特集　情報システム

原著論文

プログラミング学習システムCAPTAINのための自動ダミー断片処理

大城正典*・岸本頼紀*・布広永示*

要旨：著者らは，プログラミング学習システムCAPTAIN（Computer Aided Programming Training and INstruction）を開発してきた．本システムでは，完全な実行可能プログラムが構造指向の考え方に従って意味的に分析され，断片化される．ソースプログラムの断片はランダムに並べ替えられて表示され，学習者はそれらをパズルゲーム要領で，元のソースプログラムになるように並べ替える．学生が並び替えた断片は，以前の論文で提案した識別子依存グラフの方法を用いて意味的に分析される．本稿では，新たにダミー断片を元の断片に追加して提示する演習と，そのダミー断片の自動的な取扱いについて紹介する．学生が並べ替えたプログラム断片は，コンパイルによりチェックされ，改善された方法によって依存グラフとして分析され，あらかじめ分析された元のプログラムの依存グラフと比較される．それらが同じであると考えられる場合，学生の解答は正しいと判断する．

キーワード：プログラミング教育，e-ラーニング，Java

Automatic Dummy Program Fragments Handling for Programming Training System CAPTAIN.

Masanori OHSHIRO*, Yorinori KISHIMOTO* and Eiji NUNOHIRO*

Abstract: The authors have developed the programming learning system CAPTAIN (Computer Aided Programming Training and INstruction). In this system, a complete executable program is semantically analyzed and fragmented according to the structure-oriented concept. Fragments of the source program are randomly sorted and displayed, and the learner rearranges them so as to become the original source program in a puzzle game manner. The fragments rearranged by students are semantically analyzed using the method of identifier dependence graph proposed in the previous paper. In this paper, we introduce the exercise to add a new dummy fragments to the original fragments and present them, and the automatic handling of that dummy fragments. Program fragments sorted by students are checked by compilation and analyzed as dependency graphs in an improved way and compared with dependency graphs in pre-analyzed original program. If they are considered to be the same, the student's answer is judged to be correct.

Keywords: Programming education, e-learning, Java

*東京情報大学　総合情報学部
Faculty of Informatics, Tokyo University of Information Sciences

2018年６月１日受付
2018年９月18日受理

1．はじめに

著者らは，プログラミング学習システムCAPTAIN (Computer Aided Programming Training and INstruction) を開発し，実際のJavaプログラミングの授業で使用してきた（図1）[1]-[9]．この学習システムでは，学生はパズルを解くようにプログラムを完成させる．問題の元となる正しく動作するプログラムは，本システムによって1行～数行の断片に分割され，ランダムに並べ替えられて学生に提示される．学生は，これを正しく並べ替えるように指示される．完成されたソースプログラムは，学生の解答としてコンパイラでコンパイルされ正しいか検査される．

より最近のバージョンでは，構文指向の断片化法が追加された[7]．この方法では，単純にソースコードを行単位に分割するだけではなく，ブロック構造を持った構文は，ブロック構文要素にまとめられて断片化される．例えば，図2は最も簡単なクラス定義を構文指向の断片化法で断片化した例である．

この図のように，外側のブロック部分と内側の部分に分かれて断片化される．学生は，ドラッグ＆ドロップ操作によって，断片を並べ替えたり，ブロック構文の断片の中に挿入して，プログラムを完成させる（図3）．

この構文指向の断片化によって，学生がソースプログラム上の構造を強く意識することになり，文法を理解する能力が向上することが期待できる．後に，著者らはこの構文指向の断片化に基づいた自動弱点検出機能などを提案した[8] [9]．

1．1　解答コードの正解判定の問題

初期のシステムでは，学生の解答コードをコンパイルし，成功すれば正解，失敗すれば不正解としていた．しかし，コンパイルに成功したとしても意味的には正しくない場合があるのは自明である．問題の元となる正解コードと学生の解答コードが厳密に一致した場合のみ正解とする処理も可能だが，プログラム断片の位置を変えても意味的には同じ処理になる場合もあるので，教育的な観点から採用しなかった．そこで，学生の解答コードが意味的に正しいか否かを判断する手段が必要となった．

しかしプログラムの意味的な正しさは，一般にそ

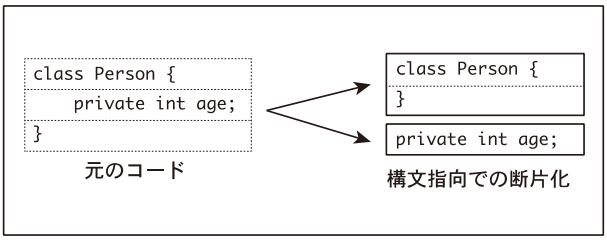

図2　構文指向によるコードの断片化例

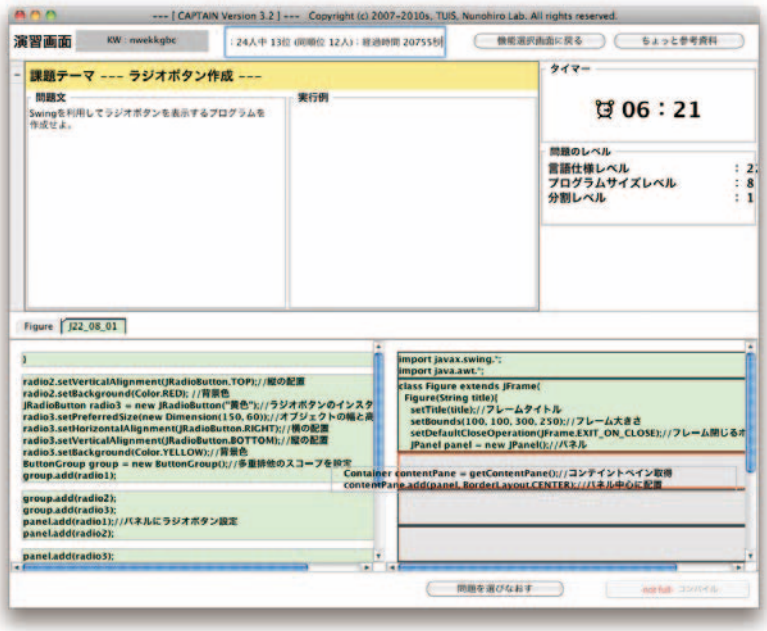

図1　CAPTAINのメインウィンドウ

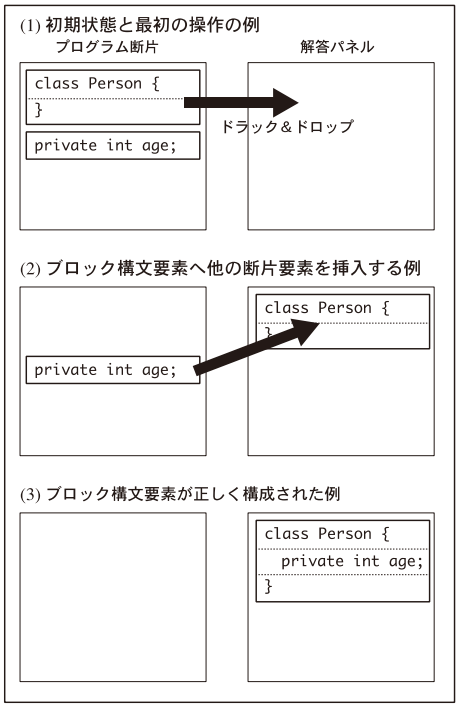

図3　断片の表示と学習者のタスク

のプログラムが意図した目的に依存しているため難しい．そこで，著者らは，本システムが意味的に正しいコードを保持していることに着目し，識別子依存グラフによって学生の解答コードの意味的正しさを検証する手法を考案した．

1．2　識別子依存グラフ

著者らは，「プログラムがある目的のために正しく動作している」とは，「各識別子の値が適切な順序で参照・変更されている」ものと仮定した．

また，本システムのために正解として用意されているコードは，特定の目的のために注意深くコーディングされており，文法的な誤りが無いことはもちろん，意味的に十分正しいと仮定できる．

そこで，正しいコードにおける識別子の参照・変更の依存関係をグラフとして表すことによって，学生が提示された断片を並べ直して再構成したコードの正しさを検証する方法を考案した．我々はこのグラフを識別子依存グラフと呼んでいる．この手法では，

・正解コード（意味的に正しい）
・正解コードの断片を学生が再構成したコード（意味的に正しいか不明のコード）

のそれぞれの識別子依存グラフを比較し，それが同等のものであるなら，学生が再構成したコードは意味的正しさを失っていない，すなわち学生の解答コードも正しいと見なす．

基本的な識別子依存グラフの例を以下に示す．図４は正解コードの一例である．コードは，一般的な構文解析手法によって解析されて，行単位・構造単位の記号（行記号）が割り振られ（図４-(1)），行記号のシーケンスとして取り扱われる（図４-(2)）．以降の解析において，空白行は無視され，閉じ中括弧のみの行はすべて同値と見なして良いとする．その他，同値の記号列は，＝で同値であることを表す．たとえば，

A1 = B1, B2, B3；

は，A1は，記号列B1，B2，B3と同値であることを表しており，これはA1をB1，B2，B3という記号列に書き換えて良いことを意味している（一般的な言語理論の生成規則に相当する）．ひとつのコードから生成されたこのような書き換え規則の左辺に位置する記号を言語理論と同様に非終端記号，書き換え規則の左辺に位置しない記号，すなわち，それ以上書き換えられない記号を終端記号と呼ぶこととする．なお，カンマとセミコロンは，表現上の便宜的な区切り文字であり，解析の処理内容には一切関係ない．行記号の列に記述において，

A1，A2，A3　または　［A1，A2，A3］

のように，まったく括弧に囲まれていない列表記，または［ ］に囲まれた列表記は，順序が固定された記号列を表す．また，

｛A1，A2，A3｝

のように ｛｝ に囲まれた列表記は，任意の順でかまわない記号列を表す．記号列を ｛｝ で囲むか，［ ］で囲むかは，Javaの文法によって形式的に判断される．

図４-(1) のコードから得られる書き換え規則を使って，プログラム全体を表す記号A1に関する書き換え規則

A1 = ［B1，B2，B3］；

を，これ以上書き換えられない状態まで書き換えを行うと，プログラム全体A1がどのような行から構成されているかを表す書き換え規則

A1 = ［B1，｛C1，C2，［D1，E1，D3］，［D4，［E2，F1，E4］，D6］，［D7，E5，D9］，［D10，E6，D12］，［D13，｛E7，E8｝，D15］，B3］；

が得られる．

このようにして得られた行記号を使って，識別子の依存性を表す規則を表す識別子依存グラフを作成する．その過程をより単純かつ複数の識別子が存在する例で示す．

ある識別子の参照と変更処理は，その宣言から始まるスコープの中で，次の様なパターンで現れる．まず，

識別子の宣言　０回以上の識別子の参照処理

から始まり，その後に

識別子の値の変更　０回以上の識別子の参照処理

というパターンが０回以上順不同で繰り返される．これを著者らの行記号列で表すと，

［識別子の宣言行，｛識別子の参照処理を行う記号列｝，｛［識別子値の変更を行う行，｛識別子の参照処理を行う記号列｝］，…｝，…］

と形式的に表現できる．このように，プログラムスライシング[10][11]を行単位で行うことで，特定の識別子に関する値の変更・参照の依存関係を行単位

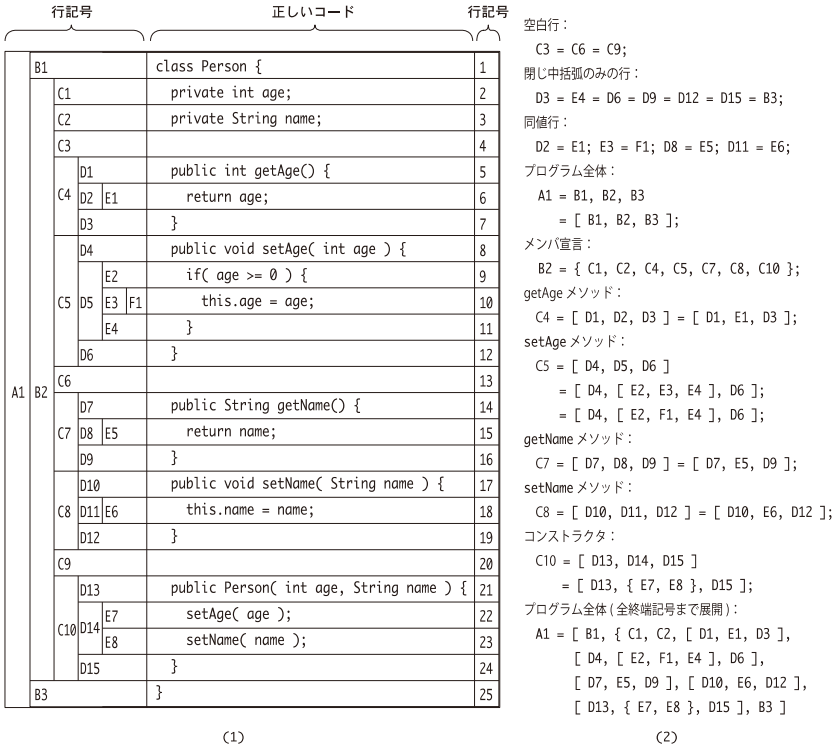

図４　行記号シーケンスによるプログラムの構造表現

で抽出することができる．著者らは，これを（プログラム全体に関するものではなく各識別子に関するものであるという意味で）部分識別子依存グラフと呼んでいる．図５では，あるコード中の変数a, b, c, d に関するそれぞれの識別子依存グラフとその記述方式を示している．

部分識別子依存グラフの生成は，次の様に行われる．まず，あらかじめ一般的な構文解析手法によって，宣言された識別子とそのスコープ，各識別子の宣言行または値の変更行を調べておく．各識別子について，空の作業用行番号列を用意しておき，行単位でプログラムを走査していく．その過程で，識別子の宣言行または値の変更行を見つけたら，‘[’ とその行記号を作業用行番号列に追加する．その後，その識別子の値を参照しているだけの行を見つけたら，それらをまとめて中括弧 {} で囲み，作業用行番号列に追加して，‘]’ で閉じる．これを走査過程でその識別子のスコープを過ぎるまで繰り返し，最後に作業用行番号列全体を [ ] で囲み，部分識別子依存グラフとする．

図５の例では，変数aに関する部分識別子依存グラフは

#1 B2：a = ［C1，｛C2｝，｛C3，｛C4，C5，C6｝｝］

となっている．これは，「C1の直接左に ‘[’ 」，「C2の直接左に ‘{’ 」，「C2の直接右に ‘}’ 」，…というように解釈する．

識別子依存グラフは，図６-(1) に示す自明な変形で単純化・変形できる．図６-(2) に，図５の各部分識別子依存グラフを単純化した例を示す（下線は単純化・変形の適用箇所を表している）．

これらの単純化した各部分識別子依存グラフの内容を，元の正解コードから生成され文法的制約を反映している行記号シーケンスに適用することで，文法的制約と意味的制約を持つプログラム全体の識別子依存グラフを求めることができる．

図７に，その過程を示す（下線は部分識別子依存グラフ・単純化・変形の適用箇所を表している）．処理過程は次の様に実行される．まず，適用する部分識別子依存グラフを最初からスキャンする．行記号が現れたら，その左右の ‘[’，‘]’，‘{’，‘}’ を調べる．そして，適用対象となる行記号シーケンスに同じ行記号があるか前方から調べ，見つかったらその左右に部分識別子依存グラフ中の同じ行記号の左右にある ‘[’，‘]’，‘{’，‘}’ を挿入する．その

A1	B1		{
	B2	C1	int a = 0;
		C2	int b = a + 3;
		C3	a = 3;
		C4	int c = b + a;
		C5	int d = a + c;
		C6	System.out.println(a);
	B3		}

部分識別子依存グラフ

識別子aに関する依存グラフの例

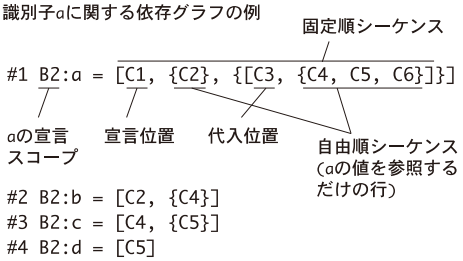

図５　部分識別子依存グラフの例

識別子依存グラフの単純化・変形

① [a1] -> a1, {a1} -> a1
② {[s1]} -> [s1], [{s1}] -> {s1}
③ [s1 [s2] s3] -> [s1, s2, s3]
④ {s1 {s2} s3} -> {s1, s2, s3}
⑤ [s1 {s2] s3} -> [s1, {s2, s3}]
⑥ {s1 [s2} s3] -> {s1 [s2, s3]}

※a1 は単終端行記号，s1,s2,s3 は行記号列

(1)

図 4 の部分識別子依存グラフを単純化した例

#1 B2:a = [C1, {C2}, {[C3, {C4, C5, C6}]}]
↓ ①と②を適用
= [C1, C2, [C3, {C4, C5, C6}]]
↓ ③を適用
= [C1, C2, C3, {C4, C5, C6}]

#1’ B2:a = [C1, C2, C3, {C4, C5, C6}]
#2' B2:b = [C2, C4]
#3' B2:c = [C4, C5]
#4' B2:d = [C5]

(2)

図６　識別子依存グラフの単純化・変形とその例

後，単純化ルール（図６-(1) ①～④）と変形ルール（図６-(1) ①～④）を適用箇所が無くなるまで適用する．そして，次の部分識別子依存グラフの適用処理に進む．この処理をすべての部分識別子依存グラフを適用するまで繰り返すことで，プログラム全体の識別子依存グラフが生成される．

行記号の順序が固定された［ ］で囲まれた行記号列も，順序が任意の {} で囲まれた行記号列も，行記号列を入力とした決定性有限オートマトンと考えられ，それらを組み合わせた行記号列も行記号列を入力とした決定性有限オートマトンと言える．したがって，得られたプログラム全体の識別子依存グラフは，一般的に知られる状態遷移を基本とした決定性有限オートマトン処理アルゴリズムによって，入力された行記号列が受理されるかどうかを検査することができる．図７に示したとおり，図５のプログラムの識別子依存グラフは

A1 = ［B1，C1，C2，C3，｛［C4，C5］，C6｝，B3］… ※

となっている．この識別子依存グラフが表すオートマトンが，学生の解答コードから得られた行記号列を受理するか調べれば，学生の解答を正しいかどう

A1 = [B1, { C1, C2, C3, C4, C5, C6 }, B3]
↓ #1’を適用
A1 = [B1, { [C1, C2, C3, {C4, C5, C6}] }, B3]
↓ 図 6-(1)②を適用
A1 = [B1, [C1, C2, C3, {C4, C5, C6}], B3]
↓ 図 6-(1)③を適用
A1 = [B1, C1, C2, C3, {C4, C5, C6}, B3]
↓ #2’を適用
A1 = [B1, C1, [C2, C3, {C4], C5, C6}, B3]
↓ 図 6-(1)⑤を適用
A1 = [B1, C1, [C2, C3, {C4, C5, C6}], B3]
↓ 図 6-(1)③を適用
A1 = [B1, C1, C2, C3, {C4, C5, C6}, B3]
↓ #3’を適用
A1 = [B1, C1, C2, C3, {[C4, C5], C6}, B3]
↓ #4’を適用
A1 = [B1, C1, C2, C3, {[C4, [C5]], C6}, B3]
↓ 図 6-(1)①を適用
A1 = [B1, C1, C2, C3, {[C4, C5], C6}, B3]

図７　プログラム全体の識別子依存グラフ導出の例

か判断できることになる．たとえば，学生の解答コードから得られた行記号シーケンスが，

A1 = B1，C1，C2，C3，C4，C6，C5，B3

である場合，これはオートマトン（※）で受理されないので不正解となる．また，学生の解答コードから得られた行記号シーケンスが，

A1 = B1，C1，C2，C3，C6，C4，C5，B3

の場合は，オートマトン（※）で受理するので正解となる．

このように識別子依存グラフを利用した判定によって，本システムで得られた学生の解答コードが，Java文法およびあらかじめ準備された正解コードの意味を維持しているかどうか，すなわち正解か否かを判別することができた．

2．ダミー断片を混ぜた問題表示

著者らのシステムは前述したように，これまで一つの正しいコードから生成された断片しか使用していなかった．そのため，学生が問題のプログラムの意味を正しく理解しておらず，断片をランダムに置いてしまった場合でも，解答が正しいコードと一致してしまい正解と見なされる場合があった．実際に，授業で使用した際に，試行を何回も繰り返して総当たり的に正解を見つけるなどの行為が観察された．問題は，出題された断片を適当に列べただけで正解になってしまう可能性が高い点にある．

そこで著者らは，正しいソースコードから生成された断片の他に，それ以外のダミーとしての断片を追加して表示し，学生に正しい断片を意識的に選ばせる必要性があると判断した．難しい問題は，解答のプログラムにダミーの断片が入っていた場合でも，元の正しいプログラムと同じ意味を持つ可能性があると言うことである．つまり，学生の解答にダミーの断片が入っているというだけで，不正解と判断することはできない．本論文では，ダミーの断片を混ぜて問題を表示し，上述した問題を解決するダミー断片の自動的な取扱い方法を提案する．

正しいコード例を図８に示す．このプログラムから得られる行記号シーケンスと識別子依存グラフを図９に示す．図８の正しいソースコードは，前述した構文指向の断片化法により，それぞれの構造ごとに断片化される．図10は，図８の正しいコードから生成した構文指向断片とその内部情報である．

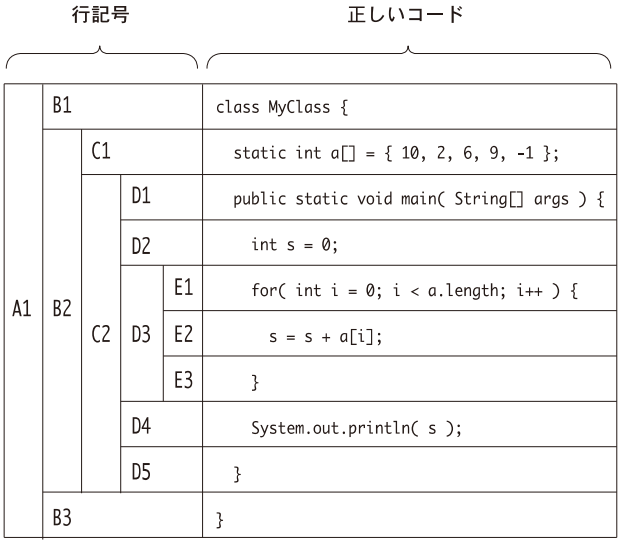

図８　正しいコードの例

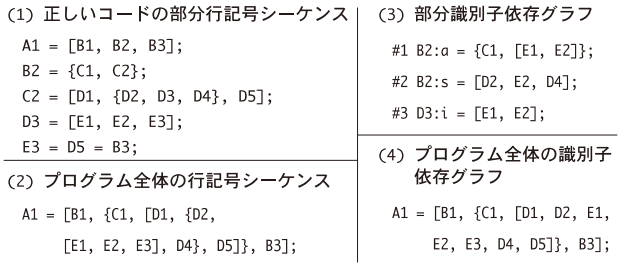

図９　正しいコード例に対する行解析

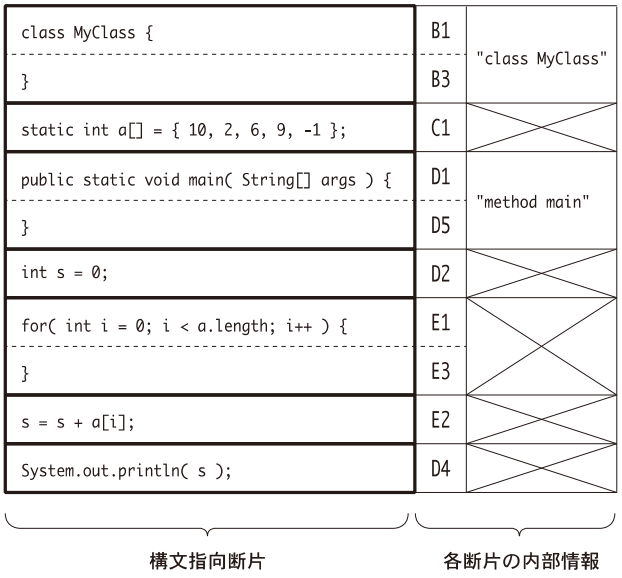

図10　正しいコード（図８）の構文指向断片と内部情報

表示に混ぜられるダミーの断片は，本システムの保持する正しいコードの集合から生成された構文指向断片の中からランダムに選ばれる．我々は，正しいコードから生成される構文指向断片の数と，挿入されるダミー断片の数の関係を表１のように定めた．これは，本システムがプログラミング学習の初心者用であり，「多くの中から少ないものを探す」という学習者の意欲を低減させるタスクとならないように配慮したものになっている．

表１　挿入されるダミー断片の頻度

問題として使用される正しいコードの断片数	挿入されるダミー断片の数
1 ～ 5	0 ～ 1
6 ～ 10	0 ～ 2
11 ～ 15	0 ～ 3
⋮	⋮

ひとつの正しいコードから生成されたすべての構文指向断片と表１にしたがってランダムに選ばれたダミー断片は，ランダムに並べ替えられた状態で学生に対して表示される．それと同時に「int型要素を持つ配列 a をMyClassクラスのstaticフィールドとして宣言し，その要素の初期値は順に 10, 2, 6, 9, -1 にせよ．そして，mainメソッドで，配列 a の全要素の合計値を求めて表示せよ.」という指示を学生に対して表示する．

３．ダミー断片の取扱い

実際にダミー断片が混入した解答コードの例を示す（図11）．元の正解コードに含まれていない断片（またはそれを構成する行）には，図11のF1のように，元の正解コードに割り振られた行記号とは異なる行記号が割り振られる．図11の解答コードの行記号列は先頭から

B1，C1，D1，D2，E1，E2，F1，E3，D4，D5，B3

で，元の正解コードから得られた識別子依存グラフが表して言うオートマトンは，この記号列を受理しないのは明かである（元の正解コードに含まれない行記号があるため）．しかし，もし図11のF1の行内容が

System.out.println（ i ）;

であれば，余計な変数表示は行われるものの，元の正解コードのプログラムの処理には影響をあたえていないと言える．著者らは，このような場合の解答も正解として扱うこととした．しかし，識別子の値を変更しない行であっても，クラスの名やメソッドシグネイチャ，if文やfor文のような処理の流れを変える制御文の先頭行（条件判定を含む）は，プログラムの意味に大きく影響を与える．

A1	B1					class MyClass {
	B2	C1				static int a[] = { 10, 2, 6, 9, -1 };
		C2	D1			public static void main(String[] args) {
			D2			int s = 0;
			D3	E1		for(int i = 0; i < a.length; i++) {
				E2		s = s + a[i];
				F1		i = i + 1;
				E3		}
			D4			System.out.println(s);
			D5			}
	B3					}

図11　不正解コードの例とその依存グラフ解析

また，図８の正解コードの特定の行とまったく同じ内容の行が偶然ダミー断片として採用され（学生には同じ内容の行が複数表示されることになる），そのような同値の行が解答として指定された場合も，正解として扱わなければならない．

そこで，ダミー断片を取り扱うために，我々は依存フラグによる解析方法を拡張した．以前の依存フラグは，識別子だけに着目していた．しかし，我々は他のコードからもたらされたダミー断片を取り扱うためには，識別子依存グラフの各行（ノード）について

・識別子の宣言，識別子の値を変更する行
・メソッド定義の先頭行おいて
・クラス定義の先頭行
・for文，while文，if文の先頭行

の場合には，その行の内容を端的に表す情報をタグとして付与しておくこととした（表２）．この情報の抽出とタグ付けは，正解コードに対してあらかじ

め実行される．結果的に正解コードの集合から作られるダミー断片の各行にも，元の正解コードの内容にしたがった情報がタグとして付与されていることになる．

表2　重要なノードのタグ内容

各ノードの内容		タグの内容		
ノードタイプ	例	例		
変数宣言	int a = 10, b;	int	a	10
		int	b	0
代入式	i = 10;	i		10
メソッド定義	s1 m(s2) s3 {	s1	s2	s3
クラス定義	s1 class c s2 {	s1	c	s2
for 文	for(e1;e2;e3) {	e1	e2	e3
while 文	while(e) {	e		
if 文	if(e) {	e		

*a,b などの記号はより詳細なトークン列の場合もある

学生の解答コードが正しいか判断する処理は，次の様に行う．まず，学生の解答コードはコンパイルしてエラーが出た場合は，間違いとする．次に，学生の解答コードの行記号列が，識別子依存グラフのオートマトンに受理されるか調べる過程で，もし受理されない行記号が現れた場合は，

1．受理されない行記号がダミー断片のものの場合，
　1－1．受理されない行記号に表2の情報タグがついている場合は，正解コード中の対応する行記号（受理可能な行記号候補）の情報タグと比較する．両者の内容が一致すれば，受理されない行記号を，正しい行記号と同一と見なして受理する．もし両者のタグ情報が一致しなければ，解答コードを間違いとみなす．
　1－2．受理されない行記号に表2の情報タグがついていない場合は，元の正解コードの処理内容に影響を与えないものとして，無視して読み飛ばす．
2．受理されない行記号がダミー断片のものではない場合
　2－1．解答コードは不正解とする．

とする．

4．間違った解答コードの判定例

実際に，図11の内容が解答コードとして与えられた場合，行記号F1が読み込まれた時点で，図9-(4)の識別子依存グラフが表しているオートマトンは受理不可能となる．F1はダミー断片由来の行なので，F1に表2に従った情報タグがついているか調べる．F1は代入文のため，表2に従った情報タグが付与されている．次に，F1の情報タグとこの時点でオートマトンが受理可能なE3の情報タグの内容と比較する．しかし，E3には表2に従った情報タグは付与されていない．したがって比較不能であり，図11の解答コードは不正解と判断できる．

5．本研究の意義・考察

本研究で開発している学習システムCAPTAINは，

1．意味的に正しい正解コードが用意されている
2．行単位でプログラムを完成させる

という性質を持っている．その特徴を活かし，かつ識別子依存グラフという考え方を導入することで，学生の解答コードにある程度の自由さを認めつつ意味的正しさを確かめる方法を提案した．また，ダミー断片を問題に混ぜることによって，単なる断片の入れ替え作業で正解に到る可能性を少なくし，学生に問題のプログラムに何が必要かを意識させることに成功した．

一般に，学生にプログラムを完成させる問題では，学生の解答コードの意味的正しさを自動的に確かめるのは難しい．よく用いられるのは，入力・出力を前提としたコードを学生に書かせ，入力データを与えて仕様に従った出力が得られるかどうかを確かめる方法である．しかし，問題に合った入力データを十分用意しないと，処理が間違っていても偶然正しい出力が得られるという問題がある．また，クラスの定義，メソッドの定義，などはこのようなテスト手法では学習させることは困難である．一方，著者らのシステムでは，クラスの定義，メソッドの定義などの学習には有効と言える．

6. おわりに

著者らは，CAPTAINのための新しい特徴として，正しいプログラム断片にダミーの断片をまぜて提示し，学生の解答コードの正しさを検証する機能を提案した．今後，この機能の実装と実際の授業での使用を行い，効果を検証する予定である．

謝辞

本研究は，文部科学省科学研究費基盤研究（C）課題番号15K01086の補助を受けた．

【引用文献】

［1］ Nunohiro, E., Mackin, K., Ohshiro, M., Matsushita, K. and Yamasaki, K.: Implementation of a GA driven programming training support system, *The 12th International Symposium Artificial Life and Robotics,* pp.517-522 (2007).

［2］ Nunohiro, E., Matsushita, K., Mackin, K., Ohshiro, M. and Yamasaki, K.: Design and development of a puzzle based programming learning support system with genetic algorithm (in Japanese), *Transactions of Japanese society for information and system in education,* Vol.25, No.2, pp.207-213 (2008).

［3］ Yoneyama, Y., Matsushita, K., Mackin, K., Ohshiro, M., Yamasaki, K. and Nunohiro, E.: Puzzle based programming learning support system with learning history management, *The 16th International Conference on Computers in Education (ICCE2008),* pp.623-627 (2008).

［4］ Yamakawa, Y., Ohshiro, M., Matsushita, K., Mackin, K. and Nunohiro, E.: Programming learning support system 'CAPTAIN' with motivational study model, *The 18th International Conference on Computers in Education (ICCE2010),* pp.171-175 (2010).

［5］ Ohshiro, M., Mackin, K., Matsushita, K., Nunohiro, E. and Yamakawa, Y.: Programming learning support system with learning progress monitoring feature, *Joint 5rd International Conference on Soft Computing and Intelligent Systems and 11th International Symposium on Advanced Intelligent Systems (SCISISIS 2010),* pp.1465-1468 (2010).

［6］ Ohshiro, M., Matsushita, K., Mackin, K., Yamaguchi, T. and Nunohiro, E.: Programming learning support system with competitive gaming using monitoring and nicknames, *The 17th International Symposium Artificial Life and Robotics,* pp.238-241 (2012).

［7］ Ohshiro, M., Yamaguchi, T. and Nunohiro, E.: Programming learning support systems focused on structures of programming language and code, *The 18th International Symposium Artificial Life and Robotics,* pp.106-109 (2013).

［8］ Ohshiro, M., Yamaguchi, T., Kawano, Y. and Nunohiro, E.: Automatic weak points detection method using syntax-oriented code fragmentation for programming learning systems, *The 19th International Symposium Artificial Life and Robotics,* pp.596-599 (2014).

［9］ Ohshiro, M., Yamaguchi, T., Kawano, Y. and Nunohiro, E.: Automatic learning planning systems for programming learning support system, *The 22nd International Symposium Artificial Life and Robotics,* pp.492-495 (2017).

［10］ Weiser, M.: Program slicing, *Proceedings of the 5th International Conference on Software Engineering,* pp.439-449, IEEE Computer Society Press (1981).

［11］ Krinke, J.: *Advanced Slicing of Sequential and Concurrent Programs,* Vdm Verlag Dr. Mueller E.K. (2013).