プロジェクト内で共通するプログラミング知識に基づいたソースコード簡略化手法

M2017SE004 加藤宗一郎
指導教員：吉田敦

1 はじめに

オープンソース・ソフトウェアや GitHub[1] のようなオープンなリポジトリサイトの普及により，プロジェクトに関心のあるプログラマが機能追加や不具合の修正での貢献が容易になった．また，ソースコードを変更しても公開する必要のないプロジェクトも存在し，企業の製品に組み込まれることも増えている．

プロジェクトに途中から参加する場合や，ソースコードを修正して製品に組み込もうとした際には，まず，プロジェクトのソースコードの構成や，主要な処理の流れなどの概要を把握する必要がある．しかし，実用的なソフトウェアの場合，その規模は大きく，ソースコード全体を把握することは容易でない．関数の呼び出し関係や制御フローの可視化によって，プログラムの制御構造の理解支援を行うツールが存在する [2]．しかし，ソースコードの編集を考えると，ソースコードとそれを抽象化したものを見比べながら作業するより，それらが一体になった表示の方が理解が進み，作業は容易になる．さらに，プロジェクト内で共通するプログラミング知識に基づいて記述されたコード片は，すでにプログラマが理解しているのであれば，その詳細を確認することなく，ソースコードを読み進めることができる．

本研究の目的は，ソースコード全体の基本的な処理の理解を支援するために，プロジェクト内で共通なプログラミング知識に基づいて記述されたコード片を簡略化して表示する方法を提案することである．技術的課題は，プログラミング知識をどのように獲得するか，さらに，その獲得したプログラミング知識に基づくコード片をどのように探すかの 2 点である．プログラミング知識そのものを獲得することは困難であるので，本研究では，プロジェクト内で共通のプログラミング知識に基づいた共通な記述が，プロジェクト内のソースコードに複数存在することに着目し，近似的に，プロジェクト内で出現するソースコード群に共通な表現の集合として獲得する．さらに，簡略化対象とするコード片を求めるために，共通な表現から構成される末端の文を特定したうえで，構文木上で，ボトムアップに簡略化対象を広げていく．最終的に，簡略化対象とした文を簡略化表記に変換してプログラマに提示する．

本論文の構成は次の通りである．2 章でプロジェクト内で共通するプログラミング知識について分析し，3 章で共通のプログラミング知識に基づいたソースコードの簡略化手法を提案する．4 章で提案手法の効果を評価する．5 章で関連研究について議論をし，6 章でまとめを述べる．

2 共通するプログラミング知識

2.1 プロジェクト内で共通なコード片

プロジェクト内での共通な知識について整理する．ソースコードを理解する上で前提となるプログラミング知識として，(1) プログラミング言語の文法，(2) 標準ライブラリや各種の外部ライブラリの使い方，(3) 開発ツールの使い方，(4) 特定のプロジェクトやフレームワークにおける規約が挙げられる．

本論文では，コード片の表現に盛り込まれる (2) と (4) を獲得するべきプログラミング知識とする．ここでの規約とは，ある処理を実現するために守るべき規則を指す．例えば，例外処理やオプション処理を記述する際に，呼び出すべき関数が決まっている．このような処理は，特定の手続きや関数に依存しないので，複数のソースファイルにまたがって出現する．例えば，Coreutils[3] では，プログラムの初期化設定の規約として，プロジェクト内の複数のソースファイルに共通して Listing 1 のような処理が記述される．このような初期化処理は，実行環境の差異を隠蔽するように定義されるので，よく似た記述のコード片となる．

Listing 1 Coreutils における初期化設定

```
1 initialize_main (&argc, &argv);
2 set_program_name (argv[0]);
3 setlocale (LC_ALL, "");
4 bindtextdomain (PACKAGE, LOCALEDIR);
5 textdomain (PACKAGE);
6
7 atexit (close_stdout);
```

2.2 ソースコードの簡略化による理解支援

初めてソースコードを読むときには，まず，全体の処理の流れを確認することが必要である．そのときに， プログラマが着眼している主要な処理の流れとそれ以外に分けて，主要な流れのみをプログラマに提示できれば，プログラマの理解の負担は軽くなる．しかし，どの処理が主要な流れに対応するのかを判定することは難しい．ただし，少なくともプログラマが理解しているプログラミング知識に基づいた処理の詳細は，着目すべき処理ではない．そこで，プロジェクト内で共通な知識に基づいて記述されたコード片を簡略化して，主要な処理を把握しやすくする．このとき，簡略化するコード片を単純に隠すと，逆に理解が困難になるので，そのコードの内容を想起できる表現が必要である．

プログラミング知識に基づくコード片を特定するためには，ソースコード群内で共通する表現を多く含むコード片を特定する必要がある．あるコード片が共通する表現を含

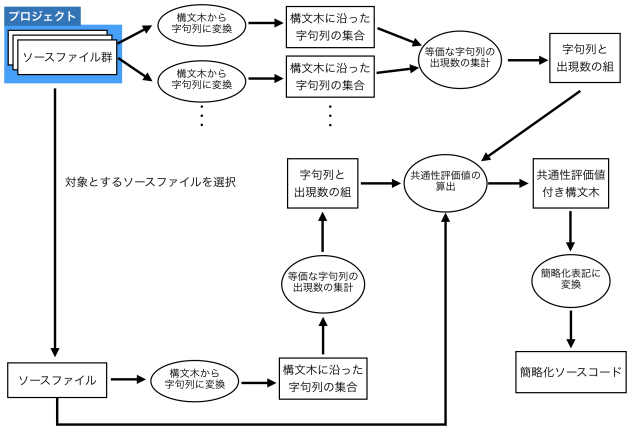

図 1　提案手法の概要

むかどうかは，その表現の構成要素を比べて判定する．表現の構成要素として字句が考えられるが，単に字句を用いると，字句の構成が同じでも内容が異なるものも共通していると判定することになり，望ましくない．字句の並びの関係も考慮すべきであり，自然言語処理で用いられる N-gram の考え方の方が望ましい．ただし，自然言語処理では文字の並びを対象とするが，ソースコードは構造化された記述であり，構文要素の親子関係の共通性も考慮すべきである．自然言語における N-gram を参考としつつも，構文木を前提とした字句の並びを比較し，類似しているかどうかを判定する必要がある．

簡略化する要素の単位として文が考えられるが，連続する複数の文が対象となることがあり，また，制御文などによる文の包含関係もどこまで一体と考えるかは難しく，すべての組み合わせを試すことは現実的ではない．そこで，構文木上の末端の文から判定し，葉から根に向かいながら候補を決定していく．

3　共通する知識に基づく簡略化手法

本論文における提案手法について説明する．提案手法の概要は図 1 の通りである．本手法は，大きく次の 2 つの手順によって構成される．

1. ソースコード群から，各構文木ごとに字句列を求め，その出現数を算出
2. 簡略化したいソースコードで，出現数の多い字句列で構成される文を簡略化候補としてボトムアップに決定

以下では，2 つの手順に共通する構文木に沿った字句列の生成について説明し，各手順について説明する．

3.1　抽象構文木に沿った字句列

まず，字句列で用いる字句について説明する．ここでの抽象構文木は，識別子，値，演算子，関数呼び出し，文で構成されたもので，それぞれ構文要素を代表する字句を持つものとする．ただし，一部の要素は例外的に扱う．式の構成要素のうち，関数呼び出しだけは，関数名が単一の識別子で構成されるときに限り，関数呼び出しの要素を代表する字句を関数名とし，関数名を参照する式を無視する．複合文は，複数の文を持つという情報しか持たないので，抽象構文木に沿った字句列を生成する際には省略する．式文や制御文はセミコロンや if, while などの予約語を持つが，それらは使用しない．

次に，抽象構文木に沿った字句列を抽出する方法について説明する．まず，ある構文木 T の根の構文要素 e について，それぞれ葉に向かう経路に N 個の構文要素を取得し，それぞれ構文要素列として生成する．そのあと，e の子要素についても，再帰的に N 個の構文要素列を生成する．以下，この構文要素列を N-gram と呼ぶ．

構文要素は，それぞれ独立したオブジェクトであり，区別される．本論文では，共通な表現を求めるにあたって，共通な字句を持つことを前提とするので，構文要素列の等価性は，各構文要素をそれぞれ代表する字句の並びが等しいことと定義する．以降では，抽象構文木から抽出した構文要素列は，特に必要がない限り，字句列と同一のものとみなして説明する．

3.2　ソースコード群からの字句列の抽出と出現数の集計

ソースコード群の各抽象構文木から抽出した N-gram の出現数の求め方について説明する．ある N-gram を x とする．x は 3.1 節で述べたように，オブジェクトであり，x が属するファイルは一意に決まる．x が属するファイルを $F_s(x)$ と定義する．また，プロジェクト内で共通するプログラミング知識を取り出すために用いた全ソースファイルの集合を F_{all} と定義する．ファイル集合 F に出現する x と等価な N-gram の出現数を $t(x, F)$ と定義すると，全ソースファイルでの x と等価な N-gram の出現数は $t(x, F_{all})$ と表される．また，x が属するファイルに出現する x と等価な N-gram の出現数は，$t(x, \{F_s(x)\})$ となる．x の共通出現値 $c(x)$ は，式 (1) のように定義する．

$$c(x) = t(x, F_{all}) \times w(x) \tag{1}$$

ここで，$w(x)$ とは，偏って出現する字句列の影響を軽減するための重みであり，式 (2) のように定義する．

$$w(x) = 1 - \frac{t(x, \{F_s(x)\})}{t(x, F_{all})} \tag{2}$$

3.3　簡略化ソースコードの生成

3.2 節で定義した共通出現値を用いて簡略化対象とする文の候補を決定し，簡略化表記に変換する方法を説明する．以下の手順で簡略化ソースコードを生成する．

1. 末端の文および条件式ごとに，そこに含まれる字句列の共通出現値の平均を求める
2. 共通出現値の平均値が閾値以上なら簡略化候補と設定
3. 根に向かう経路上の各要素について，簡略化候補の要素を一定の割合以上含む要素を簡略化候補と設定
4. 構文木からソースファイルを再構成する．そのとき，

```
while ((opt = getopt_long (argc, argv, "diw:", long_options, NULL)) != -1)
  switch (opt)
    ▶{ ... }

if (argc - optind > 1)
  ▶{ ... }

if (optind < argc)
  infile = argv[optind];
else
  infile = "-";

if (STREQ (infile, "-"))
  {
    xset_binary_mode (STDIN_FILENO, O_BINARY);
    input_fh = stdin;
  }
else
  {
    input_fh = fopen (infile, "rb");
    ▶if(input_fh == NULL) ...
  }

fadvise (input_fh, FADVISE_SEQUENTIAL);

if (decode)
  do_decode (input_fh, stdout, ignore_garbage);
else
  do_encode (input_fh, stdout, wrap_column);

if (fclose (input_fh) == EOF)
  {
    if (STREQ (infile, "-"))
      die (EXIT_FAILURE, errno, _("closing standard input"));
    else
      die (EXIT_FAILURE, errno, "%s", quotef (infile));
  }

return EXIT_SUCCESS;
```

図 2　簡略化表記に変換したソースコード

葉から根に向かう各経路上において，最も根に近い簡略化候補を根とする部分木を簡略表記に変換

簡略化表記には次のように変換する．簡略化候補が兄弟方向に連続して並んでいる場合は，それらをひとまとめにし，そのコード片の最初の文と最後の文の各先頭の字句に置き換える．ただし，制御文については，先頭の予約語のみでは簡略化されている内容の推測が困難であるので，予約語と条件式に置き換える．単体の複合文が簡略化候補の場合は，波括弧のみを出力する．単体の式文が簡略化候補の場合は，簡略化しても行数を減らす効果はないので，灰色にするなど，視覚的に目立たない表現に変換する．実際に簡略化したソースコードを図 2 に示す．

4　評価・考察

本章では，ソースコードの理解支援を達成できるかを検証するために，以下の点について評価する．

1. 読むべきソースコードの量がどの程度削減されたか
2. 簡略化ソースコードの精度の高さはどの程度か

実験対象は，Coreutils-8.29 の src ディレクトリ以下の C ファイル 126 個である．src 以外はテストケースや標準ライブラリの補完など，目的が異なるので，対象から外した．提案手法を実現するツールは Perl で実装し，構文解析には TEBA[4] を用いた．

4.1　ソースコードの量に関する評価

行数が削減されることで，プログラマがコードを読む時間が減る．そこで，共通出現値の閾値を変化させながら，(1) 対象のソースコード群から簡略化コードを生成し，(2) 元のソースコードとの行数の差を求めた．

ソースコードの行数の差を図 3 に示す．ここでは，src ディレクトリから無作為に選んだ chroot.c, hostname.c, mv.c, nohup.c の結果を示している．図 3 から，全体的に

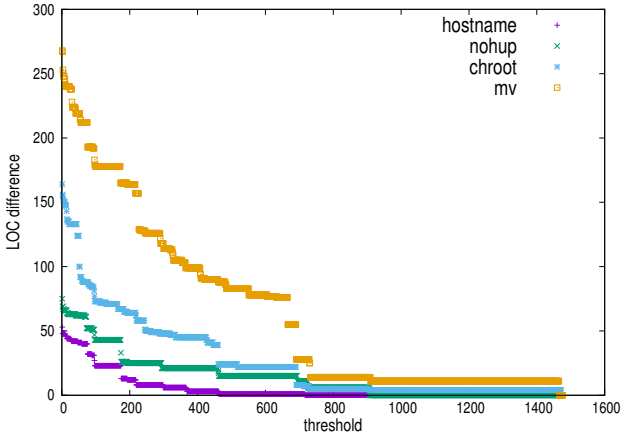

図 3　簡略化前後の行数の差

は閾値がおよそ 700 以下で削減効果が出ていることが読み取れる．閾値が 200 以下では特に変化が大きく，閾値が 1 から 200 の間で，少なくとも 40 行以上の差が生じた．

4.2　提案手法の精度の評価

行数が大きく削減されても，ソースコード上に残るべき文も消えるとプログラムの理解に支障が出る．そこで，適合率と再現率により精度を評価した．適合率と再現率はそれぞれ式 (3), (4) のように定義した．

$$Precision = \frac{TP}{TP + FP} \tag{3}$$

$$Recall = \frac{TP}{TP + FN} \tag{4}$$

ここで，TP は簡略化が期待される文のうち正しく簡略化された文の数，FP は簡略化が期待されていない文のうち簡略化された文の数，FN は簡略化が期待されるにも関わらず，簡略化されなかった文の数とする．

評価にあたっては，共通出現値の閾値を複数設定し，次のように行った．(1) 手作業で，正解となる簡略化すべきコード片に印を付け，(2) 対象のソースコード群から簡略化ソースコードを生成する．(3) 各文について印の有無および簡略化されたか否かを調べる．(4) (3) に基づき，TP, FP, FN を求め，適合率と再現率を算出する．

評価対象には，無作為に選んだ base64.c と echo.c を用いた．ここでは，base64.c の結果を図 4 に示す．閾値が小さいと，簡略化対象でない文も多く隠すので，適合率が低く，再現率は大きくなる．よって，適合率と再現率はトレードオフの関係にあるが，精度の目安となる交点ではともに 0.8 弱と高い値になっている．

4.3　精度と閾値に関する考察

簡略化をするにあたって閾値の決め方が問題となる．最適な閾値として，適合率と再現率の交点が考えられるが，実際に利用する際には，正解の情報がないので，適合率と再現率は求まらない．図 4 の結果から，閾値を小さい値の

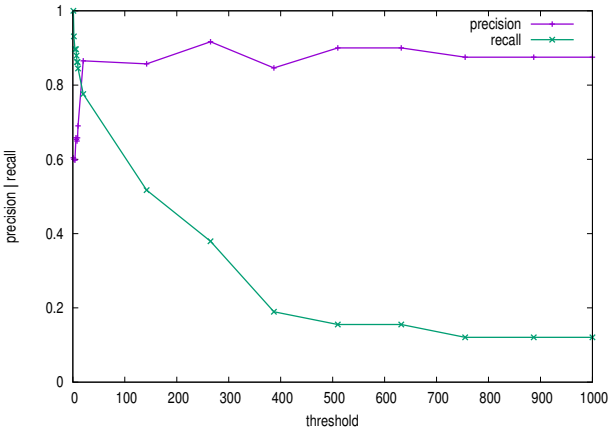

図 4　base64 の簡略化コードの適合率と再現率の変化

範囲で調整すれば，適切な簡略化コードが生成できると考えられる．全体の傾向として，一部のコード片のみが共通出現値が大きく，残りは小さくなるので，下限を中央値，上限を平均値にすることで，若干の絞り込みが可能である．

プログラミング知識に該当するコード片と共通性の高いコード片は必ずしも一致するとは限らない．ユーザからのフィードバックに基づいて簡略化コードを再構成できれば，簡略化の対象はプログラミング知識に近づき，精度が向上する．例えば，関数は処理の内容を 1 つにまとめたものであるので，簡略化して良い関数名を指定して共通出現値に反映させる方法がある．

5　関連研究

Fowkes ら [5] は，プログラムの概要の理解に不必要なコードを折り畳むことでソースコードの要約を生成する手法を提案している．折り畳むべきコードの選定にはトピックモデルを利用している．Sonia ら [6] は，ソースコードの字句から，テキスト検索によって重要な字句を抽出し，構文解析で得た構造の情報と合わせてソースコードの要約を生成する手法を提案している．

コードクローン検出では，処理を構成する文が同じでも，並びが異なると，類似したコード片として検出できない．本研究では文の並びが異なっても，柔軟に対応できる．向井ら [7] は，コードクローンの検出に *pq*-gram[8] を利用している．本研究では N-gram を採用したが，*pq*-gram の利用も今後検討が必要である．

プログラムスライシング [9] は，プログラム中の命令間に存在する依存関係を明らかにし，依存関係にない命令を簡略化する．本研究では，データの依存関係ではなく，共通性の高さに基づいて簡略化対象を抽出する．

加藤 [10] は，条件文の共通性に基づいて例外処理を抽出する手法を提案している．加藤は if 文内の文を対象に，字句列の N-gram を取得しているが，本研究では例外処理以外も対象としている．

6　おわりに

本論文では，ソースコードの理解支援のために，プロジェクト内で共通するプログラミング知識に基づいてソースコードを簡略化する方法を提案した．評価実験では，Coreutils-8.29 の src ディレクトリ以下を対象に，プロジェクト内で共通のプログラミング知識が簡略化されることを確認した．今後の課題は，簡略化ソースコードを補正するために，ユーザからのフィードバックに基づいて簡略化コードを再構成する方法の議論と，簡略化対象とそうでない文が混在する場合の簡略化方法の改善である．

参考文献

[1] “Github”, `https://github.com` (2019). GitHub, Inc.

[2] “グラフィカルビュー — ソースコード解析ツール understand — テクマトリックス株式会社”, `https://www.techmatrix.co.jp/product/understand/function/graphicalview.html` (2019). TechMatrix Corporation.

[3] “Coreutils - gnu core utilities”, `https://www.gnu.org/software/coreutils/` (2017). Free Software Foundation, Inc.

[4] 吉田, 蜂巣, 沢田, 張, 野呂：“属性付き字句系列に基づくソースコード書き換え支援環境”, 情報処理学会論文誌, pp. 1832–1849 (2012).

[5] J. Fowkes, P. Chanthirasegaran, R. Ranca, M. Allamanis, M. Lapata and C. Sutton: “Autofolding for source code summarization”, IEEE Transactions on Software Engineering, pp. 1095–1109 (2017).

[6] S. Haiduc, J. Aponte and A. Marcus: “Supporting program comprehension with source code summarization”, 2010 ACM/IEEE 32nd International Conference on Software Engineering, Vol. 2, pp. 223–226 (2010).

[7] 向井, 小林, 紫藤, 山本, 宮本, 松村, 嶺：“トークンの n-gram によるプログラム表現を用いたコードクローン検出手法”, 京都大学大学院総合生存学館，京都大学大学院情報学研究科，(株) 日立製作所基礎研究センタ (2018).

[8] B. Augsten, M. Bohlen and J. Gamper: “The pq-gram distance between ordered labeled trees”, ACM Transactions on Database Systems, pp. 4:1–4:36 (2010).

[9] 下村：“プログラムスライシング技術と応用”, 共立出版 (1995).

[10] 加藤：“例外処理のコーディング規約の理解のための共通性に基づく条件文選別手法の提案”, 南山大学大学院理工学研究科 (2018).