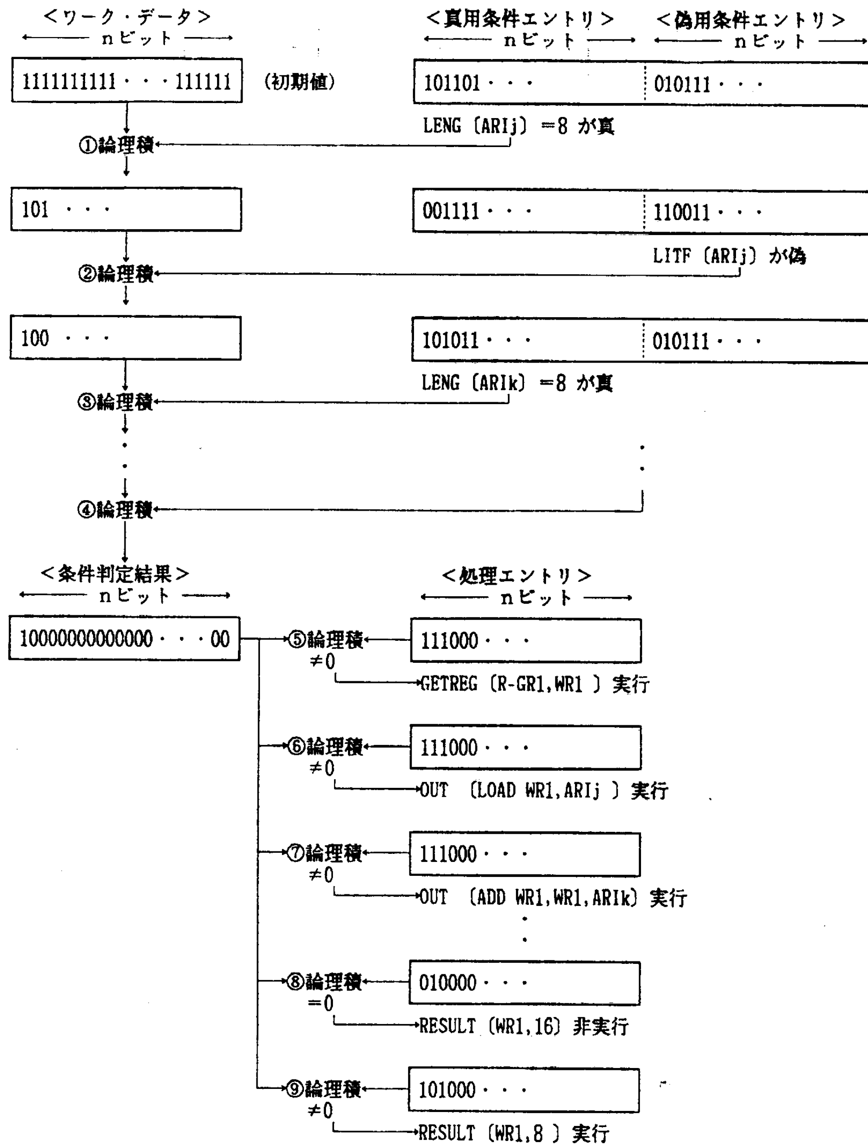


図 8 コード生成処理手法
Fig. 8 Code-generation technique.

表 5 コード生成処理方式の比較注)
Table 5 Performance of code-generator.

比較 ＼ コード生成処理方式	ロジック方式 (Pollack の tree 法)	テーブル方式 (今回提案した方式)
スケルトン・テーブルの形式	ロ ジ ッ ク	ビット・パタン
1 スケルトン・テーブルの規模	1,260バイト	175バイト
スケルトン・テーブルの総規模	128kバイト	23kバイト
コード生成共通機能の規模	1kバイト	1.8kバイト
条件判定処理機能の所在	スケルトン・テーブル	コード生成共通部
条件判定回数	1～判定条件数分	判定条件数分
コード生成用論理積実行回数	0	処理エントリ数分
コード生成処理時間比	1	1.2
使用可能主記憶容量に制限がある場合の総合評価	×	○

注) 16ビット系μP用スケルトン・テーブルの試作結果

表 6 MAPS-C のコード生成サブフェーズ構成
Table 6 Sub code-generator configuration of MAPS-C.

サブフェーズ / コンパイラ名	CGS-1 (第一コード生成)	CGS-2 (第二コード生成)	CGS-3 (第三コード生成)	DAS (データ接近)	RMS (レジスタ管理)
MAPS-C1注)	全中間語命令のコード生成	なし	なし	なし (CGS-1 サブフェーズで処理)	
MAPS-C3	算術・ポインタ演算のコード生成	論理・文字列・比較・代入演算のコード生成	その他の演算のコード生成	なし (CGS-1, -2, -3 の各サブフェーズで処理)	
MAPS-C5	算術・ポインタ演算のコード生成	論理・文字列・比較・代入演算のコード生成	その他の演算のコード生成	データ接近方法の決定と，そのコード生成	レジスタ使用状態管理と，レジスタ退避・回復用のコード生成

注) MAPS-Ci: i はコード生成サブフェーズ数を示す

② コード生成用，およびデータ接近用のスケルトン・テーブルは，レジスタ名はいっさい使用せず，すべてレジスタ・クラスだけによって記述するように制限する．こうすることによって，レジスタを割付ける前に，コード生成処理・データ接近処理ができる．

これらの性質を利用して，①第1～第4コード生成サブフェーズ⇨②データ接近処理サブフェーズ⇨③レジスタ管理サブフェーズ，の順に処理すれば，矛盾なくコード生成できる．本手法は，仮想レジスタのレベルでアドレッシング用オブジェクトの生成可否を判断する必要があるため，オブジェクト・プログラムの性能低下を誘発する場合もある．

以上，分割コード生成アルゴリズムと，データ接近機能・レジスタ管理機能のコード生成サブフェーズ化手法を組み合わせることにより，コード生成部を最大6サブフェーズに分割することが可能となった．

4. 評　　価

MAPS-CGEN を用いて作成したコード生成部と，個別作成した語い・構文・意味解析部を結合して，PL/I ライクの言語仕様をもつ次の3種類のセルフ・コンパイラ(MAPS-C)[12]を作成した（表 6）．

(1) MAPS-C1

分割コード生成アルゴリズムを適用しない，コード生成部が1フェーズ構成のコンパイラ

(2) MAPS-C3

分割コード生成アルゴリズムを適用した，コード生成部が3サブフェーズ構成のコンパイラ

(3) MAPS-C5

分割コード生成アルゴリズムに加えて，データ接近機能・レジスタ管理機能のコード生成サブフェーズ化手法も適用した，コード生成部が5サブフェーズ構成のコンパイラ

ビット・パタン形式展開したスケルトン・テーブルを用いたコード生成処理手法は，全コンパイラに適用した．

評価[13]は，次の前提条件のもとで行った．

① ソース・プログラム，中間語命令列，オブジェクト・プログラム等の入出力単位は，1,024 バイト単位．

② 評価対象ソース・プログラムは，コンパイル処理，画像端末用図形処理，大型計算機用通信処理，仮想端末用通信処理，機械制御，およびテレ

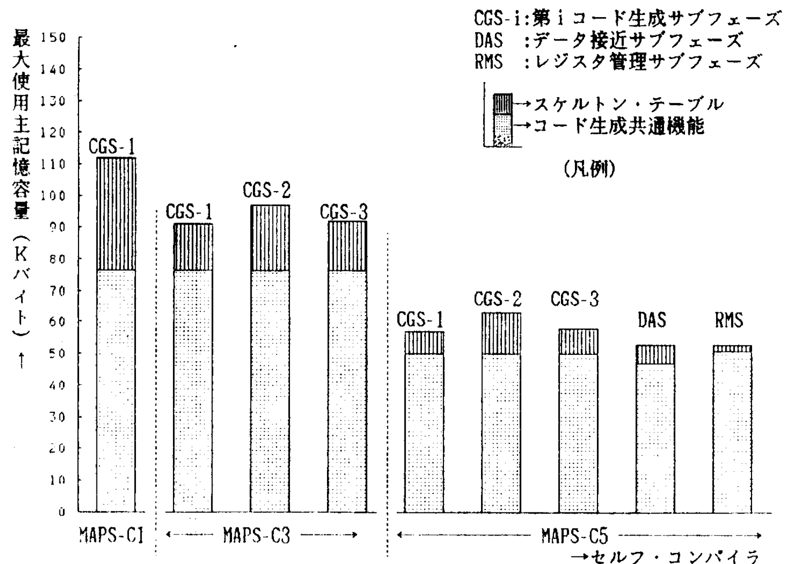


図 9 MAPS-C コード生成サブフェーズごとの最大使用主記憶容量
Fig. 9 Code-generator size of MAPS-C.

ビ・ゲームから，総計約 10k ステップを抽出.

4.1 コード生成部の最大使用主記憶容量

MAPS-C コード生成サブフェーズごとの最大使用

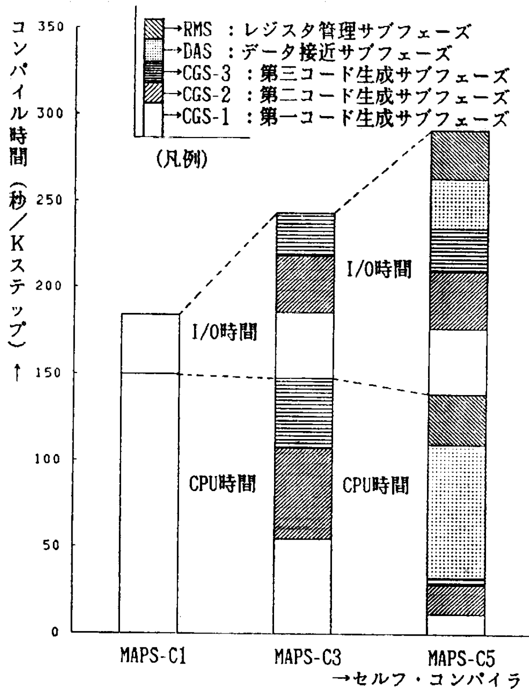


図 10 MAPS-C コード生成部のコンパイル時間
Fig. 10 Compiling speed of MAPS-C code-generator.

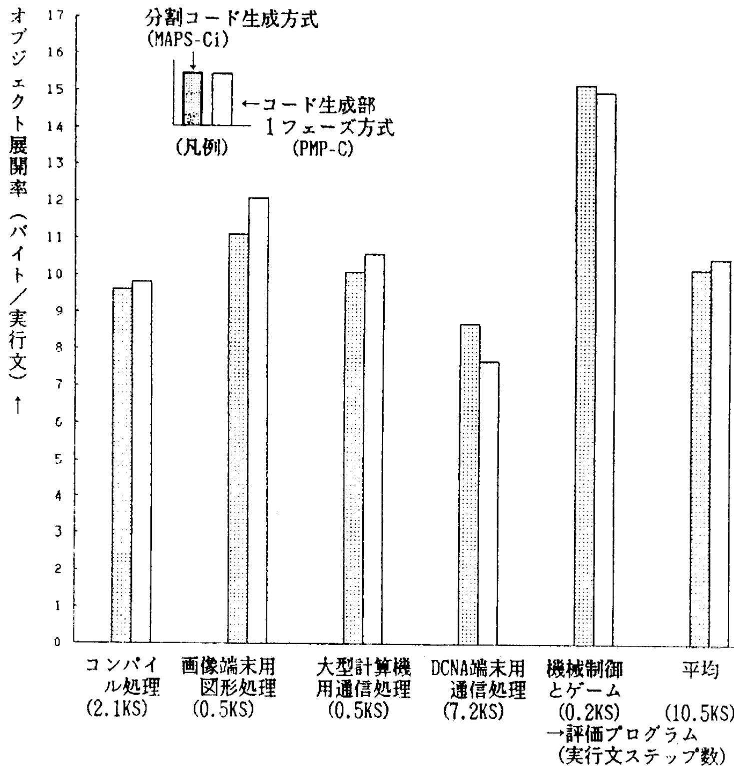


図 11 オブジェクト・プログラムのメモリ占有量に関する性能
Fig. 11 Space efficiency of MAPS-C object program.

主記憶容量と，最大スケルトン・テーブル規模を，**図 9** に示す.

コード生成部が使用する最大主記憶容量は，MAPS-C1 で約 120k バイト，MAPS-C3 で約 100k バイト，MAPS-C5で約 64k バイトである.

4.2 コード生成処理時間

1k ステップのソース・プログラム（データ宣言文：実行文の割合がほぼ 1：3 のもの）に対するサブフェーズ別コード生成時間の比較を，**図 10** に示す.

CPU 時間については，コード生成部のサブフェーズ分割によるオーバヘッドがほとんど発生しない.

データ入出力時間については，コード生成サブフェーズ数が一つ増加するごとに，1k ステップ当たり約 30 秒のオーバヘッドを生ずる.

なお，1k ステップのソース・プログラム入力からオブジェクト・プログラム生成までの総コンパイル時間は，MAPS-C1 が約5分，MAPS-C3 が約6分，MAPS-C5 が約7分である.

4.3 オブジェクト性能

MAPS-C が生成するオブジェクト・プログラムのメモリ占有量と実行速度に関する性能を，次の二つの観点から比較した.

① PMP-C との性能比較

語い・構文・意味解析処理方式がほぼ同一で，コード生成処理方式のみが異なる PMP-C クロス・コンパイラ（コード生成部1フェーズ方式）と性能比較した結果，ほぼ同等のオブジェクト性能が得られた. オブジェクト・プログラムのメモリ占有量に関する性能評価結果を，**図 11** に示す.

② サブフェーズ分割数とオブジェクト性能との関係

分割コード生成アルゴリズムが，スケルトン・テーブルの記述内容，およびコード生成処理ロジックの変更を伴わないため，コード生成部のサブフェーズ分割数によらず，オブジェクト性能は一定となった.

4.4 コード生成部作成の生産性

MAPS-Cセルフ・コンパイラはMAPS-CGEN ソース・プログラム定義から，デバッグ完了まで約8人月で作成することができた. コード生成部の作成規模は，MAPS-CGEN ソース・プログラムが約2k ステップ，OS インタフェース・ルーチンが約1

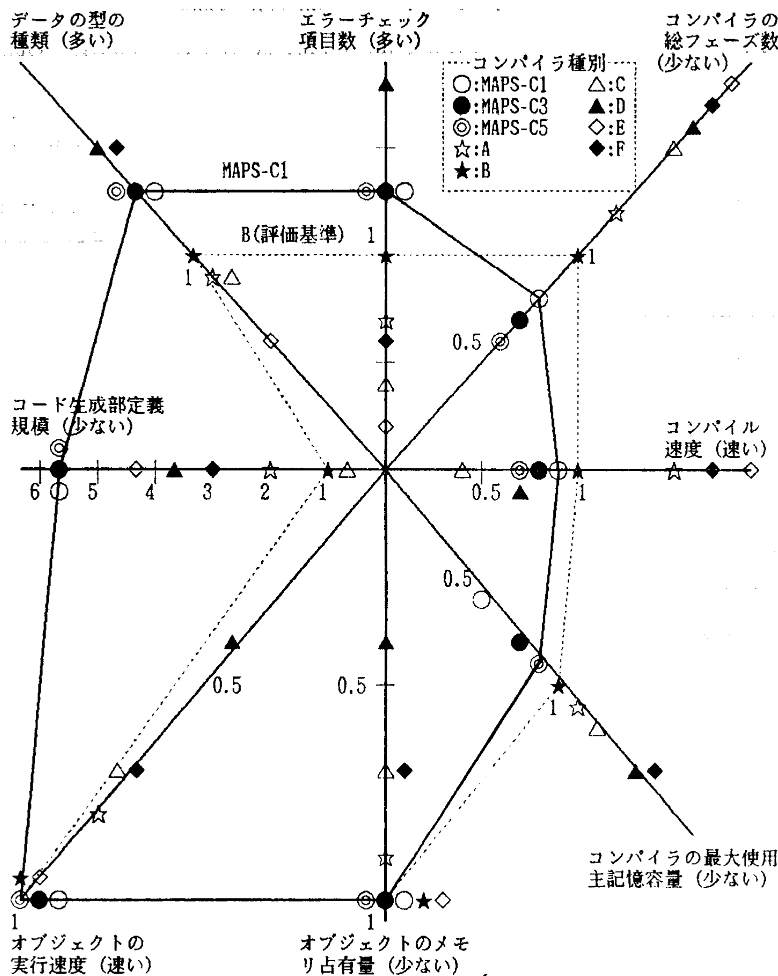


図 12　MAPS-C の総合評価（Bを基準とした相対評価）
Fig. 12　Total evaluation of MAPS-C.

kステップ，オブジェクト・ファイル・フォーマット定義ルーチンが約1kステップ，合計4kステップである．ちなみに，個別作成の専用セルフ・コンパイラのコード生成部作成規模は，10〜30kステップにも達する．

4.5 総合評価

MAPS-C セルフ・コンパイラと，特定 16 ビット系 μP 用に個別作成された，市販の専用セルフ・コンパイラ（図中の A〜F）の，同一環境条件下における総合評価結果を，**図 12** に示す．

コンパイル時間については，個別作成方式で有力な専用セルフ・コンパイラ (B) に比べて，コード生成部1フェーズ構成の MAPS-C 1で1割，5サブフェーズ構成の MAPS-C 5で3割の入出力時間のオーバヘッドが生ずる (CPU時間のオーバヘッドは生じない)．

セルフ・コンパイラの最大使用主記憶容量，オブジェクト・プログラムの性能（メモリ占有量，実行速度）については，Bセルフ・コンパイラと同等である．MAPS-Cコード生成部の定義規模は，Bセルフ・コンパイラに比べて，1/6 程度であり，セルフ・コンパイラ作成の生産性は，大幅に向上することができる．

5. むすび

本論文では，最大使用主記憶容量が小さく，しかも高性能なセルフ・コンパイラ用コード生成部を，ジェネレータ方式で作成可能とするために，次のようなコンパクト化手法を提案した．

① 強い型付けをもった高級言語仕様に対して適用可能な，分割コード生成アルゴリズム

② ビット・パタン形式に展開したスケルトン・テーブルを用いたコード生成処理手法

③ データ接近機能・レジスタ管理機能のコード生成サブフェーズ化手法

本手法を，クロス・コンパイラ作成用コード生成部ジェネレータ技術 (PMP-CGEN) と組み合わせて，セルフ・コンパイラ作成用コード生成部ジェネレータ (MAPS-CGEN) を作成し，ジェネレートされるセルフ・コンパイラ (MAPS-C) の性能を評価した．

コード生成部が使用するワークステーションの最大使用主記憶容量については，②だけを用いて，コード生成部1フェーズ構成とした場合，120k バイト程度に削減可能である．これに，①の分割コード生成アルゴリズムを適用して，コード生成部を3コード生成サブフェーズに分割することにより，100k バイト程度に削減可能である．さらに，③を適用することにより，64k バイト程度にまで削減可能である．

コード生成処理時間については，1k ステップのソース・プログラム当たりで，約 30 秒/1 コード生成サブフェーズの入出力時間のオーバヘッドが生ずる．

オブジェクト・プログラムの性能（メモリ占有量，オブジェクトの実行速度）は，コード生成部のサブフェーズ分割数による影響はなく，専用セルフ・コン

パイラを個別作成した場合と同等である.

セルフ・コンパイラ・コード生成部作成の生産性については，定義規模が約4kステップとなり，専用セルフ・コンパイラのコード生成部を個別作成した場合に比べて，1/6程度に削減できた.

MAPS-Cは，データ通信用端末や高周波伝送システム等において，移植性を考慮したμP用ソフト開発の有効手段として利用されている.

MAPS-CGENは，語い・構文・意味解析部を自動生成するコンパイラ・ジェネレータと組み合わせることにより，容易に多機種・多言語仕様を適用対象とする高性能セルフ・コンパイラ・ジェネレータが実現できる．この点に着目して，MAPS-CGENの設計思想は，NTTで開発中のμP用マルチ・ターゲットADAコンパイラ・ジェネレータ作成に活用されている.

謝辞 最後に，MAPS-CGENおよびMAPS-Cセルフ・コンパイラの作成・評価にあたってご指導いただいた，花田収悦ソフトウェア生産技術研究所長，論文作成にあたり貴重なコメントをいただいた高橋宗雄同調査役，ならびに関係各位に感謝の意を表する.

参考文献

1) Aho, A.V.: Translator Writing Systems: Where Do They Now Stand?, *IEEE Comput.*, Vol. 13, No. 8, pp. 9-14 (1980).
2) Ganapathi, M. et al.: Retargetable Compiler Code Generation, *ACM Comput. Surv.*, Vol. 14, No. 4, pp. 573-592 (1982).
3) Mckeemann, W.M. et al.: *A Compiler Generator Implemented for the IBM system/360*, Prentice-Hall, Inc., Englewood Cliffs (1970).
4) 佐々：コンパイラ生成系，情報処理，Vol. 23, No. 9, pp. 802-817 (1982).
5) 国立他：スケルトンの作成を容易化した多機種向きコード生成部を持つコンパイラ，情報処理学会論文誌，Vol. 22, No. 4, pp. 281-287 (1981).
6) Bunza, J.: *Automatic Multi-Target Code Generation for μPs*, MIT, Cambridge (1978).
7) Johnson, S.C.: A Portable Compiler-Theory and Practice, Proc. 5th ACM Symp. on Principles of Programming Languages, pp. 97-104 (1978).
8) NTT: PMP（汎用）説明書，その11（PMP-CGEN), DEMOS マニュアル (1983).
9) Kirk, H.W.: Use of Decision Tables in Computer Programming, CACM 8, pp. 41-43, (1965).
10) Pollack, S.L.: *Analysis of the Decision Rules in Decision Tables*, The Rand Corp., Santa Monica (1963).
11) Pollack, S.L.: Conversion of Limited-entry Decision Tables to Computer Programs, CACM 8, pp. 677-682 (1965).
12) NTT: MAPS-C 説明書（拡張形 PMP), DEMOS マニュアル (1983).
13) 杉山 他：多機種用コンパイラ・ジェネレータの評価，信学会研究会，EC 83-13 (1983).

(昭和60年4月19日受付)
(昭和60年9月19日採録)