誤解の解消のためのソースコード等価変換手法の提案

2020SE071 所孝樹　2020SE086 福島諒大

指導教員：吉田敦

1　はじめに

ソースコードを読むとき，言語仕様の不理解や見落としなどにより誤解が生じる．誤解が生じると，どこが間違っているかを特定することが難しく，長時間悩むといった問題が生じる．誤解が生じる原因は人間の記憶の曖昧さである．プログラムの意味や動作を理解するために，構文木や制御フローグラフに相当するイメージを頭の中に描くが，演算子の優先順位のように経験的に漠然と把握しているものや，多数の演算子で結びついた式があることで曖昧になりやすい．また，多数の変数が存在すると，変数と型の関係の理解も曖昧になりがちである．

記憶の曖昧さを解消するためには，読み手が把握すべき情報を減らすことが必要である．それを裏付けるものとしてワーキングメモリの考え方がある．ワーキングメモリとは，行動を計画し実行するために使用される記憶領域である [1]．つまり，人が何かを目的として思考するときに，一時的に情報を保持するメモリであり，その容量には限界がある．この容量不足が誤解を生み出す原因だといえる．読み手が把握すべき情報が多いほど，その情報を処理する過程での保持すべき情報が多くなり，ワーキングメモリに負荷がかかる．

本研究では，誤解を減らすためのソースコードの等価変換手法を提案する．ここでの等価変換とは，プログラムの動作を変えることなく，より単純な式や文で構成し，構文を満たせる範囲で情報を明示するものである．等価変換を用いる理由は，ソースコードが持つ情報を欠落させることなく別の表現にすることで，自分の理解と異なる箇所に気づきやすくするためである．また，読み手が把握すべき情報が少なくなり，必要となるワーキングメモリの量を減らす効果もある．ただし，等価変換は抑制的に適用されることが必要である．ソースコード全体に対して変換を適用したり，一部分に複数の種類の変換を適用したりすると，変換前後の乖離が大きいことで，対応関係が取りにくくなり，誤解の原因に気づきにくくなる．このことから，等価変換をしつつも，抑制的に適用させる必要がある．

本研究の課題として，以下の 2 つが挙げられる．

- 誤解を減らす等価変換にはどのようなものがあるか
- 抑制的に変換を適用するためのユーザインターフェイスはどのようなものか

1 つめの課題に関して，単に等価変換をするといっても，どのような変換でも良いわけではない．読み手が誤解しやすい，あるいは，理解にあたってワーキングメモリの負担が大きい箇所を変換するべきである．また，誤解をしやすくても等価変換ができない場合も想定される．等価変換でできる支援で何ができるかを明確にする必要がある．この課題に対して，理解を困難にする要因を分析することで，その要因を減らす等価変換方法を整理する．

2 つめの課題に関して，既に述べたように，抑制的に変換をしなければ，変換前後で乖離が進み対応関係が取りづらくなる．抑制の方向性は，変換箇所を絞ることと，変換を解除することの 2 つに分けられる．前者に関して，本研究ではソースコードを操作対象としているが，構文要素を単位として，個別に操作できるべきである．しかし，その中には式や文といった，構文木において非終端要素と呼ばれるものがある．その存在範囲は直接的に見えないので，字句からの指定がしにくい．また，非終端要素を指定できたとして，そこからなる部分木全体に変換を適用したいときとある程度まで適用したいときがある．ある程度まで適用したいとき，部分木全体に変換が適用されると，読み手にとって不要な情報が生まれ，過度な変換となる．後者に関して，等価変換では記述量が増え可読性が下がる．よって理解して不要になった変換箇所は元に戻し，可読性を保ちたい．これらの課題に対して，非終端要素を可視化し，変換を段階的に適用するための仕組みや適用した変換を解除できる仕組みを構築することでアプローチする．

2　関連研究

立道ら [2] は，プログラム記述の操作支援を目的として，Web ページ記述内のプログラム解析手法を機能として実現する中で，構文解析により得られた構文木を DOM 木に変換する方法を提案している．本研究の実装では，構文木を DOM 木に変換し，実際に DOM 木の操作を行うことによりソースコードの操作を実現する．

TEBA[3] は吉田らによって提案された，ソースコードを属性付き字句系列に変換するプログラム開発支援環境である．属性付き字句系列に対する処理として，プログラム解析やプログラム変換ができる．また，属性付き字句系列というテキスト形式で抽象構文木を構成するため，ツール構築に必要な基礎知識が少なくてすむほか，複数のツールを組み合わせて大きな処理を実現しやすい．本研究では，TEBA の機能を用いてソースコードを AST に変換する．

3　提案方法

3.1　概略

誤解の具体例をソースコード 1, 2 に，それに対する変換例をソースコード 3 に示す．

ソースコード 1　誤解する例 [4] の csplit.c を改変

```
1 if (*f == '%' && *++f != '%')
2         return end;
```

ソースコード 1 では，if 文の条件式内に存在する論理積演算子の短絡評価により，制御が複雑になっている．論理積演算子の左式が偽であった場合，右式を評価することなく条件判定が終了する．この知識が曖昧だった場合，ソースコード 2 のように誤解することがある．

ソースコード 2 誤解の例

```
1 a = (*f == '%');
2 b = (*++f != '%');
3 if (a){
4     if(b)
5         return end;
6 }
```

ここでは if 文の条件式の評価において，常に右式も評価され変数 f の値が変化するという誤解をしている．一方，論理積演算子を使わないようソースコード 1 を変換すると，ソースコード 3 のようになる．これにより右式が評価されるのは左式が真のときのみということが表現され，誤解に気づく．

ソースコード 3 制御構造の明確化の例：変換後

```
1 if (*f == '%') {
2     if(*++f != '%')
3         return end;
4 }
```

このような誤解の解消のために等価変換によりアプローチしていく．

本研究の提案手法では，AST の等価変換を行う．ユーザはソースコードを読んでいるので，ソースコードのテキストに対して操作するが，本質的には AST を操作する．ユーザがテキストに対して行う操作イベントに対して，ノードごとに設定された構文木の変換処理が実行される．

3.2 等価変換

本研究では，等価変換操作として次の 2 種類を採用する．

- 追加操作
- 構造操作

追加操作とは，ソースコードの木構造に対して，既に存在するノードを書き変えることなく，新しいノードを付け加える方法である．構造操作とは，ソースコードの木構造に対して，部分木を新しい部分木で置き換える方法である．ただし，等価を保つためには，実行に直接的に影響しない範囲での変換が求められる．

3.2.1 追加操作

追加操作の変換の種類として，「型の埋め込み」と「文法的な構造の明示」の 2 種類に分類する．型の埋め込みの例をソースコード 4 とソースコード 5 に示す．

ソースコード 4 型の埋め込みの例：変換前

```
1 double sum;
2 int n, ave;
3 (省略)
4 ave = sum / n;
```

ソースコード 4 では，変数の宣言と参照の位置が離れているので，式を読む時点で変数の型が何か覚えていないという問題がある．その要因として，宣言と参照の間の行数が多いことや，その間で多くの変数が使用されていることが挙げられる．また，代入式において右辺の式の型が左辺とは異なるときに，暗黙の型変換が起こることがあるが，コンパイラは警告を出さないので，その型変換に気づかないという問題がある．これに関して，それぞれの変数の型を覚えていないと，暗黙の型変換が行われることに気づきにくい．例えば，ソースコード 4 の 4 行目の代入式では，左辺は int 型だが，右辺は double 型で，ここで型変換が起きているが，右辺が double 型と覚えていると，左辺も double 型だと誤解することがある．これを，等価変換により，ソースコード 5 の状態にする．

ソースコード 5 型の埋め込みの例: 変換後

```
1 double sum;
2 int n, ave;
3 (省略)
4 (int)ave = (double)(sum / n);
```

ソースコード 5 では，ソースコード 4 に対して追加操作をし，変数と式の型を表している．これにより，宣言と参照の位置が離れていても，各変数の宣言を調べ直すことなく，型の誤解に気づける．文法的な構造の明示の例をソースコード 6 とソースコード 7 に示す．

ソースコード 6 文法的な構造の明示の例：変換前

```
1 if (a != b && b != c)
2     return b;
```

ソースコード 6 では，1 つの文に式が複数存在しており式と式の構造を把握しにくい．言語仕様を理解していないことや見間違いによって，論理積演算子を先に処理するといった誤解が生じる．これを，等価変換により，ソースコード 7 の状態にする．

ソースコード 7 文法的な構造の明示の例: 変換後

```
1 if ((a != b) && (b != c))
2     return b;
```

ソースコード 7 では，ソースコード 6 に対して追加操作をし，構文の木構造を括弧で示すことで，演算子の優先順位の知識がなくても，式の構造を正確に把握できる．

3.2.2 構造操作

構造操作による変換の種類として，「宣言の細分化」と「制御構造の明確化」の 2 種類に分類する．宣言の細分化の例をソースコード 8 とソースコード 9 に示す．

ソースコード 8 宣言の細分化の例：変換前

```
1 int * p, q;
```

ソースコード 8 には，左側の変数 p だけがポインタ型の変数として宣言されている．しかし，この書き方をすると，変数 p だけでなく変数 q もポインタ型の変数として宣言されていると誤解しやすい．これを，等価変換により，ソースコード 9 の状態にする．

ソースコード 9　宣言の細分化の例：変換後

```
1 int * p; int q;
```

このように宣言を分割することで，それぞれの変数がどのようなものなのかを吟味する必要がなくなる．

制御構造の明確化について示す．ソースコード 3 は，ソースコード 1 を変換したものである．論理積演算子によってまとめられていた条件判定式を分割し，2 つの if 文で入れ子構造にしている．これにより，短絡評価による制御構造が明確化し，誤解したときに期待する変換との差異から誤解を減らすことができる．

3.3　操作対象の指定方法

抽象構文木 (AST) には，終端要素と非終端要素が存在する．本研究で提案するシステムは，ユーザがソースコード上の字句を指定して操作を行う．終端要素に対する操作は，字句を指定することで実現できる．ただし，字句は終端要素と 1 対 1 に結びつくので，非終端要素の指定に字句を用いることが難しい．すなわち，字句を指定したときに，その字句に対応する終端要素を選択しているのか，また，そうでないときは，終端要素から根に向かうまでの，どの非終端要素を指定しているのかを判定する必要があるが，それは難しい．また，式や文といった非終端要素は通常，複数の字句 (終端要素) で構成されるが，その範囲が必ずしも明示されていない．したがって，非終端要素がどの終端要素からどの終端要素まででで構成されているのかという構造を可視化する必要がある．

そこで，非終端要素に対する操作は，構造の可視化を行い，可視化に用いた表現要素を指定することで実現する．具体的には，非終端要素の周りを終端要素の括弧で囲む．その流れを以下で説明する．まず，非終端要素を指定するための仮の終端要素となるものを作る．次に，作った終端要素を非終端要素の兄弟ノードとして挿入する．兄弟ノードの順序はテキストで書かれている順序と同じである．それを利用して，非終端要素の兄と弟として括弧を挿入する．ここでの兄とは，ソースコード上での出現順において，その直前の位置になる兄弟ノードであり，弟はその逆である．これにより，括弧で囲んだ範囲が非終端要素であることを可視化する．文の場合，文であることが容易にわかることと，文を括弧で囲む構文は存在しないことから，弟として終端要素を挿入し，文の終わりに存在するその要素から文を指定する．

3.4　抑制的な変換操作

1 章で述べたように，本来のソースコードの解釈を妨げないように変換する必要がある．変換を抑制的に行うために，次の 2 つの方法を用いる．1 つ目は，指定された要素の子孫の深さに基づく反映の抑制である．非終端要素に対する操作の場合，子孫が存在する．変換を適用する子孫の深さを指定しておくことにより，変換の反映される範囲を抑制する．2 つ目は，操作の適用の解除である．そのために，変換操作を適用する際，変換前のノードや部分木の構成を保持しておく．また，変換により追加されたノードや部分木は，変換により得られたものであることを識別するための属性を追加することで，変換操作の適用の解除を実現する．

3.5　等価変換のための構文木の構成

本研究では AST の操作によるソースコードへの等価変換の適用を提案している．ただし，AST をそのまま表現するだけでは，3.4 節で提案した操作の適用の解除に対応しにくい．ある変換操作を定義すると，その逆の変換の操作を定義する必要がある．しかし，構造操作においては，逆変換の対象が部分木全体に及び，どの範囲を戻せばよいかわからなくなることや，どこを変換したかを覚えておくことによる記憶量の負担を増やすことにつながる．その問題を解決するため，AST の部分木を構成するときに，その根の下に本来の部分木と変換後の部分木を持つようにする．図 1 は，そのイメージである．

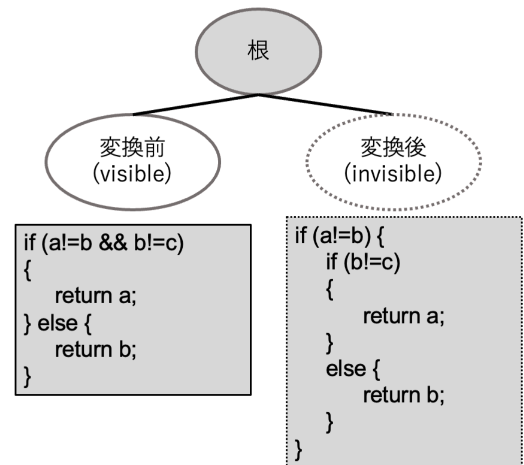

図 1　変換対象となる部分木の構成イメージ

根は変換前後の部分木を子に持ち，変換した部分木は不可視の状態で埋め込む．図 1 では仮に，不可視の状態を点線で表す．変換前後の部分木の可視状態を切り替えることで，ユーザに見えているソースコードの変換を実現する．

4　実装

4.1　ユーザインターフェース

提案手法に基づいた支援ツールを，Web ブラウザ上で動作する JavaScript のプログラムとして実装した．

1 つの構文要素には，複数の操作が適用可能である．どの操作を適用するかを指定するために，チェックボックス

を用意した．チェックが外れた際，それに対応する変換は解除する．チェックボックスは，操作のしやすさの観点から，常時画面上に見えている必要がある．そこで，iframeによりウィンドウを分割し，常に画面右側に表示されるようにした．図 2 は実際の画面である．

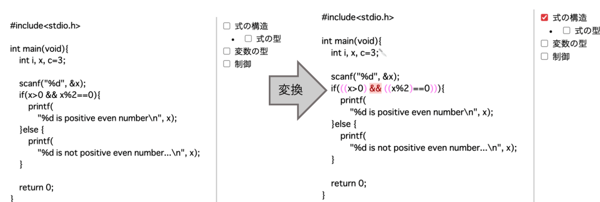

図 2　実際の画面

4.2　DOM 木の構成方法

本ツールでは，等価変換を DOM 操作により実現した．TEBA を用いてソースコードを AST に変換し，AST の各ノードを DOM のノードに変換する．ここで，等価変換をするために，3 つの構文要素に対して AST での構文木構造と異なる構造を構成している．1 つは演算子である．TEBA の AST において，演算子の字句は非終端要素に属するが，字句はすべて終端要素としたいので，演算子も終端要素となるよう構成を変更している．if 文や宣言文の場合，非終端要素であるので，ソースコード上で指定するための終端要素を作り，兄弟要素として挿入する．

DOM として構成したのち，4.1 節で述べたチェックボックスの状態に応じて，イベントハンドラの追加・削除をする．追加する要素は変換する対象の構文要素ごとに異なっている．その要素は，変換対象が終端要素であればそこに，非終端要素であればそれを示すために新しく作る．図 2 にあるナイフの絵文字は，直前の宣言文を指定してイベントを起こすために作った要素である．

構文木を用いた等価変換には，追加操作と構造操作がある．追加操作においては，イベントを受けるたびに追加する要素を生成，追加している．この操作で追加する字句は，現状括弧と型情報のみであり小規模な木の操作である．構造操作においては，3 章で述べたように，部分木を置き換えるので，大規模な変換になりやすい．そのため，部分木の中にも追加操作の対象が含まれていることが多い．追加操作をした要素が含まれる部分木に対して構造操作をすると，追加操作の要素が干渉してうまく変換できないことがある．したがって，構造操作を先に行い，あらかじめ変換した部分木を作った状態にしている．

5　評価

本研究では，提案した等価変換のうちの一部を実装し，DOM による構文木の操作を通したソースコードの等価変換ができることを確認した．その中で，各構文要素を指定した変換および，変換の解除，子孫の深さに基づいた抑制的な変換を適用できた．また，10 個のソースコード片に対し誤解の要因を分析し，等価変換でその要因が取り除けることを確認した．そのうち実装した if 文の条件式が論理積演算子だった場合の構造操作による等価変換を，Coreutils[4] の dd.c において見つかった論理積演算子を条件式に含むソースコード片 43 個に適用できることを確認した．そのうち 1 つのソースコード片において，含まれる論理積演算子をすべて変換できないことを確認した．これは，条件式内の論理積演算子よりも先に発見される演算子に対する操作が未定義であることで起きたと考えられる．

本ツールでは，条件式の部分木を探索する際，演算子が見つかったのち，次の兄弟要素が式か変数であった場合にその演算子に対応した構造操作をする仕組みになっている．しかし，見つかった演算子に対する操作が定義されていなかったとき，探索を続けることができない．その演算子を無視して次の演算子を探索しても，無視した演算子が次の演算子を含む式全体に作用することから，論理が変化するので変換できない．したがって，すべての演算子に対応した操作を定義しておく必要がある．

6　考察

本研究では，提案した等価変換の一部を実装した．等価変換には追加操作と構造操作の 2 つがあることから，構文木に対する操作のフレームワーク化が可能であると考えられる．プラグイン式に対応したフレームワークを開発することで，今回実装しなかった等価変換の実現が容易になり，汎用性が向上すると考えられる．その際，システムとして組み込むには，各変換用のチェックボックスの変更と，変換にあたって対象となる構文要素の持つ属性やノードを取得するためのメソッドを記述することが必要になる．

7　おわりに

本研究では，誤解を解消するための等価変換手法について提案した．プラグインの仕組みを作ることにより，拡張性の高いシステムにすることが今後の課題である．

参考文献

[1] Nelson Cowan: “What are the differences between long-term, short-term, and working memory?”, Progress in Brain Research, Vol.169, pp.323–338 (2008).

[2] 立道 昂太, 吉田 敦, 蜂巣 吉成, 張 漢明, 野呂 昌満：Web ページ記述内のプログラム断片に対する DOM tree を用いた構文木の構成手法，ソフトウェアエンジニアリング 2012 論文集, pp.1–6 (2012).

[3] 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).

[4] “Coreutils - GNU core utilities.”, https://www.gnu.org/software/coreutils/coreutils.html