n-gram解析手法を応用したプログラム中の欠損の検出

吉川裕之† 貴島寿郎† 梅村恭司†

プログラムでは同じようなパターンの繰返しが多い．したがって，プログラム中に一定回数以上現れる，一定長さ以上の文字列を取り出すと，プログラムのほとんどの部分を取り出すことができる．取り出せた部分は複数回現れていることから，プログラムとして意味がある部分であるといえる．一方，取り出せない部分はプログラム中の特異な部分であると考えられる．我々はプログラムの誤りなどが取り出せない部分に含まれるのではないかと考え，この部分にマークを付けるツールを作成した．本稿では特異部分を検出するアルゴリズムを説明し，コンパイラの検出できない誤りの発見に，付与されたマークが役に立つことを示した．

Detecting Faults in Programs Using n-gram Analysis Method

Hiroyuki Yoshikawa,† Toshiro Kijima† and Kyoji Umemura†

Detecting errors in programs is always a key for programmers. Yet, it is a very difficult task. The objective of this paper is to demonstrate the usefulness of a tool which we developed as a detector of errors in programs using n-gram analysis method. In general, we can differentiate valid and invalid parts of a program. Valid parts can be identified by strings whose frequency and length are larger than given values. Logically, invalid are the parts which remained unidentified. Errors are most likely to exist in such parts. This paper presents the algorithm of the tool, and verifies that the tool we developed can detect errors which compilers usually cannot.

1. はじめに

近年，プログラムの記述言語も多種多様になり，システムごとに独自のインタプリタを持ち，独立の言語を持つことも多くなってきた．このような言語においては，プログラムを検査するツールが利用できず，些細な編集ミスが発見できないことを経験する．

一方，アプリケーションなどのプログラムは巨大化の傾向にある．このような巨大なプログラムでは同じようなパターンが数多く存在する．パターンの出現の状況を調べてみると，プログラムの大部分が，繰返しのパターンで構成されており，その中で，繰返しパターンに当てはまらない一部分である部分，つまり，非定型部分が存在する．

プログラムの編集上の誤操作による欠損が存在する部分であれば，高い確率で非定型部分（非定型パターンからなる部分）が生じる．もちろん，非定型部分は必ずしも誤りとは限らない．しかし，大きなプログラム中の非定型部分を検出し，そこをさらに調べることによって，プログラムの表層上の誤り箇所等が効果的に検出できるのではないかと考えた．そこで，プログラムの非定型部分を検出するツールの1つを作成した．

このツールの作成にあたり，文法に添ってプログラムを検査するのでは発見できない誤りで，かつ，編集ミスなどで混入する誤りを検出することを目標とした．そして，多種多様なプログラミング言語やユーザが自ら定義する言語などを念頭に置き，プログラミング言語の個別の文法に依存しないことをツールの設計条件に加えた．

本稿では，プログラム中の非定型パターンを検出するアルゴリズムについて詳しく記述し，自然言語における n-gram 解析の手法との差を説明する．そして，その手法を用いたツールのプログラムの誤りの検出の性能について，編集ミスのような欠損に対する検出効果を評価した．

2. アプローチ

あるテキストに含まれる長さ n の部分文字列を n-gram と呼ぶ．たとえば，1-gram はテキストに含まれるすべてのアルファベットである．

自然言語処理の分野では，n-gram が文書中にどの

† 豊橋技術科学大学情報工学系
Department of Information and Computer Sciences, Toyohashi University of Technology

程度の頻度で現れるかを調べ，一定回数以上現れたn-gramを取り出すことで，文書中の意味のある固まりを取り出す試みがなされている[1)~4)]．ここにおいて，与えられた長さLと頻度Fに関して熟語が定まる．熟語は，プログラムテキスト上の長さL以上の部分文字列であって，同一の部分文字列がF回以上出現するもののこととする．これは文献にあるn-gram解析によって取り出された固まりに一致する．定型パターンは，この熟語の前後をさらに分析して，区切りを正確にしたものである．その結果，熟語の部分文字列として定型パターンを切り出す．

プログラムリストには定型パターンの繰返しが数多く現れる．この傾向はプログラムサイズが大きくなると特に顕著になる．大きなプログラムでは，定型パターンのほとんどが熟語であることが観測できた．そこで我々はプログラムリストに対して熟語の検出を行うと，ほとんどの部分が熟語として取り出すことができると予想した．すなわち，熟語の部分はプログラムとして当然の性質を持つ部分であると考えられる．一方，熟語と熟語の間の取り出せなかった部分は特異な性質を持つと考えられる．我々はこの熟語間の取り出せなかった部分に着目し，グルー（glue）と呼ぶことにした☆．

図1に熟語とグルーの例を示す．図1では，2回以上現れた3文字以上の文字列を熟語としている．図1の例では，**abc**，**bcd**，**abcde**が熟語となり，そのどれにも属さない部分，前から4文字目のcと10，11文字目のf，gがグルーである．なお，**bcde**も熟語としての条件を満たしているが，どの**bcde**も**abcde**の一部分としてしか現れていないのでここでは熟語とは見なさない．この例では2回以上現れた3文字以上の文字列を熟語としたが，熟語の長さと頻度はツールのパラメータとして指定する．

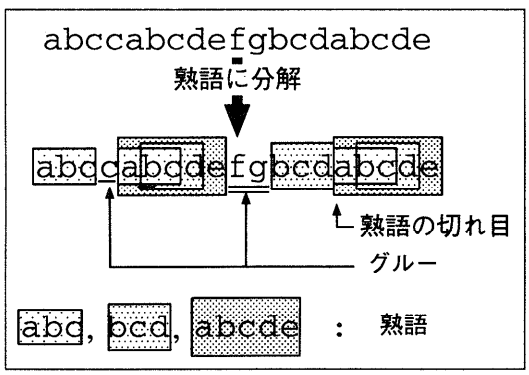

図1 熟語とグルーの例
Fig. 1 The example of idioms and glues.

また，熟語の切れ目にも着目した．隣接する2文字について，2文字ともいずれかの熟語に含まれているが，この2文字を同時に含む熟語がない場合，この2文字の間を熟語の切れ目とする．図1では**bcd**と**abcde**の間が熟語の切れ目である．

図2にプログラムに関して処理したグルーの例を示す．入力は同じような構造を持つFORTRANのサブルーチンが3つ存在するプログラムである．連続したハットでマークされた部分が長さのあるグルーである．スラッシュで示されたところには，長さ0のグルーが存在する．プログラム中には，サブルーチンを呼び出している部分が存在しないので，サブルーチンの名前がグルーとして取り出されている．また，2番目のサブルーチンのDO文において，コンマとすべきところをピリオドになっているところがグルーとして取り出されている．FORTRANの文法では，この文は「**do10i**」という変数に，「**1.100**」を代入する文と解釈することになっているため，コンパイラはエラーを検出しないが，グルーと検出される．人間にとっては，熟語であると考えられる部分にグルーが存在することになるので，ここに何らかの問題がある可能性が

```
      subroutine foo
                 ^^^
      do 10 i=1,100
10      a = a + i
                /
      return
      end

      subroutine bar
                 ^^^
      do 10 i=1.100
                ^
10      a = a + i + 1
                  ^^
      return
      end

      subroutine qwe
                 ^^^
      do 10 i=1,100
10      a = a - 1
              ^^^
      return
      end

      a = 1.1
          /
      b = 1.2
      ^     ^
      c = 1.3
      ^     ^
      end
```

図2 プログラムにおけるグルー
Fig. 2 The glues in a program.

☆ 熟語と熟語の軟らかい部分と考えたのが名前の由来である．

あると気付くことになる．また，このような機能が，FORTRANの文法情報を用いないで実現することができる．

我々はプログラムリストについて熟語の検出を行い，その切れ目の補正処理を行ったあとに，グルーと熟語の切れ目を検出した．通常のn-gram解析は熟語に注目するが，このツールでは熟語にならない部分を扱う．

3. グルー検出アルゴリズム

本章では，はじめにアルゴリズムの概要の例をあげながら解説し，次に詳細なアルゴリズムを示す．

3.1 アルゴリズムの概要

Tを長さnのあるテキストとする．Tのi番目$(0 \leq i \leq n-1)$の文字から最後までの部分文字列を辞書順（昇順）に並べたものをS[0], S[1], ···, S[$n-1$]とする．ただし，テキストの終わりは最も大きな文字と見なす．たとえば，T=abccabcdefgbcdabcdeの場合，S[0]～S[18]は図**3**のようになる．

S[i]とS[$i+1$]が先頭から一致している文字数をlen[i]とする．ただし，len[$n-1$]は0とする．先の例の場合のlen[i]をS[i]と並べたものを図**4**に示す．

len[i]<len[$i-1$]となっているところに注目する．len[i]<len[$i-1$]ということは，S[$i-1$]の先頭からlen[$i-1$]文字までの(len[$i-1$])-gramがこれ以上ないということである．そして，その数は$(i-(i-1)+1)=2$個であることが分かる．len[i] < len[$i-2$]の場合も同様で，そのときの数は$(i-(i-2)+1)=3$である．つまりjを$(j<i)$とすると，len[i] < len[j]ならばS[j]の先頭からlen[j]文字までの(len[j])-gramはこれ以上存在せず，その数は$(i-j+1)$個である．

図4では$i=2$のときにlen[i]<len[$i-1$]となる．len[2]=0 < len[0] < len[1]であるから，S[0]の先頭から3 (=len[0])文字までの3-gram“abc”は，$2-0+1=3$個，S[1]の先頭から5 (=len[1])文字までの5-gram“abcde”は，$1-0+1=2$個あることが分かる．

このようにlen[i]を順に調べていくことで，テキストに現れるn-gramの出現回数を調べることができる．ある回数以上，かつ，ある長さ以上のn-gramは偶然以上の確率で現れたn-gramと判断し，熟語とする．この長さと回数は可変パラメータにしてある．そして，偶然以上の確率で現れたn-gramは意味のある塊であると推定する．

Tの熟語部分に印を付ける．印には熟語の端を表す‘E’と，熟語の内部を表す‘I’の2種類がある．‘E’は熟語の先頭と最後の文字の部分に，‘I’はそれ以外の部分に付けられる．ただし，先に‘I’が付けられてい

```
S[0]  =  abccabcdefgbcdabcde
S[1]  =  abcdefgbcdabcde
S[2]  =  abcde
S[3]  =  bccabcdefgbcdabcde
S[4]  =  bcdabcde
S[5]  =  bcdefgbcdabcde
S[6]  =  bcde
S[7]  =  cabcdefgbcdabcde
S[8]  =  ccabcdefgbcdabcde
S[9]  =  cdabcde
S[10] =  cdefgbcdabcde
S[11] =  cde
S[12] =  dabcde
S[13] =  defgbcdabcde
S[14] =  de
S[15] =  efgbcdabcde
S[16] =  e
S[17] =  fgbcdabcde
S[18] =  gbcdabcde
```

図3 S[i]の例
Fig. 3 The example of S[i].

i	len[i]	S[i]
0	3	abccabcdefgbcdabcde
1	5	abcdefgbcdabcde
2	0	abcde
3	2	bccabcdefgbcdabcde
4	3	bcdabcde
5	4	bcdefgbcdabcde
6	0	bcde
7	1	cabcdefgbcdabcde
8	1	ccabcdefgbcdabcde
9	2	cdabcde
10	3	cdefgbcdabcde
11	0	cde
12	1	dabcde
13	2	defgbcdabcde
14	0	de
15	1	efgbcdabcde
16	0	e
17	0	fgbcdabcde
18	0	gbcdabcde

図4 S[i]とlen[i]
Fig. 4 S[i] and len[i].

テキスト : abccabcdefgbcdabcde
印 : EIE EIIIE EIEEIIIE

図5 熟語部分のマーク例
Fig. 5 The example of marked parts of idiom.

る場合，熟語の先頭や末尾であっても'E'は付けない．T中のすべての熟語に対して印付を行う．先の例で3文字以上の2回以上現れたn-gramを熟語とした場合の印付結果を**図5**に示す．

図5を見て分かるように，印の付いていない部分がグルーである．また，'E'の印が付いているところは，どの熟語の内部にも含まれない部分であるから，'E'が連続しているところ（図5では14文字目のdと15文字目のa）は，熟語には含まれるが同時に含む熟語はない部分，すなわち熟語の切れ目である．

上記のアルゴリズムで一応の結果を得ることができる．しかしながら，出現頻度が非常に高い文字列があると，その後ろに同じ文字が付くということが，偶然に指定頻度以上に起こることがある．熟語の定義では，その文字列がある頻度以上出現した場合であるので，これは熟語であるが，この結果，グルーの検出力が落ちることが観測された．

これに対処するために，ある熟語の候補が他の候補と先頭から何文字一致しているかを調べる．そして，一致数が候補の文字列長とほとんど変わらない場合，一致部分だけを熟語の部分としてマークの処理をする．これにより偶然付いた文字（以下語尾と呼ぶ）をグルーとして検出できるように変更した．

3.2 アルゴリズムの詳細

グルー検出アルゴリズムを以下に示す．

(1) 入力データのすべての部分文字列についてソートを行う，すなわち，先頭 $+i$ 文字目で始まり最後（n 文字）までの部分文字列を辞書順に並べる．このとき，部分文字列はすべてポインタで表現し，本体を共有する．これにより，必要なメモリの量は，入力データに比例する量で済む．

(2) 配列 len を用意する．len[i] は辞書順で i 番目と $i+1$ 番目の部分文字列が先頭から一致する文字数を表す．

(3) i に0を代入する．
- (a) 辞書順で i 番目の部分文字列と $i+1$ 番目の文字列が先頭から何文字一致しているかを調べ，その値を len[i] に代入する．
- (b) i に1を加える．
- (c) $i < n-1$ ならば (3)-(a) へ．
- (d) len[$n-1$] に0を代入する．

(4) データ中に複数回現れる文字列で始まる部分文字列の辞書順での場所を表す配列 start と，現在の一致文字数を表す変数 level を用意する．

(5) i と level に0を代入する．
- (a) len[i] > level ならば
  - (i) start[level] に i を代入する．
  - (ii) level に1を加える．
  - (iii) (5)-(a) へ．
- (b) len[i] < level ならば
  - (i) 長さ level の，ある文字列が i−start[level]+1 個あることが分かる．この文字列の長さと個数が別に定める基準以上ならば，その文字列を熟語の候補と見なす．
  - (ii) 初めての候補ならば (5)-(b)-(v) へ．
  - (iii) 1つ前の候補と現在の候補が先頭から一致している数を調べ，その数と1つ前の候補の長さの差が一定値以下なら一致部分を，そうでなければ1つ前の候補を熟語とする．
  - (iv) 熟語は辞書順で start[a], start[a]+1, start[a]+2, $\cdots$, $i-1$ 番目の文字列の先頭部分である．これと長さからデータ中の熟語部分を求める．その熟語に含まれる各文字 c について，
    - (A) cが熟語の端の文字かつ，まだcの部分に印がなければ印'E'を付ける．
    - (B) cが熟語の端でなければ印'I'を付ける．
  - (v) level から1を引く．
  - (vi) (5)-(b) へ．
- (c) i に1を加える．
- (d) $i < n$ ならば (5)-(a) へ．

(6) テキストを印に基づいて出力し，グルーを示す．

(7) 終了．

3.3 計 算 量

このアルゴリズムのデータの長さ n に関する平均的な計算量について考える．比較回数の平均値を m とすると，部分文字列のソートは $O(mn\log n)$，一致数の計算のところは，一致数の最大値を m とすると $O(mn)$，熟語の候補を調べるのには $O(n)$，候補の不要部分を調べるのには $O(n)$，熟語にマークするのには

表1 コンパイラとunique_partによるエラー検出数
Table 1 The number of errors detected by compiler and uniqu_part.

言語	プログラム	コンパイラ	unique_part	検出できず
C	netlisp	912	53	35
FORTRAN	appbt	822	120	58
	appsp	745	209	46
	applu	729	208	53
KCL	ns-parser	150	821	29
emacs Lisp	calendar	65	678	257
	wnn-egg	167	651	182
	gnus	71	713	216

$O(mn)$の計算量がかかる．したがって，mを定数とすると，このアルゴリズム全体の計算量は$O(n \log n)$となる☆．

4. 評　価

本章ではunique_partの有効性を評価するため，まずunique_partの位置付けについて述べる．次に，unique_partを評価するための指標を導入し，この指標に関する実験結果を示す．また，unique_partでどのような誤りが検出できたかを例示する．次に，グルー率や実行時間といったunique_partの性質を示し，最後に評価のまとめを行う．

4.1 unique_partの位置付け

コンパイラは言語の文法上の誤りや，それぞれの計算機環境での実行に問題のある記述を検出する．しかしながら，文法に合致している誤りもあり，コンパイラはこれを検出することはできない．これを検出するためには，コンパイラ以外のツールが必要となる．unique_partはコンパイラの検査機能を補うプログラム検査ツールの1つであり，コンパイラの検査を終えたソースプログラムを対象に検査を行う．

しかしながら，unique_partは一般のプログラム検査ツールと異った性質を持つ．プログラム検査ツールの多くは，文法知識を利用してプログラムの構造を解析するが，unique_partは文法情報をまったく使用せず，検査対象のプログラムリストの情報のみで検査を行う．基本の考え方は，誤りは繰返しの構造を持たない部分にあることが多いという経験則である．このため，unique_partはプログラミング言語の文法に依存しないという特徴も生じる．そして，検出の方針がコンパイラとまったく異なることから，コンパイラで発見できない誤りの発見を助ける能力がある．

☆ 熟語を構成する文字を袋から任意に取り出したデータに対しては$m = \log n$となる．この場合の計算量は$O(n(\log n)^2)$である．

4.2 コンパイラとの併用実験

unique_partを評価する指標として，コンパイラで検出できなかった誤りを検出できた数を考える．この数を調べるために以下の実験を行った．

(1) 動作するプログラムのリスト中の任意の1文字を別の文字で置き換える．
(2) そのプログラムをコンパイルし，エラーが検出されるか調べる．
(3) エラーが検出されなければ，置き換えた文字がグルーとして検出されるか調べる．

実験では以上の手続きを1000回行い，コンパイラとunique_partでそれぞれいくつ検出できるかを調べた．置き換える文字には'a'を使用し，熟語条件の長さと出現頻度はそれぞれ5文字以上と2回以上とした．また，熟語候補の末尾の差が2文字以下の場合に，共通部分を熟語とした．実験に使用したプログラムはnetlisp（C言語），appbt，appsp，applu（以上FORTRAN），ns-parser（KCL），calendar，wnn-egg，gnus（以上emacs Lisp）の8つである．これらのプログラムを選んだ理由は，1つのファイルで数十キロバイト以上の量があることだけである．プログラムの内容を分析して選んだものではない．**表1**に実験結果を示す．

ここで，コンパイラの検出数はエラーだけでなく警告（Warning）を含めた数である．また，プログラム中のコメント部分は動作に影響しないので，あらかじめ削除してから実験を行った．

表1を見ると，CやFORTRANのような宣言の度合の強い言語の場合はコンパイラの検出する割合が大きく，逆にLispのような宣言の度合が弱い言語の場合はその割合が小さくなることが分かる．

unique_partの有効性を測定する尺度として**拾上げ率（gleaning率）**Gを導入する．Gはこの実験においてコンパイラで検出できなかったもののうち，unique_partで検出できた割合を表す．その定義を以下に示す．

表2 プログラミング言語と拾上げ率
Table 2 The gleaning ratio and programming language.

言語	プログラム	拾上げ率[%]
C	netlisp	60.2
FORTRAN	appbt	67.4
	appsp	82.0
	applu	76.8
KCL	ns-parser	96.6
emacs Lisp	calendar	72.5
	wnn-egg	78.2
	gnus	76.7
平均		76.3

$$G = \frac{D_u}{N - D_c} \times 100 \qquad (1)$$

ここで，N は置換えを行った総数，D_c はコンパイラが検出した数，D_u はコンパイラでは検出できず，unique_part で検出できた数をそれぞれ表す．表1の各プログラムにおける拾上げ率を**表2**に示す．

表2から，拾上げ率の平均は76.3%，最も低いケースで60.2%であることが分かる．ns-parserの拾上げ率が他のものよりも高くなっているのは，ns-parserが機械で自動生成された非常に規則正しいデータを内部に多く含むためである．

この結果から，コンパイラの検出できなかったエラーを安定して検出できることが分かる．また，表1，表2から，Lispのような宣言の度合の弱い言語において，unique_partは特に有効であるといえる．

4.3 グルー率

安定した拾上げ率であったとしても，プログラムの大部分がグルーとして検出されるならば，ツールとして有効ではない．プログラム全体の中でグルーが占める割合をグルー率と定義する．拾上げ率とグルー率の比がグルーに誤りが濃縮される程度を示すことになる．実験での置換えに関しては，拾上げ率に比べてグルー率は小さく，前出の拾上げ率を考慮すれば，このツールは検出に有効であると判定できる．プログラム中の拾上げ率とプログラムの大きさとグルー率を**表3**に示す．

ここで，熟語条件の長さと出現頻度などの条件は拾上げ率を求めたものと同一である．すなわち，熟語条件の長さと出現頻度はそれぞれ5文字以上と2回以上とした．また，熟語候補の末尾の差が2文字以下の場合に，共通部分を熟語とした．利用したプログラムは拾上げ率を求めたものと同一である．

表3に示されるように，どの条件でもグルー率は拾上げ率よりも小さく，ツールで検出されたグルーに注意を集中することで，より容易に誤りを発見できることが分かる．

表3 拾上げ率とグルー率
Table 3 The glue ratio and gleaning ratio.

プログラム	拾上げ率	プログラムサイズ	グルー率
netlisp	60.2%	237648 byte	1.05%
appbt	67.4%	138837 byte	0.65%
appsp	82.0%	82232 byte	1.22%
applu	76.8%	87128 byte	1.38%
ns-parser	96.6%	160799 byte	1.39%
calender	72.5%	107605 byte	4.00%
wnn-egg	78.2%	93789 byte	7.18%
gnus	76.7%	193015 byte	1.67%

表4 グルー検出の効果
Table 4 The effect of glue detection.

プログラム	サイズ [byte]	1文字が誤りである確率 全体	1文字が誤りである確率 グルー	比
netlisp	237648	4.21×10^{-6}	2.41×10^{-4}	57.33
appbt	138837	7.20×10^{-6}	7.47×10^{-4}	103.69
appsp	82232	1.22×10^{-5}	8.17×10^{-4}	67.21
applu	87128	1.15×10^{-5}	6.39×10^{-4}	55.65
ns-parser	160799	6.22×10^{-6}	4.32×10^{-4}	69.50
calender	107605	9.29×10^{-6}	1.68×10^{-4}	18.13
wnn-egg	93789	1.07×10^{-5}	1.16×10^{-4}	10.89
gnus	193015	5.18×10^{-6}	2.38×10^{-4}	45.93

誤りを発見する労力を概算するために，グルーの1文字を取り出したときに，それが誤りである確率と，テキストの1文字を取り出したときにそれが誤りである確率を推定する．実験において，誤りは1つだけという条件であるので，前者の推定値は 拾上げ率 ÷ (プログラムの大きさ × グルー率) であり，後者の推定値は 1.0 ÷ (プログラムの大きさ) である．これを表にまとめたものを**表4**に示す．これに示されるように，グルーの部分で誤りを調べるときに1文字の中に誤りが発見できる確率は，グルーを検出しないで先頭から誤りを調べるときに1文字の中に誤りが発見できる確率の10倍から100倍であることが分かる．

4.4 実行時間

大きなプログラムにおいて，検査にかかる時間は重要になる．特にパターンを検出する処理は実行時間が長いことが多い．したがって，実行時間に関する計測が必要になる．**表5**に各プログラムに対する実行時間を示す．大きなプログラムとして，Kcl（Lisp処理系），Emacsを例として示した．

表5の値はSPARC station 5（110 MHz，Memory 192 Mbyte）で実行したときのものである．

表5で示した実行時間は，それぞれのプログラムをコンパイルしたときにかかる時間と同程度である．このことから，大きなプログラムに対しては対話的に使

表5 実行時間
Table 5 The execution time.

プログラム	大きさ [byte]	時間 [sec] real	user	sys
autoref	33137	2.4	2.4	0.0
emacs	2066571	935.3	928.1	1.8
kcl	2904887	1141.2	1083.8	3.2

表6 実行プロセスの大きさ
Table 6 The memory size.

プログラム	大きさ [Kbyte]	プロセスの大きさ [Kbyte]
emacs	2018	24420
kcl	2834	39160
autoref	32	420

用することはできないが，開発が終了したときに，1度使用するという形態ならば問題にならない処理時間である．

4.5 実行プロセスの大きさ

実行時間と同様，パターンを検出する処理はメモリを消費することが多いので，実行に必要なメモリの量を計測する必要がある．実行プロセスの大きさを**表6**に示す．

表6より実行中のプロセスの大きさは，データの11～14倍程度であることが分かる．小さくはないが，記憶容量の必要量は$O(n)$なのでデータが大きくなってもプロセスが爆発的に大きくなることはない．

4.6 評価のまとめ

宣言の度合の弱い言語で書かれたプログラムは，コンパイラが検出する割合が小さいことと，unique_partの拾上げ率の平均が76.3%であることが分かった．このことから，宣言の度合の低い言語で書かれたプログラムにおいて，unique_partは特に有効であるといえる．グルーによって，エラーがピンポイントで特定できるのではないが，実験したプログラムでは，10倍から100倍以上の比率で誤りの可能性が濃縮されていることが分かった．unique_partの実行時間や実行時に消費するメモリもツールとして使用できる程度であることも示した．以上のことより，unique_partはプログラムを検査するツールとして有効であるといえる．

5. 考　　察

本章では，まずunique_partの特徴と限界を述べる．次にunique_partの応用として，自然言語で書かれたテキストへの適用についての問題点を報告し，最後にunique_partに言語依存情報を導入することについて，考慮した項目と，その問題点を述べる．

5.1 unique_partの特徴

unique_partは繰り返し現れる部分を正しいものとしてプログラムの検査を行う．このため，何度も繰り返し起こるとは考え難いタイプミスや編集ミスの検出は，unique_partが効果のある対象である．

unique_partはプログラミング言語の文法に依存しないことを特徴とする．したがって，データ部分の検査も行うことができる．また，複数の言語で書かれたプログラムを一度に検査することもできる．

5.2 unique_partの限界

プログラミングにおいて，すでに書いた部分をコピーして修正を加えるということはよくある．このような場合，コピーする元の部分に誤りが含まれていると，uinique_partは繰り返し現れる部分をプログラムとして正しい部分と認識するので，その誤りを検出することができない．このようにunique_partは繰り返し現れる誤りに弱い．このため，unique_partで検査しても，誤りがなくなったということを保証できない．

しかしながら，いろいろな検出の方式ごとに，効果がある対象が異なるのは当然である．そして，コンパイラを補う性質があるのでツールとしての存在価値がないような限界とはいえない．

5.3 自然言語への適用

n-gramは自然言語で書かれたテキストから特定の部分を取り出すために用いられてきた．そこで，unique_partの応用として自然言語で書かれたテキストへの適用が考えられるが，実際に行ってみると，熟語でない部分に注目するというアプローチは自然言語のテキストでは効果が得られなかった．

表7に新聞記事にunique_partを適用したときのグルーの数を示す．表7より，プログラムにおけるグルーの割合よりも，新聞記事における割合の方が高いことが分かる．これは，プログラムがいかに繰返しの性質が強いことを反映している．自然言語のテキストでは，これだけの大きさであってもグルーの割合が大きく，人間で検査する労力を軽減するとはいえない．このため自然言語に対してはunique_partを適用しても効果があるとはいえない．

5.4 言語依存情報の利用に関する検討

unique_partではいっさいの言語依存情報を用いずにプログラムの検査を行っている．しかし，言語に関する性質を導入することでグルーの検出効率を高めることができると考えられる．本節ではunique_partに導入できそうな言語依存情報について述べる．実際には，プログラミング言語独立を条件にしたので，これから述べる手法は評価を行わなかったが，実現するこ

表7 自然言語のグルーの数
Table 7 The number of glues in the natural language text.

ファイル	大きさ		グルー		
新聞記事	17558458	[字]	1849527	[字]	(10.53%)
(参考)KCL	2904887	[byte]	8800	[byte]	(0.30%)

とは容易であり，言語依存が許される状況では有用であると考えられる．

5.4.1 プログラミング言語の文法

プログラミング言語の文法，たとえばトークンの構造を導入することによって，誤りでないグルーを減らすのに役立つと考えられる．この構造は，解析をする前にトークンを処理することで実現できる．たとえば，自由書式の言語について，空白はプログラムの意味を変化させない．したがって，空白を無視するようにプログラムを変更することによって，パターンを検出する能力が上がり，グルー率を下げることができる．

5.4.2 予測される落し穴のパターンの記憶

パターンでない残りに注目するのが本稿でのアルゴリズムであったが，誤りである可能性が高いパターンを記憶しておき，そのパターンに合致したら，積極的に注意を促すような印を付ける機能を後処理としてアルゴリズムに追加することもできる．

この機能を導入すると，グルーのうち誤りである部分の割合を増加させることができると予想される．たとえば，C言語における条件部の代入などのよく知られた落とし穴に対して，ifのあとの代入に印を付けるような機能を付けることは可能である．ただし，これによって言語に非依存であるという性質が失われること，パターンの与え方によっては不必要にグルーが増えてしまう可能性があることに注意が必要である．

5.4.3 模範的なプログラムにおける熟語

あらかじめ模範となるいくつかのプログラムに対して処理を行って，出現頻度の高い熟語を予約熟語として記憶することが考えられる．そうすれば，プログラムの検査を行う際に，この予約熟語に該当する部分を熟語としてマークすることで，1つの関数からなるような短いプログラムにおいても，グルーを絞り込むことができるようになる．これはツールに辞書を持たせると考えてもよいし，機械学習を行うものともいえる．

この方法では，模範となるプログラムをどの基準で選ぶかが問題となる．広範囲の模範プログラムを記憶すると，グルーが減る代わりにプログラムの誤りを検出する能力も減る．また，狭い範囲の模範となるプログラムを考えると，そのプログラムと対象のプログラムの適合度が問題となる．模範プログラムが適切かどうかを確かめる方法は，実際にプログラムを検査したときの性能を調べるしかないが，これは検査を行う前に調べることは難しい．

6. 関連研究

6.1 文法に従ってプログラムを検査するツール

言語定義上では正しくてもエラーは存在する．たとえば，宣言してありながら使用していない変数などは，プログラム上は無害でありながらも何らかの問題があると考えるのが自然である．

このような状況を検出するためには，コンパイラが警告も与えるという方針が一般的であり，多くの商用システムが作成されている．これが，現在のプログラミングにおけるチェックツールの標準となっている．

さらには，検査専用のツールもある．よく知られるようにC言語にはlintのようにコンパイラで検出できないエラーに対応するプログラムの検査ツールがある[5]．また，プログラミング上の経験や，よく生じる誤りをターゲットにプログラムを検査するシステムもある[6),7]．これらもCのプログラムを検査して，誤りや移植性に欠ける箇所，無駄な記述等を検出する強力な検査ツールである．

これらのプログラムは，基本としてプログラミング言語の構造に従って構文を解析するのが処理の最初のステップである．構文を知っているので，このような検査ツールにおいては，エラーの存在の場所を狭い範囲に特定できる．したがって，unique_partに比べて，指摘する部分にエラーが存在する確率は高い．エラーの検出率と検出能力は，unique_partに比べて高い．

しかしながら，どのようなツールも完全ということは難しい．「他に類似のない場所に注意する」という，新しい別の方針を使用することで，そこで発見できない可能性を拾い上げるところにunique_partの存在意味がある．文献にあるツールは対象の文法に基づいて検査しているので，C言語で書かれたプログラムしか検査できない．これに対して，unique_partはプログラミング言語の文法に依存しないので，さまざまな言語で書かれたプログラムを検査できる．

6.2 プログラミング言語非依存のツールとの比較

プログラミングのための言語に依存しないツールとしては，ソースプログラムの履歴管理などのツールが商用システムでは使われる．また，設計の決定過程や，

機能設計の情報を保持するシステムなども提案されている．それらはコーディングの前の段階の作業を支援するシステムである．それゆえ，プログラミング言語には依存しないのであるが，uniqu_part のようにコーディングを終了したあとのプログラム検査の支援としているものの報告はあまりない．

文法情報を利用するときには，コンパイラの技法を利用すればよい．そして，コンパイラの作成方法は標準的な技法であり，あえて言語の文法情報に依存しない形態でプログラムを入力とするシステムは，その性能が低いと考えられがちである．このように，文法を使うことが簡単であるという理由で，実験的に公表されないで作成された可能性はあるが，標準的なプログラミングの検査ツールで，対象言語の構造を扱わないものはない．

6.3 プログラミングにおけるパターン

以前から，プログラムを記述するときに，あるパターンに従って記述するという考え方は存在した[8]．そして，近年，プログラミングの中で，積極的にパターンを扱うという手法[9]がある．これは，多くのプログラムの中から，頻繁に出現するパターンを抽出して，プログラムの再利用に役立てることを目標としている．パターンを積極的に取り上げていくという考え方は，自然言語の n-gram の解析による言語分析と同様の考え方である．

しかし，上記で扱っているパターンは，言語の構造まで立ち入ったパターンであり，unique_part で扱っている文字のパターンとはパターンのレベルが異なる．また，構造を持たない部分に意味を持たせるという考え方は，上記の中には存在しない．

6.4 ノイズを分析対象とする研究

定型的でない部分に注目するというパターン処理は，特徴のあるものである．このアプローチの背景にはデータベースの情報をパターン処理して，背後に隠れている情報を取り出すというデータマイニング[10),11)]の処理がある．そこでは，データの中に混入するノイズからも，意味のある情報を検出しようとしている．unique_part におけるデータ処理は，この枠組みに従ったものである．

6.5 n-gram を用いた欠損の検出の研究

基本のアルゴリズムは文献 12) に記述されているが，ここでは有効性の評価がなされておらず，アルゴリズムだけの記述となっている．また，文献 13) は実例を並べることでアルゴリズムの有効性を述べたものであり，検出効果を定量的に評価するということは行っていない．一方，本稿では定量的評価を行っている．

7. ま と め

n-gram 解析を用いてプログラム中の欠損を検出する手法を提案し，この手法を用いたプログラム言語に依存しないプログラミングツール unique_part を作成した．

コンパイル可能なプログラムリスト中の任意の 1 文字を変更するという誤りに対して，unique_part はコンパイラが検出できなかったものの約 7 割を検出した．また，C 言語のような宣言の度合の強い言語よりも Lisp などの宣言の度合が弱い言語の方が，コンパイラで検出される数が少なかった．このことから unique_part は宣言の度合の弱い言語に対して特に有効であるといえる．実行時間や実行プロセスの大きさ，プログラム中のグルーの割合もプログラミングツールとして使用できる程度であった．

しかしながら，コンパイラで簡単に検出できる誤りでも，複数回現れると unique_part では検出できない．このため，unique_part はコンパイラの誤り検出の機能に置き代わるものではない．

これらのことから，unique_part をコンパイラのエラー検出機能を補うツールとして評価した．1 文字が置き換わるというタイプミスを想定した状況で，エラーが発見できる確率を推定した．そして，コンパイラが検出しない誤りを検査するツールとして，効果があることを示した．

今後の課題としては，表層的でない誤りにも検出効果があるかなど，誤りの性質の範囲を広げて，さらに評価を深めることがあげられる．また，熟語検出条件の文字列の長さと出現頻度を変更すると異った結果が得られる．これらの条件の最適な値を求めることも今後の課題である．

謝辞 本研究は NTT 基礎研究所の工学研究育成の援助を受け，さらに住友電気電気工業および NTT 基礎研究所との共同研究の成果である．また，東京大学の岩崎英哉氏には，長さ 0 のグルーの存在について指摘をいただいた．また，的を得た査読により，評価データの客観性が改善された．査読の労をとっていただいた方々に謹んで感謝する．

参 考 文 献

1) 長尾 眞，森 信介：大規模日本語テキストの n グラム統計の作り方と語句の自動抽出，情報処理学会研究会報告自然言語処理，96-1, pp.1–8 (1993).
2) 長尾 眞，森 信介：n グラム統計によるコー

パスからの未知語抽出，情報処理学会研究会報告自然言語処理，108-2, pp.7–12 (1995).
3) 下畑さより，杉尾俊之，永田淳次：隣接文字の分散値を用いた定型表現の自動抽出，情報処理学会研究会報告自然言語処理，110-11, pp.71–78 (1995).
4) 中渡瀬秀一，木本晴夫：統計的手法によるテキストからの重要語抽出メカニズム，情報処理学会研究会報告情報学基礎，39-6, pp.41–48 (1995).
5) Johnson, S.C.: *Lint, A C Program Checker*, 1st edition, Unix Use's Manual, Vol.1 (1986).
6) 青木圭子，瀧塚孝志，橋本和男，小花貞夫：C言語プログラム検査ツールの実装と適用結果，情報処理学会研究会報告ソフトウェア工学，100-9, pp.63–70 (1994).
7) 掛下哲郎，小田まり子：正規表現によるCプログラムの落し穴検出ツール，情報処理学会研究会報告ソフトウェア工学，86-7, pp.43–49 (1992).
8) Kernighan, R. and Plauger, P.: *The elements of Programming Style*, McGraw-Hill (1974).
9) Gamma, E., Helm, R., Johnson, R. and Ulissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley (1995).
10) Adriaans, P. and Zantinge, D.: *Data Mining*, Addison Wesley Longman (1996).
11) Fayyad, U.M., Piatetsky-Shapiro, G., Smyth, P. and Uthurusamy, R. (Eds.): *Advances in Knowledge Discovery and Data Mining*, AAAI Press/MIT Press (1996).
12) 吉川裕之，貴島寿郎，梅村恭司：n-gram解析を用いたプログラム中の非定型パターン・欠損の検出，情報処理学会プログラミングシンポジウム (1997).
13) 吉川裕之，貴島寿郎，梅村恭司：n-gram解析を用いたプログラム中の非定型パターン・欠損の検出，情報処理学会研究会報告プログラミング，15-2, pp.9–15 (1997).

(平成10年 3 月 20 日受付)
(平成10年10月 2 日採録)

吉川 裕之（正会員）

1973年生．1996年豊橋技術科学大学工学部情報工学課程卒業．1998年同大学大学院工学研究科情報工学専攻修士課程修了．同年，日本電信電話（株）入社．電子図書館システム関連業務に従事．データマイニングに興味を持つ．

貴島 寿郎

1987年佐世保高等専門学校電気卒業．1989年豊橋技術科学大学情報工学課程卒業．1991年同大学大学院修士課程情報工学専攻修了．1995年同大学情報工学系教務職員．1997年逝去．研究分野は超並列計算機，プログラミング言語，言語処理系等．

梅村 恭司（正会員）

1959年生．1983年東京大学工学系研究科情報工学専攻修士課程修了．同年，日本電信電話公社電気通信研究所入所．1995年豊橋技術科学情報工学系助教授，現在に至る．博士（工学），システムプログラム，記号処理の研究に従事．ACM，ソフトウェア科学会，電子情報通信学会，計量国語学会各会員．