プログラミングスタイルの診断システムに関する研究

2001年3月

萱津(関本) 理佳

プログラミングスタイルの診断システムに関する研究

2001年3月

萱津(関本) 理佳

概要

プログラムの可読性、理解容易性に影響を及ぼすようなプログラムの書き方のことをプログラミングスタイルと呼び、一般的なガイドラインがいくつか存在する。プログラミングスタイルに従って作成されたプログラムは、テストやデバックにおける労力を軽減させることができ、保守性の高いものとなる。しかしながら、プログラミング教育においては、文法やアルゴリズムの講義が中心となっており、プログラミングスタイルの教育にはあまり時間が割かれていないのが現状である。この原因の一つとして、プログラミングスタイルの教育に関する支援システムが実用化されていないことが挙げられる。

本研究では、プログラミングスタイルに違反する箇所の検出を行なうことにより、プログラミング学習者による悪いプログラミングスタイルの認識を支援するプログラミングスタイルの診断システムの開発を目的としている。本システムは、プログラムパターンと呼ぶ記述形式で表現された個々のプログラミングスタイルについて、プログラムのソースコード探索を行ない、マッチしたプログラム断片がある場合には、マッチした個数とその場所、修正のためのアドバイスを表示する。システム利用者は、その診断メッセージを参考にプログラムの修正を行なうというものである。

本論文は6章で構成されている。以下に各章の概要について述べる。

第2章では、プログラミングスタイルの個々のガイドラインを表現するためのプログラムパターンの記述形式について述べた。提案したプログラムパターンの記述形式では、テンプレートを記述する本体部と制約条件を記述する制約条件部に分けて表現する記法を用いた。この分離により、(1) 文脈に依存したパターンであっても、文脈条件は制約条件として述語的に記述できる。これにより、テンプレートの記述は、文脈に依存しない構文パターンとして記述でき、そのまま有限状態オートマトンによるソースコード探索により検出できる。制約条件は、事後条件として個々に実現すればよく、アルゴリズムもシンプルになり、記述自体の再利用性も増す。(2) プログラムのフロー情報などを制約条件として利用する場合でも、システムのアルゴリズムをほとんど変更することなく、フロー解析ツールおよびその情報を利用する関数の導入だけで済む、という特徴を持つ。また、ベースパターンをテンプレートとして記述する複合パターンにより、離れた場所での利用に関するパターンや、ある命令の存在ではなく、存在

しない事を問題とするパターンも記述できるという特色がある。

第3章では、2章で提案したプログラムパターンをソースプログラム中から認識する手法について述べた。提案したプログラミングスタイルの診断システムは、(1) 個々のガイドライン（各悪形パターン）を一つのプログラムパターンとしてデータ化し、これを用いてソースコードの探索を行なう。このようなオープンな方式を採用することにより、プログラミングスタイルの検査項目の追加が容易に行なえる。(2) 構造的なスタイルに関するパターンや文脈に依存したパターン、また、二つ以上の関数など離れた位置に分散するパターンなど、単純なテキスト上のマッチングでは検索が難しいパターンの認識も可能である。(3) 制約関数が不十分な場合は、新たな関数を追加することになるが、基本的には対象となるプログラム断片の走査であるので、関数自体の実現は容易であり、またシステムへの追加も関数のみの追加であるので容易に実現できる、という特徴を持つ。

第4章では、実際のプログラミングスタイルをプログラムパターンで記述する実験、および、それらのプログラムパターンを実プログラム中から認識する実験について述べた。ここでは、文献等に載っている標準的なスタイルガイドラインのいくつかをプログラムパターンとしてデータ化し、テキストユーティリティのプログラム群を対象にスタイルガイドラインの検査を行うことにより、標準的なガイドラインが正しく検出可能であることを確認した。この結果より、本システムが、プログラミングスタイルの種々の悪形パターンの検出に有効な手法であることが言える。

第5章では、初心者プログラマを対象に行なった二つの評価実験について述べた。初心者プログラマ86名を対象として行なった評価実験1を通して、本システムが初心者プログラマのプログラミングスタイルに関する悪形パターンの検出、および修正の支援に有効に働くことを示した。また、本システムの利用がプログラミングスタイルの検査の際の時間短縮に貢献することが確認できた。さらに、9名を対象として行なった評価実験2を通して、本システムの利用が、初心者プログラマのプログラミングスタイルの悪形パターンの検査能力の向上に役立つことが確認できた。

第6章では、第2章から第5章までを要約するとともに、今後残された研究課題について述べた。

目 次

第1章　序論

1.1　背景

プログラムは、単に仕様通り動けばよいというものではなく、読み易く、わかり易いものでなければならない。プログラムの良い書き方のことを**プログラミングスタイル**と呼ぶ。プログラミングスタイルには、信頼性、効率、操作性、テスト容易性、可読性、理解容易性、変更容易性、移植性など種々の要件が存在し、一般的なガイドラインや企業毎の独自のコーディング規約等が存在する [1, 2, 3, 4]。プログラミングスタイルに従って作成されたプログラムは、プログラムを作った本人ばかりでなく、第3者にとっても理解しやすく、テストやデバックにおける労力を軽減させることができ、保守性の高いものとなる。

プログラム中に構文エラーがあればコンパイラが実行可能コードを作ってくれないし、意味的なエラーがあれば、そのプログラムが自分の意図した通りには動作してくれないので、プログラマはデバックを余儀なくされる。しかし、プログラミングスタイルに違反していても、意図した通りにプログラムが動いてくれさえすれば、すぐに不都合が生じるわけではないので、それらの違反はそのままにされることが多い。特に、初心者プログラマは、プログラムのテストを十分に行なわず、偶然意図した通りに動いただけの場合でも、その事に気付きさえしないことがある。そこで、プログラミング教育においては、文法やアルゴリズムの習得だけではなく、プログラミングスタイルの習得を促すことも必要である。良いプログラマを育てるためには、プログラミングスタイルを意識させることは重要な要素である。しかしながら、プログラミングの学習においてこのようなプログラミングスタイルに関する教育には、あまり時間が割かれていないのが現状である。この原因の一つとして、プログラミングスタイルの教育に関する支援システムが実用化されていないことが挙げられる。

プログラミングスタイルには、コードのレイアウトや変数名の付け方、コメントの方法など体裁に関するものから、構文の使い方に関するものなど様々なガイドラインが存在する。体裁に関するガイドラインの内、コードのレイアウトに関しては、検査

は難しいが、整形ツール (例えば、UNIXではC言語のプログラムを清書するツールとしてindentコマンドがある) が実用化されており、これの利用によりレイアウトの標準化は容易である。また、変数名の付け方や、コメントに関しては、プログラマ自身が心がけることでわかりやすいプログラムを作成できると考えられる。しかしながら、構文の使い方に関するスタイルについては、初心者プログラマが自然言語で記述されたプログラミングスタイルを理解し、それらについて、正しい使い方をしているかどうかを自分自身で検査するのは容易なことではない。さらに計算機システムを利用して、プログラミングスタイルに違反する種々のパターン（悪形パターン）を検出しようとした場合でも、命令の単独の利用ではなく、特定の文脈の中での使用を問題としているものや、プログラム中の離れた場所に存在する二つ以上のプログラム要素の利用に関わるものなどがあり、UNIXのgrepに代表されるような正規表現を用いた既存のソースコード探索ツールでは検出が困難である。そこで、本研究ではプログラミングスタイルのうち、構文の使い方に関するガイドラインに焦点をあてる。プログラム中からこのような悪形パターンの検出を自動で行なうことが出来れば、初心者でもプログラミングスタイルの検査が容易に行なえるようになり、プログラミング学習、また、プログラムの品質の向上に有効に働くと考えられる。

本研究を進めるにあたり、初心者プログラマが作成した「ファイルに記述されたデータ2組をマージするプログラム（100行程度)」30個について、マニュアルでスタイルの検査を行なった。構文に関するプログラミングスタイルに違反していると思われる悪形パターンの種類は全部で7つあり、一番多いものでは9割の人が違反しているものがあった。このことから、実際に初心者プログラマの作成するプログラム中には多くの悪形パターンが存在しており、プログラミングスタイルの学習を支援するツールが必要であることがいえる。

1.2 目的

本研究では、プログラミングスタイルに違反する箇所の検出を行なうことにより、プログラミング学習者による悪いプログラミングスタイルの認識を支援するプログラミングスタイルの診断システムの開発を目指している。本論文では、プログラミングスタイルの構文の使い方に関するガイドラインの内、可読性、理解容易性に影響を及ぼすものに焦点をあて議論する。

プログラミングスタイルには、「この文はこう利用するべきである」という推奨事例や、「この文はこう利用するべきではない」という好ましくないプログラムの書き方を指示したものがある。ここでは、プログラミングスタイルの推奨事例に反しているパターンや、好ましくないプログラムの書き方のことを**悪形パターン**と呼ぶ。

本研究で提案するプログラミングスタイルの診断システムは、以下のような機能要求を基に設計し、実装されている。

- 悪形パターンを検出するための方法として、プログラム組み込み型のシステムではなく、プログラムパターンの検索に基づくシステムとする。

  本研究ではまず、種々の悪形パターンを検出するために、独自なプログラムパターンの表現方法を考案し、これをプログラム中から検出するためのプログラムパターン認識システムを構築する。このプログラムパターンの認識システムを用いて、プログラミングスタイルの診断を行なうことにより、個々のガイドライン（各悪形パターン）を一つのプログラムパターンとして表現し、それをデータとして利用した認識により検査を行なう。このような方法を採用することにより、検査項目の追加が容易に行なえ、拡張性に優れたシステムとなる。

- 既存システムで対象としていないような、命令の単独の利用ではなく、特定の文脈の中での使用を問題としているパターンや、プログラム中の離れた場所に存在する二つ以上のプログラム要素の利用に関わるもパターンも対象とする。

本システムは、C言語のプログラムソースから悪形パターンを使っているプログラム箇所の検出を行ない、診断メッセージを表示する。プログラミングのエキスパートは、自らの失敗経験等によりいくつかの典型的な悪形パターンの知識を持っていると思われるが、初心者では、それが悪形だと知らずに利用していることも多い。これらの指摘は、プログラムの質の向上だけでなく、プログラマの教育、つまり、プログラミングスタイルにそった良形度の高いプログラムの作成を習得させることにも効果が期待できる。

1.3 関連研究

1.3.1 スタイル検査システム

プログラミングスタイルの検査ツールとしては、プログラミングの初期教育におけるサポートを目的としたPascalプログラムの自己評価ツールCAP [5]や、プログラムのモジュール性、構造、レイアウト、ドキュメンテーションなど6つの視点から品質の検査を行なうSTYLE [6]などが開発されている。これらの研究では、様々な視点からプログラムのスタイルに関する考察を行なっているが、実現されたシステムでは、構造的なスタイルや文脈に依存した要素に関する検査は対象としておらず、インデントやコメントに対する検査や、モジュールの大きさなどの統計量に関する検査が中心となっている。また、各々の検査がシステムに手続き的に組み込まれているので拡張性に問題がある。これに対し、本手法では検査項目をプログラムパターンとして追加するだけで新たな項目に対する検査が可能なので、拡張性に優れている。

また、同様の構文パターンマッチングに基づきCプログラムの落し穴検出を行なうツールに、Fall-in C[12, 13]がある。Fall-in Cでは、パターンをCプログラムの構文要素と特殊記号を用いた木として表現している。木による表現を利用する場合、人間がパターンを見た時にそれを直観的に理解しにくく、また、言語知識とは別の文法構造の知識をきちんと理解したエキスパートでなければ記述が難しいという問題がある。さらにFall-in Cのように解析木をマッチングのベースとした場合、複数の関数に渡るパターンや、ある命令の存在ではなく、存在しないことを問題としたパターンには対応できない。本手法では、複合パターンを用いることによりこのようなパターンも認識可能である。

プログラムの構造をシステムがガイドするツールに、構文指向エディタまたは構造指向エディタと呼ばれるものがある。しかしこれらは、構文的まとまり（文法論的には文脈自由文法での構造）の簡易入力または学習を目的とするもので、本システムで対象としているような文脈依存文法に基づく構造を対象としているものではない。また、より上流の構造を考えるものにデザインパターン[19]があるが、これは個々の詳細な実行文については考慮しておらず、本研究が対象としているものとは粒度が異なる。

1.3.2 ソースコード探索

本システムでは、プログラムのソースコード中からプログラムパターンを探索することにより、プログラミングスタイルの悪形パターンを認識する。既存のソースコードの探索システムには、以下のようなものがある。

UNIX の grep に代表されるような正規表現を用いたソースコード探索のツールがある。パターンマッチングとしては強力であるが、これらをプログラミングスタイルの悪形パターンの探索に利用しようとした場合、順序がリジットにきまってしまう、複数行にわたってのパターンを探索できない、などのいくつかの限界がある。また、テキスト上の文字列のみに着目した探索を行なうため、任意の繰り返し文の探索等において、構文要素を対象とした抽象表現による探索が行なえない。

上記のような制限を解決するためのシステムとして、独自のパターン言語を利用したソースコード探索システム REFINE [14], SCRUPLE [15] などがある。REFINE システムでは、探索パターンをパターン列と制約列で表現し、制約を表現するためのライブラリを用意している。しかしながら、プログラムの抽象的概念の認識と保守のためのプログラム変換を主目的にしているため、粒度のあらい（抽象度の高い）パターンしか表現できず、本研究で対象とするような細かいパターンの認識には適さない。SCRUPLE システムは、プログラム言語に正規表現等の拡張を許したパターン言語を使うことにより、構文に基づくパターンの探索が可能となっている。また、パターン認識においては、効率よくマッチングパターンを見つけるために、有限状態オートマトンを利用した方法を採用している。筆者が提案するシステムにおいても、テンプレートのパターン照合において文献 [15] の方法を採用している。しかし、SCRUPLE システムでは制約の概念を導入しているものの、利用できるのはワイルドカードの文字列の限定のみに限られており一般性に欠ける。

文献 [26] では、プラン認識を利用したソフトウェアアーキテクチュアの認識法を提案している。ここでは、ソフトウェア全般に渡るパターンからのアーキテクチュアの認識を主目的としており、モジュール内およびモジュール間の各々のアーキテクチュアを定義する記法を用意し、与えられた認識ルールを組み合わせて、種々のアーキテクチュアの認識ルールを記述できる。

文献 [25] では、プラン認識に基づく標準化によるプログラムのバリエーションの除去システムを提案している。ここではプランは固定されたものであり、プランダイアグラムを採用している。ユーザーが随意にプランを記述できる形態ではない。

本システムでは、grep のような単なる文字列探索ではなく、構文を主体とし、様々な粒度に対応可能な認識を目指している。

1.4 本論文の構成と各章の概要

本論文では、プログラミングスタイルに関する様々な悪形パターンを検出するためのプログラムパターンの表現方法、および、このプログラムパターンの認識手法について提案する。さらに、この認識手法を用いて構築したプログラミングスタイルの診断システム、初心者プログラマを対象とした評価実験について報告する。

本論文は全6章より構成される。以下に各章の概要を述べる。

第2章においては、プログラミングスタイルの個々のガイドラインを表現するためのプログラムパターンの記述形式を提案する。2.2節では、まずプログラミングスタイルの悪形パターンの記述上の特徴を述べ、次に提案するプログラムパターンの記述形式について述べる。プログラムパターンにはベースパターンと複合パターンという二つのカテゴリが存在する。2.3節および2.4節では、それぞれの特徴と知識表現について述べる。そして、2.5節ではプログラムパターンの記述例を示す。

第3章においては、プログラミングスタイルの診断システムにおけるプログラムパターンの認識方法について述べる。3.2節では、システムの概要およびインターフェイスについて述べる。3.3節では、システムの構成および認識のおおまかな方法について述べる。3.4節では、ベースパターンの認識方法について、3.5節では、複合パターンの認識方法についてそれぞれ述べる。3.6節では、システムによる診断結果の一例を示す。

第4章においては、実際のプログラミングスタイルをプログラムパターンで記述する実験、および、それらのプログラムパターンを実プログラム中から認識する実験について述べる。4.1節では記述実験について、4.2節では認識実験について述べる。そして、4.3節では記述実験および認識実験の評価と考察を行なう。この実験により、プログラミングパターンの認識に基づく手法が、プログラミングスタイルの検査に有効な手法であるか検証する。

第5章においては、初心者プログラマを対象とした二つの評価実験について述べる。5.2節では、評価実験1について述べる。評価実験1では、システムを利用することによる直接的な効果に焦点を当て、初心者プログラマ86名を対象として実験を行なった。この実験により本システムが初心者プログラマのプログラミングスタイルに関する悪

形パターンの検出、および修正の支援に有効に働くことを検証する。5.3節では、評価実験2について述べる。評価実験2では、システムを利用することによる教育的効果に焦点を当て、初心者プログラマ9名を対象として実験を行なった。この実験により本システムの利用が、初心者プログラマのプログラミングスタイルの悪形パターンの検査能力の向上に役立つことを検証する。

第6章においては、第2章から第5章までを要約するとともに、今後残された研究課題について述べる。

第2章　プログラムパターンの記述形式

2.1　はじめに

本研究では、まず、プログラミングスタイルの悪形パターンを表現するための記述形式を提案する。本論文では、この記述形式で表現されたものを**プログラムパターン**と呼ぶ。プログラムパターンは、ある意味を持つプログラム断片を表している。

本章では、まずプログラムパターンの記述形式について述べる。プログラムパターンには、その記述形態からベースパターンと複合パターンの二つのカテゴリが存在する。二つのプログラムパターンに分けた理由、およびそれぞれのプログラムパターンの特徴について述べる。そして、プログラミングスタイルの悪形パターンをプログラムパターンで記述した例を示す。

2.2　プログラムパターン

プログラムパターンの記述形式を図 2.1 に示す。

各プログラムパターンはそれぞれユニークなパターン名 (pattern-name) を持ち、テンプレート (template) を記述する本体部 (BODY) と、それらのテンプレートに対する制約条件 (condition) を記述する制約条件部 (CONSTRAINTS) から構成される。ただし、何の制約も必要としない場合、制約条件部の記述は省略できる。本体部のテンプレート中においてパラメータが使われる場合、パターン名の次にそれらのテンプレートを列挙する (parameter-list)。

本研究で対象とするプログラミングスタイルにおける悪形パターンは、プログラムの可読性や理解容易性を損なうようなパターンの他、副作用を伴うようなパターンや、構文上の誤った理解から生じるエラーパターン、意味の誤解から生じる未定義な結果を与えるようなパターンを含む。これらのプログラミングスタイルの悪形パターンには、以下のような記述上の特徴を持つものが含まれる。

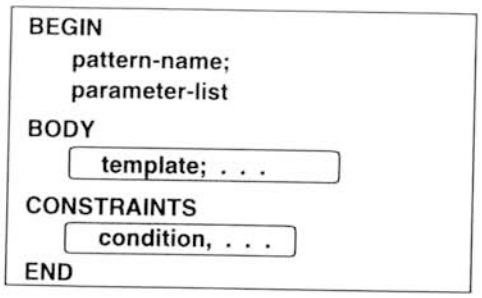

図 2.1: プログラムパターンの記述形式

- 命令の単独の使用ではなく、特定の文脈の中での使用を問題としているものがある。

  これには、例えば演算子の優先度に関するもので、等価式を含む代入文 (4.1.3.1 演算子の優先度の関する例、参照) や関係式を含む代入文が挙げられる。また、副作用に関するもので条件式に代入文を含むパターン (2.5 プログラミングスタイル例 1、参照) などがある。

- 2つ以上の命令のプログラム中での離れた場所（場所については制御フローが存在する以外の条件はない）での利用に関わる場合がある。

  これには、副作用に関するもので「複数の関数の中で、同じ名前のグローバル変数を定義するのは好ましくない」というガイドライン等がある。

- ある命令の存在ではなく、存在しない事を問題としている場合がある。

  これには、ファイル処理に関する「オープンしたファイルは、明示的にクローズするのが好ましい」というものや (2.5 プログラミングスタイル例 2、参照)、「malloc 関数で確保した領域は、free 関数で解放するのが好ましい」というガイドラインが該当する。

以上のような特徴を持つパターンを記述し、認識するために、図 2.1 に示したような本体部と制約条件部を分けて表現する形式を考案した。プログラムパターンでは、テンプレートを記述する本体部とそのテンプレートに対する制約条件を記述する制約条件部に分けて表現する。このような分離により、提案するプログラムパターンの記述、およびその認識において以下の特徴を持つ。

(1) 文脈に依存したパターンであっても、文脈条件は制約条件として述語的に記述できる。これにより、プログラムパターン自体の記述は、文脈に依存しない構文パターンとして記述でき、またそのような物として認識を行なえば良いことになる。制約条件は、事後条件として個々に実現すればよく、アルゴリズムもシンプルになり、記述自体の再利用性も増す事になる。

(2) プログラムのフロー情報などを制約条件として利用する場合でも、システムのアルゴリズムをほとんど変更することなく、フロー解析ツールおよびその情報を利用する関数の導入だけで済む。

(3) プログラムパターンを記述する際は、システムへ組み込み済みの制約関数が利用できる。制約関数が不十分な場合は、新たな関数を追加することになるが、基本的には対象となるプログラム断片（の解析木）の走査であるので、関数自体の実現は容易であり、またシステムへの追加も関数のみの追加であるので容易に実現できる。

プログラムパターンには、テンプレートの記述形態からベースパターンと複合パターンの二つのカテゴリが存在する。ベースパターンは、検索パターンの基本となるもので、テンプレート部は、C言語の文法に基づくコードの断片（正規表現、抽象表現を含む）により記述する。複合パターンでは、ベースパターンをテンプレートとして記述する。複合パターンの本体部で複数のベースパターンを記述した場合、それぞれがパターンの集合として定義されるので、お互いのプログラム上での位置関係については、何の制約も持たない。これにより、離れた場所に分散して存在するパターンも、ベースパターンの組合せとして表現することが可能となる。また複合パターンでは、あらかじめデータベース化してあるベースパターンを利用する。ベースパターンの再利用により、複雑なパターンの記述を簡単化するねらいもある。

2.3 ベースパターン

ベースパターンの本体部は、C言語の文法を拡張した記法を使って定義される。構文要素の代わりに、以下のような抽象的なテンプレートとして働くいくつかの正規表現が利用できる。

- パターン変数
  - 構文要素のワイルドカード

    @, #, $ (それぞれ任意の単一文、式、識別子を表現する)
  - 名前付きワイルドカード

    @v1 ～ @v9, #v1 ～ #v9, $v1 ～ $v9

    同一パラメータの名前付きワイルドカードは、同じコード断片を意味する
  - 複合文

    @*：0あるいはそれ以上の文列

    @+：1つあるいはそれ以上の文列

- 構造に関するワイルドカード
  - カテゴリー名 (@if, @while, @do, @for, @switch, @assign)

    それぞれ任意の、if文、while文、do文、for文、switch文、代入文を表現する。
  - クラス名 (@alternate, @loop)

    @alternateは任意の選択文 (if文, switch文)、@loopは任意の繰り返し文 (while文, do文, for文) を表現する。

ここで、構造に関するワイルドカードとそれに照合する実際のプログラム断片との関係を表2.1に示す。対応行数は、以下のプログラム例2.1クイックソートのプログラム中での行数である。

[プログラム例 2.1]\\

```
01 void
02 quick(int data[], int left, int right)
03 {
04     int     base, temp, r, l;
05
06     if (left < right) {
07         base = data[(left + right)/2]; /* 基準値の設定 */
08         l = left;       /* 左ポインタの設定 */
```

表 2.1: 構造に関するワイルドカードの照合例

ワイルドカード	対応行数
@assign	07, 08, 09, 17,18, 19 行
@if	06 行から 26 行
	15 行から 16 行
@while	10 行から 22 行
	11 行から 12 行
	13 行から 14 行

```
09        r = right;       /* 右ポインタの設定 */
10        while (1) {
11           while (data[l] < base)  /* 左側からの走査 */
12                l++;
13           while (data[r] > base)  /* 右側からの走査 */
14               r--;
15           if (l >= r)      /* 操作の終了判定 */
16               break;
17           temp = data[l]; /* 交換 */
18           data[l] = data[r];
19           data[r] = temp;
20           l++;     /* 左右の操作領域更新 */
21           r--;
22        }
23
24     quick(data, left, l - 1);  /* 左部分列に対する再帰呼びだし */
25     quick(data, r + 1, right); /* 右部分列に対する再帰呼びだし */
26    }
27 }
```

ベースパターンの制約条件部では、テンプレートで利用するパターン変数に対する型、構文、字句に関する制約条件やフロー情報に関する制約を定義する。制約条件は、テンプレートに記述された構文的な枠組から多くの候補を探しだし、さらに限定したパターンや文脈に依存したパターンを検索したい場合に利用する。

表 2.2: ベースパターンの制約条件の例

制約条件	意味
type(p, c)	p のデータ型は c である
type_equal(p1, p2)	p1 と p2 の型は等しい
equality_exp(p)	p は等価式である
assign_exp(p)	p は代入式である
incre_op(p)	p は増分演算子または、減分演算子を含む
not_break(p)	p の中には break 文は含まれていない
literal_equal(p, c)	p と c は定数として等しい
constant(p)	p は定数である
integer_constant(p)	p は整数定数である
integer_large(p, c)	p は整数定数であり、c より大きい
reaching_def(p1, p2)	p1 での定義は p2 での使用に到達する
cflow(p1, p2)	p1 から p2 への制御フローパスが存在する

表2.2に、ベースパターンの制約条件の例を示す。表2.2に示した制約条件のパラメータ中のキーワードは、p はパターン変数を、c は定数をパラメータとして指定することを表す。また、意味の中で記述された p は、パターン変数 p にバインドされたプログラム断片を意味し、c はパラメータとして指定された定数 c そのものを表す。

2.4 複合パターン

複合パターンの本体部は、以下のように記述されたベースパターンを構成要素とする。また、認識の際の便宜上、プログラムパターンの記述において、BODY の後ろにキーワード COMPLEX を記述する。

```
label: pattern_name(parameter);
```

表 2.3: 複合パターンの制約条件の例

制約条件	意味
node_literal_equal(p1, p2)	p1 と p2 の文字列が等しい
node_equal(p1, p2)	p1 と p2 のノードが等しい（同一のプログラム断片である）
loop_body(l)	l が繰り返し文の本体部の中に存在する
condition_part(l)	l が選択文または繰り返し文の条件部の中に存在する
with_in(l1, l2)	l1 は l2 に含まれている
before(l1, l2)	l1 は l2 よりプログラムテキスト上で前に存在する
same_function(p1, p2)	p1 と p2 は同じ関数の中に存在する
not_found(l)	l は存在しない
same_variable(p1, p2)	p1 と p2 は同じ識別子である

各要素（ベースパターン）は、and の関係にあり、それぞれのプログラム中での位置に関係なく、対応するプログラム断片が存在すれば良いことを意味する。label は、制約条件の記述において本体部中のベースパターンを識別する為の要素で、不必要な場合は省略可能である。また、ここでの parameter（実パラメータ）は、pattern_name で示される呼び出しベースパターンで定義されているパラメータ（仮パラメータ）と同じ文法カテゴリのパターン変数を記述しなければならない。

複合パターンにおける制約条件は、要素であるベースパターン相互の関係やパターンの位置に関して限定したい場合に利用し、ベースパターンで利用できる制約の他、テンプレート間の位置制約に関するものが含まれる。

複合パターンに固有な制約条件の例を表 2.3 に示す。パラメータ中のキーワード l はラベル (label)l で指示されるベースパターンにマッチするプログラム断片を意味する。制約条件 not_found(l) は、ある命令の存在ではなく、存在しない事を問題としてるパターンを探索するために必要となる条件で、これにより否定の表現が可能となる。

2.5 プログラムパターンの記述例

プログラミングスタイルの悪形パターンをプログラムパターンで記述した例を以下に示す。また、そこで利用されているベースパターンも一緒に示す。

プログラミングスタイル例1：

選択文、または繰り返し文の条件式の中で代入式を利用するのは副作用の観点から好ましくない。

プログラミングスタイル例1の診断のためのプログラムパターンは、side_effect1_pattern のように定義できる。

```
BEGIN
side_effect1_pattern;
BODY COMPLEX
  l1:assignment_pattern()
CONSTRAINTS
  condition_part(l1)
END
```

```
BEGIN
assignment_pattern;
BODY
   @assign
END
```

この side_effect1_pattern は、プログラム中の条件式の中で利用されている代入式とマッチする。condition_part(l1) は、パターン照合で見つかったプログラム断片（この場合は代入式）が条件式の中に存在する事を主張している。

例えば、以下のようなプログラム断片と照合する。

```
while( c = getc() )    中の    c = getc()
if(a = b)              中の    a = b
```

プログラミングスタイル例2：

ファイル処理に関して、プログラムが終了する時にオープンされたままのファイルは自動的にクローズされるが、明示的にクローズするのが好ましい。

プログラミングスタイル例2の診断のためのプログラムパターンは、file_not_colse_pattern のように定義できる。

```
BEGIN
file_not_close_pattern;
$v1,$v2
BODY COMPLEX
  l1:file_open_pattern($v1);
  l2:file_close_pattern($v2)
CONSTRAINTS
   found(l1),
   not_found(l2)
END
```

```
BEGIN
file_open_pattern;
$v1
BODY
  $v1 = fopen(#);
END
```

```
BEGIN
file_close_pattern;
$v1
BODY
 fclose($v1);
END
```

この file_not_close_pattern は、ファイルオープンのパターンは存在するが、ファイルクローズのパターンが存在していない時、ファイルのオープンを行なっているプログラム断片とマッチする。found(l1) は、l1 にマッチするプログラム断片（この場合は、ファイルの open 関数）が存在することを、not_found(l2) は、l2 にマッチするプログラム断片（この場合は、ファイルの close 関数）が存在しないことを意味する。file_not_close_pattern は、この二つの条件を満たすプログラム断片を検出し、下記のプログラム例 2.2 において（省略部分に、close 関数を含まない）、10 行目のプログラム断片とマッチする。

[プログラム例 2.2]

```
1   #include<stdio.h>
2    FILE *infile;
       .  .  .
10   infile=fopen("scores","r");
       .  .  .
```

しかしながら、このパターン表現ではプログラミングスタイル例 2 の検査を行なうという観点から、以下の問題が残る。

- プログラム中にファイルのクローズが一つも存在しない場合のみ有効で、複数のファイルをオープンしている場合には対応できない。
- ファイルポインタ変数について考慮していない。

具体的には、以下のようなプログラム例2.3(ファイル scores がオープンしたまま)において、10行目を検出することができない。

[プログラム例 2.3]

```
1    #include<stdio.h>
2     FILE *infile, *outfile;
        .  .  .
10    infile=fopen("scores","r");
11    outfile=fopen("grades","w");
        .  .  .
111    fclose(outfile);
        .  .  .
```

この原因としては、存在しないものを検索の対象としているため、file_close_pattern にマッチするプログラム断片が一つでも存在する場合、file_not_close_pattern の制約 (not_dound(l2)) を満たさないことになってしまうからである。また、ファイルポインタが対応しているオープン関数とクローズ関数を検索することは容易に行なえるが、存在した場合のみパターン変数の検査を行なうような記述は行なえない。

2.6 おわりに

本章では、プログラミングスタイルの個々のガイドラインを表現するためのプログラムパターンの記述形式について述べた。

プログラムパターンの記述を、テンプレートを記述する本体部とそのテンプレートに対する制約条件を記述する制約条件部に分けて表現することにより、文脈に依存したパターンやプログラムのフロー情報などを必要とするパターンであっても記述が行なえ、パターンの検索アルゴリズムが複雑になるのを避けることができた。さらに、制約関数の追加のみで、様々なパターンへの対応が可能である。

また、ベースパターンをテンプレートとして記述する複合パターンにより、離れた場所に分散して存在するパターンも記述できるという特色がある。

ある命令の存在ではなく、存在していないことを問題としているパターンも、特別な制約 not_found と複合パターンの組合せにより、一部については記述可能である。

第3章　プログラミングスタイルの診断システム

3.1　はじめに

本システムは、診断対象となるプログラムと検査したいスタイルガイドラインをユーザーに指定してもらい、プログラム中からガイドラインに違反する箇所を検出するというものである。

スタイルガイドラインは、あらかじめプログラムパターンの記述形式で表現し、データベース化されている。また、直接テキストエディタを用いて記述したものを検索することも可能である。

システムの基本動作としては、「入力として診断対象のプログラムと検索したいプログラムパターンが与えられと、プログラム中に存在する該当パターンの全数探索を行ない、その結果としてパターンとマッチしたプログラム断片の数とそれぞれの場所（プログラム中での行数）を表示する。」というものである。

本章では、まずシステムの概要について述べる。次に、プログラミングスタイルの診断システムの構成およびプログラムパターンの認識方法について述べる。最後に、システムの実行結果の例を示す。

3.2　システムの概要

本システムの実行画面イメージを図3.1に示す。

ユーザーインターフェイスは、各種ボタンのあるメニューと3つのウィンドウから構成されている。メニューには7つのボタンがあり、以下の機能から成る。

Program select 診断プログラムの選択

Pattern select 検索プログラムパターンの選択

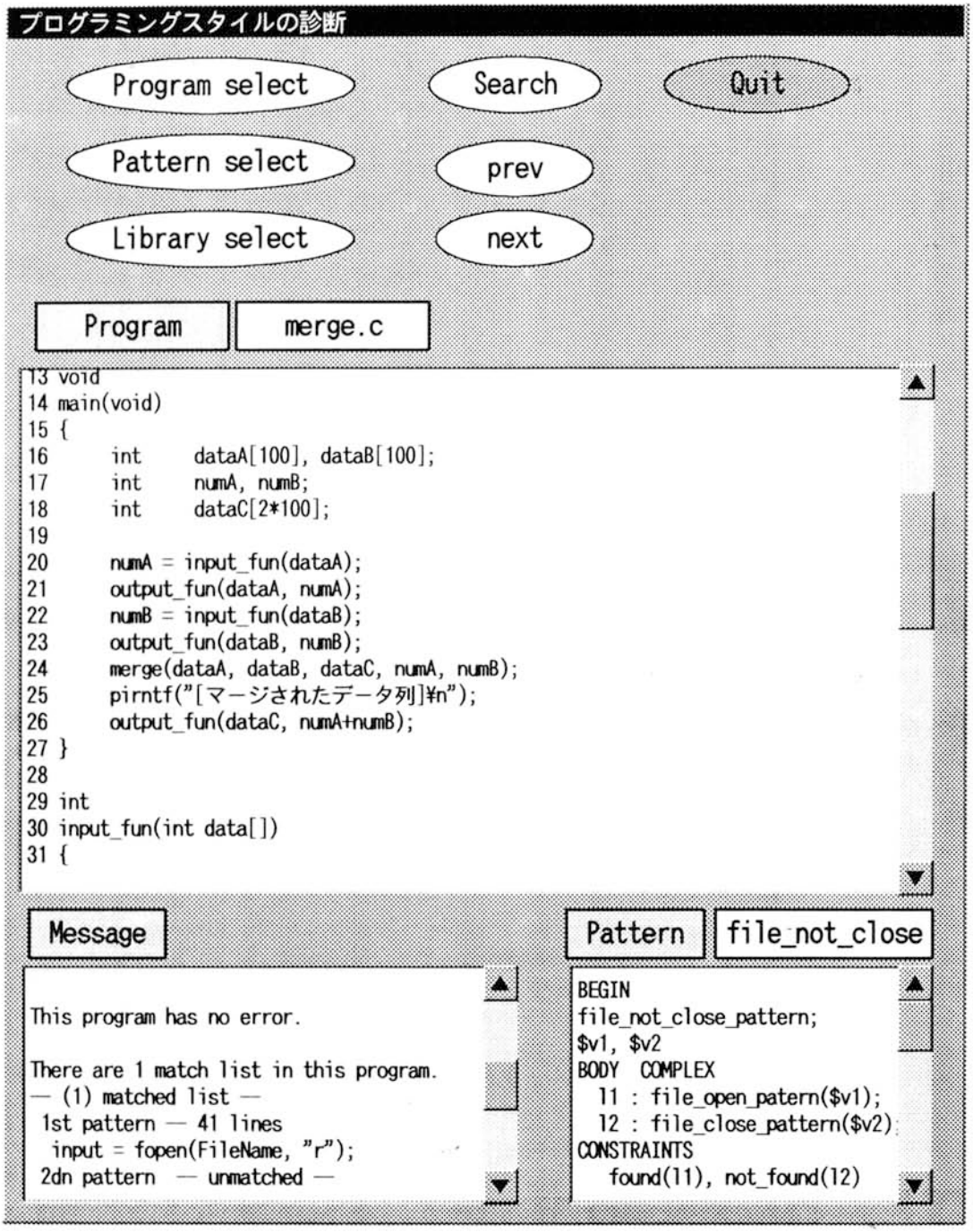

図 3.1: ユーザインターフェイス

Library select 検索ライブラリの選択

Search 診断の開始

prev 一つ前の検索結果へ

next 一つ後の検索結果へ

Quit 終了

また、3つのウィンドウはそれぞれ、Cのソースプログラム (中央)、検索プログラムパターン (右下)、診断メッセージ (左下) が表示される。

本システムは、UNIXのXウィンドウ上で実現している。このため、どのユーザーサイドからも利用可能であり、また異なったUNIXサイドへの移植も容易である。

ユーザは、まず[Program Select]ボタンを押し、診断対象となるプログラムを指定する。次に、探索したいプログラムパターンを選択する。[Pattern Select]ボタンを押すとポップアップウインドウが現れ、あらかじめパターンデータベースに登録されている検索パターンの名前の一覧が表示されるので、その中から検索パターンを指定する。検索パターンを表示するウインドウ (図3.1の右下) はテキストエディターとなっており、ウィンドウ上で直接パターンを編集することも可能である。そして[Search]ボタンを押すと、システムが診断を開始する。

診断が終了すると、診断メッセージウィンドウ (図3.1の左下) にメッセージが表示され、プログラムウィンドウ上の検索パターンにマッチしたプログラム断片がハイライトされる。複数のプログラム断片とマッチしている場合、[prev][next]ボタンで前後の照合箇所に移ることができる。

診断では、複数の検索パターンを一つのライブラリにあらかじめ登録しておくことにより、複数パターンの一括探索を行なうことができる（[Library Select]ボタンでライブラリの指定)。その場合は、診断終了後にプログラム上のマッチング箇所のハイライトは行なわれず、プログラムの診断メッセージがファイルに保存される。この一括探索は、あらかじめ決められた複数のプログラミングスタイルに関しての診断を行なう際に有効となる。

3.3 プログラムパターンの認識方法

システムの構成図を図3.2に示す。本システムは、パターンデータベース、入力されたプログラム、プログラムパターンを構文解析するパーサー、そして、2つのサブモジュール（ベースパターン探索モジュール、複合パターン探索モジュール）から構成される。

ベースパターンの探索においては、パーサーで解析されたプログラムとプログラムパターンがベースパターン探索モジュールに渡され、その match list が表示される。複合パターンの探索においては、パーサーで解析されプログラムパターンが複合パターン探索モジュールに渡され、個々のベースパターン毎に探索をおこなった後、その結果をもとに該当プログラムパターンの認識をおこない、match list を表示する。

以下の節で、ベースパターンの探索、複合パターンの探索それぞれの詳細について述べる。

3.4 ベースパターンの探索

ベースパターンの探索における処理手順を図 3.3 に示す。

入力されたプログラムは、パーサーで抽象構文木 (AST: attributed syntax tree) に変換され、ベースパターンテンプレートマッチングモジュールに渡される。探索対象となるプログラムパターン (ベースパターン）は、パーサーで本体部のテンプレートと制約条件部の制約条件の情報に分けられ、まずテンプレートの情報が、オートマトンジェネレータ (Automaton Generator) に渡される。ベースパターンテンプレートマッチングモジュールでは、オートマトンジェネレータで作成されたパターンオートマトン (pattern automaton) にプログラムの抽象構文木を入力として与え、状態遷移関数に基づき動作（シミュレーション）させる。プログラムパターンオートマトンが最終状態に達した時、プログラムパターンとマッチしたプログラム断片を'template match list' として受理する。ここでは、オートマトンを受理する全てのプログラム断片を探索する。'template match list' は、ベースパターン制約チェックモジュールで、プログラムパターンの 制約条件部 に記述された制約条件の検査が行なわれ、そのうち全ての制約条件を満たすものが、検索プログラムパターンの'Base Pattern matche list' として出力される。

本システムにおけるテンプレートマッチングの方法は、Michigan 大学の S.Paul ら

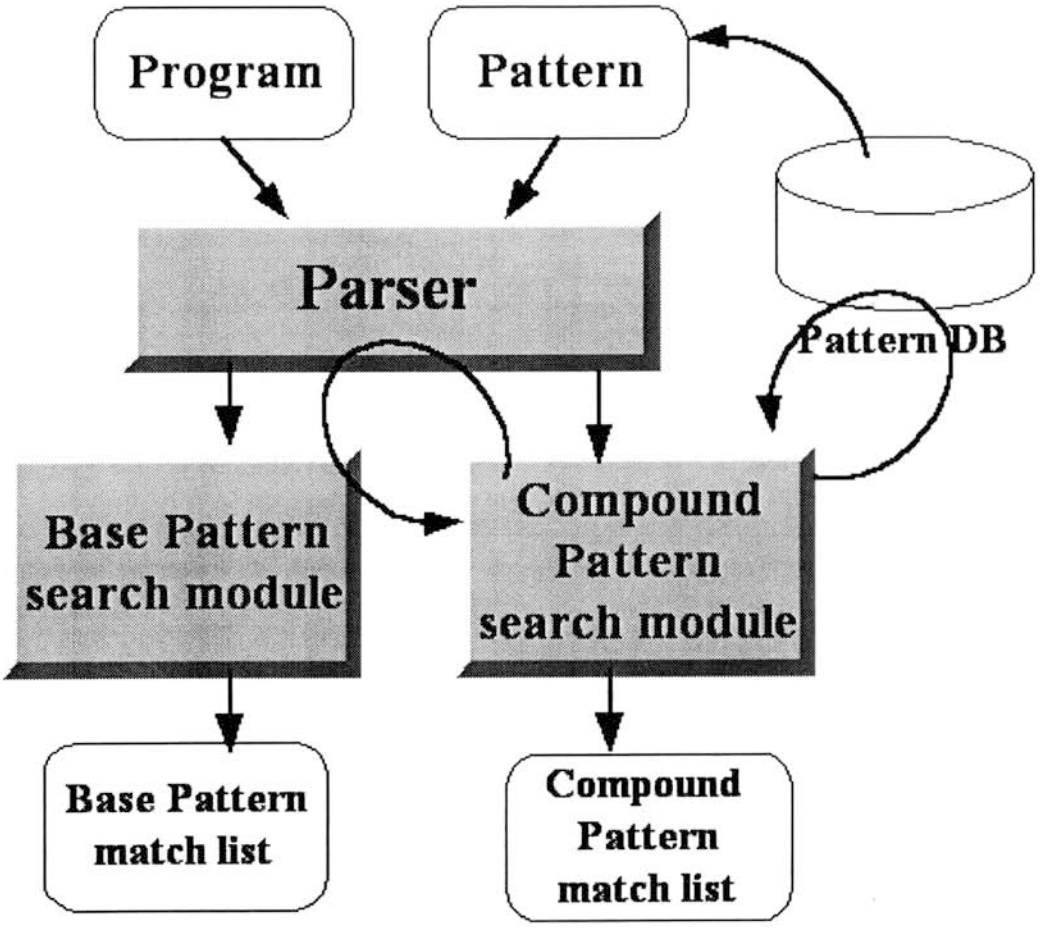
Program
Pattern
Parser
Pattern DB
Base Pattern search module
Compound Pattern search module
Base Pattern match list
Compound Pattern match list

図 3.2: システム構成図

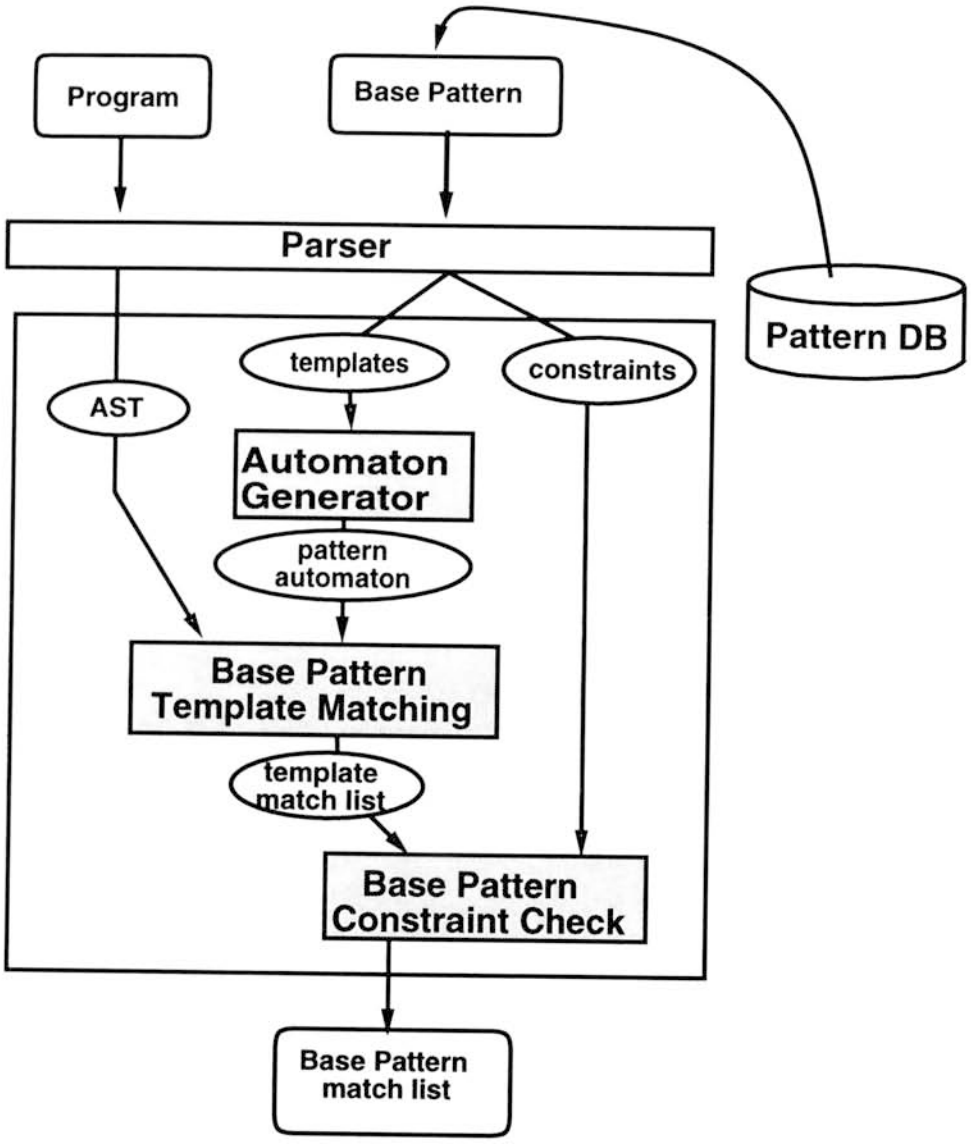

図 3.3: ベースパターン探索の流れ

が提案する有限状態マシーンに基づくソースコード探索の枠組 [15] を応用している。

3.4.1 プログラム解析木

パーサー は、プログラムを解析して抽象構文木を作る。本システムにより作成される抽象構文木の例を図 3.4 に示す。

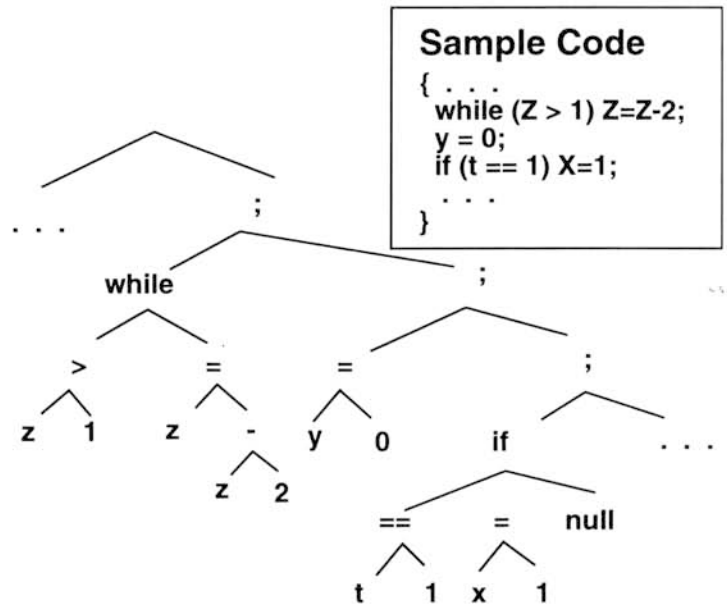

図 3.4: 抽象構文木の例

これは、右上のサンプルコードに対する抽象構文木の例である。抽象構文木の各ノードは、C 文法の終端記号および非終端記号からなる。

3.4.2 パターンオートマトン

本システムで利用するパターンオートマトンは、非決定性の有限状態オートマトンを発展させたものである。

パターンオートマトンは、次のような 6 個組 $< Q, \Sigma, A, \Gamma, q_0, F >$ で定式化する。

(1) Q は状態の有限集合

(2) Σ は構文要素を表す AST のノードから構成されれ有限の入力アルファベット

(3) A は AST のナビゲーション関数の集合で、以下によって与えられる。

$A = \{ mlc, mpmrs \}$

mlc は、*moveto_leftchild* の省略で、名前の通り左の子どもへ移動することを意味する。*mpmrs* は、*moveto_parent;...;moveto_rightsibling* の省略で、*moveto_parent* の適用は、現在の *node* と次の *node* の高さの差の数だけ必要となる。つまり、*mpmrs* は、*rightchild* を持つ *parent* まで移動し、さらにその右の子どもに移動することを意味している。

(4) Γ は遷移関数の集合で、以下によって与えられる。

$\Gamma = \{ CAT, VAL \}$

- *Arc Label:*

  $CAT < category >; ACTION < actions >$

  CAT アークは、入力が属する構文カテゴリーを指定する。入力される要素が指定された *category* の中の要素である時、状態は遷移される。例えば、もし if-statement が現れたとすると、$CAT < statement >$ で指定されたアークは、遷移される。

- *Arc Label:*

  $VAL < value >; ACTION < actions >$

  VAL アークは、入力列の値が $< value >$ と照合できた場合、遷移される。

(5) q_0 は Q の元で初期状態である。

(6) $F \subseteq Q$ は、最終状態の集合である。

初期状態から最終状態に導くような入力 AST に対応する遷移の列が存在する場合、AST は PA によって入力として受理される。

3.4.3 オートマトンジェネレータ

オートマトンジェネレータへの入力は、パーサー でテンプレート解析木に変換されたプログラムパターンの本体部の情報である。この解析木を pre-order で走査しながら、パターンオートマトンを作成する。

オートマトンジェネレータは、現在の状態とノードのラベルにより次の状態と動作を決定していく。ノードのラベル別によるオートマトンの生成についての例を図3.5に

示す。例えば、ノードのラベルが単一の wildcard の場合、図 3.5(a) のような遷移となる。また、 ある statemant st の後に、0 文以上の statement [@*] が続くようなパターンの場合、これとマッチするプログラムには図 3.5(b) のような 3 つのパターンがあり、その状態遷移は 図 3.5 (c) のように表せる。

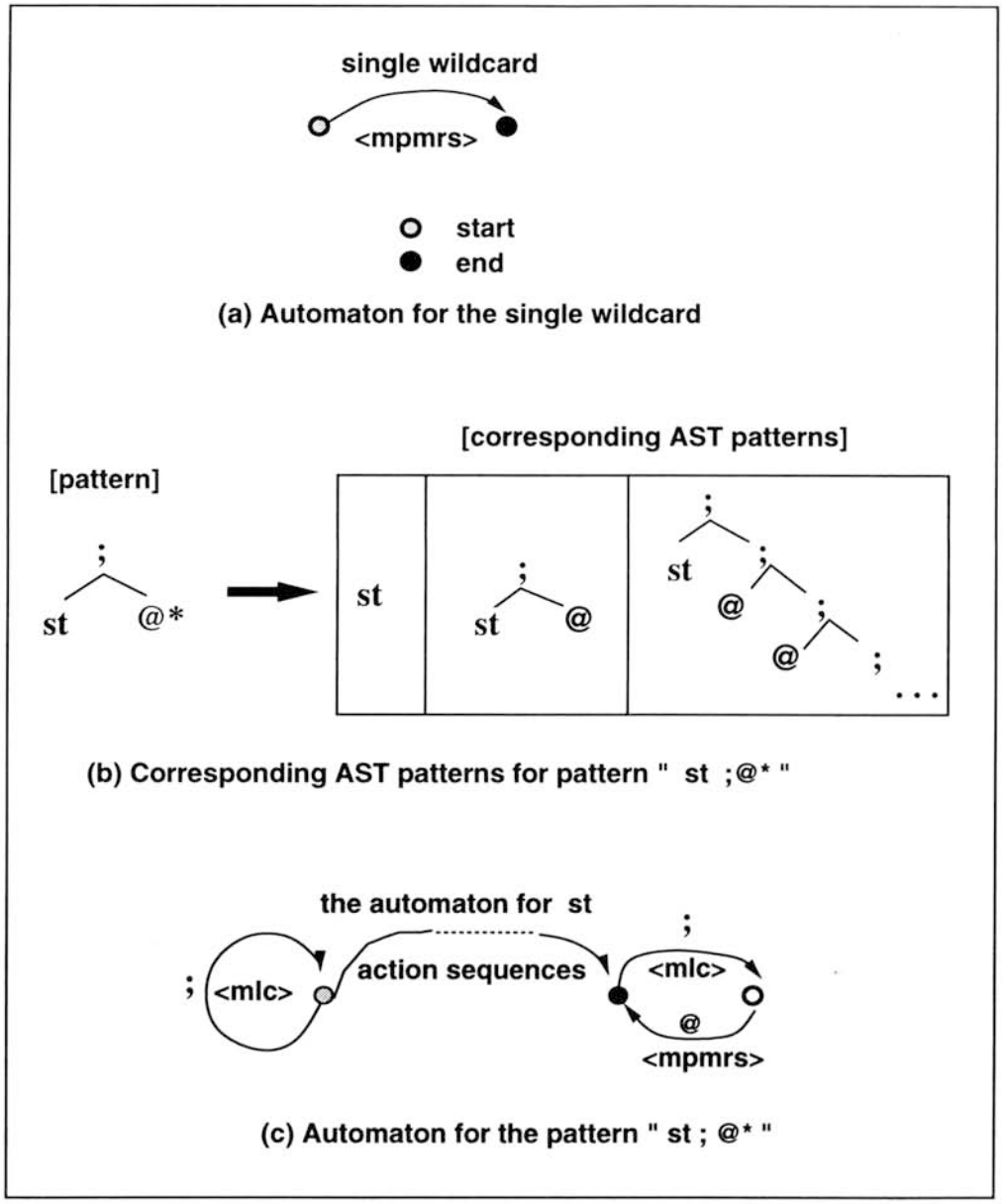

図 3.5: オートマトンの生成の例

図 3.6 に、以下のパターンに対応するパターンオートマトンの例を示す。

```
BODY:  while ( # ) @
```

```
@*
if (#) x = 1;
```

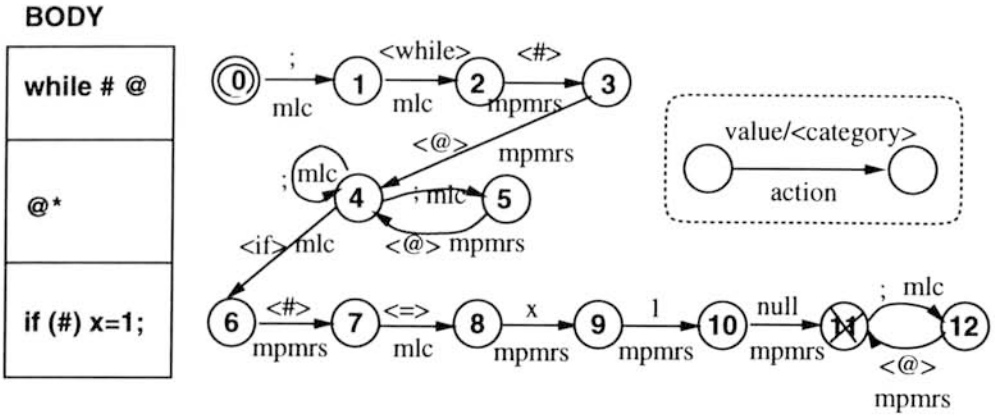

図 3.6: パターンオートマトンの例

3.4.4 テンプレートマッチング

ベースパターンのテンプレートマッチングモジュールでは、パターンオートマトンによりプログラム解析木 (AST) を走査して、template match list を生成する。

本システムでは、正規表現のパターンを利用できる（例えば、@*)。このため、生成されるパターンオートマトンは非決定性となる。そこで、非決定性に伴うバックトラックの処理を行なうために、以下のような 2-tuple からなる姿態 (configuration) を利用する。

```
conf = < PA_node, AST_node >
```

configuration は、ある状態における状態遷移関数である PA_node と、次の状態へ遷移可能な（その関数が適用できる）AST_node の組で表わされる。非決定動作を含む場合、この configuration をスタックに格納していき、このスタックが空になるまで backtrack をシミュレートすることにより、複数の match list が作られる。

また、パターンが named wildcards の場合の処理を行なうために、バインディングテーブルを利用する。VAL が named wildcards の場合、まず VAL の属性と AST の

node の属性との照合を行なう。これに成功したら、VAL の値が既にバインディングテーブルに登録されているか、いないか（bind されているか、されていないか）の検査をおこない、登録されていなければプログラムのノードの情報を VAL の値として登録する。既に登録されている場合、bind されている値と、AST のノード情報との照合を試みる。非決定性に伴うバックトラックの際には、戻った場所に応じて bind の解消を行なう。

3.4.5 ベースパターンの制約検査

match list は、AST 上のノードとパラメータに関する情報（named wildcards に関する bind 情報）からなる。

ここでは、探索プログラムパターンの制約条件部に記述された制約各々について、match list の情報が適合するかどうかの検査を行なう。例えば、型制約であれば、named wildcards に bind された変数名からその型を調べ検査する。フロー制約は、プログラムのフロー解析情報を利用する。

3.5 複合パターンの探索

複合パターンの探索の処理手順を図 3.7 に示す。

パーサーで解析された複合パターンの本体部であるテンプレートは、テンプレート解析木として、複合パターンテンプレート認識モジュールに入力される。

複合パターンテンプレートマッチングモジュールでは、テンプレート解析木の node がベースパターン呼びだしの場合、プログラムパターンデータベースから該当ベースパターンを呼びだし、パーサーを通し、ベースパターン認識モジュールへと引き渡す。そして、ベースパターン探索モジュールからその出力結果である match list を受けとる。全てのベースパターンの match list を受理したら、その match list の list を複合パターン制約チェックモジュールに渡す。

複合パターン制約チェックモジュールでは、個々のベースパターンの match list の組合せを作り、全ての組合せ（直積）に対し、個々の制約条件の検査を行なう。例えば、ベースパターンが 3 つあり、各々に対して 5 つのプログラム断片が見つかったとすると、その全ての組合せ (125 通り）に対して複合パターンの制約条件を検査する。全ての制約条件を満たした組合せのみが、そのプログラムパターンの match list となる。

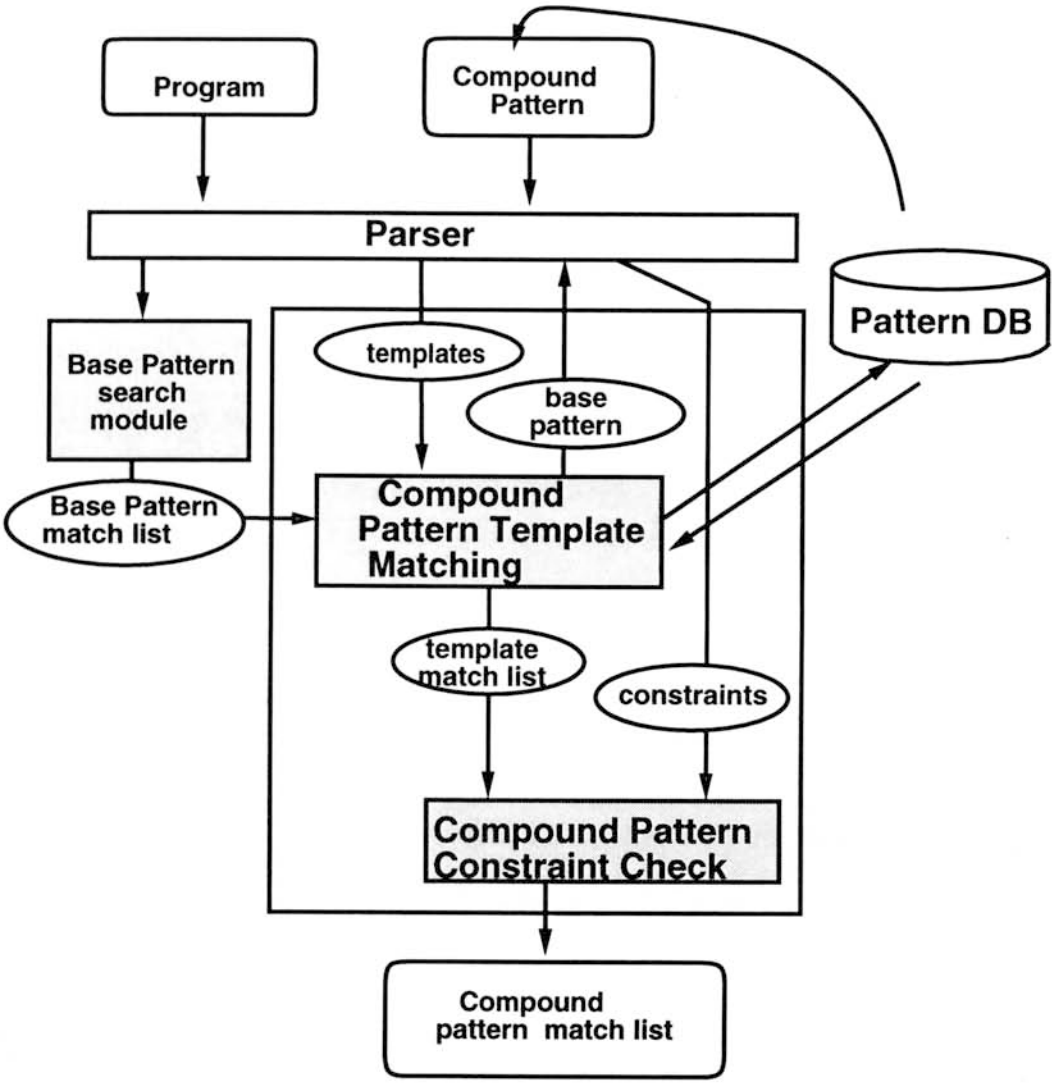

図 3.7: 複合パターン認識の流れ

3.6 実行結果

システムによる診断結果の一例を以下に示す。

検索 [1]: ファイル処理に関するガイドライン

マッチング箇所が (1) つ見つかりました。

オープンされたままのファイルがあります。ファイル処理において、「ファイルは読み書きを行なう前にオープンし、使い終ったらクローズする」というの基本です。プログラムが終了する時にオープンされたままのファイルは、自動的にクローズされますが、明示的にクローズするように心がけましょう。

——(1) 番目の照合リスト——-
(1) th pattern ——- 41 行目
input = fopen (FileName , "r") ;
(2) th pattern == unmatched ==

システムは、診断メッセージとして対象としたスタイルガイドラインの名称、マッチング数（違反の数)、違反したスタイルガイドライン関するアドバイス、マッチングしたプログラム断片の場所（ベースパターンの場合は照合した部分のプログラム行数、複合パターンの場合は各要素のベースパターンごとに行数とプログラム断片）を表示する。

3.7 おわりに

本章では、プログラミングスタイルの診断システムにおけるプログラムパターンの認識方法について述べた。

提案したシステムでは、個々のガイドライン（各悪形パターン）を一つのプログラ

ムパターンとしてデータ化し、これを用いてソースコードの探索を行なう。このような方式を採用することにより、プログラミングスタイルの検査項目の追加が容易に行なえ、拡張性に優れているといえる。また、ベースパターンのテンプレートマッチングでは、効率よくマッチングパターンを見つけるために、有限状態オートマトンを利用した方法をとっている。プログラムパターンの記述がテンプレートと制約条件に分かれているため、複雑なパターンでも制約関数の実現で対応でき、アルゴリズムがシンプルなものとなっている。

第4章　プログラムパターンの記述実験および認識実験

プログラミングスタイルの検査において、プログラムパターンの認識に基づく手法が有効であるかを評価するために、以下の二つの実験を行なった。一つは、プログラムパターンの記述能力を評価するための記述実験、そして、もう一つは定義したプログラムパターンが、実プログラムにおいて意図した通りのプログラム断片を検索することができるか調査するための認識実験である。

本研究では、現在C言語を対象としているが、C言語は柔軟な構造をもつため、構文上のわずかな違いによりプログラマの意図とは全く異なる動作を行なうプログラムとなってしまうことがある。このようなC言語特有の問題は、プログラミングスタイルに関する悪形パターンの利用からくるものもあり、これらを取り除き、プログラムを明快に記述することで避けられるものも多い。このような構文上の些細な相違に由来するトラブルの事例を集めた書に、文献[18](Cプログラムの落し穴)がある。これには、スタイルガイドラインの違反とは少し異なる意味的なバグも含まれるが、可読性、了解性の観点からは密に関係のある問題であり、本実験ではこの文献中の事例を扱う。また、本学のプログラミング演習で利用しているプログラミングスタイルの項目も対象とする。文献[18]では、トラブルの事例を、7つのカテゴリ（語彙的、構文的、意味的、リンケージ、ライブラリ関数、プリプロセッサ、移植性）に分類して集めてある。本実験では、このうち構文的、意味的な落し穴を対象としプログラムパターンの記述および認識を行なった。

認識実験では、プログラミングスタイルとして、記述実験で定義したプログラムパターンを利用する。診断対象プログラムは、UNIXのGNU版テキストユーティリティのパッケージプログラム群(cat, head, wc等)の22種類のプログラムとする。

4.1 記述実験

4.1.1 実験目的

本記述実験の目的は以下の通りである。

- プログラムパターンの表現形式で、標準的なプログラミングスタイルに関する検査項目の記述が行なえるか調べる。
- プログラムパターンとして記述できないプログラミングスタイルには、どのような項目があるか探る。

4.1.2 実験方法

以下のプログラミングスタイルを対象とし、各項目をプログラムパターンとしてデータ化する。

(1) 文献[18]中に列挙されている構文的な落し穴(以下の6種類)
宣言の書き方、演算子の優先度、セミコロンの問題、break文のないcaseラベル、関数呼び出しの問題、ぶらさがりelse問題

(2) 文献[18]中に列挙されている意味的な落し穴(以下の7種類)
ポインタ変数の利用、数え上げと非対象な境界、評価順序、代入における型の不一致、ポインタ同士の代入、整数のオーバーフロー、mainからの返却値の有無

(3) 信州大学工学部情報工学科のプログラミング演習において利用しているプログラミングスタイルのうち構文レベルのもの(以下の6種類)
副作用を伴うもの、条件式に関する問題、for文の省略に関する問題、定数に関する問題、EOF検査の問題、ファイル処理に関する問題

4.1.3 実験結果

4.1.3.1 構文的な落し穴

文献[18]中の構文的な落し穴では、全6種類中5種類の項目を定義することができた。データ化を行なったのは、演算子の優先度、セミコロンの問題、break文のないcaseラベル、関数呼び出しの問題、ぶらさがりelse問題の5種類についてで、12個の

プログラムパターンを定義した。定義したプログラムパターンの一覧を表4.1に示す。データ化が行なえなかったのは、「複雑な宣言は、typedefを使って簡単にする方がよい」という宣言の書き方に関するもので、現在システムにおいて宣言部分を対象としたパターンの認識を実現していないために扱えない。ただし、パーサーによる解析は行なっているので拡張可能である。

構文的な落し穴に関するプログラムパターン定義の例を以下に示す(priority1_pattern)。

演算子の優先度に関する例

ガイドライン：等価式を含む代入文は好ましくない。代入は、等価演算子より優先度が低いため、比較の後に行なわれる。

```
BEGIN
priority1_pattern;
#v1
BODY COMPLEX
  l1:assignment_pattern();
  l2:equality_exp_pattern(#v1)
CONSTRAINTS
  with_in(l2, l1)
END
```

priority1_patternは、等価式を含む代入文を表す。

以下のプログラムは、一つのファイルからもう一つのファイルへコピーするループの例である。while文中の式は、cにgetc(in)の値を代入し、ループを終了するかどうか決めるためのEOFと比較しているようにみえるが、代入はどの関係演算子よりも低い優先度をもつので、cの値はget(in)とEOFの比較結果となる。

```
while ( c=getc(in) != EOF)
      putc(c, out);
```

この例は意味的なバグではあるが、このような紛らわしいパターンを使わないことで回避できる。priority1_patternは、上のプログラムの例では以下のプログラム断片と照合する。

```
c=getc(in) != EOF
```

表 4.1: 構文的な落し穴において定義したプログラムパターン

スタイル	Pattern Name	説明
演算子の優先度	priority1_pattern	等価式を含む代入文
	priority2_pattern	関係式を含む代入文
セミコロンの問題	extrasemi_if_pattern	if 文の条件節の括弧のすぐ後にセミコロンがある
	extrasemi_while_pattern	while 文の条件節の括弧のすぐ後にセミコロンがある
	extrasemi_for_pattern	for 文の条件節の括弧のすぐ後にセミコロンがある
	return_asgn_pattern	オペランドが代入式である return 文
break 文のない case ラベル	case1_pattern	case ラベルの前に、switch 文を抜けるための break 文がない
	case2_pattern	default ラベルの前に、switch 文を抜けるための break 文がない
関数呼び出し	func_call_pattern	引数リストのない関数呼びだし
ぶらさがり else 問題	hangelse_pattern	else 部をもたない if 文の中に、else 部をもつ if 文がくる
	hangelse_for_pattern	else 部をもたない if 文の文中に、else 部をもつ if 文を実行文とする for 文がくる
	hangelse_while_pattern	else 部をもたない if 文の文中に、else 部をもつ if 文を実行文とする while 文がくる

表 4.2: 意味的な落し穴において定義したプログラムパターン

スタイル	Pattern Name	説明
ポインタ変数の利用	alloc_check_pattern	領域を確保できなかった場合の処理がない (malloc の後に if 文がこないパターン)
	not_free_pattern	領域の解放がない (free 関数が一度も利用されていないパターン)
	alloc_not_len1_pattern	malloc で strlen を利用しているが+1 されていない。
	alloc_not_len2_pattern	同上 (strlen を 2 つ利用の場合)
数え上げと非対象な境界	for_lessthan_pattern	for 文の上限に用いられている関係演算子が ⟨= のパターン
評価順序	side_effect3_pattern	配列の添字に increment operator か decrement operator を含む配列への代入文
	side_effect4_pattern	代入式に increment operator か decrement operator を含む

4.1.3.2 意味的な落し穴

意味的な落し穴については、悪形事例を示している 7 種類の内、3 種類について定義した。データ化を行なったのは、ポインタ変数の利用、数え上げと非対象な境界、評価順序の 3 種類で、7 個のプログラムパターンを定義した。定義したプログラムパターンの一覧を表 4.2 に示す。定義を行なっていない種類の内、代入における型の不一致、ポインタ同士の代入、整数のオーバーフローについての 3 種類は、構文的な落し穴で扱えなかったのと同様の理由で、宣言部分を対象としていないためである。また、残りの 1 種類 (main からの返却値の有無) については、関数単位にその返却値の型と値を調べ、その整合性を検査するというもので、手続き的になら検査可能だが、プログラムパターンとして定義するのには不向きであると考えた。

意味的な落し穴におけるプログラムパターン定義の例を以下に示す (side_effect4_pattern)。

評価順序に関する例

ガイドライン：代入式の中に increment や decrement の演算子を含むのは、副作用

の観点から好ましくない。プログラムをシンプルに保つために、increment 演算子と decrement 演算子は常にそれ自身を単独の文で使うようにした方がよい。

```
BEGIN
side_effect4_pattern;
#v1,#v2
BODY COMPLEX
  l1:asign_exp_pattern(#v1);
  l2:incre_op_pattern(#v2)
CONSTRAINTS
  with_in(l2, l1)
END
```

side_effect4_pattern は、increment 演算子を中に含む代入式を表す。

以下は、配列 x の最初の n 要素を配列 y へコピーするプログラムの例である。

```
i = 0;
while (i < n )
   y[i] = x[i++];
```

これは、評価順序についてあまりにも多くの仮定をしているため、好ましくない。i がインクリメントされるより前に、y[i] のアドレスが評価されることの保証がなく、ある処理系では y[i] のアドレス評価が先かもしれないし、他の処理系では i のインクリメントが先かもしれない。

side_effect4_pattern は、上のプログラムの例では以下のプログラム断片と照合する。

```
   y[i] = x[i++];
```

4.1.3.3 演習で利用しているプログラミングスタイル

この他、本大学のプログラミング演習において利用しているプログラミングスタイルのうち、構文レベルの 6 種類について、19 個のプログラムパターンを定義した。定義したプログラムパターンの一覧を表 4.3 に示す。

表 4.3: プログラミング演習において利用しているプログラミングスタイルに対して定義したプログラムパターン

スタイル	Pattern Name	説明
副作用を伴うもの	side_effect1_pattern	条件式に代入文を含む
	side_effect2_pattern	条件式に increment operator か decrement operator を含む
	side_effect5_pattern	代入文の中に代入文を含む
	multi_asign_pattern	複数の関数で定義されているグローバル変数がある
条件式に関する問題	fun_if_condition1_pattern	if 文の条件式が関数呼び出し（引数なし）であるパターン
	fun_if_condition2_pattern	if 文の条件式が関数呼び出し（引数あり）であるパターン
	fun_while_condition1_pattern	while 文の条件式が関数呼び出し（引数なし）であるパターン
	fun_while_condition2_pattern	while 文の条件式が関数呼び出し（引数あり）であるパターン
for 文の省略に関する問題	abb_for1_pattern	for(式 1; 式 2; 式 3) 文において式 1 が省略
	abb_for2_pattern	for(式 1; 式 2; 式 3) 文において式 2 が省略
	abb_for3_pattern	for(式 1; 式 2; 式 3) 文において式 3 が省略
	abb_for4_pattern	for(式 1; 式 2; 式 3) 文において式 1, 式 2 が省略
	abb_for5_pattern	for(式 1; 式 2; 式 3) 文において式 1, 式 3 が省略
	abb_for6_pattern	for(式 1; 式 2; 式 3) 文において式 2, 式 3 が省略
	abb_for7_pattern	for(式 1; 式 2; 式 3) 文において式 1, 式 2, 式 3 が省略
定数に関する問題	constant_undef_pattern	ある大きさ以上（プラン内で指定）の定数が複数存在するパターン
EOF 検査の問題	eof1_scan_pattern	EOF の検査を feof 関数で行ない（条件が!feof() の場合）、fsanf 関数で読み込んだのデータに対して EOF の検査を行なっていない
	eof2_scan_pattern	eof1_scan_pattern と同様で、EOF の検査において条件が feof()==0 の場合
ファイル処理に関する問題	file_not_close_pattern	オープンされたままのファイルがあるパターン

プログラミング演習で利用しているプログラミングスタイルに対して定義したプログラムパターンの一例を以下に示す (multi_asign_pattern)。

副作用に関する例

ガイドライン：異なる関数の中でグローバル変数の値をそれぞれに定義すると、副作用の可能性があるので好ましくない。グローバル変数は、一つの関数内のみで定義するのが好ましい。

```
BEGIN
multi_asign_pattern;
$v1,$v2
BODY COMPLEX
  l1:asign_pattern($v1);
  l2:asign_pattern($v2)
CONSTRAINTS
  node_literal_equal($v1,$v2),
  same_declared($v1,$v2),
  different_function($v1,$v2),
  before_text(l1, l2)
END
```

multi_asign_pattern は、同一の変数に異なる関数において代入をおこなっている場合、それぞれの代入式を表す。

4.1.3.4 まとめ

本記述実験において、文献 [18] 中に列挙されている構文的な落し穴について 12 個、意味的な落し穴について 7 個、本学のプログラミング演習で利用しているプログラミングスタイルについて 19 個、合計 38 個のプログラムパターンを定義した (表 4.1, 表 4.2, 表 4.3 参照)。

4.2 認識実験

4.2.1 実験目的

本認識実験では、プログラミングスタイルの検査において、プログラムパターンの認識に基づく手法が有効かを調査することを目的とする。

具体的には、以下の2点について調査を行なう。

- プログラミングスタイルに関して記述したプログラムパターンが、実プログラムにおいてプログラムパターンの定義者の意図した通りのプログラム断片を検索することができるかどうか。
- プログラムパターンの定義者の意図したプログラム断片以外のものとマッチング(ミスマッチ)したプログラムパターンについて、意図通りのマッチングが行なえるよう、記述の変更のみで容易に対処できるのか?

4.2.2 実験方法

UNIXのGNU版テキストユーティリティのパッケージプログラム群(cat, head, wc等)の22種類のプログラムをプログラミングスタイルの診断対象プログラムとし、各々のプログラムに対し、記述実験で定義した38個のプログラムパターンの認識を行なう。

各プログラムに対し、マニュアルで対象としたプログラムパターンに相当するプログラミングスタイルの検査を行ない、システムの認識結果と比較する。プログラムパターンの定義者の意図したプログラム断片以外のものとマッチング(ミスマッチ)したプログラムパターンについて、意図通りのマッチングが行なえるよう、プログラムパターンの記述を修正する。再度、修正したプログラムパターンについて全22種類のプログラムでの認識を試みる。

対象プログラム

対象としたのは、UNIXのGNU版テキストユーティリティの22のコマンドに関する以下のプログラムである。

cat.c, cksum.c, comm.c, csplit.c, cut.c, expand.c, fold.c, head.c, join.c, nl.c, od.c, paste.c, pr.c, sort.c, split.c, sum.c, tac.c, tail.c, tr.c, unexpand.c, uniq.c, wc.c

なお、これらのプログラムのシステムでの構文解析後(コメントを除く)のプログラム行数は、平均497行(最小107行、最大1354行)である。

4.2.3 実験結果

各プログラムパターンに対する22のプログラムのマッチング数の合計を表4.4に示す。

1つのプログラムにおける延べマッチング数は平均 45.7(最大162、最小7)、また、1つのプログラムパターンの延べマッチング数は平均 26.5(最大195、最小0) であった。

4.3 評価・考察

本実験より、テキスト上の単純なパターンマッチングではマッチングが難しい、以下に挙げるような各種パターンをプログラムパターンとして容易に記述が行なえ、探索可能であることが確認できた。

- 文脈依存なパターン

  (e.g. side_effect_pattern, priority_pattern)

- 複数の関数にまたがるパターン

  (e.g. multi_assign_pattern, constant_undef_pattern)

- ある命令の存在ではなく、存在しないことを問題としたパターン

  (e.g. not_free_pattern, file_not_close_pattern, etc.)

これより、本システムがプログラミングスタイルの悪形パターンの検出に有効な手法であると言える。

認識実験において、プログラムパターン定義者の意図したプログラム断片以外のものとのマッチング (ミスマッチ) は、制約の調整で対処できた。最初の実験において、いくつかのプログラムパターンで冗長なマッチングが生じた。ここに、二つの例を挙げる。

- 2つの同じベースパターンを要素とする複合パターンは (例えば、multi_assign_pattern, constant_undef_pattern)、マッチしたプログラムパターンの順序が逆のみで、プログラム断片は全く等しいマッチングリストも検索してしまうという問題が生じた。これについては、プログラムテキスト上での位置を制限する新たな制約関数を準備することで、冗長なマッチングをなくすことができた。

表 4.4: 認識実験における各プログラムパターンのマッチング総数

Pattern Name	マッチング数
side_effect1_pattern	52
side_effect2_pattern	49
side_effect3_pattern	18
side_effect4_pattern	117
side_effect5_pattern	59
multi_asign_pattern	187
priority1_pattern	7
priority2_pattern	1
hangelse_pattern	1
hangelse_for_pattern	0
hangelse_while_pattern	0
case1_pattern	37
case2_pattern	4
extrasemi_for_pattern	8
extrasemi_if_pattern	0
extrasemi_while_pattern	4
return_asgn_pattern	0
func_call_pattern	0
fun_if_condition1_pattern	195
fun_if_condition2_pattern	68
fun_wh_condition1_pattern	7
fun_wh_condition2_pattern	6
abb_for1_pattern	20
abb_for2_pattern	1
abb_for3_pattern	1
abb_for4_pattern	0
abb_for5_pattern	0
abb_for6_pattern	1
abb_for7_pattern	16
eof1_scan_pattern	0
eof2_scan_pattern	0
alloc_check_pattern	0
not_free_pattern	1
file_not_close_pattern	2
alloc_not_len1_pattern	0
alloc_not_len2_pattern	0
constant_undef_pattern	130
for_lessthan_pattern	14
合計	1006

- for 文において条件節の一部、または全てを省略しているプログラム箇所を検索するプログラムパターン (abb_for1_pattern ~ abb_for7_pattern, 表4.3 参照) の定義を、図4.1の左のように本体部のテンプレート表現のみで定義していたところ、7つのプログラムパターン全てに無限ループである for(; ;)がマッチしてしまった。これは、本来 abb_for7_pattern にのみマッチするべきプログラム断片である。この原因は、for(; ;) はプログラム内部表現において、for(null-exp; null-exp; null-exp) となっており、abb_for1_pattern ~ abb_for6_pattern において式のパターン変数 # が null-exp とマッチしてしまうためである。そこで、引数のパターン変数にマッチするプログラム断片が存在する、しない、という2つの制約 (null, not_null) を新たに追加し、これを利用して図4.1の右のよう再定義することによって意図したパターンのみにマッチングさせることが可能となった。

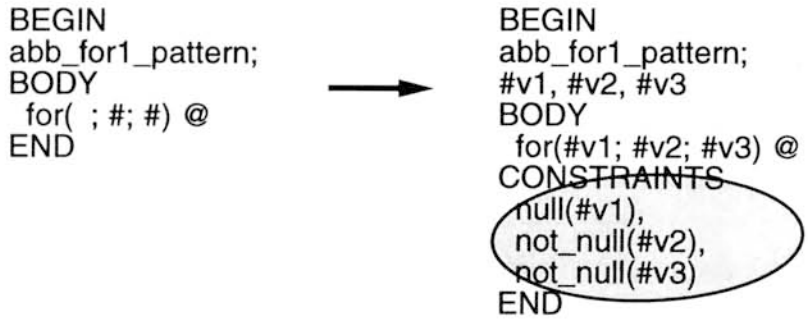

図 4.1: abb_for1_pattern の再定義の例

これらの例から、制約の調整により冗長なマッチングを減らすことが可能であることが確認できた。

また、制約関数は、基本的には対象となるプログラム断片（の解析木）の走査であるので、関数自体の実現は容易であり、またシステムへの追加も関数のみの追加であるので容易に実現できた。このことからシステムの拡張容易性が言える。

第5章　初心者プログラマによるシステムの評価実験

5.1　はじめに

既存の検査ツール [5, 6, 12] では、システムの検査能力の評価を行なっているものはあるが、プログラマがシステムを実際に利用し評価を行なっているものはない。我々は、初心者プログラマが本プログラミングスタイルの診断システムを利用することにより本当に支援効果が得られるのか調査することを目的とし、二つの評価実験を行なった。

評価実験1では、システムを利用することによる直接的な効果に焦点を当て、86名を対象に実験を行なった。評価実験2では、システムを利用することによる教育的効果に焦点を当て、9名を対象に実験を行なった。

以下、本章では二つの評価実験について、その目的、方法、結果、および考察について述べる。

5.2　評価実験1

5.2.1　実験目的

本実験では、システムを利用することにより直接的な効果が得られるか調査することを目的とする。具体的には、システムを利用した場合とシステムを利用しない場合で、プログラミングスタイルの検査において、悪形箇所の検出における検出率、悪形の修正における正解率、検査にかかる時間等に差異があるのか調査する。

5.2.2　実験方法

プログラミングスタイルの診断システムの有効性を評価するために被験者を2つの組に分け、片方の組にはシステムを利用させプログラミングスタイルの検査を行なわ

せ、もう片方の組はシステムを利用しないで、同様の検査を行なわせる。そして、システムを利用した組と利用しない組との検査結果を比較する。

まず、被験者を2組のグループ（A組, B組）に分ける。組分けをするための、簡単なプログラミングに関する能力テストを行なう。ここでは、簡単なプログラム作成問題を出題し、プログラムの考え方、完成度を基準に、優、良、可の3段階の評価を行なう。そしてこの評価に基づき、なるべく同等のレベルになるよう組分けをする。

そして、いくつかの指定されたプログラミングスタイルの違反をソースプログラム中から検出し、修正してもらうという実験を、A組はプログラミングスタイルの診断結果 (付録A.3参照) を参照しながら、B組はシステムの結果を利用しないで行なう。

実験では、プログラムとプログラミングスタイルの記述された用紙を配布し、悪形パターンの検出、プログラムの修正は直接プログラム上に書き込むものとした。

実験対象者

被験者は、信州大学工学部情報工学科の2年生86人で、1年間のプログラミング演習を受講している学生である。

対象プログラム (付録A.1参照)

検査対象としたプログラムは、ファイルに記述されたデータ2組のマージを行なう100行程のプログラムである。プログラミングスタイルの部分のみを検査対象とするため、プログラムは、被験者が過去に演習問題として取り組んだことのある問題を採用した。これにより、プログラムの理解に時間を費やすことなく、プログラミングスタイルの検査に集中できると考えた。

このプログラムに、実際に学生が作成したプログラム中に存在したプログラミングスタイルの違反を5種類埋め込んだものを、検査対象のプログラムとした。違反の種類は、下記に挙げた対象プログラミングスタイルの1から5までのガイドラインである。

対象プログラミングスタイル (付録A.2参照)

検査するよう指示したプログラミングスタイルは、以下の6種類のガイドラインである。

(1) 定数に関するガイドライン
(2) 条件式の副作用に関するガイドライン

表 5.1: 評価実験1の組分け問題における結果

	A組(人)	B組(人)
優	31	32
良	8	5
可	4	6
人数	43	43

(3) インクリメント/デクリメント演算子の利用に関するガイドライン
(4) ファイル処理に関するガイドライン
(5) 条件節に関するガイドライン
(6) for 文の利用に関するガイドライン

検査対象としたプログラムは、上記の1から5までの5つのガイドラインについての違反を含む。

5.2.3 実験結果

まず、被験者の組分けを行なうためのプログラミングに関する能力テストを行なった。ここでは、簡単なプログラムの作成問題を出題し、プログラムの考え方、完成度により優、良、可の3段階で評価を行なった。そして、この評価に基づき、なるべく2つの組が同等のレベルになるよう被験者86名をA組（43人）とB組（43人）に組分けした。A組とB組の評価結果を表5.1に示す。

そして、A組、B組それぞれ43人に対してプログラミングスタイルの検査実験を行ない、1) 悪形パターンの箇所が正しく検出されているか、2) 悪形パターンが正しく修正されているか、の2項目について10点満点で採点を行なった。

図5.1に、A組、B組それぞれの平均点を示す。

また、検出におけるA組、B組のそれぞれの標準偏差は、1.36, 2.52であった。

図5.2に、5つのガイドライン毎の検出率、および修正の正解率をA組、B組に分けて折れ線グラフで示す。

また、解答に要した時間の平均は、A組が15.3分、B組が17.5分であった。

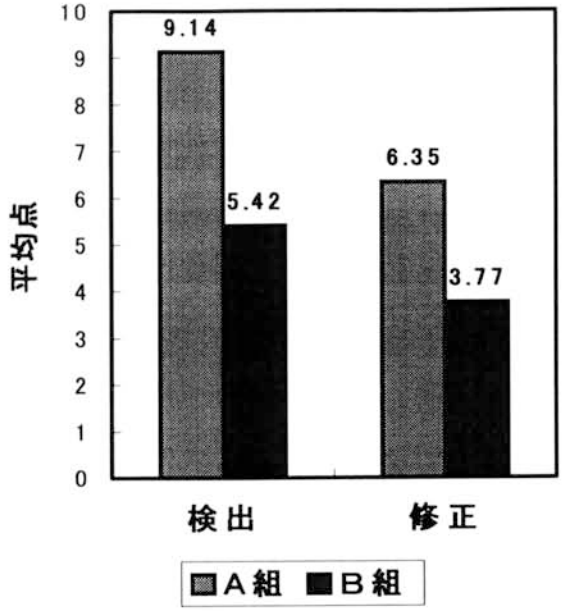

図5.1: 検出および修正の平均点

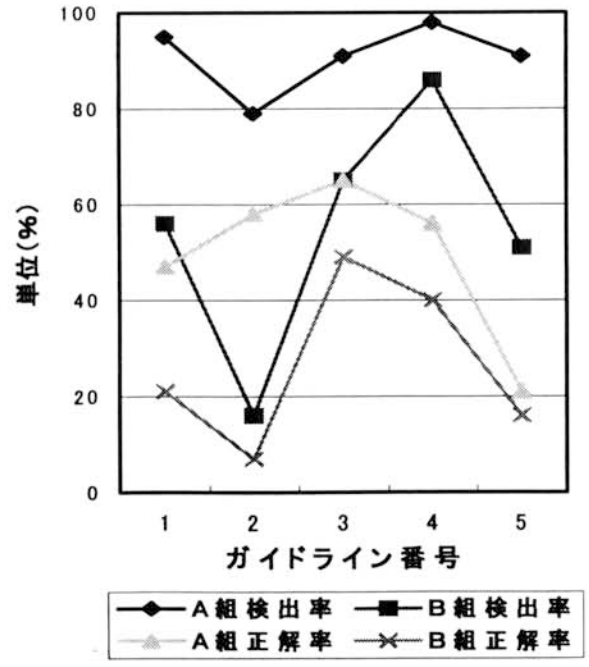

図5.2: 検出および修正の平均点

5.2.4 考察

実験の組分けに関して、実験前に行なったプログラミング能力テストの評価において、評価の優を10点、良を8点、可を6点とした二つの組の平均点は、A組が9.26点、B組が9.21点であり、ほぼ同等のレベルであるとみなせる。

図5.1より、検出、修正の両方において、システムの診断結果を利用したA組の方が、システムを利用しないB組より、明らかに平均点が高いことがわかる。ここで、平均値に関しての両側検定を有意水準0.01で行ない、検出、および、修正の両方でA組の方が有意性が高いことを確認した。以下にその結果を示す。

平均値の両側検定 [20]

A組とB組の結果に有意な差がみられたかどうか統計的決定を下すために、仮説検定を行なう。ここでは、平均値に関しての両側検定を有意水準0.01で行なった。ある仮説を採択すべきときに誤ってそれを棄却する確率をその検定の有意水準という。有意水準が0.01というのは、決定を誤る可能性が1%、すなわち正しい決定がなされるのが99%の信頼度であることを意味する。有意水準0.01の両側検定では、zの値(標本統計量Sの実現値)が −2.58から2.58の領域の外にあれば、その仮説を棄却する。

まず、検出について2つの組が、平均がそれぞれ μ_A、μ_B の2つの母集団から生じたものであると仮定する。次の2つの仮説を考える。

$H_0 : \mu_A = \mu_B$、差は単に偶然のものである。

$H_1 : \mu_A \neq \mu_B$、有意差がある。

仮説 H_0 のもとでは、2つの組は同じ母集団から生じたものである。平均の差と平均と標準偏差は、次のように与えられる。

$\mu_{\bar{A}-\bar{B}} = 0$

$\sigma_{\bar{A}-\bar{B}} = \sqrt{\frac{\sigma A^2}{n_A} + \frac{\sigma B^2}{n_B}} = \sqrt{\frac{1.36^2}{43} + \frac{2.52^2}{43}} = 0.436$

ここに、σ_A と σ_B の推定値として、標本標準偏差を用いた。そうすれば、

$Z = \frac{\bar{A}-\bar{B}}{\sigma_{\bar{A}-\bar{B}}} = \frac{9.14-5.42}{0.436} = 8.53$

有意水準0.01の両側検定では、Zの値が −2.58から2.58の領域の外にあれば、有意である。よって、水準0.01で2組の達成度の間には有意差があり、システムが有効に働いたという結論を下すことができる。同様に、修正における正解の平均点では、Z=4.62となり、こちらもA組の方が有意性が高い。修正に関してもシステムが有効であると

言える。

図5.2より、システムを利用したA組はガイドライン毎の差が少ないが、システムを利用していないB組はガイドライン毎の差が大きいことが言える。これより、検出しずらいガイドラインの場合（ここでは、2番）、特にシステムの利用が有効であると考えられる。

また、解答に要した時間の平均（A組は15.3分、B組は17.5分）より、システムを利用することにより、検出にかかる時間が短縮できたと考えることができる。

プログラミング演習において、これまでは教師が学生の作成したプログラムを個々にチェックし、面談等で個別に注意するしか方法がなかった。本プログラミングスタイルの診断システムを利用することにより、教師が介在しなくても、プログラミングスタイルの悪形パターンの検査が正しく、効率的に行なえることが実証できた。

5.3 評価実験2

5.3.1 実験目的

初心者プログラマの場合、プログラミングスタイルおよび文法等に関する知識が不足していることから、悪形パターンを認識できないことが多い。本システムは、悪形パターンの検査支援を行なうだけでなく、プログラマのプログラミングスタイルの検査能力を向上させることによって、ツールに頼らなくても悪形パターンを認識できるようにすること、つまり、プログラミングスタイルを習得させることも目的としている。

そこで、本実験ではシステムを利用することによる教育的効果に焦点をあて、本プログラミングスタイルの診断システムの利用を通して、初心者プログラマのプログラミングスタイルの検査能力を向上させることができるか調査することを目的とする。具体的には、システムを利用した人が、システムを利用する前に比べ、検査能力が向上しているか調査する。

5.3.2 実験方法

プログラミングスタイルの診断システムの有効性を評価するために被験者を2つの組に分け、片方の組のみシステムを利用させる。そして、その組におけるシステムを利用する前と後の検査結果の比較、および、初回と数回システムを利用した後でのシステムを利用しないでの検査結果の推移を、まったくシステムを利用しなかった被験者の組の推移と比較する。

まず、プログラミングスタイルの検査実験を始める前に、被験者を2組のグループ（A組, B組）に分ける。組分けをするための、簡単なプログラミングに関する能力テストを行なう。ここでは、マージプログラムに１０個のホールをあけたプログラムの穴埋め問題を出題する。そして、このテスト結果に基づき、なるべく同等のレベルになるよう組分けを行なう。

そして、配布されたソースプログラム中から指定されたプログラミングスタイル（6種類）の違反を検出し、修正してもらうという実験を行なう。

問題1、問題2、問題3の順番に、以下の要領で行なう。実験では、プログラムとプログラミングスタイルの記述された用紙を配布し、検出、修正は直接プログラム上に書き込むものとした。

(1) プログラム問題1について、まず、被験者全員（A組, B組）が、システムを利用しないで検査を行なう。その後、A組のみ再度プログラミングスタイルの診断結果を参照しながら検査を行なう。
(2) プログラム問題2について、A組は、プログラミングスタイルの診断結果を参照しながら検査を行ない、B組はシステムを利用しないで検査を行なう。
(3) プログラム問題3について、1.の手順と同様の実験を行なう。

実験対象者

被験者は、信州大学工学部情報工学科の4年生9名である。

対象プログラム

検査対象としたプログラムは、それぞれ以下のようなプログラムである。

問題1： 二つのファイルに記入されたデータのマージ

問題2： KWIC(Keyword in Context) による索引付け

問題3： クイックソート

各プログラムは100行から130行程度のプログラムである。それぞれのプログラムに、プログラミングスタイルの違反を5種類埋め込んだものを、検査対象のプログラムとした。違反の種類は、下記に挙げた対象プログラミングスタイルの1から5までのガイドラインである。

対象プログラミングスタイル (付録A.2参照)

検査するよう指示したプログラミングスタイルは、以下の6種類のガイドラインである。

(1) 定数に関するガイドライン
(2) 条件式の副作用に関するガイドライン
(3) インクリメント/デクリメント演算子の利用に関するガイドライン
(4) ファイル処理に関するガイドライン
(5) 条件節に関するガイドライン
(6) for文の利用に関するガイドライン

検査対象としたプログラムは、上記の1から5までの5つのガイドラインについての違反を含む。

5.3.3 実験結果

まず、被験者の組分けを行なうためのプログラミングに関する能力テストを行なった。ここでは、マージプログラムに１０個のホールをあけたプログラムの穴埋め問題を出題し、１０点満点で採点を行なった。そして、この評価に基づき、なるべく2つの組が同等のレベルになるよう被験者9名をA組（5人）とB組（4人）に組分けした。

A組とB組の採点結果を表5.2に示す。

次に、それぞれの問題（問題1、問題2、問題3）について、悪形パターンの箇所を正しく検出し、修正しているかについて10点満点で採点を行なった。

A組、B組の平均点をそれぞれ図5.3、図5.4に示す。また、システムを利用していない状況でのA組、B組の問題1と問題3の平均点の推移を図5.5に示す。

表 5.2: 評価実験2の組分け問題における結果

組	平均点（点）
A 組	9.0
B 組	8.2

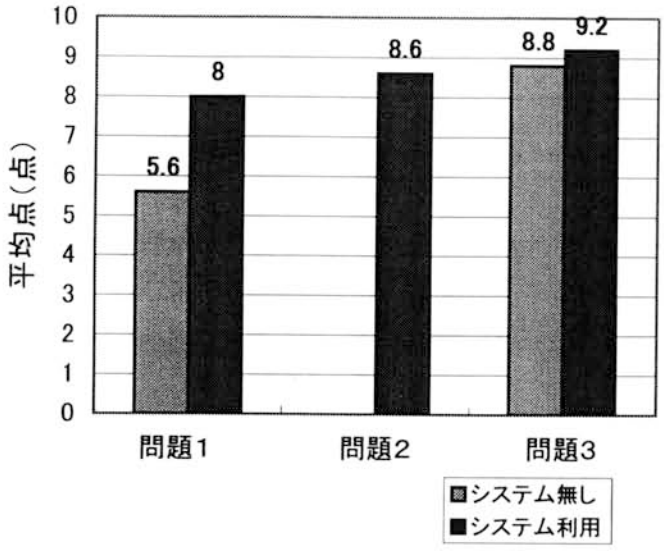

図 5.3: A 組の平均点

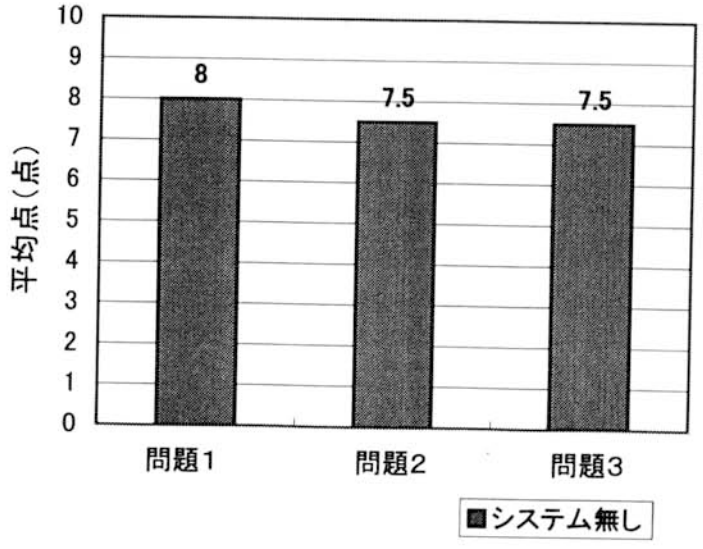

図 5.4: B 組の平均点

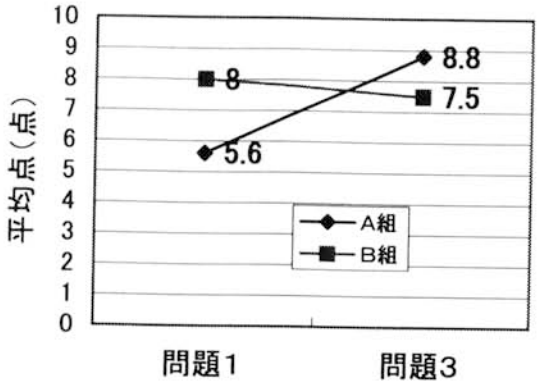

図5.5: システムを利用しない検査での平均点の推移

5.3.4 考察

実験の組分けに関して、プログラミングスタイルの検査実験前に行なったプログラムの穴埋め問題（10点満点）で、平均点はA組が9.0点、B組が8.2点であり、ほぼ同等のレベルであるとみなせる。

図5.3を見ると、A組の被験者はシステムを利用することにより平均点が上がっていることがわかる（特に、問題1）。また、図5.4と図5.4を比較してみると、問題2以降においてA組の方がB組より平均点が高くなっている。これより、本システムが初心者プログラマの悪形パターンの検出の支援に有効であると言える。

そして図5.5のプログラミングスタイルの診断システムを利用しない検査での平均点の推移を見ると、B組の平均点は問題1と問題3でほとんど変化が見られないが(-0.5点)、問題1、問題2においてシステムを利用したA組では、問題3の平均点の方が問題1の平均点よりだいぶ高くなっている(+3.2点)。B組の結果より、各問題プログラムにおける難易度の差はほとんどないと言える。それにもかかわらず、A組が問題3において、システムを利用しなくとも、以前より高い平均点が得られたのは、A組の被験者は、問題1、問題2でシステムを利用することにより、プログラミングスタイルの悪形パターンの検査能力が向上したと考えることができる。

5.4 おわりに

本章では、初心者プログラマを対象に行なった、プログラミングスタイルの検査に関する二つの評価実験について述べた。

これより、本プログラミングスタイルの診断システムが、初心者プログラマのプログラミングスタイルに関する悪形パターンの検出、および修正の支援に有効に働くことを示した。さらに、本システムの利用が、初心者プログラマのプログラミングスタイルの悪形パターンの検査能力の向上に役立つことが確認できた。

第6章　結論

6.1　本研究で得られた成果

本論文では、C言語のソースプログラム中からプログラミングスタイルに違反する箇所を検出するプログラミングスタイルの診断システムについて述べた。本システムは、プログラムパターンと呼ぶ記述形式で表現された個々のプログラミングスタイルについて、プログラムのソースコード探索を行ない、マッチしたプログラム断片がある場合には診断メッセージを表示する。システム利用者は、その診断メッセージを参考にプログラムの修正を行なうというものである。

第2章においては、プログラミングスタイルの個々のガイドラインを表現するためのプログラムパターンの記述形式について述べた。提案したプログラムパターンの記述形式では、テンプレートを記述する本体部と制約条件を記述する制約条件部に分けて表現する記法を用いた。この分離により、

(1) 文脈に依存したパターンであっても、文脈条件は制約条件として述語的に記述できる。これにより、テンプレートの記述は、文脈に依存しない構文パターンとして記述でき、そのまま有限状態オートマトンによるソースコード探索により検出できる。制約条件は、事後条件として個々に実現すればよく、アルゴリズムもシンプルになり、記述自体の再利用性も増す。
(2) プログラムのフロー情報などを制約条件として利用する場合でも、システムのアルゴリズムをほとんど変更することなく、フロー解析ツールおよびその情報を利用する関数の導入だけで済む。

という特徴を持つ。また、ベースパターンをテンプレートとして記述する複合パターンにより、離れた場所に分散して存在するパターンも記述できるという特色がある。さらに、ある命令の存在ではなく、存在していないことを問題としているパターンも、特別な制約 not_found と複合パターンの組合せにより、一部については記述可能である。

第3章においては、2章で提案したプログラムパターンをソースプログラム中から認識する手法について述べた。

提案したプログラミングスタイルの診断システムは、以下の特徴を持つ。

(1) 本システムは、個々のガイドライン（各悪形パターン）を一つのプログラムパターンとしてデータ化し、これを用いてソースコードの探索を行なう。このようなオープンな方式を採用することにより、プログラミングスタイルの検査項目の追加が容易に行なえ、拡張性に優れている。また、検査項目を入れ換えることにより、種々の要因のもとでの評価が可能である（例えば、初心者プログラム用の検査やあるプロジェクト用の検査等）。
(2) 本システムでは、構造的なスタイルに関するパターンや文脈に依存したパターン、また、二つ以上の関数など離れた位置に分散するなパターンなど、単純なテキスト上のマッチングでは検索が難しいパターンの認識も可能である。
(3) 制約関数が不十分な場合は、新たな関数を追加することになるが、基本的には対象となるプログラム断片（の解析木）の走査であるので、関数自体の実現は容易であり、またシステムへの追加も関数のみの追加であるので容易に実現できる。

第4章においては、実際のプログラミングスタイルをプログラムパターンで記述する実験、および、それらのプログラムパターンを実プログラム中から認識する実験について述べた。ここでは、文献等に載っている標準的なスタイルガイドラインのいくつかをプログラムパターンとしてデータ化し、テキストユーティリティのプログラム群を対象にスタイルガイドラインの検査を行うことにより、標準的なガイドラインが正しく検出可能であることを確認した。これにより、本システムが、プログラミングスタイルの種々の悪形パターンの検出に有効な手法であるといえる。

第5章においては、初心者プログラマを対象に行なった二つの評価実験について述べた。初心者プログラマ86名を対象として行なった評価実験1を通して、本システムが初心者プログラマのプログラミングスタイルに関する悪形パターンの検出、および修正の支援に有効に働くことを示した。また、本システムの利用がプログラミングスタイルの検査の際の時間短縮に貢献することが確認できた。さらに、9名を対象として行なった評価実験2を通して、本システムの利用が、初心者プログラマのプログラミ

ングスタイルの悪形パターンの検査能力の向上に役立つことが確認できた。

プログラミング演習において、これまでは教師が学生の作成したプログラムを個々にチェックし、面談等で個別に注意するしか方法がなかったが、本システムを利用することにより、教師が介在しなくても、ある程度自分でプログラミングスタイルの検査が可能となる。また、プログラミングスタイルの検査を習慣づけることで、プログラミングスタイルに関する意識が向上し、良形度の高いプログラムの作成が身に付くと期待できる。

6.2 今後の課題・展望

今後の課題および展望としては、以下のようなことが挙げられる。

- プログラムパターンの表現能力に関する問題としては、ORの概念、および、省略可能の表現を実現していないことが挙げられる。現在は、複数のプログラムパターンを定義することで対応しているが、これらの表現が利用できれば、同じようなプログラミングスタイルに関する診断を行なうのに、より少ないプログラムパターンの照合で済むことになり、システムを利用する側においても使いやすいシステムとなる。

  具体的には、第4章において示した、プログラミング演習において利用しているプログラミングスタイルに対して定義したプログラムパターンの例では (表4.3 参照)、for文において条件節の一部、または全てを省略している悪形を検索するプログラムパターンを7つのプログラムパターン (abb_for1_pattern ～ abb_for7_pattern) で定義しているが、ORの概念が利用できれば、一つのプログラムパターンとして定義できる。また、if文の条件式が関数呼び出し (引数なし、あり) であるパターン (fun_if_condition1_pattern, fun_if_condition2_pattern) も、省略の概念が利用できれば、一つのプログラムパターンとして定義できる。

- 第2章のプログラムパターンの記述例で示したような、あるプログラムパターンが存在しないようなことを問題としているプログラミングスタイルに対する記述および認識が不十分である。

これに対応するためには、制約条件の検査方法の改良が考えられる。現在の複合パターンにおける制約条件検査のアルゴリズムの概略は、以下の通りである。

[現在の条件の検査方法]

複合パターンのコンポーネントであるベースパターンを P1, P2,...Pn 、
それぞれににマッチしたプログラム断片の集合を S1,S2,...Sn、
集合に含まれるそれぞれの要素を q1, q2, ...q3 とする。

for (some element (q1, q2, ... qn) in (S1 × S2 ×... × Sn))
if ((q1, q2, ... qn) に対して、全ての条件が TRUE)
then (q1, q2, ... qn) を複合パターン matchd list へ追加

つまり、各コンポーネント（ベースパターン）に照合したプログラム断片全ての組合せ（直積）について、個々に条件の検査を行ない、全ての条件を満たすものを matched list としている。

しかしがら、主、従の観点を持つようなパターン（例えば、「パターン A が存在して、かつ、A に対してある条件を満たす B が存在しないパターン」）に対応するためには、コンポーネントパターンに照合したプログラム断片全ての組合せを同等に扱うのではなく、一つ（または、複数）のコンポーネントパターンに特化した形で条件の検査を行なう必要がある。そこで、条件の検査において以下のような新たな方法を考案した。

```
[新たな条件の検査方法]

for (each element q1 in S1)
    {for (all element (q2, ... qn)    in (S2 ×... × Sn))
        if ((q1, q2, ... qn) に対して、
            全ての条件が TRUE になるものが 1 つも存在しな
い)
            then   q1 を複合パターン matchd list へ追加
    }
```

この方法では、コンポーネントパターン P1 にマッチしたプログラム断片各々に対して、制約条件を満たす組合せが一つも存在しない q1 を検出する。

この検査方法を実現することにより、存在していないことを問題としているパターンにおいても、パターン変数等についての検査が行なえ、より正確な診断が可能となる。

- これまでは、文献等に載っているプログラミングスタイルを中心にプログラムパターンとしてデータ化し、システムのテストを行なってきた。今後は、教育効果をさらに高めるために、実際に初心者プログラムの作成したプログラムを対象に、どのような悪形パターンが多く使われているかを分析し、初心者プログラマの教育において有効となるスタイルガイドラインのデータ（プログラムパターンデータベース）を充実させていくことが重要である。また、プログラムパターンを分析することにより、プログラムパターンの分類・階層化が行なえれば、現在より高度な診断結果を表示可能なシステムになると思われる。

- 本システムは教育目的のシステムとしての利用だけでなく、実際のプログラム開発の場でのガイドライン検査としての利用も想定している。このような複数の人が開発に携わる場合は、ガイドラインを個人的なものとして利用するのではなく、共通の知識として共有することが重要となり、企業毎に独自のコーディング規約をもつことも多い。システムで、ガイドラインに違反している箇所の検出、診断

を行なうことができれば、プログラムの品質向上や保守等の効率アップに役立つと考えられる。このためにはまず、実際のサンプルを対象にコーディング規約のデータ化、およびテストを行なう必要がる。

- 構築したプログラムパターン認識システムは、様々なプログラムパターンにマッチするプログラム断片を検索することが可能である。表現する知識を入れ換えることで、プログラミングスタイルに関する検査のみではなく、何らかのプログラム認識を目的としたシステムにも応用可能と思われる。

謝辞

本研究遂行のための利便の提供、研究指導、得られた成果の発表に際しての指示など、全般に渡ってご指導、ご鞭撻を賜わった信州大学工学部情報工学科　海尻 賢二 教授に心から感謝の意を表します。

信州大学工学部情報工学科　中野 康明 教授、岡本 正行 教授、師玉 康成 教授には、本論文について貴重なご討論とご助言を戴きました。深く感謝致します。

本研究は、筆者が信州大学工学部情報工学科在職中にプログラムパターンを用いたプログラミングスタイルの診断システムに関して行なった研究をまとめたものである。本研究を進める上で日頃からご協力、ご議論していただいた信州大学工学部情報工学科 海谷 治彦 助教授、および、海尻研究室の学生各位に厚く感謝致します。

最後に、本研究を遂行するにあたり、無邪気な笑顔でたくさんのエネルギーをくれた子供、そして、多大な理解と協力をしてくれた夫に心から感謝します。

参考文献

[1] Edward J. Thomas and Paul W. Oman: A Bibliography of Programming Style, SIGPLAN Notices, Vol. 25, No. 2 (1990)

[2] David Atraker: C style standards and Guideline, Prentice-Hall (1992)

[3] C Programming Style Guide, URL://www.ntr.net/ bud/style-guide.html (1994)

[4] Brian W.Kernignhan and P.J.Plauger (木村 訳) : C The Elemetns of Programming Style (プログラム書法) 共立出版 (1982)

[5] Tom Schorsch: CAP: An automated self-assessment tool to check Pascal programs for syntax, logic and style errors, SIGCSE'95 (1995)

[6] Al Lake and Curtis Cook: STYLE -An automated program style analyzer for Pascal-, SIGCSE Bulletin, Vol. 22, No. 3 (1990)

[7] David Jacson and Michelle Usher: Grading Student Programs using ASSYST SIGCSE '97, pp. 335– 339 (1997)

[8] Jacson D.: Using Software Tools to Automate the Assesment of Student Programs, Computers and Education, Vol.17, No.2, pp. 133-143 (1991)

[9] McCabe T.A.: A complexity measure, IEEE Trans, Softw. Eng., Vol.SE-2, No.4, pp.308–320 (1976)

[10] Woodward M.R., Hedley D. and Hennell M.A.: Experience with path analysis and testing of programs, IEEE Trans. Softw. Eng., Vol.SE-6, No. 3, pp.278–286 (1980)

[11] R.E. Berry and B.A.E. Meekings: A style analysis of C programs, Communications of the ACM, Vol. 28, No. 1 (1985)

[12] 小田 まり子, 掛下 哲郎: 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方法 , 情報処理学会論文誌, Vol.35, No. 11 (1994)

[13] Tetsuro Kakeshita, Mariko Oda and Yoshihiro Imamura: Fall-in C : A Software Tool for Pitfall Detection in C Programs, Proceedings of APSEC'94, pp. 256–265 (1994)

[14] W. Kozaczynsky, J. Ning and A. Engberts : Program Concepts Recognition and Transformation, IEEE Trans. Software Eng., Vol.18, No.12 (1992)

[15] S. Paul and A. Prakash : A Framework for source code search using programming patterns , IEEE Trans. Software Eng., Vol.20, No.6 (1994)

[16] S. Paul et.al. A Query Algebra for Program Databases , IEEE Trans. Software Eng., Vol.22, No.3 (1996)

[17] Yih-Farn Chen, et al.: The C Information Abstraction System , IEEE Trans. Software Eng., Vol.16, No.3 (1990)

[18] Koenig A. (中村 明 訳): Cプログラミングの落し穴 , (株) トッパン (1990)

[19] E.Gammma et al.: Design Patterns , ADDISON-WESLEY (1995)

[20] スピーゲル著：マグロウヒル大学演習シリーズ 統計, マグロウヒル好学社,(1981).

[21] Charles Rich and Richard C. Waters: The Programmer's Apprentice, ACM Press (1990)

[22] A.Adam and J.Laurent: LAURA : A System to Debug Student Programs, Artificial Intelligence 15 (1980)

[23] Johnson.W.L: Understanding and Debugging Novice Programs, Artificial Intelligence 45, pp. 51–97 (1990)

[24] W.Lewis Johnson: PROUST : Knowlede-Based Program Understanding, ICSE (1984)

[25] 服部徳秀, 石井直宏：ソースコードのバリエーション除去システム, 電子情報通信学会論文誌 D-I, Vol.J80-D-I, No.1, pp.50-59 (1997)

[26] David R. Harris, et al.: Recognizers for extracting Architectural Features form Source Code , 2nd Working Conf. on R.E. (1995)

[27] P. Tonella, R. Fiutem and G.Antonil: Augmenting Pattern-Based Architectural Recovery with Flow Analysis: Mosaic - A Case Study, Proceedings of WCRE'96, pp. 198 207 (1996)

[28] 藤原 博文： Cプログラミング診断室, 技術評論社 (1993)

[29] Steve Oualline: Practical C Programming, O'Reilly and Associates, Inc. (1991)

[30] 小川優介, 西田雅昭： プログラミングお作法, 技術評論社 (1989)

[31] B.W. カーニハン, D.M. リッチー（石田晴久訳） プログラミング言語C 第2版, 共立出版 (1989)

[32] Kenji Kaijiri: Support of plan library construction, JCKBSE'94 : Japan-CIS Symposium on Knowledge-Based Software Engineering (1994)

[33] Rika Sekimoto and Kenji Kaijiri: Plan Representation and Its Recognition Approach for Program Recognition, second joint conference on knowledge-based software engineering (JCKBSE'96), pp.198 – 201 (1996)

[34] Rika Sekimoto and Kenji Kaijiri: A detection of ill-formed patterns about programming style, Knowledge-Based Software Egineering(JCKBSE'98),IOS Press, pp. 165–168 (1998)

[35] kenji Kaijiri and Rika Sekimoto Program diagnosis system on World Wide Web, Proc. of ICCE'99(7th International Conference on Computers in Education), (1999)

[36] 関本理佳, 海尻賢二: プログラミングスタイルの診断システムの構築, 教育システム情報学会誌, vol.17, No.1, pp.21–29 (2000)

[37] Rika Sekimoto and Kenji Kaijiri: A Diagnosis System of Programming Styles Using Program Patterns, IEICE TRANS. INF.&SYST., Vol.E83–D, No.4, pp.722–728 (2000)

[38] 関本理佳, 中島歩, 上野晴樹: 意図に基づくプログラム理解の方法, ソフトウェアシンポジウム '92 論文集, E-19, pp. 19 – 26 (1992)

[39] 関本理佳, 海尻賢二: プログラム認識による仕様記述の生成について, 電子情報通信学会 技術研究報告, Vol.93, No.146, pp. 43–50 (1993)

[40] 関本理佳他: プログラム認識による仕様記述の生成, 電気関係学会東海支部連合大会講演論文集 (H5) (1993)

[41] 関本理佳, 海尻賢二: プラン認識を利用したプログラミングスタイルの診断, 電子情報通信学会技術研究報告, Vol.97, No.175, pp.9–16 (1997)

[42] 関本理佳, 海尻賢二: プラン認識に基づくCプログラムの保守ツール, 電子情報通信学会技術研究報告, Vol.97 , No.502 , pp.1–8 (1998)

[43] 関本理佳, 海尻賢二: プログラムパターンを利用したプログラミングスタイルの診断システム, 電子情報通信学会技術研究報告, Vol.98, No.634, pp.37-44 (1999)

[44] 関本理佳, 海尻賢二: プログラミングスタイルの診断システムの構築, 教育システム情報学会研究報告, vol.98, No.5, pp.18–23 (1999)

[45] 海尻賢二, 関本理佳: WWW を利用した教育法に関する一考察 –現状と新方式の提案–, 教育システム情報学会研究報告, vol.98, No.5, pp.29–34 (1999)

研究業績

学術論文

[1] Rika Sekimoto and Kenji Kaijiri: " Diagnosis System of Programming Styles Using Program Patterns", IEICE Transactions on Information and Systems, Vol.E83-D, No.4, pp.722–728 (2000)

[2] 関本 理佳, 海尻 賢二: "プログラミングスタイルの診断システムの構築", 教育システム情報学会誌, vol.17, No.1 (春号), pp.21–29 (2000)

国際会議

[3] Rika Sekimoto and Kenji Kaijiri: "Plan Representation and Its Recognition Approach for Program Recognition", Second joint conference on knowledge-based software engineering (JCKBSE'96), pp.198–201 (1996)

[4] Rika Sekimoto and Kenji Kaijiri: "A detection of ill-formed patterns about programming style", Knowledge-Based Software Egineering(JCKBSE'98), pp.165–168 (1998)

研究会

[5] 関本 理佳, 海尻 賢二: “プラン認識を利用したプログラミングスタイルの診断 ”, 電子情報通信学会技術研究報告, Vol.97, No.175, pp.9–16 (1997)

[6] 関本 理佳, 海尻 賢二: “プラン認識に基づくCプログラムの保守ツール ”, 電子情報通信学会技術研究報告, Vol.97 , No.502 , pp.1–8 (1998)

[7] 関本 理佳, 海尻 賢二: “プログラムパターンを利用したプログラミングスタイルの診断システム ”, 電子情報通信学会技術研究報告, Vol.98, No.634, pp.37–44 (1999)

[8] 関本 理佳, 海尻 賢二: “ プログラミングスタイルの診断システムの構築”, 教育システム情報学会研究報告, vol.98, No.5, pp.18–23 (1999)

その他の学術論文

[9] 関本 理佳, 海尻 賢二, 山形 昌也: “ネットワークを利用したレポート受付・評価支援システムの実現 ”, 教育システム情報学会誌, vol.14, No.5, pp.217–222 (1998)

[10] Kenji Kaijiri and Rika Sekimoto: “Program Diagnosis System on World Wide Web ”, Proceedings of the 7th International Conference on Computers in Education (ICCE'99), pp.729–735 (1999)

付 録A　初心者プログラマによる評価実験で用いた資料

A.1　評価実験１で用いた診断プログラム

評価実験１で用いた診断プログラムを以下に示す。

```
/*      ■問題■
* 昇順に並んだ２組のデータ列を、昇順に並んだ１組のデータ列にマージ（併合）
* するプログラムを作成せよ。
* ２組のデータは、それぞれ別々のファイルに記入されているものとする。
*/

 1  #include<stdio.h>
 2  #define MAX 999                    /* データの値（入力値）の上限 */
 3
 4  /* 指定されたファイルからデータを読み込み、data[] へ代入する関数 */
 5  int             input_fun(int data[]);
 6
 7  /* dataA[] と dataB[] のマージを行ない、dataC[] へ代入する関数 */
 8  void  merge(int dataA[], int dataB[], int dataC[], int numA, int numB);
 9
10  /* 配列 data[] の表示を行なう関数 */
11  void  output_fun(int data[], int num);
12
13  void
14  main(void)
15  {
16      int   dataA[100], dataB[100];   /* 入力データ用配列 */
17      int   numA, numB;               /* 入力されたデータの数 */
18      int   dataC[2 * 100];           /* マージ用配列 */
19
```

```
20      numA = input_fun(dataA);/* データ列Aの入力 */
21      output_fun(dataA, numA);/* データ列Aの表示 */
22
23      numB = input_fun(dataB);/* データ列Bの入力 */
24      output_fun(dataB, numB);/* データ列Bの表示 */
25
26      merge(dataA, dataB, dataC, numA, numB); /* データ列AとBのマージ */
27      printf("[マージされたデータ列] \n");
28      output_fun(dataC, numA + numB); /* マージされたデータ列の表示 */
29  }
30
31
32  int
33  input_fun(int data[])
34  {
35          char    FileName[16];   /* 入力ファイル名 */
36          FILE    *input;  /* 入力ファイルポインタ */
37          int     i, temp;
38
39          printf("入力ファイル名 ==> ");  /* 入力ファイルの指定 */
40          scanf("%s", FileName);
41          if ((input = fopen(FileName, "r")) == NULL) {
42              printf("ファイルが見つかりません  --- %s\n", FileName);
43              exit(-1);
44          }
45          /* ファイルからの読み込み */
46          i = 0;
47          while (!feof(input)) {
48             fscanf(input, "%d", &temp);
49             if (temp >= 0 && temp <= MAX)
50                 data[i] = temp;
51             else {
52                printf("\n不適切なデータがあるので処理を中止します.\n");
53                exit(-2);
54             }
55             i++;
56          }
57          return (i);         /* 入力されたデータの数を返す */
```

```
58  }
59
60
61  void
62  merge(int dataA[], int dataB[], int dataC[], int numA, int numB)
63  {
64          int       indexA, indexB, indexC;
65
66          indexA = indexB = indexC = 0;    /* 各添字の初期化 */
67
68   /* dataA[],dataB[]が終りでない間,dataA,dataBの小さい方をdataCに代入 */
69          while (indexA < numA && indexB < numB) {
70              if (dataA[indexA] <= dataB[indexB]) {
71                      dataC[indexC] = dataA[indexA];
72                      indexC++;
73                      indexA++;
74               } else {
75                      dataC[indexC] = dataB[indexB];
76                      indexC++;
77                      indexB++;
78             }
79          }
80
81         while (indexA<numA)/* dataA[]のデータを終末までdataCにコピー */
82              dataC[indexC++] = dataA[indexA++];
83
84         while (indexB<numB)/* dataB[]のデータを終末までdataCにコピー */
85              dataC[indexC++] = dataB[indexB++];
86  }
87
88
89  void
90  output_fun(int data[], int num)
91  {
92          int         i;
93
94          for (i = 0; i < num; i++)  /* data[0] ～ data[num-1] を表示 */
95              printf("%d  ", data[i]);
```

```
96          printf("\n");
97  }
```

A.2　評価実験1，2で対象としたプログラミングスタイル

評価実験1，2で対象としたプログラミングスタイルを以下に示す。

(1). 定数に関するガイドライン

意味を持つ定数は、マクロとして定義して利用するのが好ましい。

(悪い例)	(良い例)
	`#define SIZE  100  /* 表の最大の大きさ */`
	`. . .`
`char table[100];`	`char table[SIZE];`

(2). 条件式の副作用に関するガイドライン

繰り返し文 (for 文, while 文) や選択文 (if 文) の条件式に代入文を含めるなど、副作用のある式は書かない方が良い。

ここで、「副作用」とは主たる操作のついでに行なわれてしまう、副次的な操作のことである。

(3). インクリメント/デクリメント演算子の利用に関するガイドライン

インクリメント演算子 (++)、デクリメント演算子 (--) は、それ自身を単独の文で使うようにするのが好ましい。

特に、代入演算子は評価順序に関して何の保証もされていないので、代入演算子とインクリメント／デクリメント演算子を一緒に使うべきではない。

(4). ファイル処理に関するガイドライン

ファイル処理において、ファイルは使い終ったらクローズするというの基本である。プログラムが終了する時にオープンされたままのファイルは自動的にクローズされるが、明示的にクローズするのが好ましい。

(5). 条件節に関するガイドライン

条件式は原則的に条件を明示する。

C 言語では特に論理型を用意していない。そのため、条件式がゼロであるかどうかで条件式の真偽を判定しているが、これは間違い易いので条件を明示するのが好ましい。

```
            int flag;
（悪い例）  if(flag) { ... }
（良い例）  if(flag != 0) { . . .}
```

(6). for 文の利用に関するガイドライン

for 文のカッコの中の各式は、省略可能であるが、解釈の間違いを防ぐために全て明記するのが好ましい。

A.3 プログラミングスタイルの診断結果

以下に、評価実験 1 の出題プログラムに対し、指定された 6 項目のスタイルガイドラインの検査を行なったプログラミングスタイル診断システムの結果を示す。診断結果は、それぞれ以下の内容を含む。

(1) 検索したガイドラインの名前
(2) 検出した違反の数
(3) 診断メッセージ
(4) 検出した違反の場所（プログラム中の行数）

▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼

検索 [1]: 定数に関するガイドライン

マッチング箇所が (3) つ見つかりました。

ある大きさ以上の同一定数が複数存在します。
有意な定数は、マクロとして定義して利用しましょう。

複合プランにおける各プラン要素とのマッチングリスト

```
------( 1) 番目の照合リスト------------------------------
*(1) th plan -------   16 行目
 100
*(2) th plan -------   16 行目
 100

------( 2) 番目の照合リスト------------------------------
*(1) th plan -------   16 行目
 100
*(2) th plan -------   18 行目
 100

------( 3) 番目の照合リスト------------------------------
*(1) th plan -------   16 行目
 100
*(2) th plan -------   18 行目
 100
```

▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼

検索 [2]: 条件式の副作用に関するガイドライン

マッチング箇所が (1) つ見つかりました。

条件式に代入文が含まれます。
これは、副作用を含むので好ましくありません。
条件式を直しましょう！

複合プランにおける各プラン要素とのマッチングリスト

```
------( 1) 番目の照合リスト------------------------------
*(1) th plan -------  41 行目
 input = fopen ( FileName , "r" )
```

▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼

検索 [3]: インクリメント/デクリメント演算子の利用に関するガイドライン

マッチング箇所が (2) つ見つかりました。

配列の添字に increment か decrement の演算子が含まれます。
これは、副作用を含むので好ましくありません。
プログラムをシンプルに保つために、 ++と-- はつねに、
それ自身を単独の文で使うように心がけましょう。

マッチングしたのは、以下の箇所です。

(1) 82 行目
(2) 85 行目

▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼

検索 [4]: ファイル処理に関するガイドライン

マッチング箇所が (1) つ見つかりました。

オープンされたままのファイルがあります。
ファイル処理において、「ファイルは読み書きを行なう前にオープンし、
使い終ったらクローズする」というの基本です。
プログラムが終了する時にオープンされたままのファイルは、自動的に
クローズされますが、明示的にクローズするように心がけましょう。

ファイルのクローズ

```
filelose(ファイルポインタ);
```

複合プランにおける各プラン要素とのマッチングリスト

```
------( 1) 番目の照合リスト------------------------------
*(1) th plan -------  41 行目
 input = fopen ( FileName , "r" )  ;
*(2) th plan -------===unmatched==
```

▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼

検索 [5]: 条件節に関するガイドライン

マッチング箇所が (1) つ見つかりました。

while 文の条件節が、関数呼びだし（または、識別子）のみとなってます。
プログラムをわかりやすくするために、条件の値を明示しましょう。
マッチングしたのは、以下の箇所です。

(1)　47 行目 ～ 56 行目

▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼▽▼

検索 [6]: for 文の利用に関するガイドライン

--プログラム中に、このガイドラインに違反する箇所は見つかりませんでした。--