情報処理学会論文誌 Vol.53 No.7 1832–1849 (July 2012)

属性付き字句系列に基づくソースコード書き換え支援環境

吉田 敦[1,a)] 蜂巣 吉成[1] 沢田 篤史[1] 張 漢明[1] 野呂 昌満[1]

受付日 2011年9月30日, 採録日 2012年4月2日

概要：ソースコードの編集作業において，複数箇所に点在する定型的な記述を修正する作業を自動化するために，属性付き字句系列に基づくソースコード書き換え支援環境 TEBA を提案する．TEBA は，ソースコードの断片を用いて記述したパターン記述に従ってソースコードを書き換えるパターン変換系と，その変換系を実現するうえで重要な基盤となる構文解析系を提供する．ソースコードの断片の構文解析は一般的に困難であるので，ソースコードを字句系列に変換したあと，段階的に構文に関する情報を字句の属性に加えていく．パターン変換系は，パターン変換記述とソースコードの双方を属性付き字句系列に変換したうえで，字句系列の書き換えとして実現する．また，実装した TEBA によって書き換えができることを事例を用いて示し，その評価を示す．

キーワード：プログラム解析，前処理前プログラム，プログラムパターン，プログラム変形，プログラム開発支援ツール

A Source Code Rewriting System based on Attributed Token Sequence

Atsushi Yoshida[1,a)] Yoshinari Hachisu[1] Atsushi Sawada[1]
Han-Myung Chang[1] Masami Noro[1]

Received: September 30, 2011, Accepted: April 2, 2012

Abstract: For automatic replacement of code fragments in a fixed form, we propose “TEBA”: a source code rewriting system based on attributed token sequence. TEBA consists of two subsystems. First one is a pattern-based transformation subsystem which rewrites source code according to a pattern specified as a pair of code fragments. The other one is a parser subsystem which becomes a basis for syntactic analysis of the source code. For parsing source codes, TEBA converts a source code to an attribute token sequence to which the parser subsystem gradually adds syntactic information. After converting a pattern and a target code into token sequences, the transformation subsystem rewrites the token sequence of the target code according to the token sequence of the pattern. We also show an implementation and evaluation of TEBA with examples.

Keywords: program analysis, un-preprocessed program, program pattern, program transformation, program development support tool

1. はじめに

プログラミングの過程は，ソースコードをエディタで編集していくことが主たる作業であり，編集作業の労力の削減のためには，できるだけ自動化することが必要である．リファクタリングや整形，入力支援など，作業を簡素化するための様々な編集支援機能が，汎用エディタや統合環境のプラグインとして公開されたり，ツールとして提供されたりしている．これらの編集支援機能は汎用性が高く，様々なソースコードの開発において活用される．

一方，編集作業全体を見ると，ツールやプラグインなどの支援は存在しないが，自動化できれば労力を減らせる編集作業がある．たとえば，ある関数定義のシグニチャの変更や，関数呼び出しの事前条件に関わる共通な記述も含めて１つの関数やマクロにまとめる作業では，各関数呼び出

[1] 南山大学情報理工学部
Faculty of Information Science and Engineering, Nanzan University, Seto, Aichi 489–0863, Japan
[a)] atsu@nanzan-u.ac.jp

しごとに同じ作業を何度も繰り返す．このような，分散して存在する定型的な記述に対して一律に同じ書き換えを適用する作業は，コードクローンの編集や，変数や定数の識別子変更にともなう参照箇所の修正，デバッグ文の追加・削除，意味が同じだが書き方が異なる処理の記述の統一化など，様々な場面で発生する．この作業を自動化できれば，作業効率が高まるとともに，手作業に起因する誤りの混入の回避や，編集対象箇所の見落としの防止など，ソースコードの品質の維持にもつながる．ただし，書き換え方法は，編集対象のソースコードの内容に依存するので，ツールとして実現しても再利用性が低い．しかし，手作業ですべて書き換える場合より，少ない労力でツールが作成できるのであれば，ツールを使い捨てにしたとしても，全体としての労力は削減できる．そこで，できるだけ少ない労力でツールを作成できる枠組みを考える．

定型的な記述に対する書き換えをツール化するための典型的な枠組みは文字列置換である．正規表現が使えれば，書き換え対象の細かな差異も吸収でき，sed や awk など，実績のある書き換えの支援ツールも存在する．しかし，ソースコードを対象とする書き換えでは，文や宣言などの構文要素を単位として扱うことが必要であり，正規表現を用いても記述できない．一般的には，ソースコードを抽象構文木に変換し，抽象構文木に対する操作を編集作業として記述し自動化する方法が用いられる [1], [2], [3]．ただし，プログラマは，構文要素を意識しつつも，ソースコードのテキストを見ながら書き換え方法を考えるので手間がかかる．文字列置換のように書き換え方法をテキストの置換として直接的に記述でき，かつ，その記述を抽象構文木に対する操作に自動的に変換して適用する環境が必要である．

そこで，本論文では，ソースコードの書き換え前と後の 2 つのテキストの断片から構成されるパターン変換記述を与えると，抽象構文木内で，書き換え前の断片に適合する部分木を書き換え後の断片の部分木に置き換えるパターン変換系を持つソースコード書き換え環境 TEBA [4] を提案する．パターン変換記述の書き換え前を表現するテキストの断片を「変換前パターン」と呼び，書き換え後を表現する断片を「変換後断片」と呼ぶ．どちらも基本的にはソースコードの実際の記述であるが，書き換え箇所ごとの差異を吸収するためにパターン変数も記述できる．パターン変数は，変換前パターンの中で任意の文などの構文要素を表現し，変換後断片で参照するとパターン変数に適合した構文要素への置換を意味する．パターン変換記述により，抽象構文木に対する基本操作である挿入，削除，移動，複製を実現できる．

パターン変換系を実現するうえで，構文解析の方法と，抽象構文木の表現方法が重要な鍵となる．パターン変換系では変換前パターンと変換後断片を構文解析して抽象構文木を作り，書き換え対象のソースコードの抽象構文木から

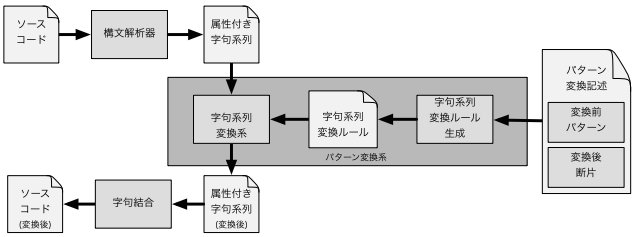

図 1 TEBA の処理の流れ
Fig. 1 The outline of TEBA.

変換前パターンの抽象構文木に適合する部分木を求め，その部分木を変換後断片の部分木に置き換える仕組みが必要である．しかし，ソースコードの断片の構文解析では，識別子の定義不足など，構文解析に必要な情報の欠落や，パターン変数などの存在により，プログラミング言語本来の構文規則と一致しない箇所が含まれ，一般的な構文解析の手法を適用できない．これを，抽象構文木と同等の情報を持つ字句系列を用いることで解決する．字句解析は，定義不足や構文規則との不一致の影響を受けない．字句解析で得られた字句系列に対して，粗い粒度の構文解析を適用し，その結果を字句に属性値として埋め込むことで，抽象構文木を構成する．このとき，文や宣言など，非終端節点に相当する構文要素は，その構文要素を構成する字句系列の前後に加える仮想的な字句で表現する．得られる抽象構文木は，文と宣言の区別や識別子の種別などの情報がなく，精度が低い状態になるが，字句の並び方などから判定できる情報に基づいて補正を行い，段階的に精度を高める．この方法では，精度が低い箇所が残るが，それを許容することで，全体として構文解析を達成する．パターン変換記述と対象ソースコードをそれぞれ属性付き字句系列に変換できれば，字句系列の変換系を用いてパターン変換系を実現できる．

TEBA では，C 言語のソースコードを書き換え対象とする．C 言語は，前処理系を用いることが必須であり，書き換えは前処理前の状態で適用する必要がある．一般的に，前処理前の状態は前処理命令が存在することや，ヘッダファイルが展開されないことで定義が不足するので，構文解析ができないが，この問題はパターン変換記述を解析するときと同じである．前処理命令に関して若干の前提を置く必要があるが，前処理命令に対する簡単な解析処理を加えるのみで，前処理前のソースコードの書き換えも実現できる．

TEBA におけるパターン変換系の構成と処理の流れを**図 1** に示す．書き換え対象のソースコードを，構文解析器で属性付き字句系列に変換し，パターン変換系でその字句系列を書き換える．書き換え後の字句系列を結合すると，書き換え後のソースコードが得られる．パターン変換系は，パターン変換記述から字句系列の変換ルールを生成し，書き換えを実行する．以降では，2 章でパターン変換

```
%before
if ((${v:ID_VAR} = ${cast:EXPR}malloc(${size:EXPR})) == NULL)
  ${stmt:STMT}$;
%after
${v} = ${cast}my_malloc(${size});
%end
```

図 2 パターン変換記述の例
Fig. 2 An example of source code pattern transformation description.

記述とその構文解析の方法について議論し，3 章で属性付き字句系列とソースコードの解析方法について説明する．4 章で構文解析器とパターン変換系の実装と評価を行い，5 章で，TEBA の技術的課題について考察する．

2. パターン変換記述

2.1 ソースコードに対する書き換えの表現

本論文の支援対象の作業は，コードクローンや，識別子の参照など，分散して存在する定型的な記述を一律に書き換える作業である．この作業は，ソースコードの記述内容に依存し，汎用性がないので，別のソースコードの開発には適用できない．専用のツールとして作り込んでも無駄になるので，プログラマが必要に応じて記述し，すぐに廃棄することが前提となる．作業効率の向上に貢献するには，文字列置換のように，短時間に直感的に分かりやすく記述できることが必要である．ただし，ソースコードは，構文規則によって制約された記述であり，構文要素を単位として書き換える必要がある．文字列置換では，構文要素を単位として扱えないので，識別子の変更など，きわめて単純な書き換えしか実現できない．

構文要素を単位とする書き換えの実現方法としては，抽象構文木を操作する方法が一般的である．抽象構文木に対する操作を記述することで，自動化も可能になる．しかし，プログラマは，構文規則を意識しつつも，ソースコードをテキストとして編集している．すなわち，エディタを用いて，字句の挿入や削除などを行っており，抽象構文木の操作とはとらえていない．よって，抽象構文木に対する操作を記述しようとすると，テキストと抽象構文木との対応関係をつねに把握しながら，テキストとしての編集操作を抽象構文木の操作に写像する必要があり，短時間に直感的に分かりやすく記述することはできない．そこで，本論文では，書き換え前と後の 2 つのソースコードの断片をそれぞれ変換前パターンと変換後断片として記述したパターン変換記述に従い，抽象構文木の操作として書き換えを適用するパターン変換系を提案する．プログラマは，編集中のテキストから書き換え対象の断片をコピーすることで書き換え作業を記述でき，また，抽象構文木などの他の表現に写像する必要がない．

ソースコード内の書き換えたい箇所がすべて完全に一致することは期待できず，パターン変換記述に書き換え前後の断片をそのまま記述しただけでは，意図どおりには書き換えられない．そこで，差異のある要素を可変部分として抽象化する仕組みとしてパターン変数を導入する．パターン変数は，書き換え前の指定された構文要素に適合し，書き換え後は参照された位置に適合した構文要素をあてはめる．なお，差異を広く吸収するためには，文字列の正規表現のように，繰返しや選択などのメタな記述も必要ではあるが，複数のパターン変換記述を作ることでも対処が可能であるので，本論文では，パターン変換系に対する基本的な基盤を確立することを優先し，最低限必要な要素として，パターン変数のみを用いる．

パターン変換記述に基づいて，抽象構文木を操作するためには，抽象構文木に対する基本操作を実現できることが必要条件となる．基本操作は，部分木の挿入，削除，移動，複製の 4 つで構成される．挿入は，変換前パターンに，挿入したい構文要素に該当する字句を追加することで実現できる．削除は，変換後断片を記述するときに変換前パターンから該当する要素を除いて記述する．移動と複製は，変換前パターンの中の対象の構文要素の箇所をパターン変数に置き換え，変換後断片の移動先または複製を置きたい箇所にパターン変数の参照を記述する．移動と複製は，パターン変数の参照の数が 1 つか複数かの違いのみである．

2.2 パターン変換記述の記法

パターン変換記述は，**図 2** のように，`%before` の直後に変換前パターンを，`%after` の直後に変換後断片を記述する．この例は，**図 3** のように，標準関数 malloc() の呼び出しとそれを含む if 文を，独自の関数 my_malloc() へ置き換えるパターン変換を表す．

パターン変数は，可変部分となる箇所に記述する．図 2 では，パターン変数を用いて，malloc() の返り値を保持する変数を`${v:ID_VAR}`や`${v}`で，キャストの型を`${cast:EXPR}`や`${cast}`と表現している．パターン変数は，変換前パターンでは`${名前:型}`の形式で，変換後断片では`${名前}`と記述する．型は適合すべき構文要素を表し，後述する字句の種別か，ユーザが定義する字句パターンの名称を用いる．なお，この例には記述されていないが，型名の後ろに*をつけると，その構文要素の繰返しを表す．

パターン変数は，パターン変換記述を記述する立場から考えれば，任意の位置に記述できることが理想である．しかし，パターン変換記述の構文解析器を構築する立場では，制約する必要がある．C 言語の場合，関数定義や制御文を

```
                                        //* 書換え前 *//
void test(void)
{
    Obj *p;  /* Obj is an undefined type. */
    if ((p = (Obj *)malloc(sizeof(Obj) * SIZE)) == NULL) {
        fprintf(stderr, "no memory\n");
        exit(1);
    }
    proc(p);
    free(p);
}
```

```
                                        //* 書換え後 *//
void test(void)
{
    Obj *p;  /* Obj is an undefined type. */
    p =  (Obj *)my_malloc(sizeof(Obj) * SIZE);
    proc(p);
    free(p);
}
```

図 3 ソースコードの書き換え例
Fig. 3 An example of source code transformation.

記述するときに，中括弧で囲ってブロックを表現しており，また，可読性の観点から，ブロックを基準としたインデントを用いるプログラミングスタイルが広く使われている．これは，プログラマが，ソースコードを記述するときに，文や宣言のレベルまでは抽象構文木の構造を強く意識していることを意味する．一方で，文や宣言を構成する式や宣言子などの構造に関しては，我々の経験上，括弧やカンマによる区切りは意識するが，右線形や左線形など，木構造の詳細は意識していないことが多い．そこで，本論文では，関数定義や宣言，文のレベルまでは構文規則に従い，それ以下の詳細なレベルでは括弧の整合がとれていれば，どこにパターン変数を用いてもよいものとする．なお，図 2 で用いている “$;” は，文の終わりを表す特殊な記号である．文や宣言を表すパターン変数を用いたときに，構文規則と合わない箇所が生じるので，構文的に完結させる目的で用いる．

パターン変数を用いれば差異がある箇所を抽象化できるが，空白やコメントは様々な字句の間に出現しうるので，すべての可能性を考えて記述することは現実的ではない．そこで，適合するときは，空白やコメントの有無は無視し，書き換え後については，パターン変換記述に書かれた空白やコメントがそのまま残るものとする．パターン変数が適合した範囲に，空白やコメントが含まれる場合にはそれらも書き換え後に残る．なお，書き換え前の空白やコメントを適切に残すことは難しいので，その方法は本論文とは別の議論とする．

2.3 パターン変換系における構文解析

パターン変換系の実現には，パターン変換記述に記述されるソースコードの断片から抽象構文木が生成できることが重要である．抽象構文木が生成できれば，編集対象のソースコードの抽象構文木から適合する部分木を探し，部分木を置き換えればよい．構文解析器を作るうえで，パターン変換記述が含む断片は前処理前のソースコードであり，かつ，パターン変数を含むので，次の 2 つの点が問題となる．

- 構文規則を満たさない部分を含むこと
- 識別子の定義の情報が不足していること

構文を満たさない部分は，前処理命令やパターン変数が含まれることや，条件コンパイルで切り替わる断片が存在することで発生する．識別子の定義の不足は，ソースコードの断片であることや，前処理のヘッダファイルが展開しないことで発生する．C 言語の場合，未定義の識別子を含むと，文と宣言の区別がつかず，一般的には構文解析ができない．たとえば，“a (b);” という記述は，a が型なら宣言であり，そうでなければ関数呼び出しを表す文であるので [5]，構文要素を確定できない．

これらの問題により，一般的な方法では，抽象構文木を構成することができない．しかし，記述の大部分は構文規則に従っており，また，識別子の定義がなくても解釈できる部分も存在する．そこで，解析が困難な箇所は精度の低い状態になることを許す抽象構文木を用いる．ここで，精度が低いとは，誤りを含む状態や，木構造を構成できない状態，字句の種別が不明な状態を指す．精度の低い状態を許容した構文解析を実現するために，本論文では，字句解析で得られる字句系列に，精度の低い解析を適用したあと，精度を向上させることができる部分を徐々に解析していく方法を用いる．我々の経験では，ソースコードの断片のみが与えられた場合であっても，プログラマは構文規則の持つ制約から類推して構文を把握していることが多い．そのようなヒューリスティックな規則を用いれば，部分的に解析して精度を向上させることは可能である．

まず，前節で述べたとおり，パターン変換記述は，文・宣言のレベルまでは構文規則を満たすことを前提とするので，そのレベルまでの構文解析を行う．ただし，文と宣言は区別できないことがあるので，その場合には文として扱う．また，記号表も構築できないので，字句の種別も区別しない．次に，ヒューリスティックな規則を用いながら，

文や宣言の細部を解析する．たとえば，“a b;” は識別子の定義がない場合でも，構文規則の制約から a を型，b を変数とする宣言であることが確定できる．また，“a()” のように識別子の後ろに括弧の組が並べば関数呼び出しであると解析する．このような解析を繰り返し適用していくことで精度の向上を達成できる．

なお，前処理前のソースコードを正確に解析するためには，コンパイル条件によって異なる記述をすべて解析する必要がある．また，マクロの定義によっては見かけ上，構文規則を逸脱した記述になることもあり，解析が難しい．我々の経験上，プログラマは前処理命令を含むソースコードを，前処理を命令を無視したときに構文規則を満たすように記述することが多いことから，近似的に前処理命令を空白と見なして解析することとし，前処理に関わる解析の精度の向上方法については，本論文では対象としない．この近似方法による影響については，4 章で述べる．

3. 属性付き字句系列

3.1 属性付き字句系列とは

TEBA の構文解析では，字句系列に精度の低い解析を適用してから，徐々に精度を向上させる．精度の向上は，字句の並びから推測をするので，その多くが字句系列の書き換えになる．そこで，抽象構文木を字句系列のまま表現する方法を採用する．パターン変換系も字句系列の書き換えに帰着できるので，全体として統一的な仕組みで実現できる．

TEBA で扱う字句系列は，字句をソースコード中での出現順に並べたものである．空白や改行，コメントなど，一般的な抽象構文木の構成では破棄される文字列も字句に含み，すべての字句を並び順のまま結合すると元のソースコードが復元される．さらに，ヌル文字（長さ 0 の文字列）も字句として扱い，「仮想字句」と呼ぶ．仮想字句は，主に，非終端の構文要素の開始と終了の位置に配置し，構文要素の区切の表現に用いる．また，これにより，抽象構文木の親子関係を表現する．

すべての字句は種別属性を持ち，その値は，字句解析の字句定義で与えられる名前である．ただし，仮想字句は，構文要素に対応する種別名を持つ．括弧や仮想字句は，対応関係属性を持つ．右括弧と左括弧の組や，構文要素の開始と終了の位置を表す仮想字句の組に対して，同一の識別記号を割り振ることで，それらの字句の対応関係を表現する．なお，本論文ではこの 2 つの属性を扱うが，パターン変換系を実現するうえで必要となる付加的な情報がある場合には属性を加えてもよく，任意個の属性を持つことを前提とする．

3.2 属性付き字句系列の表現形式

字句系列は，1 行が 1 つの字句であるテキスト形式とし，

```
#include <stdio.h>

#define NUM 100

int main(void)
{
    int i, t = 0;
    for (i = 0; i < NUM; i++) {
        t += i;
    }
    printf("t = %d\n", t);
    return 0;
}
```

図 4 C 言語のソースコードの例
Fig. 4 An example of C code.

各字句の表現形式は，属性値を文字列の正規表現で取り出すことを前提に，次のとおりとした．

種別 (属性値)* ’<’ テキスト ’>’

先頭には**種別**を記述し，その後に任意個の**属性値**を並べ，最後に字句の**テキスト**を “<” と “>” で囲って記述する．種別は，各字句に必須であることや可読性を考えて先頭に配置した．テキスト内で，改行文字，タブおよびバックスラッシュは，“\n” などバックスラッシュを用いたエスケープ表現で記述する．

図 4 を解析して得られる字句系列を**図 5** に示す．この字句系列は TEBA による解析の最終結果であり，種別の詳細化や仮想字句の追加も行われている．先頭と最後尾の UNIT_BEGIN と UNIT_END は，字句系列を変換するときに先頭または最後尾の字句の処理を例外的に扱うことを避けるために，番兵として入れている仮想字句である．また，BEGIN_や END_で始まる種別の字句は構文要素の範囲を表す仮想字句である．

行単位のテキスト形式とした理由は，UNIX 系の既存のツールとの連携を想定したことにある．たとえば，図 2 のパターン変換記述の例の場合，実際に変換するときは，変換漏れがないか調べる必要があるが，関数名の malloc が残っているかどうかは，字句系列に対して grep で調べることができる．また，専用の API を用意せず，各字句の属性値の取り出しに正規表現を用いる理由は，字句系列の書き換えツールを様々なプログラミング言語で実装できることにある．なお，これらの判断は，書き換えを目的とした抽象構文木のテキスト形式の表現として StreamCode [6] を提案したときの経験にも基づいている．

3.3 段階的詳細化に基づく属性付き字句系列への変換

TEBA では，精度の低い解析を適用したあと，徐々に精度を向上させていくが，具体的には，以下の解析を**図 6** のように組み合わせる．

- 字句解析
- 前処理命令解析
- 括弧対応関係解析
- 粗粒度構文解析
- 仮想字句対応関係解析
- カンマ対応関係補正

```
UNIT_BEGIN <>
BEGIN_DIRECTIVE #E0001 <>
PRE <#include>
SPACE_B < >
PRE_H <<stdio.h>>
END_DIRECTIVE #E0001 <>
SPACE_NL <\n>
SPACE_NL <\n>
BEGIN_DIRECTIVE #E0002 <>
PRE <#define>
SPACE_B < >
ID_MACRO <NUM>
SPACE_B < >
BEGIN_MACRO_BODY #E0003 <>
LITERAL <100>
END_MACRO_BODY #E0003 <>
END_DIRECTIVE #E0002 <>
SPACE_NL <\n>
SPACE_NL <\n>
BEGIN_FUNC #E0004 <>
ID_TYPE <int>
SPACE_B < >
ID_FUNC <main>
PAR_L #B0001 <(>
ID_TYPE <void>
PAR_R #B0001 <)>
SPACE_NL <\n>
BEGIN_STMT #E0011 <>
CUR_L #B0002 <{>
SPACE_NL <\n>
SPACE_B <    >
BEGIN_DECL #E0005 <>
ID_TYPE <int>
SPACE_B < >
ID_VAR <i>
CMA #E0005 <,>
SPACE_B < >
ID_VAR <t>
SPACE_B < >
OPE <=>
SPACE_B < >
LITERAL <0>
SEMI <;>
END_DECL #E0005 <>
SPACE_NL <\n>
SPACE_B <    >
BEGIN_STMT #E0006 <>
CTRL_BE <for>
SPACE_B < >
PAR_L #B0003 <(>
ID_VAR <i>
SPACE_B < >
OPE <=>
SPACE_B < >
LITERAL <0>
SEMI <;>
SPACE_B < >
ID_VAR <i>
SPACE_B < >
OPE <<>
SPACE_B < >
ID_VAR <NUM>
SEMI <;>
SPACE_B < >
ID_VAR <i>
OPE <++>
PAR_R #B0003 <)>
SPACE_B < >
BEGIN_STMT #E0007 <>
CUR_L #B0004 <{>
SPACE_NL <\n>
SPACE_B <\t>
BEGIN_STMT #E0008 <>
ID_VAR <t>
SPACE_B < >
OPE <+=>
SPACE_B < >
ID_VAR <i>
SEMI <;>
END_STMT #E0008 <>
SPACE_NL <\n>
SPACE_B <    >
CUR_R #B0004 <}>
END_STMT #E0007 <>
END_STMT #E0006 <>
SPACE_NL <\n>
SPACE_B <    >
BEGIN_STMT #E0009 <>
ID_FUNC <printf>
PAR_L #B0005 <(>
STR <"t = %d\\n">
CMA #B0005 <,>
SPACE_B < >
ID_VAR <t>
PAR_R #B0005 <)>
SEMI <;>
END_STMT #E0009 <>
SPACE_NL <\n>
SPACE_B <    >
BEGIN_STMT #E0010 <>
RESERVE <return>
SPACE_B < >
LITERAL <0>
SEMI <;>
END_STMT #E0010 <>
SPACE_NL <\n>
CUR_R #B0002 <}>
END_STMT #E0011 <>
END_FUNC #E0004 <>
SPACE_NL <\n>
UNIT_END <>
```

図 5 図 4 を解析して得られる属性付き字句系列
Fig. 5 The attributed token sequences generated from Fig. 4.

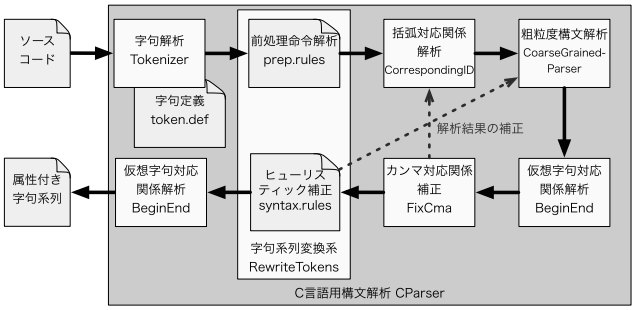

図 6 解析処理の流れ
Fig. 6 The outline of the analysis.

- ヒューリスティック補正

字句解析以外の解析は，字句系列を入力として受け取り，解析した結果を埋め込んだ字句系列を出力として返すフィルタ型の構成をしている．図 6 では，仮想字句対応関係解析が 2 回使用されているが，1 回目はカンマ対応関係補正およびヒューリスティック補正で対応関係を使用するためであり，2 回目はヒューリスティック補正で仮想字句が追加される場合があるためである．また，カンマ対応関係補正およびヒューリスティック補正から出ている点線矢印は，どの解析結果の補正をしているかを示している．各解析の内容について，以降の項で説明する．

3.3.1 字句解析

字句解析は，字句定義に基づいて字句を切り出し，種別を付与して属性付き字句系列を生成する．TEBA では，字句の種別を 28 個定義しており，その一部を**図 7** に示す．字句定義は，字句の種別とその字句を切り出すための正規表現で構成する．TEBA は実装言語に Perl を用いており，その拡張正規表現をそのまま利用できる．字句解析器は，内部で字句定義から解析用メソッドを生成する．字句定義内は並び順で上にあるものが優先され，コメントや演算子など競合する可能性がある字句については順序を考慮して定義する．

字句のうち識別子については，宣言などの種別を特定する情報があるとは限らないことから種別を IDENT に統一して扱う．予約語 “struct” の直後の識別子をタグと判別できるように，出現位置の前後の字句の並びからより正確な種別を特定できる場合もあるが，正確な種別への置換は，後段のヒューリスティック補正で実施する．

制御文を構成する予約語の種別は “CTRL_” とその直後に続く数文字の記号で構成する．記号は，式やブロックの有無を表し，後段の粗粒度構文解析がその記号に従って処理することで，実装が簡潔になる．改行や空白文字も同様の理由で種別を命名している．

3.3.2 前処理命令解析

C 言語の構文は，テキストの行の概念とは独立に字句の並びに基づいている．しかし，前処理命令は行指向で記述され，基本的に任意の行に記述できる．前処理前のソースコードをそのまま解析するためには，任意の行に出現しう

```
CTRL_BE   /(?:for|while|switch)\b/
CTRL_BEI  /if\b/
CTRL_B    /(?:else)\b/
CTRL_BD   /(?:do)\b/
PAR_L     /\(/
PAR_R     /\)/
SPACE_NL  /\n/
SPACE_CNL /\\\n/
SPACE_B   /\f|[ \t]+/
RESERVE   /(?:typedef|struct|union|enum|case|default|return|continue|goto|break)\b/
CMT       (/\*.*?\*/|//.*?(?=\n))
STR       /"(?:(?<!\\)[^\\]|\\\\[^\\"]|\\.)*?"/
CHAR      /'(?:(?<!\\)[^\\]|\\.)*?'/
IDENT     /\w+/
```

図 7 C 言語の字句定義の例
Fig. 7 Some token definitions for C.

```
# 0 個以上の空白・コメントの連続を表す字句
@SP  => "(?:(?:SPACE_B|CMT)\s+<.*>\n)*?"
# 前処理命令の行頭になる字句
@LINETOP => "(?:(?:UNIT_BEGIN|SPACE_NL)\s.*?\n)"
# 前処理命令内に出現しうる字句の 0 個以上の並び
@ANYTOKENS => "(?:(?:[^SBE]\w+|STR|SEMI|SPACE_(?:CNL|B)|SIZEOF)\s.*?\n)*?"
# マクロの仮引数部分
@ARGS => "(?:(?:IDENT|CMA|SPACE_B)\s.*?\n)*?"

# 前処理命令の前後に仮想字句 (BEGIN|END)_DIRECTIVE を挿入
{ $top:LINETOP $sp:SP $pre:PRE $any:ANYTOKENS $nl:SPACE_NL}
=> {$top $sp '':BEGIN_DIRECTIVE $pre $any '':END_DIRECTIVE $nl}

# 引数付きマクロの本体に仮想字句 (BEGIN|END)_MACRO_BODY を追加
{ $begin:BEGIN_DIRECTIVE $pre:PRE $sp1:SP $id:IDENT $p1:'(' $args:ARGS $p2:')'
  $sp2:SPACE_B $body:ANYTOKENS $end:END_DIRECTIVE}
=> { $begin $pre $sp1 $id:ID_MACRO $p1 $args $p2 $sp2
     '':BEGIN_MACRO_BODY $body '':END_MACRO_BODY $end }
```

図 8 前処理命令の解析定義
Fig. 8 Analysis rules for preprocess directive.

る前処理命令を考慮する必要がある．そこで，まず前処理命令の識別だけを独立に実行し，後の解析では前処理命令を空白の一種として扱うことで対処する．

前処理命令の解析では，字句系列変換系を利用して，命令を構成する字句系列の両端に仮想字句を追加する．4 つの字句系列の変換ルールを定義しており，その一部を**図 8** に示す．後半の中括弧で始まる記述が字句系列の変換ルールであり，それぞれ前処理命令の前後と，前処理命令のうち引数付きマクロの本体の前後への仮想字句の挿入を表す．各ルールは “=>” の前の字句系列パターンと直後の置換字句系列で構成され，前者に適合した字句系列が後者に置き換わる．

字句系列パターンの各字句は “**$変数名:種別**” または “**$変数名:’字句’**” と表現し，それぞれ**種別**または**字句**に適合する字句を表す．実際の処理系内では，これらは “(**種別** \s+(?:\s+#\w+)?\s+<.*>\n)” や “(\w+(?:\s+#\w+)?\s+<**字句**>\n)” などの文字列の正規表現に変換している．なお，任意の文字を表すメタ文字 “.” には，改行を含めない設定で適合させている．一方，置換字句系列には，置換後の字句の並びを変数名を用いて記述する．変数は，字句系列パターンでその変数に適合した属性付き字句を表現する．ただし，“**$変数名:種別**” と記述した場合は，変数に適合した字句の種別を変更したものを表す．また，新たに字句を挿入したい場合は，その字句と種別を指定して “**’字句’:種別**” と記述する．空白やコメントの 0 個以上の連続など字句パターンを使用したいときには，特殊な種別として定義できる．図 8 の前半の “@” で始まる行がその定義であり，それぞれ “=>” の右側には字句系列を文字列と見なした正規表現を記述する．

字句系列変換系では，各ルールを字句系列全体に対して定義順に適用し，変換する．字句系列パターンに適合する箇所が複数存在するときは，それぞれ先頭から書き換える．すべてのルールを適用したあと，再度，先頭のルールからの適用を繰り返す．最終的に，どのルールを適用しても変換されなければ終了する．なお，前処理命令の解析では，ルールを 1 回適用すればすべての仮想字句が挿入され，2 回目の適用では書き換わることなく終了する．

3.3.3 括弧対応関係解析

対応関係にある括弧の組とその括弧の中の区切りとして使われるカンマに対して，その対応関係を明示するために，同一の識別記号を属性値として与える．括弧の対応関係については，後段の粗粒度構文解析で関数引数の範囲の特定などに使用する．

カンマに識別記号を与えることで，関数呼び出しの実引数に別の関数呼び出しを含む入れ子の状態でも，カンマがどの括弧内の区切りかを区別できる．図 5 において “#B0001” のように括弧やカンマが持つ “#B” で始まる属性値が識別記号である．ただし，この時点では宣言の区切りであるカンマが，その宣言を直接含む複合ブロックの中括弧の区切りとして識別されるという誤りが生じる．そこで，宣言の範囲が確定したあとで，補正を行い，正しい状態にする．なお，現状の実装では，カンマの対応関係の情報は，

```
# (r1) tag への分類
{ $struct:'struct' $sp:SP $tag:IDENT } => { $struct $sp $tag:ID_TAG }

# (r2) 変数への分類 (注: $type は型ではない可能性がある)
{ $type:IDENT $sp:SP $var:IDENT } => { $type $sp $var:ID_VAR }

# (r3) 文を宣言に修正
{ $decl_begin#1:BEGIN_STMT $type:ID_TYPE $any:ANY $decl_end#1:END_STMT }
=> { $decl_begin:BEGIN_DECL $type $any $decl_end:END_DECL }
```

図 9 ヒューリスティック補正のルールの一部
Fig. 9 Example rules of heuristic correction.

補正が行われるまで使用されることがないので，補正のときにすべて付けるようにすることもできる．ただし，式に関する解析の精度を上げるときや，マクロに関する解析を追加する場合に必要になることが想定されるので，拡張性を考慮してここで適用している．

3.3.4 粗粒度構文解析

各構文要素を構成する字句の部分列の前後に仮想字句を挿入することで，構文要素の範囲を表現する．式のレベルまでは解析しないので，以下の構文要素を対象とする．

- 変数宣言，関数プロトタイプ宣言，関数定義
- ユーザ型定義（typedef）
- 構造体，共用体，列挙体
- 文，ラベル

パターン変換記述には，文が単体で出現することがあるので，本来は文が存在しない大域的なレベルに文が出現することを許容する．また，文全体に適合するパターン変数を用いる場合，文の区切り位置を表す字句が存在せず，解析できないので，代わりに文末を表す字句として種別 END_STMT_MARK の仮想字句を用いる．この仮想字句を「文末仮想字句」と呼び，図 2 では “$;” が該当する．

文レベルまでの構文解析は，再帰下降構文解析などの既存の技術を応用すれば実現できる．文と宣言は区別できない場合があるが，typedef によるユーザ型定義のように宣言が明確であるものを除き，文と宣言は区別せず，後段のヒューリスティック補正で区別を行う．

3.3.5 仮想字句対応関係解析

仮想字句は，主に構文要素の開始と終了の位置を表現し，括弧と同様に，その対応関係に関する情報が必要である．そこで，対応する仮想字句には同一の識別記号を属性値として追加する．図 5 で，“#E” で始まる属性値が対応関係を表す識別記号である．

3.3.6 カンマ対応関係補正

前述の括弧対応関係解析は，図 4 の局所変数の宣言に含まれるカンマのように，括弧内の区切りではないカンマを，その外側の複合文を構成する中括弧に対応付けており，これは誤りである．カンマ対応関係補正では，宣言や文を区切るカンマが，その宣言や文の識別記号を持つように修正する．

3.3.7 ヒューリスティック補正

ヒューリスティック補正は，前段階までの解析では不足している情報の追加や，不正確な解析結果の修正などの補正を行う．補正は，前処理命令の解析と同様に，字句系列の変換ルールとして記述し，字句系列変換系で処理する．各ルールは，ソースコードを読むときの経験的な知識に基づき，以下の 3 種類で構成した．

(1) 識別子の分類の詳細化（ルール数 41）
(2) 文から宣言への修正（ルール数 5）
(3) その他（ルール数 1）

(1) は，種別が IDENT である識別子について，前後に出現する字句の並びから，型（ID_TYPE），タグ（ID_TAG），メンバ（ID_MEMBER），変数（ID_VAR），関数（ID_FUNC），ラベル（ID_LABEL）に可能な限り分類する．(2) は，型識別子などの存在に基づいて，文として解析した宣言を補正する．(3) は，関数定義の本体の複合ブロックを文にするルールで，ルールの共通化を目的とする．

補正用の変換ルールの一部を**図 9** に示す．変換ルールは，「2 つの識別子が並んでいれば変数宣言である」など，経験的な知見に基づいて定義している．ルール (r1) は，予約語 struct の直後にはタグのみが記述されるので，識別子の種別を ID_TAG に変更する．(r2) は，識別子が連続して並ぶ場合には，宣言の型と変数の組合せであるので，2 つ目の識別子の種別を ID_VAR に変更する．なお，1 つ目の識別子はタグである可能性もあるので，種別を変更しない．(r3) は，型で始まる文は，本来，宣言であることから，仮想字句を文から宣言に変更する．パターン変数の名前の直後の記号 “#1” は対応関係を表現し，同じ記号を持つパターン変数は対応関係にある字句を参照することを意味する．これにより，非終端の構文要素の字句の範囲を正確に特定できる．

これらの補正用の変換ルールは字句系列変換系により繰り返し適用される．識別子の種別を補正するルールを適用すると文を宣言に変更するルールが適用されるなど，ルール間には依存関係があるので，変換が終わるまでに複数回の適用が必要である．ルールの書き方によっては停止しない可能性があるので，字句の種別の変更をできるだけ 1 方向にするなど，停止性が得られるように構成している．たとえば，識別子の種別は，字句解析の段階では IDENT とし，(r2) のように文脈に基づいて分類しているが，元の IDENT に戻すルールは定義していない．なお，停止性が得られるようにルールを定義することは容易ではなく，実際には，

```
{ $t01#E0001:BEGIN_STMT $t02:'if' $t03:SP $t04#B0001:'('
    $t05#B0002:'(' $v:ID_VAR $t06:SP $t07:'='
      $t08:SP $cast:EXPR $t09:'malloc' $t10#B0003:'(' $size:EXPR $t11#B0003:')'
    $t12#B0002:')' $t13:SP $t14:'==' $t15:SP $t16:'NULL'
  $t17#B0001:')' $t18:SP
  $t19#E0002:BEGIN_STMT $stmt:STMT $t20#E0002:END_STMT $t21#E0001:END_STMT
} => {
   '':BEGIN_STMT $v ' ':SPACE_B '=':OPE ' ':SPACE_B
    $cast 'my_malloc':ID_FUNC '(':PAR_L $size ')':PAR_R ';':SEMI  '':END_STMT }
```

図 11 パターン変換記述から生成される変換ルール

Fig. 11 A transformation rule generated from a source code transformation description.

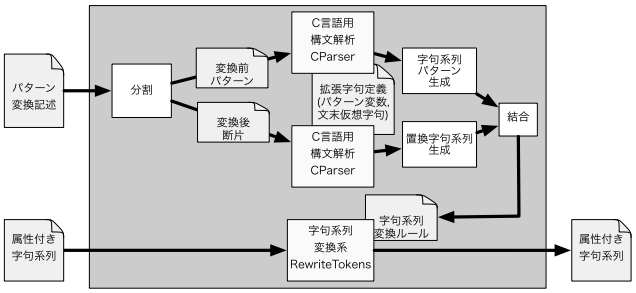

図 10 パターン変換系の概要

Fig. 10 The outline of source code pattern transformation system.

相互に逆の変換を含むルールの組が存在する．ただし，特定の順序で適用すれば正しく補正されるので，ルールの記述を分離し，段階的に適用することで停止性を確保する．

3.4 パターン変換系

パターン変換系は，パターン変換記述を字句系列の変換ルールに変換したうえで字句系列の書き換えを行う．字句系列の書き換えには，前処理命令解析やヒューリスティック補正で用いている字句系列変換系を用いる．全体の構成と処理の流れを**図 10** に示す．変換系は図 2 のようなパターン変換記述に対し，変換前パターンと変換後断片をそれぞれ属性付き字句系列に変換したあと，字句系列パターンと置換字句系列に変換し，字句系列の変換ルールを生成する．字句系列変換系は，属性付き字句系列を読み込み，字句系列の変換ルールに従って字句系列を書き換える．図 2 から生成される字句系列の変換ルールを**図 11** に示す．なお，図 11 は読みやすさを考慮して整形している．

パターン変換記述を構成する各記述に対する構文解析は，パターン変数と文末仮想字句を C 言語の字句定義に追加するのみで，通常のソースコードの解析と同じである．パターン変数は識別子として定義する．ただし，パターン変数の意味と解析結果が合わない場合があるので，その補正が必要である．型が `STMT*`のパターン変数は文として識別されるが，単一の文ではないので，仮想字句を取り除く．また，宣言に合致する型 `DECL` のパターン変数も文として識別されるので，仮想字句を文から宣言に修正する．

各断片の属性付き字句系列から字句系列の変換ルールへの変換は，各字句を変換ルールの書式に合うように変換するのみである．ただし，パターン変数以外の字句は，重複しないように名前を付け，パターン変数はその名前をそのまま用いる．また，変換前パターンに記述されている空白と変換対象のソースコードに記述されている空白は異なるので，その差異を吸収するために，空白およびコメントの連続を種別 SP の 1 つの字句に置き換える．種別 SP は，任意個の空白およびコメントの列に適合する．

表 1 実装規模

Table 1 The size of the implementation.

解析処理名	モジュール名	行数
字句解析器	Tokenizer	119
括弧対応関係解析	CorrespondingID	46
粗粒度構文解析	CoarseGrainedParser	430
仮想字句対応関係解析	BeginEnd	35
カンマ補正	FixCma	32
字句系列書き換え	RewriteTokens	192
小計		854
定義名	**ファイル名**	**行数**
字句定義	token.def	48
前処理命令解析	prep.rules	45
ヒューリスティック補正 1	syntax1.rules	359
ヒューリスティック補正 2	syntax2.rules	187
ヒューリスティック補正 3	syntax3.rules	42
小計		681

4. 実装と評価

4.1 構文解析器の実装

実装した解析器の規模を**表 1** に示す．各モジュールはクラスとして定義しており，インスタンスを生成して利用する．字句解析については，Tokenizer のインスタンスに字句定義ファイル token.def を読み込ませることで，その定義に従った字句解析器を構成する．前処理命令解析とヒューリスティック補正については，RewriteTokens のインスタンスをそれぞれ生成し，各ルールを読み込ませることで解析器を構成する．解析器全体の組合せの記述例を**図 12** に示す．これは図 6 に従っている．また，構文解析器は単独のモジュール CParser（104 行）として定義し，その内部には図 12 と同じ処理を含む．

RewriteTokens は，そのインスタンスに字句系列の変換ルールを読み込ませると，文字列の正規表現に用いた置換記述に変換して保持する．インスタンスに，字句系列の並

```
use Tokenizer;
use RewriteTokens;
use CorrespondingID;
use CoarseGrainedParser;
use BeginEnd;
use FixCma;

# get path of definition files.
$path = $INC{"Tokenizer.pm"};
$path =~ s/Tokenizer\.pm$//;

## get all tokens.
$text = join('', <>);

## parse tokens
$tn = Tokenizer->new()->load("$path/token.def");
$text = $tn->parse($text);

## Preprocess directives
$tn = RewriteTokens->new()->load("$path/prep.rules");
$text = $tn->rewrite($text);

## Parens analysis
$ci = CorrespondingID->new();
$text = $ci->conv($text);

## Coarse-grained source code analysis
$cgp = CoarseGrainedParser->new();
$text = $cgp->parse($text);

## Add Begin-End Identifier.
$be = BeginEnd->new();
$text = $be->conv($text);

## Correct identifier of commas.
$fc = FixCma->new();
$text = $fc->conv($text);

## Refining Parse Tree
$rt1 = RewriteTokens->new()->load("$path/syntax1.rules");
$rt2 = RewriteTokens->new()->load("$path/syntax2.rules");
$rt3 = RewriteTokens->new()->load("$path/syntax3.rules");
$text = $rt1->rewrite($text);
$text = $rt2->rewrite($text);
$text = $rt3->rewrite($text);

## Add Begin-End Identifier (again)
$text = $be->conv($text);

print $text;
```

図 12 構文解析器の実現例
Fig. 12 An implementation of parser.

表 2 GNU Coreutils 8.5 の解析結果
Table 2 The result of Analysis of GNU Coreutils 8.5.

対象ファイル	976 個，227,609 行（平均約 233.2 行）
解析実行時間	約 531.0 秒（平均約 0.5 秒/ファイル）
最大行数	lib/vasnprintf.c（5,560 行，約 26.08 秒）
最長時間	gnulib-tests/test-vasprintf-posix.c（3,654 行，約 36.71 秒）

びであるテキストを入力すると正規表現に基づいて変換を繰り返し適用する．文字列の正規表現を利用することで，Perl の正規表現の処理エンジンをそのまま使用できる．

4.2 構文解析器の実行性能

現実のソースコードが属性付き字句系列へ変換できることを確認するために，GNU Coreutils 8.5 と FreeBSD 8.2R を解析したところ[*1]，それぞれ無限ループに陥ることなく，解析が終了した．GNU Coreutils 8.5 の規模と解析時間を**表 2** に，ファイルの行数と解析時間の関係を**図 13** に示す．図中の曲線は最小二乗法によって推定した近似曲線 $y = (1.1 \times 10^{-4}) \times x^{1.45}$ を表す．全体の約 95%が 1,000 行以下のファイルであり，それらの解析時間は 4 秒未満であった．FreeBSD 8.2R の規模と解析時間を**表 3** に示す．最長時間がかかったファイルが最大の行数でもあった．1 万行以上のファイル（28 個）を除くと，平均 1.3 秒（平均約 431 行）であった．どちらも，一部の大規模なファイルを除けば，個々のファイルは実用的な時間で解析できている．

前処理ついては，2.3 節で述べたように前処理命令を空白と見なす近似的な方法を用いたが，解析できないファイルの全体に占める割合は小さかった．GNU Coreutils 8.5 ではすべてのファイルを解析でき，FreeBSD 8.2R では 144 個のファイルで括弧の対応がとれず解析できなかった．このうち，102 個のファイルは制御文の条件部分を条件分岐命令で切り分けており，複合ブロックの中括弧や，条件式を囲う括弧が重複していた．20 個のファイルは，関数の仮引数定義や呼び出しの一部を同様に切り分けていた．これらは，括弧の数が合わないことで発見したが，括弧の数が

[*1] Intel Xeon 2.27 GHz，4 GB，FreeBSD 8.2R，Perl v5.10.1

表 3 FreeBSD 8.2R の解析結果
Table 3 The result of analysis of FreeBSD 8.2R.

対象ファイル	約 22 千個，約 1,000 万行（平均約 456.0 行）
解析実行時間	約 41,084 秒（約 11 時間 25 分）（平均約 1.9 秒/ファイル）
最長時間	sys/contrib/octeon-sdk/cvmx-csr-db.c（74,292 行，約 8,057 秒）

```
/* (a) coreutils-8.5/src/expr.c */
VALUE *v IF_LINT (= NULL);

/* (b) coretuils-8.5/src/sum.c */
bool (*sum_func) (const char *, int) = bsd_sum_file;
```

図 14 誤判定の事例
Fig. 14 Errors in analysis.

```
/* (c) coreutils-8.5/src/ls.c */
DEFINE_SORT_FUNCTIONS (ctime, cmp_ctime)
DEFINE_SORT_FUNCTIONS (mtime, cmp_mtime)
DEFINE_SORT_FUNCTIONS (atime, cmp_atime)
DEFINE_SORT_FUNCTIONS (size, cmp_size)
DEFINE_SORT_FUNCTIONS (name, cmp_name)
DEFINE_SORT_FUNCTIONS (extension, cmp_extension)

/* (d) coretuils-8.5/src/head.c */
#if HEAD_TAIL_PIPE_BYTECOUNT_THRESHOLD < 2 * READ_BUFSIZE
    "HEAD_TAIL_PIPE_BYTECOUNT_THRESHOLD must be at least 2* READ_BUFSIZE"
#endif
```

図 15 構文規則と異なる記述
Fig. 15 Illegal descriptions in source codes.

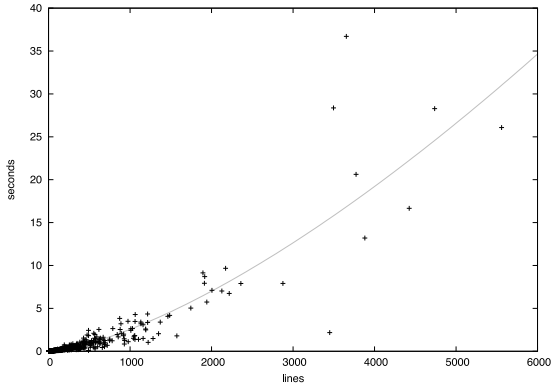

図 13 行数と解析時間
Fig. 13 Lines and analysis times.

合っているが対応関係が正しくないファイルもあり，そのようなファイルも含めて解析方法を検討する必要がある．また，残りの 22 個は，括弧の対応がとれない誤った記述を含んでいた．マクロ定義内の式や，特定のマクロを定義すると有効になる文の中に存在しており，これらは 1 度もテストされたことがないと想定される．

4.3 文の識別の精度

TEBA では，対象ソースコードおよびパターン変換記述を構文解析して得られた属性付き字句系列に基づいて書き換えを行うので，正確に書き換えが行えるかどうかは，構文解析器の精度に依存する．解析精度を確認するために，前処理前のソースコードを解析できる srcML [3] との比較を行った．比較では，それぞれの解析結果から文を抽出し，一致するかどうかを確認した．なお，TEBA が関数の範囲を不適切に識別した場合には，その影響が文の解析にも波及するので，結果的に関数の識別の正しさも確認できる．解析対象は，GNU Coreutils 8.5 のディレクトリ src に含まれる拡張子が .c のファイル（113 個）のみとし，解析結果から文のリストを深さ優先で取り出すスクリプトを記述し，TEBA および srcML の文のリストの差分を分析した．なお，srcML では，キャストを含む文や空文が文にならないので正しいことを目視で確認した．さらに，タグ名に line を用いた構造体が存在すると，それ以降の解析を誤る問題があるので，それを回避するよう解析対象のソースコードを修正した．また，TEBA については，後述する問題があり，TEBA の実装を修正をした．

比較の結果，TEBA が識別した 26,305 個の文の中に，**図 14** に示す 2 個の宣言が誤りとして含まれていた．(a) は宣言の中で特殊なマクロを用いており，そのマクロを関数と識別したことで，`VALUE` を型と識別せず，宣言として識別できなかった．(b) は，関数へのポインタの識別が実装できていないことが原因であった．どちらも，識別子の種別の分類を正確にし，(b) についてはさらにルールを整備すれば解決できる．

発見された誤りはこの 2 個のみのであり，識別されなかった文はなかったので，文の識別精度は実用的なレベルにあると考える．なお，見掛け上，文の形をしているが，展開すると宣言になるマクロも存在していた．展開結果まで考慮すれば，これらも誤りであるが，前処理前のソース

表 4 識別子の解析結果
Table 4 The result of identifiers analysis.

種別の数	IDENT の有無	識別子の個数	割合
1 種類	なし	12,997	91.5%
1 種類	あり（IDENT のみ）	35	0.2%
2 種類	なし	999	7.0%
2 種類	あり	131	0.9%
3 種類	なし	19	0.1%
3 種類	あり	16	0.1%
合計		14,197	100%

コードを解析するという目的においては正しい結果である．また，前処理の展開結果を調べる仕組みを持たないので，これ以上の検証はしていない．

誤りではないが，srcML と TEBA で識別結果が異なる特殊な記述も存在した．その例を**図 15** に示す．(c) は関数を定義するマクロを用いている例であり，(d) はコンパイル時にマクロの値に関する制約条件を検証し，制約条件を満たさない場合には文字列を有効にすることでコンパイルエラーを引き起す記述である．

(c) に対して，srcML はマクロの参照と識別し，文や宣言としては識別しない．TEBA は，文末のセミコロンが存在しないので，一連の記述をひとまとめに式として扱い，その後に出現する最初の文の切れ目までを 1 つの文として識別する．また，その結果，それ以降の解析結果が誤った状態となった．そこで，このような構文に合致しないマクロ呼び出しの後ろに文末仮想字句を挿入する変換ルールを持つフィルタを作成し，粗粒度構文解析の直前に追加したところ，すべて文として識別され，それ以降の解析も正しくなった．なお，このマクロの展開結果に基づけば，これらも宣言として識別すべきものである．解析対象には，このようなマクロが 3 つ定義されており，計 19 カ所で使用されていた．

(d) に対して，srcML は文字列を文として識別し，TEBA は文字列とその後に存在する制御文までの前処理命令と空白を文として識別する．TEBA と srcML の解析結果を一致させる補正フィルタは作れるが，構文的に正しくない記述の扱いは，その解析結果を利用する目的によって異なるので，これ以上の補正は行わなかった．

4.4 識別子の解析精度

TEBA では，字句解析の段階では，識別子の種別を一律に IDENT としているが，ヒューリスティック補正で，前後の文脈から変数や型，タグなどの識別の種別を区別し，種別を置き換えている．文を宣言に補正する際には，型の存在に基づいており，前節で述べた宣言以外は正しく補正できることから，型に対する識別の精度は高い．

識別子の種別については，srcML においても区別をしておらず，比較によって確認することができない．そこで，

```
%before
${type:TYPE} ${others:DECR}, ${decr:DECR} ;
%after
${type} ${others};
${type} ${decr};
%end
```

図 16 宣言の分解
Fig. 16 Decompoistion of declarations.

識別子は，多くの場合，同じ名前を異なる種別で使うことは少ないという経験則に基づき，各ファイルごとに同じ名前の識別子に対して，種別が何種類割り当てられているか調査した．解析対象は，GNU Coreutils 8.5 のディレクトリ src に含まれる拡張子が.c のファイル（113 個）である．種類の数と個数を**表 4** に示す．全体としては，約 9 割が単一の種別に区別できており，一定の精度が得られていると解釈できる．なお，種別を区別できず IDENT のままになった識別子が存在するが，その多くはマクロの定義の引数や定義の中で参照されている識別子などマクロの参照箇所の文脈を調べないと確定できない識別子であった．また，関数のポインタ型を正しく解析できず，型を関数名と誤判定する事例が含まれていた．これについては今後の改善が必要である．

4.5 パターン変換系の実装と評価

パターン変換系を，図 10 に基づき，Perl で実装した．コメントやデバッグ文も含めて 113 行であった．実際に書き換えられることを事例を用いて確認した．すでに示した図 3 は，図 2 を用いて実際に書き換えた例である．また，宣言を分解するパターン変換記述の例を**図 16** に，これを用いて書き換えた例を**図 17** に示す．パターン変換記述が抽象構文木の操作になっていることを確認するためには，抽象構文木に対する基本操作が実現できていることと，書き換えたあとも妥当な抽象構文木になっていることを確認する必要がある．

4 つの基本操作は，2 つの例に含まれている．図 3 では，malloc() の呼び出しを含む if 文が，my_malloc() の文に置き換わっており，文を単位とした削除と挿入ができている．また，パターン変数の`$cast` や`$size` により，各式が移動している．図 16 では，宣言を分解したあとで，同じ型を 2 カ所で参照しており，複製されていることが，図 17 で確認できる．

書き換えによって，仮想字句や括弧，カンマが持つ識別記号に不整合が生じる可能性があり，その場合，妥当な抽象構文木を構成できていない．図 16 は宣言を 2 つに分解するが，図 17 の書き換え前には 1 つの宣言に 3 つの宣言子が含まれており，この書き換えを 2 回適用する必要がある．1 回目の適用で識別記号が正しく補正されないと，2 回目が適用されない．図 17 では，宣言の範囲や，宣言を区切るカンマを適切に識別して書き換えていることから，識

```
//* 書換え前 *//
static char a, b = 'b', **c[];
size_t s[] = {0, 1, 2}, f(int c, int d),
    *p = NULL;
```

```
//* 書換え後 *//
static char  a;
static char   b = 'b';
static char   **c[];
size_t  s[] = {0, 1, 2};
size_t   f(int c, int d);
size_t
    *p = NULL;
```

図 17 宣言の分解の例
Fig. 17 The result of decompoistion of declarations.

```
# デバッグ文の挿入
%before
${v:ID_VAR} = total(${ar:EXPR}, ${n:EXPR});
%after
${v} = total(${ar}, ${n});
#ifdef DEBUG_TOTAL
fprintf(stderr, "DEBUG: total = %d (at %d)\n", ${v}, __LINE__);
#endif
%end
```

```
# DEBUG_TOTAL の削除
%before
#ifdef DEBUG_TOTAL
${stmts:STMT*}$;
#endif
%after
%end
```

図 18 デバッグ文の挿入と削除
Fig. 18 Injection and removal of debug statements.

```
%before
draw_line(${arg1:EXPR}, ${arg2:EXPR}, ${arg3:EXPR}, ${arg4:EXPR}, ${type:EXPR});
%after
struct point tmp_p1, tmp_p2;

tmp_p1.x = ${arg1}; tmp_p1.y = ${arg2};
tmp_p2.x = ${arg3}; tmp_p2.y = ${arg4};
draw_line(tmp_p1, tmp_p2, ${type});
%end
```

図 19 引数の構造体化
Fig. 19 Replacement of arguments to struct variables.

別記号が正しく補正されていることが確認できる．なお，TYPE と DECR は，それぞれ宣言指定子と宣言子を構成する任意の字句の連続と定義しており，空白や改行，コメント，カンマを含む．Perl の正規表現のエンジンを利用しているので，前方のパターン変数から字句列が最長となるように適合する．宣言子が 3 つ以上含まれる場合には，others は宣言を区切るカンマのうち最後の 1 個の直前までの字句列に適合し，パターン変数 decr は最後の宣言子のみに適合する．また，出力結果の最後の変数 p の宣言に改行が含まれている理由は，改行がパターン変数 decr の適合範囲に含まれたことにある．

図 16 を GNU Coreutils 8.5 のディレクトリ src に含まれる拡張子が.c のファイルに適用し，書き換えにかかる時間を測定した．27 個のファイルに含まれる 57 個の宣言に対して書き換えが適用され，ファイル 1 つあたり平均で約 0.2 秒かかった．最長は sort.c で約 0.8 秒であり，書き換えられる宣言を含まないファイルに対しては平均 0.1 秒であったことから，いずれも実用的な時間で書き換えが実現できている．

その他，パターン変換記述の中に前処理命令を含めても正しく書き換えができることを確認するために，**図 18** を作成した．これは特定の代入文の直下に前処理命令で囲ったデバッグ文を挿入する例と，それを取り除く例である．デバッグ文は，作業対象のソースコードやそのときの作業目的によって記述する内容が異なり，その挿入は一時的に発生する定型的作業の例である．なお，図 18 の挿入を適用した字句系列に対し，直接，削除のパターン変換記述を適用すると，元に戻ることも確認しており，前処理命令を含めても適切に抽象構文木が構成できている．また，**図 19** に示す引数を構造体に置き換える例を作り，目的の引数を書き換えられることを確認した．この場合，変数の宣言を追加しているが，同じスコープの中で複数箇所を書き換えた場合など，名前が衝突する可能性がある．また，ただ変数を追加するのみでは，変数から変数への代入だけをするなど冗長な記述になる可能性があり，書き換えた箇所はすべて見直す必要がある．しかし，手作業ですべての作業を行う場合に比べ，一部の単純作業が自動化されるだけでも，作業効率の向上につながる．

5. 考察

本論文で目的とするソースコードの書き換え環境は実現できたが，基盤のみであり，パターン変換記述の表現力など，発展させていくべき課題も多い．以下では，パターン変換記述の表現力，構文解析器の解析精度，空白の扱い，前処理命令の扱いについて議論する．また，発展させていくうえで考慮すべき拡張性と保守性についても述べる．

5.1 パターン変換記述の表現力の問題

本論文のパターン変換記述は，ソースコードの断片にパターン変数を追加したのみである．抽象構文木に対する基本操作は実現できるが，抽象構文木の操作を直接記述する環境に比べると，適合すべき箇所に対する制約条件に関する記述能力が弱い．表現方法を高める方法としては以下の2つが考えられる．

- 繰返しや選択などのメタ記号の導入
- パターン変数に対する制約条件

パターン変換系は，字句系列の変換ルールに変換して書き換えており，字句系列の変換ルールは字句をアルファベットとする正規表現と見なせる．よって，繰返しや選択などのパターンを扱うことは難しくない．また，特定の変数を参照する式など，条件付きの部分木を表現する方法がない．パターン変数の型に制約条件を記述できるようにする必要がある．

5.2 解析の精度の課題

TEBA は，一般的な構文解析の方法と比べると，記号表と式の抽象構文木に関する情報が不足している．記号表の構築は，一般的な方法とは異なり，次の点を考慮する必要がある．

- 異なる定義を複数持つ識別子の扱い
- 文脈によって異なる種別になる識別子の扱い
- 誤った種別で識別された識別子の補正

異なる定義を持つ識別子は，コンパイル条件によって定義を変えている場合に生じる．また，同名の識別子に異なる種別を識別する場合があり，解析の誤りなのか，名前空間の違いなのか区別できない場合がある．解析の誤りの場合には，何が正しいのか判別できない可能性がある．さらに，パターン変換系で記号表の情報を扱うためには，字句系列にどのように加えればよいかという点も問題であり，検討しているところである．

式については，括弧の対応関係のみを識別しており，正確な構文解析は行っていない．プログラマの意図と結合則が合わない場合にも対応でき，パターン変換記述の表現力を高める効果があるが，ソースコードの書き換えの応用を広く考えると，式の正確な解析も必要になる．実際に，TEBA の応用研究 [7] で必要となり，式を詳細化する解析器を試作した．ただし，解析に時間がかかるといった問題があり，改善を進めている．

5.3 書き換え環境の実用性

TEBA では，解析精度が低い部分を許容することで前処理前のソースコード断片の解析を実現している．書き換え環境としての実用性を高めるためには，前節で述べたように解析精度を高め，解析精度が低い箇所を減らすことが必要である．しかし，前処理前のソースコード断片を対象とする限り，解析精度が低い部分は残ることは避けられない．特に，以下の書き換えは容易には実現できない．

- 前処理の条件分岐命令によって分断される箇所の書き換え
- マクロの展開結果を用いた書き換え
- 書き換え対象のソースコードと変換前パターンの解析精度が一致しない書き換え

条件分岐命令で分断される問題は，すべての分岐の組合せに基づいたスライスを作ればパターンが適合し，スライスごとに書き換えてから合成することで解決できる．ただし，現実的に分岐の組合せ数は膨大になる．たとえば，実験に用いた FreeBSD 8.2R には 114 個の分岐命令が含まれるファイル[*2]が存在し，すべての分岐の組合せを考えることは現実的ではない．パターン変換の対象となる範囲に含まれる分岐を推定するなど，分岐の組合せ数を抑える方法が必要である．

プログラマがソースコードを理解するとき，前処理前の単純な字面で見るときと，前処理後を想像して見るときの2つの視点があるが，TEBA は前者の視点に基づいている．よって，マクロの展開結果まで調べないと書き換えられないものには対応できない．典型的な例は，マクロの展開結果の中に含まれる識別子であり，ある識別子の名前を変更するときに，それと同名の識別子がマクロに含まれていても，マクロがどこで参照されているかまで調べないと正しい書き換えができない．Spinellis らの解析手法 [8] などを利用し，前処理前後の展開に関する情報を取り入れる方法を検討する必要がある．

解析精度が一致しない問題とは，書き換え対象のソースコードと変換前パターンのどちらか一方では識別子の種別が詳細化され，もう一方では IDENT のまま残るなど，字句の種別が異なることや，構文と合わない形式でマクロを参照することで，文や宣言などの仮想字句の存在の有無の違いや，仮想字句の位置が異なることである．識別子の種別の問題は記号表の導入で解決できる場合があるが，変換前パターンは定義や宣言が不足している可能性があり，パターンを記述する際に何らかの補助的な情報を加える必要がある．また，字句の種別間の包含関係を定義し，それらの差異を吸収させることも解決につながる可能性がある．ただし，適合条件を緩めることで，意図しない箇所にパターンが適合する可能性が生じるので，慎重に検討すべきである．仮想字句の有無や位置の問題を直接避ける方法はないが，構文と合わない形でのマクロの利用を禁止するなど，コーディング規約を規定することで問題の発生を抑えることはできる．

これらの問題に加え，書き換えが意図しない箇所に行われる可能性も検討する必要がある．現状は，式の結合則に

[*2] contrib/ntp/ntpd/ntpd.c

基づかない適合など，解析精度の低さに起因する場合と，条件分岐命令の`#else`命令を空白として取り扱うことで，本来は関係のない前後のテキストにまたがって適合する場合が考えられる．前者は，解析精度を上げることで解決し，後者は前処理の分岐に基づいたスライスの書き換えで解決することが見込まれる．

精度の向上や前処理に関する取扱いを改善することは容易ではない．現状のTEBAでの実用性を考えた場合，対象となるソースコードの書き方を十分把握した状況で，特定の条件に合うソースコードを対象とした書き換えのみに適用可能である．仮想字句や括弧，カンマの対応関係はヒューリスティックな知識に基づかずに行われており，括弧の組や仮想字句の組をまたがった不適切な適合は発生しない．よって，対象となるソースコードは条件分岐命令で分断されるテキストを含まないか，含む場合には前処理命令を無視しても構文的に正しい場合に限定し，文や宣言，括弧やカンマを区切りの単位とした書き換えを対象とすれば，正しく処理できる．

5.4 空白・コメントの維持の課題

パターン変換前後の空白やコメントの維持は，実用性の点で重要である．パターン変換系は，変換前パターンに適合した箇所のみを書き換えるので，書き換えた箇所のコメントと空白の維持方法が必要である．コメントを維持するか否かや，変換後の位置は，その内容に依存する．しかし，意味に基づいた判定は難しいので，一般的にはすべて維持し，書き換え前後で対応する構文要素の説明となるよう配置する方法が必要である．空白は，変換後にインデントのレベルが変わる場合や，変換後断片のインデントの方法が対象ソースコードと異なる場合があるので，書き換え箇所の前後の空白の状況に合わせて調整する仕組みが必要である．

5.5 拡張性と保守性の課題

前節までに述べたような多くの課題に対処するためには，拡張性の確保も重要である．TEBAの開発の過程では，徐々に仕様を修正したが，次の特徴により，比較的，修正が小規模にとどまり，見通し良く拡張できている．

- 抽象構文木ではなく，字句系列の書き換えに基づいたこと
- 対象のソースコードと断片の両方に同一の解析器を用いたこと
- 解析器をフィルタ型の構成にし，段階的に解析精度を上げる方法を採用したこと

たとえば，オープンソースの解析では，構文規則と合わない表現が含まれたが，フィルタの追加のみで対処できた．また，解析にあたっては，前処理命令は行を単位とし，C言語そのものは字句や構文要素を単位とする2つの視点が必要であるが，別々のフィルタとして実現したことで見通し良い設計となった．今後も，この特徴を維持していくことが必要である．

全体の記述量の少なさも重要である．特に大学で開発支援環境を保守する場合，数年で入れ替わる学生が保守に貢献することは難しい．一方，大学の教員も日常の業務に多くの時間を割かれるなかで，必ずしも研究成果に直結しない保守作業を続けることは難しい．大規模になるほど，保守作業をするためにソースコードを読み直して理解する時間が長くなり，その時間の確保に苦慮する．規模を抑えつつ，実用性が確保できる程度の精度で実現していくことが必要である．

6. 関連研究

ソースコードの書き換えを前提としたソースコード解析ツールや環境の代表としてはDMS [1]，Proteus [2]，srcML [3] がある．これらは抽象構文木を構築し，抽象構文木を操作して書き換えを実現する．TEBAは，ソースコードの断片を用いてパターン変換を記述するので，構文木の操作に写像する必要がなく，書き換えを容易に記述できる．また，TEBAは字句の並びを中心としたモデルであり，構文規則に合わない記述を許容するので，前処理命令を含めて書き換えができるなど，自由度が高い．ただし，抽象構文木を操作する場合に比べると，細かな制約条件を書けず，記述可能な書き換えの範囲が狭い．なお，TEBAと同じようにソースコードの断片を用いてパターン変換を記述する研究は知りうる範囲には存在しなかった．

TEBAと同様に，字句系列を中心とした考え方はLSME [9] で用いられており，字句の並びに対するパターンを記述できる．ただし，構文に関する情報がないので，パターンの記述能力が弱く，また，情報の抽出のみが可能であり，書き換えは実現できない．

前処理前の状態で構文解析を行う解析器としては，srcMLとyacfe [10] がある．どちらも一般的な慣習に基づいた記述方法を想定し，構文規則を定義している．srcMLは，TEBAと同様に式のレベルは解析しない．識別子についても，すべて同じタグを割り当てており，種別を区別しない．yacfeは前処理命令の記述方法に前提を置くことで，式のレベルまで解析する．また，解析を2段に分け，簡素な解析器で解析をした後，ヒューリスティックなルールを適用して，詳細化する．TEBAは複数の段階を持つが，ヒューリスティックなルールを用いて詳細化しており，基本的な考え方はyacfeと同じである．このようなヒューリスティックな知識に基づく補正を行う方法は，字句系列の一部だけを解析していくisland grammer [11] と共通する部分が多い．なお，srcMLとyacfeは，JavaとC++の解析にも対応しており，精度や粒度も含めると，直接的な比較は難しいが，srcMLは全体で約12千行，yacfeはC言語の解析に関す

る部分で約 22 千行の規模であり，規模は TEBA の方が小さい．

前処理前の状態で解析する代わりに，前処理を適用してから構文解析を行い，解析結果に対して，前処理の展開情報の逆写像をかけて，前処理前の解析情報を得る方法もある [12], [13], [14]．展開情報を用いて，ソースコードの書き換えを実現することも可能であるが，つねに展開の前後の状態を意識して書き換え方法を記述する必要がある．また，特定のコンパイル条件で有効になるテキストのみを解析するので，無効になった箇所は解析できず，さらに，前処理命令そのものも解析できない．TEBA の場合はこれらの問題はないが，前処理の条件分けによってテキストが分断されるような場合については考慮していない．この点については，先行研究 [15], [16] を参考に，今後取り組んでゆく必要がある．

本論文で対象とする定型的な記述には，コードクローン [17] も含まれ，TEBA を用いたコードクローンの編集支援も可能である．編集作業においてクローンを追跡し管理する研究 [18], [19] やクローンの同時編集をエディタ上で支援する研究 [20], [21] が行われている．追跡については，クローンに対するパターンを記述すれば TEBA でも支援可能である．ただし，クローンを手作業でパターン化することは手間が大きく，クローンからパターンを自動生成するような仕組みがない限りソースコード中に存在するすべてのクローンを管理するような目的には適さない．同時編集については，Linked Editing [20] や LAPIS [21] がエディタ上でインタラクティブに同時編集をする環境を提供している．TEBA では，その作業を記述したうえで適用することになるが，同時に書き換えるという点では同等であり，かつ，あとから同じような作業が発生したときに再利用できるという利点を持つ．ただし，エディタ上ではカーソルを用いてプログラマが意図する箇所を指定して編集できるが，TEBA のパターンでは必ずしも指定できるとは限らない．修正箇所に特徴的な記述があり，かつ，慎重に編集内容を見極める必要がある場合には TEBA を使用し，短時間に修正をして試験をするような場面ではこれらのエディタを用いるなど，使い分けが必要である．

ソースコードの書き換えを支援するという点ではリファクタリング [22] の支援とも関連する．リファクタリングの手順を記述する専用の言語としては JunGL [23] が提案されており，専用のブラウザとして Xrefactory [24] などがある．また，Eclipse のプラグインなど，統合環境にもリファクタリングの操作が実装されている．その他，先に述べた解析環境である DMS，Proteus なども応用例としてリファクタリングをあげている．リファクタリングは，ソースコードの書き換えであるので，TEBA でも理論上は実装は可能である．ただし，リファクタリングはその前後で同じ振舞いを保持している必要があり，その制約が厳しい．リファクタリングが適用可能かどうかを調べるためには，抽象構文木の構成関係や依存関係などをたどりながら検査する必要があるが，あらゆる状況を考慮して実装することは難しく，既存の統合環境が持つリファクタリングの機能にもバグが存在する [23]．JunGL は，抽象構文木を書き換えた際に依存関係の更新が簡単に行えるよう遅延評価型の依存関係を定義できるようにするなど，制約条件の検査と書き換えがより簡潔になるよう専用言語を提案している．TEBA は，抽象構文木を提供するのみで，制約条件の検査を支援する仕組みを持たず，依存関係の解析も含めてすべて記述する必要がある．C 言語に限定した場合，前処理があることから，そのリファクタリングは難しい [25]．前処理を前提とした支援環境としては Xrefactory や CScout [26] があり，前処理前後の関係を解析したうえで，それぞれ特定のリファクタリング操作のみを支援する．JunGL のような汎用的な記述支援環境が望まれるが，前処理の解析を含めたそのような環境は存在しない．

書き換えを実装しているリファクタリング支援環境では抽象構文木を持ち，TEBA が対象とする一時的な定型作業も抽象構文木への操作として記述可能である．ただし，木構造を理解しつつ操作を記述する必要がある．TEBA は，書き換え前後のソースコードの断片を用いてパターンを記述できるので，構文要素間の構成関係に基づいた書き換えであれば容易に実現できる．

7. おわりに

本論文では，定型的な記述を書き換える作業を支援するために，ソースコードの断片を用いてパターン変換を記述すると，ソースコードの抽象構文木を操作して書き換えを実現する方法を提案した．定義不足や構文規則に合わない記述を含んだまま解析するために，精度が低い部分を許容する抽象構文木を属性付き字句系列を用いて表現し，字句系列の変換系を用いて，書き換えを実現できることを示した．さらに，実装を行い，実用的な精度で構文解析ができることと，ソースコードに対する書き換えが実現できることを示した．

今後の課題は，5 章で述べたように，パターン変換記述の表現力の向上や，構文解析器の精度の向上などがある．また，属性付き字句系列に対する操作は，パターン変換のみに固定する必要はなく，より複雑な編集作業への応用も考えられる．特に，データフローやコントロールフローを取り扱う枠組みがあれば，リファクタリングへの応用も可能である．さらに，小さな編集作業であっても，記述をして残すことができれば，過去の版に遡って作業を理解するときに，作業内容が明確化されるといった利点もある．様々な作業の自動化に向けた基盤となることを目指していく必要がある．

謝辞 本研究は，文部科学省研究費補助金基盤（C）（課

題番号：21500042，22500036，22500037，24500049）の助成を受けた．

参考文献

[1] Baxter, I.D., Pidgeon, C. and Mehlich, M.: DMS®: Program Transformations for Practical Scalable Software Evolution, *Proc. 26th International Conference on Software Engineering*, pp.625–634, IEEE Computer Society, Washington, DC, USA (2004) (online), available from ⟨http://dl.acm.org/citation.cfm?id=998675.999466⟩.

[2] Waddington, D. and Yao, B.: High-fidelity C/C++ code transformation, *Sci. Comput. Program.*, Vol.68, pp.64–78 (online), DOI: 10.1016/j.scico.2006.04.010 (2007).

[3] Collard, M.L. and Maletic, J.I.: Document-Oriented Source Code Transformatin using XML, *Proc. 1st International Workshop on Software Evolution and Transformation*, pp.11–14 (2004).

[4] 吉田　敦，蜂巣吉成，沢田篤史，張　漢明，野呂昌満：属性付き字句系列に基づくプログラム書換え支援環境の試作，ソフトウェアエンジニアリング最前線（ソフトウェア・エンジニアリング・シンポジウム 2010 予稿集），pp.119–126 (2010).

[5] 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，Vol.J86-D-I, No.3, pp.159–168 (2003).

[6] 吉田　敦：軽量下流 CASE ツール構築のためのソースプログラム表現形式の提案，情報処理学会論文誌，Vol.46, No.9, pp.2326–2336 (2005).

[7] 曽我展世，吉田　敦，蜂巣吉成，沢田篤史，張　漢明，野呂昌満：表現の違いを考慮したマクロ逆置換方法の提案，ソフトウェアエンジニアリングシンポジウム 2011 論文集，pp.1–6 (2011).

[8] Spinellis, D.: Global Analysis and Transformations in Preprocessed Languages, *IEEE Trans. Softw. Eng.*, Vol.29, pp.1019–1030 (online), DOI: 10.1109/TSE.2003.1245303 (2003).

[9] Murphy, G.C. and Notkin, D.: Lightweight lexical source model extraction, *ACM Trans. Softw. Eng. Methodol.*, Vol.5, pp.262–292 (online), DOI: http://doi.acm.org/10.1145/234426.234441 (1996).

[10] Padioleau, Y.: Parsing C/C++ Code without Preprocessing, *Proc. 18th International Conference on Compiler Construction: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, pp.109–125, Springer-Verlag, Berlin, Heidelberg (2009).

[11] Moonen, L.: Generating Robust Parsers using Island Grammars, *Proc. 8th Working Conference on Reverse Engineering*, pp.13–22, IEEE Computer Society Press (2001) (online), available from ⟨http://www.cwi.nl/~leon/papers/wcre2001/⟩.

[12] Livadas, P.E. and Small, D.T.: Understanding Code Containing Preprocessor Constructs, *IEEE 3rd Workshop on Program Comprehension*, pp.89–97 (1994).

[13] 福安直樹，山本晋一郎，阿草清滋：細粒度リポジトリに基づいた CASE ツール・プラットフォーム Sapid，情報処理学会論文誌，Vol.39, No.6, pp.1990–1998 (1998).

[14] 権藤克彦，川島勇人，今泉貴史：TBCppA: 追跡子を用いた C 前処理系解析器，コンピュータソフトウェア，Vol.25, No.1, pp.105–123 (2008).

[15] Aversano, L., Di Penta, M. and Baxter, I.D.: Handling Preprocessor-Conditioned Declarations, *Proc. 2nd IEEE International Workshop on Source Code Analysis and Manipulation*, pp.83–92, IEEE Computer Society, Washington, DC, USA (2002) (online), available from ⟨http://dl.acm.org/citation.cfm?id=827253.827741⟩.

[16] Garrido, A. and Johnson, R.: Analyzing Multiple Configurations of a C Program, *Proc. 21st IEEE International Conference on Software Maintenance*, pp.379–388, IEEE Computer Society, Washington, DC, USA (online), DOI: 10.1109/ICSM.2005.23 (2005).

[17] 神谷年洋，肥後芳樹，吉田則裕：コードクローン検出技術の展開，コンピュータソフトウェア，Vol.28, No.3, pp.29–42 (2011).

[18] Duala-Ekoko, E. and Robillard, M.P.: Clone region descriptors: Representing and tracking duplication in source code, *ACM Trans. Softw. Eng. Methodol.*, Vol.20, pp.3:1–3:31 (online), DOI: http://doi.acm.org/10.1145/1767751.1767754 (2010).

[19] Hou, D., Jablonski, P. and Jacob, F.: CnP: Towards an environment for the proactive management of copy-and-paste programming, *ICPC*, pp.238–242, IEEE Computer Society (2009).

[20] Toomim, M., Begel, A. and Graham, S.L.: Managing Duplicated Code with Linked Editing, *Proc. 2004 IEEE Symposium on Visual Languages – Human Centric Computing, VLHCC '04*, pp.173–180, IEEE Computer Society, Washington, DC, USA (online), DOI: http://dx.doi.org/10.1109/VLHCC.2004.35 (2004).

[21] Miller, R.C. and Myers, B.A.: Interactive Simultaneous Editing of Multiple Text Regions, *Proc. General Track: 2002 USENIX Annual Technical Conference*, Berkeley, pp.161–174, USENIX Association, CA, USA (2001) (online), available from ⟨http://dl.acm.org/citation.cfm?id=647055.715910⟩.

[22] Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D.: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Professional (1999).

[23] Verbaere, M., Ettinger, R. and de Moor, O.: JunGL: A scripting language for refactoring, *Proc. 28th international conference on Software engineering, ICSE '06*, pp.172–181, ACM, New York, NY, USA (online), DOI: http://doi.acm.org/10.1145/1134285.1134311 (2006).

[24] Vittek, M.: Refactoring Browser with Preprocessor, *Proc. 7th European Conference on Software Maintenance and Reengineering*, p.101, IEEE Computer Society, Washington, DC, USA (2003) (online), available from ⟨http://dl.acm.org/citation.cfm?id=872754.873566⟩.

[25] Garrido, A. and Johnson, R.: Challenges of refactoring C programs, *Proc. International Workshop on Principles of Software Evolution, IWPSE '02*, pp.6–14, ACM, New York, NY, USA (online), DOI: http://doi.acm.org/10.1145/512035.512039 (2002).

[26] Spinellis, D.: CScout: A refactoring browser for C, *Sci. Comput. Program.*, Vol.75, pp.216–231 (online), DOI: http://dx.doi.org/10.1016/j.scico.2009.09.003 (2010).

吉田 敦 （正会員）

1991 年名古屋大学工学部情報工学科卒業．1996 年同大学大学院工学研究科博士後期課程単位取得退学．同年豊橋技術科学大学知識情報学系助手，2000 年和歌山大学システム情報学センター講師，2009 年南山大学情報理工学部准教授を経て，2011 年同教授，現在に至る．博士（工学）．プログラム解析および開発支援環境に関する研究に従事．電子情報通信学会，日本ソフトウェア科学会各会員．

蜂巣 吉成 （正会員）

1994 年名古屋大学工学部情報工学科卒業．1999 年同大学大学院工学研究科情報工学専攻博士後期課程修了．同年南山大学経営学部情報管理学科助手，2000 年同大学数理情報学部講師を経て，現在，同大学情報理工学部ソフトウェア工学科准教授．博士（工学）．構造化文書の記述，ソフトウェアの開発環境に関する研究に興味を持つ．電子情報通信学会，日本ソフトウェア科学会，IEEE Computer Society，ACM 各会員．

沢田 篤史 （正会員）

1990 年京都大学工学部情報工学科卒業．1995 年同大学大学院工学研究科情報工学専攻博士後期課程研究指導認定退学．同年奈良先端科学技術大学院大学情報科学研究科助手，1997 年京都大学大学院工学研究科助手，同年同大学大型計算機センター（2002 年より学術情報メディアセンターに改組）助教授を経て，2007 年南山大学数理情報学部（2009 年より情報理工学部に改組）教授，現在に至る．この間，2011 年から 2012 年までチューリヒ大学客員教授．博士（工学）．ソフトウェア工学，組込みシステム工学，ソフトウェア開発支援環境などの研究に従事．電子情報通信学会，日本ソフトウェア科学会，システム制御情報学会，IEEE Computer Society，ACM 各会員．

張 漢明 （正会員）

1989 年同志社大学工学部卒業．1992 年までオムロンソフトウェア（株）勤務．1999 年奈良先端科学技術大学院大学情報科学研究科博士後期課程修了．（財）九州システム情報技術研究所研究員を経て，2000 年より南山大学数理情報学部情報通信学科講師．現在，同大学情報理工学部ソフトウェア工学科准教授．博士（工学）．形式手法に関する研究に興味を持つ．日本ソフトウェア科学会，IEEE Computer Society，ACM 各会員．

野呂 昌満 （正会員）

1981 年慶應義塾大学工学部管理工学科卒業．1986 年同大学大学院工学研究科管理工学専攻後期博士課程単位取得退学．1986 年より南山大学経営学部情報管理学科講師を経て，現在，同大学情報理工学部ソフトウェア工学科教授．この間 1988 年から 1990 年まで米国メリーランド大学計算機科学科客員研究員．工学博士．ソフトウェアアーキテクチャ，アスペクト指向計算，プログラミング言語の意味論および処理系等の研究に興味を持つ．電子情報通信学会，日本ソフトウェア科学会，IEEE Computer Society，ACM 各会員．