組込みシステムのためのコンパイラフレームワークライブラリ

A Compiler Framework Library for Embedded Systems

中西 恒夫 [1)]　福田 晃 [2)]　平尾 智也 [3)]
Tsuneo Nakanishi　Akira Fukuda　Tomoya Hirao

[1)] 奈良先端科学技術大学院大学情報科学研究科（〒 630-0101 奈良県生駒市高山町 8916-5，E-mail: tun@is.aist-nara.ac.jp）
[2)] 九州大学大学院システム情報科学研究院（〒 816-8580 福岡県春日市春日公園 6-1，E-mail: fukuda@f.csce.kyushu-u.ac.jp）
[3)] 奈良先端科学技術大学院大学情報科学研究科（〒 630-0101 奈良県生駒市高山町 8916-5，E-mail: tomoya-h@is.aist-nara.ac.jp）

ABSTRACT. This paper describes implementation of Compiler Framework Library, or CFL, which is a C++ class library to ease embedded compiler construction by differential programming based on class inheritance. Programmers derive subclasses from those classes and describe codes dependent on target embedded processors. CFL keeps the amount of processor-dependent codes in a Z80 prototype code compression assembler less than about 15 percent of the whole codes including CFL itself.

1　背景

大容量の半導体メモリ素子が安くなった今日においても，組込み向けプロセッサのオンチップメモリは貴重なハードウェア資源である．コスト削減，信頼性向上，軽量化のためには外部メモリの利用は避けられるならば避けるべき選択である．また，限られた電池電力で長時間動作しつづけることが要求される携帯電話，携帯情報端末等の携帯組込み機器においては，消費電力削減の点からも外部バスを駆動する必要のないオンチップメモリの活用は重要である．プログラムをオンチップメモリに格納するためには，プログラムのサイズそのものを小さくする必要がある．

筆者らはメモリ制約の厳しい組込み機器向けに，目的コードのサイズが最小化されるような命令セットを有するプロセッサとその上で動作する目的コードとを協調生成する，プロセッサ／目的コードコジェネレータの開発を進めている．プロセッサ／目的コードコジェネレータでは，いったんプログラムを適当なコンパイラでコンパイルし，得られた目的コード中の頻出命令列を新しい複合命令として定義し，最終的に得られる目的コードのサイズを縮小する．同時に複合命令を処理するロジックをプロセッサに組み込み，そのハードウェア記述を VHDL の形で出力する．図 1 はプロセッサ／目的コードコジェネレータの概要図である．

一方，組込みプロセッサ向けのコンパイラでは，プロセッサと目的コードの協調生成はしないものの，同様のコード最適化処理が以前から行われており，頻出命令列を関数化し，頻出命令列の出現を関数呼出に変換することにより，目的コードサイズの縮小を図っている．しかしながら，複雑なコーディングと高い信頼性が同時に要求されるコンパイラの実装は，多数のプログラマと長い開発工期を必要とする．組込み機器では，一部の有力メーカによるプロセッサの寡占化が進んでいる PC とは異なり，白物家電や自動車のエンジン制御に使用される 4 ビットプロセッサから，ハイエンド家庭用ゲーム機や情報家電に使用される 32 ビットプロセッサまで，様々のメーカの

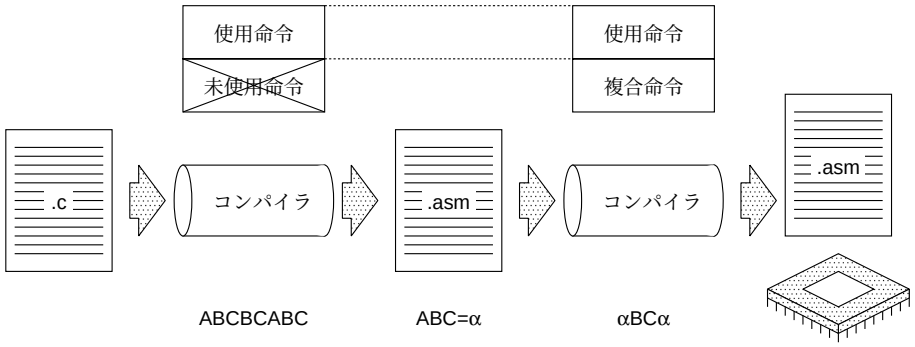

図 1: プロセッサ／目的コードコジェネレータ

様々のプロセッサが使用されている．組込み機器を支えるこれら多種多様の組込み向けプロセッサのための，コンパイラ，アセンブラ，デバッガ等の開発環境そのものの実装を支援する技術の開発が必要である．

2 目的

前節に述べた背景を鑑み，筆者らは，多種多様な組込みプロセッサ向けコンパイラの開発工期の削減を目的として，継承による差分プログラミングによって容易に対象プロセッサ向けのコンパイラを実装し，またその機能を拡張できる C++ クラスライブラリ，Compiler Framework Library (CFL) の実装を企図している．CFL の実装はプロセッサ／目的コードコジェネレータ実現のための布石でもある．CFL 実装の第一段階として，組込みシステムの分野では今後もメモリの問題が重要であり続けることを考慮にいれて，CFL にコード圧縮機能を有するアセンブラを容易に実装するためのクラス群を，平成 12 年度未踏ソフトウェア創造事業の支援を受けて実装した．

続く節では，CFL の概要，現在の実装，ならびに CFL を用いた対象プロセッサ向けのコード圧縮アセンブラの実装方法について概説する．また，CFL による実装容易化効果，ならびに CFL を用いて実装されたコード圧縮アセンブラのコード圧縮性能を示す．

3 CFL の概要とクラス構成

本節では CFL の概要とそのクラス構成について述べる．クラス構成については，誌面の都合上，CFL でコード圧縮アセンブラを実装する際に重要なものにとどめる．CFL の詳細については文献 [4] を参照されたい．

(1) CFL の概要

CFL によって提供されるクラスには，字句解析や構文解析など本質的に対象プロセッサに依存しないコンパイラのコードと，命令など対象プロセッサに依存する部分を抽象化することで対象プロセッサ依存を回避するためのコードが実装されている．プログラマは，これら対象プロセッサ非依存のクラスから導出した派生クラスに，対象プロセッサ依存のコードや，CFL の機能を拡張するコードを実装する．

CFL は，C，C++，Java 等の高級言語のコンパイラを実装するためのフレームワークライブラリを目指しているが，現在のクラスで実装できるのはコード圧縮アセンブラのみである．CFL で実装される全てのコード圧縮アセンブラには，1) コード圧縮機能，2) Strength Reduction，3) コード最適化スイッチング擬似命令，4) 論理セグメント機能，5) オペランドにおける定数式記述，6) 値マクロ機能，7) 条件付アセンブル機能，8) ファイルインクルード機能，9) エ

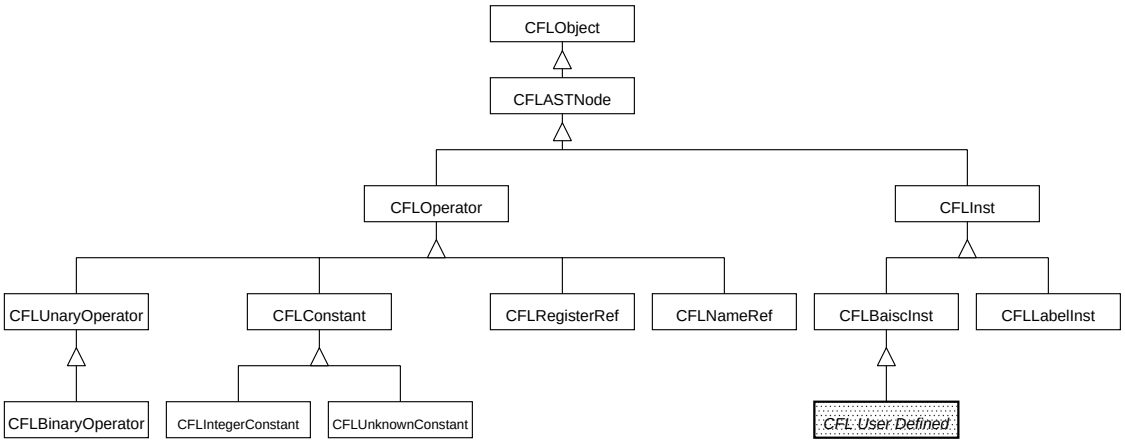

図 2: 抽象構文木関連クラス階層図

スケープシーケンスを用いた文字定数ならびに文字列記述機能，10) マルチバイト文字対応，の機能が標準で提供される．

CFL が依存するライブラリは，ANSI C/C++ で標準化されているライブラリと Standard Template Library (STL) のみであり，またコンパイル環境および実行環境の差を吸収するクラスの整備により，CFL 自体の可搬性は高く保たれている．

(2) アプリケーション関連クラス

CFL によって構築されるアセンブラの骨格部分となるのは CFLAsmApp クラスである．CFLAsmApp クラスには，アセンブラを初期化するメンバ関数，字句解析，構文解析，意味解析，およびコード最適化を行うメンバ関数やそれらを駆動するメンバ関数，アセンブラの終了時処理を行うメンバ関数，ならびに名前表などの共通的なデータ構造が実装されている．CFLAsmApp クラスまたはその派生クラスのインスタンスは唯一つ存在し，アセンブラの起動から終了に至るまでの一連の処理を制御する．

プログラマは最初に，CFLAsmApp クラスから新しいクラスを導出し，メンバ関数を適当にオーバーライドすることで，対象プロセッサ用のアセンブラを構築する．一般には，字句をトークンに変換するメンバ関数，プロセッサに依存する論理セグメントを生成するメンバ関数，ならびに名前表から論理セグメントを検索するメンバ関数をオーバーライドするのみでよい．

(3) 抽象構文木関連クラス

CFL は構文解析の際に抽象構文木ベースの中間表現を構築する．抽象構文木の節点は全て CFLASTNode クラスの派生クラスのオブジェクトである．

抽象構文木の節点は，大きくオペランド式などの式の要素を表す節点と，命令を表す節点とに分類される．前者は CFLASTNode クラスの派生クラス CFLOperator クラスの派生クラスのオブジェクトで，後者は CFLASTNode クラスの派生クラス CFLASTNode の派生クラスのオブジェクトである．

抽象構文木関連クラスのクラス階層図を図 2 に示す．

CFLOperator 系クラスは，オペランドや値マクロのボディ部，.IF や .ELSEIF などの条件アセンブル指令の条件式など，式の抽象構文木の節点を表す．式を表す抽象構文木の節点は，その機能や性質によって以下のクラスに分類される．

- **CFLOperator:** 式の要素を表すあらゆる節点

の抽象基底クラス．式を表す抽象構文木に共通して適用される操作がメンバ関数として実装されている．このクラスのオブジェクトが生成されることはない．

- **CFLUnaryOperator:** +や-などの単項演算子を表す節点．このクラスの節点が抽象構文木の葉となることはない．
- **CFLBinaryOperator:** +，-，*，/などの二項演算子を表す節点．このクラスの節点が抽象構文木の葉となることはない．
- **CFLConstant:** あらゆる定数を表す節点．このクラスは抽象クラスであり，オブジェクトが生成されることはない．
- **CFLIntegerConstant:** 整定数を表す節点．このクラスの節点は常に抽象構文木の葉となる．
- **CFLUnknownConstant:** 未初期化定数（?）を表す節点．このクラスの節点は常に抽象構文木の葉となる．
- **CFLRegisterRef:** レジスタ参照を表す節点．このクラスの節点は常に抽象構文木の葉となる．
- **CFLNameRef:** 名前参照を表す節点．このクラスの節点は常に抽象構文木の葉となる．

CFLInst 系クラスは，対象プロセッサの命令，データ定義などの擬似命令，ラベル命令など，命令の抽象構文木の節点を表す．命令を表す抽象構文木の節点は，その機能や性質によって以下のクラスに分類される．

- **CFLInst:** 命令を表すあらゆる節点の抽象基底クラス．命令を表す抽象構文木に共通して適用される操作がメンバ関数として実装されている．このクラスのオブジェクトが生成されることはない．
- **CFLBasicInst:** プロセッサの実命令，または擬似命令を表す節点．このクラスには，データ定義命令などプロセッサに依存しない命令に関する項目が実装されている．
- **CFLLabelInst:** ラベル命令を表す節点．

プログラマは，CFLBasicInst クラスから対象プロセッサの命令を表す派生クラスを導出し，プロセッサに依存する項目を実装する．具体的には，命令の意味的正当性を検査するメンバ関数，他の命令との等価性を検査するメンバ関数，命令のサイズを返すメンバ関数などを実装する．

4 コード圧縮の原理

CFL で実装されるコード圧縮アセンブラは，プログラム中に重複して出現する頻出命令列を関数呼出に置換することで，コードサイズの縮小を図る．

CFL に実装されている頻出命令列検出アルゴリズム 1 は，Fraser らによるアルゴリズム [2] を改良したものであり，頻出命令列検出のために Suffix 木と呼ばれる木構造を生成する．Suffix 木の根を除く個々の節点にひとつの命令とひとつのカウンタが属性として付与されている．アルゴリズム 1 は，個々の節点 v_n のカウンタの値として，v_n から根 v_0 へ至るパス $\langle v_n, v_{n-1}, v_{n-2}, \ldots, v_0 \rangle$ の節点に属性として与えられた命令の列 $\langle c_n, c_{n-1}, c_{n-2}, \ldots, c_1 \rangle$ の出現回数を設定する．

アルゴリズム 1 【頻出命令列検出アルゴリズム】
プログラムの命令列 $\langle c_1, c_2, \ldots, c_n \rangle$ を入力として受け取り，入力された命令列に対応する Suffix 木を生成する．カウンタの値の大きな節点に対応する命令列が頻出命令列である．

1. $i := n$
2. $i = 0$ ならばアルゴリズム終了．
3. 入力命令列より命令 c_i を取得する．
4. 根に新たなトークンを配置する．
5. 全てのトークン t_j について

(a) t_j の位置する節点が属性として命令 c_i を持つ子を有する場合，t_j をその子に移動．

(b) t_j の位置する節点が属性として命令 c_i を持つ子を有しない場合，c_i を属性として持つ子を生成し，t_j をその子に移動する．生成した子のカウンタを 0 にリセットする．

(c) t_j の位置する節点のカウンタをインクリメントする．

6. $i := i - 1$

7. 2 へ飛ぶ． □

頻出命令列を関数に変換し，頻出命令列の出現を関数呼出に置き換えた場合のコードサイズの削減効果は，おおよそ当該頻出命令列のバイト数と出現回数の積となる．この積の値は，v_n に付与されているカウンタの値と，v_n から根 v_0 に至るパス上の節点に付与されている命令の列 $\langle c_n, c_{n-1}, ..., c_1 \rangle$ から求められる．CFL は，この積の値が閾値以上の全ての命令列を積の値の大きい順に関数に変換し，命令列の出現を関数呼出に置き換える．但し，スタック操作を伴う命令，スタックポインタを更新する命令を含む命令列を関数に変換すると，関数呼出前にスタックに退避された戻り番地が正しく取得できなくなりプログラムが誤動作する．また，ジャンプ命令を含む命令列を関数に変換することも，やはりプログラムの誤動作につながる．このような命令列は関数に変換しない．

頻出命令列検出アルゴリズムの詳細については文献 [3] を参照されたい．また，CFL ではコード圧縮と同時に Strength Reduction[1] も適用可能となっているが，誌面の都合上その説明を割愛する．

5 CFL によるコード圧縮アセンブラの実装

CFL で実装されるコード圧縮アセンブラは，従来のコンパイラと同様に，原始プログラムに対して字句解析ならびに構文解析を施して構文上のエラー検査と中間表現の生成を行う．その後，中間表現に対して意味解析を施して意味上のエラー検査を行った後に，コード圧縮とコード生成を行う．本節では，CFL を用いたコード圧縮アセンブラの，字句解析機能，構文解析機能，意味解析機能，コード圧縮機能の実装方法について述べる．

(1) 字句解析機能の実装

CFL では，字句解析機能はプロセッサ非依存部とプロセッサ依存部に分けて実装される．

演算子やデータ定義命令，擬似命令などプロセッサに依存しない字句は CFLAsmApp クラスの字句解析メンバ関数群でトークンに変換される．一方，ニーモニックなどプロセッサに依存する字句は，プログラマが新たに実装する CFLAsmApp クラスの派生クラスにおいて，オーバーライドされた字句解析メンバ関数群でトークンに変換しなければならない．

プログラマが一般的にオーバーライドするのは，識別子や予約語となる字句をトークンに変換するメンバ関数 `getIdentifier (i18nint)` である．CFLAsmApp クラスの同メンバ関数は，プロセッサに依存しない予約語しか予約語トークンに変換せず，他は全て識別子トークンに変換する．ゆえに，CFLAsmApp クラスの派生クラスの同メンバ関数は，CFLAsmApp クラスの同メンバ関数を呼び出して予約語と認識されなかった，すなわち識別子と認識された字句が，さらに対象プロセッサのニーモニック等の予約語でないかを検査し，対象プロセッサの予約語ならば相当するトークンを返し，そうでなければ識別子トークンを返すように実装する．

(2) 構文解析機能の実装

CFL の構文解析機能は，CFLAsmApp クラスのメンバ関数 `parse ()` において実装されており，このメンバ関数は yacc によって生成される．CFL の yacc の形式文法記述は，プロセッサ非依存文法記述とプロセッサ依存文法記述とに分離されている．プ

ロセッサ非依存文法記述は，演算子やデータ定義命令，擬似命令などプロセッサ非依存の終端記号の記述，文や式の生成規則の記述である．一方，プログラマが新たに記述しなければならないプロセッサ依存文法記述は，対象プロセッサのニーモニックを表す終端記号の記述，ならびにこれらの終端記号からプロセッサ非依存文法記述の非終端記号を導出する生成規則のみである．

(3) 中間表現の構築とパターンマッチング

構文解析の際に CFL はプログラムの中間表現を構築する．この中間表現は図 3 に示すような構造をしている．プログラムで唯一の CFLAsmApp クラスまたはその派生クラスのオブジェクトには，論理セグメント，すなわち CFLSegment クラスまたはその派生クラスのオブジェクトが出現順にリストで接続される．各論理セグメントオブジェクトには，その論理セグメントに属する命令，すなわち CFLInst クラスの派生クラスのオブジェクトが出現順にリストで接続される．各命令オブジェクトには，その命令のオペランド式の根，すなわち CFLOperator クラスの派生クラスのオブジェクトへのポインタが，第一オペランド，第二オペランド，…の順にリストで接続される．

CFLInst 系クラス，ならびに CFLOperator 系クラスには，中間表現のリンク構造にパターンマッチングを適用するメンバ関数 `match (char *)` が実装されており，プログラマによる意味解析を容易にしている．同メンバ関数の引数には Lisp 様のパターン記述言語で中間表現の形状を指定する．同メンバ関数は，中間表現が引数で指定した形状にマッチする場合は真を，マッチしない場合は偽を返す．

本稿では誌面の都合により，パターン記述言語の詳細な記述は割愛し，例を挙げるにとどめる．例えば，`(#add | #sub, @a, 1)` は，中間表現が「`add a, 1`」，または「`sub a, 1`」にマッチすることを求める．また，`(#add, @a, @b | @c | (#op_bracket, (#op_add, @ix, %CEXPR)))` は，中間表現が「`add a, b`」，「`add a, c`」，「`add a, [ix + 定数式]`」のいずれかにマッチすることを求める．上述の例で，`#x` は予約語または演算子 `x`，`@r` はレジスタ `r`，`%CEXPR` は定数式を意味している．

(4) 意味解析機能の実装

CFL の意味解析機能は CFLInst クラスのメンバ関数 `check ()` において実装されている．このメンバ関数は，当該命令が意味的に正しいものならば真を，そうでなければ偽を返す．意味解析は，中間表現中の個々の命令，すなわち CFLInst クラスのオブジェクトの `check ()` を順次呼び出すことにより行われる．

前節で述べたように，プログラマは CFLBasicInst クラスから対象プロセッサの命令を表す派生クラスを導出し，プロセッサに依存する項目を実装する．プログラマは，その派生クラスの `check ()` をオーバーライドし，対象プロセッサの当該命令が意味的に正しいものかどうかを検査するコードを記述する．意味的正当性を検査するコードは，抽象構文木のパターンマッチングを行う `match (char *)` メンバ関数を用いて簡単に記述することができる．

図 4 はその実装例である．CFLBasicInst クラスの派生クラス CFLSampleInst クラスにおいて，メンバ関数 `check ()` をオーバーライドしている．データメンバ `token` には当該命令のトークンが格納されている．トークン，すなわち命令の種類ごとに `match (char *)` メンバ関数を用いて意味的に正当な命令かどうかを検査し，正当な場合は真を返している．当該命令が意味的に正当でないものならば，`check ()` の最後で偽が返される．

(5) コード圧縮機能の実装

CFL は，意味解析が完了すると，前節で述べたアルゴリズム 1 により Suffix 木を生成する．アルゴリズム 1 では，命令の詳細はまったく問題ではなく，

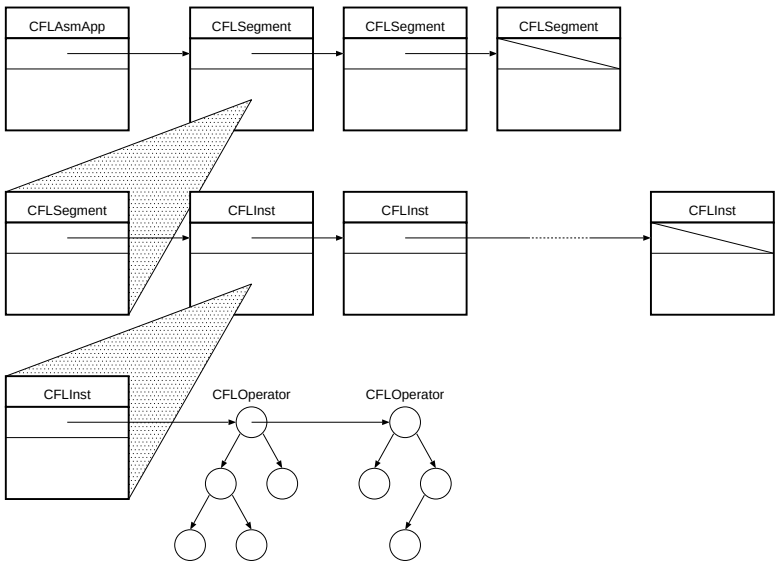

図 3: 中間表現

```
bool
CFLSampleInst::check ()
{
  switch (token)
    {
    case CFL_AT_LD:
      if (match ("(#ld, @a, @a|@b|@c|@d|@e|@h|@l|\
@i|@r|(#op_bracket, @hl|@bc|@de))"))
        return true;
      if (match ("(#ld, @a, (#op_bracket, (#op_add|\
#op_sub, @ix|@iy, %CEXPR)))"))
        return true;
      break;
    ... (中略) ...
    }
  return false;
}
```

図 4: check () の実装例

Suffix 木を生成するためには命令が同じものか否かと，圧縮効果を見積もるための命令のバイト数の情報が必要となる．また前節で述べたように，Suffix 木から得られる頻出命令列を関数に変換する際は，プログラムを正しく動作させるべく，頻出命令列中の各々の命令がスタックを操作するものであったり，ジャンプ命令ではないかの情報が必要となる．

CFL では，命令を CFLBasicInst クラスとして抽象化することにより，コード圧縮アルゴリズムのフレームワークを提供している．CFLBasicInst クラスには，他の命令と自身とが等しいか否かを判別するオーバーロード演算子 ==，自身のサイズを返すメンバ関数 `getCodeSize ()`，自身がスタックを操作するものであったりジャンプ命令でないかを返すメンバ関数 `isPackable ()` が実装されている．プログラマは，CFLBasicInst クラスから対象プロセッサの命令を表すクラスを導出し，そのクラスにおいて ==，`getCodeSize ()`，`isPackable ()` をオーバーライドすることで，コード圧縮アセンブラを実装することが可能となる．これらのメンバ関数の実装は，図 4 の `check ()` メンバ関数と同様に，中間表現に対してパターンマッチングを施し，真偽値，あるいはコードサイズを返すものとなる．

6 評価

CFL によってコード圧縮アセンブラの実装がどのくらい容易になるかを評価するべく，CFL を用いて

Z80 用のコード圧縮アセンブラを実装し，Z80 依存部分のコード量を調査する．また，実装したコード圧縮アセンブラを用いて ADPCM の原始プログラムをアセンブルし，そのコード圧縮能力を評価する．

CFL のみのコード量は 11,000 行強（yacc による自動生成コードの行数は含まない）に対し，Z80 用にカスタマイズした派生クラスの追加実装のコード量は 2,000 行程度であった．Z80 は，豊富な種類の命令を持つ CISC であり，命令の直交性も低く，ゆえに意味解析などプロセッサ依存部分のコードが多くなるにも関わらず，プロセッサ依存部分は CFL 自体も含めた全体の約 15% 程度に抑えられている．コード量は必ずしも実装の難易度を図る尺度となりえないことを認めても，前節で述べたように，追加実装部分は抽象構文木にパターンマッチングを施し，命令の意味的正当性やバイト数を返すのみの単調なパターン化されたコードであり，CFL による対象プロセッサ向けコード圧縮アセンブラの実装容易化効果は極めて高いといえよう．命令の種類が少なく，命令の直行性が高い RISC では，さらにプロセッサ依存部分のコード量が抑えられるものと考えられる．

また，ADPCM の圧縮なしの場合のコードセグメントのサイズは 6,000 バイトであったのに対し，圧縮した場合のコードセグメントのサイズは 5,006 バイトになり，約 17% の圧縮率を達成していることが確認された．なお，上述のコード圧縮能力の評価では，圧縮対象とはならないデータセグメントのサイズ，ならびに原始プログラムを入手できなかったライブラリのコードセグメントのサイズは含まれていない．

7　まとめ

本稿では，多種多様な組込みプロセッサ用のコンパイラを，継承による差分プログラムによって容易に実装できる C++ クラスライブラリ，Compiler Framework Library (CFL) の実装について述べた．現在の CFL では，コード圧縮機能を有するアセンブラを容易に実装するためのクラス群が提供されている．CFL を用いて Z80 用コード圧縮アセンブラを実装したところ，プログラマが新たに記述するプロセッサ依存部分のコード量は，CFL 自体も含めた全体の約 15% に抑えられていた．また，そのコード圧縮アセンブラを用いて ADPCM のソースコードをアセンブルしたところ，コードセグメントのサイズが約 17% 圧縮されることを確認した．

C 言語への対応，ならびに種々のコード最適化アルゴリズムの実装が当面の課題である．併せて，CFL によって実装されるコンパイラの信頼性を向上させるべく，テスト支援機構を実装していきたい．また，CFL によるプロセッサ／目的コードコジェネレータの実装も今後の課題である．

参加機関

本プロジェクト（契約件名：組込み向けコード圧縮クロスアセンブラフレームワークの開発，プロジェクトマネージャ：高田広章・豊橋技術科学大学助教授）は，情報処理振興事業団による平成 12 年度未踏ソフトウェア創造事業の支援を受けて，奈良先端科学技術大学院大学，(財）九州システム技術研究所によって実施されました．

参考文献

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools,* Addison-Wesley, 1986.

[2] C. W. Fraser, E. W. Myers, and A. L. Wendt, “Analyzing and Compressing Assembly Code,” *Proc. of the 1984 ACM SIGPLAN Symp. on Compiler Construction, SIGPLAN Notices,* Vol.19, No.6, pp.117–121, 1984.

[3] 中西 恒夫, 中野 猛, 福田 晃, 「辞書式コード圧縮支援機構の遺伝的アルゴリズムによる最適化」, 情処研報, Vol.2001, No.39, 2001-ARC-143, pp.31–36, 2001 年 5 月.

[4] 中西 恒夫, 福田 晃, 「Compiler Framework Library (CFL): コード圧縮アセンブラの実装」, コンピュータシステムシンポジウム論文集, （2001 年 11 月発表予定）.