



HOKKAIDO UNIVERSITY

Title	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に関する研究
Author(s)	坂本, 悠輔
Degree Grantor	北海道大学
Degree Name	博士(情報科学)
Dissertation Number	甲第11746号
Issue Date	2015-03-25
DOI	https://doi.org/10.14943/doctoral.k11746
Doc URL	https://hdl.handle.net/2115/58715
Type	doctoral thesis
File Information	Yusuke_Sakamoto.pdf



平成26年度

博士論文

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
に関する研究

Study on Keyword Programming Systems
with Ambiguous Keyword Matching

北海道大学 大学院情報科学研究科
複合情報学専攻 複合情報工学講座
知能ソフトウェア研究室

坂本 悠輔

平成27年3月25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よ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）であってなるべく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。

目次

第1章	はじめに	7
第2章	関連研究	15
第3章	キーワードプログラミング	18
3.1	モデル	18
3.2	ノードの評価と特徴ベクトル	19
3.3	説明ベクトルによる関数木の評価	20
3.4	関数木生成アルゴリズム	22
第4章	複数候補を提示するキーワードプログラミング	27
4.1	キーワードと文脈に基づく関数木の評価	27
4.2	特徴ベクトルと評価ベクトル	28
4.3	関数木生成アルゴリズム	29
第5章	キーワードあいまい一致	34
5.1	最長共通部分列	34
5.2	レーベンシュタイン距離	35
5.3	単語間の対応関係の計算	36
5.4	キーワード分割	36
5.5	実験	37
5.6	考察	38
第6章	ツールの実装	41
6.1	統合開発環境 Eclipse	41
6.2	Eclipse プラグイン開発	43
6.3	ツールの設計	45
6.3.1	ツールの使用方法	47
6.4	ローカルサーチによる特徴の重みの学習	48
6.5	学習の実行時間の短縮	49
6.6	グリッドサーチ	53
6.7	実装上の工夫	53
6.8	実験と考察	56

6.8.1	実験 1	56
6.8.2	実験 1 の考察	57
6.8.3	実験 2	57
6.8.4	実験 2 の考察	58
6.8.5	実験 3	59
6.8.6	実験 3 の考察	59
6.8.7	その他の考察と考察のまとめ	59
6.9	関連研究との比較実験	60
第 7 章 まとめ		64
謝辞		66
参考文献		67
研究業績		70

図 目 次

1.1	ソースコード検索サイト Ohloh Code の結果出力画面のスナップショット. この結果出力画面は, 入力クエリとして sort と list の 2 つのキーワードを入れ, 結果の絞りこみ機能を用いて, プログラミング言語は Java のみを指定し, 定義はメソッドのみを指定した時のものである. 検索結果の総数は 3268 件であり, このスナップショットには 2 つのプロジェクト (Apache Tuscany と boon) のコードスニペットが表示されている.	10
1.2	統合開発環境 Eclipse の画面のスナップショット.	11
1.3	統合開発環境 Eclipse のコードコンプリーション (コード補完) を実行し, ポップアップが表示された際の画面のスナップショット. この画面は, ユーザが現在エディタ上に見えているソースコードの上の方で既に定義されている String 型 (文字列型) のオブジェクトであるローカル変数 keyword の名前を入力後に, ピリオドを入力したときの画面である. ピリオドを入力すると自動的にエディタは, String 型のクラスに定義されているメソッド一覧をメソッドの名前のアルファベット順にポップアップに提示してくれるので, ユーザはこのポップアップの中から自分の使いたいメソッドを選択することができる.	12
1.4	ウェブ検索エンジン Google. この画像はキーワードとして, 「キーワード」と「プログラミング」の 2 語を入力した時の検索結果である.	13
1.5	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. キーワードプログラミングではユーザが幾つかのキーワードを入力し, コマンド (Ctrl+Space) を押す事でその現在の文脈 (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.	14
1.6	実際に作成したツールを起動し, ポップアップを表示させたときのスナップショット. このツールはポップアップに表示された複数の候補からユーザの欲しいものを選択することができる. この図はユーザの欲しい出力が fruits.contains (food) である時に, クエリーとしてその欲しい出力に存在する全てのラベル fruits, contains, food を入力したときのスナップショットである.	14

3.1	array.add(src.readLine()) を木構造で表現した図. ノードは 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.	19
3.2	クエリーが図 2 の状況で add line であるときに, アルゴリズム 1 と 2 によって作られる bestRoots オブジェクトを表した, 木の高さを縦軸, 型を横軸とした関数木の根ノードの表. 表の各交点には最大で 3 つの根ノードを保持することができ, 上段の大きい枠 1 つと下段の小さい 2 つでそれを表現している. 上段の大きい枠が評価値が最大の根ノード, 下段左が 2 番目, 下段右が 3 番目の根ノードを表す. 枠で表現されている各根ノード内部の記述は, 1 行目が評価値 (および説明ベクトル), 2 行目が関数の返り値の型または変数の型, 3 行目が関数または変数名, 4 行目が関数の引数の型である.	25
5.1	あいまい一致の機能が無効の時と, $LCS1 \sim LCS4$ を類似度としたあいまい一致の機能が有効な時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.	38
5.2	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.	39
5.3	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.	40
6.1	実際に作成したツールを起動し, ポップアップを表示させたときのスナップショット. このツールはポップアップに表示された複数の候補からユーザの欲しいものを選択することができる. この図はユーザの欲しい出力が fruits.contains (food) である時に, クエリーとしてその欲しい出力に存在する全てのラベル fruits, contains, food を入力したときのスナップショットである.	42
6.2	本研究で実装したツールでは, ツールに学習させたいタスク (入力と出力の組み合わせ. 出力はユーザがドラッグして選択するコード片. 入力はそのコード片から自動生成されるキーワードと周辺ソースコード状況) を選択して追加することができる. この図は xr.parse(new InputSource(is)) というコード片を正解とするタスクを追加する作業を表している.	43
6.3	図 6.2 で示される作業によって追加したタスクを一覧することができるビュー. 1 行が 1 つのタスクを表す. ある 1 つの出力に対応するクエリーはその出力関数木に含まれる全ての関数および変数のラベルとして自動的にセルに入るが, ユーザが後から編集することも可能となっている.	44

6.4	Eclipse プラットホーム	45
6.5	Eclipse ワークベンチ	46
6.6	作成したツールの設計概略図	47
6.7	Eclipse のコンテンツ・アシスト機能を起動したところ. JDT のエディタ上では, オブジェクト名の後に'.'(ピリオド)を打つとその後 に記述可能なフィールドやメソッドの候補が表示される. 任意の位置 でコンテンツ・アシスト機能を使用したいときは『Ctrl+Space』 の入力で起動できる.	48
6.8	ツールを使用する前に提示候補のソーティングを有効にしておく.	49
6.9	ローカルサーチの処理時間短縮手法の処理の流れを表した図. ロー カルサーチの前処理として, アルゴリズム 4, 5, 6 によって生成さ れた出力候補群から各候補の特徴ベクトル $(x_1, x_2, x_3, y_1, \dots, y_n)$ を 算出しておく. そして, その特徴ベクトルと近傍の重みベクトル (w_1, w_2, w_3, w_4) から, その近傍における評価値(説明ベクトルの要 素の値の総和)を計算する. 次に, その評価値の降順に出力候補群 をソートする. 最後に, 正解の順位の逆数としてスコアを求める. この手法の特徴は, はじめに一度だけアルゴリズム 4, 5, 6 を実行 し特徴ベクトルを抽出しておけば, 近傍についてはそれらを実行せ ずに評価値を計算することが可能な点である.	52
6.10	Eclipse のエディタ上で編集するソースファイルの切り替え前後の状 態を表した図. 初めに IceCreamDemo.java が選択されていて, 次に ユーザが Dialog.java にファイルを切り替えた所である(タブの色が変 化している). この際, メモリ上に保持されていた IceCreamDemo.java 中でインポートされたクラス群の情報は破棄され, Dialog.java 中で インポートされているクラス群の情報が読み込まれる.	55
6.11	ツールを起動する場所の状況のパターンの例を表した 4 つの図. こ の 4 つの図は青く表示されている行にあるキャレットの位置でユー ザが補完コマンド『Ctrl+Space』を押してツールを起動する直前の 状態を示したものである. 上から 1 番目の図 (Task10) が「クエリー と同じ行の左方向に何もない場合」, 2 番目の図 (Task08) が「変数 宣言が左側にある場合」, 3 番目の図 (Task02) が「メソッドの引数 の中で使用した場合」, 4 番目の図 (Task04) が「return 文の中で使 用した場合」を表した図である. それぞれの状況により, ツールが 出力するコードが返すべき返り値の型が異なるため, その状況ごと に場合分けをしてその型情報を得ている.	62

- 6.12 先行研究 [1] の著者作成の 14 問のタスクに対して，関数および変数を評価する特徴の重みの組み合わせを一定の範囲で動かし「出力中に出現した正解の順位の逆数」をスコアとし，その全 14 タスクの合計値をプロットした 3 次元グラフ．図の X, Y, Z 座標はそれぞれ重み w_1, w_2, w_3 を表す (重み w_4 は 1.0 で固定)．スコアが多い交点ほど点が大きく赤く，少ない交点ほど小さく青く表示されていて，問題数が 14 問なので最大値は 14 となっている．また，このグラフは検索結果上位 10 件までの出力結果を調べたグラフである． 63
- 6.13 関連研究 GraPacc [2, 3] を使用中のスナップショット．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，ユーザはそこから 1 つのスニペットを選択することができる．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． 63
- 7.1 本研究で作成したプラグインを使用中のスナップショット．ユーザにとって望ましいコード片が `encodeText(value, charset)` の時に，クエリとして `encl 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． 65

第1章 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 [4] (図 1.1) 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 [5, 6]。またもう1つの例として、Eclipse などの統合開発環境 [7] (図 1.2) には、ユーザが現在編集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ション (code completion) という機能が備わっている (図 1.3)。

近年、このコードコンプリ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 [8, 9, 10, 11] 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 [1] 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、まるでウェブ検索エンジン (図 1.4) 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 1.5)。

この研究における、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捉えることができる。キーワードプログラミング以前の研究 [12, 13] 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事が可能となった。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 [14] (図 6.1)。

キーワードプログラミングは大規模なソースコードのコーパスからコード例を探索する研究との関連がある。それらの研究の中でも、最近発表された代表的なものを紹介する。Prospector [12] は、入力として Java の型を 2 つ必要とし、そのうちの 1 つの型からもう 1 つの型への型変換を行うようなコード片を返すシステムである。Prospector はある型から違う型への変換が分からない時にとても有効なシステムである。また違うシステムに XSnippet [13] がある。このシステムはユーザの欲しい型や、ソースコードのコンテキストに含まれているローカル変数の全ての型に基づいてコード片を扱うものである。また、このシステムはコード片を幾つかのヒューリスティックス（コード長や、使用頻度など）を用いてどのコード片がよりコンテキストに近いかをランク付けをしている。3 つ目の研究としては、Automatic Method Completion [15] がある。これは部分的に記述したメソッド本体を使用して、そのコードを完成させるようなコード片をコーパスから検索するというものである。これらの関連研究とキーワードプログラミングの違いは大きく 3 つある。

1. ユーザの入力が型に制約されない。また、出力はある特定の型を返すようなコード片ではなく、その場面に適切だと判断されたものが出てくる。
2. ユーザから与えられる比較的曖昧なキーワードクエリによってアドバイスを与えるものである。『曖昧な』というのは、それらのキーワードはメソッドにマッチするかもしれないし、ローカル変数にマッチするかもしれない、はたまたフィールドかもしれない、という意味である。
3. この研究は新しいコードを生成する。したがってコード片を検索するのに必要な既存のコーパスを必要としない。このため今始まったばかりのとても小さなプロジェクトに対して特に有効である。

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通りである。続く第 2 章で、コードコンプリション分野の関連研究の紹介を行う。第 3 章では、先行研究 [1] にて提案されたキーワードプログラミングについての解説を行う。第 4 章では、複数候補を提示するように先行研究 [1] を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。第 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。第 6 章では、本研究で統合開発環境 Eclipse のプラグ

インとして実装したツールの解説と，そのツールを用いた実験および考察について述べる．最後の第7章では，本論文のまとめと今後の課題を述べる．

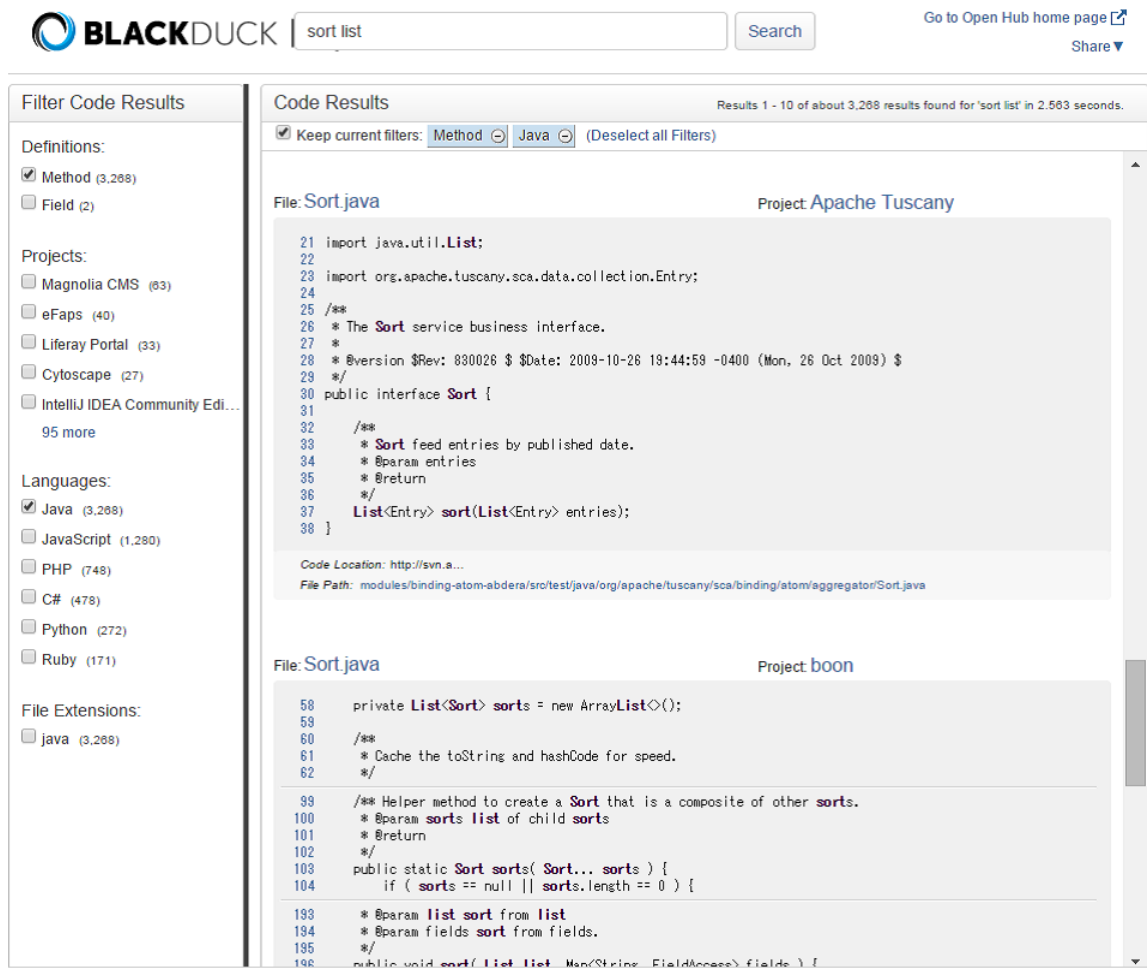


図 1.1: ソースコード検索サイト Ohloh Code の結果出力画面のスナップショット。この結果出力画面は、入力クエリとして sort と list の 2 つのキーワードを入れ、結果の絞りこみ機能を用いて、プログラミング言語は Java のみを指定し、定義はメソッドのみを指定した時のものである。検索結果の総数は 3268 件であり、このスナップショットには 2 つのプロジェクト (Apache Tuscany と boon) のコードスニペットが表示されている。

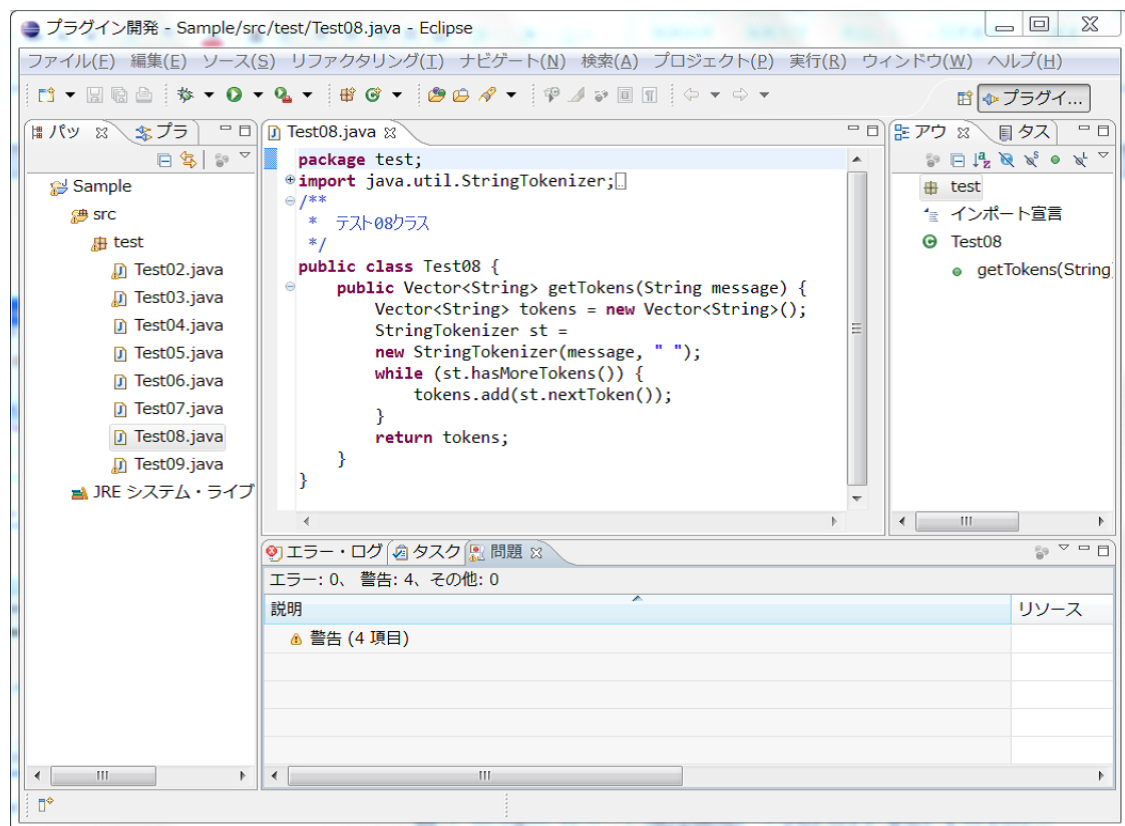


図 1.2: 統合開発環境 Eclipse の画面のスナップショット.



図 1.4: ウェブ検索エンジン Google. この画像はキーワードとして、「キーワード」と「プログラミング」の 2 語を入力した時の検索結果である。

```

public List<String> getLines(BufferedReader src) throws Exception{
> List<String> array = new ArrayList<String>();
> while(src.ready()){
>     add Line
> }
> return array;
}

```



Ctrl + Space

```

public List<String> getLines(BufferedReader src) throws Exception{
> List<String> array = new ArrayList<String>();
> while(src.ready()){
>     array.add(src.readLine());
> }
> return array;
}

```

図 1.5: 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. キーワードプログラミングではユーザが幾つかのキーワードを入力し, コマンド (Ctrl+Space) を押す事でその現在の文脈 (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.

```

public boolean isFruit(String food) {
    Set<String> fruits = new HashSet<String>(getFruitList());
    return fruits contains food
}

```

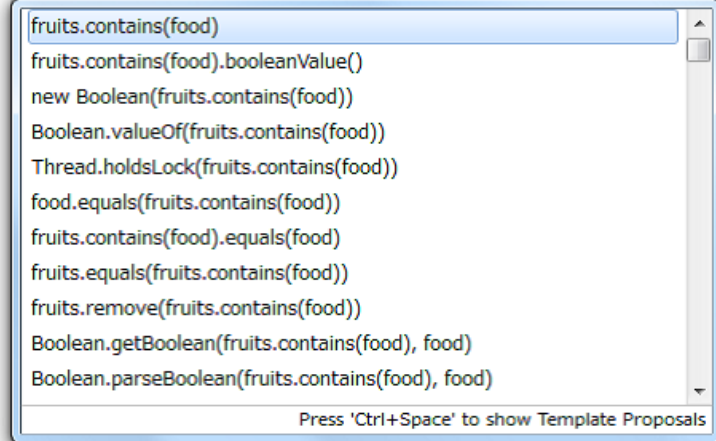


図 1.6: 実際に作成したツールを起動し, ポップアップを表示させたときのスナップショット. このツールはポップアップに表示された複数の候補からユーザの欲しいものを選択することができる. この図はユーザの欲しい出力が `fruits.contains(food)` である時に, クエリーとしてその欲しい出力に存在する全てのラベル `fruits`, `contains`, `food` を入力したときのスナップショットである.

第2章 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 [16] 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 [9] 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．例えば、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、既に望むコード片を明確に把握している時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み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[12] と XSnippet[13] 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．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．XSsnippet 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、XSsnippet 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 [2, 3] は、ツールが起動された場所よりも前に既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 XSsnippet 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無い点が

特徴である。

Graphite [17] は、1つのインター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ーフェースを提供するツールの研究である。例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、数値で指定する事が一般的であるが、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ー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 2.1 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。

表 2.1: 関連研究との比較

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 [7]	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 [9]	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 [12]	2 つの型名	スニペット	API 定義とコードコーパス
XSnippet [13]	2 つの型名	スニペット	コードコーパス
GraPacc [2, 3]	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 [17] (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

第3章 キーワードプログラミング

本章では、先行研究 [1] 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。

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**、メンバー変数及びローカル変数を**変数**と定義し、 $T \times L \times T \times \dots \times T$ に属するタプルで表現し、その集合を F とする。先頭の T は関数の戻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。例えば、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 の戻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

$$params(f) = \{params(f)_i \mid 1 \leq i \leq n\} \quad (3.1)$$

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。

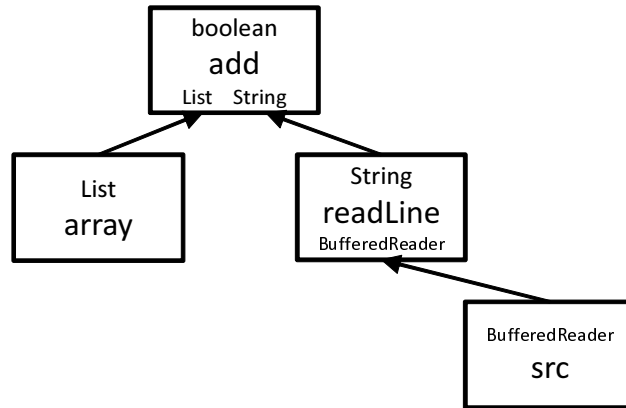


図 3.1: `array.add(src.readLine())` を木構造で表現した図. ノードは 1 行目が関数の返り値の型または変数の型, 2 行目が関数名または変数名, 3 行目が関数の引数の型である.

また, レシーバ r のメソッド f を k 個の引数 $a_1, \dots, a_k$ で呼び出すメソッド呼び出し $r.f(a_1, \dots, a_k)$ は, 関数呼び出し $f(r, a_1, \dots, a_k)$ とみなされ, f を根ノード, $r, a_1, \dots, a_k$ 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 3.1) .

3.2 ノードの評価と特徴ベクトル

システムが生成する関数木の好ましさを評価するために, 木を構成する各ノードをその「特徴」に基づいて評価する. 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. 先行研究 [1] では,

$$\begin{aligned} w_1 &= -0.05, \\ w_2 &= -0.01, \\ w_3 &= 0.001, \\ w_4 &= 1.0. \end{aligned}$$

と定義されていた (これらの固定値についての根拠は明示されていなかった). そして, ソースコード上のある位置でユーザがキーワードの集合をクエリーとして入力し, 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, そのコード片を表す関数木の好ましさを評価するため, 次の 4 つの**特徴**に従ってその関数木の各ノード (すなわち, 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. 全てのノードについて, 無条件で評価値を $w_1 (< 0)$ に初期設定する. これは, なるべく少ないノード数の関数木が好ましいとするためである.

2. ノードのラベルに存在する単語のうち、クエリーの中にキーワードとして与えられていない単語 1 つごとに評価値に $w_2(< 0)$ を加算する. これは, ラベルに含まれるキーワード以外の単語が少ないほど, 冗長性が小さく好ましいと考えるからである.
3. ノードが今の**周辺ソースコード状況**におけるローカル変数, メンバー変数やメンバーメソッドである場合には, 評価値に $w_3(> 0)$ を加算する. 周辺ソースコード状況とは, 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) 内に当該コード片の挿入位置が含まれているものの名前の集合と定義する. これは, 周辺ソースコード状況により表現されたユーザの文脈で用いられているラベルは, そうでないものよりも好ましいという考えに基づいている.
4. ノードのラベルに含まれるキーワード 1 つにつき, 評価値に $w_4(> 0)$ を加算する. これは, ラベルにキーワードが多数含まれているほど好ましいと考えるからである.

以上の特徴 1,2,3 に関係して, 評価値を数式で表現するために, ノードに対し, つぎの 3 つの**特徴量**を定義する.

x_1 : 値は常に 1.

x_2 : ノードのラベルには存在するが, クエリーの中には存在しないキーワードの数.

x_3 : ノードが今の周辺ソースコード状況におけるローカル変数またはメンバー変数またはメンバーメソッドであれば 1, そうでなければ 0.

次に, 特徴 4 に関連して, クエリーの中に含まれるキーワードが $k_1, k_2, \dots, k_n$ の n 個 ($k_1, \dots, k_n$ には重複する要素があってもよい) であるとき, i 番目のキーワードに関する**特徴量**を

$$y_i = \min(P_i/Q_i, 1) \quad (3.2)$$

(P_i はノードのラベルの中に含まれる k_i の出現回数, Q_i はクエリーに含まれる k_i と同じキーワードの出現回数) と定義する. また, $n+3$ 個の要素を持つベクトル $(x_1, x_2, x_3, y_1, \dots, y_n)$ を**特徴ベクトル**と定義する. 関数木を構成する全てのノードの特徴ベクトルの和を, その関数木の特徴ベクトルという.

3.3 説明ベクトルによる関数木の評価

関数木全体の評価値を定義するため, **説明ベクトル** (explanation vector) と呼ばれるベクトルを定義する. このベクトルは, 3.2 節の考え方にに基づき, まず関数木

のノード（関数および変数）に対して定義される．その後，定義が関数木全体に対するものに拡張され，関数木の評価値が定義される．まず，ノードに対する説明ベクトルであるが，これは，キーワードの個数を n とするとき， $e_0, e_1, e_2, \dots, e_n$ の $n+1$ 個の実数値を要素として持つベクトルである．ただし， e_0 以外の要素は 0 以上 1 以下である．このベクトルの要素のうち， e_0 は 3 つの特徴量とその重みを用いて，

$$e_0 = \sum_{i=1}^3 w_i x_i \quad (3.3)$$

で定義される．また， e_0 以外の要素 e_i は各 k_i と 1 対 1 に対応して， i 番目のキーワードをどの程度良く説明するかを表現する要素であり，対応する特徴量と第 4 の重みを用いて，

$$e_i = \min(w_4 y_i, 1) \quad (3.4)$$

で定義される． f がノードであるとき，その説明ベクトルを $\text{expl}(f)$ で表す． $\text{expl}(f)$ の全ての要素の和を， $\text{expl}(f)$ の評価値，または f の評価値という．また，説明ベクトルの和 $(e_0, \dots, e_n) + (e'_0, \dots, e'_n)$ を $((e_0 + e'_0), \min(e_1 + e'_1, 1), \dots, \min(e_n + e'_n, 1))$ と定義する．これを用いて，関数木の説明ベクトルを，その木に含まれる全てのノードの説明ベクトルの和と定義する．また，関数木の評価値を，その関数木の説明ベクトルの評価値と定義する．なお，便宜上，説明ベクトル同士あるいは説明ベクトルと実数値の大小比較を行う数式の記述があるときは，説明ベクトルはその評価値を表すものと解釈して，実数値同士の大小比較に帰着させる．

ここで，説明ベクトルを使って評価値を計算する例を示す．計算に使用する重みの値は，3.2 節の冒頭に掲載した先行研究 [1] にて定義された固定値を用いる．今，クエリーが，

is queue empty

であり，評価値を計算したいノードが，関数

(boolean, (**is**, **empty**), List)

であり，この関数は周辺ソースコード状況に存在する関数（メンバーメソッド）であるとする．この関数の説明ベクトルの e_0 の値は，まず，特徴 1 により -0.05 に初期設定される．特徴 2 については，ラベルには存在するが，クエリーの中には存在しない単語はないので，0 が加算される．特徴 3 については，関数が周辺ソースコード状況に存在する関数であるため，0.001 が加算される．最終的に， $e_0 = -0.049$ となる．特徴 4 に基づき，各キーワードの出現回数を追加すると，説明ベクトルは，

$(-0.049, 1_{\text{is}}, 0_{\text{queue}}, 1_{\text{empty}})$

となる（ここでは数値とキーワードの対応を示すために便宜上，上の表現を用いているが，実際のベクトルは $(-0.049, 1, 0, 1)$ のように，要素はすべて数値である）．

別な例として，クエリーが，

node parent remove node

であり，変数が，

(TreeNode, (**node**))

というローカル変数であった場合、説明ベクトルは、

(-0.049, 0.5_{**node**}, 0_{**parent**}, 0_{**remove**}, 0.5_{**node**})

となる．ここでは、キーワード **node** に対応する部分の評価値が 0.5 という値になっているが、これは **node** というキーワードがクエリーには 2 個存在し、変数のラベルには 1 つだけ存在する状況を表現するために、(3.2) 式により、評価値が 2 つの要素に分散されたためである．

3.4 関数木生成アルゴリズム

先行研究では、はじめにアルゴリズム 1 を実行し、次にアルゴリズム 3 を実行することによって、クエリーから望ましい出力を算出している．アルゴリズム 2 はアルゴリズム 1 の中から呼び出されて実行される．

アルゴリズム 1

アルゴリズム 1 は、3.1 節で定義した集合 T に存在する全ての型 t について、返り値の型が t で高さが i ($1 \leq i \leq h$) の関数木のうち、説明ベクトルの評価値が上位 r 番目までのものの根ノードを $\text{bestRoots}(t, i)$ に格納するアルゴリズムである．説明ベクトルを計算する際にアルゴリズム 2 を呼び出して計算を行う．先行研究では最大の高さ h を 3 としている．なお、先行研究では集合 S と単元集合 $\{a\}$ の和集合を $S \cup a$ と略記しており、本論文でもアルゴリズム 1 などでの表記を採用している．

アルゴリズム 2

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（つまり、引数を表す関数木の高さが $h_{\max}$ 以内）のうち評価値が最大のものを探査し、その説明ベクトルを出力するアルゴリズムである．アルゴリズム 1 と 2 では、関数木の組み合わせが膨大であり計算に時間がかかるため、計算量を削減するために、動的計画法と同様な考え方に基づき、一度計算した返り値の型が t で高さが i の関数木を上位 r 番目まで保存しておき、あとからそれより高い高さ ($> i$) の関数木の子ノードとして同じ関数木の説明ベクトルを計算する必要があるよう工夫を行っている．アルゴリズム 1 とアルゴリズム 2 によって作成された bestRoots の例を図 3.2 に示す．

アルゴリズム 1 先行研究 [1] の動的計画法のアゴリズム

```
procedure DynamicProgram()
/*この関数は bestRoots を改変する関数である. */
for each  $1 \leq i \leq h$  do
  for each  $t \in T$  do
     $bestRoots(t, i) \leftarrow \emptyset$ 
    for each  $f \in F$  where  $ret(f) \in sub(t)$  do
       $e \leftarrow \text{GetBestExplForFunc}(f, i - 1)$ 
      if  $e > -\infty$  then
         $bestRoots(t, i) \leftarrow bestRoots(t, i) \cup (e, f)$ 
      end if
      /*評価値上位  $r$  番目までのノードを残す*/
       $bestRoots(t, i) \leftarrow \text{GetBestN}(bestRoots(t, i), r)$ 
    end for
  end for
end for
```

アルゴリズム 2 先行研究 [1] の評価値が最大の説明ベクトルを計算するアルゴリズム

```
procedure GetBestExplForFunc( $f, h_{max}$ )
 $e_{cumulative} \leftarrow expl(f)$ 
for each  $p \in params(f)$  do
   $e_{best} \leftarrow (-\infty, 0, 0, 0, \dots)$ 
  for each  $1 \leq i \leq h_{max}$  do
    for each  $(e', f') \in bestRoots(p, i)$  do
      if  $e_{cumulative} + e' > e_{best}$  then
         $e_{best} \leftarrow e_{cumulative} + e'$ 
      end if
    end for
  end for
   $e_{cumulative} \leftarrow e_{best}$ 
end for
return ( $e_{cumulative}$ )
```

アルゴリズム 3

アルゴリズム 3 は、返り値の型 t と高さ h と説明ベクトル e が与えられたときに、返り値の型 t で高さが最大 h の関数木のうち、評価値が最大のものを抽出し、その関数木を構成するノード n_{best} と、その説明ベクトル e_{best} を出力するアルゴリズムである。ただし、この再帰的アルゴリズムを呼び出すトップレベルでは、 e の初期値としてゼロベクトルを与える。再帰呼び出しの過程においては、 e に種々の説明ベクトルが加算され、現在再帰的に抽出中の（不完全な）関数木の説明ベクトルが e に保持される。 $|sub(t_1)| < |sub(t_2)|$ となるとき、 t_1 は t_2 よりも小規模な型と定義し、より小規模な型が前方に来るようにソートが実行される。

アルゴリズム 3 先行研究 [1] の関数木抽出アルゴリズム

```
procedure ExtractTree( $t, h, e$ )
   $e_{best} \leftarrow 0$ 
   $n_{best} \leftarrow null$ 
  for each  $1 \leq i \leq h$  do
    for each  $(e_{ignore}, f) \in bestRoots(t, i)$  do
      /*関数木のノードのタプルを作成する*/
       $n \leftarrow (f)$ 
       $e_n \leftarrow e + expl(f)$ 
      /*より小規模な型が前方に来るようにソートする*/
       $P \leftarrow Sort(params(f))$ 
      for each  $p \in P$  do
         $n_p, e_p \leftarrow ExtractTree(p, i - 1, e_n)$ 
        /*  $n_p$  を  $n$  の子ノードとして加える*/
         $n \leftarrow append(n, n_p)$ 
         $e_n \leftarrow e_n + e_p$ 
      end for
      if  $e_n > e_{best}$  then
         $e_{best} \leftarrow e_n$ 
         $n_{best} \leftarrow n$ 
      end if
    end for
  end for
  return  $(n_{best}, e_{best})$ 
```

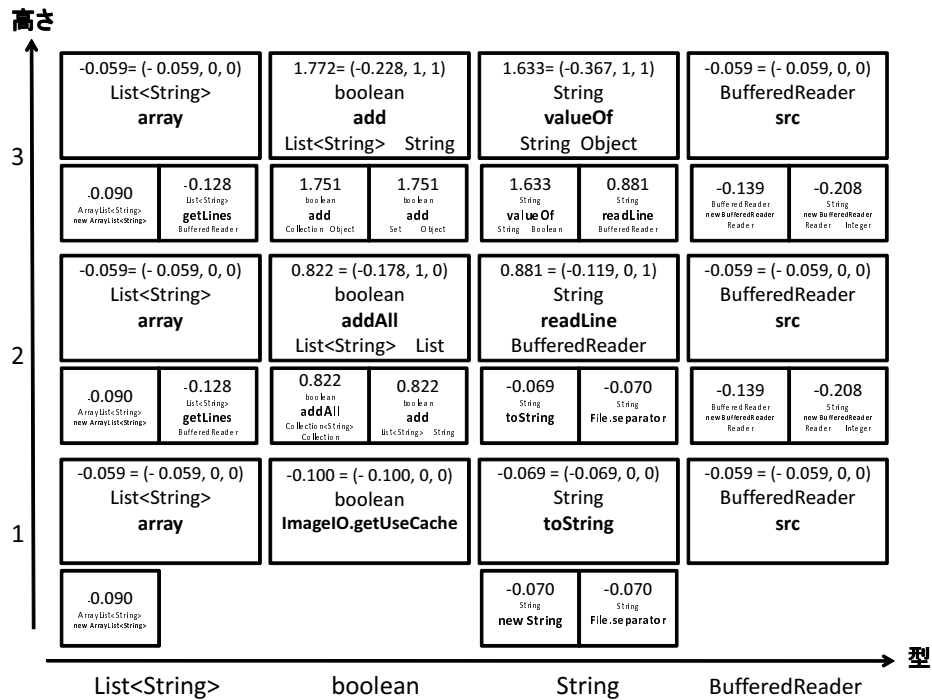


図 3.2: クエリーが図 2 の状況で add line であるときに、アルゴリズム 1 と 2 によって作られる bestRoots オブジェクトを表した、木の高さを縦軸、型を横軸とした関数木の根ノードの表。表の各交点には最大で 3 つの根ノードを保持することができ、上段の大きい枠 1 つと下段の小さい 2 つでそれを表現している。上段の大きい枠が評価値が最大の根ノード、下段左が 2 番目、下段右が 3 番目の根ノードを表す。枠で表現されている各根ノード内部の記述は、1 行目が評価値 (および説明ベクトル)、2 行目が関数の戻り値の型または変数の型、3 行目が関数または変数名、4 行目が関数の引数の型である。

アルゴリズムの動作の概要を理解するために、ここで図 3.2 において、型が String で高さが 2 の所にある関数ノード readLine について考えてみる。この図はちょうど、図 1.5 のソースコードの状況における bestRoots オブジェクト (アルゴリズム 1 を参照) の内部を表したものである。この readLine を根とする関数木の説明ベクトルを計算すると、図に記載されている通り $(-0.119, 0, \text{add}, 1, \text{line})$ となるのであるが、これがどのようにして計算されているのかを順を追って説明する。まずノード readLine 単体での説明ベクトルを計算すると $(-0.06, 0, \text{add}, 1, \text{line})$ となる。この説明ベクトルの e_0 の値 -0.06 は、初期値 -0.05 と、クエリーに含まれない余分な単語 read についての -0.01 の和である。次に、readLine は引数に型 BufferedReader を取るのでこの部分も計算に含めなくてはならない。図 3.2 を見ると高さ 2 にある関数 readLine の 1 つ下の高さで型が BufferedReader の位置にある関数ノードは src しか無いので、これが readLine を根ノードとする関数木の子ノードとなる。

この src の説明ベクトルは、 $(-0.059, 0_{\text{add}}, 0_{\text{line}})$ であり、これと先程計算した readLine の説明ベクトル $(-0.06, 0_{\text{add}}, 1_{\text{line}})$ の和を取ることで、関数木に対する説明ベクトル $(-0.119, 0_{\text{add}}, 1_{\text{line}})$ が得られる。

第4章 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 [1] 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, [1] 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 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 [1] 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. [1] 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.

つぎに, 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. 全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, メンバー変数またはメンバーメソッド) 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、つぎの3つの**特徴量**を定義する。

x_1 : ノードの数. すなわち、値は常に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つぎの式で与えられる。

$$e_0 = w_1x_1 + w_2x_2 + w_3x_3 \quad (4.1)$$

最終評価値を得るためには、つぎ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、キーワード毎に加算値の上限を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。

4.2 特徴ベクトルと評価ベクトル

前節末のステップ (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つぎの式で与えられる。

$$e_i = 0 \oplus w_4y_i \quad (1 \leq i \leq n) \quad (4.2)$$

ただし、演算子 $\oplus$ は、[1] 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。

$$x \oplus y = \min(x + y, 1) \quad (4.3)$$

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 $(x_1, x_2, x_3, y_1, \dots, y_n)$ 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 $(e_0, e_1, \dots, e_n)$ を**評価ベクトル**と呼ぶ. ([1] 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 $eval(f)$ で表し, 式 (4.1), (4.2) 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.

$$\begin{aligned} v2e((x_1, x_2, x_3, y_1, \dots, y_n)) \\ = (w_1x_1 + w_2x_2 + w_3x_3, 0 \oplus w_4y_1, \dots, 0 \oplus w_4y_n) \end{aligned} \quad (4.4)$$

すなわち,

$$eval(f) = v2e(vec(f)). \quad (4.5)$$

$eval(f)$ の全成分の和 (実数値) を f の**評価値**または $eval(f)$ の評価値といい, $eval(f)$ で表す. 例えば, 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 を構成する全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 $vec(f_t)$, 木を構成する全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 $eval(f_t)$, 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

$$\begin{aligned} (e_0, e_1, \dots, e_n) \oplus (e'_0, e'_1, \dots, e'_n) = \\ (e_0 + e'_0, e_1 \oplus e'_1, \dots, e_n \oplus e'_n) \end{aligned} \quad (4.6)$$

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)

$$v2e(v + v') = v2e(v) \oplus v2e(v') \quad (4.7)$$

この関係式は特徴ベクトルの概念を導入していない [1] 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, 評価ベクトル同士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, 実数値同士の大小比較に帰着させる.

4.3 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 [1] 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、[1]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 $FT(h)$ の中でもっとも評価値の高い関数木（最適解）

$$f_t^* = \arg \max_{f_t \in FT(h)} eval(f_t) \quad (4.8)$$

を求めることである。

実際、[1]では、動的計画法の考え方にに基づき、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）。[1]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、4.2で述べたように、任意の特徴ベクトル v, v' に対して、

$$v2e(v + v') = v2e(v) \oplus v2e(v') \quad (4.9)$$

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[8]では、この議論は暗黙的である。その意味で、本論文は[1]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

$$e < e' \text{ ならば } e \oplus e'' < e' \oplus e'' \quad (\text{単調性}) \quad (4.10)$$

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

$$e = (-0.049, 1.0, 0) < (-0.1, 1.0, 1.0) = e', \quad (4.11)$$

上位の1引数関数ノードの評価ベクトルを

$$e'' = (-0.049, 0, 1.0) \quad (4.12)$$

とすると，

$$\begin{aligned} e \oplus e'' &= (-0.098, 1.0, 1.0) \\ &> (-0.149, 1.0, 1.0) = e' \oplus e'' \end{aligned} \quad (4.13)$$

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，[1] 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，探索の枝刈りを一切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はなく，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，[1] 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，[1] 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4～6として示す．

アルゴリズム4

アルゴリズム4は， T に属する全ての型 t について，戻り値の型が t で高さが i ($1 \leq i \leq h$) の関数木のうち，評価値が上位 r 番目までのものを $\text{goodRoots}(t, i)$ 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，異なる f に対して最大 r 個の木を保持する．[1] 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 4 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  $1 \leq i \leq h$  do
    for each  $t \in T$  do
       $goodRoots(t, i) \leftarrow \emptyset$ 
      for each  $f \in F$  such that  $ret(f) \in sub(t)$  do
         $tree \leftarrow \text{GetBestForFunc}(f, i - 1)$ 
         $e \leftarrow \text{eval}(tree)$ 
        if  $e > -\infty$  then
           $goodRoots(t, i) \leftarrow goodRoots(t, i) \cup tree$ 
        end if
      end for
      /*評価値上位  $r$  番目までの木を残す*/
       $goodRoots(t, i) \leftarrow \text{GetBestN}(goodRoots(t, i), r)$ 
    end for
  end for
```

アルゴリズム 5

アルゴリズム 5 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。[1] 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 6

アルゴリズム 6 は、型 t と高さ h が与えられたとき、 $goodRoots(t, i)$ ($1 \leq i \leq h$) 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。[1] では、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 1.6GHz) では $r = 20$ 程度、性能が高いデスクトップ PC (Core i7 3.4GHz) では $r = 150$ 程度であった。

アルゴリズム 5 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*/
 $tree_{best} \leftarrow \text{CreateFunctionTree}(f)$ 
 $e_{best} \leftarrow \text{eval}(f)$ 
for each  $ptype \in \text{params}(f)$  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*/
 $tree_{param} \leftarrow \text{null}$ 
 $e_{param} \leftarrow (-\infty, 0, 0, \dots, 0)$ 
for each  $1 \leq i \leq h_{max}$  do
  for each  $ptree \in \text{goodRoots}(ptype, i)$  do
    if  $e_{best} \oplus \text{eval}(ptree) > e_{param}$  then
       $tree_{param} \leftarrow ptree$ 
       $e_{param} \leftarrow e_{best} \oplus \text{eval}(ptree)$ 
    end if
  end for
end for
/*最良の関数木を  $tree_{best}$  の子ノードに追加*/
 $tree_{best} \leftarrow \text{AddChild}(tree_{best}, tree_{param})$ 
 $e_{best} \leftarrow e_{param}$ 
end for
return ( $tree_{best}$ )
```

アルゴリズム 6 関数木の抽出

```
procedure ExtractTreeForType( $t, h$ )
 $trees \leftarrow \emptyset$ 
for each  $1 \leq i \leq h$  do
   $trees \leftarrow trees \cup \text{goodRoots}(t, i)$ 
end for
return ( $\text{Sort}(trees)$ )
```

第5章 キーワードあいまい一致

本章では、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 [1] 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。例えば、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) [18, 19] 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) [20, 21, 19, 22] 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。例えば 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、

$$LCS1 = \frac{LCS}{n} \quad (5.1)$$

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，

$$LCS2 = \frac{2LCS}{m+n} \quad (5.2)$$

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． $LCS1 \sim LCS4$ 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．例えば 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．つぎに，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，

$$\text{削除コスト} \leq \text{置換コスト} \leq \text{挿入コスト} \quad (5.3)$$

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事を目的としていないので，ある整数 k について，

$$\text{削除コスト} = 1 \quad (5.4)$$

$$\text{置換コスト} = 1 + k \quad (5.5)$$

$$\text{挿入コスト} = 1 + 2k \quad (5.6)$$

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。

$$LD(k, C) = \frac{1}{1 + \frac{LD(k)}{C}} \quad (5.7)$$

ここで $LD(k)$ は k の値に基づき(17)～(19)式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、キーワード毎に加算値の上限を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. 例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, 全ての組み合わ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, [1] で使用されていた4つのプロジェクト (Apache Commons Codec[23], CAROL[24], jMemorize[25], CMU Sphinx[26]) と, 単体テストのためのフレームワーク JUnit[27] の合計5つのプロジェクトを用い, 19844問のタスクを作成した. 重み $w_1, \dots, w_4$ は, 4.1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.

改変無し P に含まれる関数および変数のラベルに含まれる単語全てを集めたものを Q とする.

母音削除 英単語を短縮する際の慣習に習い, 「改変無し」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全て削除したものを Q とする.

先頭3文字 「改変無し」で得た Q に含まれるキーワードのうち, 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 「改変無し」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 「改変無し」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.

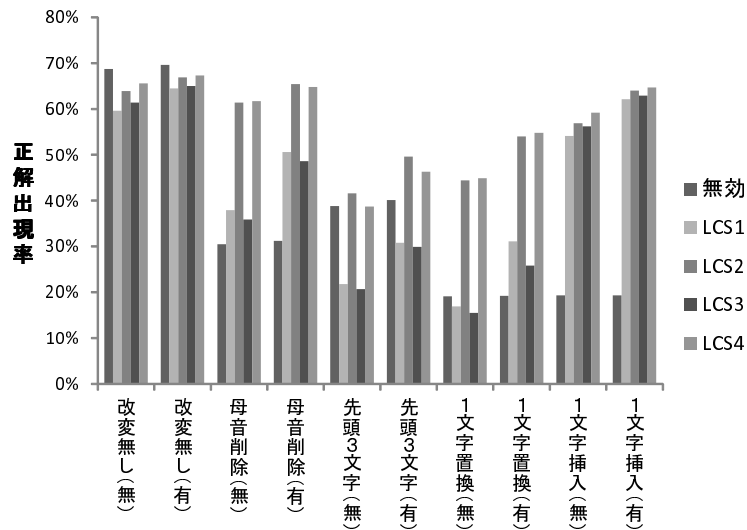


図 5.1: あいまい一致の機能が無効の時と, $LCS1 \sim LCS4$ を類似度としたあいまい一致の機能が有効な時の正解出現率を表したグラフ. 横軸は, 5通りのクエリ生成方法のそれぞれについてキーワード分割が無効(無)および有効(有)の場合を区別している.

5.6 考察

実験結果を図 5.1, 5.2, 5.3 に掲載した. 図 5.1 は, 類似度に $LCS1 \sim LCS4$ を使用した時の, 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言い換える.)

- 「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, 精度は全ての類似度の場合で 10% 以内, 特に $LCS2$ と $LCS4$ の場合は 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 $LCS2$ の時に最大で精度が約 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.)
- 全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 5.2 と図 5.3 は, 類似度に $LD(k, C)$ を使用した時の上位 $U = 10$ 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 5.2 では, k を 1 に固定し, C

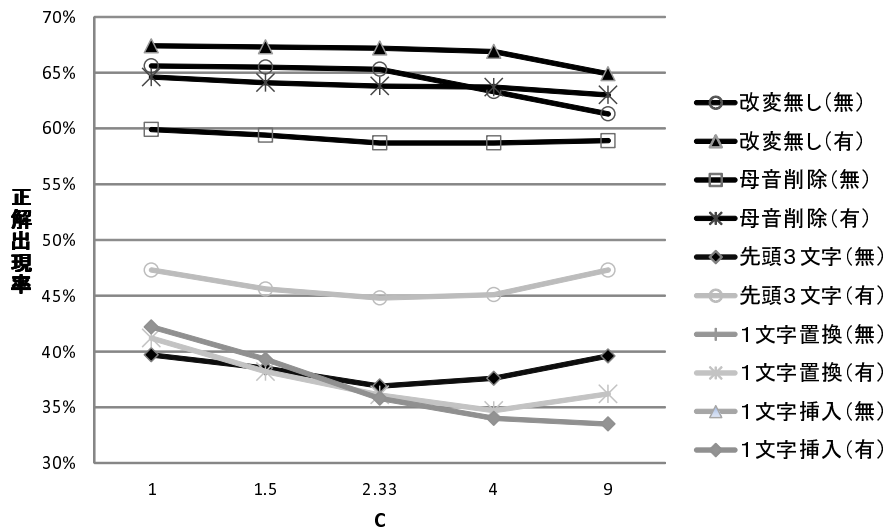


図 5.2: あいまい入力の種類にレーベンシュタイン距離を用いた場合について、整数 k の値を 1 に固定したときの正解出現率を表したグラフ。

を 1 から 9 まで動かす、図 5.3 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 5.2 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、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 5.3 の結果から、 k を動かすと、「改変無し」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無い分、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関わらず、今回の実験ではほぼ全ての場合で LCS の方が精度

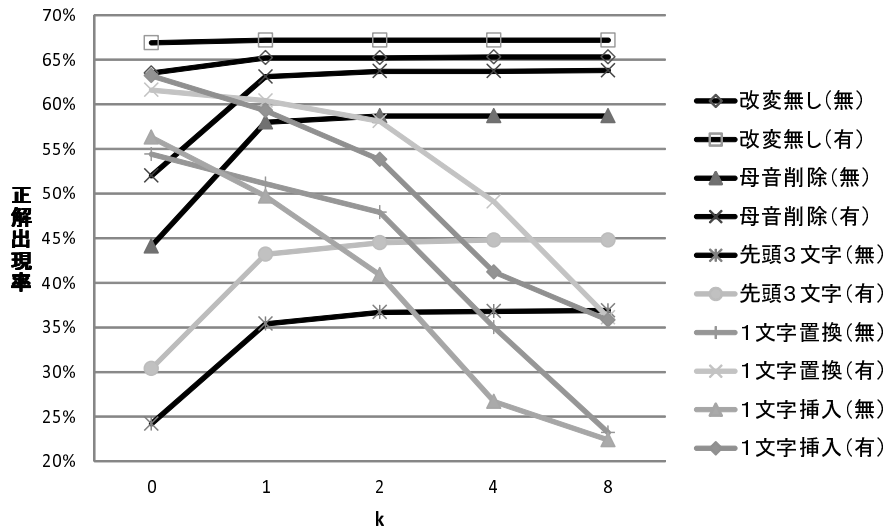


図 5.3: あいまい入力の種類にレーベンシュタイン距離を用いた場合について、実数 C の値を 2.33 に固定したときの正解出現率を表したグラフ。

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 (5.1), (5.2) には、LD の式 (5.7) 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、実用的には問題が無いと考えられる。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 100ms 程度、LD の場合 300ms 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 200ms 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 [28, 29] 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。

第6章 ツールの実装

先行研究 [1] では、ツールが提示する出力はただ 1 つであり、それは必ずしも正解（ユーザの欲しい出力）とは限らない．そのため本研究では正解が出力に含まれる可能性を大きくするため、複数の候補を出力できるように、アルゴリズムの改良を行った (図 6.1).

また、3.2 節で述べた重みの値は先行研究では定数であったが、様々なユーザの嗜好やソースコード環境に対応できるようにするため、それぞれの値を自由に動かせるように改良し、さらにその適切な値を学習させることができる複数の機能を実装した．本研究で行う機械学習は、訓練データとしてツールに入力されるクエリーと周辺ソースコード状況を用い、できるだけその正解を出力できるように特徴の重みを調整するものである．本研究ではこの訓練データと正解の組みを**タスク**と呼び、タスクをソースコードから取得する手段として、ユーザが正解として選びたいコード片をドラッグして選択し、登録ボタンを押すことにより、そのコード片から自動生成されるキーワードと周辺ソースコード状況を、正解と共にタスクとして保存できる機能を実装した (図 6.2).

保存したタスクは「学習タスク一覧ビュー」(図 6.3) に表示され、過去に追加したタスクを含めて一覧および編集を行うことができる．そして、登録された複数のタスクを一括して学習できるバッチ学習の機能を実装した．また、ユーザがタスクを 1 つ選択するたびに、それを含む過去数回分のタスクを用いて即時に学習するオンライン学習の機能もあわせて実装した．

6.1 統合開発環境 Eclipse

本研究では改良したキーワードプログラミングを統合開発環境 Eclipse のプラグインとして実現した．Eclipse は多くのプログラミング言語を使用できる統合開発環境として非常に人気が高く、そのプラグインを開発することは、多くの開発者にとって有益となる．現在、数多くの Eclipse のプラグインが世界中で開発されており、これらのプラグインを組み合わせることにより、開発者の好きなように Eclipse IDE に機能を追加することが可能になっている．実は Eclipse の本体自体も、最下層に位置するランタイム以外は全てプラグインであり、それらが組み合わさって構成されている．例えば図 6.4 において SWT と JFace はユーザ・インターフェースを提供するためのプラグインである．

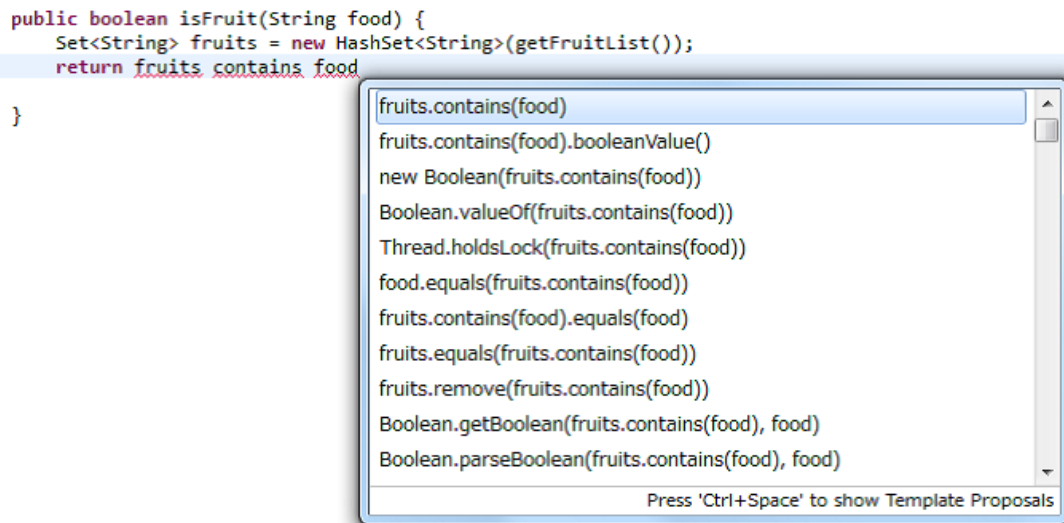


図 6.1: 実際に作成したツールを起動し、ポップアップを表示させたときのスナップショット. このツールはポップアップに表示された複数の候補からユーザの欲しいものを選択することができる. この図はユーザの欲しい出力が `fruits.contains(food)` である時に、クエリーとしてその欲しい出力に存在する全てのラベル `fruits`, `contains`, `food` を入力したときのスナップショットである.

次に Eclipse のワークベンチ (図 6.5) と、その各部の名称について簡単に説明する. Eclipse ではユーザ・インターフェース全般のことをワークベンチとよび、以下のような構成となっている.

ワークベンチウィンドウ

Eclipse を起動したら表示されるウィンドウのこと. 1 つのワークベンチが複数のウィンドウを表示することも可能である.

ワークベンチページ

ワークベンチで開いているパースペクティブ 1 つに対して 1 つのページが対応する. パースペクティブとはビューやエディターのレイアウトを特定用途に合わせて使いやすくした物で、複数のビューやエディターを含むことが出来る.

メニューバー

メインメニューを表示する領域である.

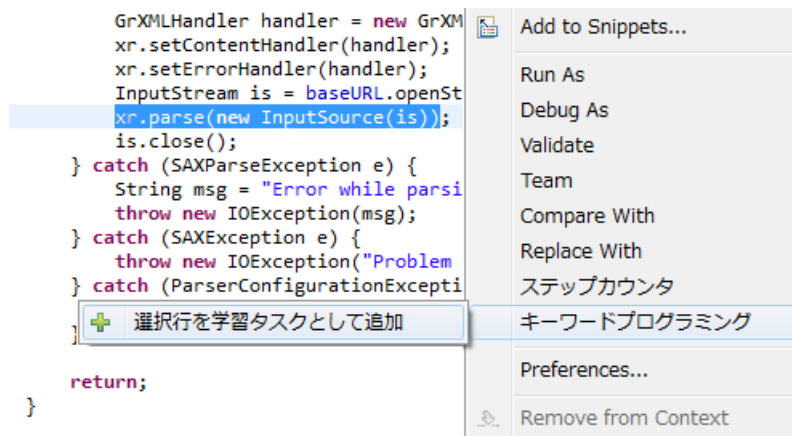


図 6.2: 本研究で実装したツールでは、ツールに学習させたいタスク (入力と出力の組み合わせ。出力はユーザがドラッグして選択するコード片。入力はそのコード片から自動生成されるキーワードと周辺ソースコード状況) を選択して追加することができる。この図は `xr.parse(new InputSource(is))` というコード片を正解とするタスクを追加する作業を表している。

ツールバー

ツールバーを表示する領域である。

ステータスライン

ウィンドウ最下部に表示される領域である。エディターのキャレット位置、ビルドの進捗状況など様々な情報を表示する。

ビュー

ソフトウェアの階層構造や、コンソール画面など様々な情報を提供する領域である。

エディター

主にソースファイルの編集に使用する領域である。

6.2 Eclipse プラグイン開発

Eclipse のプラグインを開発するに当たり、その開発をサポートするプラグインである PDE(Plugin Development Environment) が eclipse.org から提供されてい

所属クラス名	オフセット	出力文字列	入力キーワード
com.sayuu.Element	102	Math.abs(x)	math abs x
jp.sayuu.File	566	BufferedReader(filename)	buffered reader filename
ac.hokudai.Recognition	24	setDaemon(true)	set daemon true

入力キーワードは出力文字列が大文字の部分で分割されて自動的に生成される(後からユーザーが編集することが可能).

図 6.3: 図 6.2 で示される作業によって追加したタスクを一覧することができるビュー. 1 行が 1 つのタスクを表す. ある 1 つの出力に対応するクエリーはその出力関数木に含まれる全ての関数および変数のラベルとして自動的にセルに入るが、ユーザが後から編集することも可能となっている.

る. 通常, 1 つのプラグインは, 1 つの MANIFEST.MF と 1 つの plugin.xml という 2 つのマニフェスト・ファイルを含んでいる. PDE の最大の機能はこれら 2 つのマニフェストを記述するためのマニフェスト・エディターを備えている事である. マニフェスト・エディターを使用することにより, より簡便にプラグインを作成することが出来る. MANIFEST.MF には, プラグインの ID, 名称, 作成者, 他のプラグインとの依存関係などが記述されている. また, plugin.xml はプラグイン・マニフェスト・ファイルと呼ばれ, プラグインの拡張と, 拡張ポイントの定義などが記述されている.「拡張ポイント」とは新たなプラグインを既存のプラグインに繋ぐ場所のことであり, 新たなプラグインを作成する時は, 既存のプラグインの, どの拡張ポイントに接続するかを指定する必要がある. Eclipse はその基本部分もプラグインの集合体であるため, 初期状態の Eclipse にプラグインを追加する時を含め,『新たなプラグインを Eclipse に追加する時には必ず, 既存のプラグインの拡張ポイントに繋ぐ』という形をとる. プラグイン開発者は新しく開発するプラグインにどのような機能を持たせるかによって, どの既存プラグインの拡張ポイントを選択するかを考える必要がある. また, プラグインの動作部分の記述は Java クラスファイルとして記述しなければならない.

Help	Update	Text	IDE Text	Compare	Debug	Search	Team/ CVS
			IDE				
UI (Genetic WorkBench)					Resources		
JFace							
SWT							
Runtime (OSGi)							

図 6.4: Eclipse プラットホーム

6.3 ツールの設計

本研究で作成したツールの設計について解説する．本研究では Eclipse のプラグインとして実装することを前提にツールの設計を行ったため，Eclipse の API やプラグインの機構に沿ったものとなった．概略図は図 6.6 である．キーワードプログラミングの研究自体が，Eclipse のコンテンツ・アシスト機能（入力補完機能とも呼ばれる．図 6.7）を拡張することを想定した研究となっているので，そのコンテンツ・アシスト機能を中心に設計を行った．概略図（図 6.6）の番号に沿って解説していく．

- 1 はじめにユーザがエディターにキーワードを入力し，それを取得する部分であるが，これは，まずエディタ上のキャレット位置を把握し，その 1 行を取得，キーワードの部分以外を切り落とす，と言った順でなされている．また，ソースコードに記述されているそのキャレット位置において期待される返り値の型，ローカル変数，フィールド定義，メソッド定義，インポート文などの情報の取得には EclipseJDT に用意されている AST(Abstract Syntax Tree) を使用した．
- 2 キーワードプログラミングのアルゴリズムに対して 1 で取得したキーワードとソースコード情報を入力する．
- 3, 4, 5 ProposalSorter がキーワードプログラミングのアルゴリズムによって生成された補完候補のソーティングを、『選択候補履歴データベース』から得られる選択履歴を加味して行い，CompletionProposalComputer へ渡す．
- 6, 7 CompletionProposalComputer がキャレット位置に補完候補を提示し，ユーザ

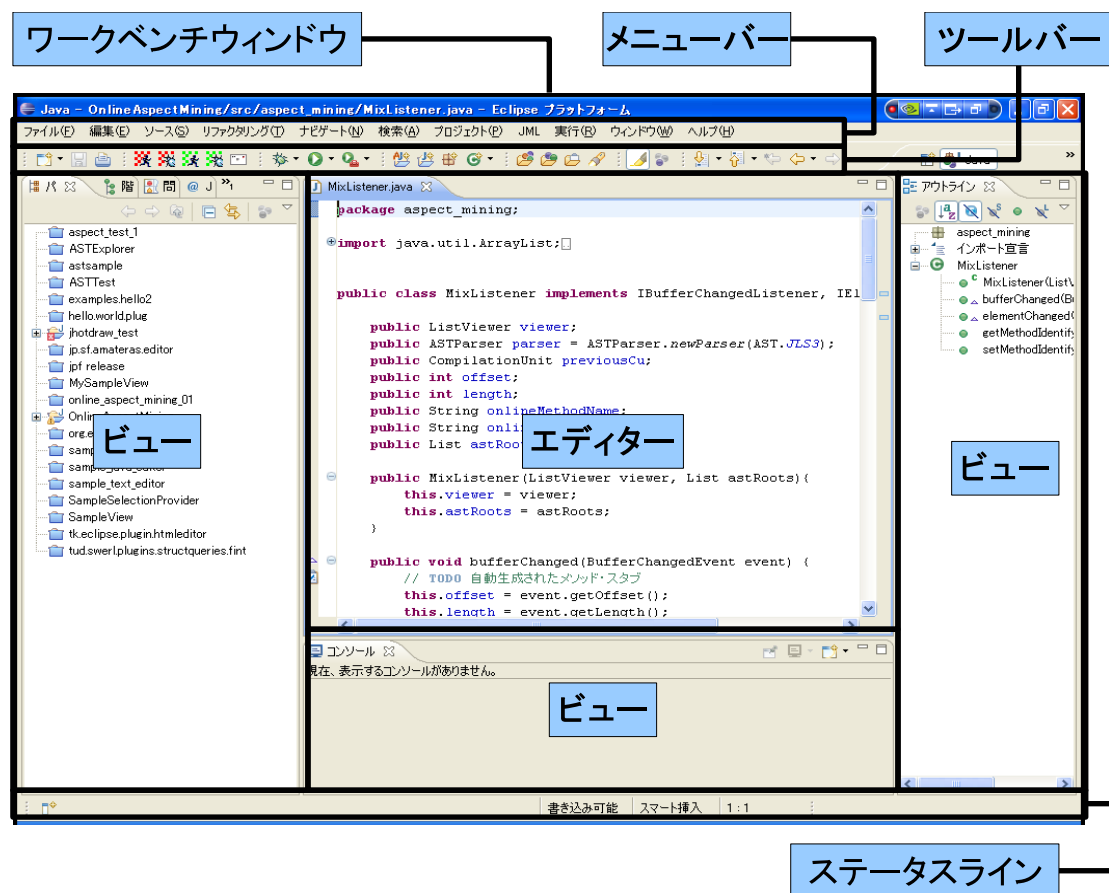


図 6.5: Eclipse ワークベンチ

がその候補の中から好きな候補を選択する。もし候補が選択されれば、CompletionProposalComputer がエディタ上にその文字列を出力し、選択候補履歴 DB に選択された候補文字列と、その時の期待されるされる返り値の型と選択回数を記録する。

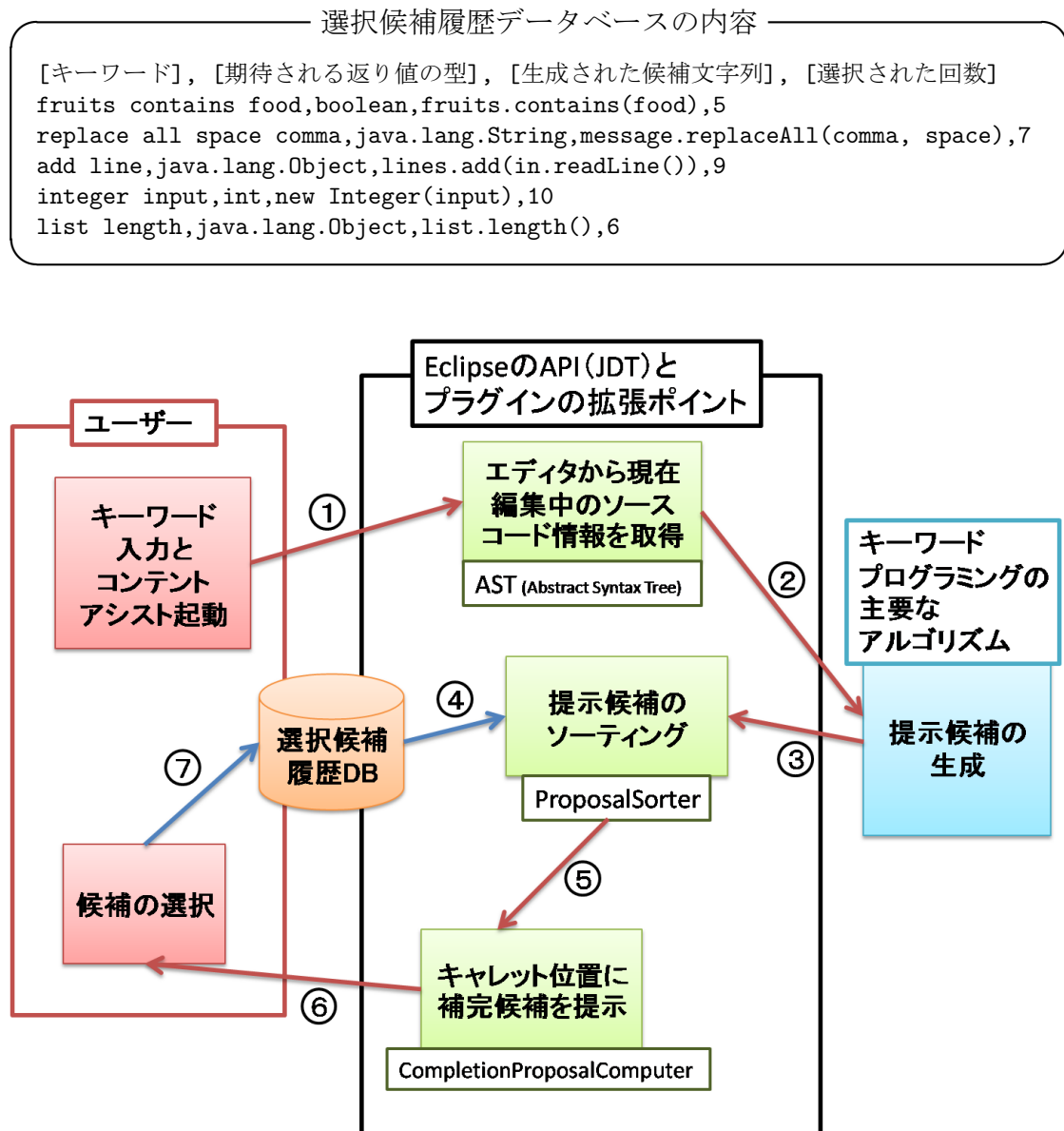


図 6.6: 作成したツールの設計概略図

6.3.1 ツールの使用方法

本研究で開発したキーワードプログラミングのプラグインを使用するためには、プラグインをインストール後に、提示候補のソートングを有効にする必要があ

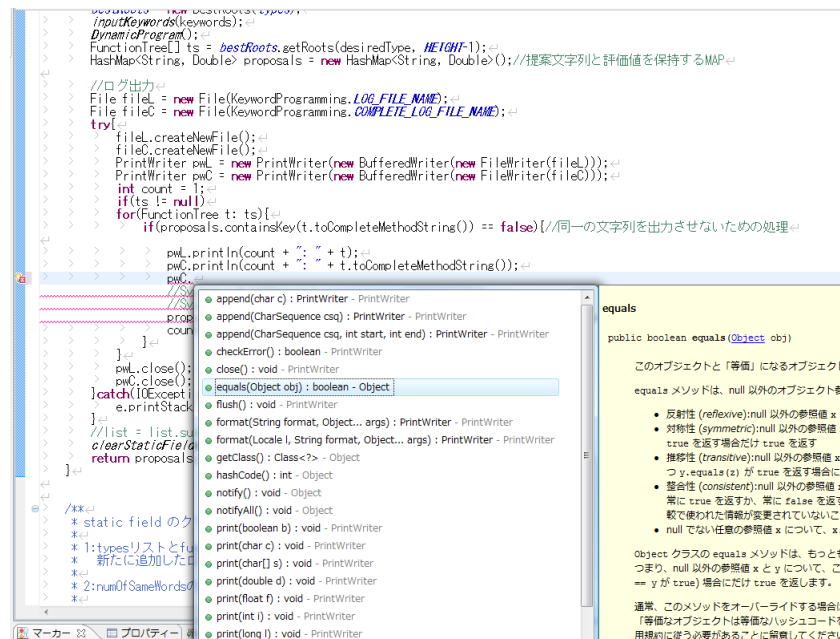


図 6.7: Eclipse のコンテンツ・アシスト機能を起動したところ. JDIT のエディタ上では, オブジェクト名の後に '.' (ピリオド) を打つとその後に記述可能なフィールドやメソッドの候補が表示される. 任意の位置でコンテンツ・アシスト機能を使用したいときは『Ctrl+Space』の入力で起動できる.

る. 「Eclipse のメニュー」 → 「ウィンドウ」 → 「設定」 → 「Java」 → 「エディター」 → 「コンテンツ・アシスト」 → 「ソートおよびフィルター」 → 「プロポーザルをソート」 の設定を「キーワードプログラミング」に変更する (図 6.8) .

6.4 ローカルサーチによる特徴の重みの学習

本研究では, 機械学習に用いる探索の方法として単純な探索方法であるローカルサーチ (山登り法) を選択する. ローカルサーチとは, 適当な初期解から出発し, 探索空間内において, 現在の解の近傍のうちで最も目的関数の値 (本論文では, スコアと呼ぶ) が高い解が現在の解よりもスコアが高ければ, それを現在の解に置き換えるという操作を反復し, 解の入れ替えが行われなくなるか, ステップが指定した回数に達した場合に終了する方法である. 本研究の場合, 解とは 4 つの重みの値からなる重みベクトル (w_1, w_2, w_3, w_4) であり, その近傍とは, 重みをそれぞれ $+step$, 0 (動かさない), $-step$ だけ加算して得られる重みベクトルと定義する. 重みの数は全部で 4 つあるので, 近傍の合計は $3^4 - 1 = 80$ 個となる ($step$ の値はユーザが任意に指定できる). 実際にユーザにツールを提供する際は, 予め提供者が適当な重みの組み合わせを初期解として設定しておくことにする.

アルゴリズム 7 ローカルサーチのアルゴリズム

```
procedure LocalSearch( $p_{current}$ ,  $tasks$ )  
   $taskTrees \leftarrow \emptyset$   
  for each  $task \in tasks$  do  
     $trees \leftarrow \text{KeywordProgramming}(task)$   
     $taskTrees \leftarrow taskTrees \cup (task, trees)$   
  end for  
   $score_{best} \leftarrow \text{Score}(p_{current}, taskTrees)$   
   $p_{best} \leftarrow p_{current}$   
   $peak \leftarrow \text{false}$   
  while not  $peak$  do  
     $peak \leftarrow \text{true}$   
     $p_{current} \leftarrow p_{best}$   
    for each  $p_n \in neighborhood(p_{current})$  do  
       $score_n \leftarrow \text{Score}(p_n, taskTrees)$   
      if  $score_n > score_{best}$  then  
         $score_{best} \leftarrow score_n$   
         $p_{best} \leftarrow p_n$   
         $peak \leftarrow \text{false}$   
      end if  
    end for  
  end while  
  return ( $p_{best}$ )
```

アルゴリズム 8 重みベクトル p の下でのスコアを求めるアルゴリズム

```
procedure Score( $p, taskTrees$ )
   $score \leftarrow 0$ 
  for each ( $task, trees$ )  $\in taskTrees$  do
     $eList \leftarrow \emptyset$  /*説明ベクトルを入れるリスト*/
    for each  $tree \in trees$  do
       $fvec \leftarrow tree.getFeatureVector()$ 
       $eList \leftarrow eList \cup (CalcExpl(p, fvec), tree)$ 
    end for
    /*ペアの第一要素の大きさ順に  $eList$  をソートする*/
     $eList' \leftarrow Sort(eList)$ 
    /*スコア (正解の出力の順位の逆数) を得る*/
     $score \leftarrow score +$ 
       $1/GetDesiredOutputOrder(task, eList')$ 
  end for
  return ( $score$ )
```

ローカルサーチは現在の重みベクトルから少しだけ一部の重みを変化させるだけなので、重み変更の前後においてアルゴリズム 4, 5, 6 によって生成される出力候補群に著しく大きな変動はないと考えられる。本研究ではこの仮定の下、80 通りある近傍のスコアを算出する際に重みベクトルを変更する度にアルゴリズム 4, 5, 6 を毎回行うのではなく、ローカルサーチを実行する直前の重みベクトルに対して 1 度だけアルゴリズム 4, 5, 6 を実行し、十分な数の出力候補群を生成しておき、それらを近傍の重みベクトルの下で評価することにより正解の順位を算出し、各近傍のスコアを計算する事により計算時間を短縮することにした。

ここで、3.3 節で定義した特徴ベクトルおよび説明ベクトルとの関係で説明を加える。先行研究では、重みが固定されているため、全ノードの「説明ベクトル」を基礎として、その和により関数木の説明ベクトルを計算している。それに対し本研究におけるローカルサーチでは、様々な重みの下で説明ベクトルを効率良く計算する必要があるため、全ノードの「特徴ベクトル」を基礎として、その和により関数木の「特徴ベクトル」 $(x_1, x_2, x_3, y_1, \dots, y_n)$ を計算し、これを保存しておき、その後に様々な重みの下で、説明ベクトルを $((w_1x_1 + w_2x_2 + w_3x_3), \min(w_4y_1, 1), \dots, \min(w_4y_n, 1))$ により計算する。

ここまで説明してきたローカルサーチの処理時間短縮手法の大まかな処理の流れを図 6.9 に表した。

次に、ローカルサーチを実行するアルゴリズム 7 と、ある重みベクトルの下でのスコアを算出するアルゴリズム 8 の説明を行う。

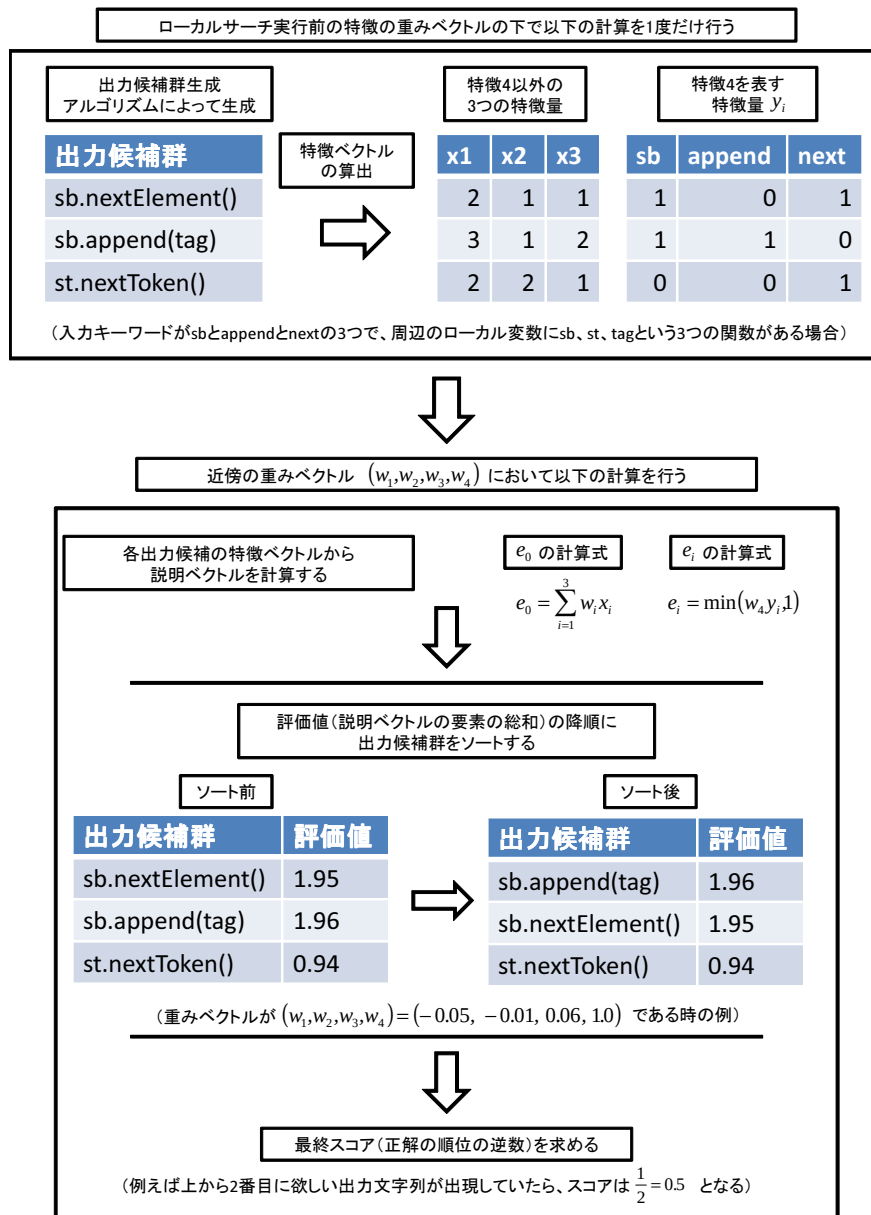


図 6.9: ローカルサーチの処理時間短縮手法の処理の流れを表した図。ローカルサーチの前処理として、アルゴリズム 4, 5, 6 によって生成された出力候補群から各候補の特徴ベクトル $(x_1, x_2, x_3, y_1, \dots, y_n)$ を算出しておく。そして、その特徴ベクトルと近傍の重みベクトル (w_1, w_2, w_3, w_4) から、その近傍における評価値(説明ベクトルの要素の値の総和)を計算する。次に、その評価値の降順に出力候補群をソートする。最後に、正解の順位の逆数としてスコアを求める。この手法の特徴は、はじめに一度だけアルゴリズム 4, 5, 6 を実行し特徴ベクトルを抽出しておけば、近傍についてはそれらを実行せずに評価値を計算することが可能な点である。

アルゴリズム 7

アルゴリズム 7 は、入力として重みベクトル $p_{current}$ と、タスクのリスト $tasks$ を取り、ローカルサーチを行い、最適な重みベクトルを返すアルゴリズムである。関数 `KeywordProgramming` は 1 つのタスクを引数に取り、タスクに含まれるクエリーと周辺ソースコード状況を用いて、アルゴリズム 4, 5, 6 を内部で実行し、関数木の集合 $trees$ を出力する関数である。

ローカルサーチが終了するのは、現在の重みベクトルのいずれの近傍においてもスコアが改善されないときである。ただし、現実の実装では、反復回数が所定の上限に達したときにも終了するようにしてある。

アルゴリズム 8

アルゴリズム 8 は、入力として重みベクトル p と、タスク ($task$ とする) およびそれに対応する出力候補の関数木の集合 ($trees$ とする) の組 ($task, trees$) の集合 $taskTrees$ を引数に取り、出力として全タスクの合計スコア $score$ を計算するアルゴリズムである。関数 `getFeatureVector` は関数木が保持している特徴ベクトルを取得するメソッドである。関数 `CalcExpl` はその引数に、重みベクトル p と特徴ベクトル $fvec$ を取り、新しい説明ベクトルを作成して返す関数である。関数 `GetDesiredOutputOrder` は引数に、正解の関数木を含むタスクと、評価値でソート済みのリスト $eList'$ を取り、正解の関数木の順位を出力する関数である。

6.6 グリッドサーチ

グリッドサーチとは、指定した範囲内でパラメーターの値の組み合わせを一定の値ずつ規則的に変化させていき、一番正解率の良い値の組み合わせを最適なパラメータの値として出力する探索方法である。本研究におけるローカルサーチによる学習では、正解の順位を上げるような重みベクトルを探しているため、1 つのタスクについて、初めに 1 度だけ行うキーワードプログラミングにより生成された出力候補群の中に正解が出現しない時はそのタスクについて学習を行うことができない。そのような時、グリッドサーチの機能を利用し、与えられたタスク群に対してスコアが最大となる重みベクトルをより広い範囲から探索することができる。

6.7 実装上の工夫

本研究では Java 用のキーワードプログラミングのツールを Eclipse のプラグインとして実装した。Java で記述してあり、ソースコードの大きさは約 8000 行である。実装上いくつかの工夫を行ったのでその説明を行う。

工夫の1つ目は、型や関数の探索範囲を、十分に実用的な範囲まで拡大したことである。型の集合 T に含まれる型は、Java 言語の仕様上必ず非明示的にインポートされる `java.lang` パッケージで定義されている型に加えて、現在編集中のソースファイル上で明示的にインポートされているパッケージ群の内部で定義されている型、現在編集中のクラスが所属するパッケージで定義されている型、および現在編集中のクラスと定義している。また総称型については、現在編集中のクラスで定義されている関数および変数が、総称型の返り値の型を持っている時に限り、 T に追加することにした。また、 F としては、 T に含まれる型の中で定義されている関数および変数全てと、ユーザが『Ctrl+Space』を押してツールを実行したキャレットの位置から利用できる全てのローカル変数と定義している。これらの型と関数および変数の情報を『Ctrl+Space』を押してツールを実行するたびにメモリに読み込むと非常に時間がかかるので、ソースコードの状況が変わっても再読み込みする必要のないパッケージはツール実行以前に予め読み込んでおくことによって、処理の高速化を行った。具体的には、`java.lang` パッケージ中のクラスは Eclipse が起動されるのと同時にメモリに読み込まれ、Eclipse が終了されるまで保持される。また、その他のクラス群は、ソースファイルが選択された時点でメモリに読み込まれ、その後別のソースファイルが選択された時点で破棄される (図 6.10)。

工夫の2つ目は、ツールが出力するコード片の望ましい返り値の型の取得である。ソースコード上のどの場所で『Ctrl+Space』を押してツールを起動するかによって、その周辺状況によって出力コード片の望ましい返り値の型が定まるため、その型を Eclipse の抽象構文木を扱う API を用いて得ている。起動する場所の状況パターンは大きく 4 つあり、クエリーと同じ行の左方向に何もない場合、変数宣言または代入文の左辺及び代入演算子 (=) が左側にある場合、メソッドの引数の中で使用した場合、`return` 文の中で使用した場合である。このそれぞれの場合について説明を行う。各状況の具体例を図 6.11 に示した。

クエリーと同じ行の左方向に何もない場合

クエリーと同じ行の左方向に何もない場合は、周辺のソースコードの状況から期待される型に特に制約が無いので、全ての型のスーパークラスである型を特別に用意し、出力コード片の望ましい返り値の型とした。この場合の具体例として図 6.11 の1番上の図 (Task10) を示す。この図はユーザが欲しい出力として `buf.append(s)` を期待していて、クエリーとして **buf append s** を入力し、ツールを起動する直前の状況を表している。ユーザが入力したクエリーと同じ行の左方向には何の記述もないという状況である。

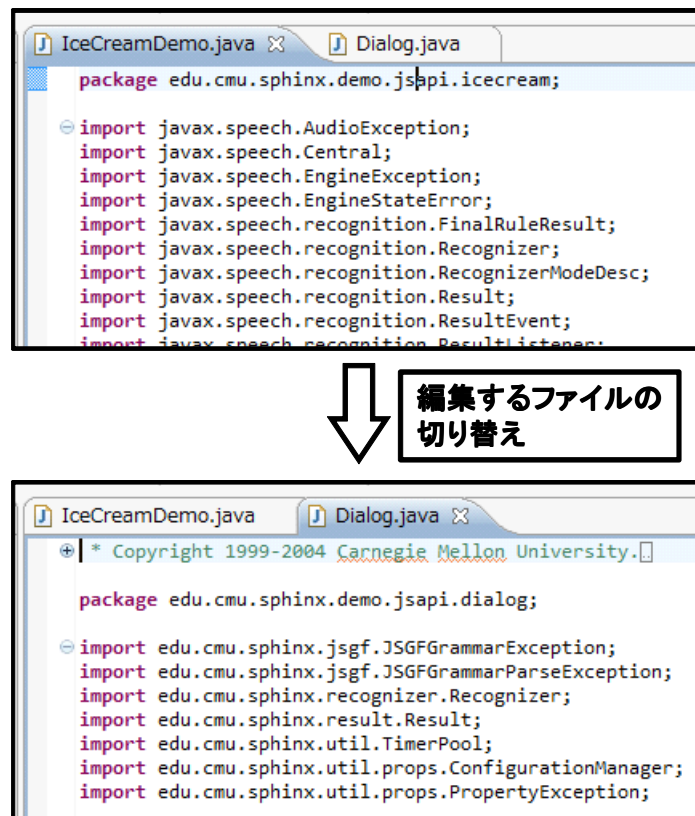


図 6.10: Eclipse のエディタ上で編集するソースファイルの切り替え前後の状態を表した図. 初めに IceCreamDemo.java が選択されていて, 次にユーザが Dialog.java にファイルを切り替えた所である (タブの色が変化している). この際, メモリ上に保持されていた IceCreamDemo.java 中でインポートされたクラス群の情報は破棄され, Dialog.java 中でインポートされているクラス群の情報が読み込まれる.

変数宣言または代入文の左辺及び代入演算子 (=) が左側にある場合

クエリーと同じ行の左方向に変数宣言または代入文がある場合は, その宣言または代入される変数の型が出力コード片の望ましい戻り値の型となる. この場合の具体例として図 6.11 の上から 2 番目の図 (Task08) を示す. この図はユーザが欲しい出力として `message.charAt(i)` を期待しており, ツールに入力するクエリーとして **message at i** を入力し, ツールを起動する直前の状況を表している. ユーザが入力したクエリーの前には `char c =` というローカル変数の宣言文がある. この場合, 出力コード片の望ましい戻り値の型は `char` となる.

メソッドの引数の中で使用した場合

メソッドの引数の中でクエリーを入力しツールを起動した場合は、メソッドの引数の型が出力コード片の望ましい返り値の型となる。この場合の具体例として図 6.11 の上から 3 番目の図 (Task02) を示す。この図はユーザが欲しい出力として `list.size()` を期待していて、`remove` メソッドの本来引数を入力するべき括弧の中に、クエリーとして **list size** を入力し、ツールを起動する直前の状況を表している。この場合、出力コード片の望ましい返り値の型は `remove` メソッドの引数の型である `java.lang.Object` となる。

return 文の中で使用した場合

`return` 文の中で使用した場合は、編集集中のメソッドの返り値の型が出力コード片の望ましい返り値の型となる。この場合の具体例として図 6.11 の一番下の図 (Task06) を示す。この図はユーザが欲しい出力として `Math.abs(x)` を期待していて、クエリーとして **math abs x** を入力し、ツールを起動する直前の状況を表している。ユーザが入力したクエリーと同じ行の左方向に `return` 文がある。この場合、出力コード片の望ましい返り値の型は、編集集中のメソッド `absoluteValue` の返り値の型である `int` となる。

6.8 実験と考察

本章では、3つのリサーチ・クエスチョンを設定し、それぞれのクエスチョンに対する実験と考察を述べる。リサーチ・クエスチョンは以下の通りである。

1. 特徴の重みの変化に応じて正解率はどのように変化するのか。重みの組み合わせを調整することによって正解率を向上させることができるのか。
2. キーワードプログラミングは、オープンソースプロジェクトなどの実プロジェクトにおいては、どの程度の正解率を達成できるのか。
3. ローカルサーチに実装した単純な評価基準（正解が出現した順位の逆数の総和の最大化）はどのような特性を持つか。

6.8.1 実験 1

特徴の重みの変化に応じて正解率はどのように変化するのかという事と、重みの組み合わせを調整することによってツールの正解率を向上させることができるのかを調査するために、実験 1 を行った。実験に用いるタスクとして、先行研究 [1] の著者が作成し公開していた 14 間のタスクを用いた。この 14 間のタスクは、重

みが先行研究が定義したものと等しい場合では、14問中12問、正解が出力中に出現するものである。

重みの変化に応じて正解率がどのように変化するかをグリッドサーチを用いて調べたところ、図 6.12 の結果が得られた。この実験では、各タスクのクエリーとして、正解コード片に含まれる関数および変数のラベルに含まれる単語全てを用いた。例えば、図 6.1 のスナップショットのように、正解が `fruits.contains(food)` である 1 つのタスクに対して、そのクエリーとしては、正解に存在する全ての単語 `fruits`, `contains`, `food` をツールに入力し、ツールが正解を上位 10 件までに出力していれば、その順位の逆数を正解率と定義して、スコアに加算した。ただし、アルゴリズム 4 の r の値を 250 とし、特徴 4 の重み w_4 を 1 に固定した。全 14 タスクのスコアの合計を 3 次元グラフの各交点にプロットして得たのが図 6.12 である。図の X, Y, Z 座標はそれぞれ特徴 1, 2, 3 を表す重み w_1, w_2, w_3 であり、それぞれ 0.0 から ± 5.0 の間を 0.5 刻みで絶対値を動かした。重みの変化に応じた正解率の変化を視覚的に認識することができる。また、重みを調整することによって、正解率が幅広く変化することがわかる。さらに、正解率が高い周辺を一層細かな刻みで探索することにより、14 問全問正解する重みの組み合わせ (例えば $(-1.1, -0.6, 0.31, 1.0)$) を発見することもできた。

6.8.2 実験 1 の考察

実験 1 ではグリッドサーチを用いて、重みの組み合わせを調整することによって正解率の分布を調べることができた。その結果を用いると、与えられたタスク群に対して、正解率の高い重みの組み合わせを求めることができる。

ただし、グリッドサーチを用いれば網羅的な探索が可能である一方、探索に時間が非常にかかる事や、刻み幅をどの値に設定するか of 基準が無いなどの問題点があることが分かった。この実験では、たった 14 問のタスクについて図 6.12 を作成するために、ノート PC (Core i5 1.6GHz) で 2 時間以上、デスクトップ PC (Core i7 3.4GHz) でも約 1 時間かかった。数百～数千問などの大量のタスクについてグリッドサーチを行うことは実用的ではないと考えられる。今回の実験のように全探索空間にわたって最適解を探すのではなく、キーワードプログラミング分野のヒューリスティックスを開発し、それを用いた何らかのヒューリスティック探索の手法により、全探索空間を探索せずに準最適解を求めるようなアプローチが今後の研究課題となる。例えば、複雑系に関わる研究分野では、遺伝的アルゴリズム (GA) などが、実用的に広く用いられている。

6.8.3 実験 2

キーワードプログラミングは、オープンソースプロジェクトなどの実プロジェクトにおいては、どの程度の正解率を達成できるのかを調査するために、実験 2 を

表 6.1: 先行研究 [1] の著者が自作した 14 問のタスクとオープンソースプロジェクトから適当に選んだ 455 問のタスク (OSP タスク) のデータの平均値を比較した表. 総タスク数と正解が出現したタスク数以外は 1 タスク当たりの平均値である.

実験タイトル	総タスク数	正解が出現したタスク数	入力キーワード数	探索範囲の型の総数	インポートされた型の数	探索範囲の関数と変数の総数	インポートされた関数と変数の数
先行研究タスク	14	13	4.1	112.1	9.1	3032.9	45.1
OSP タスク	455	334	4.3	140.0	37.0	3915.2	925.2

行った. 先行研究において定義されている特徴の重み ($-0.05, -0.01, 0.001, 1.0$) は, 実験 1 で述べたように, 先行研究の著者が自作し公開していた学習タスク全 14 問のうち 12 問に正解することができる重みの組み合わせである. この組み合わせの下で, 他のタスク群をテストデータとして用いた場合の結果はどのようなものになるかを調べた. テストデータとして, 先行研究の実験でも使用されている Java のオープンソースプロジェクト Sphinx[26] から適当なタスクを 455 問 (パッケージ数 11, クラス数 24) 選択して先行研究が定義したものと等しい特徴の重みでキーワードプログラミングを実行した (実験 1 と同様に, クエリーは正解出力コード片に含まれる関数および変数のラベルに含まれる単語全てとした. また, アルゴリズム 4 の r の値は 100 とした). すると, 正解が出力中に出現したタスク数は 455 問中 334 問 (正解率 73.4%) であった. 結果を表 6.1 に示した. 以下に具体的なタスクの正解出力の例を示す.

- `Toolkit.getDefaultToolkit().getImage(url)`
- `synthesizer.getSynthesizerProperties().setVoice(voice)`
- `getNormalizedWeights(ruleAlternatives.getWeights())`

6.8.4 実験 2 の考察

同じ重みの組み合わせを使用しているのにも関わらず, 先行研究の 14 問のタスク (先行研究タスク) では 14 問中 12 問正解 (正解率 85.7%) であるのに対し, オープンソースプロジェクトから作成したタスク (OSP タスク) では 455 問中 334 問 (正解率 73.4%) であった. 両者のタスク群と正解率の違いについて, さらに調査と考察を行った.

調査を行う前は, 先行研究の実験結果においてキーワード数が少ないほど精度が高い事が示されていたので, 我々の選んだタスクの方が入力キーワード数が多い事が原因であろうと考えていた. しかし, OSP タスクのキーワード数の平均は 4.3 個であり, 先行研究タスクのキーワード数の平均は 4.1 個と大きな違いは見られないので他に原因があると考えた. 他に考えられる原因としては, 標準ライブラリに加

えてインポートされる関数および変数の数の違いがある．インポートされる関数および変数とは，6.7節で説明した関数および変数の集合 F のうち `java.lang` パッケージの中で定義されているもの以外の全てである．ちなみに `java.lang` パッケージ全体で型の総数 103，関数および変数の総数 2990 である．OSP タスクの探索範囲の型の総数の平均は 140.0(内，インポート分 37.0)，関数および変数の総数の平均は 3915.2(内，インポート分 925.2 であり，先行研究タスクの探索範囲では，型の総数の平均が 112.1(内，インポート分 9.1)，関数および変数の総数の平均 3032.9(内，インポート分 45.1) であった．型の総数を比較すると大差無いが，関数および変数の数は OSP タスクの方がインポートされる数が非常に大きくなっていることが分かった．インポートする関数，変数，および型を事前にできるだけ減らす方法や，インポートしたオブジェクトを探索の対象とする優先度をうまく制御して平均的な計算時間を削減することなどが今後の研究課題となる．

6.8.5 実験 3

ローカルサーチに実装した単純な評価基準（正解が出現した順位の逆数の総和の最大化）はどのような特性を持つかを調査するための実験 3 を行った．実験 2 で得られた結果をより詳しく分析したところ，OSP タスク全 455 問のうち 334 問の出力中に正解が出現したが，これらのタスク 334 問のうち，正解の順位が出力中において第 1 位であったタスクは 329 問であった．これは正解が出現したタスク 334 問のうちの 98.5% を占める．

6.8.6 実験 3 の考察

本研究におけるローカルサーチは，正解の順位を上げるように特徴の重みを調節するものなので，まず出力中に正解が出現しなければ，効果を発揮することができない．また，正解の順位が 1 位であれば，それ以上順位を上げることはできないので，適用する意味はない．実験結果では，正解が出力中に出現するタスクのうちの 98.5% は，正解が順位第 1 位に出現するものであった．つまり，ローカルサーチが効果を発揮できる状況である，正解が順位 2 位以下に出現したタスクはたった 2.9% しか存在しないことが分かった．今後はもっとローカルサーチに有効なスコアリング手法の開発が研究課題となる．たとえば，6.8.7 で述べるように，出力が正確に一致しなくても何らかの「距離」に基づいて正解への近さを評価することなどが考えられる．

6.8.7 その他の考察と考察のまとめ

先行研究の実験であっても，そのやり方を踏襲した今回の実験であっても，正解に完全に一致した出力でないかぎり正解とはせずスコアに反映してはいない．し

かし実用面から言えば、正解に近い出力が得られれば、後はそれをユーザが手で修正すれば良いと考えられる。例えば、ある出力と正解との（何らかの意味での）「距離」を定義し、その距離の近い出力も含んで評価できるようなスコアリング手法を取り入れることも今後の課題として考えられる。

また、先行研究では、クエリーとして、正解に含まれる単語全てを含めずに、キーワード数を少なくした場合の実験も行っている。その実験結果を見ると元のキーワード数が大きいものはキーワード数を減らしてもそれほど正解率に変化は無く、逆に元のキーワード数が少ないものは正解率が大きく悪化する結果となっている。例えば元のキーワード数6のタスクをキーワード数5で行ったとしてもあまり変化は無いが(0.75 から 0.704)、元のキーワード数2のタスクをキーワード数1で行うと0.964から0.348というように急激に精度が悪化する。ここでは我々が普段 Web 検索エンジンを使うときに経験するものと同様の現象が発生していると考えられるため、先に述べたスコアリング手法と共に情報検索の知見を取り入れて研究を進めていく必要があると考えている。

今回の実験によって得られた考察を下記にまとめた。

1. 型と関数および変数の総数、その内でもインポートされたものの数が出力の正解率に影響を及ぼしている可能性があり、今後の調査が必要である。
2. グリッドサーチは、重みに依存した正解率の全体的な分布を把握するのに有効であるが、処理時間の短縮の工夫が必要である。
3. ローカルサーチを適用して有効な結果を得るには、適切なスコアリング手法を今後開発する必要がある。

6.9 関連研究との比較実験

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 [2, 3] 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無く、ツールが起動された場所よりも前に既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 (図 6.13)。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つとして例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 などの変数名は予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時に有用である。

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。例えば、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時に、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 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[30] 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。

```

public class Task10 {
    public String repeatString(String s, int thisManyTimes) {
        StringBuffer buf = new StringBuffer();
        for (int i = 0; i < thisManyTimes; i++) {
            buf.append s
        }
        return buf.toString();
    }
}

```

```

public class Task08 {
    public void test(){
        // count the a's
        String message = "how many a's are in this message?";
        int count = 0;
        for (int i = 0; i < message.length(); i++) {
            char c = message.at i
            if (c == 'a') {
                count++;
            }
        }
    }
}

```

```

public class Task02 {

    public void trimSize(List list, int noMoreThan) {
        while (list.size() > noMoreThan) {
            list.remove( list.size );
        }
    }
}

```

```

public class Task06 {

    public int absoluteValue(int x) {
        return math.abs x
    }
}

```

図 6.11: ツールを起動する場所の状況のパターンの例を表した 4 つの図. この 4 つの図は青く表示されている行にあるキャレットの位置でユーザが補完コマンド『Ctrl+Space』を押してツールを起動する直前の状態を示したものである. 上から 1 番目の図 (Task10) が「クエリーと同じ行の左方向に何もない場合」、2 番目の図 (Task08) が「変数宣言が左側にある場合」、3 番目の図 (Task02) が「メソッドの引数の中で使用した場合」、4 番目の図 (Task04) が「return 文の中で使用した場合」を表した図である. それぞれの状況により、ツールが出力するコードが返すべき返り値の型が異なるため、その状況ごとに場合分けをしてその型情報を得ている.

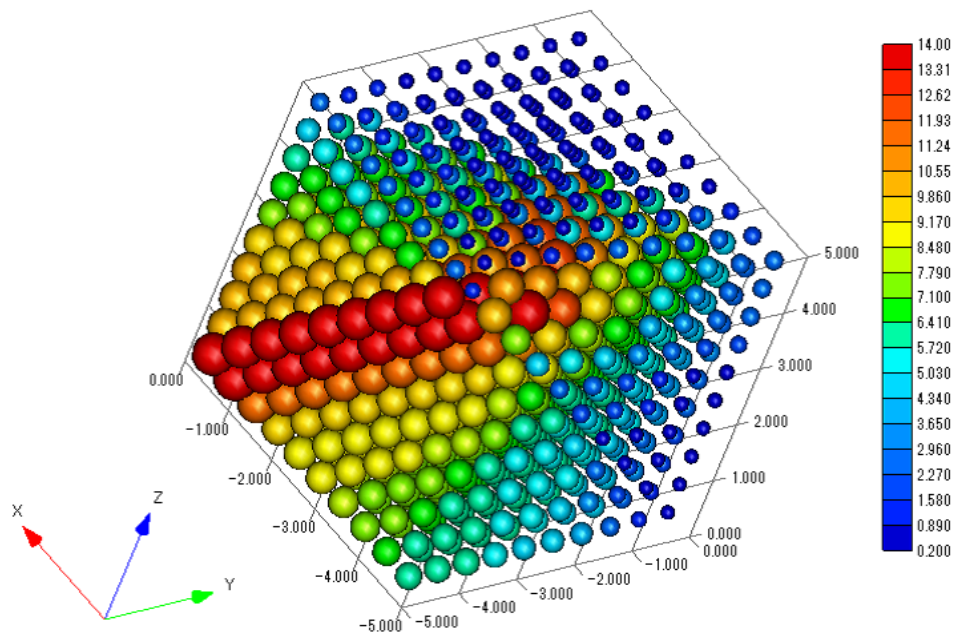


図 6.12: 先行研究 [1] の著者作成の 14 問のタスクに対して、関数および変数を評価する特徴の重みの組み合わせを一定の範囲で動かし「出力中に出現した正解の順位の逆数」をスコアとし、その全 14 タスクの合計値をプロットした 3 次元グラフ。図の X, Y, Z 座標はそれぞれ重み w_1, w_2, w_3 を表す (重み w_4 は 1.0 で固定)。スコアが多い交点ほど点が大きく赤く、少ない交点ほど小さく青く表示されていて、問題数が 14 問なので最大値は 14 となっている。また、このグラフは検索結果上位 10 件までの出力結果を調べたグラフである。

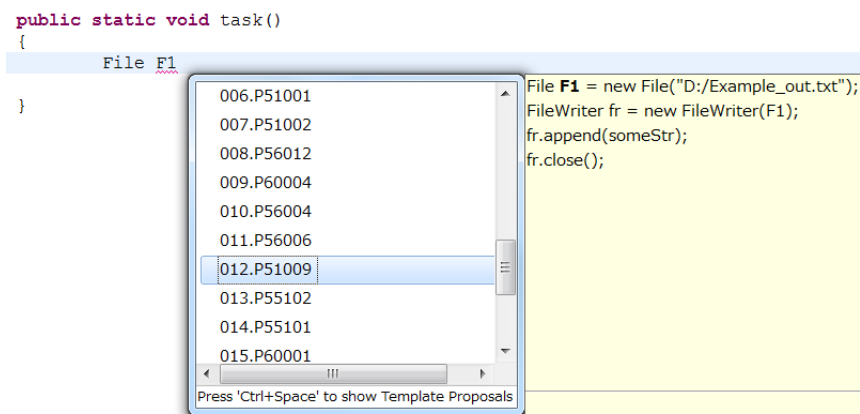


図 6.13: 関連研究 GraPacc [2, 3] を使用中のスナップショット。左側のポップアップに複数のスニペットの番号が表示され、ユーザはそこから 1 つのスニペットを選択することができる。ある 1 つの番号の上にカーソルを置くと、そのスニペットの内容が右側にポップアップ表示される。

第7章 まとめ

本論文の貢献は、先行研究 [1] の Little と Miller が提唱したキーワードプログラミングシステムに対して、

1. システムをさらに使いやすくするために、WWW の検索エンジンが多数のウェブページを順位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アルゴリズムを考案したこと
2. 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にツールの改良を行ったこと
3. ツールを統合開発環境 Eclipse のプラグインとして実装したこと

の3つである。特に2番目の貢献では、あいまい（不正確）なキーワード入力を導入したときの効果を、オープンソースプロジェクトを用いた実験により議論したところ、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ということが分かった。

現在、先行研究 [1] 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 7.1）。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 $w_1, \dots, w_4$ 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 [1] の設定、すなわち「改変無し」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無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ると考

```

/
public String encode(final String value, final Charset charset) throws EncoderException {
    if (value == null) {
        return null;
    }
    return encd txt/v
}

```

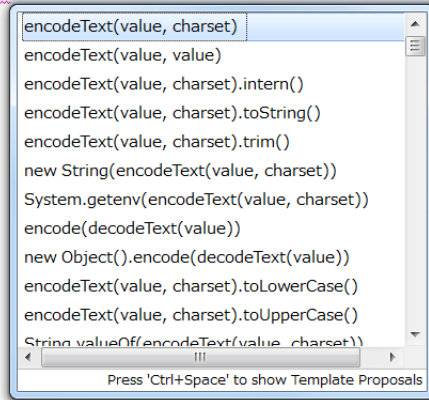


図 7.1: 本研究で作成したプラグインを使用中のスナップショット. ユーザにとって望ましいコード片が `encodeText(value, charset)` の時に, クエリとして `encd txt/v` と 3 語をスラッシュで関数と変数の 2 種類に区切って入力した例である.

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.

謝辞

本研究を行うにあたり、的確なご助言、ご指導を頂きました知能ソフトウェア研究室の栗原正仁教授、小山聡准教授、佐藤晴彦助教に深く感謝いたします。栗原先生には大学4年生から博士課程までの間の長きに渡り、さまざまなご指導を頂き、深く感謝しております。小山先生は研究分野は異なりましたが、普段とは別の視点からのご指導、ご助言があり、大変参考にさせていただきました。佐藤先生は、私と年齢が近いこともあって話しかけやすく、細かいことまで何でも質問してしまっていたのですが、いつも親身になってご指導くださり、とても感謝しております。次に、副指導教員としてご指導頂きました調和系工学研究室内の鈴木恵二教授、自律系工学研究室内の山本雅人教授、ヒューマンコンピュータインタラクション研究室内の小野哲雄教授にも、心より感謝申し上げます。また、知能ソフトウェア研究室の先輩後輩たちにも、感謝申し上げます。5年間の博士課程を楽しく過ごせたのも、研究室内での彼らとの交流があったからこそだと考えております。次に、博士課程まで進学することを許してくれ、支援を下さった私の両親にも深く感謝いたします。両親が居なければ、この論文を書く事はできませんでした。最後に、終戦直後に道内の米軍基地で働き、真空管コンピュータの真空管の点検交換作業を行っていたという私の祖父にも感謝いたします。私が工学部への進学を希望し、こうして博士論文を書くまでに至ったのも、小さいころに半田ごてを使っただけの工作のやり方を教えてくれたりなどした祖父の影響があったからだと感じております。本当にありがとうございました。

参考文献

- [1] G. Little and R. C. Miller. Keyword programming in Java. *Automated Software Engineering*, Vol. 16, pp. 37–71, 2009.
- [2] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Graph-based Pattern-oriented, Context-sensitive Source Code Completion. In *Proc. 34th International Conference on Software Engineering*, pp. 69–79, 2012.
- [3] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. GraPacc: A Graph-based Pattern-oriented, Context-sensitive Code Completion Tool. In *Proc. 34th International Conference on Software Engineering*, pp. 1407–1410, 2012.
- [4] Black Duck Software, Inc. Ohloh Code Search. <http://code.ohloh.net/> 参照日 2014/05/28.
- [5] J. Kim, S. Lee, S. Hwang, and S. Kim. Towards an Intelligent Code Search Engine. In *Proc. Twenty-Fourth AAAI Conference on Artificial Intelligence*, pp. 1358–1363, 2010.
- [6] S. P. Reiss. Semantics-Based Code Search. In *Proc. 31st International Conference on Software Engineering*, pp. 243–253, 2009.
- [7] Eclipse Foundation. Eclipse - The Eclipse Foundation open source community website. <https://eclipse.org/home/index.php> 参照日 2014/11/17.
- [8] M. Bruch, M. Monperrus, and M. Mezini. Learning from Examples to Improve Code Completion Systems. In *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.
- [9] Sangmok Han, David R. Wallace, and Robert C. Miller. Code completion of multiple keywords from abbreviated input. *Automated Software Engineering*, Vol. 18, pp. 363–398, 2011.

- [10] Daqing Hou and David M. Pletcher. An Evaluation of the Strategies of Sorting, Filtering, and Grouping API Methods for Code Completion. In *Proc. 27th IEEE International Conference on Software Maintenance*, pp. 233–242, 2011.
- [11] R. Robbes and M. Lanza. Improving code completion with program history. *Automated Software Engineering*, pp. 181–212, 2010.
- [12] D. Mandelin, L. Xu, R. Bodik, and D. Kimelman. Jungloid Mining: Helping to Navigate the API Jungle. In *Proc. 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 48–61, 2005.
- [13] N. Sahavechaphan and K. Claypool. XSnippet: Mining For Sample Code. In *Proc. 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 413–430, 2006.
- [14] Yusuke Sakamoto, Haruhiko Sato, and Masahito Kurihara. Improvement and Implementation of Keyword Programming. In *Proc. 2010 IEEE International Conference on Systems, Man, and Cybernetics (SMC 2010)*, pp. 474–480, 2010.
- [15] R. Hill and J. Rideout. Automatic Method Completion. In *Proc. 19th IEEE International Conference on Automated Software Engineering*, pp. 228–235, 2004.
- [16] Eclipse Foundation. Code Recommenders. <http://www.eclipse.org/recommenders/> 参照日 2014/10/01.
- [17] Cyrus Omar, YoungSeok Yoon, Thomas D. LaToza, and Brad A. Myers. Active Code Completion. In *Proc. 34th International Conference on Software Engineering*, pp. 859–869, 2012.
- [18] L. Bergroth, H. Hakonen, and T. Raita. A Survey of Longest Common Subsequence Algorithms. In *Proc. the Seventh International Symposium on String Processing Information Retrieval (SPIRE'00)*, pp. 39–48, 2000.
- [19] Gonzalo Navarro. A Guided Tour to Approximate String Matching. *ACM Comput. Surv.*, Vol. 33, pp. 31–88, 2001.
- [20] 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.

- [21] 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.
- [22] Teruo Okuda, Eiichi Tanaka, and Tamotsu Kasai. A Method for the Correction of Garbled Words Based on the Levenshtein Metric. *IEEE Trans. Comput.*, Vol. 25, pp. 172–178, 1976.
- [23] The Apache Software Foundation. Codec - Home. <http://commons.apache.org/proper/commons-codec/> 参照日 2014/12/22.
- [24] OW2 Consortium. CAROL - Home Page. <http://carol.ow2.org/> 参照日 2014/12/22.
- [25] Riad Djemili. jMemorize — SourceForge.net. <http://sourceforge.net/projects/jmemorize/> 参照日 2014/12/22.
- [26] Carnegie Mellon University. CMU Sphinx - Speech Recognition Toolkit. <http://cmusphinx.sourceforge.net/> 参照日 2014/12/22.
- [27] JUnit. JUnit - About. <http://junit.org/> 参照日 2014/12/22.
- [28] Eugene W. Myers. An $O(ND)$ Difference Algorithm and Its Variations. *Algorithmica*, Vol. 1, pp. 251–266, 1986.
- [29] S. Wu, U. Manber, G. Myers, and W. Miller. AN $O(NP)$ SEQUENCE COMPARISON ALGORITHM. *Information Processing Letters*, Vol. 35, pp. 317–323, 1990.
- [30] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based Mining of Multiple Object Usage Patterns. In *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.

研究業績

学術雑誌掲載論文

1. 坂本悠輔, 佐藤晴彦, 小山聡, 栗原正仁, 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, 情報処理学会論文誌, Vol.56, No.3, 2015. 掲載決定.

国際会議

1. Yusuke Sakamoto, Haruhiko Sato, Masahito Kurihara, Improvement and Implementation of Keyword Programming, 2010 IEEE International Conference on Systems, Man, and Cybernetics (SMC2010), pp.474-480, 2010.
2. Yusuke Sakamoto, Haruhiko Sato, Masahito Kurihara, Comment-based Keyword Programming, the International MultiConference of Engineers and Computer Scientists 2011, pp.750-754, 2011.
3. Yusuke Sakamoto, Haruhiko Sato, Masahito Kurihara, A Study on Online Aspect Mining, Proceedings of the International MultiConference of Engineers and Computer Scientists 2010, pp.735-738, 2010.

受賞

1. Certificate of Merit (Student) , 2011 IAENG International Conference on Software Engineering, 発表タイトル「Comment-based Keyword Programming」, 2011年3月.
2. 研究奨励賞, 情報処理北海道シンポジウム 2009, 発表タイトル「キーワードプログラミングの改良」, 2009年10月.

国内学会発表

1. 坂本悠輔, 佐藤晴彦, 小山聡, 栗原正仁, キーワードプログラミングの改良と実装, 日本ソフトウェア科学会第 29 回大会, 東京, 2012 年 8 月.
2. 坂本悠輔, 佐藤晴彦, 小山聡, 栗原正仁, 単語の出現頻度を用いたキーワードプログラミングの改良, 日本ソフトウェア科学会第 28 回大会, 那覇, 2011 年 9 月.
3. 坂本悠輔, 佐藤晴彦, 小山聡, 栗原正仁, コメントベースドキーワードプログラミング, 日本ソフトウェア科学会第 27 回大会, 東京, 2010 年 9 月.
4. 坂本悠輔, 佐藤晴彦, 小山聡, 栗原正仁, キーワードプログラミングの改良と実装, 情報処理学会創立 50 周年記念 (第 72 回) 全国大会, 東京, 2010 年 3 月.
5. 坂本悠輔, 栗原正仁, キーワードプログラミングの改良, 情報処理北海道シンポジウム 2009, 旭川, 2009 年 10 月.
6. 坂本悠輔, 栗原正仁, ソフトウェア開発中におけるアスペクトマイニングツール, 電子情報通信学会 2009 年総合大会 情報・システムソサイエティ特別企画 学生ポスターセッション, 愛媛, 2009 年 3 月.
7. 坂本悠輔, 栗原正仁, オンラインアスペクトマイニングの研究, 平成 20 年度電気・情報関係学会北海道支部連合大会, 札幌, 2008 年 10 月.