パターンマッチングに基づいたCプログラムの落し穴検出法

小田 まり子† 掛下 哲郎††

C言語は柔軟な構造を持ち，コンパクトで高速なプログラムを書くことができる．しかし，コンパイルは成功しても構文上の些細な相違等に由来する誤記（落し穴）により，実際の動作がプログラマの意図と異なる場合がある．我々は，Cプログラムの落し穴を検出するためのソフトウェアツール Fall-in C を開発している．既存の落し穴検出ツールとして lint 等が存在するが，Fall-in C は，エディタ Emacs 上で実行されるため，ツールの呼び出しから落し穴の表示，修正に至る一連の動作がエディタ上で即座に行える．また，落し穴の検出にパターンマッチングを用いているため，アドホックな落し穴項目に対する拡張性が高い．本稿では Fall-in C の落し穴検出法を示して，その評価を行う．Fall-in C は，正規表現検索と構文パターンマッチングを併用して落し穴を検索する．落し穴検出に正規表現検索を用いた場合，構文的な落し穴検出において誤検出が生じるが，字句解析上の落し穴は高速に正しく検出できる．また，構文パターンマッチングを用いた検出は字句解析上の落し穴のいくつかを検出できないが，構文的な落し穴検出における誤検出は生じない．180個（約2.9 MB）のファイルを用いて本落し穴検出法を評価したところ，16項目の落し穴を正しく検出できた．

Pitfall Detection of C Programs Using Pattern Matching

Mariko Oda† and Tetsuro Kakeshita††

Programming Language C has a flexible structure and its compiler generates compact and efficient object codes. However the compiler cannot detect some types of bugs hidden in the program. We are developing a software tool Fall-in C to detect such types of bugs, or pitfalls, in C programs. In contrast with lint, Fall-in C is executed within GNU Emacs editor in order to enable a programmer to correct the detected pitfalls immediately. Furthermore Fall-in C uses pattern matching for pitfall detection so that programmers can augment the detectable pitfall types by adding the corresponding patterns. Fall-in C uses both regular expression searching and structural pattern matching in order to detect pitfalls in C programs. Regular expression can efficiently detect lexical pitfalls while it detects erroneous pitfalls during syntactic pitfall detection. Although structural pattern matching requires syntactic analysis and thus cannot detect certain types of lexical pitfalls, it can properly detect syntactic pitfalls. We evaluated the tool using 180 C source files (2.9 MB total) and demonstrated that Fall-in C correctly detects 16 types of pitfalls.

1. はじめに

C言語はプログラムをコンパクトにするために冗長度の低い文法を用いており，構文上のわずかな違いによって全く意味の異なるプログラムができあがる場合がある．この場合，C言語の文法に合致していればコンパイラは何のエラーメッセージも出さない．このように，コンパイルは成功しても，構文上の些細な相違などに由来する誤記によりコンパイラによって検出できないバグを落し穴と呼ぶ[1]．

このような落し穴を検出するためには，コンパイラ以外のツールが必要であり，UNIX には lint プログラムが用意されている[2]．lint は，主に複数のファイル間での定義と参照の矛盾検出や，プログラムの意味的な落し穴検出に優れているが，字句解析上の落し穴や構文的な落し穴の検出に関しては改良の余地が大きい．また，lint はユーザインタフェースが貧弱なため，不適当なメッセージを出力することがあり，落し穴に対応するソースコードを即座に変更することは難しい．

我々は，このような問題点を解決するために，Cプログラムの落し穴検出ツール Fall-in C を開発している[3)~6)]．利用者は，Emacs エディタ[7]を用いたCプログラミングの途上で本ツールを利用することができ

† 久留米工業大学工学部電子情報学科
Department of Information Science and Electronics Engineering, Faculty of Engineering, Kurume Institute of Technology
†† 佐賀大学理工学部情報科学科
Department of Information Science, Faculty of Science and Engineering, Saga University

表 1 Fall-in C が検出する落し穴項目
Table 1 List of pitfalls detected by Fall-in C.

項目番号	落し穴項目	説明
1	条件判定部における代入	**if, while, for** 文の条件判定部が代入式の場合に指摘する.
2	曖昧な表現	`-+++-`, `+-------+`, `+=-1` のように曖昧さを含んでいる部分を指摘する.
3	入れ子コメント	コメントの中にコメントの始まりを表す `/*` が現れる場合に指摘する.
4	整数定数における基数	ファイル中に出現する 8 進数を指摘する.
5	文字列と文字定数	**printf** 中での単一引用符で囲まれた第一引数を指摘する.
6	演算子の優先度 (代入と比較)	一つの式の中に代入と比較の演算子が現れる場合に指摘する.
7	演算子の優先度 (&演算子と比較)	一つの式の中に&演算子と比較の演算子が現れる場合に指摘する.
8	演算子の優先度 (シフトと算術演算子)	一つの式の中にシフトと算術演算子が現れる場合に指摘する.
9	演算子の優先度 (代入と関係演算子)	一つの式の中に代入と関係演算子が現れる場合に指摘する.
a	演算子の優先度 (代入演算子)	一つの式の中に代入演算子が二つ以上現れる場合に指摘する.
b	余分なセミコロン	**if** 文 や **while** 文の条件節の括弧の後にセミコロンがある場合に指摘する.
c	**return** の後の不足したセミコロン	**return** 文の直後にセミコロンがない場合に指摘する.
d	**struct** の後の不足したセミコロン	関数定義の直前の **struct** 宣言の終りにセミコロンがない場合に指摘する.
e	**break** のない **case** ラベル	**switch** から抜け出るための **break** 文等がない場合に指摘する.
f	引数リストのない関数呼び出し	引数リストのない関数呼び出しは何も行なわないので, これを指摘する.
g	ぶらさがり **else**	ぶらさがり **else** が出現する場合, 対応する **if** 文を指摘する.

る．本ツールは編集中のCソースファイルを解析し，落し穴が存在すればそれを表示する．また，ソースファイルの落し穴箇所にカーソルを移動し，利用者にその位置を知らせる機能もある．エディタ内部での処理なのでその場で即座に落し穴を修正できる．

Fall-in C は落し穴の検出方法として，正規表現検索と落し穴パターンを用いた構文パターンマッチングを採用している．パターンマッチングに用いる正規表現や落し穴パターンは容易に追加や変更が行えるため，Fall-in C は保守性や拡張性に優れている．落し穴は本質的にアドホックなものであるため，新たな落し穴項目に対する拡張性は落し穴検出ツールにとって，非常に重要な機能である．

パターンマッチングに基づいた落し穴検出によって，字句解析上の落し穴と構文的な落し穴が検出できる．正規表現を用いたパターンマッチングは，構文的な落し穴検出において誤検出が生じるが，字句解析上の落し穴は高速に正しく検出できる．また，検出対象となるCプログラムの構文解析木に対して落し穴検索を行う構文パターンマッチングは，字句解析上の落し穴を検出できない場合があるが，構文的な落し穴の誤検出は生じない．本稿では，これらの落し穴検出法について議論する．

本稿は，次のように構成されている．2章では，パターンマッチングに基づいた落し穴検出法の検索対象となる16項目の落し穴を列挙した後，Fall-in C の全体構成を説明する．3章では，正規表現を用いた落し穴検出法について説明し，4章では，正規表現検索の限界を指摘する．5章で構文パターンマッチングを用いた落し穴検出法の概要を示した後，6章では落し穴パターンの定義を行い，その後，いくつかの落し穴に対応する落し穴パターン例を紹介する．7章では，本検出方式の評価および考察を行う．

2. Fall-in C の全体構成

パターンマッチングに基づいた落し穴検出法は，字句解析上の落し穴と構文的な落し穴を検出するために汎用的に利用できる．これを示すために，文献1)にこの種の落し穴として列挙されている16項目の落し穴を検索対象とした．**表1**に落し穴の一覧を示す．

Fall-in C は，Emacs エディタ上のCプログラム編集画面から直接呼び出すことができる（**図1**）．ツールを起動すると，ウインドウが三分割される．それぞれのウインドウには，Cのソースプログラム，落し穴を表示する Search バッファ，落し穴項目を表示する Summary バッファが割り当てられる．利用者が右下のウインドウに表示された Summary バッファから検出項目を選択すると，落し穴検出が実行され，その一覧が左下の Search バッファに表示される．Search バッファは落し穴へのメニューにもなっており，Cプログラムの対応する部分にジャンプすることができる．従って，落し穴の検出後，その場で直ちに修正ができる．また，ツールの使用方法や検出項目についてのドキュメントも参照できる．

3. 正規表現を用いた落し穴検出

3.1 正規表現検索の概要

正規表現パターンマッチングによる落し穴検出の流れを**図2**に示す．正規表現パターンマッチングによる落し穴検出では，最初に対象となるCプログラムを別

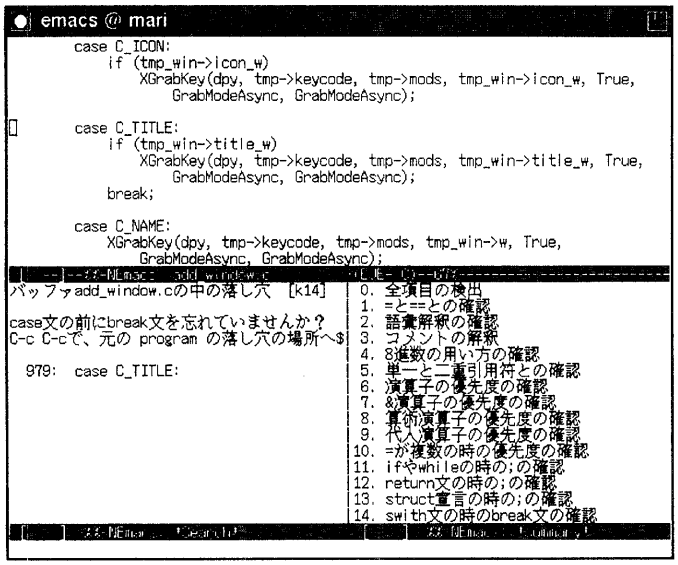


図 1 Fall-in C における落し穴検出画面
Fig. 1 Pitfall detection by Fall-in C.

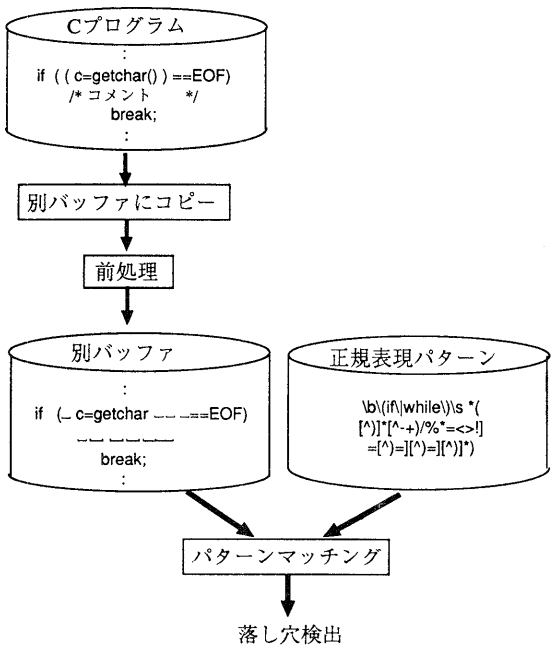


図 2 正規表現を用いた落し穴検出
Fig. 2 Pitfall detection using regular expression.

バッファにコピーする．次に，コピーされたファイルに対して，落し穴の検出項目に応じた前処理を行う．その後，項目ごとに用意した正規表現とのパターンマッチングを行うことで落し穴の検索を行う．

3.2 正規表現検索における前処理

Cプログラムにおいて，コメントは空白あるいは改行が書ける任意の場所に挿入できる．コメントの中には任意の文字列を書くことができるが，このことを考慮して正規表現を設計することは不可能である．この問題を解決するため，正規表現検索を行う前に，コメントを空白に置換する．削除を行わないのは，オリジナルのCプログラムとの間でポイント位置が変化しないようにするためである．

文字列や文字定数においてもコメントと同様の問題が生じる．そこで，検出を行う前に文字列や文字定数も別バッファ上で空白置換する．

3.3 Emacs における正規表現の構文

Emacs の正規表現において，特別な文字がマッチするパターンを以下に示す．

- ・ 改行以外の任意の一文字
- ＊ 直前の正規表現の0回以上の繰り返し
- ＋ 直前の正規表現の1回以上の繰り返し
- .$ 行の先頭および終端
- […] 文字集合‘…’に含まれる任意の一文字．集合の中に文字範囲を書くこともできる．
- [^…] 文字集合‘…’に含まれない任意の一文字
- \b 単語の切れ目
- \c cが特殊文字である場合，その文字自体
- \sc cが空白の時はスペース文字，wの時は単語構成要素
- \| 複数の正規表現からの選択
- \(…\) 正規表現のグループ．これによって，‘\|’の適用範囲が定められ，複雑な表現に対する‘*’が使えるようになる．

通常，正規表現による検索は行単位で行われるが，[^…] 構文を使うと改行コードもマッチする対象となるので，複数行にわたる落し穴の検出も可能になる．

3.4 落し穴を検出する正規表現

本節では，落し穴を検索するための正規表現を紹介する．

●曖昧な表現の落し穴

```
\([^<-]---+[^>-]\|\+\+\+\++
\|[^-+*/=><%!]\(=-\|=\+\)[0-9]+\)
```

この正規表現は [^〈−]−−−+[^〉−], \+\+\++, または [^−+*/=〉〈%!]/(=−\|=\+\)[0−9]+ の三つの正規表現のいずれかにマッチするパターンを意味する.

最初の [^〈−]−−−+[^〉−]は,−が三つ以上続き,その前後に−以外の文字があるパターンにマッチする.ただし,ポインタを表す際に用いられる〈−や−〉にマッチしないように,前後の条件に〈や〉を加えている.\+\+\++は,+が三つ以上続くパターンにマッチする.[^−+*/=〉〈%!]\(=−\|=\+\)[0−9]+は,=−や=+の後に整数が現れるパターンにマッチする.ただし,[^−+*/=〉〈%!]は,=−や=+の=が単独の=で出現するための制約を与えるものである.

- 入れ子コメント

  /*\([^/*]\|*[^/]\)*/*

  この正規表現は,/*で始まり,/および*以外の文字,または,*/以外の文字列が現れた後,/*が現れるパターンにマッチする.
- 整数定数における基数

  [^A−Za−z0−9_.]0[0−9]+
- 文字列と文字定数

  \bprintf\s␣*(\s␣*'.'\s␣*)
- return の後の不足したセミコロン

  \breturn\s␣*$

  これは,return の後にセミコロンが現れず,改行されているパターンにマッチする.

曖昧な表現,入れ子コメント,return の後の不足したセミコロンの落し穴は,構文解析を行うとトークン解釈やコメント削除が行われるため検出できない.

4. 正規表現検索における問題

本章では,構文的な落し穴検出において正規表現を用いた場合に生じる問題について述べる.

4.1 複雑な前処理や検出処理

字句解析上の落し穴は,3章で述べた前処理のみを行えば,正規表現を用いた落し穴検出が簡単に行える.しかし,構文的な落し穴を検索する場合,これだけでは十分でなく,項目ごとに異なる処理を考えねばならない.例えば,正規表現では入れ子の括弧が表現できない.このため,入れ子の括弧の空白置換処理や入れ子の括弧単位の再帰的な検索処理が必要になる.また正規表現では文字列の否定が十分に表現できないので,do-while 構文でない while 文や,break 文のない case 文を簡単に検索できない.この問題を解決するためには,必要に応じた削除の前処理を行ったり,否定を含む正規表現を検索するための関数を作成しなければならない.このように構文的な落し穴を検索する場合,落し穴検出を統一的に行うことが困難である.

4.2 検出速度

字句解析上の落し穴検出において,正規表現を用いると非常に高速な検出が行える.しかし,構文的な落し穴検出に正規表現を用いると検出に時間がかかる.これは,Emacs の正規表現検索は最長一致法を用いているため,不必要なバックトラックが生じる場合があるからである.例えば,構文的な落し穴を検出するためには,\s␣* や [^…]* をしばしば用いるが,これらの表現は,'*' や [^] に起因するパターンマッチング中のバックトラックが多発するため,検出速度が低下する.

4.3 誤検出

字句解析上の落し穴検出に正規表現を用いた場合,正しい検出が行われる.しかし,構文的な落し穴の検出を試みたところ,以下のような誤検出(落し穴でない部分を落し穴として検出する現象)が生じた.

- 条件判定部における代入

  前処理として行う入れ子の括弧の空白置換は,正規表現検索を可能にはしたが,誤検出の原因にもなった.なぜなら,この処理によって,条件判定部における代入の落し穴である if(x=getchar()!=EOF) と落し穴でない if((x=getchar())!=EOF) が区別できないためである.
- break 文のない case ラベル

  break; と case の間に '}' が出現した場合,誤検出が生じた.これは,以下の二つを正規表現で識別するのが非常に困難なためである.

```
case… : {              case… : …
      :                    if(… ){ …
    break;                   break;
  /*誤検出*/            /*落し穴*/
  }                      }
case…                  case…
```

- 引数リストを持たない関数呼び出し

  正規表現検索では型の情報が得られないので,以下のような場合の識別は不可能である.

```
int foo(), bar();
bar(){
```

```
    int foo, fee ; /* foo は落し穴でない */
    fee=foo ;
    fee=bar ;  /* bar は落し穴 */
}
```

● ぶらさがり else の落し穴

ぶらさがり else のような構文的な落し穴は簡単な正規表現検索では検出できない．そこで，ぶらさがり else の検出のために専用関数を用意する．まず，正規表現検索を用いて else を検索する．これによって検索された else と対になる if をソースプログラムの先頭方向に向かって探すが，if の実行部が複文ならば，{…} の中をスキップする．if が見つかれば，同様の方法で余分な if があるかどうかを検索する．ここで if が存在すれば，ぶらさがり else が検出できたことになる．しかし，この方法では以下のような場合の識別ができない．

```
if( )                    if( )
    if( ){                   if( ){
    …                            …
    }                        }
else/* 落し穴 */         else/* 誤検出 */
    …                            …
                         else
```

これらの誤検出を解決するためには，検出対象プログラムから構文情報を抽出しなければならない．「引数リストを持たない関数呼び出し」の落し穴については，さらに型情報の抽出が必要になる．本稿では，型情報などの意味的情報の抽出は必要としないタイプの落し穴について，構文パターンマッチングによる落し穴検出を行う．

5. 構文パターンマッチングによる落し穴検出

前章で述べた問題点を解決するために，構文的な落し穴の検出には構文解析処理を取り入れる．構文パターンマッチングによる落し穴検出の流れを**図3**に示す．

字句解析処理および構文解析処理は Emacs Lisp[8] によって実現されている．プログラム「if (c=EOF) break ;」に対して，これらの処理を実行すると，**図4**に示すリストが返される．

このリストの要素は，トークンの意味，ポイント位置，トークン自身に対応している．ただし，演算子を表すトークンについては，優先度を表す1から15までの数字と結合規則 (right or left) も示す．ポイント位置は，検出された落し穴に対応するプログラムテキストへ移動するために利用する．構文解析に用いるCの文法規則は，ANSI C[9]の文法を用い，トップダウンの構文解析を行っている．このため部分的な構文解析も可能である．

図4のリストは**図5**の構文解析木に対応している．

この構文解析木と落し穴の項目ごとに用意した落し穴検出用構文木（落し穴パターン）とのパターンマッ

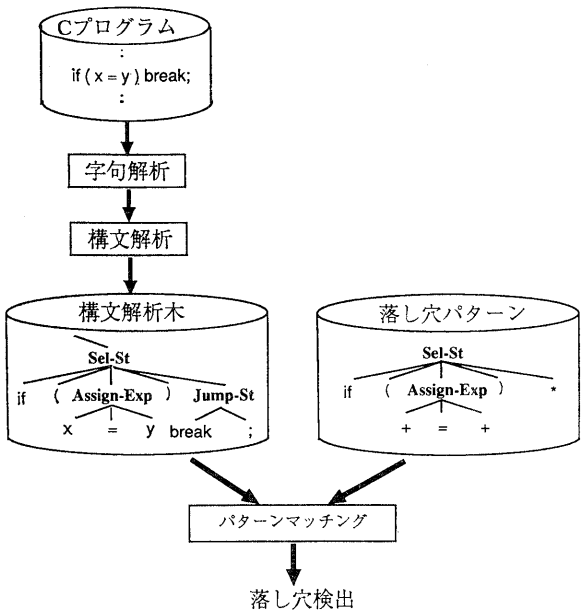


図3 構文パターンマッチングを用いた落し穴検出
Fig. 3 Pitfall detection using structural pattern matching.

```
((selection-statement (1 18)
   (identifier (1 3) if)
   (open-bra (4 5) "(")
   (assignment-expression (5 10)
      (identifier (5 6) c)
      (op (6 7) ("=" 14 right))
      (identifier (7 10) EOF))
   (close-bra (10 11) ")")
   (jump-statement (12 18)
      (identifier (12 17) break)
      (semi-colon (17 18)))))
```

図4 構文解析結果のリスト
Fig. 4 A sample parsing list.

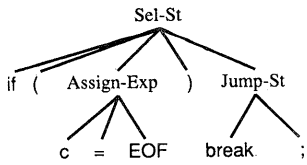


図 5 構文解析木の一例
Fig. 5 A sample parse tree.

チングによって落し穴の検出を行う．複数の落し穴パターンを用意することで，種々の落し穴を検出できる．

6. 落し穴パターン

本章では，構文パターンマッチングに用いる落し穴パターンについて説明する．6.1 節では，落し穴パターンの構文およびパターンの特殊記号にマッチする構文解析木を定義する．6.2 節では，落し穴パターン例のいくつかを示す．残りは，付録に示す．

6.1 落し穴パターンの定義

落し穴パターンは，Cプログラムの構文要素 (if, '(', Sel-St, Assign-Exp 等)，および特殊記号 (*, +, !, \|, *ignore, +this) を節点とする木である．これらの特殊記号のうち*, +, +this は引数を持たない．! はただ一つの引数を持ち，引数の木は落し穴パターンの定義を満足する．\| は，二つ以上の引数を持ち，それぞれの引数の木は落し穴パターンである．*ignore は二つの引数を持つ．左（最初）の引数の木を ignore 木と呼ぶ．ignore 木は，引数に+this を少なくとも一つ含み，*ignore を含まない落し穴パターンになっている．また，右（2番目）の引数の木を目的木と呼ぶ．目的木は落し穴パターンの定義を満足する木である．+this は ignore 木の中でのみ使われる記号であり，それ以外の部分木および引数の中には出現しない．これらの定義により，+this に対応する *ignore が唯一に定まる．

以上のように定義された落し穴パターンの特殊記号は以下に示す構文解析木とマッチする．

＊ 0個以上の任意の構文解析木にマッチする．ただし＊は木の根には出現しない．

＋ 一つの任意の構文解析木にマッチする．

！ 引数 P_1 とマッチしない構文解析木とマッチする．

\| 引数として持つ $P_1, \cdots, P_m$ のいずれかとマッチする構文解析木にマッチする．

*ignore 次のいずれかの落し穴パターンとマッチする構文解析木とマッチする．

1. 目的木
2. ignore 木の+this 節点を ignore 木で置き換える操作を有限回繰り返す．これによって生成された落し穴パターンの +this節点を目的木で置き換えたパターン．

*ignore は，連結な木構造では与えられない落し穴パターンを表現するために定義している．ignore 木の部分は，パターンマッチングにおいて無視される部分である．最終的な検索パターンは目的木なので，ignore 木中で目的木が出現する箇所を +this 節点で記述している．+this 節点の場所を確定するために ignore 木も正確に定義する必要がある．

+this 節点は ignore 木中にのみ出現し，最終的には目的木で置き換えられることによって落し穴パターンから消去される．従って，+this 節点にマッチする構文解析木は特に定義しない．

6.2 落し穴パターン例

図 6 は条件判定における代入の落し穴を検出するた

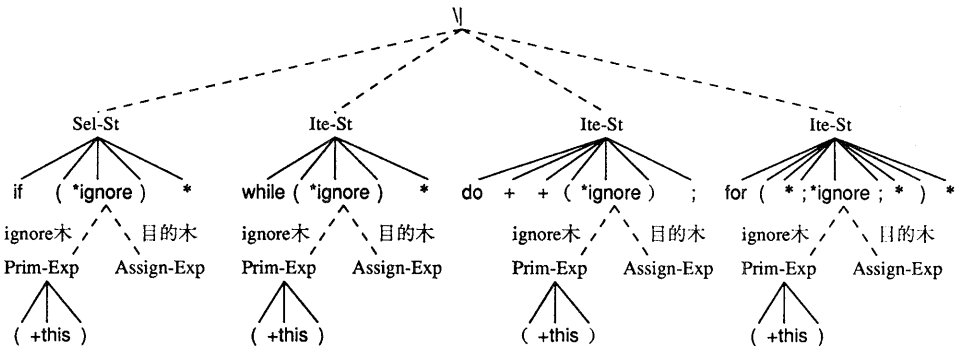


図 6 条件判定部における代入の落し穴パターン
Fig. 6 The pitfall pattern of assignment expression within conditional part.

めの落し穴パターンである．Cの条件判定文には，選択文の if 文および，反復文の while 文，do-while 文，for 文の四つの構文がある．従って，この四つの構文の選択を表すために落し穴パターンの根は \| となる．その下の四つの部分木は，それぞれの構文における条件判定部の式が代入である木を意味している．ただし，*ignore を用いることにより，代入式が余分な括弧に囲まれている場合も考慮している．

これまで誤検出されていたプログラム

```
if ((c = getchar()) !=EOF)
```

の構文解析木は**図7**になるため，この落し穴パターンにはマッチしない．従って，誤検出は発生しない．

図8は break のない case ラベルの落し穴を検出するための落し穴パターンである．これは，一つの case が終了した後，抜け出る手段であるジャンプ文(goto, continue, break, return, exit) が現れず，次の case が現れるような場合に落し穴として検出する．しかし，以下のように case ラベルが複数個付いているジャンプ文は，落し穴とみなさない．

```
case 'A':
case 'B':
case 'C':
        :
    break;
```

同様に，以下のように case ラベルの付いた文が複文であり，その最後の文がジャンプ文である場合も落し穴とみなさない．

```
case 'E':{ …
          :
      break;
}
```

図8では *ignore を用いてこの2点を考慮している．

図9はぶらさがり else の落し穴を検出するための落し穴パターンである．この部分木の根の選択文は else が省略された if 文でなければならない．逆に，この if 文において，条件が真であった場合の実行文は else 部を持つ if 文でなければならない．*ignore は，

```
if( )
  for( ; ; )
    if( ){
      …
    }
```

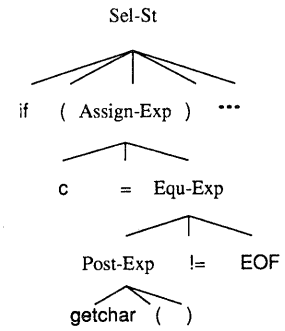


図7 代入を正しく用いた if 文の構文解析木
Fig. 7 A parse tree of correct statement with assignment expression.

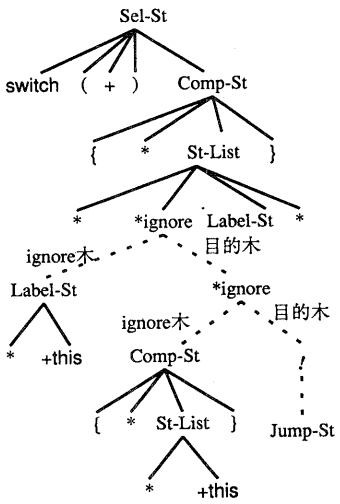


図8 break 文のない case ラベルの落し穴パターン
Fig. 8 The pitfall pattern of case labels without break statement.

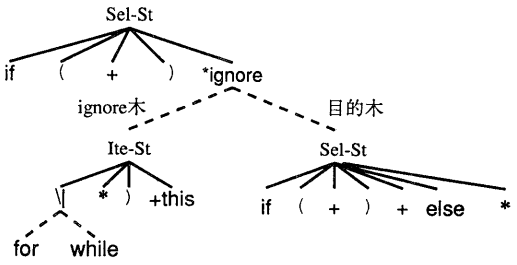


図9 ぶらさがり else の落し穴パターン
Fig. 9 The pitfall pattern of dangling else.

```
  else
    …
```

のように二つの if 文の間に出現する for 文や while

文を考慮したものである．

7. 落し穴検出方式の評価

X window V11R5 のクライアントプログラムのソースファイル180個，約2.9 MB に対して本ツールを適用し，落し穴の検出時間とファイルサイズの関係を求めた*．検出時間は落し穴の項目によって差があるので，図10および図12では検出項目ごとに表1に示した項目番号を印字している．

正規表現検索を用いた場合，項目間の検索時間の差が大きい（**図10**）．この中で検索時間が非常に短いのは，曖昧な表現，入れ子コメント，整数定数における基数，文字列と文字定数，return の後の不足したセミコロン，break 文のない case ラベル，ぶらさがり else の落し穴である．これらの中で，break 文のない case ラベルとぶらさがり else の落し穴は誤検出が生じるが，それ以外の項目は誤検出の問題も生じない．従って，これらの項目には，正規表現検索が適している．

図11はファイルサイズに対する字句解析（s）および構文解析（p）の所要時間を示す．**図12**では，正規表現検索で問題が残った項目のうち，条件判定部における代入文，演算子の優先度，余分なセミコロン，struct の後の不足したセミコロン，break 文のない case ラベル，ぶらさがり else について構文パターンマッチング時間とファイルサイズの関係を示す．

構文パターンマッチングによる検索の場合，図11に示した字句解析と構文解析の時間と，図12のパターンマッチングに要する時間の合計が検索に必要である．構文解析や字句解析の時間は，yacc 等を用いることによって高速化できる．しかし，字句解析や構文解析を落し穴検出に必要な部分だけに限定すれば，解析時間はあまり問題とならない．

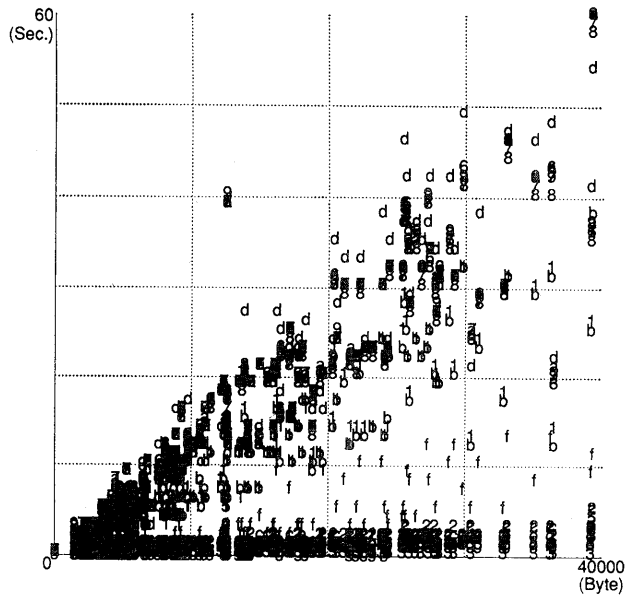


図 10 正規表現を用いた検出時間に対するファイルサイズの関係
Fig. 10 File size vs pitfall detection time using regular expression.

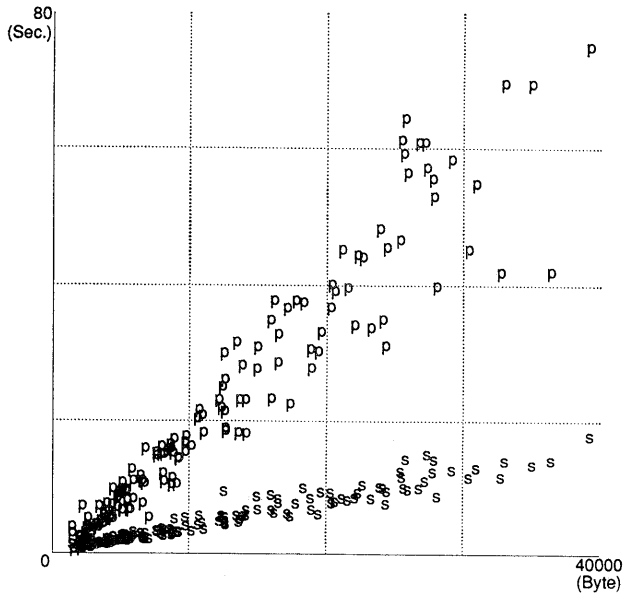


図 11 字句解析時間と構文解析時間に対するファイルサイズの関係
Fig. 11 File size vs lexical analysis time [s] and syntactic analysis time [p].

従って，構文パターンマッチング時間と正規表現検索時間の比が速度差に対応しているが，これらを比較した場合，構文パターンマッチングが約2.4倍高速で

* 図10，図11，図12の縦軸のスケールはそれぞれ異なっていることに注意されたい．

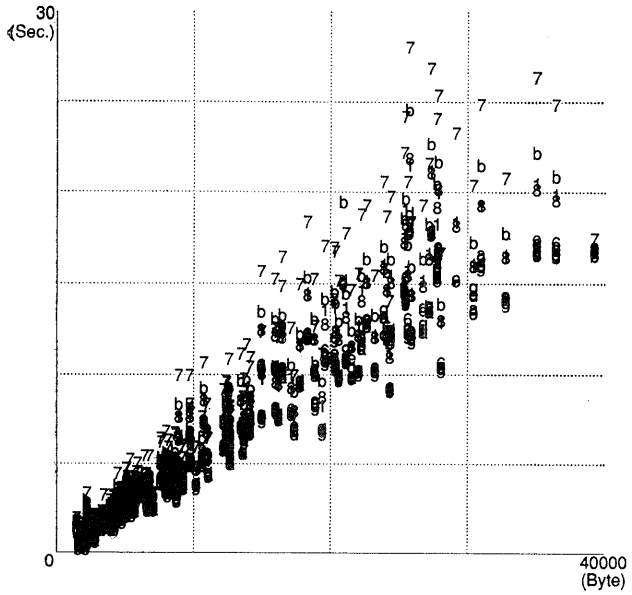


図 12 構文パターンマッチング時間に対するファイルサイズの関係
Fig. 12 File size vs pitfall detection time using structrual pattern matching.

ある．なお，落し穴の検出項目ごとに構文パターンマッチングを用いた場合と正規表現を用いた場合の検出時間を比較し，その速度向上率を**表 2** に示す．項目番号 e および g においては正規表現検索の方が高速であるが，以下に述べるように誤検出の問題は解決されていない．

次に，同一ファイルに対してFall-in C を適用し，誤検出率を調べた．この結果を**表 3** に示す．ただし，誤検出率は，本ツールが検出した落し穴候補の個数を total，落し穴の個数を pitfall とする時，

$$\frac{\text{total}-\text{pitfall}}{\text{total}}$$

で定義される．

正規表現検索で誤検出が生じた項目も，構文パターンマッチングによって誤検出がなくなった．演算子の優先度の落し穴は，正規表現検出においても誤検出がない．しかし，このためには入れ子になった括弧を考慮し，さらに ‘,’ や ‘;’ によって区切られた式の連続構造において異なる式に出現する演算子を区別する特別な検出処理を行う必要があった．ぶらさがり else は専用の正規表現検索を用いた検索関数を用意したにもかかわらず誤検出が生じた．しかし，構文パターンマッチングを用いた場合，ぶらさがり else のような構文的な落し穴も簡単にパターンで表すことができると同時に誤検出もなくなった．

表 2 構文パターンマッチングの正規表現検索に対する速度向上率
Table 2 Speed up rate (structural pattern matching vs regular expression search).

項目番号	速度向上率	項目番号	速度向上率	項目番号	速度向上率
1	1.1 倍	9	2.3 倍	e	0.1 倍
6	2.4 倍	a	2.4 倍	g	0.07倍
7	1.6 倍	b	9.5 倍	—	—
8	2.0 倍	d	2.2 倍	—	—

表 3 誤検出の比較
Table 3 Comparison of erroneous pitfall detection rate.

検出項目	誤検出率	
	正規表現	構文解析
条件判定部における代入	75%	0%
演算子の優先度	0%	0%
余分なセミコロン	3.8%	0%
struct の後の不足したセミコロン	—	—
break 文のない case ラベル	10%	0%
ぶらさがり else	38%	0%

8. おわりに

Fall-in C では，拡張と保守の容易さを考えて，落し穴の検出にパターンマッチングを用いた．落し穴検出に正規表現を用いた場合，構文的な落し穴検出においては複雑な前処理や検出方法が必要であると同時に誤検出が生じる．しかし，字句解析上の落し穴は非常に高速に正しく検出できる．また，構文パターンによる検出は字句解析上のいくつかの落し穴を検出できないが，構文的な落し穴検出における誤検出は生じない．そこで，両者を組み合わせることによって双方の長所を引き出すことができた．

Fall-in C の検出する落し穴項目については，lint 等の検出項目との一致は少ない．lint は型チェック等の意味的な落し穴検出に優れているため，両者を統合した落し穴検出ツールを現在開発中である．また，C プログラムが変更された部分に限って構文解析処理を実行することによって，ツールの高速化を図りたいと

考えている．

参考文献

1) Koenig, A.（中村 訳）：Cプログラミングの落し穴，トッパン (1989).
2) Darwin, I. F.（矢吹 監修・菊池 訳）：lint, 啓学出版 (1990).
3) 都野：正規表現による C プログラムの落し穴検出ツール，佐賀大学理工学部情報科学科卒業論文 (1992).
4) 掛下，小田：正規表現による C プログラムの落し穴検出ツール，情報処理学会研究報告，Vol. 92-SE-86-7 (1992).
5) 小田，掛下：C プログラムの落し穴検出ツールにおける誤検出評価，電気関係学会九州支部連合大会，1166 (1992).
6) 小田，掛下：C プログラムの落し穴検出ツール Fall-in C の構文解析による改良，佐賀大学理工学部集報，Vol. 22, pp. 187-195 (1994).
7) Stallman, R.（竹内・天海 訳）：GNU Emacs マニュアル，共立出版 (1985).
8) Lewis, B., LaLiberte, D. and the Manual Group：GNU Emacs Lisp Reference Manual, for Emacs Version 18, Free Software Foundation (1990).
9) Kernighan, B. W. and Ritchie, D. M.（石田 訳）：プログラミング言語C（第2版），共立出版 (1989).

付録 落し穴パターン

構文パターンマッチングを用いる落し穴検出においては，6.2節で説明したもの以外に，**図13～15**に示すパターンを用いる．

図13はシフトと算術演算子の優先順位の落し穴パターンである．＋パターンを4個使った部分は，＊パターン2個で置き換えることもできるが，＋パターンを用いた方がパターンマッチング時間が短い．図13の Shift-Exp を Assign-Exp, Add-Exp を Equal-Exp, Mul-Exp を Rel-Exp とそれぞれ交換すれば，代入と比較演算子の優先度の落し穴パターンになる．このように，優先順位の落し穴パターンは，式の種類を変更することにより簡単に記述できる．

（平成5年12月24日受付）
（平成6年 7 月14日採録）

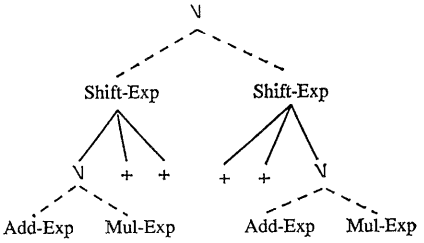


図 13 優先順位の落し穴パターン
Fig. 13 The pitfall pattern of operator priority.

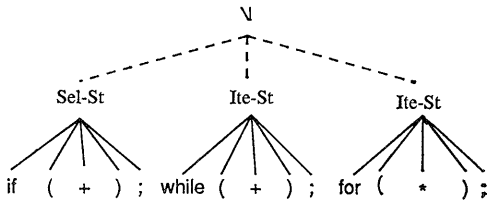


図 14 余分なセミコロンの落し穴パターン
Fig. 14 The pitfall pattern of extra semicolon.

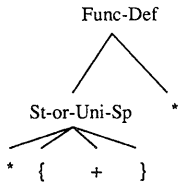
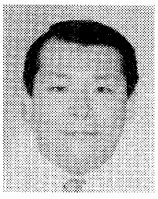


図 15 struct の後の不足したセミコロンの落し穴パターン
Fig. 15 The pitfall pattern of struct definition without semicolon.

小田まり子（正会員）

昭和40年生．平成4年佐賀大学情報科学科卒業．平成6年同修士課程修了．理学修士．同年情報処理学会九州支部新人奨励賞受賞．同年久留米工業大学電子情報学科助手．大学院時代はソフトウェアツールの研究，現在は自然言語処理の研究に従事．



掛下 哲郎（正会員）

昭和37年生．昭和59年九州大学情報工学科卒業．平成元年同博士課程修了．工学博士．同年佐賀大学情報科学科講師を経て，現在，助教授．データベースおよびソフトウェアツールの研究に従事．電子情報通信学会，ACM，日本ソフトウェア科学会等会員．