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]　吉田 敦[2]　蜂巣 吉成[2]　張 漢明[2]　野呂 昌満[2]

概要：プログラムの開発や保守の作業量を減らすために，書換え作業を自動化するための環境が提案されている．書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．

A Symbol Table Construction Method for Un-preprocessed Programs

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

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 could 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.

1. はじめに

プログラムの開発や保守は，プログラムの書換えの連続であり，書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．例えば，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．プログラムの書換えを自動化して開発支援を行うためには，この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[6] 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

[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
[2] 南山大学 情報理工学部
Faculty of Information Sciences and Engineering, Nanzan University

© 2012 Information Processing Society of Japan

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，解析の精度向上や，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対するクロスリファレンサを実装でき，部品化したテンプレートのような従来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も対象に含められる．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 節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 節でその具体的な構成手法を示す．4 節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 節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，コンパイル時に行な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，識別子の定義の欠落または重複や，前処理命令やテンプレートの記号など構文と合致しない表現を含む，C 言語のプログラム記述の断片である．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[6] が提案されている．

```
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

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できるとする．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 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，種別，属性値，文字列からなる属性付き字句の並びにより，抽象構文木と同等の情報を持つ．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，ヒューリスティックなルールに基づいて行われ，識別子の種別として，型，変数，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，この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

© 2012 Information Processing Society of Japan

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，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，区別できる．ただし，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．例として，`((sp)p + 1)->x;` と記述した場合，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 では部分式の解析を行っておらず，式の型を求められないので，本論文では対象としない．ただし，型が求められれば，構造体変数が記述されている場合と同様の方法で解析は可能である．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みなす，近似的な方法で構文解析を行う．これにより，前処理の条件分岐命令によって括弧の対応が取れない場合を除き，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，マクロの 3 状態と識別子を結びつけるフィルタを 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．

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

図 2 定義の多重化の例

2.3 記号表構成の問題

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得る．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．

識別子の定義がないと，型や有効範囲などの情報が欠落する．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．

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．例えば，**図 2** 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) の記述の解釈は関数呼出しのみではない．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) の文において，変数宣言の可能性を考慮せずに関数呼出しと解析している．(2) の場合も同様に，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．例えば `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

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

対処方法として，のちの解析を踏まえて不正確な情報を補正する方法と，重複した定義の登録を許容する方法が必要である．以下に，各問題点への具体的な対処方法を示す．

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．型は不明であり，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するなど，登録が混在しない構成が必要である．

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関わらず，それを無視して識別子だけに着目する．その例として，**図 3** 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 3 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．

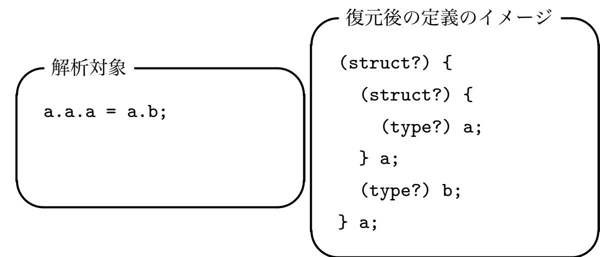

図 3　定義の復元の例

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．

種別の補正において，(1) “型”，(2) “関数”，(3) “変数”，(4) “種別が 1 種類に定まらないもの”，の順で優先順位を設定する．この優先順位は，字句の並びから確定しやすいものの順に並んでおり，順位の高いものから低いものへの補正を禁止することで，不正確な情報に補正することを防ぐ．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．識別子が型として出現する文脈では，解釈は常に一意であり，文法的な誤りが無ければ，その識別子の種別は型で確定できる．よって，種別を型に補正した場合は，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得ないが，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そのエントリを返すべきであるが，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 4** 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，該当するエントリーをすべてを参照し，それを統合した情報を取得する．図 4 の識別子 x は，種別として「型または変数」を持つ．

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，しかし，のちの解析で解釈が一意に定まる場合は，それに基づき補正可能である．よって，

© 2012 Information Processing Society of Japan

```
#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 4 多重化した定義の記号表での取扱い

```
foo (bar); /* 型または関数 */
foo baz;   /* 型に確定できる */
```

図 5 解釈の多重化の例

多重の解釈を許容する構文解析をしたのちに，記号表を構成し，解釈を一意に定める情報に基づき記号表を補正することでより正確な記号表を得られる．解釈の多重化は，前述の手順で得られた記号表を利用して再度解析することで解消する．

図 5 の例では，TEBA や srcML では字句の並びから，1 行目の foo を関数として解析する．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．

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 5 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，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 を拡張することで，本手法を実現した．TEBA の実装は，段階的詳細化に基づくフィルタ型の構成であり，記号表を構築するフィルタを解析器に組み込む拡張を施した．記号表を構成するフィルタと，記号表を基に解析結果を補正するフィルタを別々に実装しており，これらを段階的に適用することで TEBA の属性を補正する．

支援目的によっては，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[2] 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．評価基準は，定義の欠落や多重化を含むソースプログラムで識別子を正しく区別していること，記号表が型を正しく保持すること，解析を誤る箇所において記号表を用いて修正されることが挙げられる．

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 16405 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 17619 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区別されたことによる．拡張後の解析結果から，識別子 17619 個のうち，8976 個は型が特定されており，それらの型が正しいことを目視で確認をした．残りの識別子は，定義の欠落により型が不明となった．前処理を行ないヘッダファイルを展開すると，前処理前で型が不明であった識別子も型を特定できたが，記号表は解析対象プログラムと無関係の識別子を保持するので，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，16045 個中 893 個の識別子について種別が変更されていた．変更点は，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，曖昧な種別を許容する解析に変更したことによって種別が変更された点などである．しかし，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．解決策として，標準ライブラリで定義される識別子などに，あらかじめ種別を設定しておくことが挙げられる．また，「型または関数」のまま補正されなかったものは，一般的に関数として記述していると想定されるので，このような書き方の場合には関数に補正する方法が考えられる．

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 秒となり伸びているが，実用範囲内である．[*1]．

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

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

© 2012 Information Processing Society of Japan

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．例えば，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，ヒューリスティック補正を用いて特殊な事例に対処することで，正しく解析できる．

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，識別子の対応関係を求められる．また，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応用できると考えられる．

6. 関連研究

前処理前プログラムに対する記号表の構成手法に関連する研究として，CScout[5] 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．

本論文で提案する方法は，解析結果を用いた書換えを想定しており，その場合には前処理前プログラムをそのまま解析することが必要であるので，前処理を行なわない．前処理後では，前処理の条件分岐によって無効になる行があり，参照できなくなるが，本手法では，すべての分岐の記述を解析可能である．ただし，マクロの展開後の字句列や，前処理の条件分岐における条件式を対象としていない．

CScout は前処理可能なプログラムが対象であるのに対し，本論文で提案する手法では，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も対象としており，適用できる対象が広い．また，CScout は定義の欠落や多重化の問題を想定しておらず，対象のプログラムが構文として完全である必要がある．

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[4] と GLOBAL[3] が挙げられる．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挙げられる．また，前処理の条件分岐における条件式を解析し，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:21500042, 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, *Proceedings of 1st International Workshop on Software Evolution and Transformation*, Vol. 75, No. 12, pp. 11–14 (2004).

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

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

[4] 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).

[5] 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).

[6] 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).

© 2012 Information Processing Society of Japan