情報処理学会論文誌 Vol.54 No.2 912–921 (Feb. 2013)

# 前処理前プログラムに対する記号表の構成手法

前林 達也[1,a)] 吉田 敦[2] 蜂巣 吉成[2] 張 漢明[2] 野呂 昌満[2]

受付日 2012年5月14日, 採録日 2012年11月2日

**概要**：プログラムの開発や保守の作業量を減らすために，書き換え作業を自動化するための環境が提案されている．書き換えを目的としたプログラム解析器は，コメントやスタイルを維持する必要があり，特にC言語の場合には，前処理前のプログラムをそのまま解析する必要がある．しかし，解析器は，記号表を構成せずに解析するので，解析結果に誤りを含む場合がある．前処理前プログラムは，定義の欠落などにより，構文として完全でないことがあり，これに対処した記号表の構成手法は知られていない．本論文では，前処理前プログラムにおける問題点を3つに分類し，これらに対処した記号表の構成手法を提案する．これにより，構文解析の結果を補正できることを示す．

**キーワード**：プログラム解析，前処理前プログラム，記号表

# A Symbol Table Construction Method for Un-preprocessed Programs

Tatsuya Maebayashi[1,a)] Atsushi Yoshida[2] Yoshinari Hachisu[2] Han-Myung Chang[2]
Masami Noro[2]

Received: May 14, 2012, Accepted: November 2, 2012

***Abstract:*** Automatic program transformation is a solution for reducing costs of program development and maintenance. Program analyzers for transformation need to parse un-preprocessed programs for preserving program styles. Because those programs are incomplete in syntax, and lack some definitions of identifiers, the analyzers can not construct a symbol table and the results of analysis become inaccurate. In this paper, we divide obstacles of constructing a symbol table into three types, and then propose a symbol table construction method based on them. We also show that our method contributes accuracy enhancement of a program analyzer by examples.

***Keywords:*** program analysis, un-preprocessed program, symbol table

## 1. はじめに

プログラムの開発や保守は，プログラムの書き換えの連続であり，書き換え作業をできるだけ自動化することで開発や保守の作業量を減らすことができる．このとき，書き換えの対象となるプログラムは，必ずしも構文的に完全とは限らない．たとえば，プログラムの断片をテンプレートとして部品化し，プログラムを自動構成するような場合，テンプレートの編集支援が必要となる．書き換えは，抽象構文木に対する操作を自動化することで実現されるが，その場合，定義の一部が不足したり，テンプレート用の表現を含んだりした状態で構文解析する必要がある．さらに，C言語の場合には，前処理系が存在しており，前処理命令を用いて記述することで，C言語の構文を満たさないプログラム記述になる場合がある．しかし，構文解析をするために，前処理を行って構文として完全な状態にした場合，マクロが展開されることや，条件分岐によって一部の記述が無効になることで，解析前のプログラムが持っていたプログラムのスタイルや，一部の記述が失われる．プログラムの書き換えを自動化して開発支援を行うためには，この

[1] 南山大学大学院数理情報研究科
Graduate School of Mathematical Sciences and Information Engineering, Nanzan University, Seto, Aichi 489–0863, Japan
[2] 南山大学情報理工学部
Faculty of Information Sciences and Engineering, Nanzan University, Seto, Aichi 489–0863, Japan
[a)] m12mm016@nanzan-u.ac.jp

© 2013 Information Processing Society of Japan

ような構文的に不完全なプログラムを構文解析する必要がある．

構文的に不完全なプログラムを対象とした解析を実現するために，srcML [1] や TEBA [10] が提案されている．srcML は，C 言語や C++ 言語のプログラムを前処理前の状態で解析する．TEBA は，C 言語のみを対象とするが，C 言語に独自の記述を加えたプログラムパターン記述なども解析できるよう拡張性を持っている．これらの解析器では，テンプレートやプログラム断片を入力とする場合，識別子の定義が存在しないなどの問題があるので，記号表を作らずに構文解析を行う．記号表を用いない解析では，スコープ規則に基づく識別子の区別がなく，また，宣言を文として解析するなど，誤りを含む問題がある．この問題を解決するには，srcML や TEBA による構文解析の結果に基づいて記号表を構成し，その記号表を用いて，解析結果を補正することが必要である．

記号表を構成することができれば，解析の精度が向上するだけでなく，スコープ規則に基づく識別子の区別が可能となる．データフロー解析など，プログラムの意味に関する解析も可能となり，前処理前プログラムに対する精度の高いリファクタリングなどの実現にもつながる．また，その応用例として，前処理前プログラムに適用できるクロスリファレンサを実装できる．しかし前処理前プログラムを解析する際には，未定義識別子の参照や，前処理の条件分岐によって選択される定義を同時に扱うことによる定義の多重化などが発生し，それらに対処した記号表の構成手法は知られていない．

本論文では，前処理前プログラムに対して記号表を構成する方法を提案する．まず，記号表の構成を困難にする問題を 3 つに分類し，それぞれについて解析方法を示す．さらに，適用実験を行い，記号表が構成できること，またそれにより解析結果が補正され，精度の向上に貢献することを示す．以降では，2 章で前処理前プログラムにおける記号表の構成について問題を整理し，3 章でその具体的な構成手法を示す．4 章では解析器に記号表を導入し，その効果について評価する．5 章で記号表を用いた解析器の応用や，残存した課題について議論する．

## 2. 前処理前プログラムにおける記号表構成の問題

### 2.1 前処理前プログラムとは

本論文での前処理の定義とは，コンパイル時に行う前処理だけでなく，独自に部品化したプログラムの合成などの処理も含めたものであり，その入力となる記述をすべて前処理前プログラムと呼ぶ．前処理前プログラムとは，次の特徴を持った C 言語のプログラムの記述の断片とする．

- 識別子の定義が欠落している．
- 識別子の定義が重複している．
- 前処理命令やテンプレートの記号など，C 言語の構文と合致しない表現を含む．

```
void
usage (int status)
{
  if (status != EXIT_SUCCESS)
    fprintf (stderr, _($ERROR), program_name);
  else
    {
      printf (_($MESSAGE1), program_name);
      fputs (_($MESSAGE2), stdout);
      fputs (HELP_OPTION_DESCRIPTION, stdout);
      fputs (VERSION_OPTION_DESCRIPTION, stdout);
    }
  exit (status);
}
```

**図 1** usage 関数を作成するテンプレート
**Fig. 1** A template of the function ‘usage’.

プログラム断片は，ソースプログラムの一部分を切り取ったものであり，識別子の定義が欠落あるいは重複する場合がある．テンプレートとは，**図 1** のように，テンプレート記号の箇所を置き換えることでプログラムとして完成するものである．図 1 の例は，コマンドなどの使用方法を出力する usage 関数を作成するテンプレートであり，$ で始まる字句を，置換対象となるテンプレート記号としている．また，図 1 の例では，stderr などの識別子の定義が欠落している．これらの特徴を持つ前処理前プログラムの解析を実現する環境として，srcML [1] や TEBA [10] が提案されている．

### 2.2 前処理前プログラムの構文解析

本論文では，前提として，解析器に TEBA を用い，記号表を構成しないで構文解析ができるものとする．ただし，srcML など同様の解析器についても，本論文の手法は適用可能である．TEBA による構文解析は，機能ごとに特化したフィルタによる段階的詳細化を行う．TEBA の解析結果として得られる出力は，属性付き字句系列と呼び，次の形で表現される属性付き字句の並びによって，抽象構文木と同等の情報を持つ．

種別 (属性値)* ‘<’ テキスト ‘>’

属性付き字句系列は，プログラムの字句を出現順に並べたものであり，空白やコメントなどを含むので，スタイル情報が維持される．また，長さ 0 の文字列を字句として扱い，これを仮想字句と呼ぶ．仮想字句は，主に非終端の構文要素の開始と終了を表し，BEGIN_ や END_ で始まる種別に用いる．以後，種別とは，属性付き字句の種別を指すものとする．

TEBA における識別子の詳細化は，前後の字句の並びのパターンに基づいて行われ，識別子の種別として，型，変

© 2013 Information Processing Society of Japan

```
int p;
typedef struct t *sp;
sp f();

((sp)p + 1)->x;
f()->g;
```

図 2　ポインタを扱う式の例

**Fig. 2** An example of pointer-typed expressions.

数，関数，タグ，メンバ，ラベル，マクロを区別する．この処理はヒューリスティック補正と呼ばれ，TEBA における段階的詳細化の 1 つである．ヒューリスティック補正における識別子の詳細化は，識別子が出現する箇所の前後の文脈のみに基づくので，誤りを含むことがある．

#### 2.2.1　名前空間

構文として完全であることが保証されない前処理前プログラムの解析では，名前空間の区別は前後の文脈から推定する必要があるが，字句の並び方によりおおよそ確定できる．C 言語の名前空間は，タグ，ラベル，メンバと，その他すべての識別子（型，変数，列挙子）の 4 つに分けられる．前処理前プログラムでは，マクロ定義が存在するので，#define と#undef 命令によって，すべての名前空間にまたがって独自の名前空間が作られる．よって，5 つの名前空間を区別する．ただし，文脈のみでマクロを特定することは不可能である．記号表で扱う対象となるのは，タグ，ラベル，メンバ，型，変数，列挙子であり，マクロは独立した対処を行う．

タグは，struct，union，enum の直後に付けられるので，問題なく区別できる．ラベルは，goto，case ともに直後に付加される “:” で区別できる．メンバは，ドット演算子やアロー演算子のあとに存在するので，メンバであることは区別できる．ただし，**図 2** のようにポインタを扱う式である場合，演算子の直前の式の型に依存するので，どのメンバであるか不明な場合がある．たとえば，`((sp)p + 1)->x;` では，x が sp 型の構造体のメンバであることを確定するには，sp 型のキャストであることを知る必要がある．しかし，TEBA や srcML では部分式の解析を行っておらず，式の型を特定できないので，本論文では対象としない．ただし，型を特定できる場合は，構造体変数が記述されている場合と同様の方法で解析が可能である．型，変数，列挙子は，同一の名前空間にあるが，文と宣言を区別するために，型とそれ以外を区別する必要がある．

#### 2.2.2　前処理命令

前処理前プログラムでは，前処理を行わないので#define や#ifdef などの前処理命令がそのまま残されているが，TEBA では，これらの命令を空白と見なす，近似的な方法で構文解析を行う．これにより，#ifdef などの前処理の条件分岐命令によって括弧の対応がとれない場合を除き，解析可能となる．

前処理前プログラムでは，マクロは#define で定義され，#undef で無効になる．その間に出現する，マクロとして定義された識別子は，C 言語の 4 つの名前空間と関係なく，すべてマクロとなる．ただし，前処理前では，マクロが有効，無効になるかどうかは必ずしも確定しない．よって，マクロ定義は「有効である」,「有効の可能性がある」,「無効である」の 3 通りの状態を考える必要がある．

プログラマの立場から考えると，識別子がマクロかどうかは必ずしも意識せず，変数や関数といった文脈上で決まる要素として理解している．プログラムの書き換え支援を考えた場合，マクロであることを意識せずに，文脈上の区別に基づく方が自然である．一方，クロスリファレンサを構築するときなど，識別子の定義を正確に取り扱う必要があるときは，マクロかどうか区別する必要がある．すなわち，マクロに関する名前空間を取り扱うかどうかは，目的によって異なる．本論文で取り扱う記号表の構成では，マクロを扱わず，マクロを扱う必要があるときには，別途解析を行うものとする．なお，すでに，識別子とマクロの 3 状態を結び付けるフィルタを TEBA 用に試作し，実現可能であること，また，本論文で提案する記号表の構成手法と併用できることを確認している．

マクロ定義内に記述される展開後の字句列については，展開される箇所によってスコープが異なり，記号表の構成手法には議論を要するので，本論文では対象としていない．これに対する記号表の構成手法は，今後の課題とする．

### 2.3　記号表構成の問題

前処理前プログラムに対する記号表の構成に，一般的なコンパイラの手法を用いると，必要な定義が不足する「定義の欠落」と，同じ識別子に複数の定義が記述される「定義の多重化」が問題となる．定義の欠落がある場合には，複数の解釈が可能な文が出現することがあり，これを「解釈の多重化」と定義する．以下に，各問題について説明する．

#### 2.3.1　定義の欠落

定義の欠落とは，解析対象のプログラムにおいて，定義のない識別子が使用され，記号表を探索した際に定義が見つからないという問題であり，プログラムの断片を解析対象とする場合などに定義の欠落が起こりうる．前処理前プログラムでは一般的に，対象プログラム内にすべての識別子の定義が含まれることは期待できず，定義は欠落していることが前提となるので，定義がない識別子を記号表で取り扱う方法が必要である．

定義がないと，型や有効範囲など，識別子に関する定義が欠落する．しかし，解析の過程で，欠落した情報を補完できる場合もある．記号表の構成にあたっては，情報の欠落を許容しつつ，解析の過程で情報を補完する仕組みが必要である．

```
#ifdef A
unsigned int hoge;
#else
int hoge;
#endif
hoge = 0;
```

図 3　定義の多重化の例
**Fig. 3**　An example of duplicate definitions.

解析対象
```
/* foo は関数か型 */
foo (bar);
```

TEBA による解析結果
```
UNIT_BEGIN <>
BEGIN_STMT #E0001 <>
ID_FUNC <foo> /* 関数 */
SPACE_B < >
PAR_L #B0001 <(>
ID_VAR <bar>
PAR_R #B0001 <)>
SEMI <;>
END_STMT #E0001 <>
SPACE_NL <\n>
UNIT_END <>
```

図 4　TEBA が解析を誤る例
**Fig. 4**　The programs that TEBA can not analyze correctly.

#### 2.3.2　定義の多重化

前処理前プログラムでは，条件分岐命令によって選択される断片に，同一識別子に対するそれぞれ異なった定義が記述されていることがある．これにより，複数の定義がそのまま残る場合がある．これを定義の多重化と呼ぶ．たとえば，**図 3** のように，名前，名前空間，有効範囲がすべて同じ定義が存在することがある．記号表を構成するためには，重複した定義を共存させる方法が必要である．よって，重複した定義をそのまま登録することを許容し，探索では重複したすべての定義を参照する仕組みが必要である．

#### 2.3.3　解釈の多重化

次のような複数の解釈が可能な構文があり，記号表なしでは解析を誤ることがある．

(1) `foo (bar);`
(2) `a = (x) - 1;`

(1) において，foo の解釈は関数呼び出しのみではない．この場合，foo は関数，型，マクロの 3 つの解釈が考えられる．このような複数の解釈が可能な文を，解釈が多重化した文と呼ぶ．解釈が多重化した文は解析を誤る場合があり，実際に TEBA や srcML は (1) の文を関数呼び出しと解析している．**図 4** は，TEBA が誤った解析をした結果で，型の可能性がある識別子 foo について，その字句の属性である種別を参照すると，関数を表す ID_FUNC となっている．(2) の右辺は，式の場合と，x 型のキャストの場合がある．

解釈の多重化は，識別子の定義と参照を対応付ける場合か，前後の文脈で識別子の種別を確定できる場合に，解釈を一意に定めることができ，解消できる．たとえば `foo (bar);` の多重化した解釈を一意にするには，foo について関数または typedef の定義が記述されているか，`foo baz;` のように，foo を型と確定できる文脈が存在する必要がある．記号表を用いて識別子の種別を補正することで対処する．このとき，foo が関数ならば文であるが，型の場合は，`foo (bar);` は宣言であり，文であると構文解析するのは誤りである．

## 3.　記号表の構成手法

記号表の役割は，識別子を管理することである．本論文における記号表構成の操作として，抽象構文木を深さ優先探索で走査し，宣言では登録処理を行い，識別子が使用される箇所では探索処理を行う．これによって識別子の定義と参照を対応付ける．また，識別子には，記号表に登録した時点で割り振った ID を保持させる．

### 3.1　記号表の定義

本論文で用いる記号表で必要とする情報は，識別番号，名前，型，種別，有効範囲の 5 つの要素とする．記号表は，プログラムの書き換えを想定した解析器において，識別子の対応関係を表す情報を付随することや，解析精度を向上させることに用いる．したがって，コンパイラが要求する領域や番地などを含む完全な記号表は対象としない．そのような拡張は容易であるが，型が不明な識別子などが含まれるので，この場合は完全な記号表を作成することが不可能であり，拡張を施しても完全な記号表にはならない．本論文では，これらの要素が欠落した状態での登録や，欠落した要素の補正を許容する記号表を構成する．

### 3.2　問題の対処方法

#### 3.2.1　定義の欠落

定義の欠落は，未定義識別子の参照が出現した時点で発生する．定義が欠落した識別子に関しては，確定できる情報を用いて定義を復元し，未定義の印をつけて，記号表に登録する．

定義として復元すべき要素は，名前，種別，有効範囲である．型は不明であるが，型推論を行わないので，復元できない．種別は，前後の文脈から推定されたものを用い，有効範囲は大域とする．この場合，局所変数より後で大域変数を登録するので，大域変数を管理する表を分離して作成するなどして，局所変数と大域変数の登録が混在しないようにする．

構造体の変数とメンバの関係は，ドット演算子またはアロー演算子の直前に現れる識別子を構造体型の変数とし，直後の識別子をそのメンバとする方法で復元する．このとき，その識別子を含む式の形にかかわらず，それを無視し

© 2013 Information Processing Society of Japan

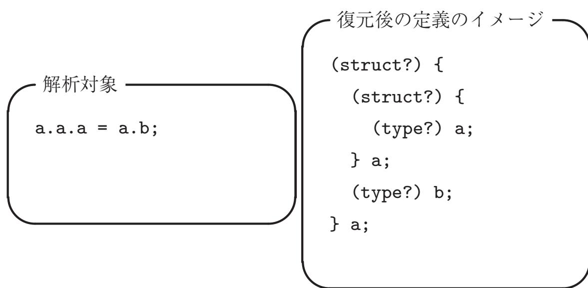


図 5　定義の復元の例

**Fig. 5**　The reconstructed definitions from the symbol table.

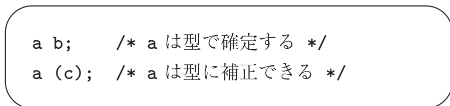


図 6　種別の補正が可能な例

**Fig. 6**　The correctable identifiers used in specific contexts.

て識別子だけに着目する．その例として，図 **5** のように，構造体変数とそのメンバの対応関係を復元できる．解析対象として与えられた文から復元される定義は，図 5 の右側のように，型は不明であるが，対応関係は明確である．

未定義の印とは，推定によって復元された定義であることを示す印であり，前後の文脈によって補正されることを表す．ここで，補正とは，TEBA のヒューリスティックルールに基づいて区別された種別に誤りが含まれる場合，その種別を正しく修正することである．srcML では，種別の区別がないので，識別子の詳細化が必要である．図 **6** の場合，識別子 a について，TEBA は 1 行目は型に，2 行目は関数とする．この場合，1 行目は解釈が一意なので，2 行目の a を型に補正することで正しい解析結果となる．

補正には，次の優先順位を設定する．

(1) 型，(2) 関数，(3) 変数，(4) 種別が 1 種類に定まらないもの

優先順位は，補正が矛盾することを防ぐために用いる．矛盾は，図 6 における識別子 a のように，型として出現する文脈と，型か関数か確定しない文脈の両方が存在するような場合に発生する．識別子が型として出現する文脈では，解釈はつねに一意であり，文法的な誤りがなければ，その識別子の種別は型で確定できる．しかし，型か関数か確定しない文脈や，関数として定義された識別子が，関数ポインタの利用により変数の文脈で出現する場合がある．すなわち，変数の文脈で出現した識別子が実際には関数である可能性や，型である可能性がある．以上より，字句の並びから種別が確定しやすい順に優先順位を設け，補正が 1 方向になるようにし，補正における矛盾を防ぐ．これにより，誤った補正が繰り返されることがなくなり，正しい結果を得られる．種別を型に補正した場合は，それ以上の補正が発生しないので，未定義の印を消去できる．関数や変数として出現した場合は，より優先度の高い種別である

```
#ifdef A
signed int hoge;
#else
int hoge;
#endif
hoge = fuga;
```

| 識別番号 | 名前 | 種別 | 有効範囲 | 型 | 印 |
|---|---|---|---|---|---|
| 1 | hoge | 変数 | 大域 | signed int | |
| 1 | hoge | 変数 | 大域 | int | 重複 |
| 2 | fuga | 変数 | 大域 | ? | 欠落 |

図 7　多重化した定義の記号表での取扱い

**Fig. 7**　An example of a symbol table including duplicated definitions.

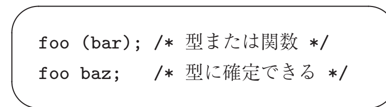


図 8　解釈の多重化の例

**Fig. 8**　An example of expressions with various interpretations.

可能性が残るので，未定義の印を消去しない．

#### 3.2.2　定義の多重化

前処理前プログラムにおいて，#ifdef などにより重複した複数の定義は，1 つだけを有効にするのではなく，すべて同時に存在する．記号表は，すべての定義を多重に登録することで，型や名前空間について，複数の可能性があることを記録する．本来は，記号表に同じ識別子についての定義が存在することは起こりえないが，前処理前プログラムでは，この状態を許容した記号表を構成する必要がある．

定義の多重化に該当する定義が出現したとき，重複の印をつけて記号表に登録する．重複の印とは，名前，名前空間が等しく，かつ有効範囲が同じ識別子を登録する際に，記号表に重複した定義が存在していることを示す目印である．通常は，記号表の探索において該当する定義を発見した時点で探索を終了するが，印がついている場合は，別の定義がまだ存在することを示しているので，探索を継続する．

定義の多重化が出現したとき，記号表は図 **7** のようになる．登録時は，同一の識別番号を付与し，重複の印をつける．参照時は，該当する定義をすべてを参照し，それを統合した情報を取得する．

#### 3.2.3　解釈の多重化

多重の解釈が可能である記述は，断定せずに複数の可能性がある状態にしておき，前後の文脈から解釈を一意にする情報を得られる場合は，それに基づき補正する．図 **8** の例では，TEBA や srcML では字句の並びから，1 行目の foo を関数として解析している．しかし 1 行目の foo は，型の可能性を持つので，「型または関数」とすることが適切である．よって記号表には，種別を「型または関数」とし，解釈が多重化した定義を作成して登録する．2 行目で foo は型に確定するので，以降は foo を型に補正できる．このとき，1 行目の foo も型に補正し，文から宣言に直す必要があるが，解析は字句の出現順に行われるので，後方で確定した情報を前方に反映する必要がある．

前処理の条件分岐を用いることで，foo が「型または関数」である場合と，型である場合が同時に存在する可能性

© 2013 Information Processing Society of Japan

がある．対処法として，前処理の条件分岐を無視せず，どの条件下で種別が何であるか正確に解析する方法が考えられるが，同じ識別子に対して異なる種別を定義することは想定しにくく，前処理命令を無視した解析と比較して，精度が向上することは期待できない．

### 3.3　構文解析結果の補正

型と変数・関数識別子の区別がついたことで，宣言を文として誤って解析している箇所を補正することができる．記号表を用いて種別を修正した抽象構文木を用い，再度構文解析を行い，文から宣言へ補正する．

図 8 では，`foo (bar);` が宣言に補正される．よって，bar は foo 型の変数である．しかしこのままでは，bar は未定義識別子として扱われているので，再度記号表を構成することで，bar を foo 型の変数として記号表に登録することができ，正しい解析結果となる．

## 4.　実装と評価

### 4.1　実装

TEBA を**図 9** のように拡張した．以下に，新規作成および変更したフィルタの概要を示す．

- MakeSymbolTable
  属性付き字句系列を入力とし，記号表を構成するフィルタである．
- FixType
  後方で確定した種別の情報を，前方に反映するフィルタである．
- FixTree
  文から宣言への補正を行うフィルタである．
- syntax.rules
  ヒューリスティック補正を行うルール記述である．解釈の多重化が起こりうる字句の並びにおいて，種別を断定せず曖昧な解析を行うよう変更した．

ソースプログラム例として**図 10** を用いて TEBA を拡張した解析結果を比較した結果を，**図 11** に示す．スペースの都合上，空白の字句は消去している．TEBA は，括弧の対応関係を示す #B で始まる ID と，構文の開始と終了の対応関係を示す#E で始まる ID を付与する．本論文ではこれに従い，識別子には，#I で始まる ID を付与する．

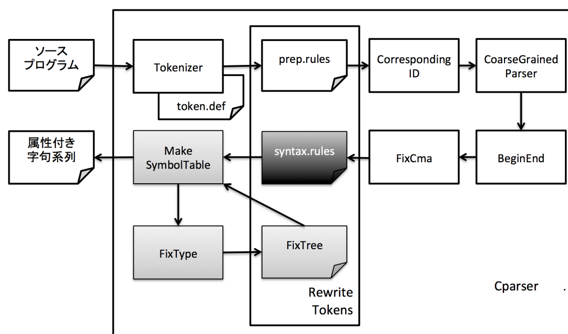


図 9　拡張後の TEBA の全体像
**Fig. 9**　The outline of extended TEBA.

図 11 では，拡張前の TEBA による解析結果と異なる箇所の左側に数字を追記した．1) は，定義の多重化に対処した結果を示している．宣言ではそれぞれ異なる定義に対して同一の ID を付与し，それを参照する場合は，同一の ID で，「型か変数」を表す ID_VorT を付与している．2) は，同じ名前の識別子について，名前空間が異なる場合は異なる ID を付与していることを示す．3) は，解釈の多重化によって複数の種別を持つことを示す．ここで，`f z;` のように f を型に確定できる文が前後に含まれる場合，`f (g);` は変数宣言に補正される．

記号表を基に TEBA の属性を補正するが，支援目的に

```
#ifdef M
typedef int x;
#else
int x;
#endif

void a(){
  struct a a;
  f (g);
  a.a = (x) - 1;
  goto a;
}
```

図 10　解析対象ソースプログラム例
**Fig. 10**　The target program of the analysis.

```
   UNIT_BEGIN <>
   BEGIN_DIRECTIVE #E0001 <>
   PRE <#ifdef>
   ID_MACRO <M>
   END_DIRECTIVE #E0001 <>
   BEGIN_TYPEDEF #E0002 <>
   RESERVE <typedef>
   ID_TYPE  <int>
1) ID_TYPE #I0001 <x>
   SEMI <;>
   END_TYPEDEF #E0002 <>
   BEGIN_DIRECTIVE #E0003 <>
   PRE <#else>
   END_DIRECTIVE #E0003 <>
   BEGIN_DECL #E0004 <>
   ID_TYPE  <int>
1) ID_VAR #I0001 <x>
   SEMI <;>
   END_DECL #E0004 <>
   BEGIN_DIRECTIVE #E0005 <>
   PRE <#endif>
   END_DIRECTIVE #E0005 <>
   BEGIN_FUNC #E0006 <>
   ID_TYPE  <void>
2) ID_FUNC #I0002 <a>
   PAR_L #B0001 <(>
   PAR_R #B0001 <)>
   BEGIN_STMT #E0012 <>
   CUR_L #B0002 <{>
   BEGIN_DECL #E0007 <>
   BEGIN_STRUCT #E0008 <>
   RESERVE <struct>
2) ID_TAG #I0003 <a>
   END_STRUCT #E0008 <>
2) ID_VAR #I0004 <a>
   SEMI <;>
   END_DECL #E0007 <>
   BEGIN_STMT #E0009 <>
3) ID_TorF #I0005 <f>
   PAR_L #B0003 <(>
2) ID_VAR #I0006 <g>
   PAR_R #B0003 <)>
   SEMI <;>
   END_STMT #E0009 <>
   BEGIN_STMT #E0010 <>
2) ID_VAR #I0004 <a>
   OPE <.>
2) ID_MEMBER #I0007 <a>
   OPE <=>
   PAR_L #B0004 <(>
1) ID_VorT #I0001 <x>
   PAR_R #B0004 <)>
   OPE <->
   LITERAL <1>
   SEMI <;>
   END_STMT #E0010 <>
   BEGIN_STMT #E0011 <>
   RESERVE <goto>
2) ID_LABEL #I0008 <a>
   SEMI <;>
   END_STMT #E0011 <>
   CUR_R #B0002 <}>
   END_STMT #E0012 <>
   END_FUNC #E0006 <>
   UNIT_END <>
```

図 11　拡張後の解析結果
**Fig. 11**　The result of the analysis by extended TEBA.

© 2013 Information Processing Society of Japan

よっては，記号表の詳細な情報が必要になる場合があることから，字句系列内にコメントとして記号表を書き出している．その際，すべてのスコープの情報が必要となるので，記号表を構成する過程で，処理が完了した局所スコープの情報も残している．

### 4.2 評価方法

評価には，拡張する以前の TEBA と，本論文で拡張を行った TEBA を用い，記号表構成前後での解析結果の変化から，本論文の提案手法の精度を評価する．また，評価対象には，GNU coreutils 8.9 [3] の src ディレクトリに含まれる，113 個の .c ファイル（平均行数：615 行）を用いる．記号表を構成することで，解析精度が向上したことを，次の評価基準を満たすことで確認する．

- 同じ識別子に対し同じ ID を割り当てられている．
- 記号表が保持している型が正しい．
- 記号表なしでは正しく解析できない箇所が，記号表を用いて修正されている．

### 4.3 評価結果

GNU coreutils 8.9 のうち，対象とした 113 個のソースプログラムでは，名前のみで区別して識別子の個数を数えると，合計で 16,405 個の識別子があった．記号表を用いて識別子を区別し，それぞれに ID を割り当てた結果は，全部で 17,619 個となった．これは，区別されていなかった同名の識別子が，名前空間やスコープ規則に基づいて区別されたことによる．したがって，拡張前の TEBA では 17,619 個の識別子のうち 1,214 個（約 6.9%）が，誤って他の識別子と同一に見なされていたが，記号表を用いることで改善されたといえる．識別子 17,619 個のうち 8,976 個（約 50.9%）は型が特定されていた．識別子の正しい分類を自動的に解析する方法はないので，それらの型が正しいことは，スコープ規則に基づき，目視で確認をした．残りの識別子は，定義の欠落により型が不明となった．前処理を行い，ヘッダファイルを展開すると，前処理前では型が不明であった識別子も型を特定できたが，前処理の条件分岐により無効となった箇所を解析できないことや，見かけ上変数や関数として記述されていたマクロが置換されることにより，解析対象であるソースプログラムのそのままの表現が失われるので，支援目的に適合しない．

TEBA の拡張前後の解析結果を比較すると，16,045 個中 893 個の識別子について種別が変更されていた．主な変更箇所は，字句の並びから種別を確定できない識別子について記号表を用いて正しい種別に補正した点や，解釈の多重化を無視した解析から曖昧な種別を許容する解析に変更したことにより，種別が変更された点などである．典型的な関数の記述である `exit (EXIT_SUCCESS);` や，`free (p);` などに対しても，「型または関数」と補正された．これらの記述は，識別子と括弧の間に空白があり，括弧には識別子が 1 つだけある，文とも宣言とも解釈できる構成をしているからである．種別が変更された 893 個の識別子のうち，378 個が「型または関数」のまま補正されなかった識別子であり，そのうちの 374 個は，`free (p);` のように，実際には関数であるにもかかわらず「型または関数」と補正していた識別子で，4 個は関数型のマクロであった．よって，残りの 515 個の識別子が，補正された解析誤りであると考えられる．すなわち，本論文の手法を用いて TEBA を拡張したことによって，全体の約 3.2%の解析誤りを補正できたといえる．ただし，誤りの詳細や，記号表を用いた解析で検出できなかった誤りの発見には，ソースプログラムと解析結果をすべて目視によって確認する必要があり，困難である．

ファイル 1 個あたりの解析時間は，拡張前の TEBA では平均約 0.39 秒だったのに対し，拡張後の TEBA では平均約 0.96 秒であった[*1]．相対的に見れば時間は大幅に伸びているが，実用範囲内であるといえる．

## 5. 考察

評価実験で，識別子の種別が補正された箇所を調べたところ，変数から関数へ変更されたものが含まれていた．変更自体は正しいが，C 言語の規格上は，変数と関数の区別がない．関数へのポインタなどの利用を考えると，TEBA の種別で変数と関数を区別しないよう変更する必要がある．

正しく解析できない記述を目視で確認したところ，マクロの特殊な使い方が原因で，適切に補正できないものが含まれていた．たとえば，GNU coreutils 8.9 の expr.c では，`VALUE *v IF_LINT ( = NULL);` という記述が用いられており，v が型，IF_LINT が関数となる誤った解析結果となった．マクロの記述方法には制約がないので，一般的な解決は難しいが，マクロの特殊な利用例に基づく字句列のパターンを定義し，そのパターンに適合する識別子は記号表でマクロとして扱うよう拡張することで対処できると考えられる．

実験結果から，前処理前プログラムに対して，記号表を構成できることが確かめられた．定義の欠落や重複がある場合でも，記号表を用いた解析を行い，その解析結果を用いてデータフロー解析を行うことで，リファクタリングなどにつながると考えられる．また，精度をより向上させる方法として，記号表を構成することで得られた情報に基づく，経験的な出現パターンを利用する方法が考えられる．たとえば，前述の `free (p);` のように，実際には関数であるにもかかわらず「型または関数」と補正される問題は，「変数 p の定義のあとに `free (p);` が出現するならば，free は関数である」といった，経験則を用いて改善できる．現

*1 Intel Core i7 1.8 GHz，4 GB，Mac OS X 10.7.3，Perl 5.12.4

© 2013 Information Processing Society of Japan

状の TEBA は，記号表の情報を使わずに字句の並びのパターンのみを用いてヒューリスティック補正用の規則を記述するので，経験則を適用する枠組みを実現するには，記号表に関する情報をパターンに加えられるよう拡張が必要である．ただし，同一の名前を持つ識別子の字句が存在するといった条件は記述可能であるので，名前の代わりに識別番号を参照するように拡張することは容易である．

本論文で提案している手法は，前処理前の C 言語のプログラム断片を前提としている．2.3 節で示した 3 つの問題のうち，定義の欠落と定義の多重化は，プログラミング言語に依存せず，前処理前のプログラム断片であれば共通して起こる問題である．この問題に対する対処方法も，手続き型の言語であれば基本的には同じであるが，種別を推測するときの方法は，経験的にとらえた構文的な特徴に基づいている．また，3 つの問題のうち，解釈の多重化は，C 言語に特有なものであり，これも構文に対する経験的な知見に基づいている．したがって，提案手法の一部は Java などの他の言語にも適用可能であるが，全体としては C 言語以外への適用は難しい．しかし，C 言語は広く使われていることから，C 言語に特化した手法であっても，プログラム開発や研究に貢献する．特に，構文に対する経験的な知見は自明ではなく，構文規則のみから導出することは難しいことから，それらを明示し，また，実験によりその妥当性を確認することは重要である．

## 6. 関連研究

記号表の構成手法に関連する研究に，Knapen らの手法 [5] と，CScout [8] がある．手法 [5] では，ヘッダファイルが欠落した C++ プログラムを構文解析することを目的としており，識別子の定義が欠落することにより生じる解釈の多重化を，記号表を用いた手法で解消している．定義の欠落を許容する記号表を用い，2 パスの構文解析を行うことで，多重化した構文の解釈を一意に定める．しかし，手法 [5] は，構文として完全であることを前提としており，マクロの特殊な用い方などにより構文と合致しない記述を考慮していない．これを正確に解析するために，前処理後プログラムを解析対象としている．よって，前処理の条件分岐により無効となる箇所を解析できない点や，マクロの存在を考慮していない点が問題としてあげられる．本論文の手法では，解析対象のソースプログラムに誤りが含まれることを前提とした前処理前プログラムを解析するので，精度は劣るが，マクロを解析できることや，前処理の条件分岐による定義の多重化に対処できる利点がある．また，解釈の多重化を解消するだけでなく，文と宣言を誤った抽象構文木を補正できる特徴がある．CScout では，前処理前後のプログラムの対応関係を用いて，完全に近い記号表を構成可能にしている．また，プリプロセッサ演算子 ## を用いた文字列結合を追跡可能であるなど，マクロと識別子の対応関係を正確に扱える．これにより，見かけが異なる同一の識別子であっても，相互に参照可能としている．しかし，CScout では，前処理後プログラムの情報を用いるので，解析対象が構文として完全な状態であることが必要であり，また，前処理の条件分岐により無効となる箇所を解析できない．本論文の手法は，前処理前プログラムをそのまま解析することを前提としているので，完全な記号表を構成することはできないが，前処理後に無効となる箇所を解析可能である．また，前処理不可能なプログラム断片や，構文と合致しない記号を含むテンプレートなどを解析でき，適用可能な範囲に優位性がある．よって，独自に部品化したプログラム断片などの解析・書き換えには本論文の手法を用い，完成後のプログラムには CScout を用いるなどの使い分けによって，より精度の高い解析を行えると考えられる．

定義の欠落を前提とした構文解析手法として，C 言語以外の言語を対象とするものには，先述した C++ を対象とする Knapen らの手法 [5] と，Java を対象とする Dagenais らの手法 [2] がある．手法 [2] では，Java プログラムの断片において，クラスの定義が欠落することによって解釈が一意にならない構文を特定し，正しく構文解析するアルゴリズムを提案している．どちらの手法も，定義の欠落を許容した解析をしており，基本的な対応方法は本論文の手法と同じである．また，識別子の種別を推定するヒューリスティックなど，言語に依存する個別の対応は，対象とする言語に特化しているという点でも，本論文と共通する．ただし，手法 [5] では前処理後プログラムを対象としており，手法 [2] では Java を対象としており前処理が存在しない．よって，定義の多重化の問題や，マクロを解析することなど，前処理前特有の問題に対処していないという点で，本論文の手法と異なる．

プログラム断片を解析するアプローチには，PARSEWeb [9] がある．PARSEWeb は，プログラム断片の検索を目的としており，類似するものを発見するために，プログラム断片を構文解析する．PARSEWeb では，型が不明な識別子において，その識別子の前後の識別子から型を推定するヒューリスティックを用いている．本論文では，型が不明な識別子はすべて，そのまま不明としているが，同様の方法によって型を推定することは可能であると考えられる．ただし，構文解析に用いた TEBA は，式の解析に対応していないので，拡張が必要である．

前処理前プログラムを対象とした解析器には，srcML [1]，Yacfe [7] がある．srcML は，TEBA と同様に前処理命令を無視した構文解析を行い，XML 形式で抽象構文木を表現する．Yacfe は，マクロや条件分岐を用いた典型的な記述を抽出し，言語の構文規則に含める拡張を施すことで，前処理前プログラムをそのまま構文解析可能としている．どちらも応用例としてリファクタリングをあげているが，記

© 2013 Information Processing Society of Japan

号表を構成しないので，あるスコープ内の特定の変数名を一括して変更するなどの操作に対応できない．これらの解析器にも，本手法は適用できると考えており，その場合，リファクタリングの精度を向上させることにつながる．

拡張した TEBA の応用例として，前処理前プログラムに適用できるクロスリファレンサの実装が考えられる．拡張した TEBA を用いることで，前処理の条件分岐を選択する必要がなくなることや，部品化したテンプレートなど既存のクロスリファレンサが前提とするプログラム以外のものを対象に含められるといった利点がある．既存のクロスリファレンサには，SPIE [6] と GLOBAL [4] があげられる．SPIE は，CScout と同様に，前処理後の情報を用いて前処理前プログラムのクロスリファレンスを行うので，精度は高いが，適用できる範囲が狭い．また，前処理の条件分岐によって無効化された行を参照できない．GLOBAL は，前処理前プログラムをそのまま用いるので構文として完全である必要がないが，記号表を構成しないので，スコープ規則に基づいた識別子の区別が行われず，誤りを含むことがある．本論文の手法を用いることで，これらの欠点を解消できる．

## 7. おわりに

本論文では，前処理前プログラムに対する記号表の構成手法を提案した．プログラム断片などが対象であっても，近似解としての記号表を得られることが確認できた．記号表を用いることで，識別子の対応関係を明示でき，解析の精度を向上させた．応用例として，前処理前プログラムを対象としたクロスリファレンスツールの実装が可能である．また，データフロー解析に応用することで，前処理前プログラムのリファクタリングにつながる．

今後の課題として，マクロ展開後の字句列に対して記号表を構成することがあげられる．また，前処理の条件分岐における条件式の意味を解析し，識別子の有効範囲として用いることで，多重化を低減させることができると考えられる．

**謝辞**　本研究は，文部科学省研究費補助金基盤（C）（課題番号：22500036，22500037，24500049）の助成を受けた．

### 参考文献

[1] Collard, M.L. and Maletic, J.I.: Document-Oriented Source Code Transformatin using XML, *Proc. 1st International Workshop on Software Evolution and Transformation*, Vol.75, No.12, pp.11–14 (2004).

[2] Dagenais, B. and Hendren, L.: Enabling static analysis for partial java programs, *SIGPLAN Not.*, Vol.43, No.10, pp.313–328 (online), DOI: 10.1145/1449955.1449790 (2008).

[3] Free Software Foundation: Coreutils – GNU core utilities (online), available from ⟨http://www.gnu.org/software/coreutils/⟩ (accessed 2011-11-20).

[4] Free Software Foundation, I.: GNU GLOBAL source code tagging system (online), available from ⟨http://www.gnu.org/software/global/⟩ (accessed 2011-11-20).

[5] Knapen, G., Lague, B., Dagenais, M. and Merlo, E.: Parsing C++ Despite Missing Declarations, *International Conference on Program Comprehension*, p.114 (online), DOI: http://doi.ieeecomputersociety.org/10.1109/WPC.1999.777750 (1999).

[6] 大橋洋貴，山本晋一郎：SPIE – Source Program Information Explorer, Sapid Project (online), available from ⟨http://www.sapid.org/html2/mkSpec/SPIE-0.html⟩ (accessed 2011-11-20).

[7] 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, CC '09*, Berlin, Heidelberg, pp.109–125, Springer-Verlag (online), DOI: 10.1007/978-3-642-00722-4_9 (2009).

[8] Spinellis, D.: CScout: A refactoring browser for C, *Sci. Comput. Program.*, Vol.75, No.12, pp.216–231 (online), DOI: http://dx.doi.org/10.1016/j.scico.2009.09.003 (2010).

[9] 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).

[10] 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, No.7, pp.1832–1849 (2012).

**前林 達也**

2012 年南山大学数理情報学部卒業．現在，同大学大学院数理情報研究科博士前期課程在学中．

**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，IEEE Computer Society 各会員．

© 2013 Information Processing Society of Japan

**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．

**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 1988～1990 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．

© 2013 Information Processing Society of Japan