TBCppA: 追跡子を用いたC前処理系解析器

TBCppA: tracer-based C preprocessor analyzer

権藤克彦*　川島勇人†

あらまし Cプログラムの正確な解析には，C前処理系（以後，CPP）の処理前後のマッピング情報が必須である．従来は既存のCPP（例：Cpplib）を拡張して，このマッピング情報を得ていた．しかしこの手法はCPPの実装に強く依存するため，**保守や移植のコストが大きい**という問題があった．

この問題を解決するために，我々は新しい**追跡子方式**を文献 [19] で提案した．追跡子方式では既存のCPPに手を加えず，そのままCPPを利用してマッピング情報を得る．このため追跡子方式には**低い保守・移植コスト**と**高い適用性**という大きな利点がある．

文献 [19] では基礎となる重要なアイデアを提案したが，その実現性や有効性を十分には示していなかった．そこで本論文では追跡子方式を用いたCPP解析器であるTBCppAを実装することで，我々の**追跡子方式は実現可能**であることを示す．また予備評価では機能と性能の両面で**追跡子方式は有用**という結果を得たことを示す．

1　はじめに

C言語 [16] は現在でも幅広く使用され，OSやサーバアプリケーションなど重要な基盤ソフトウェアで特に多用されている [23]．このためC言語用の開発ツールは重要であるが，C前処理系（以後，CPP）やポインタエイリアスなどの問題のため，未だに十分なツールを提供できていない．例えば，既存のC言語用のコールグラフ生成系やリファクタリングツールは，精度が低かったり，機能が限定的であることが知られている [6,8,10,13,15]．本論文ではこの**CPPの問題**に焦点をあてる．

```
/* gzip-1.2.4, gzip.c, 888行目 */
#ifdef NO_FSTAT
  if (stat(ofname, &ostat) != 0) {
#else
  if (fstat(ofd, &ostat) != 0) {
#endif
    fprintf(stderr, "%s: ", progname);
      ⋮
  }
```

図1　前処理前は中カッコの対応が取れていないため，直接，構文解析するのが難しい例．

CPPの主な機能は，マクロ定義，条件付きコンパイル，ファイルの取り込みである．ほとんどのCプログラムはCPPの機能を多用している．例えばErnstらの調査 [4] では`#define`などの前処理指令は全体行数のうち平均8.4%，一行あたりのマクロの使用は平均0.28個である．このためErnstの指摘通り [4]，C言語の解析ツールはCPPの前処理指令やマクロ呼出しを扱う必要がある．

CPPの問題には大きく次の2つがある．（詳細は [4,5] を参照）．

- **CPPの問題1**: 前処理**前**のCプログラムの正確な構文解析は非常に難しい．例えば，図1は実在するコードで，前処理前は中カッコの対応が取れていないため，構文解析が難しい．CPPではマクロ展開で予約語を書き換えたり，任意のトークン間に前処理指令を置ける．この非構造性がCPPに多大な柔軟性を与えると同時に，ツール構築を著しく困難にしている．
- **CPPの問題2**: 前処理**後**のCプログラムは正確に構文解析できる．しかし解析後に，展開されたマクロや前処理指令を前処理**前**の状態に復元することが難しい．前処理後にはマクロ名などの情報が失われるためである．例えば`stdin`は`(&__iob[0])`に展開され，元に戻すための情報をコンパイラは提供しない．

正確な解析結果が必要なツールでは**問題1**の解決は事実上不可能である．例外として，字句解析主体で近似的な構文解析が許されるツール（例えばGNU GLOBAL）や，前処理指令の出現場所に強い制限（例えば`#ifdef`は関数前後のみ出現）があるツールは，ある程度，前処理前のCプログラムを直接処理できる．しかし，これら

*Katsuhiko Gondow, 東京工業大学

†Hayato Kawashima, 北陸先端科学技術大学院大学

は正確な解析をできない，適用範囲が限定されるという欠点を持つ．正確な解析は，特にソースコードを書き換えるツール（例：リファクタリングツール）では，バグを埋め込まないために必須となる．

問題 2 も切実である．前処理後のソースコードを解析しても，例えば (`&__iob[0]`) から `stdin` を得られないため，コードは非常に読みにくいし，ユーザに有用な情報 (CPP による抽象化) を提供できないからである．**問題 2** の解決には前処理前後のマッピング情報，例えば「12 行目のマクロ `stdin` は (`&__iob[0]`) に展開した」という情報を正確に提供することが必須である．

従来研究 [1–3, 7, 9, 11–15, 22, 23, 25, 26] は次の方法で前処理前後のマッピング情報を得ている．

- **コンパイラオプション方式，デバッグ情報方式**: 例えば，GCC の `-dI` や `-dM` オプションの出力や，DWARF2 デバッグ情報から前処理情報を得る．
- **エミュレータ方式**:CPP のエミュレータを実装，マッピング情報を出力させる．
- **既存 CPP 改造方式**: 既存の CPP を修正して，マッピング情報を出力させる．

例えば PCp^3 [1, 2] は既存 CPP 改造方式であり，GCC の C 前処理系 Cpplib を改造して利用している．

残念ながら，従来研究にはどの方法にも問題がある．

- **コンパイラオプション方式，デバッグ情報方式の問題**:
  容易に正確な情報を得られるが，肝心なマクロ呼出しの展開情報が欠けている．
- **エミュレータ方式の問題**:
  精度が低い．これは次の点で CPP の動作の模倣が難しいからである．
  - CPP 仕様の処理系定義・未規定動作: 例えば `#`や`##`の評価順序 ([16], 6.10.3), ファイル検索パス[1] ([16], 6.10.2)，非標準の `#pragma` の意味 ([16], 6.10.6).
  - CPP 実装の独自拡張: 例：GCC の `#include_next`，可変長引数マクロ[2].
  - CPP 実装の独自定義マクロ: システム定義のマクロ（例えば `stdin`）の展開結果は言語処理系に付随するヘッダファイルの内容により異なる．またヘッダファイルで定義されない独自定義マクロがある．例えば，Cygwin-1.5.19, GCC-3.4.4 には `unix`，`i386` など独自定義マクロが 79 個ある．
  - CPP 仕様への準拠の困難さ: CPP の動作は一見単純だが実際は複雑である．仕様や既存実装のオプション機能を正確に模倣すること自体に手間がかかる．GCC-3.4.4 での CPP の実装規模は約 15,000 行である．
- **既存 CPP 改造方式の問題**:
  改造した CPP は特定プラットフォームでしか動作せず適用範囲が狭い．改造した CPP を保守・移植するコストも高い．例えば PCp^3 のソースコードは公開されているが，我々の知る限り，1999 年で保守が止まっている．PCp^3 の CPP は非常に古い GCC 2.7.2.2 ベースのため，そのままでは使用できない．

この問題の解決のため我々は新しい**追跡子方式**を提案した [19]．追跡子方式では既存の CPP に手を加えず，そのまま CPP を利用してマッピング情報を得る．このため追跡子方式には上記の問題が少ないという大きな利点がある．

[19] のアイデアは，前処理前の C ソースファイル中に**追跡子**(tracer) を埋め込んで，CPP の動作を観測する点にある．一般に追跡子とは環境調査などの用語で，観測を容易にする特殊な微量物質 (例えば放射性同位体) を指す．追跡子を付加して追跡子を観測することで，観測しにくいものの動きや分布を簡単に観測できるようになる．本論文の追跡子方式では，CPP の処理対象のトークン列に対して追跡子となるトークンを付加し，これ

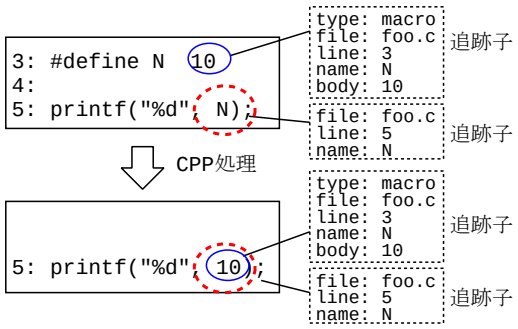


図 2　追跡子のアイデア：マクロ展開の変化を観察

[1] <>で囲ったファイル名の検索パス．たいてい/usr/include だが/usr/include とは限らない．

[2] C99 [16] で導入された可変長引数マクロと機能はほぼ同じだが構文が異なる．

を観測して CPP の動作を簡単に観測する．例えば，図 2 ではマクロ呼出し (N) の文脈とマクロ定義のボディ (10) に追跡子情報を付加してから，通常の CPP で CPP 処理を行う．CPP 処理後，2 つの追跡子は同じトークン 10 を指す．これは CPP がマクロ N を展開するとき，マクロボディ 10 に付随する追跡子も一緒に展開するからである．この結果から「10 の展開前は N であり，3 行目のマクロ定義で展開された」というマッピング情報を得られる．

文献 [19] では追跡子の重要なアイデアを提案したものの，その実現性や有効性を十分には示していなかった．追跡子方式では既存の CPP を利用し，追跡子はその CPP の動作に影響を与えてはいけないため，**追跡子方式の実装はトリビアルではない**．そこで本論文では追跡子方式のシステム TBCppA を実装して，我々の**追跡子方式は実現可能**であることを示す．また予備評価では機能と性能の両面で**追跡子方式は有用**という結果が得られたことを示す．

本論文の構成は次の通り：2 節は [19] の追跡子のアイデアに具体的な定義を与える．3 節は TBCppA の設計と実装を，4 節は予備評価を与える．5 節は制限を述べ，6 節は関連研究と比較する．7 節でまとめと結論を述べる．

2 追跡子

1 節では追跡子 [19] の基本的かつ概念的なアイデアを述べた．この節でトークン列による追跡子の具体的な表現方法を示す．ただし，ここでは例だけを用い，形式的な定義はしない[3]．正確な定義は TBCppA のソースコード [24] を参照されたい．

2.1 追跡子：基本構文

以下が追跡子の基本構文である（注意：元の定義 [19] とは異なる）．

```
@”タグ名 属性名=’ 値’ 属性=’ 値’ ...” 内容 @”/タグ名”
```

ただし,「値」は XML エスケープ済みとする[4].「内容」はこの段階では XML エスケープしてはいけない．この追跡子は CPP 処理後に次の XML 要素に変換する．

```
<タグ名 属性名=’ 値’ 属性=’ 値’ ...> 内容 </タグ名>
```

ただしこの「内容」は XML エスケープ済みとする．C の言語仕様上，文字@と文字列リテラルは CPP 処理で変化しないので，この追跡子の基本構文は CPP に対して透明である．ただし次の文脈は追跡子が CPP の動作に影響を与える：#if などの条件式（2.3 節），演算子#と## （3.5 節），関数マクロ名に展開するマクロ（5 節）．

2.2 追跡子：マクロ定義とマクロ呼出し

以下の関数形式マクロ定義とマクロ呼出しがあるとする．

```
#define F(x)   ((x)+10)
F(20)
```

これに次の通り追跡子を埋め込む.（ただし替え玉マクロ（2.3 節）は未使用）．静的にはマクロ呼出しかどうか判断できないので，追跡子は保守的に埋め込む[5].

```
@"define type=’func’ id=’F23_1’ name=’F’ args=’x’ macro_body=’((x)+10)’ locinfo"
  @"param name=’x’/"
@"/define"
#define F(x) @"macro_body ref=’F23_1’" @"param name=’x’" x @"/param"   \
               ((@"macro_call type=’param’ name=’x’ locinfo"            \
                   x                                                   \
                 @"/macro_call" )+10)                                  \
```

[3] この追跡子は細かいテクニックの集合体なので，形式的・網羅的な定義は退屈で膨大になるため．

[4] 文字< > & ’ "をそれぞれ < > & ' " に変換にすること．

[5] つまり「識別子」や「識別子 (···)」の形式は全てマクロ呼出しと判断して追跡子を埋め込む．CPP 処理後に変化が無ければ非マクロ呼出しと判断する．

```
            @"/macro_body"
@"macro_call type='func' name='F' args='20' locinfo"
  F(@"arg exp='20' locinfo"  20  @"/arg")
@"/macro_call"
```

ただし *locinfo* は次の形式の属性名と値の集まりであり，元ソースコードのファイル番号と位置情報（開始行，開始カラム，終了行，終了カラムの番号）を表す．

```
filename_ref='F23' first_line='1' first_column='6' last_line='3' last_column='7'
```

XML 変換時に filename_ref は IDREF 型にする．この IDREF 情報は次の形式の XML 要素へのリンクとなる．

```
<filename id="F23" index="23" file="foo.c" path="/tmp/foo.c"/>
```

追跡子を埋め込んだソースコードを CPP 処理して XML 形式にすると以下になる．

```
<define type='func' id='F23_1' name='F' args='x' macro_body='((x)+10)' locinfo>
  <param name='x'/>
</define>
<macro_call type='func' name='F' args='20' locinfo>
  <macro_body ref='F23_1'> <param name='x'>  20  </param>
    ((  <macro_call type='param' name='x' locinfo>
          <arg exp='20' locinfo>  20  </arg>
        </macro_call>   )+10)
  </macro_body>
</macro_call>
```

上の XML 文書で，最初の要素（define タグ）は#define によるマクロ定義の情報を保持する．2 番目の要素（macro_call タグ）はマクロ呼出し F(20) の展開の様子を表している．具体的には「F(20) はマクロ定義#define F(x) ((x)+10) で展開され，仮引数 x は実引数 20 で置換された」ことが分かる．なお，ここでは簡単な例だけを示したが，多段のマクロ展開など，より複雑なマクロ展開にもこれで対応できる（ただし替え玉マクロ（2.3 節）と制限（5 節）に該当する場合を除く．）

2.3 追跡子：替え玉マクロと条件付きマクロ

2.2 節の追跡子が，#if や#ifdef などの条件式中に出現すると，その条件式を CPP は評価できなくなる．例えば，2.2 節の方法で，次の前処理指令に対して，

```
#define  VER  2
#if      VER >= 2
#endif
```

追跡子を埋め込むと以下になる．

```
#define  VER  @"macro_body ref='F1_1'"2@"/macro_body"
#if @"macro_call type='obj' name='VER' locinfo" VER @"/macro_call" >= 2
#endif     /* これはまずい例 */
```

#if の条件式中 VER >= 2 に追跡子が出現して不正な条件式となるため，VER >= 2 を評価できなくなる．しかし，追跡子を埋めなければ，条件式のマクロ展開の情報を得られない．そこで本論文では**替え玉マクロ**[6]を提案する．替え玉マクロのアイデアは「#if 評価用の本物のマクロ」に加えて,「マクロ展開解析用の替え玉のマクロ」を別に用意する点にある．替え玉のマクロは，本物のマクロと定義の内容の本質は同じだが，マクロ展開の解析という仕事も引き受ける点が異なる．

上記の例を替え玉マクロを使って追跡子を埋め込んだ例を次に示す．

```
@"define type='obj' id='F1_1' name='VER' macro_body='2' locinfo"
#define VER         2                                    /* 本物 */
#define VER_1528552 @"macro_body ref='F1_1'" 2 @"/macro_body" /* 替え玉 */
@"cond id='F1_2'"
```

[6] **替え玉**(double) は代役，影武者，ダミー等の意味で用いている．ラーメンのお代わりではない．

```
#if VER >= 2  /* ← ここは本物マクロを使う */
  @"cond_selected type='if' ref='F1_3' locinfo"
    @"exp orig='VER &gt;= 2'" /* #if の条件式の展開情報
      @"macro_call type='obj' name='VER' locinfo"
        @"bare"   VER         @"/bare"   /* 本物のマクロ呼出し */
        @"double" VER_1528552 @"/double" /* 替え玉のマクロ呼出し */
      @"/macro_call" >= 2
    @"/exp"
    /* ここに #if〜#else 間のコードに追跡子を埋めたトークン列が来る */
  @"/cond_selected"
#endif
  @"if   id='F1_3' exp='VER &gt;= 2' locinfo" /* #if 行に相当 */
@"/cond"
```

ここでは本物のマクロ名の後に_1528552 をつけたものを替え玉マクロ名としている．_1528552 は名前が衝突しなければ何でも構わない．上記の追跡子を埋めたコードを CPP 処理した結果を XML 形式に変換すると以下になる．

```
<define type='obj' id='F1_1' name='VER' macro_body='2' locinfo>
<cond id='F1_2'>
  <cond_selected type='if' ref='F1_3' locinfo>
    <exp orig='VER &gt;= 2'>
      <macro_call type='obj' name='VER' locinfo>
        <bare>2</bare>
        <double><macro_body ref='F1_1'>2</macro_body></double>
      </macro_call>
      >= 2
    </exp>
    /* ここに #if〜#else 間のコードに追跡子を埋めたコードを CPP 処理した結果が来る */
  </cond_selected>
  <if   id='F1_3' exp='VER &gt;= 2' locinfo>
</cond>
```

上の XML 文書から以下が分かる．

- #if の条件式 VER >= 2 には（替え玉ではなく）本物のマクロを使用したため，CPP は正しく条件付きコンパイルを処理した．
- 7 行目の double 要素から，条件式#if VER >= 2 の VER はマクロ定義#define VER 2 で展開されたことが分かる．これは替え玉マクロのおかげである．

この追跡子の手法では CPP の実行結果を解析するので，条件付きコンパイルが排除したコード（例えば上で#if の条件式が偽の場合の#if の中身）は解析できない．

2.4 追跡子：ファイルの取り込み

ファイルの取り込みにも同様に追跡子を定義した．詳細は [24] を参照されたい．

2.5 追跡子の計算量

追跡子の計算量は $O(s+d+m)$ である．ただし，s はファイル取り込み後のソースコードのサイズ，d は前処理指令の出現数，m は識別子の出現数である．

3 TBCppA の設計と実装

3.1 設計方針

- 既存の CPP 実装には一切修正を行わない．
- 実用性重視：なるべく適用範囲を広く高速に少ないメモリで動作すること．
  - なるべく C99 [16] と GCC 独自拡張（#include_next など）に対応する．
  - 極端にメモリを消費する技術（XML の DOM など）は使わない．
  - 文法クラスは LALR(1) を使う．LL は記述力，GLR は実行時間が問題．
- 低い開発・保守コスト：

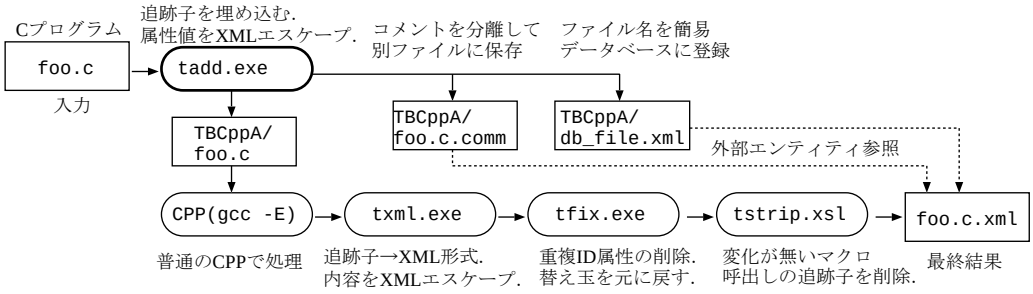


図 3　TBCppA の構成要素と処理プロセス

- なるべくコンパクトでシンプルな実装を目指す.
- 適材適所でいろいろな言語を使う.(実行時間に注意しつつ) XSLT も使う.
- チューニングはしない．例えばタグ名・属性名は無理に短くしたりしない.

3.2　TBCppA の主な構成要素と処理プロセス

2 節で述べた追跡子方式の CPP 解析器である TBCppA を設計・実装した．細部では工夫が必要だったが（3.5 節），2 節の追跡子方式を実装することができた.

TBCppA の主な構成要素と処理プロセスは次の通り（図 3 も参照）.

- `tadd.exe` が追跡子の埋め込みを行い，その結果を通常の CPP で処理し，続いて XML 形式に変換する．最後に `tstrip.xsl` が必要なのは，追跡子の埋め込み時にはどの識別子がマクロか判定できないため，`tadd.exe` は全ての識別子に追跡子を埋め込むからである．なお追跡子の XML 形式との相性のよさ（2.1 節）と利用の簡便さから XML を選択したが，XML である必然性は特に無い.
- C プログラム中のコメントの扱いを簡単にするため，字句解析時にコメントを位置情報とともに別ファイル (`foo.c.comm`) に保存することにした.
- マクロ定義などにファイル間でユニークな識別番号を割り振るために，パス名とファイル名を簡易データベース (`db_file.xml`) に登録する構造にした.

3.3　`tadd.exe`：追跡子を埋め込むツール

TBCppA で最も設計・実装が困難だったのは `tadd.exe` の構文解析器である．`tadd.exe` の構文解析器は，C 言語の構文解析器ではなく，CPP の記述（つまり前処理指令とマクロ呼出し）だけを構文解析する．1 節で述べたとおり，CPP の記述は非構造的である．前処理後のマクロ呼出しの実引数でカッコの対応が取れていればよく，前処理前にカッコが対応している必要は全く無い．例えば，図 1，図 4(a)，図 4(b) のコードは中カッコやカッコが対応していないが，CPP はこれらを処理できる．CPP とは異なり，`tadd.exe` は前処理指令を残したまま，これらのコードを構文解析する必要があるため，`tadd.exe` の構文解析は複雑になる．試行錯誤した結果，前処理字句列 (pp_tokens) を文脈に応じて 4 つに分けて文法を定義することで[7]，この問題をほぼ解決した．4 つの前処理字句とは次の通り.

- 前処理指令中の前処理字句：改行を含まない．`#`と`##`を特別扱いするため必要.
- 関数形式マクロの実引数の前処理字句：カッコとカンマを含まず，改行を含む.
- 関数形式マクロの実引数中の，カッコが対応する前処理字句：
  例えば `FOO(1,(2,3))` の `(2,3)`．カッコは必ず対応し，カンマと改行を含む.
- 上記以外の場所の前処理字句：改行を含まない.

この結果，`tadd.exe` は前処理指令の行単位構造とマクロ呼出しのカッコ構造の両方をかなりうまく扱える.

[7] この文脈依存性のため単純な文字列パターンマッチでは `tadd.exe` の構文解析器を実現できない.

例えば，図 1 と図 4(a) のコードを構文解析できる．また図 4(b) のコードはエラー回復を行って構文解析できるが，マクロ呼出し FOO に追跡子を埋められない．一般に tadd.exe は，マクロ呼出しの実引数の途中で前処理指令が出現した場合に追跡子を埋め込めない．この意味では tadd.exe は不完全だが精度への悪影響は小さい．例えば，予備実験（4.2 節）の実験対象 gzip-1.2.4（約 7300 行）と bash-3.1（約 7 万行）には（#include したものを含めて）全部で約 14 万個のマクロ呼出しがあるが，tadd.exe がエラーを報告したマクロ呼出しはわずかに 10 個だけだった．

```
#define FOO(x) (x)
#define PAREN (
PAREN )
FOO(10)
```

(a) tadd.exe は構文解析可能．

```
#ifdef WIN32
  FOO (10,
#else
  FOO (20,
#endif
  30)
```

(b) tadd.exe は構文解析可能だが，追跡子を埋め込めない．

図 4　カッコが対応せず構文解析が難しい例

3.4　TBCppA の動作環境と実装規模

TBCppA の現在の実装は次のノート PC 環境で開発・動作確認を行った．

Intel Pentium M 1.2GHz, 1GB RAM, Windows XP SP2, Cygwin 1.5.19, GCC 3.4.4, Bison 2.1, Flex 2.5.4, Libxml 2.6.22, Graphviz 2.8

GNU 開発環境が動作する他の UNIX ライク環境への移植は容易のはずである．

TBCppA の実装規模は約 3,000 行と小さい．開発には様々な言語（yacc, lex, C, csh, XSLT, Graphviz の dot）を用いている．開発時間は約一人月である．このうち，追跡子を埋め込むツール tadd.exe （図 3）の実装規模が最も大きく，tadd.y（ルール数は 85 個，行数は約 1,000 行）を含めて合計で約 2,100 行である．

3.5　The devil is in details.（細部の苦労）

追跡子のアイデア自体は画期的だが単純である：「マクロに追跡子を埋め込んで，普通の CPP にマクロ展開させて，外部から CPP の動作を観察する」．しかし，細部にはいくつもの苦労と工夫が必要であった．ここでは主なものを列挙する．

- エラー処理問題：3.3 節に加えて，エラー処理も単純ではない．いったん先読みしたトークンが前処理指令の場合，Flex の unput を用いて，そのトークンと新たな改行を入力ストリームに戻して，より正しいエラー処理を可能にした．
- 重複 ID 問題：同じファイルを何度も#include で取り込むと，静的にユニークに割り振った ID 属性値が重複して出現する．このため，XML 形式への変換時に重複する ID 属性を tfix.exe が削除するようにした．同じ ID 属性値を持つ別の追跡子の内容は同じなので，重複した ID 属性を削除しても問題は生じない．
- #と##演算子問題：例えば `#define STR(x) #x` に追跡子を埋め込んだ結果の `#define STR(x) #@"a"x@"/a"` は規則「#演算子の直後はマクロ仮引数が必ず来る」([16], 6.10.3.2) に違反してエラーとなる．これを避けるため，(情報量は落ちるが）#と##演算子のオペランドには追跡子を埋め込まないようにした．
- マクロ再定義問題：定義内容が同じならばマクロは再定義できる ([16], 6.10.3). しかし再定義したマクロの替え玉（2.3 節）は元の替え玉と ref 属性の値が異なるためエラーになる．これを避けるため，次のように#ifndef を用いた．

```
#define FOO         10  /* 本物 */
#define FOO_1528552 @"macro_body ref='F16_3'"10@"/macro_body" /* 替え玉 */
#define FOO         10  /* 本物の再定義 */
#ifndef FOO_1528552 /* 替え玉の再定義を避ける */
#define FOO_1528552 @"macro_body ref='F32_1'"10@"/macro_body" /* 替え玉 */
#endif
```

- マクロ名判定問題：追跡子埋め込み時には，例えば FOO () の FOO がオブジェクト形式マクロ，関数形式マクロ，マクロではない，のどれなのか判定できない．安全側にふって，この形式は常に関数形式マクロと tadd.exe に判断させ

表 1　システムファイルに追跡子を埋め込む処理時間とファイルサイズの変化

ファイル数	元の*.h のサイズ	処理時間	追跡子+ *.h のサイズ
480	14 万行 (5.0MB)	89 秒	73.8MB

表 2　TBCppA がマッピング情報を得る処理時間とファイルサイズの変化

	*.h のサイズ	*.c のサイズ	CPP 後の *.c のサイズ	*.h の処理時間	*.c の処理時間	追跡子+ *.c のサイズ	最終結果の *.c.{xml,lex} のサイズ
hello.c†	—	5 行 (76B)	5 行 (76B)	—	0.45 秒	2.8KB	0.8KB+4.4KB
hello2.c†	—	5 行 (63B)	941 行 (24KB)	—	1.4 秒	1.7KB	520KB+3KB
gzip-1.2.4	7.4KB	7300 行 (230KB)	1.5MB	0.6 秒	32 秒	3.1MB	11.9MB+7.0MB
bash-3.1‡	420KB	7 万行 (1.8MB)	10MB	13 秒	485 秒	33MB	191MB+61MB
gcc-4.1.1§	3.7MB	63 万行 (18MB)	102MB	72 秒	3602 秒	332MB	1.2GB+554MB

† hello.c は<stdio.h>を#include していない．hello2.c は<stdio.h>を#include している．
‡ bash-3.1 の lib ディレクトリ以下は解析の対象外にした．
§ ビルドは configure --enable-threads=win32 --with-cpu=i686 --with-arch=i686 --with-tune=i686 --enable-languages=c,c++．解析対象は gcc/{,cp/}*.c．

ることにした．なお tadd.exe の唯一のシフト還元衝突もこの問題に起因する．
- これ以外にも細かい問題が残っている．5 節でこれらの制限を説明する．

4　予備実験

4.1　tadd.exe の性能測定：システムヘッダファイルに追跡子を埋め込む

ここでは TBCppA の前処理にかかる時間が非常に短いことを述べる．

tadd.exe が追跡子を埋め込む処理時間とファイルサイズの増加量を測定した（表 1）．対象は主なシステムヘッダファイル（/usr/include/*.h など）である．結果は良好で，のべ 14 万行（ただし空行とコメントを含む）のヘッダファイルを約 53 秒と高速に処理できた．追跡子の埋め込みは我々が調べた範囲では正しく行われていた．ただし，3 つのヘッダファイルが GCC 独自の可変長引数マクロを使用していた．TBCppA は現在はこれを扱えないため，この部分をスキップして処理した．

4.2　TBCppA の性能測定：C プログラムからマッピング情報を得る

ここでは TBCppA が十分実用的な性能で動作することを述べる．

TBCppA が CPP のマッピング情報を得るのにかかる時間と，ファイルサイズの増加量を測定した（表 2）．対象は小中規模のソフトウェア (Hello world, gzip-1.2.4, bash-3.1) である．C プログラム foo.c を処理する前に，foo.c が#include するヘッダファイルに追跡子を予め埋めておく必要がある．このため表 2 では，ヘッダファイル (*.h) のサイズと処理時間を C プログラムとは別に表示した[8]．

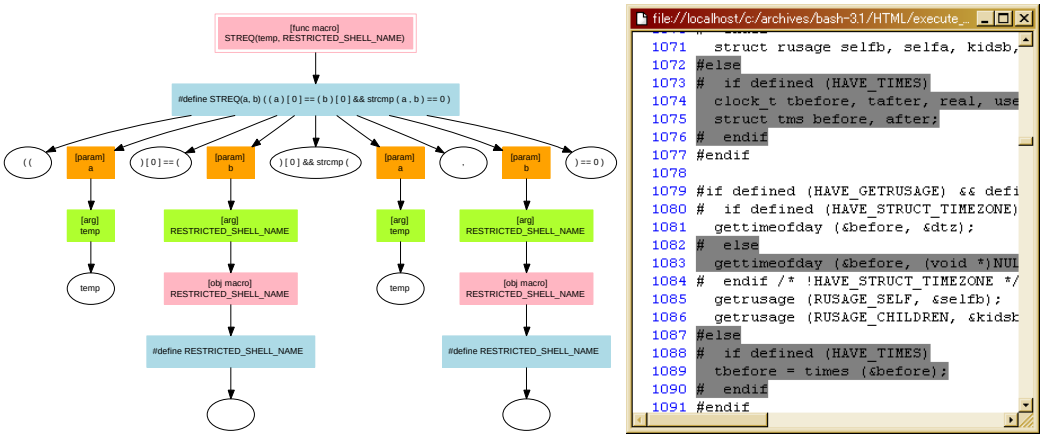
最終結果のサイズは CPP 処理後の C プログラムのサイズの 7〜13 倍であり十分許容範囲と考える．処理時間は例えば bash-3.1 は 867 秒 (15 分弱) だが，ファイル（翻訳単位）ごとにインクリメンタルに処理できること，同じ環境で bash-3.1 のビルドに約 6 分 40 秒かかる規模であること，の 2 点から十分実用的であると主張する．

4.3　マッピング情報の利用例

4.2 節で得たマッピング情報の有用性を確かめるために，次の簡単な応用ツールを作成した．小さな開発コストで有用なツールを構築できたため，この実験結果に限るが，TBCppA のマッピング情報は有用だったといえる．

- **tmacro.csh**：マクロ呼出しの展開を図示するツール．実行例は図 5(a)．
- **tifdef_src2html.exe**：条件付きコンパイルで捨てられたコード部分を灰色表

[8] このヘッダファイルのサイズと処理時間にはシステムヘッダファイルを含めてない．



(a) マクロ展開を視覚化例（bash-3.1 の `shell.c`，1104 行目）

(b) 条件付きコンパイルの未選択コードを灰色表示した例（bash-3.1 の `execute_cmd.c`）

図 5　TBCppA の応用ツールの実行例

示して，コード表示をするツール．実行例は図 5(b)．

どちらのツールも実装規模は非常に小さく．**tmacro.csh** は csh スクリプトで 65 行と XSLT で 136 行，**tifdef_src2html.exe** は XSLT で 22 行と C で 160 行しかない．開発時間も短く，それぞれを一人で半日以下で開発した．

5　制限

TBCppA には低い保守・移植コスト，高い適用性という大きな利点があるが，その反面，次の制限もある[9]（3.5 節も参照）．ただし ♠ をつけた問題は発生頻度が小さいのでほとんど問題にならない．紙面の都合で制限が生じる理由は説明しない[10]．

- ♠ `__LINE__` に依存した前処理指令の処理結果の変化．
  例えば `#if __LINE__ >= 100` は追跡子の追加で処理結果が変わりうる[11]．
- ♠ GCC 独自拡張の可変長引数マクロには未対応．
- ♠ 空白文字なしで継続行をまたぐトークンは，1 つのトークンと扱うのが正しいが，TBCppA は別々のトークンとして扱う．
- 右の `F1(a)` のマクロ呼出しでは，オブジェクト形式マクロの展開結果が関数形式マクロ名になる．この場合，`<bare>` 要素の内容は正しいが，マクロ展開の途中経過（`<double>` 要素の内容）が正しくなくなる（`F2(a)` でマクロ展開が止まる）．

```
#define F1    F2
#define F2(x) (x)
F1 (a)
```

- CPP に渡すオプション（例：GCC の `-I` や `-D`）の再現・設定が面倒．
- 条件付きコンパイルで排除されたコードは解析対象にならない．
- コンパイラオプションで指定したマクロ定義（例：GCC の `-DNDEBUG`）や CPP 実装の独自定義マクロ（例：`__GNUC__`）の `#define` 定義を現在は出力していない．
- 現在の替え玉方式は，可変長引数マクロの引数を正しく二重化できない．

6　関連研究

追跡子で CPP のマッピング情報を得る方法は，我々の知る限り [19] と本研究が初めてである．CPP エミュレータ方式と既存 CPP 改造方式 [1–3, 7, 9, 11–15, 25, 26]

[9] 我々が把握している制限はこれが全てであるが，新たな制限が見つかる可能性はある．

[10] 他の CPP 解析器では制限が不明確なものも多いので，制限の列挙だけでも大きな意味がある．

[11] TBCppA が追跡子に埋め込む位置情報（*locinfo*, 2.2 節）はこれとは関係なく，常に正確である．

は，追跡子方式にない問題を生じる（1 節）．[8] は CPP 問題が生じにくい AST ベースのマクロ言語 ASTEC を提案したが ASTEC への書き換えに人手が必要となる．DWARF2 デバッグ情報を使う手法 [22] [23] にはマクロ展開情報がない．

7 おわりに

本論文は CPP のマッピング情報を得る**追跡子方式** [19] の実装 TBCppA を与えた．予備評価は追跡子方式の有効性を示した．これから 5 節の制限を緩和しつつ，他システムに組み込んで，追跡子方式と TBCppA の有用性をさらに検証したい．

謝辞 愛知県立大学の山本晋一郎先生からは継続的に激励を，匿名査読者の方々からは有益なコメントを頂きました．文部省の科研費（研究課題番号:17500019）の助成を受けました．

参考文献

[1] G. J. Badros: PCp^3: A C Front End for Preprocessor Analysis and Transformation, Masters Thesis, 1997.
[2] G. J. Badros and D. Notkin: A framework for preprocessor-aware C source code analyses, Softw. Pract. & Exper., vol.30, Issue 8, pp. 907–924, 2000.
[3] A. Cox and C. Clarke: Relocating XML Elements from Preprocessed to Unprocessed Code, 10th Int. Workshop on Program Comprehension (IWPC'02), p.229, 2002.
[4] M. D. Ernst, G. J. Badros and D. Notkin: An Empirical Analysis of C Preprocessor Use, IEEE Trans. on Software Engineering, vol.28, no.12, pp.1146–1170, 2002.
[5] J. M. Frave: The CPP paradox, 9th European Workshop on Software Maintenance, DURHAM'95, 1995.
[6] A. Garrido and R. Johnson: Challenges of refactoring C programs, Proc. Int. Workshop on Principles of Software Evolution (IWPSE), pp.6–14, 2002.
[7] P. E. Livadas and D. T. Small: Understanding Code Containing Preprocessor Constructs, Proc. IEEE Third Workshop Program Comprehension, pp.89–97, 1994.
[8] B. McCloskey and E. Brewer: ASTEC: a new approach to refactoring C, Proc. 10th European Software Engineering Conf, pp.21–30, 2005.
[9] C. A. Mennie and C. L. A. Clarke: Giving Meaning to Macros, 12th IEEE Int. Workshop on Program Comprehension (IWPC'04), p.79, 2004.
[10] G. C. Murphy, D. Notkin, and E. S. C. Lan: An Empirical Study of Static Call Graph Extractors, 18th Int. Conf. on Software Engineering (ICSE), pp.90-99, 1996.
[11] L. Vidacs and A. Beszedes: Opening Up The C/C++ Preprocessor Black Box: Proc. 8th Sympo. on Programming Languages and Software Tools (SPLST'03), pp.45–57, 2003.
[12] L. Vidacs, A. Beszedes and R. Ferenc: Columbus Schema for C/C++ Preprocessing, 8th European Conf. on Software Maintenance and Reengineering (CSMR'04), pp.75–84, 2004.
[13] M. Vittek: Refactoring browser with preprocessor, 7th European Conf. on Software Maintenance and Reengineering (CSMR'03), 2003.
[14] D. Spinellis: Global analysis and transformations in preprocessed languages, IEEE Trans. on Software Engineering, 29(11):1019–1030, 2003.
[15] D. Spinellis: Browsing and refactoring program collections written in C, WP 2004-006, Eltrun Working Paper Series, 2004.
[16] Programming languages–C: ISO/IEC 9899:1999.
[17] comp.lang.c Frequently Asked Questions, http://c-faq.com/.
[18] 川島勇人，権藤克彦: XML を用いた ANSI C のための CASE ツールプラットフォーム，コンピュータソフトウェア [19], No.6, pp.21–34, 2002.
[19] 川島勇人，権藤克彦: 追跡子としての XML タグによる C 前処理問題への対応; 第 10 回ソフトウェア工学の基礎ワークショップ (FOSE2003), pp.105–108, 2003.
[20] 川島勇人，権藤克彦: ACML に基づくプログラム情報抽出システムの設計，コンピュータソフトウェア [21], No.5, pp.65–70, 2004.
[21] 権藤克彦，川島勇人: コンパクトな ANSI C インタプリタ XCI の設計と実装，電子情報通信学会論文誌， J86-D-I[3], pp.159-168, 2003.
[22] K. Gondow, T. Suzuki, H. Kawashima: Binary-Level Lightweight Data Integration to Develop Program Understanding Tools for Embedded Software in C, 11th Asia-Pacific Software Engineering Conference (APSEC), pp.336-345, 2004.
[23] 権藤克彦, 鈴木朝也, 川島勇人: C 言語用 CASE ツールへの DWARF2 デバッグ情報の応用, コンピュータソフトウェア [23], No.2, pp.175–198, 2006.
[24] K. Gondow: TBCppA Homepage, http://www.sde.cs.titech.ac.jp/~gondow/TBCppA/.
[25] 椴山嘉人，山本晋一郎：Sapid P-model ver. 1.0 仕様書，http://www.sapid.org/html2/P-model/P-model.pdf，2000.

[26] 三村 俊彦：プログラム理解支援のためのソースプログラムとプリプロセッサ出力との関連付け，名古屋大学工学部電気系学科卒業論文，1993.