情報処理学会論文誌 Vol.56 No.3 821–834 (Mar. 2015)

キーワードのあいまい一致を導入したキーワードプログラミングシステム

坂本 悠輔[1,a)] 佐藤 晴彦[1] 小山 聡[1] 栗原 正仁[1]

受付日 2014年6月30日, 採録日 2014年12月3日

概要：キーワードプログラミングとは，ユーザ（プログラマ）が与える少数のキーワードに基づき，現在の文脈においてユーザが意図するコード片の候補を自動生成することにより，プログラミング言語の文法規則やAPIを正確に覚えていないユーザでも正しいコードを効率良く書けるよう支援するツールである．その提唱者であるLittleとMillerの研究では，出力コード片を構成する関数や変数の名前に出現する単語と完全に一致するキーワードを入力する必要があるため，その言語に詳しくないユーザにとって，認知的なあるいは物理的な負担となるものであった．本研究では，この負担を軽減するために，筆者らがこれまで提示した複数のコード片を出力する実装について述べるとともに，キーワードがあいまい（不正確）であってもできるだけ良い出力を得ることを目的として，単語間の類似度に基づくキーワードのあいまい一致手法を導入して，オープンソースプロジェクトを対象とした実験の結果に基づき，このような拡張手法が有効であることを議論する．

キーワード：ソフトウェア工学，プログラミング補助ツール，コードサーチ，コード生成

Keyword Programming Systems with Ambiguous Keyword Matching

Yusuke Sakamoto[1,a)] Haruhiko Sato[1] Satoshi Oyama[1] Masahito Kurihara[1]

Received: June 30, 2014, Accepted: December 3, 2014

Abstract: Keyword programming is a method for generating code snippets automatically based on the keywords provided by programmers. It aims to support programmers who are not familiar with precise syntax and APIs of programming languages when they want to write correct code more efficiently. The system of the previous study puts cognitive and physical burdens on novice programmers because they must input the very same keywords that appear in the names of functions and variables that compose output code fragments. In order to lighten this burden, we present a new system which outputs two or more code fragments and show that this extension is effective for obtaining as good outputs as possible even if input keywords are ambiguous or inaccurate.

Keywords: software engineering, programming assistance tool, code search, code generation

1. はじめに

現在のソフトウェア開発では，ライブラリやフレームワーク，参考書，ウェブサイトなどに存在する既存のコードを再利用してコードを記述することが一般的である．しかし，それらのライブラリやAPIの集合は巨大かつ複雑であるため，ユーザ（プログラマ）が記憶したり検索したりして活用することが非常に困難であり，この問題に対応するために様々なツールの研究・開発が行われている．その1つの例としては，ユーザが必要とするコードスニペット（snippets of code）を，大量に存在するコード集合から抽出するOhloh Code [1]のようなコードサーチエンジンやそれに関する研究がある [2], [3]．またもう1つ

[1] 北海道大学大学院情報科学研究科
Graduate School of Information Science and Technology, Hokkaido University, Sapporo, Hokkaido 060–0814, Japan
[a)] sayuu@complex.eng.hokudai.ac.jp

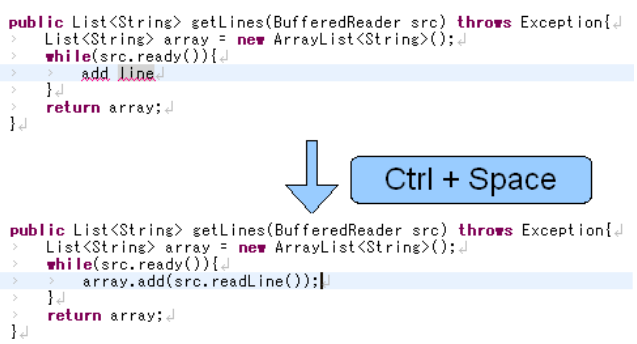

図 1 Eclipse のプラグインとして実装されたツールの動作を表した図

Fig. 1 An example of using keyword programming system implemented as an Eclipse plug-in.

の例として，Eclipse などの統合開発環境には，ユーザが現在編集中のソースコードの情報を利用して，現在位置で欲しいコードを補完するコードコンプリーション（code completion）という機能が備わっている．近年，このコードコンプリーションに関する研究は徐々に増えてきており，論文タイトルに “code completion” という単語を含むものだけでも [4], [5], [6], [7] などがある．Little と Miller が提唱し，本研究で改良と実装を行ったキーワードプログラミング [8] はそのような研究の延長上にある，さらに野心的な試みといえる．このキーワードプログラミングは，まるで Web 検索エンジンのように，ユーザが与える少数のキーワードから適切な完成型のコードを生成することにより，ユーザがプログラミング言語の文法規則や API 群の詳細部分を覚えている必要を軽減する技術である．**図 1** は Eclipse のプラグインとして実装されたキーワードプログラミングのツールの動作を表した図である．この図が示すように，キーワードプログラミングではユーザがいくつかのキーワードを入力し，コマンド（Ctrl+Space）を押すことでその現在の文脈（周辺のソースコードの状況）に適したコード片を挿入することができる．

本研究における，ユーザが入力する複数のキーワードは，現在の文脈においてユーザが意図する表現を検索するクエリであるかのようにとらえることができる．キーワードプログラミング以前の研究 [9], [10] においては，望ましいコード片をコードのデータベースから取得するため，そのコード片の返り値や入力値の型（int などの基本データ型や java.lang.String などのクラス）の名前をクエリの一部に入力しなければならなかったが，キーワードプログラミングでは，入力するクエリはプログラミング言語の型名に拘束されず自由になった．このために，ユーザは型名などのプログラミング言語の文法の詳細な知識に乏しくても，コーディングの支援を受けることが可能となった．また，広大な探索空間の中から，ユーザにストレスがかからない時間で解を探索するために探索空間を削減するなどの工夫も行っている．また，これまで筆者らは，WWW の検索エンジンが多数のウェブページを順位を付けて出力するように，キーワードプログラミングにおいても多数のコード片を順位を付けて出力する効率の良い実装を行っている [11]．

本論文では，キーワードプログラミングをさらに使いやすくするため，ユーザのキーワードの入力に着目し，その言語に不慣れなユーザが，あいまいな記憶によってキーワードの一部分だけを覚えていて短縮して入力したときや，一部を間違えて入力してしまったときにも対応できるようなツールの改良について述べる．具体的には，単語間の類似度に基づくキーワードのあいまい一致手法を導入して，オープンソースプロジェクトを対象とした実験の結果に基づき，このような拡張手法が有効であることを議論する．

本論文の構成は以下のとおりである．続く 2 章では，本研究の関連研究について述べる．3 章では，先行研究 [8] にて提案されたキーワードプログラミングのモデルと出力候補の評価方法について述べる．4 章では，本研究にて複数候補を提示するように拡張したキーワードプログラミングシステムについて述べる．5 章では，本研究で提案するキーワードのあいまい一致手法とその実装を用いた実験およびその考察を述べる．6 章では，本研究と関連研究との比較実験およびその考察を述べる．最後に 7 章でまとめと今後の課題を述べる．

2. 関連研究

本章では，本研究の関連研究について解説する．統合開発環境 Eclipse [12] のコード補完機能は，関数名や変数名を正しく途中まで入力すると，その名前の残りの文字列を補うものである．ただし，本研究のようにあいまい一致の機能はない．また，複数の関数名や変数名を組み合わせたコード断片といえるようなものを出力するわけではない．Abbreviation Completion [5] は，望ましいコード片の部分列を入力すると，省略された部分列を補って，1 行のコード片を出力する．たとえば，出力 public static void main(String[] args) を得たいときに，その出力の部分列となる pb st v m(st[] ag) を入力する．この研究は認知的負担を軽減することを目的とした本研究とは異なり，習熟者がキーストロークの削減を目的として，すでに望むコード片を明確に把握しているときに使用するツールである．これは，入力に含まれる文字列をあいまいキーワードと見なせば本研究と似ているが，入力におけるキーワードの出現順序が正しいコード片における出現順序と完全に一致している必要がある点と，たとえあいまいであっても全キーワードを少なくとも 1 文字以上入力する必要がある点で異なっている．

Prospector [9] と XSnippet [10] はともに 2 つの型を入力すると，スニペット（再利用を目的として事前にデータベースに登録されたコード片）を出力するツールである．

表 1 関連研究との比較
Table 1 Comparison with related studies.

研究またはツール	入力	出力	主な知識源
本研究	あいまいキーワード（順不同）	1 行のコード片	API 定義
Eclipse のコード補完機能 [12]	関数名または変数名の接頭辞	1 つの関数名または変数名	API 定義
Abbreviation Completion [5]	省略されたキーワード（順序が有効）	1 行のコード片	コードコーパス
Prospector [9]	2 つの型名	スニペット	API 定義とコードコーパス
XSnippet [10]	2 つの型名	スニペット	コードコーパス
GraPacc [13], [14]	定義済みの変数	スニペット	コードコーパス
Automatic Method Completion [15]	途中まで定義されたメソッド	スニペット	コードコーパス
Graphite [16]（色選択の場合）	カラーパレットから色を選択	1 行のコード片	カラーパレットの API

Prospector は，2 つの型を入力すると，一方の型からもう一方の型へと変換するような複数のスニペットを出力する．XSnippet は，2 つの型を入力すると，文脈（ローカル変数の型など）に基づき，ユーザが意図していると考えられるスニペットを，その長さや使用頻度に関するヒューリスティックスを利用して選択し出力する．Prospector はある型から別の型に変換したいときに有効なツールであり，XSnippet は文脈に適したスニペットが欲しい場合に便利なツールである．この 2 つのツールと本研究との違いは，型の入力を必要とすることと，出力がスニペット（すなわち事前に登録が必要）であること，知識源にコードコーパスを使用することなどがあげられる．

GraPacc [13], [14] は，ツールが起動された場所よりも前にすでにユーザが定義している複数のローカル変数を入力として，スニペットを出力するツールである．Automatic Method Completion [15] は，ユーザが途中まで記述したメソッド定義を入力とし，そのメソッド定義と類似した定義を持つメソッドをコードコーパスの中から検索して，途中まで記述したメソッド定義を完成させるスニペットを提示するものである．GraPacc と Automatic Method Completion はともに，Prospector や XSnippet と同様にスニペットを出力するツールであるが，明示的な入力の必要がない点が特徴である．

Graphite [16] は，1 つのインタフェースでなるべく多くの用途をカバーしようとしてきた本研究を含む既存の研究とは異なり，正規表現の入力や色の指定など特定の用途に特化した複数の入力インタフェースを提供するツールの研究である．たとえばプログラムで色を指定する際，紫色なら (128.0.128) などと，数値で指定することが一般的であるが，それを直感的に分かりやすいようにカラーパレットをポップアップで提示し，色を選択させるといったインタフェースを提供している．

本研究の位置付けを明確にするために，本研究および関連研究のそれぞれについて，(1) ツールへの入力，(2) ツールからの出力，および (3) 出力候補の絞り込みや生成を行うための主な知識源，の 3 つの観点で整理し，**表 1** に示す．このように，本研究は，複数のあいまいキーワードを順不同で入力すると，API 定義を知識源として，（事前登録されたスニペットではなく）コード片を生成し出力する点で，他のいずれの研究とも異なっている．

3. キーワードプログラミング

本章では，先行研究 [8] で提案されたキーワードプログラミングの概要を，理論的に整理して述べる．ただし，アルゴリズムについては，[8] のものを一部拡張して，筆者らが [11] で示した多数のコード片を順位を付けて出力するものを紹介する．

3.1 モデル

Java 言語の基本データ型（byte, short, int, long, float, double, char, boolean）とクラスを**型**と定義し，Java の標準的な配布パッケージまたはアプリケーションプログラマ（ユーザ）によって定義されたすべてのクラスの中から，型の探索範囲を型の集合 T として事前に定義しておく．次に，T に属するクラスで定義されているメソッドや変数の名前を**ラベル**と定義し，すべてのラベルの集合を L とする．ただし，currentTimeMillis のように，大文字で区切った複数の単語からなる 1 つのラベルは（current, time, millis）のように分割し，単語のリストとして表現する．メソッド（コンストラクタを含む）を**関数**，メンバ変数およびローカル変数を**変数**と定義し，$T \times L \times T \times \cdots \times T$ に属するタプルで表現し，その集合を F とする．先頭の T は関数の返り値の型または変数の型であり，L がその関数または変数のラベル，それ以降の T は関数の引数の型である．たとえば，関数 String toString（int i, int radix）は，(java.lang.String,（to, string), int, int）と表現される．型 t のサブタイプの集合（t 自体を含む）を $sub(t)$，関数 f の返り値の型および変数 f の型を $ret(f)$ で表す．さらに，関数 f の第 i 引数の型を $params(f)_i$，関数 f の引数の型の多重集合（要素の重複を許す集合）を

$$params(f) = \{params(f)_i \mid 1 \leq i \leq n\} \quad (1)$$

で表す．ただし，n は f がとる引数の数である．

最終的にツールが出力するコード片を表現する**関数木**を

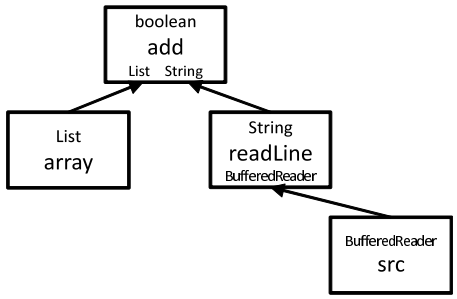

図 2 array.add(src.readLine()) を関数木で表現した図
Fig. 2 A function tree illustrating "array.add(src.readLine())".

定義する．関数木は，関数および変数を要素（ノード）とするノードの集合，および次の 2 つの条件を満たすように親ノードと子ノードを接続する辺の集合から構成される木である．

(1) 引数の数が 0 である関数および変数は，終端ノードである．
(2) 引数の数が $n(n > 0)$ である関数は，n 個の子ノードを持つ非終端ノードであり，このノードを親ノード g で表し，第 i 番目 $(1 \leq i \leq n)$ の子ノードを f_i で表すと，$ret(f_i) \in sub(params(g)_i)$ である．

また，レシーバ r のメソッド f を k 個の引数 $a_1, \ldots, a_k$ で呼び出すメソッド呼び出し $r.f(a_1, \ldots, a_k)$ は，関数呼び出し $f(r, a_1, \ldots, a_k)$ と見なされ，f を根ノード，$r, a_1, \ldots, a_k$ のそれぞれを再帰的に関数木として表現したものをそれぞれ部分木とする関数木で表される．**図 2** は array.add(src.readLine()) を関数木で表現した図である．この図のそれぞれのノードについて，その 1 行目が関数の返り値の型または変数の型，2 行目が関数名または変数名，3 行目が関数の引数の型である．

3.2 キーワードと文脈に基づく関数木の評価

ユーザが入力するキーワードの列を**クエリ**（query）と定義する．以下では，ソースコード上の任意の位置でユーザがクエリを入力し，システムがその位置に挿入するコード片の候補を生成したとき，そのコード片が表す関数木の好ましさを評価する方法を述べる．ただし，説明を簡潔に行うために，入力されるキーワードの列を「集合」，すなわち重複を排したものとみなす．重複を含む場合の詳細は [8] を参照されたい．

まず，実数値 w_1，w_2，w_3，w_4 を**重み**と呼び，事前に固定値を与えておく．[8] では，$w_1 = -0.05$，$w_2 = -0.01$，$w_3 = 0.001$，$w_4 = 1.0$ としており，以下でもそれを仮定する．

次に，以下に述べる 4 ステップの基本手順に従って関数木の各ノード（関数および変数）に評価値を与える．評価値は大きいほど望ましい．

(1) すべてのノードの評価値を $w_1(< 0)$ に初期設定する．w_1 はノード数の少ない関数木が好ましいとする考えに基づき与えるノードへのペナルティである．
(2) ノードのラベルに含まれる単語のうち，クエリ中のキーワードでないもの 1 つごとに，そのノードの評価値に $w_2(< 0)$ を加算する．w_2 はラベルに含まれるキーワード以外の単語へのペナルティである．
(3) ノードが**現文脈**に属する要素（ローカル変数，メンバ変数またはメンバメソッド）である場合には，評価値に $w_3(> 0)$ を加算する．現文脈とは，当該ソースコード内で宣言またはインポートされている関数，変数，および型のうち，その名前の有効範囲（scope）が当該コード片の挿入位置を含むものの名前の集合である．w_3 は現文脈に属するラベルに与えるゲインである．

ここまでの手順を形式的に扱うために，各ノードに対し，次の 3 つの**特徴量**を定義する．

x_1: ノードの数．すなわち，値はつねに 1．
x_2: ノードのラベルには含まれるが，クエリ中のキーワードではない単語の数．
x_3: ノードが現文脈に属する要素であれば 1，そうでなければ 0．

このとき，ノードの基礎評価値は次の式で与えられる．

$$e_0 = w_1 x_1 + w_2 x_2 + w_3 x_3 \tag{2}$$

最終評価値を得るためには，次のステップを要する．

(4) クエリ中のキーワードがノードのラベルに単語として含まれるごとに，評価値に $w_4(> 0)$ を加算する．ただし，関数木全体として，キーワードごとに加算値の上限を 1 に制限する．w_4 は使用されたキーワードに与えるゲインである．

4. 複数候補を提示するキーワードプログラミング

本章では，キーワードプログラミング [8] の理解および分析を容易にするために，本論文独自の表現（$\oplus$ 演算子，特徴ベクトル）を導入して，文献 [8] では明確に分析されていなかったキーワードプログラミングの問題点を明確に述べ，それを解決するために，複数候補を提示するようにキーワードプログラミングシステムを拡張することが本質的に必要であることを議論する．

4.1 特徴ベクトルと評価ベクトル

3.2 節のステップ (4) を形式的に扱うために，クエリ中のキーワードが n 個であるとして，i 番目のキーワード k_i に関する特徴量 $y_i(1 \leq i \leq n)$ を導入する．

y_i: 当該ノードのラベルの中に含まれる k_i の出現回数．

このとき，キーワード k_i を含むことに対するこのノードの評価値は次の式で与えられる．

$$e_i = 0 \oplus w_4 y_i \quad (1 \leq i \leq n) \tag{3}$$

ただし，演算子 $\oplus$ は，文献 [8] での議論を形式的に明確にするために本論文で導入したものであり，上限を 1 とする加算を表す．

$$x \oplus y = \min(x + y, 1) \tag{4}$$

標準的な設定 $w_4 = 1$ のもとでは，$e_i = min(y_i, 1)$，すなわち，k_i が 1 回でも出現していれば $e_i = 1$，そうでなければ $e_i = 0$ とする．これは，システムの有用性の観点から，キーワードの出現は好ましいが，2 回以上出現していても，ユーザの欲しい関数木である可能性が高まるわけではないという経験則を反映している．

以上の特徴量を並べた $n + 3$ 個の成分を持つベクトル $(x_1, x_2, x_3, y_1, \ldots, y_n)$ を**特徴ベクトル**と呼ぶこととする．また，前節で述べた基礎評価値を含むノードの評価値を並べた $n + 1$ 個の成分を持つベクトル $(e_0, e_1, \ldots, e_n)$ を**評価ベクトル**と呼ぶ（文献 [8] では説明ベクトルと呼んでいる）．ノード f の特徴ベクトルと評価ベクトルをそれぞれ $vec(f)$ および $evec(f)$ で表し，式 (2)，(3) に基づいて特徴ベクトルを評価ベクトルに変換する関数を $v2e$ で表す．

$$\begin{aligned} &v2e((x_1, x_2, x_3, y_1, \ldots, y_n)) \\ &= (w_1x_1 + w_2x_2 + w_3x_3, 0 \oplus w_4y_1, \ldots, 0 \oplus w_4y_n) \end{aligned} \tag{5}$$

すなわち，

$$evec(f) = v2e(vec(f)). \tag{6}$$

$evec(f)$ の全成分の和（実数値）を f の**評価値**または $evec(f)$ の評価値といい，$eval(f)$ で表す．たとえば，クエリが，

is queue empty

で，評価値を計算したいノードが，現文脈に属する関数

（boolean,（**is, empty**）, List）

であるとき，特徴ベクトルは $(1, 0, 1, 1, 0, 1)$，評価ベクトルは $(-0.049, 1, 0, 1)$，評価値は 1.951 となる．

ここまでノードに関して定義した概念を関数木にも拡張する．すなわち，木 f_t を構成するすべてのノードの特徴ベクトルの和をその木の特徴ベクトル $vec(f_t)$，木を構成するすべてのノードの評価ベクトルの和をその木の評価ベクトル $evec(f_t)$，その全成分の和をその木の評価値 $eval(f_t)$ と定義する．ただし，2 つの評価ベクトルの和を

$$\begin{aligned} &(e_0, e_1, \ldots, e_n) \oplus (e'_0, e'_1, \ldots, e'_n) \\ &\quad = (e_0 + e'_0, e_1 \oplus e'_1, \ldots, e_n \oplus e'_n) \end{aligned} \tag{7}$$

と定義する．（第 1 成分だけは通常の加算である．）

このとき任意の特徴ベクトル v, v' に対して次の恒等式が成立する．（証明は容易なので省略）

$$v2e(v + v') = v2e(v) \oplus v2e(v') \tag{8}$$

この関係式は特徴ベクトルの概念を導入していない [8] では明示されていないが，次節で述べる動的計画法に基づくアルゴリズムの基礎として重要である．なお，次節以降で，評価ベクトルどうしあるいは評価ベクトルと実数値の大小比較を行う数式が出てくるが，評価ベクトルはその評価値を表すと解釈して，実数値どうしの大小比較に帰着させる．

4.2 関数木生成アルゴリズム

キーワードプログラミングシステムの核となるアルゴリズムは，評価値の高い関数木を生成するものであり，その詳細は文献 [8] に示されている．しかし，そこではアルゴリズムの背景にある問題の数学的な構造について明示的に議論していないために，その理解や分析がやや困難な面がある．本論文では，そのアルゴリズムをそのまま紹介することを避けるかわりに，問題の数学的な構造について形式的に提示し，その後，文献 [8] のアルゴリズムを複数の関数木を生成するように修正した本研究独自のアルゴリズムを説明する．

3.1 節で定義したように，探索範囲を表す型の集合，および型情報を含むタプルで表される関数（または変数）の集合をそれぞれ T および F とする．F を用いて構成できる高さ h 以下の関数木の全体を表す集合を $FT(h, F)$ または F を明示せずに $FT(h)$ と記す．キーワードプログラミングシステムが解くべき基本問題は，集合 $FT(h)$ の中で最も評価値の高い関数木（最適解）

$$f_t{}^* = \arg\max_{f_t \in FT(h)} eval(f_t) \tag{9}$$

を求めることである．

実際，文献 [8] では，動的計画法の考え方に基づき，最適解を 1 つだけ求めるアルゴリズムを示している．動的計画法では，木を完全に構成してから評価値を計算するのではなく，木を再帰的に構成する途中で生成される部分木を評価し，最適でない部分木を捨てる（探索の枝刈り）．文献 [8] ではこのあたりの根拠の分析が明示的ではないため，以下で補足する．

まず重要な性質は，4.1 節で述べたように，任意の特徴ベクトル v, v' に対して，

$$v2e(v + v') = v2e(v) \oplus v2e(v') \tag{10}$$

が成り立つことである．これにより，木を構成する全ノードの特徴ベクトルの総和を直接求めずに，部分木ごとに評価値を求め，非最適なものを捨てつつ，最適なものの評価値を $\oplus$ 演算で累積して上位の木の評価値計算に効率良く活かすことができる．特徴ベクトルの概念を導入していない文献 [8] では，この議論は暗黙的である．その意味で，本論文は文献 [8] のアルゴリズムの正当性に関して，形式的な分析をもとに見直している．

もう 1 つの重要な性質（ただし，実際には成り立たない

性質）は，動的計画法で前提となる「最適性原理」である．この原理は一般に「全体が最適ならば，部分も最適である」ことを意味する．したがって，「全体」を構成する過程で生成される「部分」は，最適でなければ，捨てても全体の最適解の探索に影響はない．よって，最適な関数木の探索の過程で，最適でない部分木は捨てても良い根拠となる．

しかし，注意すべきは，本論文の問題設定においては，最適性原理は成り立たないことである．それは評価ベクトルに関わる演算の中で，$\oplus$ が用いられているため，任意の評価ベクトル e，e'，e'' について，一般に

$$e < e' \text{ならば } e \oplus e'' < e' \oplus e'' \quad \text{（単調性）} \tag{11}$$

が成り立たないためである．たとえば，キーワードを 2 個として，ある 2 つの部分木に対する各評価ベクトルを

$$e = (-0.049, 1.0, 0) < (-0.1, 1.0, 1.0) = e', \tag{12}$$

上位の 1 引数関数ノードの評価ベクトルを

$$e'' = (-0.049, 0, 1.0) \tag{13}$$

とすると，

$$\begin{aligned} e \oplus e'' &= (-0.098, 1.0, 1.0) \\ &> (-0.149, 1.0, 1.0) = e' \oplus e'' \end{aligned} \tag{14}$$

であり，不等式の向きが逆転する．したがって，動的計画法は，最適解の一部となる可能性を持っている部分木（評価ベクトル e に対応するもの）を捨ててしまうことになる．

このような背景にもかかわらず，文献 [8] では動的計画法に基づき，「最適解」を 1 つだけ求めるアルゴリズムを構成している．しかし，いま述べたように，その解が最適である保証はない．さらに問題なのは，仮にそれが最適解であったとしても，1 つだけ表示されたその解がユーザの欲しい関数木とは限らないことである．別な最適解や非最適解の方が欲しいものかもしれないのである．

しかし，ユーザの欲しい関数木を出力するために，探索の枝刈りをいっさい行わずにすべての関数木を生成するのは現実的でない．そこで本論文では，動的計画法の適用を緩和し，木を構成していく過程で，最適な部分木だけを残すのでなく，評価値が高い順に一定数の部分木を残し，最適とは限らない比較的多数の関数木を構成する．

この設計方針は，文献 [8] から概念的に飛躍したものとなるが，それを実現するアルゴリズムは，文献 [8] のものをわずかに修正するだけで済む．それらをアルゴリズム 1〜3 として示す．

アルゴリズム 1 は，T に属するすべての型 t について，返り値の型が t で高さが i $(1 \le i \le h)$ の関数木のうち，評価値が上位 r 番目までのものを $\mathrm{goodRoots}(t, i)$ に格納する．具体的には，木の根ノードの候補となる関数 f ごとに，アルゴリズム 2 を呼び出して最適解（と思われる木）を求め，

Algorithm 1 高さと型ごとに複数の関数木を生成する

```
procedure FTrees(h, T, F)
for each 1 ≤ i ≤ h do
  for each t ∈ T do
    goodRoots(t, i) ← ∅
    for each f ∈ F such that ret(f) ∈ sub(t) do
      tree ← GetBestForFunc(f, i − 1)
      e ← evec(tree)
      if e > −∞ then
        goodRoots(t, i) ← goodRoots(t, i) ∪ tree
      end if
    end for
    /*評価値上位 r 番目までの木を残す*/
    goodRoots(t, i) ← GetBestN(goodRoots(t, i), r)
  end for
end for
```

Algorithm 2 根の関数と最大高さを固定した最適解

```
procedure GetBestForFunc(f, h_max)
/*根ノード f のみの関数木の初期値を生成する*/
tree_best ← CreateFunctionTree(f)
e_best ← evec(f)
for each ptype ∈ params(f) do
  /*tree_param は f の固定されたパラメーターにおいて最良の関
  数木を保持しておくための変数*/
  tree_param ← null
  e_param ← (−∞, 0, 0, ..., 0)
  for each 1 ≤ i ≤ h_max do
    for each ptree ∈ goodRoots(ptype, i) do
      if e_best ⊕ eval(ptree) > e_param then
        tree_param ← ptree
        e_param ← e_best ⊕ eval(ptree)
      end if
    end for
  end for
  /*最良の関数木を tree_best の子ノードに追加*/
  tree_best ← AddChild(tree_best, tree_param)
  e_best ← e_param
end for
return (tree_best)
```

異なる f に対して最大 r 個の木を保持する．文献 [8] のアルゴリズムは，ここで $r = 1$ としたものに相当する（f ごとに非最適解も保持する設計もあり得るが，探索空間が膨大になりすぎるか，または応答時間がインタラクティブシステムとして遅すぎるので却下した）．なお，集合 S と単元集合 a の和集合を $S \cup a$ と略記している．

アルゴリズム 2 は，関数（または変数）f と高さ h_{max} が与えられたとき，f を根ノードとする高さ $1 + h_{max}$ の関数木のうち評価値が最大のものを求める．そのような木がないときは，評価値が $-\infty$ であるようなノードを返す．文献 [8] のアルゴリズムでは，返り値が関数木ではなく，その評価ベクトルだけを返すものであった．

アルゴリズム 3 は，型 t と高さ h が与えられたとき，$\mathrm{goodRoots}(t, i)$ $(1 \le i \le h)$ が保持しているすべての関数木を評価値の高い順にソートして出力する．文献 [8] では，

Algorithm 3 関数木の抽出

```
procedure ExtractTreeForType(t, h)
trees ← ∅
for each 1 ≤ i ≤ h do
  trees ← trees ∪ goodRoots(t, i)
end for
return (Sort(trees))
```

アルゴリズム 2 の返り値が評価ベクトルだけであったために，アルゴリズム 3 において，アルゴリズム 2 で探索済みの関数木を再度探索するという非効率なアルゴリズムとなっていた．

参考までに，後述するオープンソースプロジェクトを対象とし，木の高さ $h = 3$ に対する応答時間の目標を平均 0.5 秒程度以内とし，goodRoots に保持する木の数 r をどのくらい大きくできるかを予備実験した結果，CPU 性能が高くないノート PC（Core i5 1.6 GHz）では $r = 20$ 程度，性能が高いデスクトップ PC（Core i7 3.4 GHz）では $r = 150$ 程度であった．

5. キーワードあいまい一致

本節では，入力クエリに含まれるキーワードをあいまい（不正確）に入力しても，ツールが動作するように改良することで，ユーザのキーワード入力の認知的な負担（キーワードを正確に記憶する必要性）および物理的な負担（長いキーワードを入力したり入力し直したりする手間）を軽くすることを目的として，「キーワードあいまい一致」の仕組みを導入する．

先行研究 [8] のアルゴリズムは，入力されるキーワードはラベルに含まれる単語と完全に一致する文字列でなければ，評価値を与えない．たとえば，Message という単語をラベルに含む関数を得たいときに，キーワードとして msg と入力しても特徴 4 の特徴量は 0 であった．これがユーザの認知的あるいは物理的な負担となっている．そこで，この負担を軽減するため，単語と完全には一致しないキーワードであっても，特徴 4 の特徴量の値を両者の類似の度合いに応じて与えることにする．本研究では，文字列間の類似度によく使用されている**最長共通部分列**（Longest Common Subsequence）[17], [18] および，スペル訂正の分野でよく使われている**レーベンシュタイン距離**（Levenshtein Distance）[18], [19], [20], [21] を用いた類似度を定義し，実験に基づき，どちらの定義がより効果的かを議論する．

5.1 最長共通部分列

最長共通部分列を用いた類似度（**LCS1**, **LCS2**, **LCS3**, **LCS4**）を定義する．2 つの文字列の最長共通部分列（LCS）とは，連続している必要はない部分列で，両者に共通しているもののうち，最長のものであり，一般に複数存在する．たとえば ABRC と AKCB の最長共通部分列は AB と AC であり，その長さは 2 である．2 つの文字列の長さを m, n とすると，動的計画法による自明な方法により，LCS を $O(mn)$ で求めることができる（「メモ」用に m 行 n 列の表を作り，その i 行 j 列の要素に，各文字列のそれぞれ第 i 文字以降，第 j 文字以降の尾部どうしの LCS を，局所的な最大化をしながら再帰的に格納していく）．英単語を短縮する際には慣習的に母音を削除することが多いことを考えると，「部分列は連続している必要はない」という LCS の特徴が効果的に働いて，2 つの単語の類似度を定義する基礎として利用できる．そこで，ラベルに含まれる単語の長さを n，その単語とキーワードの LCS の長さを LCS とし，類似度 $LCS1$ を，

$$LCS1 = \frac{LCS}{n} \tag{15}$$

と定義する．この $LCS1$ では入力クエリに含まれる単語に余計な文字が誤って挿入され，完全一致する単語よりも長さが大きくなってしまっても類似度が変化せずペナルティーが存在しないため，入力クエリに含まれる単語 1 つの長さ m を考慮した類似度 $LCS2$ を，

$$LCS2 = \frac{2LCS}{m+n} \tag{16}$$

と定義する．また，非線形な式のケースも考察の対象とするため，LCS3 は，LCS1 の 2 乗，LCS4 は，LCS2 の 2 乗と定義する．LCS1〜LCS4 の最小値は 0，最大値は 1 となる．

5.2 レーベンシュタイン距離

次にレーベンシュタイン距離（LD）を用いた類似度を定義する．LD は，2 つの文字列の異なり具合を示す数値であり，両者を一致させるのに必要な編集コストに基づくため，「編集距離」とも呼ばれることもある．この数値は，文字の「削除」,「置換」,「挿入」によって，一方の文字列を他方の文字列と同一にする手順の最少回数として与えられる．たとえば 2 つの文字列 cat と cute では，cat の a を u に置換し，さらに e を最後尾に挿入して cute となるので LD は 2 となる．2 つの文字列の長さを m，n とすると，動的計画法により，LD を $O(mn)$ で求めることができる．この定義では，削除，置換，挿入のコストはいずれも 1 であるが，別の正の値を設定することもできる．その設定は，応用の目的やシステムの設計指針に適合させる必要があり，本研究では次のような考え方で設定する．

まずユーザは，正しいキーワードをそのまま入力するのが基本であるが，キーストロークを削減するために，（たとえば，message を msg のように）一部の入力を省略することを望むことがあり，システムはそれを低コストで受け入れることとする．また，ユーザが正しいスペルをよく覚えていないときは，含まれている自信のある文字だけを（たとえば，collector を colectr と）入力してもらうよう，シ

ステムの使い方のノウハウとしてユーザに周知することとし，そのような状況にも低コストで対処できるようにする．次に，文字の一部を置き換えるような（たとえば，hello を hallo にする）誤りは，できるだけ避けてほしいが，削除よりは大きいコストで許容する．最後に，文字を挿入するような（たとえば，hello を hellow にする）誤りは，あまり許容しないようコストを大きくする．以上のことから，

$$\text{削除コスト} \leq \text{置換コスト} \leq \text{挿入コスト} \tag{17}$$

の制約のもとで，値を調整するものとする．本論文では，この調整を精密に行うことを目的としていないので，ある整数 k について，

$$\text{削除コスト} = 1 \tag{18}$$

$$\text{置換コスト} = 1 + k \tag{19}$$

$$\text{挿入コスト} = 1 + 2k \tag{20}$$

という暫定的な設定で，k を指数的に 0，1，2，4，8 に変化させた 5 通りの実験を行うこととする．

また，LD は「距離」なので，単語とキーワードが完全に一致していれば 0 で，不一致の度合いに応じて大きな正の値となる．これを逆に，完全一致のときに 1 で，不一致の度合いに応じて 0 に近くなる量に変換し，それを類似度とする．この変換は，システムの実時間での応答性を保証するため，できるだけ簡単な数式によって行いたい．そのため，本研究では次の式で表される類似度を導入する．

$$LD(k, C) = \frac{1}{1 + \frac{LD(k)}{C}} \tag{21}$$

ここで $LD(k)$ は k の値に基づき式 (17)〜(19) により編集コストを定めたときの LD を表す．また，パラメータ C は正の実数であり，類似度 $= 0.5$ となるときの $LD(k)$ の値である．本論文では，$LD(k) = 1$ のときの類似度 $C/(C+1)$ が 0.5〜0.9 の範囲でおよそ 0.1 刻みになるよう，$C = 1,\ 1.5,\ 2.33,\ 4,\ 9$ の 5 通りに設定した実験結果を示す．

5.3 単語間の対応関係の計算

ここまでで定義してきた類似度を用いて，2.2 節で述べた，関数の各ノードに評価値を与える 4 ステップのステップ (4) と (2) を次のステップ (4')，(2') で置き換えることによって，各ノードの評価値を計算する．

(4') （クエリ中の）n 個のキーワードの集合 K と（ノードのラベルに含まれる）m 個の単語の集合 L の間のすべての要素の組合せ（nm 個）について類似度を計算し，それらから K と L の間に 1 対 1 対応の写像（単射）を構成するように $min(n, m)$ 個の対応を選ぶ．ここで選ぶ対応は，類似度の高いものからグリーディ（貪欲）に選び，それらの類似度の総和 S をできるだけ最大化するものとする．評価値には $w_4 S$ を加算する．ただし，関数木全体として，キーワードごとに加算値の上限を 1 に制限する．

(2') ステップ (4') において K と L の間に 1 対 1 対応の写像（単射）した際に選択されなかった L に含まれる単語 1 つにつき，評価値に w_2 を加算する．

5.4 キーワード分割

キーワード分割とは，関数（メソッドおよびコンストラクタ）と変数（フィールドおよびローカル変数）を指定の区切り文字（コンマ，スラッシュ，コロンなど）で分けて入力することにより，出力精度を高めるための方法である．たとえば区切り文字をスラッシュとし，add panel image label の 4 つのキーワードのうち，add を関数，それ以外のキーワードを変数として入力したいとき，それらをスラッシュの前後に分け「add / panel image label」のように，キーワード入力を行う．

キーワード分割におけるノードの評価値の計算は，5.3 節のステップ (4') において，K と L の各要素の組合せについて類似度を計算する際に，すべての組合せについて計算するのではなく，2 つの要素の種類（関数または変数）が一致する場合のみ，その類似度を計算することによって行う．

5.5 実験

キーワードあいまい一致とキーワード分割を用いる場合の精度を実験によって評価するため，コード片 P とその文脈 C を抽出し，後に述べる方法で P から入力クエリ Q を生成する．(P, C, Q) の組をタスクと呼ぶ．多数のタスクの各々について，C と Q を本研究で開発したキーワードプログラミングシステムに与え，システムが順位を付けて出力した複数のコード片のうち，ユーザがストレスなく目視で確認できる上位 U 件内に P が含まれていれば正解とし，全タスクに対する正解出現率を測定する．オープンソースプロジェクトとしては，文献 [8] で使用されていた 4 つのプロジェクト（Apache Commons Codec，CAROL，jMemorize，CMU Sphinx）と，単体テストのためのフレームワーク JUnit の合計 5 つのプロジェクトを用い，19,844 問のタスクを作成した．重み $w_1, \ldots, w_4$ は，3.2 節で述べた値を用いた．また，アルゴリズム 1 の r の値は 100 とした．

クエリ Q は，正解のプログラム片 P から，次に示す 5 通りで生成した．

改変なし P に含まれる関数および変数のラベルに含まれる単語すべてを集めたものを Q とする．

母音削除 英単語を短縮する際の慣習に習い，「改変なし」で得た Q に含まれるキーワードのそれぞれについて，先頭文字以外の母音をすべて削除したものを Q とする．

先頭 3 文字 「改変なし」で得た Q に含まれるキーワード

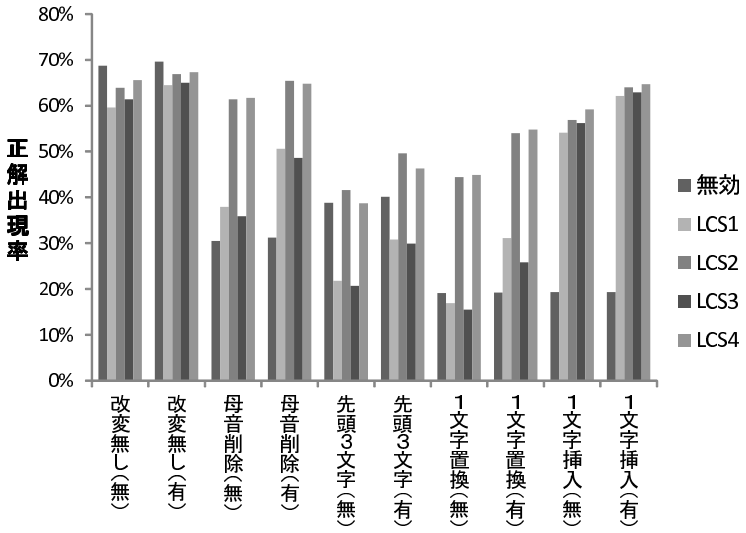

図 3 あいまい一致の機能が無効のときと，$LCS1$〜$LCS4$ を類似度としたあいまい一致の機能が有効なときの正解出現率を表したグラフ．横軸は，5 通りのクエリ生成方法のそれぞれについてキーワード分割が無効（無）および有効（有）の場合を区別している

Fig. 3 Accuracy rate in the case that ambiguous keyword matching is disabled, and in the cases that ambiguous keyword matching is enabled with one of the similarities from $LCS1$ to $LCS4$. The horizontal axis corresponds to combinations of query generation methods and keyword partition methods.

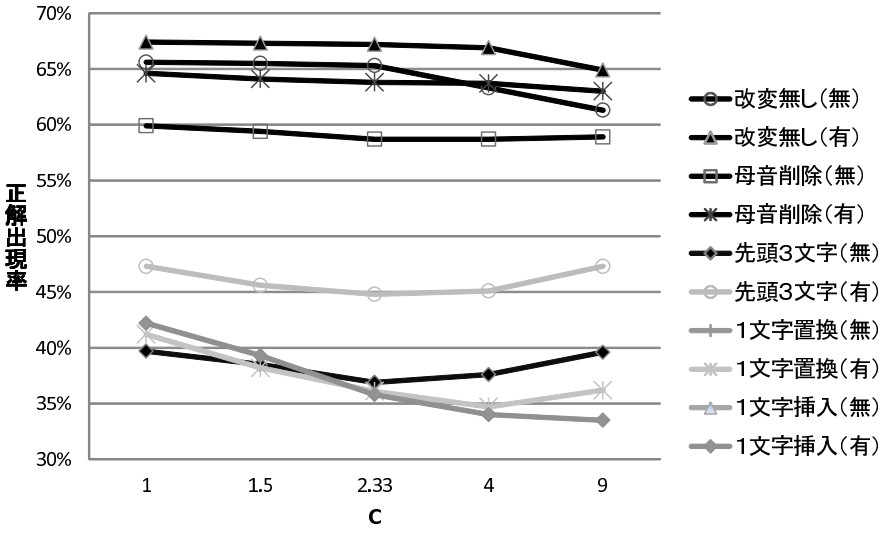

図 4 あいまい入力の類似度にレーベンシュタイン距離を用いた場合について，整数 k の値を 1 に固定したときの正解出現率を表したグラフ

Fig. 4 Accuracy rate in the cases of similarity based on Levenshtein Distance with $k = 1$.

のうち，長さが 4 以上のものについては，先頭の 3 文字を残してすべて削除したものを Q とする．

1 文字置換 「改変なし」で得た Q に含まれるキーワードのそれぞれについて，ランダムに選択した 1 文字をそれと異なるランダムな文字に置換したものを Q とする．

1 文字挿入 「改変なし」で得られた Q に含まれるキーワードのそれぞれについて，ランダムな位置にランダムな文字を 1 文字挿入したものを Q とする．

5.6 考察

実験結果を**図 3**，**図 4**，**図 5** に掲載した．図 3 は，類似度に $LCS1$〜$LCS4$ を使用したときの，上位 $U = 10$ 件までの出力に対する正解出現率を示したグラフである．この図から次のことが読み取れる（以下では，正解出現率を精度といい換える）．

- 「改変なし」のタスク群の場合，（導入しても意味のない）あいまい一致を導入しても，精度はすべての類似度の場合で 10%以内，特に $LCS2$ と $LCS4$ の場合は

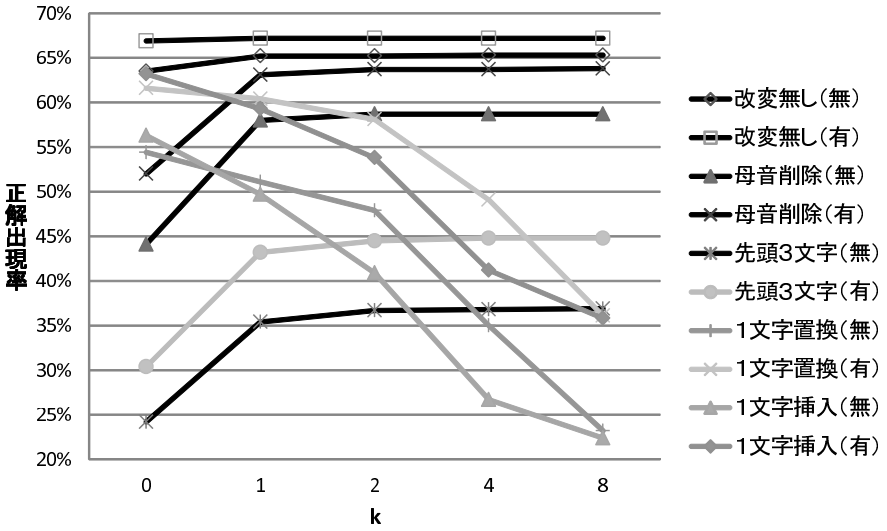

図 5 あいまい入力の類似度にレーベンシュタイン距離を用いた場合について，実数 C の値を 2.33 に固定したときの正解出現率を表したグラフ

Fig. 5 Accuracy rate in the cases of similarity based on Levenshtein Distance with $C = 2.33$.

5%以内の低下で収まることが分かった．

- 「母音削除」，「1 文字置換」，「1 文字挿入」のタスク群の場合，$LCS2$，$LCS4$ を類似度とするあいまい一致を導入すると，20%以上精度が向上することが分かった．「母音削除」のタスク群に限れば，30%以上精度が向上した．
- 「先頭 3 文字」のタスク群の場合，あいまい一致を導入しても，類似度が $LCS2$ のときに最大で精度が約 10%向上するにとどまった（その理由は，単語の先頭の 3 文字だけでは，元の単語をよく表現できていなかったためと推測される）．
- すべてのタスク群において，キーワード分割が無効の場合より有効の場合の方が，精度が平均で約 6%向上した．

図 4 と図 5 は，類似度に $LD(k, C)$ を使用したときの上位 $U = 10$ 件までの出力に対する正解出現率を表したグラフである．図 4 では，k を 1 に固定し，C を 1 から 9 まで動かし，図 5 では，C を 2.33 に固定して k を 0 から 8 まで動かしたものを示している．図 4 の結果から，C を動かしても，あまり精度には影響が表れないことが分かった．このような結果となった理由は，C を動かしても，個々の出力コード片の評価値の大小関係が逆転せず，その順序がほとんど変化しなかったためであると考えられる．より具体的には，C の大きさは類似度の計算式の中で，レーベンシュタイン距離の影響を抑える値であるが，C を大きくしてレーベンシュタイン距離の影響を小さくすることにより，個々の計算における類似度の差を小さくしても，類似度の重み w_4 の大きさ 1 が他の特徴量の重み w_1，w_2，w_3 の大きさ 0.001〜0.01 と比べて大きいために，多くの場合，その他の特徴量による差よりも類似度の差の方が依然として大きく，順序に影響を及ぼさなかったためであると考えられる．

また，図 5 の結果から，k を動かすと，「改変なし」以外のタスク群の場合には，精度に大きく影響するということが分かった．具体的には，「母音削除」と「先頭 3 文字」のタスク群の場合は，k が大きい方が精度が良く，「1 文字置換」と「1 文字挿入」の場合は，逆に k が小さい方が精度が良かった．この理由は，k を大きくすると文字の削除のペナルティに対して，置換と挿入のペナルティが相対的に大きくなるので，「母音削除」と「先頭 3 文字」のタスク群については，低コストの削除操作による文字列変換として正しく認識しやすい一方，「1 文字置換」と「1 文字挿入」のタスク群については，高コストの置換操作と挿入操作による文字列変換として正しく認識するのが困難となるためである．

LCS は文字編集の情報がない分，LD よりも少ない情報を扱うため，より実装が単純であるのにもかかわらず，今回の実験ではほぼすべての場合で LCS の方が精度が高いという結果が得られた．その理由は，LCS の類似度の式 (15)，(16) には，LD の式 (21) には考慮されていない，クエリとラベルに含まれる単語の長さが考慮されているからである．具体的には，「1 文字削除」などの改変されたクエリを入力した場合に，改変前の単語の長さが長い程，類似度の低下は小さくなる．このように単語の長さに対する誤りの長さの割合によって類似度に差が生じることにより，LD よりも精度の高い比較を行うことができていると考えられる．また，LCS は LD の k，C のように調整が必要なパラメータがなく扱いやすい．以上のことから，本論文で

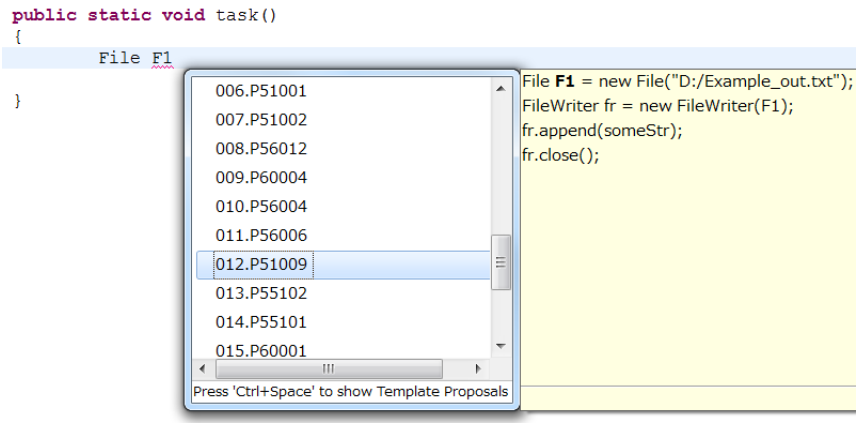

図 6 関連研究 GraPacc [13], [14] を使用中のスナップショット
Fig. 6 A snapshot of related study GraPacc [13], [14].

は，類似度として LD よりも LCS を採用することを推奨する．

なお，本研究では，キーワードの完全一致に基づく検索に比べて，類似度に関わる計算（LCS または LD を求める計算と単語間の対応関係の計算）を行うため時間が増大するが，実用的には問題がないと考えられる．その理由は，全体のアルゴリズムは指数オーダーなのに対して，類似度に関わる計算は多項式オーダーで済むためである．実際，今回本研究で実装したツールを典型的なタスクにおいて使用すると，類似度に関わる計算時間は LCS の場合 100 ms 程度，LD の場合 300 ms 程度であるが，類似度に関わる部分以外の平均計算時間が 200 ms 程度であり，全体としての計算時間をユーザがストレスを感じない応答時間である 1 秒以内に収めるために十分な速度であった．また今回の実装では，LD と LCS を求めるためのアルゴリズムとして動的計画法を用いているが，動的計画法よりも改善されたオーダーで動作するアルゴリズムとして文献 [22], [23] などが提案されており，これらのアルゴリズムを使用することにより，さらなる類似度計算の高速化を実現できると考えている．

6. 比較実験

著者らが調査したところ，本研究と同様に出力が 1 行のコード片であり，高さが 2 以上の関数木を出力することを目的とした研究のうち，そのツールが公開されているものは存在しなかった．そのため比較的似た動作を行う関連研究の中でツールが公開されている研究 GraPacc [13], [14] との比較実験を行った．GraPacc は明示的にクエリを入力する必要はなく，ツールが起動された場所よりも前にすでにユーザが定義している複数のローカル変数を入力として，ツールが起動された場所に適切なスニペットを出力するツールである．このツールを使用中のスナップショットを図 6 に示した．この図中の左側のポップアップに複数のスニペットの番号が表示され，ユーザはその中から 1 つのスニペットを選択することができる．また，ある 1 つの番号の上にカーソルを置くと，そのスニペットの内容が右側にポップアップ表示される．

以下に GraPacc を実際に使用したときの例を順を追って記述する．まずツールの実行前に，ユーザは以下のコードの F1 まで記述し，その行末でツールを実行する．

```
File F1
```

すると，変数 F1 を入力として，複数の候補がポップアップに提示されるが，そのうちの 1 つとしてたとえば以下のようなスニペットをツールから取得することができる．

```
File F1 = new File("D:/Example_out.txt");
FileWriter fr = new FileWriter(F1);
fr.append(someStr);
fr.close();
```

このツールでは，複数行のスニペットが出力され，変数や定数が自動的に定義される．また，someStr などの変数名はあらかじめ用意されたものが付けられる．GraPacc には典型的なパターンのスニペットが多数用意されており，そのようなスニペットが欲しいときに有用である．

それに対し本研究のツールでは，出力のサイズは限定されているが，原理的には関数木によって表現できるどのようなコード片でも出力することができる．たとえば，String 型（文字列型）の変数 path が表すパスに存在するファイルの読み取り権を boolean 型（真理値型）の変数 flg に設定するためのコード片が欲しいときに，クエリとして file readbl と入力しツールを実行すると，以下に示すユーザにとって望ましいコード片が出力候補の 1 番目に出現する（ローカル変数 path と flg は定義済みとする）．

```
new File(path).setReadable(flg)
```

しかし同じ条件下で GraPacc を用いても，ポップアップに提示されたスニペットの候補 67 個のうちに，同じ目的を実現するものは存在しなかった．その理由は，そもそも

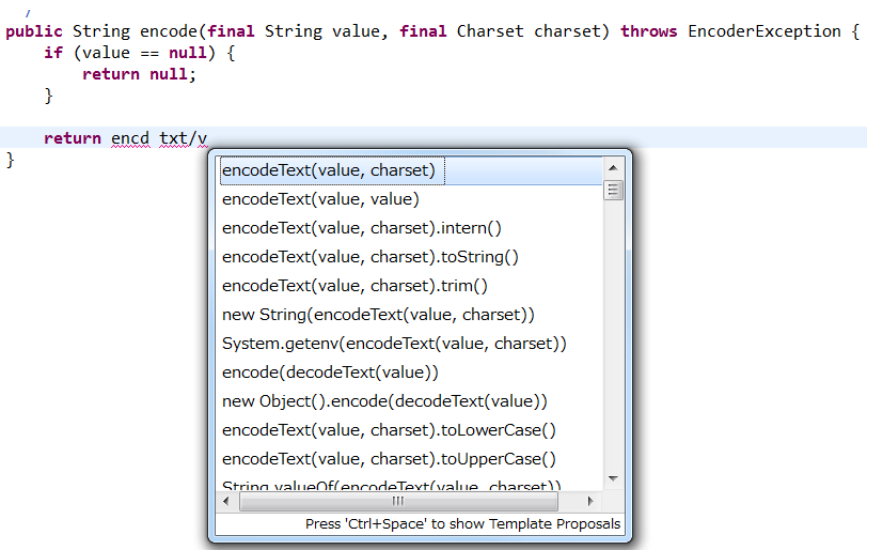

図 7 本研究で実装したプラグインを使用中のスナップショット
Fig. 7 A snapshot of our system implemented as an Eclipse plug-in.

File クラスの setReadable メソッドを用いたスニペットがツール内部のデータベースに登録されていなかったためである．このように，GraPacc のスニペットのパターンは，GrouMiner [24] などのパターン生成ツールを使用するか自作するなどして，事前に用意しておく必要があるが，本研究では API の定義からコード片を合成して出力するため，パターンの用意は必要ない．さらに，GraPacc では入力が暗黙的に取得されるが，本研究ではユーザが明示的にキーワードをクエリとして与えるため，よりユーザの意図が明確に反映されやすいという特徴がある．加えて，候補を選択する際に GraPacc では複数行のスニペットを 1 つ 1 つ確認していかなければならないが，本研究では，1 つの候補は 1 行のコード片であるため，候補選択にかかる負担は小さいと考えられる．

7. まとめと今後の課題

本論文では，キーワードプログラミングシステムに対して，あいまい（不正確）なキーワード入力を導入し，オープンソースプロジェクトを用いた実験により，その効果を議論した．あいまい入力機能の導入によってシステムの精度が大きく下がることはなく，ユーザの認知的または物理的な負担を軽減することができる．

現在，先行研究 [8] の実装は公開されていないため，本論文では筆者らが独自に Eclipse のプラグインとして実装を行った．**図 7** は本研究で実装したプラグインを使用中のスナップショットである．この図は，ユーザにとって望ましいコード片が encodeText(value, charset) のときに，クエリとして encd txt/v と 3 語をスラッシュで関数と変数の 2 種類に区切って入力した例である．実装に使用したプログラミング言語は Java で，ソースコードの行数（コメント行および空行を除く）は約 1 万 5 千行である．Eclipse のプラグインであるため，プラットフォーム非依存の Eclipse が動く多くの OS で使用可能である．作成したプラグインはドキュメントを整備して今後公開する予定である．

今後の課題としては，関数（メソッドやコンストラクタ）のオープンソースプロジェクトなどにおける使用頻度を新しい特徴として取り入れることや，各特徴に対する重み $w_1, \ldots, w_4$ のより良い値を探すための方法の研究などがあげられる．現在の 4 つの重みについては，先行研究 [8] の設定，すなわち「改変なし」のタスク群であいまい入力を無効とした場合について，先行研究が決めた値よりも良い値がないかをグリッドサーチを用いて探索したが，それほど大きく精度向上に寄与する組合せを見つけることができなかった．しかし，あいまい入力を有効とした場合や，他の特徴を取り入れた場合はまた違う値の方が適切となってくると考えられる．また，自然言語処理に関連する研究課題として，キーワードに append や put と入力したときに，プログラミングの際に類義語や同義語として使われることがある add や set も評価値を高めて出力コード片の中に出現可能にする研究などが考えられる．さらに，遠い将来には，この研究や関連研究あるいは自然言語処理などを総合したすべての研究の延長線上で，ユーザがコメントを入力すると，システムが適切なコード片の候補を生成し，入力文字列をそのままコメントとして残すというような統合開発環境の設計が大きなチャレンジとして存在する．

参考文献

[1] Black Duck Software, Inc.: Ohloh Code Search, Ohloh Code (online), available from ⟨http://code.ohloh.net/⟩ (accessed 2014-05-28).
[2] Kim, J., Lee, S., Hwang, S. and Kim, S.: Towards an Intelligent Code Search Engine, *Proc. Twenty-Fourth AAAI Conference on Artificial Intelligence*, pp.1358–

1363 (2010).
[3] Reiss, S.P.: Semantics-Based Code Search, *Proc. 31st International Conference on Software Engineering*, pp.243–253 (2009).
[4] Bruch, M., Monperrus, M. and Mezini, M.: Learning from Examples to Improve Code Completion Systems, *Proc. 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp.213–222 (2009).
[5] Han, S., Wallace, D.R. and Miller, R.C.: Code completion of multiple keywords from abbreviated input, *Automated Software Engineering*, Vol.18, pp.363–398 (2011).
[6] Hou, D. and Pletcher, D.M.: An Evaluation of the Strategies of Sorting, Filtering, and Grouping API Methods for Code Completion, *Proc. 27th IEEE International Conference on Software Maintenance*, pp.233–242 (2011).
[7] Robbes, R. and Lanza, M.: Improving code completion with program history, *Automated Software Engineering*, pp.181–212 (2010).
[8] Little, G. and Miller, R.C.: Keyword programming in Java, *Automated Software Engineering*, Vol.16, pp.37–71 (2009).
[9] Mandelin, D., Xu, L., Bodik, R. and Kimelman, D.: Jungloid Mining: Helping to Navigate the API Jungle, *Proc. 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp.48–61 (2005).
[10] Sahavechaphan, N. and Claypool, K.: XSnippet: Mining For Sample Code, *Proc. 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp.413–430 (2006).
[11] Sakamoto, Y., Sato, H. and Kurihara, M.: Improvement and Implementation of Keyword Programming, *Proc. 2010 IEEE International Conference on Systems, Man, and Cybernetics* (*SMC 2010*), pp.474–480 (2010).
[12] Eclipse Foundation: Code Recommenders, Eclipse Luna (online), available from ⟨http://www.eclipse.org/recommenders/⟩ (accessed 2014-10-01).
[13] Nguyen, A.T., Nguyen, T.T., Nguyen, H.A., Tamrawi, A., Nguyen, H.V., Al-Kofahi, J. and Nguyen, T.N.: Graph-based Pattern-oriented, Context-sensitive Source Code Completion, *Proc. 34th International Conference on Software Engineering*, pp.69–79 (2012).
[14] Nguyen, A.T., Nguyen, H.A., Nguyen, T.T. and Nguyen, T.N.: GraPacc: A Graph-based Pattern-oriented, Context-sensitive Code Completion Tool, *Proc. 34th International Conference on Software Engineering*, pp.1407–1410 (2012).
[15] Hill, R. and Rideout, J.: Automatic Method Completion, *Proc. 19th IEEE International Conference on Automated Software Engineering*, pp.228–235 (2004).
[16] Omar, C., Yoon, Y., LaToza, T.D. and Myers, B.A.: Active Code Completion, *Proc. 34th International Conference on Software Engineering*, pp.859–869 (2012).
[17] Bergroth, L., Hakonen, H. and Raita, T.: A Survey of Longest Common Subsequence Algorithms, *Proc. the Seventh International Symposium on String Processing Information Retrieval* (*SPIRE'00*), pp.39–48 (2000).
[18] Navarro, G.: A Guided Tour to Approximate String Matching, *ACM Comput. Surv.*, Vol.33, pp.31–88 (2001).
[19] 藤澤信夫，森田和宏，泓田正雄，青江順一：部分文字列を用いた綴り誤りに対する訂正候補の高速検索手法，言語処理学会第 13 回年次大会，pp.962–965 (2007).
[20] 高城泰宏，田中栄一：綴り誤りの高速訂正法，電子情報通信学会技術研究報告，Vol.95, pp.1–8 (1996).
[21] Okuda, T., Tanaka, E. and Kasai, T.: A Method for the Correction of Garbled Words Based on the Levenshtein Metric, *IEEE Trans. Comput.*, Vol.25, pp.172–178 (1976).
[22] Myers, E.W.: An O(ND) Difference Algorithm and Its Variations, *Algorithmica*, Vol.1, pp.251–266 (1986).
[23] Wu, S., Manber, U., Myers, G. and Miller, W.: AN O(NP) SEQUENCE COMPARISON ALGORITHM, *Information Processing Letters*, Vol.35, pp.317–323 (1990).
[24] Nguyen, T.T., Nguyen, H.A., Pham, N.H., Al-Kofahi, J.M. and Nguyen, T.N.: Graph-based Mining of Multiple Object Usage Patterns, *Proc. 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pp.383–392 (2009).

坂本 悠輔 （学生会員）

2008 年北海道大学工学部情報工学科卒業．2010 年同大学大学院修士課程修了．同年大学院博士課程入学．電子情報通信学会，日本ソフトウェア科学会各会員．

佐藤 晴彦 （正会員）

2005 年北海道大学工学部情報工学科卒業．2007 年同大学大学院修士課程修了．2008 年同大学院博士課程修了．情報科学博士．2009 年北海道大学助教．定理自動証明の研究に従事．電子情報通信学会，日本ソフトウェア科学会各会員．

小山 聡 （正会員）

1994 年京都大学工学部数理工学科卒業，1996 年同大学大学院工学研究科修士課程修了．日本電信電話株式会社，京都大学大学院情報学研究科博士後期課程，日本学術振興会特別研究員（DC），京都大学大学院情報学研究科助手，スタンフォード大学 Visiting Assistant Professor 等を経て，2009 年より北海道大学大学院情報科学研究科准教授．博士（情報学）．主な研究分野は機械学習，データマイニング，情報検索，クラウドソーシング等．2005 年度人工知能学会論文賞，2009 年度日本データベース学会上林奨励賞受賞．

栗原 正仁 （正会員）

1978 年北海道大学工学部電気工学科卒業．1980 年同大学院工学研究科情報工学専攻修士課程修了．同年北海道大学助手．その後，講師，助教授，および北海道工業大学教授を経て，2002 年北海道大学大学院工学研究科コンピュータサイエンス専攻教授．現在，同大学院情報科学研究科複合情報学専攻教授．工学博士．人工知能およびソフトウェア科学の研究に従事．電子情報通信学会，日本ソフトウェア科学会，日本知能情報ファジィ学会各会員．