情報処理学会論文誌 Vol.56 No.2 593–610 (Feb. 2015)

前処理指令に対する制約のない前処理前コードの構文解析手法

吉田 敦[1,a)] 蜂巣 吉成[1,b)]

受付日 2014年5月17日, 採録日 2014年11月10日

概要：C 言語の前処理前コードの構文解析では，前処理指令の位置などを制約することで構文木を構成するものが多い．本論文では，構文木の不正確さを許容する代わりに，前処理指令に対する制約のない構文解析の手法を提案する．この解析では，前処理指令を無視して構文解析を行うとともに，構文規則とは合わない記述に対しては，仮想的な字句の補正などを行う．また，解析器は字句の書き換え規則で構成し，マクロ定義の内部など，部分的な記述も解析の対象とする．応用例として，前処理の分岐指令の移動ツールとマクロの逆置換ツールを示す．評価実験では，オープンソースソフトウェアの解析に適用し，その精度と問題点について議論する．

キーワード：構文解析器，前処理前コード，書き換え規則，字句系列

A Parser for Unpreprocessed Programs without Constraints on Preprocessor Directives

Atsushi Yoshida[1,a)] Yoshinari Hachisu[1,b)]

Received: May 17, 2014, Accepted: November 10, 2014

Abstract: Parsers for unpreprocessed C programs have been proposed, but they have constraints on the positions of preprocessor directives. In this paper, we propose a parser without the constraints while allowing incorrectness of syntax trees. The basic strategy is to ignore directives, and the parser complements tokens virtually to parse syntactical incorrect parts. The parser is implemented based on token-based rewrite rules, and it parses partial codes in programs, such as the insides of macro definitions. As applications, we show two tools: an alignment tool for branch directives and a macro reverse expansion tool. We also show the experiments of analysis of an open source application and discuss the precision and the problems.

Keywords: parser, unpreprocessed program, rewrite rule, token sequence

1. はじめに

C 言語は，様々なアプリケーションや OS などの開発に広く用いられており，その開発支援ツールは生産性や品質を保つうえで不可欠である．しかし，C 言語では前処理系の利用が前提になっており，開発支援ツールを作る際に，ツールの構成に不可欠な構文解析器の実装が難しいという問題がある．コンパイラなど，構文解析で得られた結果を人間が直接見る必要がなく，また，特定のコンパイル条件のみを扱えばよいツールの場合は，前処理後のコードを解析すればよいので問題はないが，コーディングスタイルの検査やリファクタリング支援など，前処理前のコードに対する解析結果や書き換え後のコードを人間に提示するツールの場合には，特定のコンパイル条件に限定せず，前処理前の記述に含まれるすべてのコードに対する部分木を含んだ構文木が必要である．さらに，それらのツールを利用したあとのコードは，継続的に保守や開発の対象となるので，ツールは，コメントや空白，前処理指令をすべて維持する必要がある．そのためには，C 言語の構文木に，前処理指

[1] 南山大学
Nanzan University, Seto, Aichi 489–0863, Japan
[a)] atsu@nanzan-u.ac.jp
[b)] hachisu@nanzan-u.ac.jp

```
01:  #include <stdio.h>
02:
03:  #ifdef ABS_T
04:  #if (!defined(T_CMP) || !defined(T_REV))
05:  Please define T_CMP and T_REV.
06:  #endif
07:  #endif
08:
09:  #define INT_CMP ((v) >= 0)
10:
11:  #ifdef ABS_T
12:  ABS_T abs_##ABS_T(ABS_T v) {
13:  #else
14:  int abs_int(int v) {
15:  #endif
16:  #ifdef ABS_T
17:    if (T_CMP(v))
18:  #else
19:    if (INT_CMP(v))
20:  #endif
21:      return v;
22:    else
23:      return
24:  #ifdef ABS_T
25:        ABS_REV(v);
26:  #else
27:        -v;
28:  #endif
29:  }
30:
31:  int test_var;
```

(a) コード例

```
01:  [B_DIRE]#include <stdio.h>[E_DIRE]
02:
03:  [B_DIRE]#ifdef {ABS_T:ID_MC}[E_DIRE]
04:  [B_DIRE]#if {({{{!defined{({T_CMP:ID_MC})}}}} || {!{defined{({T_REV:ID_MC})}}}})}[E_DIRE]
05:  [B_ST]Please define T_CMP and T_REV.:ID_MSG[SC][E_ST]
06:  [B_DIRE]#endif[E_DIRE]
07:  [B_DIRE]#endif[E_DIRE]
08:
09:  [B_DIRE]#define {INT_CMP:ID_MC} [B_MCB]{({{{({v:ID_VF})} >= {0}})}[E_MCB][E_DIRE]
10:
11:  [B_DIRE]#ifdef {ABS_T:ID_MC}[E_DIRE]
12:  [B_FUNC][B_FD]ABS_T:ID_TP {[B_FR]{abs_:ID_VF##ABS_T:ID_TP}(ABS_T:ID_TP {v:ID_VF})[E_FR]}[E_FD] [B_ST]{
13:  [B_DIRE]#else[E_DIRE]
14:  [B_FUNC][B_FD]int:ID_TP {[B_FR]{abs_int:ID_VF}(int:ID_TP {v:ID_VF})[E_FR]}[E_FD] [B_ST]{
15:  [B_DIRE]#endif[E_DIRE]
16:  [B_DIRE]#ifdef {ABS_T:ID_MC}[E_DIRE]
17:    [B_ST]if {({[B_FR]{T_CMP:ID_MC}({v:ID_VF})[E_FR]})}
18:  [B_DIRE]#else[E_DIRE]
19:    [B_ST]if {({[B_FR]{INT_CMP:ID_MC}({v:ID_VF})[E_FR]})}
20:  [B_DIRE]#endif[E_DIRE]
21:      [B_ST]return {v:ID_VF};[E_ST]
22:    else
23:      [B_ST]return
24:  [B_DIRE]#ifdef {ABS_T:ID_MC}[E_DIRE]
25:        {[B_FR]{ABS_REV:ID_VF}({v:ID_VF})[E_FR]};[E_ST][E_ST][E_ST]
26:  [B_DIRE]#else[E_DIRE]
27:        [B_ST]{-{v:ID_VF}};[E_ST]
28:  [B_DIRE]#endif[E_DIRE]
29:  }[E_ST][E_FUNC][C_R][E_ST][E_FUNC]
30:
31:  [B_DE]int:ID_TP {test_var:ID_VF};[E_DE]
```

(b) 解析結果を埋め込んだコード例

図 1 強制的な構文エラーと括弧の数が合わないコードの例とその解析結果
Fig. 1 An example code of a forceful syntax error and an unbalanced brace, and the result of analysis.

令に対する構文木やコメントなどの情報をすべて融合した構文木，すなわち，前処理前のコードの表現を直接反映した構文木が必要である．また，前処理指令に関するリファクタリング [1] やアスペクトマイニング [2] などの前処理指令を中心とした保守作業も，単に前処理指令の構成だけでなく，指令の前後の C 言語の構文要素との関係に基づいて不吉な匂いを探したり，横断的関心事を特定する必要があり，前処理前のコードの表現を直接反映した構文木が不可欠である．

前処理前のコードを解析する既存研究では，主に次の 2 つの方法で前処理前コードの構文木，または，それに相当する情報を構成している．

- 前処理の分岐指令の各節を構成する部分木や，マクロの展開前後の部分木を 1 つの構文木に配置する [3], [4], [5], [6], [7], [8].
- 特定の前処理条件で適用した前処理の内容を解析し，前処理後の構文木と前処理前のテキストとの関係を管理する [9], [10].

前処理指令に関する解析や書き換えには前者の方法が適しているが，分岐指令の各節やマクロの展開内容が，それ単独で部分木を構成できることが前提となる．一般には，可読性や保守性を維持するために，分岐指令を文や宣言の間に配置し，マクロを閉じた式で構成するので，現実のコードの多くでは，この前提を満たすと考えられる [11]．しかし，数が少なくても，1 つでも条件を満たさない箇所が存在すれば，それを含むファイル全体が解析できない．たとえば，**図 1** (a) では，11 行目から 15 行目にかけて関数定義の開始部分が分岐指令によって分割されているが，この分割された部分は単独では部分木を構成できないので，全体としても構文木を構成できない．マクロの展開情報を得るために，インクルードファイルも含めて解析する場合には，解析できないヘッダファイルがあると，それをインクルードするすべてのファイルが解析不能になる．これを避けるために，構文木を構成する前に分岐指令の配置を変える方法もあるが [7], [8]，元のテキストとの乖離が起こる．さらに，既存研究では，マクロの参照に関する解析は行うものの，マクロ定義のそのものの構文解析は行わない．

本論文では，現実の前処理前コードに対する解析および書き換え支援を目的として，前処理指令の書き方に制約のない構文解析手法を提案する．この手法では，マクロ定義の内部も含め，前処理前コードの表現を直接反映した構文木を構成し，元のテキストとの乖離も起こらない．ただし，既存の手法と異なり，前処理指令を無視し，また，マクロも展開せずに構文木を構成するので，コードの持つ本来の意味や構造とは乖離する．このような不正確さは欠点ではあるが，分岐指令のリファクタリングなど，分岐指令を中心とした保守作業に役に立つ．また，コードの断片をパターンとする検索システムに応用した場合も，不正確さは必ずしも欠点とならない [12]．検索条件として，木構造の構成関係を直接記述するのではなく，検索条件として与えられたコード断片を構文解析し，そこから木構造に対するパターンを生成することで，コード断片に構文解析の結果が不正確になる記述を含んでいても，解析対象の該当箇所の部分木も同じ形になるので，パターンが適合する．

元のテキストとの乖離を許すのであれば，不正確さの解消も可能である．たとえば，前処理指令のリファクタリン

グを実施し，前処理指令で囲まれる箇所が部分木を構成できるようにすれば，分岐指令の位置が原因で生じる不正確さは減り，また，既存の解析手法が適用可能となることで，より深い解析も可能になる．なお，本手法では，パターン検索への応用を想定して，ソースファイルだけでなく，コード断片も解析対象に含める．また，編集途中の状態のコードは対象とせず，特定のコンパイル条件の下では構文木が構成できる状態にあるものとする．この場合，括弧については，基本的に組を構成し，例外として，マクロ定義の中で一方のみが多く存在する場合や，分岐指令によって複数の括弧を単一の括弧で受ける状態が存在しうる．

前処理指令を無視する解析では，本来，構文規則にはない表現も許容する必要がある．たとえば，図 1 (a) のコードは，12 行目と 14 行目の波括弧に対応する波括弧が 29 行目の 1 つしかないので，括弧の左右の数が合わず，また，強制的に構文エラーを引き起す自然文を含むといった問題がある．本論文では，オープンソースソフトウェアに対する解析実験などを通して見つけた解析時の障害となる問題を 8 つ取り上げ，その対処方法を示す．解析にあたっては，マクロ定義の内部や式の中に現れる文などに対して，部分的な解析が必要になることから island grammar [13] に基づく解析が有効である．本論文では，その実現に，字句系列に対する書き換え規則を用いる．書き換え規則を用いた場合，基本的な解析手順を定義したあとで，その規則とは独立に，特殊な事例に対処する規則を加えやすいという利点もある．実際に，本構文解析器の開発の過程では，C 言語の構文規則に基づいて定義をしたあと，現実のコードを解析するうえで必要な例外的な記述を処理する規則を加えて検証し，最終的に規則を整理・統合するといった手順を繰り返し，見通し良く作業を進めることができている．なお，我々は，すでに書き換え規則を一部に取り入れた解析手法を提案している [14]．また，文字列の書き換えを用いた ANSI C の解析器の実現可能性が Iterative Lexical Analysis（ILA）[15] で示されているが，いずれも現実の前処理コードを解析するときの問題点とその解決方法は示されていない．

提案する構文解析器の有効性を示すために，前処理指令を中心としたリファクタリング作業の支援の例として，本構文解析器を利用した前処理の分岐指令の移動ツールとマクロの逆置換ツールを示す．分岐指令の移動は，文や宣言の一部を囲う形で導入される分岐指令をそれらの境界に移動させる．たとえば，図 1 (a) は**図 2** に変換される．マクロ逆置換ツールは，各マクロの定義から書き換えパターンを生成し，定義の置換部分と合致する箇所をマクロ呼び出しに置き換えるツールである．マクロを定義したときの置き換え忘れの防止などに役立つ．なお，このツールについては提案済みであるが [16]，本手法の解析器を用いて再実装をしている．どちらのツールの実現にあたっても，本論文で提案する解析手法が不可欠である．すなわち，分岐指令の移動には，指令の位置を維持し，かつ，文や宣言の境界が正確な構文木が，また，マクロの逆置換にはマクロ定義の置換部分の部分木が必要であり，本論文の構文解析器はそのような構文木を生成する．

```
#include <stdio.h>

#ifdef ABS_T
#if (!defined(T_CMP) || !defined(T_REV))
Please define T_CMP and T_REV.
#endif
#endif

#define INT_CMP ((v) >= 0)

#ifdef ABS_T
ABS_T abs_##ABS_T(ABS_T v) {
  if (T_CMP(v))
    return v;
  else
    return
      ABS_REV(v);
}
#else
int abs_int(int v) {
  if (INT_CMP(v))
    return v;
  else
    return
      -v;
}
#endif


int test_var;
```

図 2 分岐指令を移動させたあとのコード
Fig. 2 A code whose branch directives are aligned.

以下では，2 章で書き換え規則を用いた構文解析手法の概要を述べたあと，前処理前コードの解析時の問題とその解決方法を示す．3 章では，応用例である 2 つのツールの実現方法を示す．4 章で，解析精度について評価を行い，問題点を考察する．5 章で関連研究を示し，6 章でまとめと今後課題を述べる．なお，説明にあたって書き換え規則は一部のみを示すが，http://tebasaki.jp/src から入手できる．

2. 書き換え規則を用いた構文解析器

2.1 属性付き字句系列とその書き換え

本論文で扱う構文木は字句系列で表現し，字句系列そのものは 1 行に 1 つの字句を記述したテキストで表現する．本解析器を用いる開発支援ツールは，その構文木を表すテキストを受け取り，検索や書き換えなどの処理を行う．検索や書き換えには，本解析器と同じ書き換え系を利用可能である．各字句は，字句解析によって型付けされ，属性値として型名を持つ．非終端記号は開始と終了を表す 2 つの「仮想字句」で表現し，そのテキストは空文字である．また，仮想字句の開始と終了，括弧の左と右のように組を構成する字句は，各組を区別する識別記号を属性値として持つ．書き換え規則はパターンマッチングの際に，適合条件に識別記号の一致を含めることで，正しい組合せに適合する．図 1 (b) は，解析結果を見やすくするために，図 1 (a) に仮想字句をコード中に埋め込んだ表現である．“[B_ST]” や “[E_ST]” などの角括弧で囲われた記号が仮想字句で，

表 1 演算子 * で結合した式に対する書き換えの例
Table 1 An example of rewriting an expression combined by operator *.

適用前:	ここまでに求めた式は _B_X と _E_X で囲われている [_B_X]a[_E_X] * [_B_X]b[_E_X] * [_B_X]c[_E_X]
ルール R1 適用 (1)	1 とつ目の演算子 * と右側の式を _B_OP03 と _E_OP03 で囲い、その式の _B_X と _E_X を B_P と E_P に変える [_B_X]a[_E_X] [_B_OP03]* [B_P]b[E_P][_E_OP03] * [_B_X]c[_E_X]
ルール R1 適用 (2)	2 つ目の演算子 * について同様に書き換える [_B_X]a[_E_X] [_B_OP03]* [B_P]b[E_P][_E_OP03] [_B_OP03]* [B_P]c[E_P][_E_OP03]
ルール R2 適用 (1)	変数 a を参照する式とその右横の _B_OP03 と _E_OP03 の組を結合して式にする [_B_X][B_P]a[E_P] * [B_P]b[E_P][_E_X] [_B_OP03]* [B_P]c[E_P][_E_OP03]
ルール R2 適用 (2)	式 a * b とその横の _B_OP03 と _E_OP03 の組を結合して式にする [_B_X][B_P][B_P]a[E_P] * [B_P]b[E_P][E_P] * [B_P]c[E_P][_E_X]

```
@OP03 => "OP\s+<[\*\/%]>\n"

# <確定式>:   B_P <式> E_P
# <未確定式>: _B_X <式> _E_X

# 規則 R1: <演算子> <未確定式> => _B_OP <演算子> <確定式> _E_OP
{
  $op:OP03 $sp:SP  $bx#1:_B_X $any:ANYEXPR $ex#1:_E_X
} =>> {
  ''#1:_B_OP03
    $op $sp $bx:B_P  $any $ex:E_P
  ''#1:_E_OP03
}

# 規則 R2: <未確定式>  _B_OP03 <演算子> <確定式> _E_OP03
#           => _B_X <確定式> <演算子> <確定式> _E_X
{
  $bx1#1:_B_X '(?>':X $any1:ANYEXPR $ex1#1:_E_X ')':X $sp1:SP
  $#2:_B_OP03  $op:OP03  $sp2:SP $any2:ANYEXPR  $#2:_E_OP03
} =>> {
  ''#1:_B_X
    $bx1:B_P $any1 $ex1:E_P $sp1 $op $sp2 $any2
  ''#1:_E_X
}
```

図 3 演算子 *，/，% の結合の規則
Fig. 3 The rule for combining operators *, /, and %.

型名の B_と E_は要素の開始と終了を意味する．また，下線付きの波括弧は仮想字句 B_P と E_P の簡略表現で，式の構成を表す．各識別子の後ろの “:ID_TP” などの記号は，その識別子の型を表す．直感的には XML で構文木を表現したものと同じであるが，書き換えの過程では開始または終了のどちらか一方の仮想字句を挿入するなど，整形式 XML では表現できない状態が存在するので，実装の都合上，XML を採用していない．

構文解析は主に字句の挿入と字句の型の変更で構成され，一時的な印として挿入した字句の削除や，複数の字句の結合も行う．**図 3** に，式の解析の書き換え規則の例を示す．この規則は，演算子 *，/，% で構成される式について，結合則に従って構文木を構成する．書き換え系は独自に実装したもの [14] で，規則は

{書き換え前パターン} => {書き換え後字句列}

の形式で記述する．書き換え前パターンには字句に適合する変数の並びを書く．変数の形式は「$名前：型」で，書き換え後に参照されない変数は名前を省略できる．書き換え後字句列には，書き換え後の字句の並び順に変数の参照「$名前」を書き，新規に字句を追加する場合には「’テキスト’:型」と記述する．仮想字句や括弧などの組に対しては，変数名の直後に同一の「#記号」を記述する．規則を

選択:	$[*name*: *alternative1* $| *alternative2* $] (*name* は省略可能)
繰り返し:	$[*name*: *elements* $] *quantifier* (*quantifier* は *, +, ?, *?, +?のいずれか)
字句結合:	*token1* $## *token2* (*token1* と *tokens2* の組み合わせが 1 つの識別子に適合)
文字列化:	$# *token* (*token* は文字列定数の内部に適合)

図 4 書き換え規則のメタ記号
Fig. 4 The meta symbols for rewrite rules.

再帰的に適用したい場合は，図 3 のように，=>の代わりに =>> を用いる．図 3 の 1 行目は，3 つの演算子 “*”，“/”，“%” の字句に適合する型 OP03 を文字列のパターンとして定義したものである．最初の規則は，演算子と右側の式を一時的な要素として識別し，2 番目の規則は，その要素と左側の式と結合させる．再帰的に書き換えるので，書き換えが停止するよう，未解析の式と解析済みの式は仮想字句を変えて区別する．**表 1** に，書き換えの過程の例を示す．なお，書き換え規則は，実際には書き換え可能な箇所を 1 度に書き換えるが，ここでは理解しやすくするために，各規則の書き換えを 2 回に分けて記述している．

書き換えに基づく解析では，実行性能が問題となりやすい．提案手法では，仮想字句は開始と終了の組で存在するが，これを単純に文字列の正規表現に変換して探索すると，検索エンジンはその組を見つけても，他の可能性を探索するので，無駄なバックトラックが発生する．よって，組を見つけた時点で探索が終るよう抑制する必要があり，図 3 の “'(?>':X” と “')':X” はその指示である．これは，実装に用いている Perl の正規表現のメタ記号をそのまま使用している．また，規則の構成によっても実行性能は変わる．特に左線形の構造に対してはバックトラックが発生しやすいので，できるだけそれを抑える構成にする必要がある．

属性付き字句系列および書き換え規則の詳細については，先行論文 [14] も参考にしていただきたい．なお，先行論文では，式レベルの解析をしていないので，括弧内を区切るカンマが括弧と同じ識別記号を属性値として持っていたが，本手法では式レベルの解析をしており，カンマは識別記号を持たない．また，実行速度の改善のために，字句の型名も簡略化している．書き換え規則については，**図 4**

表 2 現実の前処理コードの解析に対する問題 (A1) – (A4)
Table 2 The problems of analysis of real unpreprocessed code (A1) – (A4).

(A1)	問題: 括弧の組み合わせ不足: `#define LOOP_BEGIN while (0) {`
	対策: 対応する仮想括弧を挿入: `[B_DIRE]#define LOOP_BEGIN [B_MCB]while (0) {[C_R][E_MCB][E_DIRE]`

(A2)	問題: セミコロンのない文・宣言:
	`#define INITIALIZE(x) int x = 0;`
	`INITIALIZE(a)`
	`INITIALIZE(b)`
	対策: 文末に仮想セミコロンを挿入:

	`#define INITIALIZE(x) int x = 0;`
	`[B_ST]INITIALIZE(a)[SC][E_ST]`
	`[B_ST]INITIALIZE(b)[SC][E_ST]`

(A3)	問題: エラーを引き起す自然文:
	`#ifndef HAS_FUNC1`
	`This system requires FUNC1.`
	`#endif`

	対策: 1 つの字句に変換し、末尾に仮想セミコロンを挿入:
	`#ifndef HAS_FUNC1`
	`[B_ST]This system requires FUNC1.:ID_MSG[SC][E_ST]`
	`#endif`

(A4)	問題: 型や属性を表現するマクロ呼出し: `TYPE(x) func(ARG) { ... }`

	対策: 関数定義の構文を拡張:
	`[B_FUNC][B_FD]TYPE:ID_TP{({x})}`
	`[B_FR]func(ARG)[E_FR][E_FD] [B_ST]{ ...`
	`}[E_ST][E_FUNC]`

に示すメタ記号が追加されている．

2.2 現実の前処理前コードの解析に対する問題

本論文では，現実の前処理前コードの解析に対する問題として，以下の 8 つの問題を扱う．

(A1) 括弧の組み合わせ不足
(A2) セミコロンのない文・宣言
(A3) エラーを引き起す自然文
(A4) 型や属性を表現するマクロ呼び出し
(A5) 単独の else 節
(A6) マクロ定義の内部の解析
(A7) 識別子の定義不足
(A8) マクロの実引数内の文・宣言

これらは，主に Coreutils 8.22 [17] を対象に解析実験を行い，その過程で発見したものである．他のソースコードに存在する問題を網羅していない可能性はあるが，Coreuitls はコマンド用のコードのほかに，ライブラリやテストケースなど様々なコードが含まれているので，一般性があると考えている．また，以下の説明で用いる事例は，特に断わらない限り，Coreutils 8.22 を対象とする．

各問題について，以降の節で説明をする．各問題の典型例とその解決方法を簡単に記述したものを，(A1) から (A4) を**表 2** に，(A5) から (A8) を**表 3** に示し，説明にあたっては，これらを用いる．解決方法として示す解析結果は，表に示した典型例を実際に解析して得られた結果を用いて

表 3 現実の前処理コードの解析に対する問題 (A5) – (A8)
Table 3 The problems of analysis of real unpreprocessed code (A5) – (A8).

(A5)	問題: 単独の else 節:
	`if (c) { ... }`
	`#ifdef COND1`
	`else { ... } // if の一部`
	`#elif COND2`
	`else { ... } // 単体`

	対策: if と結合する else に印を付け、印のない else 節は単独の文として識別:
	(印の追加直後)
	`[B_ST]if (c) [B_ST]{ ... }[E_ST]`
	`#ifdef COND1`
	`else[_EL] [B_ST]{ ... }[E_ST][B_ST] // if の一部`
	`#elif COND2`
	`else [B_ST]{ ... }[E_ST] // 単体`
	(文としての識別後)
	`[B_ST]if (c) [B_ST]{ ... }[E_ST]`
	`#ifdef COND1`
	`else [B_ST]{ ... }[E_ST][B_ST] // if の一部`
	`#elif COND2`
	`[B_ST]else [B_ST]{ ... }[E_ST][E_ST] // 単体`

(A6)	問題: マクロ定義の内部の解析: `#define TEST(x) if (!cond(x)) err(x)`

	対策: 書き換え規則の適用と不足する仮想字句の補完:
	(補完前)
	`[B_DIRE]#define TEST(x) [B_MCB]if (!cond(x))`
	`[B_ST]err(x)[E_MCB][E_DIRE]`
	(補完後)
	`[B_DIRE] #define TEST(x) [B_MCB]if (!cond(x))`
	`[B_ST]err(x)[E_ST][E_MCB][E_DIRE]`

(A7)	問題: 識別子の定義不足:
	`a = (T)(n);`
	`b = (F)(n);`
	`p = (T*)x;`

	対策: 文脈から種別を判定:
	`a = {[B_CAST](T:ID_TP)[E_CAST]{({n})}};`
	`b = {[B_FR]{({F:ID_VF})}({n})[E_FR]};`
	`p = {[B_CAST](T:ID_TP*)[E_CAST]{x}};`

(A8)	問題: マクロの実引数内の文・宣言: `REPEAT(NUM, ++i; a[i] = f(i), c);`

	対策: 関数呼び出しとして識別されない箇所を補正:
	(補正前)
	`[B_ST]{REPEAT}{({NUM}, [B_ST]{++i};[E_ST][B_ST]{a[i] = f(i)}, {c})};[E_ST]`
	(補正後)
	`[B_ST]{[B_FR]{REPEAT}({NUM}, [B_ST]{++i};[E_ST]`
	`[B_ST]{a[i] = f(i)}[E_ST],{c})[E_FR]};[E_ST]`

いるが，説明のうえで不要な仮想字句や識別子の型の表記は省略し，特に注目すべき仮想字句と識別子の型については，やや大きめのフォントで表現している．

解析器は，これらの問題に対処しつつ構文解析を行う．全体の流れを**図 5** に示す．全体は P1 から P8 のフィルタで構成される．P1 で字句系列に変換したあと，P2 で前処理指令を識別し，以降の処理で空白と同様に無視できるようにする．P3 と P4 で文レベルの構造を正しくとらえるための準備となる補正を行い，P5 から P7 で構文解析を行う．最後に，P8 で典型的な解析の誤りを補正する．P1 と P3 以外は，基本的に書き換え規則で定義されている．ただし，書き換え規則では実現できない処理や十分な実行性

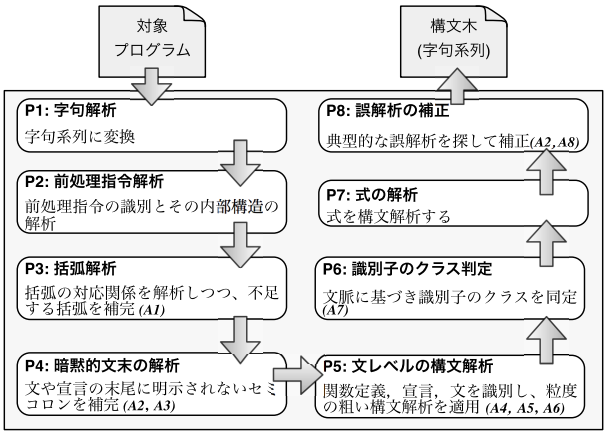

図 5 構文解析の全体の流れ
Fig. 5 The parsing process of the parser.

能を得られない規則は，補助的な専用フィルタに置き換えている．各フィルタの括弧内の記号は，そのフィルタで解決した問題を表す．以下，各問題とその対処方法について説明する．なお，文と宣言は構文木上は兄弟の位置関係で混在して出現し，同等に扱う方が書き換え規則が簡潔になるので，解析の途中では区別せず，最終段階で区別している．よって，本論文の説明では特に必要がない限り文と宣言を同一の要素として扱う．

(A1) 括弧の組み合わせ不足

マクロ定義の中に括弧が片側だけ出現する場合や，図 1 の 11 行目から 15 行目のように，前処理の分岐指令によって左右の一方が多くなる場合，単純な手法で括弧の組合せを求めるとズレが生じる．前者は，制御文を抽象化する場合に用いられ，たとえば，lib/msvc-inval.h に多く出現する．後者は，エディタで自動的にインデントを入れるときにズレが生じるなど，問題が生じやすい分岐指令の使い方の典型例であるが，Coreutils 8.22 には存在しない．また，特定の前処理条件で有効になるマクロ[*1]で，その定義内の左括弧の数が多いものがあり，これはテストされずに残ったバグと想定される．

括弧の組合せにズレがあると，式や複合文の範囲の識別を誤るので，仮想的な括弧を挿入してズレを解消する．仮想的な括弧とは，型は括弧の字句と同じで，空文字を持つものである．補完の過程では，字句系列を走査しながら，スタックを利用して括弧の対応を確認し，不一致が生じた場合には対応する括弧を挿入する．ただし，マクロ定義と分岐指令に関して，次のように対処する．

- マクロ定義は，その中で組合せが閉じるよう，定義内から走査が出るときに，不足分を挿入する．
- スタックに分岐指令のネストの深さも記録し，深い位置のある括弧を浅い位置にある括弧で受ける場合，分岐数に応じて数が合うように括弧を挿入する．

たとえば，表 2 の (A1) に示した例は，マクロ定義に左括弧のみが存在し，右括弧が存在しないので，仮想の右括弧を挿入する．“[C_R]” は，補完された右波括弧を意味する．図 1 では，関数定義の内部の最初の左括弧が 12 行目と 14 行目に合わせて 2 つ記述されているのに対し，対応する右括弧は 1 つである．この場合，左括弧は分岐指令の中に入っており，右括弧は分岐指令に含まれていないので，ネストの深さが異なる．同じ深さの左括弧は 2 つであるので，右括弧が 1 つ不足していると判定し，仮想括弧を挿入する．図 1 (b) の 29 行目の “[C_R]” は，補完された右波括弧を意味する．

[*1] LDBL80_WORDS in gnulib-tests/test-{isnanl,vasprintf-posix}.h

```
@SPR  => "(?:SP_[BC].*+\n)*+"
@LINETOP => "(?:(?:UNIT_BEGIN|SP_NL)\s.*+\n)"
@INVALID1 => "(?:ID_TP|CT|RE_|C_R|UNIT).*+\n"

# Ex1. abc()\n\n => abc();\n\n              # 次の行が空行の場合
# Ex2. abc()\n int a; => abc();\n int a;   # 次の行が宣言の場合
{ $top:LINETOP $sp0:SPR $name:IDN $sp1:SPR
    $pl#1:P_L '(?>':X $arg:ANY $pr#1:P_R ')':X $sp2:SPR $nl:SP_NL
  $sp3:SPR $[lines: $:INVALID1 $| $[: $:_DIRE $:SPR $]? $:SP_NL
    $sp4:SPR $[: $:INVALID1 $| $[: $:_DIRE $:SPR $]? $:SP_NL $] $] }
=>> { $top $sp0 $name $sp1 $pl $arg $pr '':SC $sp2 $nl $sp3 $lines }
```

図 6 仮想セミコロンの補完規則の例
Fig. 6 A rule for complementing virtual semicolons.

(A2) セミコロンのない文・宣言

変数の初期化などを行うマクロ定義が，それ単体で完全な文の構造を持つ場合，そのマクロ呼び出しの後ろにはセミコロンが付かず，文の終端の位置は明示されない．これをそのまま構文解析すると，次の文と結合した構造になる．マクロ呼び出しが単体の文なのか，そのあとに続く文の一部なのかを判定するには，マクロの展開内容の確認が必要であるが，ファイルを単体で解析する場合，そのマクロ定義が存在するとは限らない．一方，現実には，展開内容を確認しなくてもプログラマが正しく理解できるものも多い．そこで，次のヒューリスティックな知識を利用する．

- マクロ名は大文字で構成される．
- マクロ呼び出しは 1 行に 1 つのみ記述される．
- マクロ呼び出しのあとに空行や前処理指令が続く．
- マクロ呼び出しの次の行は型や制御文など，文の開始を表す字句で始まる．

これらのうち複数の条件を組み合わせて文末を判定し，そこに仮想のセミコロンを挿入する．Coreutils に含まれる事例が，必ずしもすべての条件を満たすとは限らないので，条件の組合せは，実際に解析結果を確認して決定した．よって，他のコードに適用したときに誤判定をする可能性がある．仮想のセミコロンとは型はセミコロンで空文字を持つ字句である．表 2 の (A2) の例で，各マクロ呼び出しの直後に挿入されている “[SC]” が仮想セミコロンを表し，これにより各呼び出しを単体の文として識別する．

例として，引数付きマクロ呼び出しに対する書き換え規則を図 6 に示す．この規則では，マクロ名は大文字には限定していない．補正は主にフィルタ P4 で行うが，P4 の段

```
# 自然文を _B_EM と _E_EM で囲い，その後ろに仮想セミコロンを挿入する規則
#  Ex. "invalid UTIME macros" in gnulib-tests/test-sys_stat.c
#  Ex. "missing or broken *_FILENO macros" in gnulib-tests/test-unistd.c
#  Ex. "you lose .... " in /lib/alloca.c
{ $[if: $:B_DIRE $:PRE_TOP $:SPR $:PRE_DIR/ifn?(?:def)?/ '(?>':X $:ANY $:E_DIRE ')':X $:SPR $:SP_NL $]
  $sp1:SPR $[msg: $:IDN $:SP $:/IDN/ $[: $:SP $:/IDN|OP/ $]+ $:SPR $] $nl:SP_NL
  $[sp2: $:/SP_\w+/ $]*
  $[end1: $:SPR $:B_DIRE $:PRE_TOP $:SPR $:PRE_DIR/endif/ $:SPR $:E_DIRE $] $nl1:SP_NL
  $[end2: $:SPR $:B_DIRE $:PRE_TOP $:SPR $:PRE_DIR/endif/ $:SPR $:E_DIRE $]? $nl2:SP_NL
} => { $if $sp1 '':_B_EM $msg '':_E_EM '':SC $nl $sp2 $end1 $nl1 $end2 $nl2 }

# 自然文内の字句を結合する規則
{ $b:_B_EM $x:/[^_]\w+/ $y:/[^_]\w+/ } =>> { $b $x $## $y }
{ $b:_B_EM $x:/[^_]\w+/ $e:_E_EM } => { $x:ID_MSG }
```

図 7 エラーを引き起す自然文に対する補完規則の例
Fig. 7 A rule for identifying natural sentences that forcefully cause errors.

階で誤判定が起きやすい条件については，式のレベルまでの解析結果が得られる P8 で補正する．具体的には，小文字の引数付きマクロ呼び出しは，1 行に 1 つだけ書かれた関数呼び出しと区別できない場合があるので，P8 で，連続するマクロ呼び出しが 1 つの文を構成する場合にはそれぞれを単体の文に分割する[*2]．なお，補正を後のフィルタで行うと，処理が複雑になりやすく，また，実行効率の低下を招きやすいので，可能な限り避けることが望ましい．また，この方法は完全ではなく，誤ってセミコロンを挿入する場合や，挿入できない場合がありうる．

(A3) エラーを引き起す自然文

一般に，機能が不足する環境ではコンパイルできないように，`#error` によってエラーを引き起す方法が用いられるが，自然文がそのまま書かれているコードも存在する[*3]．コンパイルエラーになるように記述するので，どのように記述されているかは想定し難いが，(A2) と同様にヒューリスティックな知識で判別する．我々は次の条件を組み合わせて判別を行った．

- 複数の単語の並び，または，文字列である．
- `#error` の直後に存在する．
- 分岐指令で囲われている．
- 分岐指令のあとに空行が続く．

自然文の後ろには (A2) と同様に仮想的なセミコロンを入れることで文として扱い，後続の要素と結合を防ぐ．また，自然文を構成する単語は字句解析の時点で分解されるが，式としての解析を避けるために，すべての字句を結合して単体の字句に変換する．**図 7** は，分岐指令に囲われ，そのあとに空行が続く自然文を探す規則の例である．自然文は，複数の識別子が並んだものとして判定している．また，発見した自然文は仮想字句 _B_EM と _E_EM で囲い，後半の 2 つの規則でその内容を結合して 1 つの ID_MSG 型の字句にする．表 2 の (A3) の例において，“[SC]” は，補完されたセミコロンを表し，その直前の “:ID_MSG” は自然文全体が 1 つの ID_MSG 型の字句になっていることを意味する．この補完により，この自然文全体を 1 つの文として識別する．図 1 の 5 行目にも自然文が記述されているが，同様に解析されている．

(A4) 型や属性を表現するマクロ呼び出し

C 言語では同じ型でも環境によって定義が変わる場合があり，マクロによる抽象化が用いられる．たとえば，関数定義では，型の部分に型と属性に対応するマクロ呼び出しが複数存在する場合がある．そのような関数定義を解析するには，構文規則どおりには解析できず，判定する条件を緩める必要がある．そこで，型の部分に引数をともなう識別子を許すように規則を拡張する．ただし，そのマクロ呼び出しが何であるかは決定できないので，マクロ呼び出しは関数呼び出し，または，後ろに括弧をともなう型として解析する．表 2 の (A4) に示す例では，識別子 `TYPE` を型と判定し，また，識別子 `func` を関数名と識別している．ここで，仮想字句 B_FD と E_FD は，関数定義の頭の部分，すなわち，返り値の型，関数名および仮引数の組合せを表す．また，仮想字句 B_FR と E_FR は，関数名と仮引数または実引数の組合せを表す．これらの字句の挿入から関数定義の構造が正しく識別されていることが分かる．なお，一般的には，識別子 `TYPE` は引数付きマクロである可能性が高い．関数定義は先頭に型が記述されるので，本解析器では，識別するうえで型として扱っているが，マクロの参照関係の解析などに応用するときには，引数をともなう型をマクロ呼び出しに補正することも必要となる．

(A5) 単独の else 節

制御構造の中で，if 文のみが 2 つの文を持ちうるが，分岐指令で文を選択することで，見かけ上，3 つ以上の文を持つ場合がある．その場合，分岐指令を無視して解析すると，if 文に含まれない else 節が生じる．この孤立した else 節を単体の else 文として扱う．構文上は正しくないが，3.1 節で示す分岐指令を if 文の外へ移動させる処理を適用すれば正しい状態になる．解析時は，正しく結合する else 節を誤って単独の else 節と識別しないよう，まず単独の else 節が存在しないことを前提に解析する．if 文の一部となった else

[*2] lib/siglist.h が典型例である．
[*3] lib/alloca.c には “Using #error here is not wise since this file should work for old and obscure compilers.” と書かれている．

には一時的な印となる字句を追加し，その印がない else 節を単体の文として識別する．表 3 の (A5) に示す解析結果の例は，この識別時の中間状態を表す．仮想字句 “[_EL]” は，if の真節と結びついた else 節の予約語 else の直後に挿入される．この仮想字句が存在しない else 節を，仮想字句 B_ST と E_ST で囲い，単体の文として識別し，仮想字句 “[_EL]” を削除する．

(A6) マクロ定義の内部の解析

本解析器では，フィルタの P2 でマクロ定義の構造を解析するが，これは前処理指令としての構造であり，マクロ定義の内容となる置換部分は解析しない．置換部分の構造は P5 から P7 で解析する．置換部分は定数や簡単な式であることが多いが，文，あるいは文の一部としての定義もありうる．書き換え規則は置換部分にも適用されるので，特別な手段を用いることなく解析される．ただし，文を特徴付ける制御文の予約語や文末のセミコロンを頼りに文を識別するので，表 3 に示した最後のセミコロンがない if 文など，不完全な文に対しては，文の仮想字句が組にならないなど，木構造を構成できない中途半端な解析結果になる．

一方で，中途半端な解析結果はこのような場合に限定されるので，補正可能である．ここでの中途半端な状態とは，置換部分の先頭または最後尾で，文の開始または終了を表す仮想字句が抜けている状態である．本解析器は，P5 の文の解析から始めるが，制御文の入れ子を解析する前に，入れ子の最外の文，式文，複合文の 3 種類を求める．この段階では，複合文には必ず仮想字句が組となって挿入され，識別される文と文の間には，終了と開始の仮想字句が挿入されている．もし置換部分で最終的に不足するとすれば，先頭または最後尾での 1 個のみとなるので，ズレた組合せに対して仮想字句を補完すればよい．表 3 の (A5) に示す解析結果の例は，この補完の前後を表す．仮想字句 “[B_ST]” が挿入されているが，文末を表すセミコロンが存在しないので，対応する “[E_ST]” が存在せず，仮想字句の対応関係を前の方から求めていくと，マクロ定義の終わりを表す “[E_MCB]” が対応する．補完は，この “[E_MCB]” の直前に “[E_ST]” を入れることで正しい対応関係にする．

(A7) 識別子の定義不足

識別子は異なる名前空間に属するものや，同一空間内で意味が異なるものがあり，ここではその種別をクラスと呼ぶ．種別クラスは，「タグ」，「ラベル」，「メンバ」，「変数」，「型」，「マクロ」である．なお，関数名は「変数」に含む．構文解析を達成するには種別クラスの区別が必要であり，たとえば，表 3 の (A7) の例の 1 行目と 2 行目に示すように，同じ形の 2 つの式であっても，キャストと関数呼び出しの場合があり，その構文木の構造は異なる．種別クラスのうち，タグ，ラベル，メンバは前後の字句で区別でき，マクロもマクロ定義内のマクロ名に限定することで特定できるが，変数と型は宣言などの文脈に基づいて区別する必要がある．なお，変数と型は同一の名前空間に属し，同名の識別子でもスコープによって種別クラスが異なることがある．ただし，同名の識別子を型と変数の両方で使うと可読性が下がり，一般的には推奨されない書き方であるので，本論文では，解析を簡単にするために，同名の識別子はどちらか一方にしか使われないとの前提をおいた．ただし，評価実験などの中で例外[*4]を確認しており，改善が必要なことが分かっている．

型と変数の区別は，ファイル内に宣言があれば，それに基づくべきである．そこで，解析器はフィルタ P5 で，まず文レベルまでの構造を解析し，宣言と想定される文脈に出現する識別子を型と判定する．続いて，sizeof(T *) の T のように型しか使えない文脈と，型と関係しない演算子の前後など，変数（または関数）のみが許される文脈に基づいて区別する．最後に，識別子の各出現のクラスの判定状況を集計し，識別子のクラスを決める．表 3 の (A7) の例の場合は，3 行目はキャスト演算子として解釈する以外に解釈の方法がないので，この記述から識別子 T を型と判定し，1 行目もキャストとして解析する．一方，識別子 F については，特定する文脈がない．理想的には，必ずどちらか一方になるはずであるが，同名の識別子が変数と型の両方の文脈に出現することで 2 つの可能性が生じた場合や，いずれの文脈にも出現せず判別できない場合には，変数として処理する．これは，一般的には，型と変数では変数の方が多く出現するので，確率的に高いと想定したためであるが，必ずしも正しい結果になるとは限らない．

(A8) マクロの実引数内の文・宣言

文がマクロの実引数として用いられる場合があり，その実引数は式ではなく文として識別する必要がある．書き換え規則を用いた場合，文はその出現位置に関係なく識別されるが，(A6) と同様に文を正しく識別できない場合がある．すなわち，開始または終了の位置に仮想字句が存在しない状態になる．マクロ呼び出しは見かけ上，関数呼び出しと同じであるので，区別を付けずに解析する．ただし，すべての関数呼び出しに対して仮想字句の不足を調べる処理を適用すると，実行効率が悪い．そこで，関数呼び出しは純粋に式で構成されるもののみを識別する．形は関数呼び出しだが，関数呼び出しとしては識別されていない箇所は文を含むマクロ呼び出しであるので，仮想字句の不足を調べ，必要に応じて補完する．表 3 の (A8) の例は補正の前後の状態を表す．関数呼び出しは，関数名と引数の組合せを仮想字句 B_FR と E_FR で囲ったものとして表現されるが，この場合は，引数内に式の構成要素にならない仮想字句 B_ST と E_ST が含まれるので，関数呼び出しとしては識別されない．また，2 番目の実引数の最後に文末を表

[*4] マクロ定義の引数は型や式に同名のものが使われることがある．また，PostgreSQL 9.3.5 [18] の contrib/oid2name/oid2name.c では eary 型の仮引数 eary が定義されている．

すセミコロンが記述されていないので，文の終わりを表す E_ST が不足している．補正後は，全体を関数呼び出しの構造に書き換え，また，第 2 引数の末尾に仮想字句 E_ST が挿入され，正しい組合せを構成している．

その他の問題

実際には，表 2 および表 3 の問題以外にも，namespace などの C++に関する記述や “@” などの記号を含む識別子など，C 言語の規格から外れた記述への対処が必要である．ただし，構文の追加や識別子の字句定義の変更など，拡張自体が容易であるので，本論文では議論しない．

3. 前処理に関する書き換え支援への応用

3.1 分岐指令の移動

提案手法では，前処理指令を無視するので，else 節を持つ前処理の分岐が文または式の一部を囲うと，構成要素の範囲や構成関係を正しく解析できず，演算子で結合されない式や，不正な文の入れ子関係などが生じる．一方，文はリストを構成するので，その境界に分岐指令を配置した場合，要素間の構成関係は正しく解析される．すなわち，本手法では正確な構文木を生成できないことが欠点であるが，分岐指令を移動させてから解析することで，より正確な構文木を得られ，欠点を補える．

分岐指令を境界に移動するには，境界の位置を正しく識別できているかが重要になる．本論文の解析器は，要素間の入れ子関係が不正確な場合があるが，境界はほぼ正確に求まる．たとえば，図 1 では，2 つの関数が多重化されており，本解析器では，関数の入れ子として解析している[*5]．このような場合，正しい木構造の定義自体も困難であるが，少なくとも関数の入れ子は正しくない．一方，関数定義の境界の位置については正しく解析されている．文や関数定義などの境界の位置は，文末を示すセミコロンや波括弧によって決まり，配列の初期化式で用いられる波括弧など例外となるものを除くことで，正確に求まる．ただし，問題の (A2) のように明示的に文末が示されていない場合には正しく求まらない可能性がある．

以下では，条件分岐は`#if`，`#else`，`#endif` の 3 つの指令で構成されたもののみを扱う．これ以外のものは，これと等価な形に変換して処理する．実際の実装では，自然な形での変換を実現するために，移動しなかった箇所は元に戻し，移動後に`#elif` と等価な形になる場合には`#elif` に置き換える処理を行っている．また，文の境界のみを用いて説明するが，実際には関数定義の境界も対象になる．

移動対象の指令には，あらかじめ印が付けられているものとする．我々の実装では，印はコメントで記述する．また，すべての分岐指令を対象とするオプションも用意している．なお，文の境界に位置する指令は，印が付けられても移動しない．

[*5] 関数定義は `[B_FUNC]` と `[E_FUNC]` で囲まれた範囲．

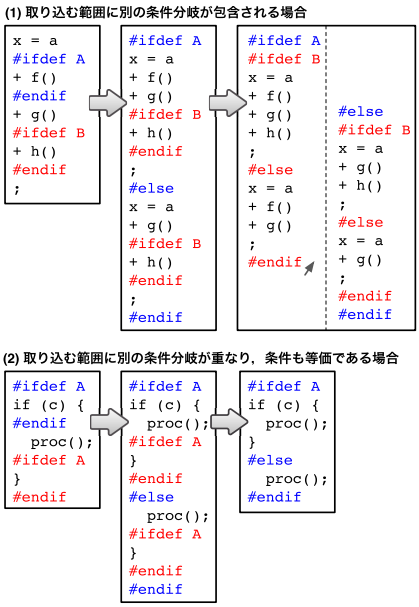

図 8 複数の条件分岐の組合せの例
Fig. 8 Examples of combinations of branch directives.

分岐指令の移動は，移動するべき指令がなくなるまで，以下を繰り返すことで実現する．

(1) 条件分岐の入れ子の関係を解析し，1 つの条件分岐を構成する 3 つ指令の組には同じ識別記号を割り振る．
(2) 印が付加された各指令の組について，条件分岐内に取り込むべきテキスト（字句系列）の範囲を求める．
(3) 印が付加された各指令の組のうち，次の 3 つのものを移動対象から除外する．a) 取り込むテキストが存在しないもの（つまり，境界に位置するもの），b) 他の組の入れ子になっているもの，c) 取り込むべきテキストの範囲がそれより前に出現する他の組と重なるもの．
(4) 移動対象となった指令の組について，条件分岐内の真節と偽節にそれぞれ境界までのテキストを取り込む．
(5) コード全体の構文解析を行う．

条件分岐内に取り込むテキストの範囲は，文の先頭から`#ifdef` の直前までと，`#endif` の直後から文の末尾までの間である．対象となる文は次のように決める．

- 真節の中に終了（開始）位置が存在するが，節内で閉じていない文がある場合には，その文の開始（終了）位置を求める．複数の文の入れ子が閉じていない場合には，最外の文を対象とする．
- 節内で閉じていない文がない場合には，節を包含する最内の文の開始（終了）位置を求める．

ただし，求めた文の開始位置が，他の指令の組の中に存在する場合には，文の開始（終了）位置の代わりに，その指令の組の開始（終了）位置を用いる．**図 8** に，複数の条件分岐の組合せの例を示す．また，図 2 は，図 1 (a) にツールを適用した結果である．

指令の移動では，求めた範囲のテキストを真節と偽節に複製し，元の範囲のテキストは削除する．ただし，複製時に分岐条件が移動対象の指令の分岐条件と等価または否定である指令の組を含む場合，その指令の組と矛盾する条件下のテキストを削除する．図 8 の (2) にその例を示す．中央のコード例のように，矛盾した条件分岐を残すと，括弧の数が合わないなどの問題が発生する可能性がある．なお，条件分岐の等価性を正確に判定することは難しいので，実装では，空白の除去などの簡単な正規化を適用したテキストを比較している．近似的な方法ではあるが，適用実験では，移動による矛盾は生じなかったので，実用性があると判断している．

複製されるテキストは，実際には，構文木を構成する字句の部分列である．文などの非終端記号は，同じ識別記号を持つ 2 つの仮想字句で表現されるが，本来，組となるべき各条件分岐内の仮想字句と複製内の仮想字句が同じ記号を持つとは限らない．また，親子関係として解析したものが兄弟関係に変わるといった構成関係の変化も起こりえ，仮想字句の配置の変更も必要になる．そこで，移動後は全体を構文解析し直す．

3.2 マクロの逆置換

リファクタリングの過程で，複数の同じ構造の式を見つけ，マクロとして定義することは，一般的に用いられる手法である．しかし，マクロを定義しても，該当箇所をすべてマクロ呼び出しに置き換える作業は容易ではない．本論文では，このマクロ呼び出しに置き換える作業を逆置換と呼ぶ [16]．定義部分を正規表現として書いて検索する方法もあるが，完全なものは記述し難く，また複雑になりやすい．マクロの逆置換を実現するには，マクロ定義の内部に対する構文木をパターンと見なして，前処理前のコードの構文木に適用し，適合する部分を置き換える仕組みが必要である．本手法では，マクロ定義内も構文解析しており，部分木が得られる．また，先行論文 [14] において，パターン変換手法を提案しており，マクロ定義をそのパターン変換の記述に変換するのみで，簡単に逆置換を実現できる．なお，マクロ定義には，“do ... while(0)” で囲うなど特有な表現があり，パターンに変換する際には，それらを除去する必要があるが，マクロ定義自体が構文木になっているので，これもパターン変換を用いて除去ができる．

ツールは以下の手順でコードの書き換えパターンを生成し，それを適用して逆置換を行う．

(1) 逆置換するマクロ定義を構文解析し，名前，引数，置換部分を取り出す．
(2) 置換部分からマクロ特有の表現を除去する書き換え規則を適用する．
(3) 置換部分内の引数をパターン変数に置き換える．
(4) 置換部分からマクロ呼び出しに置き換えるパターン記述を生成する．

```
定義: #define ADD(a, b) ((a) + (b))
定義から生成される変換パターン:
  %ex
  %before
     ${a:EXPR} + ${b:EXPR}
  %after
     ADD(${a}, ${b})
  %end
変換パターンから生成される書換え規則:
{$t01#E06:B_P $t02#B01:P_L/\(/ $t03:SP $t04#E07:B_P $t05#E08:B_P
 $a:EXPR $t06#E08:E_P $t07:SP $t08:OP/\+/ $t09:SP $t10#E09:B_P
 $b:EXPR $t11#E09:E_P $t12#E07:E_P $t13:SP $t14#B01:P_R/\)/
 $t15#E06:E_P $t16:SP $t17#B02:C_L $t18:SP $t19#B02:C_R/}/}
=> { '':B_P '':B_FR '':B_P 'ADD':ID_VF '':E_P '(':P_L '':B_P $a
     '':E_P ',':CA '':B_P $b '':E_P ')':P_R '':E_FR '':E_P }
```

図 9 マクロの逆置換の例
Fig. 9 An example of reverse macro expansion.

図 9 に，ツールが生成するパターン記述および書き換え規則の例を示す．パターン記述は “%before” の直後が置換部分で，“%after” がマクロ呼び出しである．この例では，マクロ特有の表現として，引数を囲う括弧が除去されている．“%ex” は，パターンが式であることを意味し，これがない場合には，文と見なされる．文と式の違いは，全体を囲う仮想字句 B_ST と E_ST の有無であるが，セミコロンのない文もありうるので，このような区別が必要である．パターン記述は書き換え規則に変換されて適用される．

なお，1 つのマクロの引数が複数参照される場合，各参照に該当する箇所に同じ式が存在する必要があるが，実装では部分木の等価性を判定する仕組みがなく，パターンが適合しない．これは，字句系列を文字列として等価性を判定しており，まったく同じ構造の部分木でも，仮想字句が持つ識別記号が異なるためである．仮想字句を含まない単体の変数や定数の場合には正しく適合できる．正しく適合する例は 4.1 節で示す．また，マクロ定義から生成すべきパターンは表現の差異を考慮して複数構成すべきである [16]．本論文の再実装では実現していないが，置換部分は構文木になっているので，マクロ特有の表現を除去する書き換えと同様に，表現を変更する書き換えを適用すればよい．

4. 評価と考察

実装には，書き換え規則以外には Perl を使用した．書き換え系も Perl で実装されており，内部で文字列の正規表現に変換して書き換えを適用する．実行時間の評価として，Coreutils 8.22 に対して適用実験を行った．対象ファイルはすべての *.c と *.h で，合計で 1,169 個である．解析の実行時間を図 10 に示す．横軸がファイルの行数を，縦軸が時間（秒）を表す．平均時間は，ファイルあたり 5.0×10^{-1} 秒，行あたり 2.2×10^{-3} 秒である[*6]．図中の実線は最小二乗法で求めた近似直線で，傾きは 1.8×10^{-3} 秒/行である．書き換え規則を用いているが，実用的な時間に収まって

[*6] Intel Xeon 2.27 GHz, 4 GB, FreeBSD 9.1R, Perl v5.16.3

表 4 評価結果の概要
Table 4 A summary of the evaluation results.

問題点	評価内容	Coreutils	PostgreSQL
(A1) 木構造の構成	ファイル数	1169	1695
	構成の失敗	0	1
	仮想字句の残留	0	8
(A2) セミコロンのない文・宣言の識別	識別数	939	958
	誤識別 (合計)	5	163
	内訳: 配列の初期化式	5	70
	内訳: 独自の制御文	0	93
(A3) エラー文の識別	識別数	22	0
	誤識別	0	0
	未識別	0	0
(A4) 関数	識別数	3300	16730
	誤識別	0	14
	未識別	0	0
(A5) 単体の else 文	else の総数	2691	12490
	単体の else 文	6	10
	誤識別	0	0
	未識別	0	0
(A6) マクロ定義の解析	一時仮想字句の残留	0	0
(A7) 型識別子の判定	名前の最後が_t である識別子の数	1595	732
	内訳: 型と判定した数	1409 (88.3%)	658 (89.9%)
	内訳: 変数と判定した数	186 (11.7%)	74 (10.1%)
	型または変数と判定した識別子	30000	132208
	内訳: 出現に型と変数の両方が存在した識別子	17 (0.1%)	125 (0.1%)
	内訳: 出現から一方に決まった識別子	29983 (99.9%)	132083 (99.9%)
(A8) 実引数に文を持つマクロ呼び出し	対象のマクロ呼び出し	66	5
	誤識別	0	2
	未識別	0	0

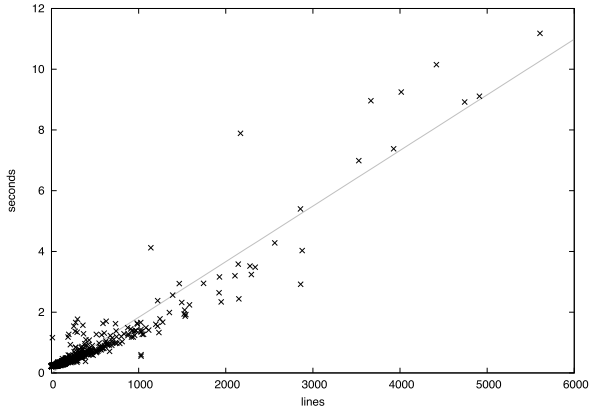

図 10 解析実行時間
Fig. 10 Lines and analysis time.

いる．

精度の評価では，まず，現実のコードの解析時に障害となる 8 つの項目（表 2，表 3）について，Coreutils 8.22 を対象に検証する．ただし，各項目への対処方法は Coreutils 8.22 の解析結果に基づいて調整されているので，他のコードへの適用可能性を確認するために，PostgreSQL 9.3.5 への適用も行う．また，前処理命令の移動ツールとマクロの逆置換ツールの評価を行う．なお，比較となる正しい解析結果が存在しないので，必ずしも正確な評価は行えない．

4.1 現実の前処理前コードを解析するときの問題

本解析器を，Coreutils 8.22 および PostgreSQL 9.3.5 に適用した結果の概要を**表 4** に示す．解析精度は，全体的に同じ傾向になったが，PostgreSQL に特有の問題で精度が落ちている部分がある．以下では，まず，Coreutils の解析結果について説明し，そのあと，PostgreSQL の解析結果における特徴的な差について説明する．

4.1.1 Coreutils に対する適用

(A1) は括弧の組合せの問題であるが，問題を拡張し，仮想字句も含め木構造が正しく構成されているかを調べた．スタックを用いて括弧と仮想字句の組合せの正しさを調べるツールを作成し，解析結果に適用したところ，組合せのズレは見つからなかった．これにより，解析結果は正当な木構造を構成していると判断できる．また，複数の書き換え規則を用いて 1 つの構文要素を抽出する際には，図 3 の仮想字句_B_OP03 のように，一時的な仮想字句を用いて規則間で字句の位置や範囲を共有するので，構文要素の抽出に失敗した場合には，この仮想字句が残留する．Coreutils の解析結果には，一時的な仮想字句の残留はなく，書き換え規則は，ほぼ意図どおりに適用されていると見なせる．ただし，解析器の開発時に Coreutils を検証対象として一時的な仮想字句が存在しなくなるようデバッグをしているので，この結果自体に意味はなく，後述する PostgreSQL の評価において意味を持つ．

(A2) および (A3) は，補完された仮想のセミコロンを調べた．全体で，961 個が補完されており，そのうち，(A2) が 939 個（111 個のファイル[*7]）で，(A3) が 22 個（10 個のファイル）であった．(A2) で補完されたセミコロンのうち，5 個については，配列の初期化式の最後に記述されたマクロ呼び出しを誤って判定していた．文と識別するなどの構造的な誤りは生じてはおらず，削除する規則を追加す

[*7] src/primes.h で 668 個，lib/siglist.h で 34 個が補完されており，これを除くと 259 個（103 個のファイル）である．

表 5 連続する式の並びの内訳
Table 5 The numbers of continuous expressions.

原因	対処箇所数	ファイル数
識別子の区別による誤り	25	12
マクロを用いた文字列の連結	207	5
宣言内の型や属性を参照するマクロ	1304	178
合計	1536	196

ることで容易に回避できる．ただ，根本的な原因は，複合文の波括弧と初期化式の波括弧の区別をしないで規則を書いていることにあるので，その改善が必要である．また，複数のファイルで確認した範囲では，判定の漏れを見つけられなかったが，マクロ参照の前後に何らかの字句が配置されていると判定できないなど，反例を容易に作成できるので，判定の漏れが存在する可能性がある．(A3) については，正しく解析されない場合，複数の識別子の並びとして解析され，その結果，演算子で結合されない式の並びが生じる可能性が高い．そこで，そのような式の並びを検索したところ，**表 5** のようになった．これらにエラーを引き起す文は含まれていないことから，(A3) については，未識別はなく，すべて正しく識別できたと判断できる．

(A4) については，マクロ呼び出しが原因で関数として識別できない場合，前述の演算子で結合されない式の並びが生じる可能性が高い．しかし，表 5 に示すように，該当するものなく，識別できない関数は発見できなかった．また，複雑な事例として，lib/wctype.in.h には，関数名が分岐指令で切り分けられたインライン関数が存在するが，これも関数として識別されていることや，書き換え規則を利用して，識別した関数や前処理指令，コメントなどを削除し，残った宣言などをすべて目で確認したが，関数が含まれていなかったので，未識別はなく，すべての関数定義が識別できたと判断できる．

(A5) の単独の else 節は，6 カ所（3 個のファイル[*8]）で識別されていた．このうち，lib/msvc-inval.h には 3 個の単独の else 節が存在したが，これらはマクロ定義の中に if をともなわないで用いられていた．6 カ所以外の else 節については，書き換え規則を利用して，正しく if 文を構成する字句の並びになっていることを確認した．よって，単独の else 節は，未識別はなく，意図どおりに識別できたと判断できる．

(A6) のマクロ定義内部の解析の正しさを定量的に評価することは難しい．本解析器の開発の過程では，典型例の解析結果を目で見て確認しているが，Coreutils に含まれるすべてのマクロ定義について確認するには数が多く，また，見落としが発生する可能性も高い．ただし，マクロ定義の内部は，制御文の一部だけなど，中途半端な構造が存在する場合があるので，一時的な仮想字句が残りやすく，その有無で解析できているかどうかを評価できる．(A1) の評価で述べたように，解析結果全体で一時的な仮想字句は残留しておらず，結果として，マクロ定義内も定義外と同様に解析できていると評価できる．また，後述するマクロの逆置換が適用できていることも，そのことを裏付けている．

複数のファイルについて，解析結果を目で確認したところ，正しく解析できないマクロが見つかった．典型的な例は，配列の初期化式の波括弧で囲まれた部分だけを定義したマクロで，セミコロンがない式文を持つ複合文として解析されていた．文や宣言などの一部分を切り出したマクロは，それが用いられる文脈を調べないと正しく解析できない．また，逆に，そのようなマクロを用いている箇所は，本来，マクロを展開したときに得られる構造とは異なる構造になりやすい．この問題については，4.4.2 項で議論する．

(A7) の識別子の定義不足に対しては，型をどの程度判定できているかを評価した．識別子ごとの正解情報はないが，一般的な名前付けの方法として，型の名前の最後を “_t” とすることが多いことから，この名前を持つ識別子を抜き出し，それらが型として判定されているか調べた．識別子は 1 つのファイル内では名前が同じものを同一のものとして扱うが，ファイルが異なる場合は別のものとして区別した．また，構造体などのタグ名も同じ名前の付け方が用いられることがあるが，タグ名は struct などの予約語から，ほぼ正確に特定できるので，これを除外した．その結果，該当する識別子のうち 88.3%が型として識別されていた．識別されなかった理由は，ファイル内で宣言には使われず，キャスト演算子として出現するのみなど，判定に必要な情報が含まれていないことにある．

変数と型の誤識別が生じる原因を調べるために，文脈に基づく各出現の種別クラスの判定で，型と変数の両方の結果が生じた識別子の数を調べた．これらは，同名の識別子が別の種別クラスとして使われる事例以外では，何らかの解析の誤りによって生じる．型と変数の両方の出現が存在した識別子は 17 個のみで，主な原因は，引数に型名と変数などの識別子の両方を持つマクロ呼び出しをプロトタイプ宣言として扱い，引数をすべて型と判定していることであった．その他，型を変数とする誤りもあり，関数のプロトタイプ宣言の引数にユーザ定義型のみが書かれている場合にプロトタイプ宣言と識別できず，引数を変数と判定していた．複数のマクロ定義で，同名のマクロの引数が使われ，それぞれで型や変数の両方の場合があることで，本来，型として扱うべき引数が変数として判定した例も見つかった．この分析の対象は，出現が 2 つの種別クラスに判定された識別子のみであるが，どちらか一方と判定された識別子にも誤りが存在する可能性を示唆している．今後，判定方法の改善をするとともに，誤判定を探す方法も検討する必要がある．

(A8) のマクロの実引数として出現する文に対しては，セミコロンや制御の予約語など，文に関する字句を含む式を

[*8] lib/{msvc-inval.h, vasnprintf.c, printf-parse.c}

表 6 マクロの実引数に含まれる文に関連する要素
Table 6 The elements of statements in actual arguments of macro calls.

実引数の種類	箇所数	ファイル数
複文式	53	7
構造体の定義を含む式	7	4
(A2) の仮想セミコロンが誤って挿入された初期化式	5	4
文	3	2
合計	66	17

調べて確認をした．結果を**表 6** に示す．実引数に文を持つマクロ呼び出しは 3 個[*9]で，それぞれ正しく識別されていた．未識別は見つからなかったので，すべて正しく識別できたと判断できる．

4.1.2 PostgreSQL に対する適用

PostgreSQL に関する適用結果は，表 4 に示すように，全体的には Coreutils と同じような精度となっており，提案手法が Coreutils 以外でも有効であることが分かる．精度が低くなっている項目は，Coreutils では見られなかった PostgreSQL の特有の記述によるものである．以下では，特に特徴的な差異が生じた (A1)，(A2)，(A4)，(A8) について説明する．

(A1) については，1 つのファイルについて，木構造を構成できなかった．これは，“`f(a, if(c) g());`” のように，引数に制御文を持ち，かつ，末尾のセミコロンを含まない場合に，引数の制御文の文末の位置を正しく特定できず，関数呼び出しの後ろのセミコロンを文末としていることが原因であった．そのために，関数呼び出しの構造と，文の構造が互い違いに重なった状態になっていた．これについては，括弧の入れ子関係や引数の区切りであるカンマを考慮して文末を判定するように規則を修正する必要がある．

また，一時的な仮想字句の残留が 8 カ所（7 個のファイル）存在した．これらの原因は次のとおりである．

- 非常に長い配列の初期化式で，パターンマッチングのエンジンの限界を超えたことで，演算子 “`=`” の右辺を識別できなかった（6 カ所，5 個のファイル）．
- 関数の内部がアセンブラ言語で書かれていた（1 カ所，1 個のファイル）．
- Perl と C の記述と混合したファイルであり，Perl が C の記述を避けるためのヒアドキュメントの記号 “`<<`” が存在していた（1 カ所，1 個のファイル）．

これらはいずれも提案手法そのものの問題ではない．1 番目の問題は，Perl のパターンマッチングのエンジンの性能の問題である．繰返しのパターンに適合する際に，繰返しの数に上限があり，初期化式を構成する字句の数がその上限を超えたことによる．この上限はエンジンの実装から決まる限界であり，変更は難しい．回避策として，繰り返す要素を字句から式に変更するなど，単位を大きくすることで繰返し数を減らす方法がある．この問題は Coreutils の関数の識別においても経験し，回避策を適用したが，配列の初期化式については未対策であった．PostgreSQL では，大量のデータで配列を初期化する記述が多く，特に長い 6 カ所でパターンの適合に失敗をした．2 番目と 3 番目の問題は，C 言語以外の言語との組合せであり，本解析器では対象としていない．ただし，アセンブラ言語は，他のコードでも使われることがあるので，今後，対応することも検討する．なお，一時的な仮想字句が残っても，それらを包含しない他の構文要素は解析できている．

(A2) は，Coretuils に比べて誤りが多い．これは，PostgreSQL は，配列の初期化式が非常に多いことに加え，マクロを利用して foreach と unless といった制御構造を定義して使用していることが原因である．foreach や unless が複合文を持つ場合は，複合文と結合して 1 つの文として構成されるが，式文の場合は，foreach をセミコロンのない文と誤識別して，そのあとの式文と切り離された構造になる．セミコロンのない文かどうかを正確に判定するには，その記述の意味を理解する必要があり，容易には改善できない．ただし，foreach を while などと同じ制御を表す予約語として字句定義に追加した場合は，正しく解析できる．

(A4) は，14 個の関数（14 個のファイル）の識別に失敗している．これは，次の 3 つの原因による．

- 木構造が構成できていない（1 カ所）
- 返り値の型がない関数定義（4 カ所）
- 頭の部分が 1 つの引数なしマクロで構成される関数定義（9 カ所）

1 番目は，すでに (A1) で延べたとおりである．2 番目は，いずれも main 関数で，3 番目は，字句解析生成系が自動的に生成したものである．2 番目と 3 番目は，このような形に特化した規則の追加で対応は可能であるが，2 番目は C99 の規格には合わず，前処理指令による問題ではないので，また，3 番目は自動生成されたコードであり，それ自体を保守することは少ないので，どちらも積極的に対応する必要はないと考える．

(A8) は 2 つの誤りがあるが，1 つは (A1) の木構造が構成できない場合のものであり，これが解決すれば正しく扱える．もう 1 つは，マクロの引数が C 言語ではなく，かつ，字句 “`default`” を含むことで，ラベル文として扱われたことによる．よって，(A1) の評価でも述べたように，他の言語との混合は対応していないので，適切には扱えない．

PostgreSQL のコードの特徴として，foreach などの独自の拡張構文の利用や，Perl などの他言語との混合などがあり，C 言語の規格と合わない部分への対処が必要となる．同様の状況は，他のコードでもあると考えられるが，本解析器は，書き換え規則を用いているので，ツールの利用者が解析対象に特化した例外規則を追加することで対処可能である．ただし，現状は，解析器の拡張に関する支援の仕組みは十分ではなく，解析器の内部の詳細に精通している

[*9] src/od.c, lib/strftime.c

表 7 分岐指令の移動結果
Table 7 A result of alignment of branch directives.

	すべての指令	else 節がある指令
移動が行われたファイル	147	32
メモリ不足で失敗したファイル	4	0
移動した指令の数	654/7232 (8.9%)	98/2119 (4.7%)
移動によるファイルサイズの増加率	202.9%	167.4%

必要があるので，今後の検討が必要である．

4.2 分岐指令の移動ツール

分岐指令を適切に移動できるか確認するために，Coreutils のすべてのファイルに対して，ツールを適用した．移動対象はすべての指令とした．その結果，**表 7** に示すように，5 つのファイル[*10]はメモリ不足で終了し，147 個のファイルで移動が行われた．複数のファイルで差分を確認をしたところ，すべての前処理指令がファイルの先頭または最後尾に移動するといった極端な移動はなく，適切に境界に移動していた．移動後のファイルをコンパイルし，付属のテストケースを適用した．コンパイルでは，src/factor.c がエラーになった．これは，分岐条件で使われているマクロの定義が分岐内に移動したために，未定義のマクロを参照したことが原因であった．マクロ定義と分岐指令の位置関係について確認する仕組みが必要である．src/factor.c を元のファイルに戻してコンパイルし，テストを適用したところ，エラーも含め移動前と移動後と同じ結果になった．また，コンパイルする代わりに，プリプロセッサの出力を保存し，移動前と後でプリプロセッサの出力に差があるか確認をした．その結果，プリプロセッサが前処理時に挿入する行番号の情報および，行番号を表すマクロ `__LINE__`，および，改行の 3 点の差異が生じた．前処理指令を移動したときに，移動先の境界が 1 つの行の途中にある場合，そこに改行を入れるために，改行が増えており，それにともない，行番号にずれが生じるので，これらの差異は正常なものである．これらのことから，1 つのファイルを除き，前処理の分岐条件を正しく維持していると判断できる．

移動した指令は全体の 8.9%で，移動が行われたファイルのサイズは元の 202.9%に拡大した．メモリ不足の原因は，多数の条件分岐を持つ宣言の存在である．たとえば，src/stty.c の変数 mode_info の宣言は 34 個の分岐指令を含む．本論文の構文解析手法は，前処理指令を無視するが，それによって，構文木の構成が変わるのは分岐指令によって複数の節に分かれる場合である．そこで，単体の節しか持たない前処理指令を移動の対象から外して適用したところ，移動が行われたファイルは 32 個に，また，移動した分岐指令は全体の 4.7%に減少し，ファイルのサイズの増加率も 167.4%であった．さらに，メモリ不足に陥いるファイルもなかった．元のコードとの乖離を避けるためにも，移動させる前処理指令は少ない方がよい．リファクタリングに応用する場合には，修正箇所に限定して移動させる方法が現実的と考える．

条件分岐指令の移動では，(A1) の評価で行ったように，括弧や仮想字句が正しく組を構成していれば，どのようなコードであっても移動は可能である．ただし，メモリ不足で移動できなかった例で分かるとおり，組合せが爆発する場合には，理論的には可能であっても，現実的には移動できない場合がある．特に，(A2) から (A5) のような構文要素の範囲を適切に決定する方法が重要となる．たとえば，Coreutils の lib/siglist.h には，シグナルを設定するマクロの呼び出しが 34 個存在し，それぞれ分岐指令で囲われているが，行末にセミコロンが存在しない．よって，(A2) の方法でセミコロンを挿入しない場合，全体が 1 つの文または宣言になり，分岐指令の移動を適用すると 2^{34} 通り以上[*11]の組合せを構成することになる．本論文の構文解析器は，不正確さを許容することが前提なので，(A2) から (A5) は対処しなくても構文木を構成できるが，このように応用においては，問題を減らすうえで重要となる．

4.3 マクロの逆置換ツール

本手法では，項目の (A6) として示したように，マクロ定義の内部まで解析をしており，定義の内側と，定義以外の部分を同じように解析することで，マクロの逆置換が可能になる．現実的なコードに対して，マクロの逆置換が可能であることを以下のように確認した．具体的には，Coreutils 8.22 のディレクトリ src と lib の全ファイルに含まれる 1,851 個のマクロ定義について，逆置換を src と lib のファイル*.c に適用した．なお，定数を定義するだけの単純なマクロは除外し，空白とコメント以外が完全に同じ定義は 1 つに集約したうえで，各定義ごとに全ファイルに適用した．その結果，374 個の定義が 686 万カ所（521 個のファイル）に適用された．膨大な数になった理由は汎用的な定義が存在することである[*12]．なお，各マクロ定義がその置換部分自体を書き換えることも可能であるが，それは除外した．**図 11** に，適用例を示す．これは，マクロが定義されているファイルの中で，マクロに置き換え忘れられた可能性がある箇所である．なお，この例では，置換部分に引数 `h_` が 2 カ所出現しているが，逆置換された部分でその引数に該当する箇所が両方とも同名の変数であったので適合できている．

一方，適用できない例も存在する．たとえば，PostgreSQL の制御文として定義されるマクロ foreach は，for 文の頭の部分のみ，すなわち，繰返しとなる文を含まずに定義され

[*10] lib/vasprintf.c, src/{sty.c, sig2str.c}, gnulib-tests/test-{signal, fcntl}-h.c

[*11] 実際には`#elif` が 1 カ所含まれている．

[*12] lib/getopt.in.h の `# define __GETOPT_CONCAT(x, y) x ## y` は，2 文字以上の識別子を分解する逆置換になるので，ほぼすべての識別子に適用される．適用箇所は約 12 万（449 ファイル）で，さらに，マクロ名のみが異なる同じ定義が 3 つ存在する．

マクロ定義 (src/pr.c)

```
/* The horizontal position we'll be at after printing a tab character
   of width c_ from the position h_. */
#define POS_AFTER_TAB(c_, h_) ((h_) + TAB_WIDTH (c_, h_))
```

適用結果:

```
--- coreutils-8.22/src/pr.c 2013-12-04 09:48:30.000000000 -0500
+++ pr.x 2014-03-20 10:30:36.000000000 -0400
@@ -1275,8 +1275,7 @@

       /* Estimate chars_per_text without any margin and keep it constant. */
       if (number_separator == '\t')
-        number_width = (chars_per_number
-                        + TAB_WIDTH (chars_per_default_tab, chars_per_number));
+        number_width = (POS_AFTER_TAB(chars_per_default_tab,chars_per_number));
       else
         number_width = chars_per_number + 1;
```

図 11 マクロの逆置換の適用例
Fig. 11 A result of reverse macro expansion.

る．この場合，構文木は 1 つの文として識別するが，本来は文末が存在しないにもかかわらず，文末を表す仮想字句が入るので，for 文に適合することがない．部分木から単純にパターンを生成しているが，項目の (A1) から (A3)，および (A8) のように字句を補完する方法を用いた場合は，補完した字句やそれと関連して加えられた字句を削除するなどの処理が必要である．

適用実験で用いたツールでは，マクロの可変引数の使用は稀と考えて，可変引数には対応していなかった．しかし，マクロ定義を確認したところ Coreutilsl 8.22 では 2 個，PostgreSQL 9.3.5 では 12 個のマクロで可変引数が利用されていた．そこで，拡張が可能かどうか検証した．可変引数は，仮引数部に記号 “...” を，実引数リストの参照にマクロ __VAR_ARGS__ を用いる．逆展開においては，任意個の引数を，そのままマクロ呼び出しの引数に移すのみであるので，仮引数部の記号 “...” とマクロ __VAR_ARGS__ を，引数のリストに適合するパターン変数 ${__VAR_ARGS__:ARGLIST} に置き換える書き換え規則を組み込み，簡単な事例で確認をしたところ，すべて期待どおりに書換えが可能であった．なお，Coreutils8.22 の 2 つのマクロ定義については，逆置換が適用可能な箇所は存在しておらず，拡張したツールでの再実験は行っていない．

4.4 考察

4.4.1 問題への対処と課題

全体としては，対象とした問題をほぼ解決できていると判断でき，また，解析の精度は Coreutils と PostgreSQL で同じ傾向になったので，ある程度の一般性はあると考えられる．しかし，(A3) のように Coreutils には存在しても，PostgreSQL には存在しない問題もあり，開発時に検証対象とした Coreutils に依存している可能性は残る．各問題に対する解析精度の正確さと解析対象に対する依存の脅威を**表 8** に示す．精度については，実験では誤りがなかったことでほぼ正確と確信できるものと，誤りを確認したことで，やや不正確と判断できるものとに分け，Coreutils と PostgreSQL を各 1 点とした点数で表現した．対象への依存の脅威には，ヒューリスティックな知識を用いており，その知識が検証対象の Coreutils に依存する可能性があるものを示した．特に問題になるのは，精度が低く，かつ，対象への依存の脅威がある (A2) と (A7) である．これらは，他のソースコードに適用したときには，対処できない可能性がある．改善にあたっては，セミコロンが含まれるかなどのマクロ定義に関する解析を行い，それを判定に用いることや，他のファイルの解析結果を利用するなど，利用する情報を増やす必要がある．

表 8 問題に対する対処の評価
Table 8 The evaluation of our method for the problems.

問題		(A1)	(A2)	(A3)	(A4)	(A5)	(A6)	(A7)	(A8)
解析精度	正確	1	0	2	1	2	0	0	1
	やや不正確	1	2	0	1	0	2	2	1
対象依存の脅威			○	○	○			○	○

4.4.2 マクロに関する不正確さの問題

本手法は，構文木の不正確さを許容するが，できるだけ正確である方が望ましい．不正確さの主な原因は，前処理の分岐指令の位置とマクロにある．このうち，分岐指令の位置については，文の境界に移動させることで，元のコードとは乖離するが，要素の範囲や構成関係を正確な状態に補正できる．

マクロは，マクロ呼び出しとマクロ定義の両方で検討すべき課題がある．マクロ呼び出しは，展開後の情報を使わない限り正確な解析はできない．ただし，マクロが単体の型や式として定義されていれば，ユーザ定義型や変数，関数呼び出しとして扱うことで，見かけ上の構造を正しく反映した木構造を構成できる．近似的な解決方法は，それ以外のマクロ定義をリファクタリングし，解析しやすい構造にすることである．マクロの逆置換ツールを用いて，展開したあと，新しいマクロに置き換えればよいが，どのようにマクロを定義すればよいか，そもそも定義できるのかといった点については調査と検討が必要である．

マクロ定義の解析は，その呼び出しの文脈に基づかない限り正確にはならない．たとえば，初期化式の一部を定義したマクロはカンマ式との区別がつかない．呼び出しの文脈から，マクロがどの要素を表すかを解析し，その情報に基づいた構文解析の手法が必要である．また，マクロが表す要素が分かれば，マクロ呼び出しを含む箇所の解析もより正確になる．マクロ定義については，仮引数のクラスの問題もある．本手法では，型と変数には同名の識別子を使わないという前提を用いた．しかし，マクロの引数名は，a や x など，単純なものが多く，同名の引数が変数や型になりうるので，変数と型の区別の精度が落ちている．また，マクロの引数が文や宣言，またその一部をとりうるが，識別子を型や変数といったクラスに分ける方法では正しく取り扱えない．クラスの定義も含めて拡張が必要である．

5. 関連研究

本手法の基礎となる部分的な解析手法は island grammar [13] で提案され，正規表現に最短マッチを用いることで，C 言語を解析する書き換え規則が実現できることは ILA [15] で示されている．本論文は，その延長として，現実の前処理前コードを対象とした解析の問題とその解決策を提示している．

前処理前を対象にした解析やリファクタリング支援のツールも多く提案されているが，完全なものはなく，以下の 3 つのうち，いずれかを制約して解決している．

- 前処理指令の出現位置
- 構文木の構成と元のテキストとの同一性
- 構文木の正確さ

最も一般的に用いられる戦略が前処理指令の出現位置を制約する方法である [3], [4], [5], [6]．これは，制約を満たさないファイルは少なく [11]，手作業などで修正することは現実的に可能という前提に基づく．また，前処理指令の出現位置を制約しない代わりに，解析時に出現位置を変更して対処する方法もある [7], [8]．この場合，指令を移動させるときにテキストを複製するので，本来は 1 つの要素であるものが構文木上に重複して出現し，元のテキストとは一致しない．よって，構文木に対する操作をテキストに反映する処理が複雑になる．一方，一定の前処理条件の下で有効になるテキストの組合せに基づいて解析する手法もある [9], [10] が，この場合，構文木には前処理条件に合わないテキストは反映されず，構文木の正確さが犠牲となる．本論文の手法も，構文木の正確さを犠牲にした手法である．ただし，前処理条件には依存せず，すべてのテキストを解析するが，要素の範囲や，要素間の構成関係に不正確な部分が生じる．なお，本論文で示した前処理指令の移動ツールを用いれば，同一性は犠牲になるが，より正確な構文木を得られる．解析時にあらかじめ移動させる方法と異なり，部分的な移動も可能であるので，その点では柔軟性がある．

Java や C++を対象として，プログラムの断片の解析手法や構文的な誤りを許容する解析器や手法も提案されている．プログラムの断片を解析する場合，識別子の定義不足により構文的な曖昧さが生じたり，型が不明になったりすることがあるが，識別子の種別や変数などの型を推論することで，曖昧さを解消したり，未定義の識別子の型を求めたりする方法がある [19], [20], [21]．ただし，これらの方法は，クラスやメソッドなどの大きな粒度の要素を対象とし，前処理指令が存在しないことが前提となる．また，本論文で扱った 8 つの問題のうち，(A7) の定義不足の問題がこれらの処理に対応するが，本論文では構文的な曖昧さを解消できればよいので，型の推論までは行っていない．型を推論する必要がある場合には，これらの既存の方法を応用することで実現可能である．fuzzy parser は，識別子の宣言など，特定の情報の抽出に特化することで，構文的な誤りの影響を回避している [22], [23]．また，JDT [24] など，エディタに組み込まれる構文解析器は，構文エラーに対する回復処理として，エラーを含むブロックを特定し，それを無視することで，コード全体を解析を達成している [25]．これらの手法は，情報抽出に不要な箇所や，エラーを含むブロックを解析をしないが，本論文の手法では，正しさを保証しないものの，エラーの箇所も含め全体を解析する．ただし，書き換え規則を用いるので，エラー箇所は，一部の規則が適用された中途半端な状態になる可能性がある．

6. おわりに

本論文では，前処理指令の位置などを制約せずに，前処理前コードをそのまま構文解析する方法を提案した．また，現実のコードが持つ問題に対する解決方法を示し，字句系列の書き換え規則を用いて解析器を実現した．前処理前のコードの解析結果を用いた応用として，分岐指令を移動するツールとマクロの逆置換ツールも示した．評価として，オープンソースソフトウェアの解析に適用し，一定の精度が得られることを示した．しかし，より高い精度を得るには，マクロ定義や呼び出しの解析方法の改善，マクロに関する解析や他のファイルの解析の結果の活用，さらに，ヒューリスティックな知識の排除などが必要である．今後の課題としては，解析精度の向上に加え，前処理指令に関するリファクタリング支援などのツールへの応用があげられる．たとえば，本論文で示した条件指令の移動はリファクタリングの一種といえるが，重複する部分を増やすので最適な方法ではない．コードの意味を考えて移動させれば重複を抑えられるので [11]，その自動化あるいは支援を検討している．さらに，様々なツールへの応用を増やす一方で，解析対象を広げていくことも重要であり，解析対象の特殊な事例に合わせて書き換え規則を拡張するときの支援方法や，拡張時にツールの機能に与える影響を抑える方法なども検討する必要がある．また，本手法は，C 言語以外に，複数の言語の混合した記述にも応用できる可能性があり，その点についても検討していく．

謝辞 本研究は, JSPS 科研費 26350344 の助成を受けた．

参考文献

[1] Adams, B., De Meuter, W., Tromp, H. and Hassan, A.E.: Can We Refactor Conditional Compilation into Aspects?, *Proc. 8th ACM Inter. Conf. on Aspect-oriented Software Development, AOSD '09*, New York, NY, USA, ACM, pp.243–254 (online), DOI: 10.1145/1509239.1509274 (2009).

[2] Adams, B., Van Rompaey, B., Gibbs, C. and Coady, Y.: Aspect Mining in the Presence of the C Preprocessor, *Proc. 2008 AOSD Workshop on Linking Aspect Technology and Evolution, LATE '08*, New

York, NY, USA, ACM, pp.1:1–1:6 (online), DOI: 10.1145/1404953.1404954 (2008).
[3] Baxter, I.D., Pidgeon, C. and Mehlich, M.: DMS®: Program Transformations for Practical Scalable Software Evolution, *Proc. 26th Inter. Conf. Software Eng.*, Washington, DC, USA, IEEE Computer Society, pp.625–634 (online), available from ⟨http://dl.acm.org/citation.cfm?id=998675.999466⟩ (2004).
[4] Padioleau, Y.: Parsing C/C++ Code without Preprocessing, *Proc. 18th Inter. Conf. Compiler Construction: Held as Part of ETAPS 2009*, Berlin, Heidelberg, pp.109–125, Springer-Verlag (2009).
[5] Kästner, C., Giarrusso, P.G., Rendel, T., Erdweg, S., Ostermann, K. and Berger, T.: Variability-aware parsing in the presence of lexical macros and conditional compilation, *SIGPLAN Not.*, Vol.46, No.10, pp.805–824 (online), DOI: 10.1145/2076021.2048128 (2011).
[6] Collard, M.L. and Maletic, J.I.: Document-Oriented Source Code Transformatin using XML, *Proc. 1st Inter. Workshop on Software Evolution and Transformation*, pp.11–14 (2004).
[7] Garrido, A.: Program refactoring in the presence of preprocessor directives, Ph.D. Thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA (2005).
[8] Gazzillo, P. and Grimm, R.: SuperC: parsing all of C by taming the preprocessor, *SIGPLAN Not.*, Vol.47, No.6, pp.323–334 (online), DOI: 10.1145/2345156.2254103 (2012).
[9] 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).
[10] 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).
[11] Liebig, J., Kästner, C. and Apel, S.: Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code, *Proc. 10th Inter. Conf. Aspect-oriented Software Development, AOSD '11*, New York, NY, USA, ACM, pp.191–202 (online), DOI: 10.1145/1960275.1960299 (2011).
[12] Yoshida, A. and Yoshinori, H.: A Pattern Search Method for Unpreprocessed C Programs based on Tokenized Syntax Trees, *Proc. 14th IEEE Inter. Workshop on Source Code Analysis and Manipulation*, pp.295–304 (2014).
[13] Moonen, L.: Generating Robust Parsers using Island Grammars, *Proc. 8th Working Conf. Reverse Engineering*, IEEE Computer Society Press, pp.13–22 (online), available from ⟨http://www.cwi.nl/~leon/papers/wcre2001/⟩ (2001).
[14] 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ソースコード書換え支援環境，情報処理学会論文誌，Vol.53, pp.1832–1849 (2012).
[15] Cox, A. and Clarke, C.: Syntactic approximation using iterative lexical analysis, *11th Inter. Work. on Program Comprehen. 2003*, pp.154–163 (online), DOI: 10.1109/WPC.2003.1199199 (2003).
[16] 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).
[17] Free Software Foundation Inc.: Coreutils – GNU core utilities, available from ⟨http://www.gnu.org/software/coreutils/⟩.
[18] The PostgreSQL Global Development Group: PostgreSQL: The world's most advanced open source database, available from ⟨http://www.postgresql.org⟩.
[19] Thummalapenta, S. and Xie, T.: Parseweb: A Programmer Assistant for Reusing Open Source Code on the Web, *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering, ASE '07*, New York, NY, USA, ACM, pp.204–213 (online), DOI: 10.1145/1321631.1321663 (2007).
[20] Dagenais, B. and Hendren, L.: Enabling Static Analysis for Partial Java Programs, *Proc. 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications, OOPSLA '08*, New York, NY, USA, ACM, pp.313–328 (online), DOI: 10.1145/1449764.1449790 (2008).
[21] Knapen, G., Lague, B., Dagenais, M. and Merlo, E.: Parsing C++ despite missing declarations, *Proc. 7th International Workshop on Program Comprehension, 1999*, pp.114–125 (online), DOI: 10.1109/WPC.1999.777750 (1999).
[22] Bischofberger, W.R.: Sniff – A Pragmatic Approach to a C++ Programming Environment, *USENIX C++ Conference*, pp.67–82 (1992).
[23] Koppler, R.: A Systematic Approach to Fuzzy Parsing, *Software Practice and Experience*, Vol.27, p.649 (1996).
[24] The Eclipse Foundation: JDT Core Component, available from ⟨http://www.eclipse.org/jdt/core/index.php⟩.
[25] de Jonge, M., Kats, L.C.L., Visser, E. and Söderberg, E.: Natural and Flexible Error Recovery for Generated Modular Language Environments, *ACM Trans. Prog. Lang. Syst.*, Vol.34, No.4, pp.15:1–15:50 (online), DOI: 10.1145/2400676.2400678 (2012).

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．この間 2013 年から 2014 年までクィーンズ大学客員研究員．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．ソフトウェア科学会，電子情報通信学会，IEEE Computer Society 各会員．

蜂巣 吉成

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．この間 2004 年から 2006 年までエジンバラ大学客員研究員．博士（工学）．構造化文書の記述，ソフトウェアの開発支援環境，e ラーニングに関する研究に興味を持つ．ソフトウェア科学会，電子情報通信学会，日本教育工学会，IEEE Computer Society，ACM 各会員．