ソフトウェア工学 86－7
（１９９２．７．２１）

正規表現によるCプログラムの落し穴検出ツール

掛下 哲郎　小田 まり子

佐賀大学情報科学科

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、実際の動作がプログラマの意図と異なる場合がある。このようなバグを落し穴と呼ぶ。本研究では、Cプログラムの落し穴を検出するためのソフトウエアツールを開発した。本ツールは、プログラミングの単純さと保守の点から落し穴の検出に正規表現検索を用いる。また、発見した落し穴を即座に修正できるようにツールはエディタ GNU Emacs 上で Emacs Lisp を利用して作成した。検索対象の落し穴は、(1) 条件判定部における代入文 、(2) 字句解析の落とし穴、(3) 整数定数における基数、(4) 文字列と文字定数 、(5) 演算子の優先度、(6) 余分 (不足した) セミコロン、(7)break 文で終了しない case ラベル、(8) 関数呼び出し、(9) ぶらさがり else、がある。本ツールの検証も合わせて行なっている。

A Pitfall Detection Tool for C Programs by Regular Expression Search

Tetsuro Kakeshita　Mariko Oda

Department of Information Science, Saga University,
Saga 840, JAPAN

Programming language C has a flexible structure, and its compiler provides us with compact and efficient object codes. But the compiler cannot detect some types of bugs hidden in the program. We developed a software tool to detect such types of bugs, or pitfalls, in C programs. This paper describes the outline of the tool. We utilize regular expression search for simplicity of the development and for the ease of maintenance. In order to enable a programmer to correct the detected pitfalls immediately, we built the tool into the GNU Emacs editor. The tool can detect the following types of pitfalls: (1)assignment statement within conditional part, (2)lexcal analysis errors, (3)radix errors, (4)string and character, (5)priority of operators, (6)extra (or dropped) semicolon, (7)case labels without break, (8)function evaluation without argument list, (9)dangling else. We also evaluated the tool by applying it against 2.9 MB of source files.

1 まえがき

C 言語 [Kern89] はアセンブリ言語並みの柔軟性を持ち、コンパクトで高速なプログラムが書けると同時に、異機種間での互換性が高く、実用性にも優れている。しかし、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いるので、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出来上がることもある。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、プログラマのミスであるにも関わらず、コンパイラによって検出できないバグを落し穴と呼ぶ [Koen89]。

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 [Darw90]。しかし、lint は余分なメッセージを出力したり、検出不能な落し穴が存在するなどの欠点がある。また、ユーザインターフェースが貧弱なため、落し穴が見つかってもソースコードの対応する部分を即座に変更することが難しい。

本研究では、このような問題点を解決するために、新たなソフトウエアツールを開発する。ユーザは、Emacs エディタ [Stal85] 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る。

本ツールは編集中の C ソースファイルを検索し、落し穴が存在すればそれを表示する。エディタの内部での処理なので、即座に落し穴を修正することが可能である。

本論文は、次のように構成されている。2 節では、Emacs の特徴および正規表現について説明を行なう。本ツールで検出できる落し穴についての説明は 3節で行なわれる。4節では、実現方法について説明する。最初に、全体構成を述べた後、いくつかの落し穴に対応する正規表現を紹介する。実現にあたっては、正規表現検索に先だって、いくつかの前処理などが必要になることがあるので、それについても説明を加えている。5節では、本ツールの評価および考察を行う。

2 開発環境

本ツールは GNU Emacs 上で Emacs Lisp [GNU90, 半田 90] を用いて開発を行なう。その目的は、Emacs エディタの優れた機能を利用して、ユーザが使いやすいツールを提供するためである。本節では、その機能と正規表現の構文を示す。

2.1 GNU Emacs の機能

本ツールで利用した GNU Emacs の機能を以下に列挙する。

1. Emacs Lisp で記述されているため、機能拡張が容易である。
2. 正規表現による検索が可能である。
3. 複数のファイルを同時に編集できるので、作業用のバッファの定義や切替えなどの操作がプログラムから自由に行なえる。
4. 画面を分割することができ、その間を自由に移動できる。
5. テキストの挿入、削除、および、変更の取消が簡単にできる。
6. 新たに拡張した関数などのドキュメントを登録できる。
7. 機能拡張したコマンドや、しばしば使うコマンドを任意のキーに割り当てることができる。

2.2 Emacs における正規表現の構文

Emacs における正規表現の構文には、特別な意味を持つ文字がある。それ以外の文字は単純な正規表現で、その文字自体にマッチする。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

\| '\|' は複数の正規表現からの選択

\(…\) '\(…\)' 正規表現のグループ

その目的を次に挙げる。

1. '\|' の通用範囲を定める。
2. 複雑な表現を囲み、'*' が使えるようにする。
3. 囲んだ部分の文字列を後で引用できるようにする。

通常、正規表現による検索は行単位で行なわれるが、[^…] 構文を使うと改行コードもマッチする対象となるので、複数行にわたるパターン検索が可能になる。

3 Cプログラムの落し穴

プログラムが落とし穴に陥った場合には、そのプログラムを注意深く眺め、コンパイラが自分の意図どおりに理解しているかどうかを確認しなければならない。しかし、落し穴を自分自身で見つけ出すことはプログラマの先入観も手伝って非常に困難である。本節では、落し穴検出ツールが検出する落し穴についてのいくつかの事例を、その種類毎に示す。

3.1 条件判定部における代入文

二重等号==は、"等しい"ことを表すCの記法である。この記号は、代入に使われる単一の=と等号のテストを区別するために使われる。実際、代入は比較より出現頻度が高いため、この方が便利である。そのうえ、多重代入が容易に書け、代入文をより大きな式の中に埋め込むこともできる。しかし、等号を書くつもりで不注意に代入演算子を書いてしまう場合がある。

例えば、次の文は、変数CがEOFならば、breakを実行するように見えるが、

```
if ( C = EOF ) break;
```

実際は、CにEOFを代入した後、ゼロでないかを調べている。従って、プログラマの意図した結果は得られない。そこで、ifやwhileなどの条件判定部に代入演算子の= がある場合、プログラマのミスでないかどうかを確認する必要がある。

3.2 字句解析上の落とし穴

Cコンパイラはプログラムをトークン列に分割する際、最長一致規則を用いている。それゆえ、== は単一トークンで、=␣= は2つのトークンである。また、式a---b はa--␣-b と同じでa-␣--b ではない。

また、x の値をポインタ p が指した値で割った結果を y に代入するつもりで

```
y = x/*p   /* x を*p で割る */;
```

と書いた場合、コンパイラは/* をコメントの始まりと解釈し、*/が現れるまでプログラムを読み飛ばしてしまう。

そこで、+++、------、また +=-1 、および*␣*␣␣*\のように曖昧さを含んでいる部分を検出する必要がある。

3.2.1 整数定数における基数

整数定数の最初の文字が数字の0であるならば、その定数は8進数である。それゆえ、10 と 010 は全く異なる。しかし、これを考慮せずに桁揃えのつもりで不用意に0をつけてしまう場合がある。そこで、ファイル中に8進数がある場合、確認する必要がある。

3.2.2 文字列と文字定数

単一引用符と二重引用符はCでは全く異なった意味を持っている。しかし、Cコンパイラは、実引数の型チェックを行なっていない。例えば、printf の第一引数に単一引用符を使った場合、コンパイラはエラーを出さないが、意図した結果は得られない。

3.3 演算子の優先度

Cプログラムには豊富な演算子が用意されている。これらの優先度を誤ると、落とし穴に陥ることがある。そこで、一つの文の中に異なる優先度を持つ演算子が二つ以上現れる場合、優先度の確認が必要である。

3.4 余分な (不足した) セミコロン

余分なセミコロンやセミコロンの不足によって落し穴に陥ることがある。

ちょうど1個の文が後に続いているif文やwhile文では、条件部の後の括弧の後にセミコロンがある場合、例えば

```
if( x[i] > big );
    big = x[i];
```

は、括弧の後に空文があるとみなされ、後の文はif文の実行部とはならない。

return の直後にセミコロンがない場合や、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も、落し穴に陥ることがある。

3.5 break 文で終了しない case ラベル

C 言語は他の言語の制御方法とは異なり、switch 文のそれぞれの case において制御が後続の case に流れ込むことができる。従って、switch から抜け出るための break 文等の書き忘れがないかどうかを確認する必要がある。

3.6 引数リストのない関数呼び出し

C の関数呼び出しでは引数がなくても空の引数リストを持つことが必要である。引数リストを持たない関数呼び出しは何も行なわない文である。そこで、引数リストを持たない関数呼び出しを指摘する必要がある。

3.7 ぶらさがり else

if-else 構文の else 部は省略が可能なので、if と else の対応がわかりにくくなることがある。特に、次のような場合、構文が曖昧になってしまう。

```
1:  if ( … )
2:       for (      …       )
3:            if ( … ){
4:                 …
5:            }
6:  else  /*   落とし穴   */
7:                 …
```

この場合、コンパイラは else を内側の if に結びつけてしまうので、プログラマが外側の if と else を結びつけたいならば、意図に反してしまう。そこで、ぶらさがり else がある場合、対応する if 文に間違いがないかどうかを確認する必要がある。

4 落し穴検出ツールの実現

本ツールの実現にあたっては 2.1節で述べた Emacs の機能を十分利用している。

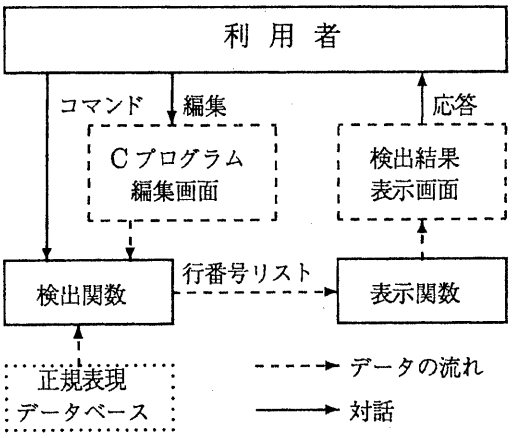

図 1: システム構成

本ツールは、C プログラムの編集画面から直接、呼び出すことができる。しかも、ツールの起動はコマンドによって簡単に行なえる。また、ソースファイルの落し穴の箇所に、即座にポイントを移動させて、ユーザにその位置を知らせる機能もある。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、検出の方法や検出項目についてのドキュメントの参照も容易である。

本節では検出ツールの構成、および落し穴の検出方法と、その問題点について述べる。

4.1 全体構成

検出ツールの全体構成を図1に示す。システムの動作は、C プログラムを編集中のユーザが、検出したい項目を本ツールに指示することから始まる。指示コマンドを受けた検出関数は、データベースから指示された項目に対応する正規表現を取り出し、それを使って C プログラム中の落し穴を検索する。検出された落し穴は表示関数によって、ウインドウ上に表示される。このウインドウは落し穴へのメニューにもなっており、C ソースの対応する部分にジャンプすることができる。図2にぶらさがり else の落し穴を検出した時の画面を示す。

4.2 正規表現による落し穴の検出

本ツールでは、項目ごとに用意した正規表現を用いて落とし穴の検出行なっている。検出項目としては、3 節で説明したものがある。

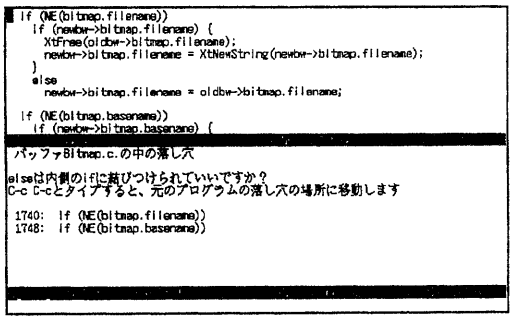

図 2: 落し穴の検出画面

正規表現探索は構文解析を行なう方法と比較して、検出プログラムが作りやすく、保守も容易になる。本節では、落し穴を検索するための正規表現についてその一部を紹介する。なお、作成した全ての正規表現を格納したデータベースは付録に示す。

4.2.1 条件判定部における代入文の検出

条件判定部における代入文を検出するには、次の正規表現を用いる。

```
\b\(if\|while\)\s␣*
    ([^)]*[^-+)/%*=><!]=[^)=][^)]*)
```

正規表現中のアンダーラインがある= の前後の [^…] は=が単独でくるような制約を与えるものである。

この正規表現は条件部中に括弧が現れないことを前提としているので、検出を行なう際の前処理としてif や while の条件判定部の中にある入れ子の括弧は削除しておく必要がある[1]。

4.2.2 引数リストのない関数呼び出しの検出

関数呼び出しにおける引数リストの書き忘れを検出するためには、2 段階の正規表現探索を行なっている。

最初に、引数を持たない関数の定義か関数の呼び出しを探す。そのための正規表現を次に示す。

```
\sw+\s␣*(\s␣*)
```

これによって、空の引数リストを持つ関数呼び出し、または関数宣言を検索できる。

第 2 段階の検索には、先の正規表現によって検索された文字列を含んだ次の正規表現を用いる。

```
\b%s\b\s␣*[^(␣]
```

ただし、正規表現の中の%s には、先の検索で得られた文字列 (関数名) が入る。この正規表現は引数リストを持たない関数呼び出しにマッチする。

4.2.3 ぶらさがり else

構文的な落とし穴の場合、一つの落とし穴の構文の中に、入れ子になった同種類の落とし穴が隠れている可能性がある。よって、字句解析の落とし穴のような簡単な方法では検出できない。しかし、複数の正規表現による探索などを組み合わせて必要な関数を定義した結果、ぶらさがり else の落とし穴も検出可能となった。その検出方法を簡単に説明する。

バッファの最初から正規表現による探索を用いて、else を探索する。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、ぶら下がり else が検出できたことになる。

4.3 実現上の問題点

正規表現で C プログラムの落し穴を検出するためには、解決しなければならない問題がいくつか存在する。そこで、検出項目に応じた前処理を行なう必要がある。この処理は、C のソースファイルを別バッファにコピーし、そのバッファ上で行なわれる。本節では、正規表現検索を可能にするために別バッファで行なう前処理について述べる。

4.3.1 コメントの処理

コメントは空白あるいは改行が書けるところならばどこにでも自由に挿入できる。しかもコメントの中には任意の文字列を書くことができる。このことを考慮して正規表現を作ることは不可能である。この問題を解決するため、正規表現探索を行なう前に、コメント削除の前処理を行なっている。

コメントの削除処理にも、正規表現探索を利用している。具体的には、コメントの始まりと終りの位置を 2 つの正規表現/* と */によって検出する。

4.3.2 文字列、文字定数の処理

正規表現で落し穴の検出を行なう際、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。

[1] 前処理については、4.3節で述べる。

そこで、検出を行なう前に文字列や文字定数も一旦、削除する。この前処理も、正規表現探索を利用することで簡単に実現できる。

4.3.3 入れ子になった括弧の処理

構文的な落し穴は正規表現で表すことが難しい。例えば、正規表現では入れ子になった括弧を表現できない。従って、入れ子になった括弧を含む構文の落し穴は単純な正規表現だけでは検出できない。そこで、必要な場合、入れ子になった括弧を空白に置換する前処理を行なっている。

まず、括弧を空白に置き換えるための関数を定義した。閉じ括弧と開き括弧は正規表現 `[()]` で検索できる。従って、`[()]` を検索した結果、マッチした部分を空白に置換すれば、括弧の削除処理を行なえる。

次に、入れ子になった括弧だけを削除する。まず、正規表現 (を用いて、開き括弧を探し、これと対応する閉じ括弧を探す[2]。この中で、先の関数を実行すれば、入れ子になった括弧を削除できる。

この処理の後、if 文の条件判定部は、

```
\s␣+if\s␣*([^)]+)
```

という簡単な正規表現で表すことが可能になる。

この例が示すように、構文的な落し穴の中には、ある特定の文字や文字列を削除することで正規表現検索が可能になるものがある。同様の処理を入れ子になった大括弧に対しても行なっている。

4.3.4 文字列の否定

正規表現では、一文字に対しての否定は表せるが、文字列の否定を表せない。このことが正規表現検索の際に問題になる。なぜなら break という文字列を正規表現で表すことは可能だが、break の否定を正規表現で表すことはできない。よって、break 文で終了しない case ラベル[3]の落し穴を検出する時、break 文がないという条件では検索できない。

そこで、次の正規表現を用いて、制御が流れ込んでこない case、もしくは連続した case を検索する。

```
\(\bbreak\s␣*;\|\bcontinue\s␣*;
\|\breturn\b[^;]*;\|\bexit\b[^;]*;
\|{\|:\)\s␣*\(case\|default\)
```

この検索の結果、得られる文字列中の case や default の部分を削除する。従って、この処理の後に残っている case や default は、制御が流れ込んでくるものだけである。よって、次に、正規表現

```
\b\(case\|default\)\b
```

を用いれば、目標とする落し穴が検出できる。

同様の処理を、do-while 構文でない while を見つける際や、アドレス演算子以外の& を検索する際にも利用している。

5 検出ツールの検証

本ツールは落し穴を検出する際、正規表現を主に用いているため、ある程度の誤検出は避けられない。しかし、その誤検出の度合がツールを利用する際に、気にならない程度であれば、十分有用なツールであると言える。

そこで、ツールの有効性を確かめるため、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 2.9MB に対してツールを適用し、誤検出率を調べた。この結果を表1に示す。ただし、誤検出率は、total を検出した落し穴の個数、pitfall を本ツールが検出した落し穴候補の個数とする時、

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

で定義される。

また、本来、検索対象であるにも関わらず、落し穴として検出できなかった箇所についても検証すべきであるが、この検証は全て人手で行なはなければならない。そこで今回は、論理的に検出もれがないように注意を払って正規表現を作成し、考えられる種々のパターンで検出もれがないかどうかを確かめることにとどめた。

この表では、誤検出率はやや高いが、これは、対象としたプログラムが既に動作の確かめられたプログラムであるためである。従って、実際に開発中のプログラムではかなり高い実用性があるものと思われる。

誤検出率が最も高かった項目は条件判定部における代入文であるが、これは、正規表現検索を行なう前に、入れ子になった括弧の処理を行なっているため、落し穴である

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

[2]対応する括弧までポイントを動かすという関数が Emacs Lisp にはある。

[3]break 以外に、exit, return, continue 等がくる場合もある。

番号	検出項目	個数	誤検出率
1	条件判定部における代入文	428	75%
2	字句解析上の落し穴	0	—
3	コメントの範囲	1	0%
4	整数変数における基数	61	0%
5	文字列と文字定数	0	—
6	演算子の優先度 (代入と比較)	246	1.5%
7	演算子の優先度 (&演算子と比較)	85	72%
8	演算子の優先度 (シフトと算術演算子)	34	38%
9	演算子の優先度 (代入と比較)	640	20%
10	演算子の優先度 (代入演算子)	556	40%
11	余分なセミコロン	52	3.8%
12	return の後の不足したセミコロン	0	—
13	struct の後の不足したセミコロン	0	—
14	break 文で終了しない case ラベル	82	61%
15	引数リストのない関数呼び出し	410	53%
16	ぶらさがり else	21	38%

表 1: 検証結果

と、落し穴ではない

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

を区別できず、両方とも落し穴として検出するためである。正規表現検索だけで落し穴の検出を行なう場合、このような誤検出は避けることができない。

しかし、正規表現は、字句解析上の落し穴検索には優れており、構文解析では検出できない項目の検出が可能である。例えば、3.2節で例に挙げた落し穴

```
y = x/*p /* x を*p で割る */;
```

は構文解析では検出できないが正規表現検索ならば検出できる。

6 むすび

C プログラムの落し穴を検出するためのツールを Emacs Lisp を用いて作成した。落し穴の検出には正規表現を用い、保守性が良い、ユーザインターフェイスが優れたツールを作ることができた。

X window V11R5 のクライアントプログラムのソースファイルを使いツールの有効性を検証した結果、役立つツールであることが確信できた。

しかし、今回対象としなかった落し穴の検出や、誤検出率を下げるためには、正規表現だけでは困難であり、今後、簡単な構文解析処理なども採り入れる必要のあることがわかった。

また、ユーザインタフェースの改良も、引続き行ないたい。

参考文献

[Darw90] I. F. Darwin (矢吹 道郎 監修・菊池 彰 訳), "lint", 啓学出版 (1990)

[GNU90] B. Lewis, D. LaLiberte, the GNU Manual Group, "GNU Emacs Lisp Reference Manual, for Emacs Version 18", Free Software Foundation (1990)

[半田 90] 半田剣一, "Emacs Lisp 入門", スーパーアスキー, Vol.1 No.4-5, Vol.2 No.1-4 (1990〜1991)

[Kern89] B. W. Kernighan, D. M. Ritchie (石田 晴久 訳), "プログラミング言語 C (第 2 版)", 共立出版 (1989)

[Koen89] A. Koenig (中村 明 訳), "C プログラミングの落し穴", トッパン (1989)

[Stal85] R. Stallman (竹内郁男・天海良治 訳), "GNU Emacs マニュアル", 共立出版 (1985)

A　正規表現データベース

```
(defvar pitfall-list
  '(
;;;if 文等の条件式の中に代入の=があるとき、==ではないかどうか確認する
    (1 "\\b\\(if\\|while\\)\\s *([^)]*[^-+)/%*=><!]=[^)=][^)]*)"
       " == ではありませんか? それとも代入式の前後に括弧を忘れていませんか？" t nil t nil)

;;; 字句解析における落し穴を探す。
    (2 "\\([^<-]---+[^>-]\\|\\+\\+\\++\\|[^-+*/=><%!]\\(=-\\|=\\+\\)[0-9]+\\)"
       "字句の解析が意図した事と違っていませんか？"  nil nil nil nil)
    (3 "/\\*\\([^/*]\\|\\*[^/]\\)*/\\*"
       "コメントが入れ子になっていませんか？"  nil nil nil nil t)

;;; 整数定数　不用意な８進数の用い方による落し穴を検出する。
    (4 "[^A-Za-z0-9_.]0[0-9]+"
       "８進数に間違いないですか？" )

;;; 文字列と文字定数　単一引用符と二重の引用符の混同
    (5 "\\bprintf\\s *(\\s *'.'\\s *)"
       "二重引用符ではありませんか？" )

;;;  関係演算子や等値演算子と代入演算子＝の優先度
    (6 "\\b\\(if\\|while\\)\\s *([^)=><!*/%]+=[^)=]+\\([=!<>]=\\|<\\|[^-]>\\)[^)]*)"
       "条件式の中にある演算子の優先度は確認できていますか？ "  nil nil t nil)
;;;  ビット毎の AND 演算子&と関係演算子や等値演算子、シフト演算子の優先度
    (7 "\\b\\(if\\|while\\)\\s *([^)]*[^)&]&[^)&=][^)]*\\([=!<>]=\\|<\\|[^-]>\\)[^)]*)"
       "優先度の確認はできていますか？&演算子の優先度は関係演算子より低いです "nil nil t nil nil nil t)
;;;  算術演算子はシフト演算子より優先度が高い。
    (8 "\\(<<\\|>>\\)[^;={]+[-*+/%]"
       "優先度の確認はできていますか？算術演算子はシフト演算子より優先度が高いです。"   nil nil t)
;;;  代入演算子は関係演算子より低い優先度を持つ。
    (9 "\\([^;=><!]\\|<<\\|>>\\)=[^=;{][^;{]*\\([=!<>]=\\|<\\|[^-]>\\)"
       "代入演算子は条件演算子より低い優先度を持ちます。確認できていますか？" nil nil t)
;;;  全ての代入演算子は同じ優先度をもち、全て右から左に結合していく。
    (10 "\\([^;=><!]\\|<<\\|>>\\)=[^=;{][^;{]*\\([^;=><!]\\|<<\\|>>\\)=[^=]"
        "右から左に結合していきます。確認できていますか？ "  nil nil t)

;;; セミコロンに気をつけろ
;;;  if や while の条件式の後にセミコロンが必要かどうかの確認を行なう。
    (11 "\\b\\(if\\|while\\|for\\)\\s *([^)]*)\\s *;"
        "if や while の条件式の閉じ括弧の後にセミコロンは必要ですか？"  nil nil t nil nil t)
;;;  return 文の後のセミコロンを書き忘れることによって陥る落し穴を検出する。
    (12 "\\breturn\\s +$"
        "return 文の後にセミコロンが必要ないですか？"  nil t)
;;;  struct の宣言の終りにセミコロンを忘れることによって陥る落し穴を検出する。
    (13 "\\bstruct\\s +\\sw+\\s *{[^}]*}\\s *\\sw+([^)]*)\\s *[^,;]"
        "}の後にセミコロンは必要ないですか？"   nil nil t t)

;;;switch 文において break 文を書き忘れていないかどうか確認する
    (14 "\\b\\(case\\|default\\)\\b"
        "case 文の前に break 文を忘れていませんか？"   t)

;;; 関数呼び出しでは引数がなくても引数リストが必要である。
    (15 (search-no-arg-function
 "[k15] " "関数呼び出しでは引数がなくても引数リストが必要です。"))

;;; ぶらさがり else の問題
    (16  (search-else "[k16] " "else は内側の if に結びつけられていいですか？")))

  "検出する落し穴の正規表現とメッセージ、前処理のフラグのリストのデータベース
ただし、例外的な処理の場合は関数呼び出しがそのまま書かれている")
```