前処理前プログラムに対する記号表の構成方法に関する研究

2008MI045 平手 公巳　　2008MI122 前林 達也　　2008MI123 真野 智貴
指導教員　蜂巣 吉成

1　はじめに

書換え支援やリファクタリング支援を目的とした，構文として完全でない C 言語のソースプログラムを対象とする解析器が存在する [3][5]．前処理前プログラムは，識別子の定義を含まないなどの構文として不完全な状態である場合がある．前処理を行うと，マクロやヘッダファイルが展開され，本来のソースプログラムの表現が失われる．よって，書換え支援などを目的とするとき，解析対象は前処理前である必要がある．

前処理前プログラムを対象とする解析器は，構文として不完全なソースプログラムを解析可能としているが，構文として不完全な状態での記号表の構成方法は確立されていない．なぜなら，ソースプログラムで定義がない識別子が使用される状態や，定義が多重になった状態では，従来の方法で記号表を構成できないからである．

本研究の目的は，前処理前プログラムを解析対象とする解析器における記号表の構成方法を確立することである．記号表を用いない限り正確な解析を行えない構文的に曖昧な状態が存在する．曖昧さを取り除き，解析結果を一意に定めるには記号表が必須である．よって，解析精度を向上させるために，記号表の導入は有効である．

本研究での課題は，構文として完全でないソースプログラムから，近似解としての記号表を得ることである．そのために，欠落した定義を適切に復元し，多重の定義を不整合なく共存させることが必要となる．また，定義の欠落などの前処理前特有の問題とは別に，従来の記号表の構成方法を用いることができない問題がある．例えば，有効範囲の管理や，前処理命令の扱いが該当する．定義の欠落により，ローカルスコープ内で新規に大域変数が現れることや，前処理によって展開済みとなるべき前処理命令が残ったままになっていることは，従来の記号表の構成方法では考慮する必要がなかった問題である．本論文では，これらの問題を解消した前処理前プログラムに対する記号表の構成方法を示す．

2　前処理前プログラム

2.1　特徴

前処理命令には，#ifdef, #define などの，全部で 13 種類の命令が存在する．前処理前プログラムは，これらの前処理命令を実行する前のソースプログラムのことである．前処理前後のプログラムの例を図 1 に示す．

図 1 では，#define によってマクロ定義の展開が行われることや，#ifdef の条件分岐によって条件に合った範囲のみが有効になることで，元のソースプログラムの形が失われる．書換え支援などを行う解析器では，前処理前プログラムを対象とする必要がある．

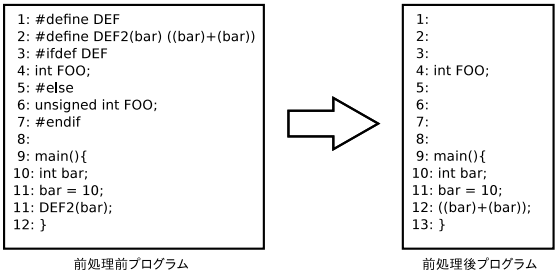

図 1　前処理前後のプログラムの比較

2.2　問題点

前処理前プログラムでは，ヘッダファイルを展開しないことで識別子の定義が欠落する場合や，前処理の条件分岐命令を実行しないことで，同一の識別子に対する定義が多重に存在する場合がある．このとき，識別子の定義と参照を一意に対応付けられないので，すべての識別子の対応関係が明確な記号表を構成できない．この場合に適用可能な記号表の構成方法は，確立されていない．C 言語の前処理前プログラムを対象とする解析器として TEBA や srcML が挙げられるが，これらは記号表を作成しない．

通常のコンパイラにおける記号表の構成方法では，定義が欠落した識別子を使用した場合や，登録済みの定義を多重登録する場合はエラーとなる．前処理前プログラムに対して，従来の方法を用いることはできない．

2.3　記号表の必要性

2.3.1　解析精度の向上

記号表を用いることで，名前空間やスコープ規則に基づいた識別子の区別や，識別子の型を保持できる．記号表を用いないと，図 2 の 2 行目のような複数の解釈ができる文に対し，解析を誤る可能性がある．

図 2 の例では，記号表がなければ，1 行目の foo と，2 行目の foo を対応付けられない．TEBA や srcML では，この場合，関数呼出しと解釈するが，記号表を用いた解析を行うことで，識別子の対応関係が明確化され，2 行目を foo 型の変数 bar の宣言として解析できる．

```
typedef int foo;
foo (bar);
```

図 2　複数の解釈ができるプログラム例

2.3.2 記号表の応用

書換え支援を想定した解析器を対象とするとき，前処理前プログラムのそのままの表現に対して記号表を構成することが必要である．前処理せずに記号表を構成することで，マクロを記号表で管理できるほか，型を用いたリファクタリングなどへの応用が可能である．

本研究で提案する記号表の構成方法を用いることで，前処理前プログラムを解析対象とした解析器を用いたクロスリファレンスツールの実装に応用できる．関連研究として，前処理前プログラムを対象とした既存のクロスリファレンスツールである SPIE[4] と GLOBAL[2] が挙げられる．しかし，SPIE は前処理の条件分岐により無効になった行を参照できず，GLOBAL はスコープ規則に基づいた記号表を構成しないので，同名の識別子はすべて同一であると判別される．

前処理前プログラムに対する記号表の構成方法を確立し，解析器に適用することで，前処理の条件分岐による無効な行を作らずに，かつスコープ規則に基づいた識別子の区別が可能になる．これを用いることで，SPIE と GLOBAL の欠点を解決できる．

3 記号表構成における問題

構文として不完全な状態から記号表を構成する場合，対象ソースプログラムに定義が存在しない「定義の欠落」と，定義が多重になる「定義の重複」の 2 つの問題がある．これらは，前処理前であることによって起こる．また，解析精度の低下を招く本質的な問題として，複数の解釈が可能な「曖昧な状態」がある．

3.1 定義の欠落

定義の欠落とは，解析対象ソース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る．文に出現する識別子は記号表を探索することで定義と結びつけられるが，定義が登録されていないので，対応付けができない．

定義がないとき，型は不明であり，構造体のメンバとフィールドの関係なども明示されていない．この状態であっても，適切に記号表を構成できる必要がある．

3.2 定義の重複

解析器は，前処理前プログラムをそのまま解析するので，前処理の条件分岐命令を実行しない．それにより，多重の定義がそのまま残る場合がある．これを定義の重複と呼ぶ．例えば，図 3 のように，名前，名前空間，有効範囲がすべて同じで，かつ型などが異なり完全に同一でない定義が存在することがある．この状態では，従来の方法によって記号表を作ることができず，重複した定義を共存させる方法が必要である．

3.3 曖昧な状態

複数の解釈が可能な曖昧な構文が C 言語では 3 種類あり，記号表なしでは解析を誤ることがある．記号表を作らない前処理前解析における問題である．図 2 の 2 行目のみを解釈する場合，foo の解釈は変数宣言のみではない．この場合，foo は関数，型，マクロの 3 つの解釈が考えられる．このような記述を複数の解釈が可能な文と呼ぶ．複数の解釈が可能な文は解析を誤る場合があり，実際に TEBA や srcML はこの文を関数呼出しと解析している．

4 記号表の構成方法

記号表の役割は，識別子を管理することである．そのための操作として，抽象構文木を走査し，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．構文として不完全なソースプログラムである場合がある前処理前プログラムにおいては，抽象構文木の走査と記号表の構成を構文解析後に行う必要がある．

本研究で用いる記号表で必要とされる情報は，クロスリファレンスに必要な範囲のものとして，識別番号，名前，型，種別，有効範囲の 5 つの要素とする．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が，そのような拡張は容易である．

4.1 各問題の対処法

4.1.1 定義の欠落の対処

定義が存在しない識別子が使用された場合の対処手順を次に示す．

- 字句の並びから推定し識別子の種別を詳細化
- 確定できる範囲で情報を登録し，併せて「欠落フラグ」を立てる

前処理前プログラムをそのまま解析するとき，言語の構文規則に従わない記述が含まれていても構文解析を行う必要があるので，構文的な誤りを誤りと見なさない．よって，経験則に基づいた規則により識別子の種別を補正する．補正は，字句の並びから識別子の種別を分類するものであり，4 種類の名前空間を区別するために必要である．規則は全体で 40 個あり，例えば，構造体に関しては次の例のような規則を含め 7 つある．

- ドット演算子，アロー演算子の直後の識別子は，メンバである
- struct の直後の識別子は，タグである

「欠落フラグ」とは，定義が欠落した識別子であることを明示するものとして用いる目印である．図 4 では，foo は定義が欠落しているが，字句の並びから種別が型であると推定でき，欠落フラグを立てて登録できる．

```
#ifdef DEF
typedef int x;
#else
typedef double x;
#endif
```

図 3 定義の重複の例

図 4　定義の欠落の対処例

```
foo (bar); /* 「型か関数」 */
foo baz;   /* 型に確定できる */
```

図 5　曖昧な状態の例

4.1.2　定義の重複の対処

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．すべての定義を多重に記号表に登録することで，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．

定義の重複の対処法として，重複フラグを用いる．重複フラグとは，名前，名前空間，有効範囲が同じ識別子を登録する際に，重複した定義が存在することを示す目印である．

定義の重複は，前処理の条件分岐内で起こり得るので，条件分岐内を走査していることをカウント変数を用いて判別する．これを用い，次の手順で定義の重複に対処する．

1. 条件分岐の開始 (#ifdef 等) の出現でカウントを 1 増やす
2. 識別子の定義が現れたら仮の記号表に記号表のエントリを保持
3. 同名の識別子を登録する場合，2 つ目以降の定義に重複フラグを立てる
4. 条件分岐の終了 (#endif) の出現でカウントを 1 減らす
5. 仮の記号表に存在する同名の識別子の情報をすべて合わせて 1 つにし，記号表に登録

仮の記号表とは，一時的に定義を保存する表である．前処理の条件分岐内での定義を仮の記号表に保存し，分岐を抜ける際に仮の記号表に含まれる重複した定義を組み合わせ，1 つにまとめた定義を記号表に登録する．これにより，多重の登録と参照を簡潔にできる．

4.1.3　曖昧な状態の対処

図 5 の例では，TEBA や srcML では字句の並びから，1 行目の foo を関数として解析している．しかし 1 行目の foo は，型の可能性を持つ曖昧な状態であるので，「型か関数」とすることが適切である．

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，あとから確定した情報を反映する必要がある．さらに，4 章で述べたように，記号表を構文解析後に構成しているので，構文解析を再度行うことによって抽象構文木を再構成する．これらの手順をまとめると，次のようになる．

1. 構文解析後の抽象構文木を入力とし，記号表を構成する
2. 後方で確定した情報を前方にも反映する
3. 再度構文解析を行う
4. 修正された抽象構文木を入力とし，再度記号表を構成する

曖昧な状態には，定義の欠落により種別の絞り込みが不可能であり，字句の並びからも特定が不可能な場合と，定義の重複により複数の種別の可能性を持ち，種別を確定できない場合がある．補正の対象となるのは前者のみであり，後者は曖昧な状態で確定する．

4.2　通常の構成方法と異なる点

本研究で提案する記号表の構成方法では，有効範囲の取り扱いや，前処理命令の取り扱いにおいて，通常のコンパイラにおける記号表の構成方法とは異なる．

4.2.1　有効範囲

C 言語では，中括弧で囲まれたブロックごとにスコープが決定される．通常の記号表構成方法では，ブロックの終わりに，ブロック内で定義された識別子のエントリを記号表から取り除くことで，大域と局所を区別する．

本研究での記号表構成方法では，定義が欠落した識別子を大域として記号表に新規登録する．しかし，従来の方法では，局所変数を登録したあとに大域変数が登録されることを想定していない．よって，次の手順での処理が必要となる．

1. スタックを用い中括弧の出現を管理する
2. 定義の有効範囲を，定義があった箇所の中括弧と対応付ける
3. ローカルスコープ内で出現した定義のない変数は，有効範囲を大域として登録する
4. ブロックの終わりに，記号表からそのブロックで定義された識別子のエントリのみを記号表から取り除く

4.2.2　前処理命令

前処理前ではマクロが展開されないので，これを記号表によって管理する必要がある．#define 命令は，変数宣言と同様に識別子の定義であるので，マクロを記号表に登録する．これにより，定義があるマクロについては，種別をマクロに補正できる．

#undef 命令によりマクロ定義の削除が可能であり，この命令に従い，該当したマクロを記号表から削除する．ただし，前処理の条件分岐中で #undef 命令が使用された場合，条件によってマクロが削除されない場合があるので，記号表からも削除しない．

4.3　実装

TEBA を図 6 のように拡張して実装した．拡張箇所において，解析の順序は 4.1.3 節で述べたとおりであり，

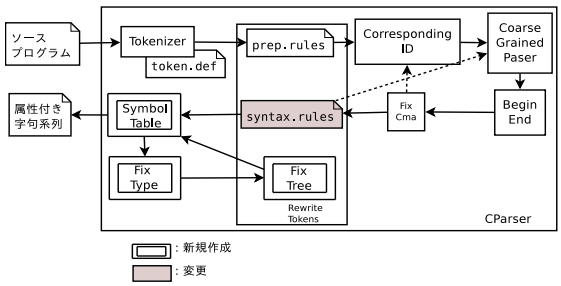

図 6　拡張後の TEBA の全体像

Symbol Table は手順 1 または 4，Fix Type は手順 2，Fix Tree は手順 3 に対応する．

5　評価と考察

5.1　評価方法

評価には，拡張する以前の TEBA と，本研究で拡張を行った TEBA を用い，記号表構成前後での解析結果の変化を評価する．

5.1.1　構成方法の評価方法

各問題が発生する状況を再現したテストケースを作成し，その解析結果を「解析可能」「解析不可能」「対象外」の 3 段階に分類することで，現実的な記述に対し記号表を構成可能であることを確かめる．

同じソースプログラムについて，構成される記号表が前処理前後でどのように異なるか確かめる．前処理後であれば，ソースプログラムは構文として完全な状態となり，完全な記号表が作成される．前処理前後の記号表の一致する割合が高ければ，解析精度が高いことになる．

5.1.2　解析結果の評価方法

GNU coreutils 8.9[1] を用いた評価で確かめる．解析結果が次の評価基準を満たすことを確かめる．

- 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
- 記号表無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
- 同一の識別子は，同じ種別に集約する

5.2　評価結果

評価対象として設けたテストケースは現実的な記述をほぼ網羅しており，これらに対し適切に記号表を構成可能であることを確認した．前処理前後で構成された記号表は，解消できない曖昧な状態が増えるにつれて一致する割合が低下したが，曖昧な状態が含まれなければ同じ意味の記号表を構成可能であった．また，前処理後では失われたマクロを，前処理前では記号表に保持できた．

拡張前後の TEBA の比較では，GNU coreutils の cp.c で 261 個ある識別子のうち，49 個が正しい種別に補正された．ID 付与は，目視での確認であるが，誤りは確認されなかった．

5.3　考察

識別子の定義が欠落，または重複していた場合でも，記号表を構成可能であった．これにより，解析精度の向上に寄与した．また，記号表を用いることで，同じ識別子に対して同じ ID を適切に割り当てられた．しかし GNU coreutils の解析では，平均して 7 割程度の識別子の定義が欠落しており，型を得られないものが目立った．

マクロの記述が C 言語の文法に反している場合，解析を誤り，正しく記号表を構成できない．実際に GNU coreutils では，`int a IFLINT ( = 0);` のような記述が用いられており，この場合，int a 型の関数 IFLINT のプロトタイプ宣言であると誤って解析される．

6　おわりに

本研究では，前処理前プログラムに対する記号表の構成方法を提案した．その結果，前処理前プログラムの解析器における解析精度は向上し，識別子の対応関係を表す識別番号や型など，情報量が増加した．これらの情報を用い，前処理前で実行可能なクロスリファレンスツールの実装などへ応用できると考えられる．

今後の課題として，マクロの記述が文法に合わない場合の対処と前処理の条件分岐命令における条件式の評価が挙げられる．文法に誤りがあると種別の推定を誤ることがあり，意図しない補正によって解析に不整合が生じた．よって，誤った補正を防ぐ方法が必要である．また，本研究では，前処理の条件分岐命令の存在は意識しているが，その条件分岐は意識していない．識別子の有効範囲として前処理の条件分岐を適切に取り入れることで，曖昧な状態をさらに絞り込めると考えられる．

参考文献

[1] GNU Project, “*Coreutils - GNU core utilities*,” http://www.gnu.org/s/coreutils/, Oct. 2011.

[2] GNU Project, “*GNU GLOBAL source code tag system*,” http://www.gnu.org/s/global/, Oct. 2011.

[3] J.I. Maletic, M.L. Collard, and A. Marcus, Source Code Files as Structured Documents, Proceedings of the 10th IEEE International Workshop on Program Comprehension (IWPC’02) Paris, France, pp.289-292, Jun. 2002.

[4] 大橋洋貴，山本晋一郎，“*SPIE – Source Program Information Explorer*,” http://www.sapid.org/html2/mkSpec/SPIE-0.html，May. 2005.

[5] 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，Aug. 2010.